DRAM-less: Hardware Acceleration of Data Processing with New Memory

Jie Zhang[1], Gyuyoung Park[1], David Donofrio[2], John Shalf[2], Myoungsoo Jung[1]
Computer Architecture and Memory Systems Laboratory,
Korea Advanced Institute of Science and Technology (KAIST)[1], Lawrence Berkeley National Laboratory[2]
http://camelab.org

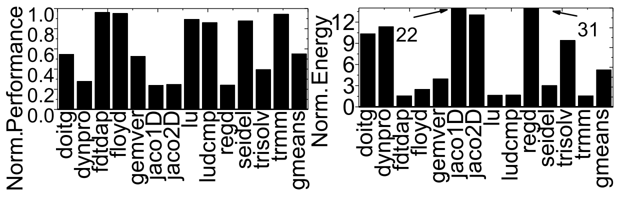
***Abstract*—General purpose hardware accelerators become a major data processing resource in many computing domains. However, the processing capability of hardware accelerators is often limited by costly software overheads and memory copies to support compulsory data movement between different processors and solid-state drives. This in turn wastes a significant amount of energy in modern accelerated systems. In this work, we propose, `DRAM-less`, a hardware automation approach that integrates many state-of-the-art phase change memory (PRAM) modules into its data processing fabric to dramatically reduce the unnecessary data copies with a minimum of software modifications. We implement a new memory controller that plugs a real 3x nm multi-partition PRAM to 28nm FPGA logic cells and interoperate its design into a PCIe accelerator emulation platform. The evaluation results reveal that `DRAM-less` achieves, on average, 47% better performance than advanced acceleration approaches that use a peer-to-peer DMA (between a discrete accelerator and an SSD), while consuming only 19% of the total energy of such accelerated systems.**

***Keywords*-PRAM; near-data processing; accelerator; FPGA controller; data movement;**

I. Introduction

Multi-core based accelerators such as graphics processing units (GPUs) or many integrated cores (MICs) have in recent years improved their computational power by employing hundreds to thousands of cores [1], [2]. The integration of hardware accelerators with fully programmable and massively parallel coprocessors accelerates a wide spectrum of scientific computations and data analytics [4], [5], [6]. Since modern hardware accelerators typically offer one order of magnitude speed-up, compared to CPU-only computing, accelerators have become a major data processing resource in many computing domains, ranging from high-performance computing to embedded systems [7], [8].

Despite notable advances in such massively parallel computing, the processing capability of modern hardware accelerators suffer from serious performance degradation in diverse data-intensive and streaming applications due to inefficient data transfers [9], [10], [11]. Specifically, in most of the accelerated systems, it is inevitable to move and copy in/output data across multiple physical and logical interface boundaries that exist between the accelerators and storage [12], [13], [14]. Such a long datapath from the SSDs to the accelerators unfortunately consumes a significant amount of energy, irrespective of their computing efficiency.



(a) Performance. (b) Energy consumption.

Figure 1: Performance degradation in near-data processing due to data movement overheads.

To be precise, the data shown in Figure 1 is from an empirical evaluation that we performed on a real accelerated system. The system employs a high-performance multi-core based accelerator [15] and an advanced solid state drive (SSD) [16] through two different PCIe slots [17]. In this evaluation, we execute representative data-intensive workloads [18] on the accelerated system and compares the results to those of an idealized environment that has enough memory space to accommodate all data within the accelerator. We normalize application-level performance and energy consumption of the accelerated system with those of the ideal system. As shown in the figure, *the performance of such data processing on the accelerated system degrades as much as 74%, while consuming energy more than the ideal system by 9 times, on average.* This is because the SSD access requests, generated by computation kernels operating in the target accelerator, introduce many software interventions at the host side. Unfortunately, the SSD accesses consume most CPU cycles to move target data among multiple PCIe physical interconnections and software interface barriers.

To address these challenges, we propose `DRAM-less`, a hardware automated near-data processing approach that removes the excessive software interventions and repetitive memory copies imposed by the data transfers across the multiple logical and physical interface boundaries. Our `DRAM-less` integrates a new type of 3x nm (i.e., 30 nm $\sim$ 39 nm) multi-partition phase change random access memory (PRAM) into a discrete accelerator's network fabric with multiple low-power processing units.

A challenge to this tight integration is that the PRAM memory's operations are completely different from DRAM operations, which limits our ability to leverage the current DRAM control logic for near-data processing. In addition,

Authorized licensed use limited to: Xiamen University. Downloaded on July 22,2021 at 07:14:30 UTC from IEEE Xplore. Restrictions apply.

the new memory cannot be managed by a conventional storage firmware that modern SSDs employ because the firmware execution latency becomes a performance bottleneck of PRAM management (cf. Section III-B). Thus, we fully automate the PRAM subsystem in hardware and integrate it into the data processing accelerator. In this design, computation kernels can be executed from the on-chip cache of each low-power core, and process massive data by directly accessing the PRAMs using a set of traditional "load/store" instructions. We implement all necessary hardware components of the automated memory subsystem in a 28 nm technology FPGA with a real PRAM device.

To explore the full design space of our accelerator, we also implement the proposed `DRAM-less` on a PCIe-based hardware platform with commercially available multi-core processors, similar to TMS320C6 series [19], and internally emulate a wide spectrum of storage devices [20], [21], [22], [23]. This emulation platform can employ various non-volatile memory (NVM) technologies as its main memory and execute diverse near-data processing applications [18]. The evaluation results reveal that, our `DRAM-less` achieves, on average, 47% better performance than the advanced accelerator approaches that use a peer-to-peer zero-overhead DMA (between an SSD and accelerator) [13], [14], while consuming only 19% of the total energy of such advanced accelerations. The main contributions can be summarized as follows:

• **DRAM-less design for efficient data processing.** We propose a hardware-automated memory subsystem that employs a crosspoint-based PRAM array for a data processing accelerator to replace a capacity-limited DRAM. The large capacity of our PRAM subsystem can persistently store extensive data, which can also eliminate the overheads of accessing the host-side in/output data from an SSD. For high throughput, `DRAM-less` tightly connects the subsystem to all low-power multi-core processors of the accelerator through a conventional memory interface [24]. Specifically, our memory subsystem loads data from the underlying PRAM modules and directly sends the data to the L2 cache of each processing element (or store them to the PRAMs), which are compatible with traditional memory instructions.

• **Hardware automation for a new memory.** We automate the PRAM subsystem by implementing our own PRAM 400 MHz physical layer and controller on a Virtex-7 FPGA platform [25] from the ground up with state-of-the-art 3x *nm* multi-partition PRAM engineering samples, which remove all firmware modules to access data from the I/O path of our accelerator platform. For efficient near-data processing, we also propose two PRAM-aware memory schedulers: i) *an interleaving method* that considers multiple row buffers and array partitions within the PRAM and ii) *a selective erasing technique* to further reduce the overheads of the PRAM overwrites. While the new memory interleaving technique can hide the memory access latency behind the corresponding data transfer time by 40%, the proposed selective erasing approach shortens the overall PRAM write latency by 44%, compared to a non-optimized PRAM subsystem.

• **New near data processing models.** An accelerator and SSD employed by a conventional computing architecture requires the support of many system software modules for their I/O management. This introduces significant communication overheads to transfer data between devices. In addition, traditional kernel scheduling methods need assistance from the host processor to coordinate the kernel executions in the target accelerator [26], which further increase software overheads. To address this, we propose new programming and kernel execution models, which can minimize the overheads imposed by the host-side software interventions. In our design, the data are not transferred per kernel execution as all processing elements can directly access PRAMs in the target accelerator. In addition, `DRAM-less` allocates a dedicated processing element to handle kernel images and schedule data processing tasks, which can simplify the near data processing models and make them easy-to-use.

II. Background

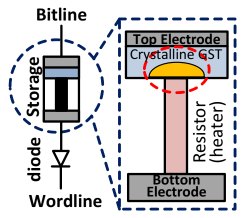
In this section, we will introduce a multi-partition architecture of PRAM and its memory interface, which will be used for our proposed `DRAM-less`.

A. Microarchitecture

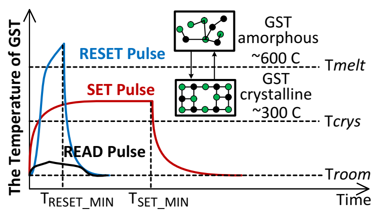
Storage core. Figures 2a and 2b illustrate the PRAM storage cell structure and its thermal characteristics, respectively. PRAMs write data (i.e., *program*) by quickly heating chalcogenide glass, an alloy of germanium, antimony, and tellurium (GeSbTe), which is called *GST* [27], [28]. Based on the applied temperature, the storage core cell can be "phase changed" to either crystalline or amorphous form, which are referred to as *SET* and *RESET*, respectively [27], [29], [30]. As shown in Figure 2b, if the temperature is near 300°C (T_{cys}), GST exhibits a low resistance that represents the digit "1". If the temperature is higher than 600°C (T_{melt}), the resistance of GST becomes higher, and it indicates the digit "0" [31], [32], [29]. In contrast to the program process, a read process only requires sensing the target resistance, which can be performed at room temperature (T_{room}). Because T_{crys} and T_{melt} should be maintained for a while, SET pulse width is much longer than RESET or READ pulse widths. A PRAM overwrite practically comprises a sequence of RESET and SET processes [33], [34], [35], [36]. Practically, this in turn makes the write latency longer than the read [37]. To address the performance asymmetry issue, our PRAM employs multiple row buffers, and its bank is composed of many storage arrays, referred to as *partitions*.

Multi-row buffers. Figure 3a depicts a high-level view of our PRAM and the corresponding interface, which can be integrated into the *processing elements* (PEs) of the accelerator via our PRAM controllers. Each PRAM module

Authorized licensed use limited to: Xiamen University. Downloaded on July 22,2021 at 07:14:30 UTC from IEEE Xplore. Restrictions apply.



(a) PRAM cell detail.



(b) PRAM thermal characteristics.

Figure 2: PRAM cell and thermal features.

employs multiple row buffers that connect the underlying storage cores (array) and exposes them through a low-power double data rate interface (LPDDR2-NVM) [24]. To make the multiple row buffers identical, each row buffer is logically paired by an address buffer and a data buffer, and selected by a given buffer identification number. These address and data buffers are referred to as *row address buffer* (RAB) and *row data buffer* (RDB), respectively. While the RAB can accommodate the address and command of an incoming request, the RDB buffers the 256-bit contents of the underlying PRAM bank. Since the address space of our multi-partition PRAM is greater than a traditional DRAM address space, the target address bits are shipped to the PRAM row decoder through two separate address parts: i) *an upper row address* and ii) *a lower row address*. The upper row address can be accommodated by a RAB, whereas the lower row address can be directly delivered to the target decoder via our FPGA memory controller. For a read, if the target data is in a RDB, the data can be directly transferred from the row buffer to the buffer address (BA) pins. For a write, our PRAM employs an additional internal buffer, called *program buffer* (cf. Section II-B).

Multi-partition architecture. One of the ways to shorten the latency of a PRAM access is to maximize array-level parallelism. The new 3x nm PRAM technology employs a multi-partition architecture that can serve 256-bit data in parallel. As shown in Figure 3b, a single PRAM bank employs 16 partitions, each containing 64 sub-arrays, which are referred to as resistive *tiles*. Individual tiles employ many PRAM cores with 2048 bitlines (BLs) and 4096 wordlines (WLs). To reduce the parasitic resistances of BLs and WLs, and to address the sneak path issue [38], [39], [40], a partition is split into two parts (referred to as *half partitions*). Each partition employs a local Y-decoder (LYDEC) on both sides of the tile. It also groups every two tiles as a *block* by using a dual-WL scheme [41]. To maximize the degree of parallelism, each tile within a partition is connected to a sub-wordline driver (SWD), and all SWDs in the partition are connected to the main wordline driver (MWD). This new architecture enables a PRAM bank to simultaneously perform 64 I/O operations per half partition, which can theoretically perform a 128-bit parallel data access for each partition. Lastly, this architecture locates the sense amplifiers and write drivers in front of each bank, which are connected to the multi-row buffers (RDBs). Through our empirical evaluations, we observed that our PRAM's multi-partition architecture exhibits a longer program latency than the latency originally reported in [42] (cf. Section VI). However, it performs 256-bit parallel I/O operations at bank-level, which is much wider than the level at which the previous PRAM technologies operate [43], thereby quickly servicing the memory references coming from processing elements.

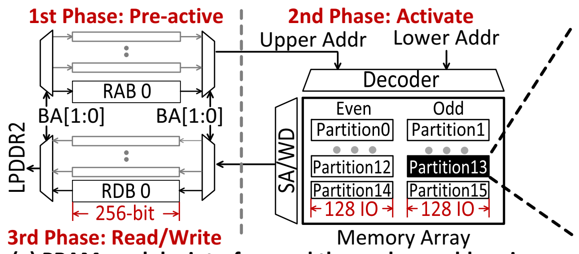
B. New Memory Interface Protocol

Our new multi-partition PRAMs provide different memory arrays, row buffer designs and cell characteristics, compared to traditional DRAMs. This makes conventional DRAM protocols and programming methods infeasible to apply in a straightforward manner.

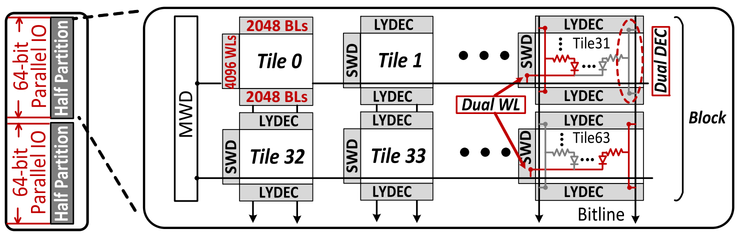
Three-phase addressing. For this reason, our PRAM employs LPDDR2-NVM to access all of its internals, whose command set to operate is different with that of conventional DDR interface protocols. Specifically, to enable a large memory space of PRAM over a limited set of I/O pins, LPDDR2-NVM uses a new interface protocol, called *three-phase addressing*. LPDDR2-NVM introduces two different addressing phases (i.e., *pre-active phase* and *activate phase*) in transferring a complete row address to the row decoder of a PRAM bank, whereas it transfers the column address in the third addressing phase (i.e., *read/write phase*). In a pre-active phase, an external memory controller can select a RAB by sending a BA selection signal and stores the upper row address (associated with the target row) into the selected RAB. The controller then needs to send the remaining lower row address to the target PRAM bank, such that it can compose the actual row address by merging the lower row address with the memory address (retrieved from the selected RAB). In the meantime, the target PRAM module loads and stores the row data to the RDB (associated with the RAB), called an activate phase. After this activate phase, a specific data location within the selected RDB (i.e., column address) can be delivered by a *read/write phase command*. If read, the target data will be available to pull out from the designated RDB for each interface clock signal.

Overlay window and program buffer. Writing to a storage core directly (by referring the RDB) will suspend all operations targeting the corresponding PRAM module. To address this, our new PRAM introduces a special set of registers (*overlay window*) and a buffer (*program buffer*), each respectively being related to a write operation handling and non-blocking write management. Figure 4 shows how the overlay window and program buffer are connected to the address space of a PRAM module. The program buffer is located in front of the multiple PRAM partitions with a write driver, and its addresses are mapped to the end of overlay window space. As shown in the left of such figure, the overlay window contains 128-byte meta-information (comprising window size, buffer offset, and buffer size)

Authorized licensed use limited to: Xiamen University. Downloaded on July 22,2021 at 07:14:30 UTC from IEEE Xplore. Restrictions apply.



(a) PRAM module, interface and three-phase addressing.



(b) Half partition and tile architectures.

Figure 3: Internal architecture of our multi-partition phase change random access memory (PRAM).

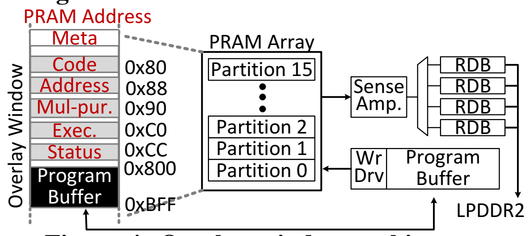


Figure 4: Overlay window architecture.

and a set of control registers that include command code, data address, execution and status registers. The address range of this overlay window can be mapped to anywhere of PRAM address space through an *overlay window base address* (OWBA). After configuring the OWBA, an external memory controller can write data into the program buffer via the three-phase addressing.

Whenever the host requires to persistently program them to a PRAM partition, its memory controller should initiate the program by updating command and execute registers of the overlay window. The target PRAM module then automatically fetches the data from the program buffer and stores it into the designated PRAM partition, based on the row address. Once the PRAM write begins, the external controller can check up the operation progress of target memory partition via the status register existing in the overlay window. These I/O operations via the overlay window are useful if there is a high data locality and long request interval. This is because the overlay window can be in parallel mapped to other addresses while writing data to the target. All register manipulations for the overlay window (and program buffer) should be performed by obeying the aforementioned three-phase addressing protocol. All these new features are not observed by conventional DRAM and even other flash technologies. Therefore, a new PRAM control logic is required appropriately translating an incoming memory instruction to multiple three-phasing addressing operations and controlling all LPDDR2-NVM timing parameters. In addition, the new controller logic needs to manage the aforementioned internal resources for better performance, such as multiple row buffers, program buffers, and overlay windows.

III. High-Level View of DRAM-less

In this section, we will analyze the root causes of performance degradation and high energy consumption issues (cf. Figure 1) and explain the architectural components that aim to eliminate such overhead.

A. Challenges

Figure 5 compares the architecture design and communication protocol between a conventional accelerated system and a system that employs our `DRAM-less` architecture. The accelerated system employs a CPU, an accelerator and an SSD as its external storage (Figure 5a). To prepare data for the accelerator, the conventional hardware acceleration approach requires retrieving a large amount of low-level data (in the form of files) from the SSD and deserializes them as a representation of objects within the host's DRAM. The host then transfers the data to the internal DRAM of the accelerator. Once the accelerator completes its data processing task(s), the results are written back to the SSD in an inverse order of data loading procedure. In this data transfer model, a CPU is required to frequently intervene to move the data among multiple user applications and OS modules. As the hardware accelerator and SSD devices, in practice, employ different software stacks, such interventions introduce many user/kernel mode switches and redundant data copies, which result in the waste of many CPU cycles.

To reduce these software overhead and redundant memory copies, as shown in Figure 5b, we build a hardware automated PRAM subsystem and tightly integrate it in the multicore accelerator. In this design, we replace the accelerator-side DRAM with our proposed PRAM subsystem and allows all processing elements to access PRAM over a set of conventional memory instructions such as loads and stores. Since the large capacity of PRAM can accommodate a whole set of data to process, applications in our `DRAM-less` directly load the input data from the internal PRAMs without an external storage access. Consequently, as shown in Figure 5b, the host needs to prepare only computation kernel(s) without any input preparations (a); it can simply issue the kernel to the target hardware accelerator for its execution (b). We also implement an FPGA-based controller to automate all data accesses in PRAM modules, which fully complies with the conventional memory instructions, `DRAM-less` can remove the data management (filesystem and storage stack) from its data processing I/O path.

Authorized licensed use limited to: Xiamen University. Downloaded on July 22,2021 at 07:14:30 UTC from IEEE Xplore. Restrictions apply.

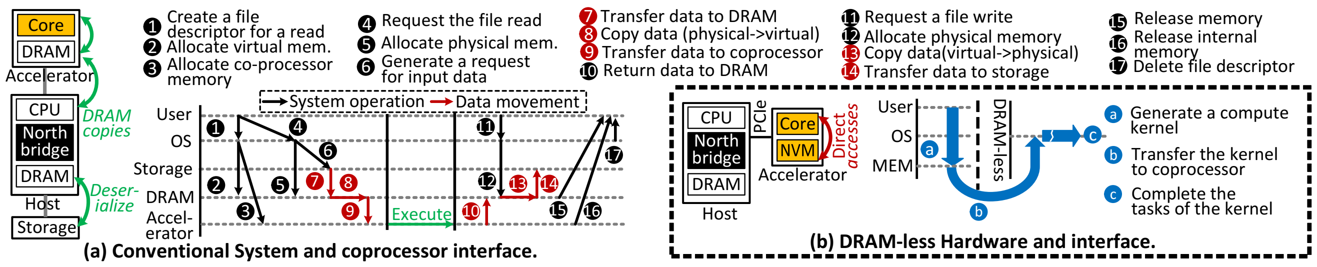


Figure 5: Comparison between a traditional accelerated approach and our `DRAM-less`.

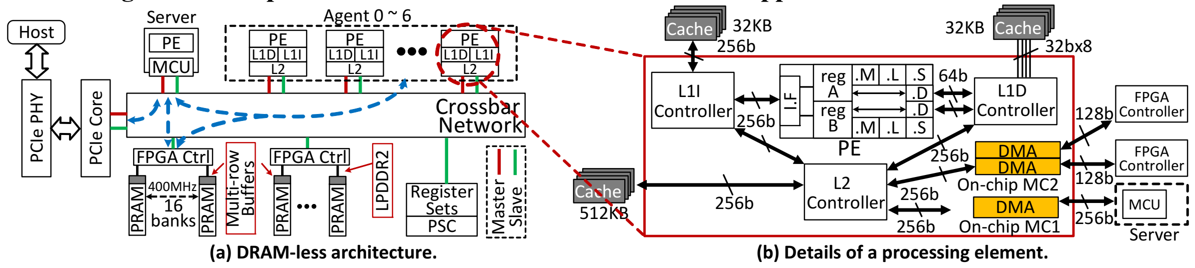


Figure 6: `DRAM-less` architecture and processing elements.

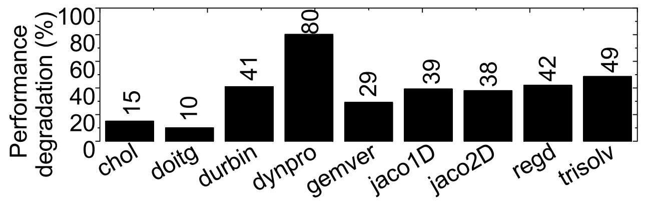


Figure 7: Performance degradation of employing traditional firmware compared to an oracle PRAM controller.

B. DRAM-less Data Processing Architecture

Processor. Figure 6 illustrates a high-level view of the proposed `DRAM-less`'s internal structure and microarchitecture of *processing element* (PE) therein [44]. The computation core of PE is designed by a multi-way single instruction, multiple data (SIMD) architecture, which is aimed to improve the performance of vector data processing. As shown in Figure 6b, A PE contains two sets of four functional units (.M, .L, .S, and .D) and two general-purpose registers (reg A and reg B), which are connected to the L1 instruction cache through an instruction fetch module. While two .D functional units execute load and store instructions, .M, .S, and .L perform multiplications, general sets of arithmetic, logical and branch function, respectively. That is, each PE has two general-purpose functional units (.S units) and two load/store units (.D units), and all they operate on a “RISC-compatible ISA”. Thus, a general-purpose application can be easily converted as a near-data processing kernel without or with a few modifications. Although not mandatory, programmers can accelerate vector processing further by embedding DSP-intrinsic that activates two .M units, such as multi-way floating-point multiply/add and 16-bit integer intrinsic, which merges multiple multiply and accumulation operations into one. In total, there are 64 functional units in the accelerator, which can offer 358.4 Gops/s data processing capability. We choose this multicore architecture for our accelerator prototype implementation as a representative low-power processor [1].

Datapath. As shown in Figure 6a, multiple PEs have their own L1 and L2 caches, which are connected to the crossbar network via a master port and a slave port. To process data in parallel, most PEs of our `DRAM-less` are allocated in handling the computational *kernel* provided by the host. While these PEs, referred to as *agents*, perform near-data processing, we designate one of PEs as a *server* to schedule all kernel executions on the agents by resuming and suspending them via a “power/sleep controller” (PSC). This server also manages the PRAM traffic requested by other agents. The server employs a memory controller unit (MCU) that takes over the L2 cache misses of an agent and administrates all the associated PRAM accesses by collaborating with the underlying FPGA controllers. Each agent's L2 controller can generate memory requests through two on-chip memory controllers (MC1/MC2). The MC1 and MC2 connect MCU and FPGA through 256-bit bus and 128-bit bus, respectively (cf. Figure 6b). The FPGA contains two separate LPDDR2-NVM channels, each channel being able to contain 16 400-MHz PRAM modules in our design. The server simply sends a memory read or write message through the bus, and then, the FPGA controllers take over all LPDDR2-NVM transactions (the corresponding messages) upon PRAMs. All these internal components are linked by the crossbar network, which is also bridged to `DRAM-less`'s PCIe module in enabling a host to communicate with the server.

PRAM subsystem. Even though the server and MCU handle

[1]Recently, [2] examines multiple low-power accelerators, including a 256-core GPU [45], Kintex FPGA 32-lanes [25] and RISC-based manycore [46]. It reports that the bandwidth per watt of our platform is better than that of [25] and [46] by 85× and 9×, respectively, and similar to what [45] provides.

Authorized licensed use limited to: Xiamen University. Downloaded on July 22,2021 at 07:14:30 UTC from IEEE Xplore. Restrictions apply.

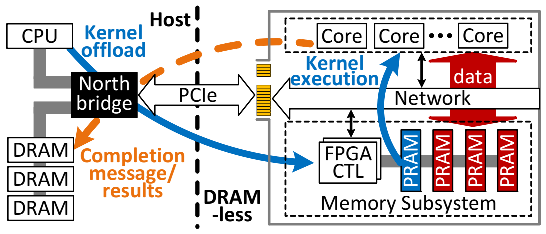
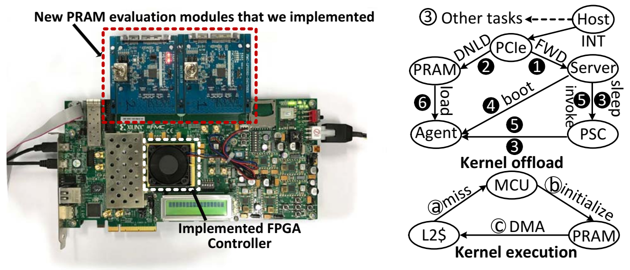


Figure 8: Kernel offloading and execution.

all the read/write operations coming from parallel PEs, their memory requests should be converted to LPDDR2-NVM transactions and managed by the three-phase addressing protocol. One potential solution is to employ conventional firmware, existing in Optane SSD, to manage PRAM. However, we observe that the conventional firmware can take longer execution time than PRAM access latency. To be precise, we compare the performance between an accelerator with the conventional firmware and an oracle environment that manages PRAM with no overhead. Figure 7 shows the system performance degradation of using the firmware compared to the oracle environment. The firmware degrades the system performance by upto 80% in the data-intensive workloads, due to its long execution time. To mitigate the performance penalty, we design a hardware-automated PRAM subsystem and controller to manage multiple row/program buffers and govern three-phase addressing protocol, respectively. Specifically, our PRAM controller within the FPGA can selectively skip parts of the three addressing phases to reduce the I/O latency by being aware of the states of PRAM internal resources (e.g., RAB and RDB). In cases where the target's upper row address already exists in a RAB, the controller skips the corresponding preactive phase and directly enables the activate phase. If the target data are ready on a RDB, the activate phase can be skipped, which can further reduce the I/O latency. On the other hand, all write requests are performed through the program buffer in the overlay window associated with the designated PRAM module. To reduce the number of PRAM accesses, the server initiates a memory request based on 512 bytes per channel (32 bytes per bank) by leveraging its L2 cache, and tries to prefetch data by using all RDBs across different banks. Since the current memory interface generator (MIG) "does not" support PRAM, in addition to the memory controller, we implement our own PRAM physical layer on a 28nm Xilinx FPGA 19K logic cells [25], which manage our multi-partition PRAM modules over LPDDR2-NVM, as shown in Figure 9a. Our physical layer (PHY) addresses the differences of operating frequency between PRAM and FPGA at 400MHz.

IV. Data Processing Models

Figure 8 shows an overview of kernel offloading and execution models of our `DRAM-less`. To execute a near-data processing task, the host needs to prepare its kernel



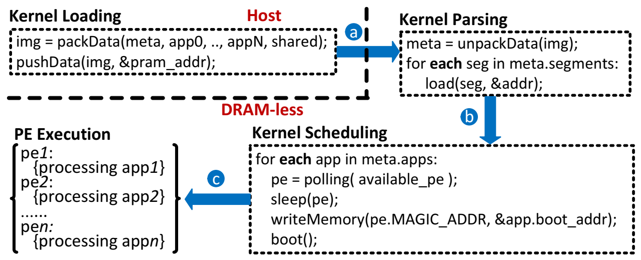
(a) Memory controller prototype. (b) Kernel controls.

Figure 9: Prototype and kernel controls.

image and offload the image to the PRAM subsystem of `DRAM-less`. `DRAM-less` then automatically schedules the offloaded tasks across the multiple PEs and executes them on its internal PRAM address space. Since the PEs can directly access data from PRAM via load/store instructions, the existing data processing applications do not require any source-level modifications. In addition, our design does not need to go through a complicated storage software stack at the host-side for its data processing. Once the task completes, `DRAM-less` responds to the host with a completion message or an execution result. In this section, we will explain these execution/programming models of our `DRAM-less` in details.

Kernel offload and execution methods. Traditional kernel scheduling methods including message passing interface (MPI) [26] and compiling offloading features require the assistance of the host to coordinate the kernel scheduling in the accelerators. In contrast, DRAM-less can fully eliminate the intervention of the host. Figure 9b shows the process of a kernel offloading and execution for our proposed `DRAM-less`. At the beginning of data processing, the host can issue a PCIe interrupt to `DRAM-less`, which is internally forwarded from the PCIe module to the server (❶). From this point onward, all kernel codes can be downloaded into a designated image space on PRAMs through the server if it needs (❷). Once the kernel download is completed, the host can process other tasks without waiting for the completion of the accelerator (③), while the server schedules the downloaded kernels to each available agent. The server puts the target agent in a sleep mode by using PSC (❸). The server then stores the image address of PRAM to L2 cache of the target agent as a boot address (❹). After saving the image address, the server revokes the agent via PSC (❺). Then, the agent will load and execute the kernel (❻). During this execution step, all the memory requests, generated by the L2 cache misses, are issued to the server's MCU (ⓐ). It consequently initiates DMA (ⓑ) between the target PE's L2 cache and the overlay window or RDBs of the target PRAM (ⓒ). Note that the underlying PRAMs are considered as a private resource of the accelerator. Therefore the host is prohibited to directly access the PRAMs as a block storage through a filesystem interface. To address this, we employ the server of our accelerator to manage the data requests

Authorized licensed use limited to: Xiamen University. Downloaded on July 22,2021 at 07:14:30 UTC from IEEE Xplore. Restrictions apply.



```
img = packData(meta, app0, .., appN, shared);
pushData(img, &pram_addr);
```

```
meta = unpackData(img);
for each seg in meta.segments:
    load(seg, &addr);
```

```
for each app in meta.apps:
    pe = polling( available_pe );
    sleep(pe);
    writeMemory(pe.MAGIC_ADDR, &app.boot_addr);
    boot();
```

```
pe1:
  {processing app1}
pe2:
  {processing app2}
......
pen:
  {processing appn}
```

Figure 10: DRAM-less's programming model.

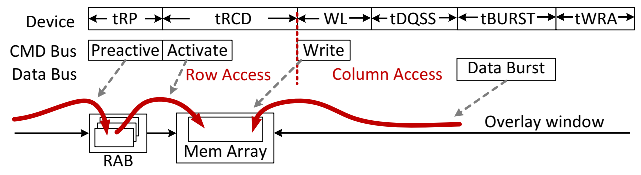
coming from the host. For example, the host can ask the server to bring data from the host-side system memory via PCIe messages. The server then records the data to PRAM-subsystem by issuing memory write requests over our memory controllers.

Programming model. Figure 10 shows the programming model of our `DRAM-less`, which is associated with its communication protocol (cf. Figure 5b). Users can pack and offload the kernel(s) to the accelerator via `packData` and `pushData` application interfaces. The formal parameters of `packData` interface include metadata (i.e., *meta*), which defines the accelerator's memory address to download code segments for multiple applications (e.g., *app0, app1, ..., appN*) and shared common codes (i.e., *shared*). On the other hand, the parameters of `pushData` interface contain the pointer, which refers to the host's memory address of the image (i.e., *img*) and the accelerator's memory address to download (ⓐ). Once the kernel image has arrived in the accelerator's memory, the server starts to extract metadata information from the image via `unpackData` interface and loads the code segments to the target addresses described in metadata (i.e., `load`). After parsing the kernel image, the server starts to check each PE for availability (i.e., `polling`). If one PE is in idle, the server will power off this PE, assign a kernel to the PE by updating PE's magic address with kernel's boot entry address, and reboot such PE (ⓑ). The PE keeps continuing the execution until completing the data processing (ⓒ). In our emulation work, we leveraged a set of TI code generation tools [47], [48], [49]. Note that the memory protection and access control are regulated by aforementioned tools, which may be limited for other accelerators.

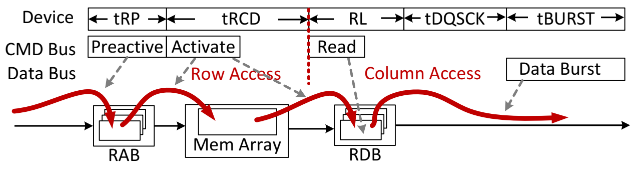
V. Hardware Automation Details

A. Microarchitecture Awareness

Timing management. Figures 11a and 11b illustrate the write and read timing diagrams of our PRAM modules, respectively. A row access for both reads and writes consists of the pre-active and activate phase operations described in three-phase addressing. Specifically, the pre-active is similar to the row precharge time (i.e., tRP) of conventional DDR, but it handles the target RAB to update an upper row address within tRP. The activate time can be classified by a row address to column address delay (i.e., tRCD), which includes



(a) Write timing diagram.



(b) Read timing diagram.

Figure 11: PRAM module timing diagrams.

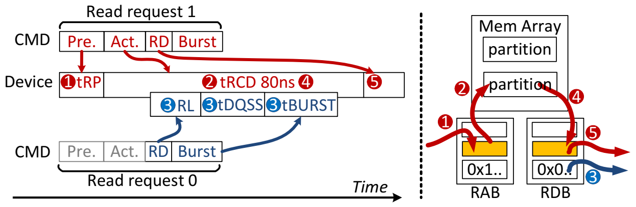


Figure 12: Multi-resource aware interleaving.

the address composition time for target row location (by combining the values stored in the target RAB and the lower row address) and the memory operation time. In tRCD, the target module also checks whether the composed row address is in the range of overlay window or not. If the designated address is matched (with the ones of overlay windows), the data associated with the target row address will be processed by the register sets and stored into the program buffer, not actual PRAM storage cores. Otherwise, the target data will be referred to by the designated PRAM array. This tRCD also includes the time to fetch the data from the target row to RDB. In a read phase, it consumes a read preamble period that consists of a read latency (i.e., RL) and data signal strobe output access time (i.e., tDQSCK). The data is then delivered out from the RDB by referring to the column address, which is embedded in the read phase command, during the data burst time (i.e., tBURST). The memory timing for a write phase command is not much different with that for such read phase command. Specifically, instead of RL, it exhibits a write latency (i.e., WL) in a write preamble period. The program time sequence also includes a write recovery period (i.e., tWR) to guarantee that all data in the program buffer are programmed to target PRAM array.

Multi-resource aware interleaving. To reduce the latency of data movements between PRAMs and L2 caches, we schedule memory requests by being aware of PRAM's multiple partitions and row-buffers. Specifically, each PRAM module in the proposed `DRAM-less` can sense data out from a partition to a RDB, while transferring data out from other RDB(s) to target cache(s), in parallel. Thus, one can fully overlap the times to transfer data with the latency to

Authorized licensed use limited to: Xiamen University. Downloaded on July 22,2021 at 07:14:30 UTC from IEEE Xplore. Restrictions apply.

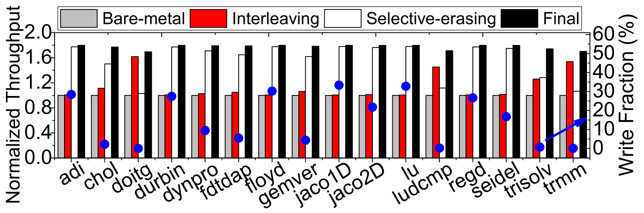


Figure 13: Interleaving and selective erasing.

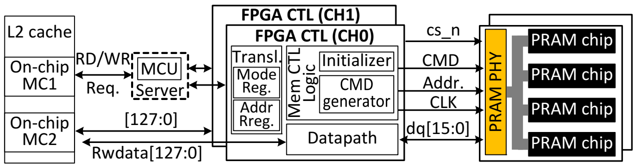


Figure 14: Implementation of FPGA controller.

access a memory partition. Figure 12 shows an example of our multi-resource aware interleaving; there are two requests (req-0/req-1), each targeting a different partition but within a same chip. Let us suppose that the pre-active and active commands of req-0 were initiated just before issuing the pre-active command of req-1 (1). While it takes tRP and tRCD related to req-1, the controller can send the read phase command for the different target partition (with a different RDB address). Since the RDB associated with req-0 is ready to transfer, during tRCD of req-1 (2 and 4), the controller can bring req-0 data out and place them on the target L2 cache, which consumes RL, tDQSS and tBURST of req-0, in tandem (3). Once it has transferred all data, req-1's RL, tDQSS and tBURST can be processed while the controller is accessing another partition (5). In this way, we can make the data transfers invisible to agent PEs. Note that our multi-resource aware interleaving is different with bank interleaving [50] in the sense that all the memory requests interleaved within a single bank.

Selective erasing. A program is in practice composed by RESET and SET, which introduces a long write latency. Similar to other NVMs (e.g., flash), PRAM also supports erase operations that reset a large number of cells (greater than cells in a program unit) to pristine state. Overwrites on erased cells only require SET operations, which can reduce the write latency. We evaluated performance of the erase operations on our multi-partition PRAM modules, and observed that the latency of erase operation itself is around 60 ms, which is 3K times longer than that of an overwrite. This extremely long erase latency can, unfortunately, block all coming requests to the target partition. To address this challenge, we investigate a simple but effective optimization for overwrites, referred to as *selective erasing*, which can reduce the overwrite latency without the penalty of erase operations. In contrast to the erase operations, selective erasing can mimic RESET operations to reset the cells in the program unit (word) in advance. Since RESET is a process to toggle the target cells from the programmed status to the pristine state, our approach prepares word-aligned data filled by all zeros and program the data to the designated address. Specifically, while the server loads the target kernel, the PRAM subsystem can selectively program the all-zero data word for only the addresses that will be overwritten soon (but before completing the corresponding computation). We implemented this selective erasing technique and tested it on the PRAM module. Our evaluation results reveal that selective erasing can reduce the overwrite latency by 55%, on average. We also observe and there is no bit error per access during selective erasing and after the erase operation.

Performance improvement. Figure 13 compares a noop scheduler [51] on multi-partition PRAM, denoted by `Bare-metal`, with the ones that employ aforementioned two optimization techniques, each being referred to as `Interleaving` and `selective-erasing`, respectively. In addition, the evaluation results include performance of our subsystem scheduler that puts both optimization techniques together, called `Final`, under the execution of Polybench [18]. The figure also shows the write ratios (pointed by circle) that we tested for each application. One can observe from this figure that `Interleaving` improves data processing bandwidth with `Bare-metal` by as high as 54% (e.g., `trmm`). However, there are several workloads (e.g., `adi`, `floyd` and `jaco1D`) that have almost zero benefit, compared to `Bare-metal`, due to the long latency overhead of overwrites. In this case, `selective-erasing` exhibits better bandwidth than `Bare-metal` by 57%, on average. By putting all them together, `Final` enhances bandwidth of `Bare-metal` across all tested applications, by 77% on average, and therefore, we apply this to our `DRAM-less` as default.

B. Logic Implementation

Our FPGA-based PRAM controller supports simple read and write interfaces, which can be used by the server's MCU. They also provide read and write data interfaces, which are mapped to two 256-bit datapath registers. Figure 14 illustrates the block diagram of our PRAM controller, which consists of translator, memory controller, and datapath in detail. All these components can communicate with PRAM chips through 400MHz PRAM physical layer (PHY).

Translator and datapath. The translator of our PRAM controller simply exposes a 32-bit address and a 32-bit mode registers, which can be linked to the server's MCU. Since PRAM module does not allow to directly write data to its storage cores through LPDDR2-NVM, our controller's translator handles the overlay window of target PRAM module on behalf of MCU. The PRAM controller first stores a write operation code to the code register, which is mapped to 0x80 of the overlay window base address (OWBA, cf. Figure 4) to inform the memory operation type to the target PRAM bank. The translator then records target row address at the address register mapped to OWBA+0x8B and informs

Authorized licensed use limited to: Xiamen University. Downloaded on July 22,2021 at 07:14:30 UTC from IEEE Xplore. Restrictions apply.

	Hetero	Heterodirect	Hetero-PRAM	Heterodirect-PRAM	NOR-intf	Integrated-SLC	Integrated-MLC	Integrated-TLC	PAGE-buffer	DRAM-less
Heterogeneous	✓	✓	✓	✓	×	×	×	×	×	×
Internal DRAM	✓	✓	✓	✓	×	✓	✓	✓	✓	×
NVM read (us)	50	50	0.1	0.1	290	25	50	80	0.1	0.1
NVM write (us)	800	800	10/18	10/18	120	300	800	1250	10/18	10/18
NVM erase (us)	3500	3500	N/A	N/A	N/A	2000	3500	2274	N/A	N/A

Table I: Important configuration parameters for all accelerated systems that we evaluated.

Parameter	Values	Parameter	Values	Parameter	Values
RL (cycle)	6	tRP (cycle)	3	tDQSS (ns)	0.75-1.25
WL (cycle)	3	tRCD (ns)	80	tWRA (ns)	15
tCK (ns)	2.5	tDQSCK (ns)	2.5-5.5	tBURST(cycle): BL4/BL8/BL16	4/8/16
RAB	4	RDB	32B,4RDBs	PRAM write (us)	10-18
Channels	2	Packages	16	Partitions	16

Table II: Charaterized PRAM parameters.

the data burst size to the designated PRAM module in terms of bytes through the multi-purpose register, which is mapped to OWBA+0x93. After setting these registers on the overlay window, the translator starts writing data into the program buffer (OWBA+0x800) and then executes the write operation by configuring the execute register (mapped to OWBA + 0xC0). On the other hand, for a read, the translator directly forwards it to a command generator of the memory control logic. Since the operand size of load and store instructions that our PEs use is 32 bytes, we implement 256-bit registers for each load and store operation in the datapath.

Memory control logic. The hardware-automated control logic in our controller consists of two parts; one is a command generator and another one is an initializer (cf. Figure 14). The initializer handles all PRAMs' boot-up process by enabling auto initialization, calibrating on-die impedance tasks and setting up the burst length and overlay window address. On the other hand, the command generator handles three-phase addressing (pre-active, activate and read/write) and LPDDR2 transactions over our PRAM PHY. It disassembles the target address into an upper row address, a lower row address, a row buffer address, and a column address. These decomposed addresses are then delivered to PRAM through 20-bit DDR signal packets. A signal packet composes the operation type (2∼4 bits), row buffer address (2 bits), target address (7∼15 bits) of either overlay window or target PRAM partition. At a pre-active phase, it selects the target row buffer by delivering 2-bit BA signal and stores the upper row address to the selected RAB. In the next phase, the command generator signals the lower row address and the buffer address to the target PRAM module. This will activate the target row and program the data to target row by delivering the data, which is stored by the program buffer address. If it is a read, the internal sensing circuit of target PRAM transfers data from the corresponding row to the RDB associated with the selected RAB.

VI. Experimental Evaluations

To evaluate diverse accelerated systems, we implement following three groups of data processing solutions. The key comparisons among all configurations are listed in Table I.

1) **Heterogeneous systems**: We prepare four traditional heterogeneous computing systems (cf. Figure 5a) by using different accelerator/storage combinations. "`Hetero`" and "`Hetero-PRAM`" are the accelerators that employ flash-based and PRAM-based SSDs (i.e., Intel Optane [23]) as their external storage, respectively. On the other hand, "`Heterodirect`" and "`Heterodirect-PRAM`" are the accelerators, which are same with aforementioned two, but use a zero-overhead peer-to-peer DMA [13], [14], [52] to communicate with the external SSDs.
2) **Aggressive integrations**: We implement three advanced accelerators, which are a more aggressive solution than the previous software-assisted approaches. "`Integrated-SLC`", "`Integrated-MLC`" and "`Integrated-TLC`" put single-level-cell (SLC) [20], multi-level-cell (MLC) [21], and triple-level-cell (TLC) [22] flash based SSDs into the hardware accelerator, respectively. These solutions are similar to active SSD approaches [53], [54], [55], [12], [56].
3) **PRAM-based accelerators**: We tightly integrate PRAMs into multi-core coprocessors with four different NVM interfaces and memory technologies. "`NOR-intf`" uses 9x nm parallel PRAM [43] that employs a serial peripheral NOR flash interface. Like our PRAM, it allows a host to access the memory at byte-granule. In contrast, "`PAGE-buffer`" leverages the 3x nm memory sample that we used for `DRAM-less`, but performs all I/Os via a page-based interface with an assistance of its internal DRAM. "`DRAM-less`" directly accesses PRAM-based subsystems over multi-resource aware interleaving and selective erase techniques. Compared to `DRAM-less`, "`DRAM-less (firmware)`" that we implemented replaces the hardware automated memory control logic with traditional SSD firmware, used in block storage devices (e.g., NVMe SSD and Optane SSD). The SSD firmware is implemented on a 3-core 500MHz embedded ARM CPU, similar to the controllers of commercial SSDs [57].

Note that all the hardware accelerators are implemented on commercially available platform that employs eight 1GHz embedded processors, each having eight functional units, 64KB L1 and 512KB L2 cache [19]. All SSDs used for this evaluation are emulated on a real system, and the size of their internal DRAM buffer is 1GB. This buffer configuration is also applied to integrated accelerator approaches whose memory should be accessed with a page granule.

The important device properties and timing parameters of our PRAM modules are listed in Table II. Overall, the read

Authorized licensed use limited to: Xiamen University. Downloaded on July 22,2021 at 07:14:30 UTC from IEEE Xplore. Restrictions apply.

Workload Abbr.	adi	chol	doitg	durbin	dynpro	fdtdap	floyd	gemver	jaco1D	jaco2D	lu	ludcmp	regd	seidel	trisolv	trmm
Total instruction (Billion)	2.6	1.1	2.2	0.25	0.29	1.5	3.0	0.51	0.29	0.34	3.0	2.7	0.17	2.81	0.14	3.0
Output size / Input size	0.33	1	1	0.0078	0.0078	0.33	1	0.016	0.5	0.5	1	0.016	0.33	1	0.016	0.5
Data volume (MB)	768	910	896	903	903	768	896	845	960	960	896	975	768	896	975	960

Table III: Workload characteristics.

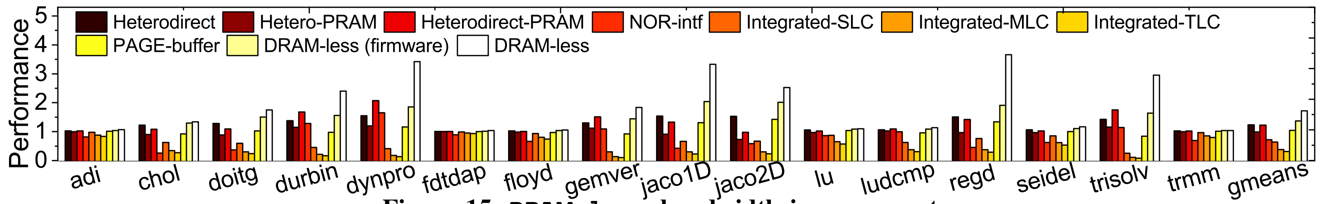


Figure 15: DRAM-less bandwidth improvement.

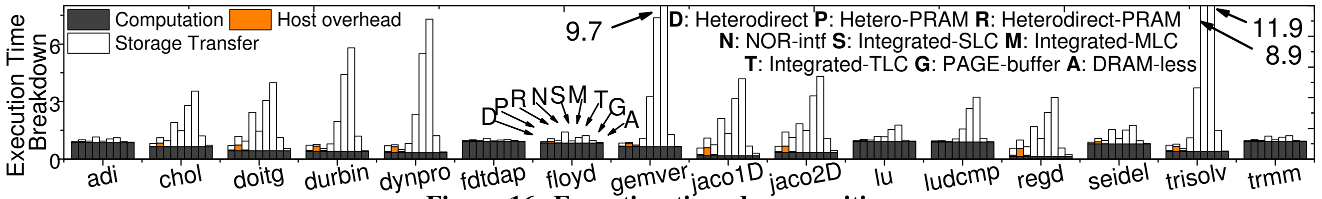


Figure 16: Execution time decomposition.

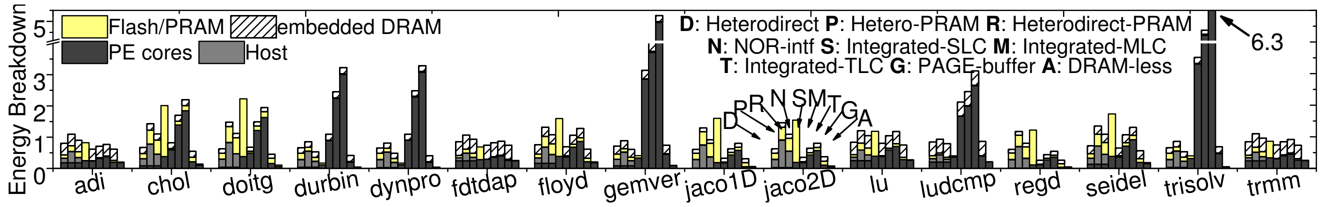


Figure 17: Energy decomposition.

latency is around 100 *n*s, including three-phase addressing (i.e., RL, tRCD, tRP and tBURST). In contrast, the write latency is approximately 10 *u*s (overwrites require extra 8 *u*s), which includes all program buffer writing delays (i.e., WL, tRCD, tRP and tBURST) and PRAM cell program delays. While we observed that the write latency can be half, this latency is used for only special cases, such as factory programs, which need an extra power source from outside and limit the number of write processes. Thus, we choose ordinary timing parameters, as 10 $\sim$ 18 *u*s, and do not consider the shorter latency. Note that, even though 9x nm PRAM [43] offers a byte addressable interface, its read/write latencies are 290 *u*s/120 *u*s.

We port Polybench suite [18] to the evaluated platform with several optimizations. Specifically, each workload is split into multiple compute kernels, which can be simultaneously executed across all different PEs of accelerators. We modify the benchmark to fully utilize the computing power of each PE by embedding DSP intrinsic (e.g., multi-way floating-point multiply/add and 16-bit integer-intrinsic) into the benchmark. The important characteristics of workloads that we tested are listed in Table III. The intensiveness of writes is classified by the amount of output size per input size. For example, *doitg* is a write-intensive application even though there are more read requests generated by last level cache at runtime (cf. Figure 13). We also show the data volume of each application. The data volume has been increased by more than 10 times, compared to the original configuration in the benchmark. Before the evaluation, we initialize the data and place it in the persistent storages (e.g., SSD for heterogeneous systems and PRAM/Embedded-Flash for integrated approaches), which is a common practice in prior research [13], [12].

A. Bandwidth Improvement

Figure 15 shows throughput of data processing for the 10 hardware accelerations we evaluated, which are normalized to those of our baseline, `Hetero`. This figure shows that `Heterodirect` exhibits, on average, 25% better performance than the baseline. The reason for this improvement in performance is because `Heterodirect` reduces the number of data copies within the host by directly forwarding target data from the underlying SSD to the accelerator. For read-intensive workloads (e.g., *durbin*, *dynpro*, *gemver* and *trisolv*), `Hetero-PRAM` and `Heterodirect-PRAM` improve the performance of `Hetero` and `Heterodirect` by 15% and 24%, on average, respectively, due to PRAM's shorter latency. However, all these benefits are unfortunately limited if applications generate a large size of output data (e.g., *chol*, *doitg*, etc.) and/or require many data to process (e.g., *jaco1D*, *jaco2D* and *regd*). This is because flash is well optimized for block interface operations (and exhibit better performance on bulk writes), whereas PRAM is designed towards supporting byte-granular accesses. Interestingly, the performance of `NOR-intf` is worse than

Authorized licensed use limited to: Xiamen University. Downloaded on July 22,2021 at 07:14:30 UTC from IEEE Xplore. Restrictions apply.

Integrated-SLC by 27%, on average, except for a few read-intensive workloads (*durbin*, *dynpro*, etc). This is because, even though NOR-intf supports byte-addressability, all PRAM write accesses are serialized by 16-bit low-level memory operations, and its bandwidth for reads and writes is 2x and 101x worse than flash's page-level bandwidth (i.e., 16KB parallel I/O). While PAGE-buffer offers the performance, 78% better than Integrated-SLC, it is even worse than NOR-intf by around 19% for the read-intensive workloads. This performance degradation is caused by the disability of byte granular accesses. In contrast, all the PE employed by DRAM-less (firmware) can access the targets with byte-granularity, which in turn eliminates the memory operations involved in data transfer across different devices. However, the traditional firmware constrains the average performance improvement of DRAM-less (firmware) to 31%, compared to PAGE-buffer. This is because memory requests, generated by the multiple PEs in the accelerator, have to be serially processed by the traditional firmware, which suffers from long delay. By replacing the traditional firmware with our hardware automated control logic, DRAMless can significantly reduce the firmware intervention latency. Therefore, DRAM-less further improves the overall performance by 25%, compared to DRAM-less (firmware). The DRAM-less performance is better than that of Hetero and Heterodirect by 93% and 47%, respectively. Even compared to Heterodirect-PRAM and PAGE-buffer with their best execution scenario, our DRAM-less exhibits around 64% better average bandwidth. We observed that the benefits brought by DRAM-less increase for the memory-intensive workloads. For example, DRAM-less performance is on average 149% better than PAGE-buffer for *durbin*, *dynpro*, *jacob1D*, and *regd*.

B. Execution Time Analysis

Figure 16 decomposes the execution time of all the accelerators that we tested. Specifically, Heterodirect can shorten the execution time as high as 16%, compared to Hetero, since it addresses the data transfer overheads imposed by host-side context switching and redundant data copies. Note that this phenomenon is also observed in Hetero-PRAM and Heterodirect-PRAM, but its long delay imposed by block-sized writes makes them worse than heterogeneous systems that employ the flash-based SSD by 5%, on average. Integrated-SLC/MLC/TLC remove the overhead imposed by data transfer via PCIe; however, these accelerators consume more cycles on flash accesses than computation by 78%, on average. This is because the internal access patterns of Integrated-SLC/MLC/TLC are not optimized by a buffer cache or file system, and still need to access the flash in a page granularity. Even though NOR-intf allows PEs to access memory with a byte granularity, its 16-bit serialized I/O operations and poor memory bandwidth in turn make the time cost of data processing longer than Integrated-SLC by 10%, on average. In contrast, the average execution time, consumed by PAGE-buffer, is 35% shorter than that of Integrated-SLC. This is because, a single page access cannot reap the benefit of flash-level from its internal parallelism, while multiple PRAM chips serve single page access together, which can enhance the performance. DRAM-less offers shorter storage latency than Integrated-SLC by 51%, on average. This is because DRAM-less fetches data with finer granularity and directly brings the data to each agent's cache through three-phase addressing. Even for write-intensive workloads (*chol, doitg, lu* and *seidel*) than others, DRAM-less reduces the overheads imposed by data movements by around 42%, compared to Integrated-SLC.

C. Energy Efficiency Study

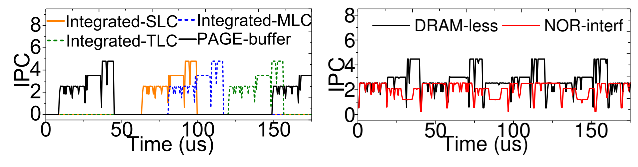
Figure 17 shows the energy decomposition for all the data processing activities of our 10 hardware accelerators. Although the performance improvement of DRAM-less is not significant (cf. Figure 15) for computation-intensive workloads (*adi, fdtdap, floyd*, *lu*, etc.), it saves great energy in processing the same amount of data than other accelerators. Hetero and Hetero-PRAM spend most of its energy on moving data within the host-side storage stack, whereas DRAM-less removes them, and can perform data processing near PRAMs with around 76% less energy, as even compared to PAGE-buffer. Note that, for write-intensive workloads, Hetero-PRAM and Heterodirect-PRAM waste energy on storing the outputs to PRAM SSDs by serializing all page-basis requests into byte-granular operations.

PAGE-buffer and Integrated-SLC/MLC/TLC dissipate energy on internal DRAM buffer for read-intensive workloads. Unlike other workloads whose data request sizes (input/output) are mostly greater than multiple pages, read-intensive workloads store a few bytes at the end of each kernel iteration, which wastes a lot of memory spaces within a page, which introduces DRAM pollution. In contrast, since all of the DRAM-less's agents can process data residing on the storage over normal store/load instructions, instead of using a file system or an OS API, it not only can eliminate the host energy overheads but also efficiently access data within the accelerator. In addition, with byte-addressability, DRAM-less can fetch data that is smaller than data that flash can fetch. Note that the long latency of NOR-intf's parallel PRAM makes its energy efficiency even worse than others (except for read-intensive workloads).

D. Time Series Analysis

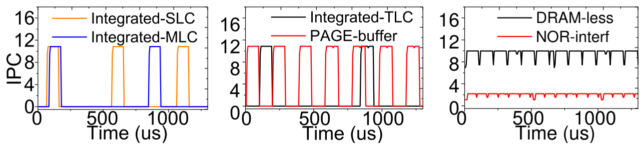
Total IPC. Figures 18 and 19 show the time series analysis of total IPC, which is aggregated by all agent PEs, under execution of a read-intensive workload (*gemver*) and a write-intensive workload (*doitg*), respectively. One can observe from the read-intensive evalua-

Authorized licensed use limited to: Xiamen University. Downloaded on July 22,2021 at 07:14:30 UTC from IEEE Xplore. Restrictions apply.



(a) Page-based accesses. (b) Byte-based acceses.

Figure 18: IPC for *gemver* (read-intensive).

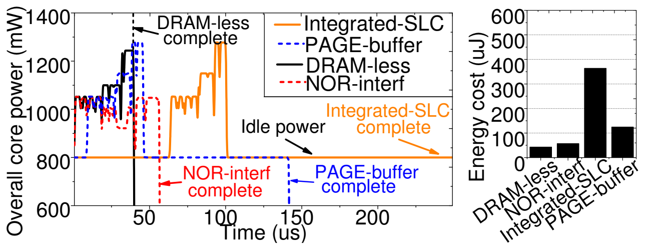


(a) (b) (c)

Figure 19: IPC for *doitg* (write-intensive).

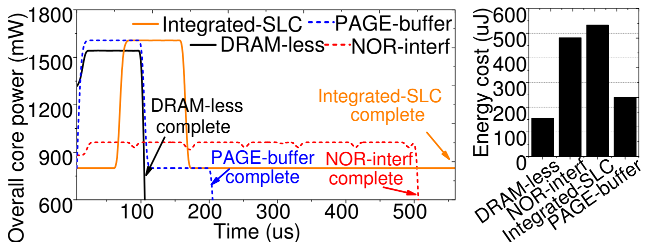
tion, `Integrated-SLC/MLC/TLC` and `PAGE-buffer` require fetching a whole page from storage to DRAM, which makes all corresponding PEs idle (zero value in Figure 18). In contrast, since `DRAM-less` and `NOR-interf` directly load and store data to/from storage, they allow PEs to keep processing data without introducing such idle. Thus, IPC of `DRAM-less` and `NOR-interf` is even better than `PAGE-buffer` by 292% and 206%, respectively, and sustainably offers 2 IPC during the executions. The shorter latency brought by multi-resource aware interleaving and new PRAM technology improves IPC and exhibit better performance than `NOR-interf` by 42%. This phenomenon becomes more significant under the write-intensive workload (Figure 19). For `Integrated-SLC/ MLC/ TLC`, the enforced idle period (i.e., stall) due to storage accesses is longer than the idles observed by *gemver* by 14.6x on average. While `NOR-interf` has no such stall, the total IPC of agents *doitg* severely degrades compared to `DRAM-less` (78% worse than that of `DRAM-less`). This is because, even though `NOR-interf` has no data pollution observed by the page-granule access based accelerators, its legacy read and write are slower than our new PRAM by 3x and 10x, respectively. The total IPC of `DRAM-less` is better than that of `integrated-SLC/MLC/TLC` and `PAGE-buff` by 5.1x, 10.3x, 15x, and 1.9x, respectively.

Power consumption and energy cost. To dig deeper power and total energy costs, we capture first 16KB data processing, and their results are illustrated by Figures 20 and 21 for the read-intensive and write-intensive workloads,



(a) PE power consumption. (b) PE energy.

Figure 20: Overall core power and total energy evaluation example in *gemver* (read-intensive).



(a) PE power consumption. (b) PE energy.

Figure 21: Overall core power and total energy evaluation example of *doitg* (write-intensive).

respectively. One can see from Figure 20, `NOR-interf` consumes, on average, 14% lower agent PE power compared to other approaches. This can explain the reason why data processing bandwidth of `NOR-interf` is often worse than other accelerators. Unlike `PAGE-buff` and/or `Integrated-SLC/MLC/TLC`, `NOR-interf` access the underlying storage over byte granularity without any help of DRAM. Thus, .D (load/store) unit is stalled and makes the most of functional units (.L, .S, and .M) idle. This low utilization issues enable `NOR-interf` to process data with low power (even lower than that of `DRAM-less`), but as increasing latency, it consumes energy higher than `DRAM-less` by 32% on average (Figure 20b). Note that, even though `Integrated-SLC` and `PAGE-buffer` exhibit the actual time and overall core power for data processing similar to those of `DRAM-less`, their actual completion time is delayed and suffers from the long latency of data transfers, which increases their energy costs by 7 times and 1.9 times compared to those of `DRAM-less`. Aforementioned characteristics become more significant and severe under the write-intensive workload. As shown in Figure 21a, `NOR-interf` takes 4x longer execution time than `PAGE-buffer` to complete a same task, and `DRAM-less` outperforms all other options with 50% ∼ 88% shorter completion time, on average.

VII. Related Work and Discussion

Data movement optimizations. Several prior studies have been proposed to minimize the overheads, imposed by data transfers; multiple solutions [58], [13], [14] apply a peer-to-peer DMA to directly forward I/O requests between an accelerator and SSD at a system level. Even though these approaches are powerful to reduce unnecessary software interventions and data copies within host-side DRAMs, they still require moving data around multiple physical boundaries. They also need extra efforts for modifying host drivers and file systems. [59] proposed a FPGA-based DMA engine to enable an inter-device communication without a help from host-side software stack. While such approach eliminates the software intervention at some extent, it unfortunately costs PCIe transfer delays and introduces the communication overheads between the DMA engine and PCIe endpoint

Authorized licensed use limited to: Xiamen University. Downloaded on July 22,2021 at 07:14:30 UTC from IEEE Xplore. Restrictions apply.

devices. In contrast, DRAM-less tightly integrates our new byte-addressable PRAMs into a multi-core accelerator, which allows data processing tasks to load/store data without a host-side OS help or any source-level modifications.

Near-data processing. Active SSDs aim to reduce data movement overheads by leveraging the existing SSD controller(s) [54], [53], [55], [12]. However, active SSDs face several challenges regarding the computation flexibility and capabilities. For instance, their user scenarios and applications are strongly limited by APIs that flash firmware needs to offer, which are fixed at a design time. [53] provides more flexible interfaces using C++11, but unfortunately, application offloading is available for only a "single" core and has no computing parallelism near storage. All these approaches are also strictly linked with their firmware and require a large size of internal DRAM buffers. In contrast, our DRAM-less automates all PRAM services over hardware and employs multiple low-power PEs, which can flexibly execute general-purpose applications with a massive parallel computing power.

PRAM integration in computing system. [60] checked the feasibility of replacing a main memory with PRAMs and revealed that multiple row buffers can reduce PRAM latency and energy, compared to a single row buffer mechanism, by around 45% and 69%, respectively. [61], [62], [63] proposed to reduce PRAM's write penalties by combining DRAM and PRAM as a hybrid memory; the hybrid memory leverages the fast DRAM banks to serve the hot data in front of PRAM banks. All these studies adopted an "ideal" PRAM latency model, assuming that PRAM exhibits a read/write latency, which is the same with or similar to DRAM latency. Unfortunately, such assumption is far away from the characteristics of real PRAMs, whose read and write latencies are 100ns and 10us, respectively. Considering the practical performance of PRAMs, we believe it is yet difficult and unable to directly replace the host-side DRAMs with PRAMs. Instead, DRAM-less integrates PRAMs in an accelerator to remove costly data movement overheads and design an efficient near data processing model.

Real PRAM prototype. Recently, industry has announced a real persistent memory prototype, called as Optane DC persistent memory (PMM) [64]. While such PRAM prototype can be seamlessly integrated in host-side environment, it may not fit for the accelerator. Specifically, Optane DC PMM needs the support of filesystem and drivers to guarantee data persistency, which unfortunately is unavailable in accelerator. In addition, many accelerators adopt customized communication interfaces [19], which are incompatible with that of Optane DC PMM. Note that the internal architecture details of Optane DC PMM are still not open to the public. In contrast, we revealed the design details of the real PRAM modules and the corresponding memory controller in this work. We also discussed on the challenges and solutions of integrating PRAM subsystem in the hardware accelerator.

PRAM optimization. Multiple approaches were proposed to improve the performance of PRAM [65], [66]. Specifically, [65] improves the PRAM write performance and extends its endurance by compressing the target data. To prevent the long PRAM writes from blocking the following read operations, [66] proposed to cancel or pause the write operations being in progress. Unfortunately, none of these work considered the architectural characteristics of real PRAMs. While [67] can leverage the multi-partition of the modern PRAMs to parallelize incoming memory requests, such approach is sub-optimal, as read-only streams cannot benefit from the low-level memory parallelism. In contrast, our memory controller modified the timing diagram of the three-phase addressing to interleave incoming memory requests by being aware of PRAM's multiple partitions and row buffers. It can fully overlap the data transfer time with the latency to access the memory partition, which can improve low-level memory parallelism. Our controller also reduces the PRAM write latency by selectively erasing the stale data in advance.

PRAM lifetime. DRAM-less addresses PRAM write endurance issue in two aspects. Specifically, the layout of PRAM cells are carefully organized in our PRAM sample to eliminate SET and RESET disturbances, thereby significantly improving endurance. In addition, DRAM-less can integrate traditional wear levellers in our PRAM controller, such as start-gap [68], to improve the PRAM lifetime.

VIII. Conclusion

The existing accelerated systems waste many CPU cycles and powers in transferring data across different logic barriers and physical boundaries. To address this, we proposed DRAM-less that integrates new multi-partition PRAMs into a multi-core based accelerator. DRAM-less enables multiple processing elements to directly access the data from its internal PRAM and automates the corresponding I/O path over hardware. DRAM-less in turn can remove the overheads imposed by the host-side interventions and costly data movement. We also proposed a new PRAM controller that addresses PRAM's write penalties and exhibits a high degree of PRAM internal parallelism by being aware of PRAM micro-architecture (multi-partition). Our evaluation results reveal that DRAM-less, on average, achieves 47% and 80% higher performance than advanced accelerated systems and active storage approaches, respectively.

IX. Acknowledgement

The authors thank Dr. Hung-Wei Tseng for shepherding their paper. This paper is a full-length version of an earlier 4-page letter [69]. This research is mainly supported by NRF 2016R1C182015312, MemRay grant (G01190170), DOE DEAC02-05CH11231, NRF 2015M3C4A7065645, and KAIST start-up package (G01190015). J. Zhang and G. Park equally contribute to this work. Myoungsoo Jung is the corresponding author.

Authorized licensed use limited to: Xiamen University. Downloaded on July 22,2021 at 07:14:30 UTC from IEEE Xplore. Restrictions apply.

References

[1] R. M. Wallace, B. Vacaliuc, D. A. Clayton, B. R. Chaffins, O. O. Storaasli, D. Strenski, and D. Poznanovic, "Consideration of the tms320c6678 multi-core dsp for power efficient high performance computing," tech. rep., Technical Report ORNL-Pub-28647, Oak Ridge National Laboratory, Feb 2011. Original URL was http://info. ornl. gov/sites/publications/Files/Pub28647. pdf. https://www. dropbox.com/s/56t9z9lav5ir45f/ORNL-Pub28647.pdf.

[2] G. Hegde, N. Ramasamy, N. Kapre, *et al.*, "Caffepresso: an optimized library for deep learning on embedded accelerator-based platforms," in *Proceedings of the International Conference on Compilers, Architectures and Synthesis for Embedded Systems*, p. 14, ACM, 2016.

[3] J. Murphy, "Deep Learning Benchmarks of NVIDIA Tesla P100 PCIe, Tesla K80, Tesla M40 GPUs https://www.microway.com/hpc-tech-tips/deep-learning-benchmarks-nvidia-tesla-p100-16gb-pcie-tesla-k80-tesla-m40-gpus," 2017.

[4] R. Solcà, A. Kozhevnikov, A. Haidar, S. Tomov, J. Dongarra, and T. C. Schulthess, "Efficient implementation of quantum materials simulations on distributed cpu-gpu systems," in *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, p. 10, ACM, 2015.

[5] Y. Wang, M. J. Anderson, J. D. Cohen, A. Heinecke, K. Li, N. Satish, N. Sundaram, N. B. Turk-Browne, and T. L. Willke, "Full correlation matrix analysis of fmri data on intel® xeon phi coprocessors," in *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, p. 23, ACM, 2015.

[6] I. Yamazaki, J. Kurzak, P. Luszczek, and J. Dongarra, "Randomized algorithms to update partial singular value decomposition on a hybrid cpu/gpu cluster," in *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, p. 59, ACM, 2015.

[7] A. M. Caulfield, E. S. Chung, A. Putnam, H. Angepat, J. Fowers, M. Haselman, S. Heil, M. Humphrey, P. Kaur, J.-Y. Kim, *et al.*, "A cloud-scale acceleration architecture," in *Microarchitecture (MICRO), 2016 49th Annual IEEE/ACM International Symposium on*, pp. 1–13, IEEE, 2016.

[8] V. Gokhale, J. Jin, A. Dundar, B. Martini, and E. Culurciello, "A 240 g-ops/s mobile coprocessor for deep neural networks," in *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition Workshops*, pp. 682–687, 2014.

[9] P. Russom *et al.*, "Big data analytics," *TDWI best practices report, fourth quarter*, vol. 19, no. 4, pp. 1–34, 2011.

[10] Q. Zhang, L. Cheng, and R. Boutaba, "Cloud computing: state-of-the-art and research challenges," *Journal of internet services and applications*, vol. 1, no. 1, pp. 7–18, 2010.

[11] V. Gulisano, R. Jimenez-Peris, M. Patino-Martinez, C. Soriente, and P. Valduriez, "Streamcloud: An elastic and scalable data streaming system," *IEEE Transactions on Parallel and Distributed Systems*, vol. 23, no. 12, pp. 2351–2365, 2012.

[12] S. Seshadri, M. Gahagan, M. S. Bhaskaran, T. Bunker, A. De, Y. Jin, Y. Liu, and S. Swanson, "Willow: A user-programmable ssd.," in *OSDI*, pp. 67–80, 2014.

[13] H.-W. Tseng, Q. Zhao, Y. Zhou, M. Gahagan, and S. Swanson, "Morpheus: creating application objects efficiently for heterogeneous computing," in *Computer Architecture (ISCA), 2016 ACM/IEEE 43rd Annual International Symposium on*, pp. 53–65, IEEE, 2016.

[14] J. Zhang, D. Donofrio, J. Shalf, M. T. Kandemir, and M. Jung, "Nvmmu: A non-volatile memory management unit for heterogeneous gpu-ssd architectures," in *Parallel Architecture and Compilation (PACT), 2015 International Conference on*, pp. 13–24, IEEE, 2015.

[15] T. Anderson, D. Bui, S. Moharil, S. Narnur, M. Rahman, A. Lell, E. Biscondi, A. Shrivastava, P. Dent, M. Yan, *et al.*, "A 1.5 ghz vliw dsp cpu with integrated floating point and fixed point instructions in 40 nm cmos," in *Computer Arithmetic (ARITH), 2011 20th IEEE Symposium on*, pp. 82–86, IEEE, 2011.

[16] Intel, "Intel ssd 750 series," *http://www.intel.com/content/www/us/en/solid-state-drives/solid-state-drives-750-series.html*, 2015.

[17] J. Zhang and M. Jung, "Flashabacus: a self-governing flash-based accelerator for low-power systems," in *Proceedings of the Thirteenth EuroSys Conference*, p. 15, ACM, 2018.

[18] L.-N. Pouchet, "Polybench: the polyhedral benchmark suite," *http://www.cs.ucla.edu/~{}pouchet/software/polybench/*, 2012.

[19] Texas-Instruments, "Tms320c6678 multicore fixed and floating-point digital signal processor," 2014.

[20] Micron, "Mt29f2g08aabwp/mt29f2g16aabwp nand flash datasheet," 2004.

[21] Micron, "Mt29f64g08cfabb/mt29f64g08cecbb nand flash datasheet," 2008.

[22] Micron, "Mt29f64g08cbaaa/mt29f64g08cbaab nand flash datasheet," 2009.

[23] B. Tallis, "The intel optane ssd dc p4800x (375gb) review: Testing 3d xpoint performance," 2017.

[24] JESD209-2B, "Jedec standard: Low power double data rate 2 (lpddr2)," in *JEDEC SSTA*, 2010.

[25] Xilinx, "Xilinx zynq-7000 all programmable soc zc706 evaluation kit." https://www.xilinx.com/support/documentation/boards_and_kits/zc706/ug954-zc706-eval-board-xc7z045-ap-soc.pdf, 2018.

[26] B. Barker, "Message passing interface (mpi)," in *Workshop: High Performance Computing on Stampede*, vol. 262, 2015.

[27] S. Lai, "Current status of the phase change memory and its future," in *Electron Devices Meeting, 2003. IEDM'03 Technical Digest. IEEE International*, pp. 10–1, IEEE, 2003.

Authorized licensed use limited to: Xiamen University. Downloaded on July 22,2021 at 07:14:30 UTC from IEEE Xplore. Restrictions apply.

[28] V. Young, P. J. Nair, and M. K. Qureshi, "Deuce: Write-efficient encryption for non-volatile memories," in *ACM SIGARCH Computer Architecture News*, vol. 43, pp. 33–44, ACM, 2015.

[29] M. K. Qureshi, M. M. Franceschini, A. Jagmohan, and L. A. Lastras, "Preset: improving performance of phase change memories by exploiting asymmetry in write times," *ACM SIGARCH Computer Architecture News*, vol. 40, no. 3, pp. 380–391, 2012.

[30] H.-S. P. Wong, S. Raoux, S. Kim, J. Liang, J. P. Reifenberg, B. Rajendran, M. Asheghi, and K. E. Goodson, "Phase change memory," *Proceedings of the IEEE*, vol. 98, no. 12, pp. 2201–2227, 2010.

[31] H. Horii, J. Yi, J. Park, Y. Ha, I. Baek, S. Park, Y. Hwang, S. Lee, Y. Kim, K. Lee, *et al.*, "A novel cell technology using n-doped gesbte films for phase change ram," in *VLSI Technology, 2003. Digest of Technical Papers. 2003 Symposium on*, pp. 177–178, IEEE, 2003.

[32] S. Hudgens and B. Johnson, "Overview of phase-change chalcogenide nonvolatile memory technology," *MRS bulletin*, vol. 29, no. 11, pp. 829–832, 2004.

[33] X. Dong and Y. Xie, "Adams: Adaptive mlc/slc phase-change memory design for file storage," in *Design Automation Conference (ASP-DAC), 2011 16th Asia and South Pacific*, pp. 31–36, IEEE, 2011.

[34] Y. Joo, D. Niu, X. Dong, G. Sun, N. Chang, and Y. Xie, "Energy-and endurance-aware design of phase change memory caches," in *Design, Automation & Test in Europe Conference & Exhibition (DATE), 2010*, pp. 136–141, IEEE, 2010.

[35] P. J. Nair, C. Chou, B. Rajendran, and M. K. Qureshi, "Reducing read latency of phase change memory via early read and turbo read," in *High Performance Computer Architecture (HPCA), 2015 IEEE 21st International Symposium on*, pp. 309–319, IEEE, 2015.

[36] J. Wang, X. Dong, G. Sun, D. Niu, and Y. Xie, "Energy-efficient multi-level cell phase-change memory system with data encoding," in *Computer Design (ICCD), 2011 IEEE 29th International Conference on*, pp. 175–182, IEEE, 2011.

[37] S. Liu, A. Kolli, J. Ren, and S. Khan, "Crash consistency in encrypted non-volatile main memory systems," in *2018 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, pp. 310–323, IEEE, 2018.

[38] Y. Cassuto, S. Kvatinsky, and E. Yaakobi, "Sneak-path constraints in memristor crossbar arrays," in *Information Theory Proceedings (ISIT), 2013 IEEE International Symposium on*, pp. 156–160, IEEE, 2013.

[39] C. Xu, D. Niu, N. Muralimanohar, R. Balasubramonian, T. Zhang, S. Yu, and Y. Xie, "Overcoming the challenges of crossbar resistive memory architectures," in *High Performance Computer Architecture (HPCA), 2015 IEEE 21st International Symposium on*, pp. 476–488, IEEE, 2015.

[40] D. Niu, C. Xu, N. Muralimanohar, N. P. Jouppi, and Y. Xie, "Design trade-offs for high density cross-point resistive memory," in *Proceedings of the 2012 ACM/IEEE international symposium on Low power electronics and design*, pp. 209–214, ACM, 2012.

[41] S. Kang *et al.*, "A 0.1 um 1.8-v 256-mb phase-change random access memory (pram) with 66-mhz synchronous burst-read operation," *IEEE Journal of Solid-State Circuits*, 2007.

[42] Y.-C. Bae, J.-Y. Park, S. J. Rhee, S. B. Ko, Y. Jeong, K.-S. Noh, Y. Son, J. Youn, Y. Chu, H. Cho, *et al.*, "A 1.2 v 30nm 1.6 gb/s/pin 4gb lpddr3 sdram with input skew calibration and enhanced control scheme," in *Solid-State Circuits Conference Digest of Technical Papers (ISSCC), 2012 IEEE International*, pp. 44–46, IEEE, 2012.

[43] Numonyx, "Omneo, p8p pcm, 128-mbit parallel phase change memory datasheet," 2010.

[44] T. Instruments, "Tms320c66x dsp corepac user guide," 2011.

[45] N. Otterness, M. Yang, S. Rust, E. Park, J. H. Anderson, F. D. Smith, A. Berg, and S. Wang, "An evaluation of the nvidia tx1 for supporting real-time computer-vision workloads," in *Real-Time and Embedded Technology and Applications Symposium (RTAS), 2017 IEEE*, pp. 353–364, IEEE, 2017.

[46] Parallella, "Epiphany-iii 16-core microprocessor." http://www.adapteva.com/epiphanyiii/, 2013.

[47] TI, "Ti code generation tools," *http://processors.wiki.ti.com/index.php/Category:Compiler*, 2014.

[48] TI, "Multicore software development kit," *https://training.ti.com/multicore-software-development-kit-mcsdk-keystone-devices*, 2011.

[49] TI, "Multicore application deployment (mad) utilities," *http://processors.wiki.ti.com/index.php/MAD-Utils-User-Guide*, 2014.

[50] V. Cuppu, B. Jacob, B. Davis, and T. Mudge, "A performance comparison of contemporary dram architectures," in *ACM SIGARCH Computer Architecture News*, vol. 27, pp. 222–233, IEEE Computer Society, 1999.

[51] J. Wallen, "How to change the linux i/o scheduler to fit your needs," *https://www.techrepublic.com/article/how-to-change-the-linux-io-scheduler-to-fit-your-needs/*, 2017.

[52] S. Bergman, T. Brokhman, T. Cohen, and M. Silberstein, "Spin: Seamless operating system integration of peer-to-peer dma between ssds and gpus," in *USENIX ATC*, 2017.

[53] B. Gu, A. S. Yoon, D.-H. Bae, I. Jo, J. Lee, J. Yoon, J.-U. Kang, M. Kwon, C. Yoon, S. Cho, *et al.*, "Biscuit: A framework for near-data processing of big data workloads," in *Computer Architecture (ISCA), 2016 ACM/IEEE 43rd Annual International Symposium on*, pp. 153–165, IEEE, 2016.

[54] S. Cho, C. Park, H. Oh, S. Kim, Y. Yi, and G. R. Ganger, "Active disk meets flash: a case for intelligent ssds," in *Proceedings of the 27th international ACM conference on International conference on supercomputing*, pp. 91–102, ACM, 2013.

Authorized licensed use limited to: Xiamen University. Downloaded on July 22,2021 at 07:14:30 UTC from IEEE Xplore. Restrictions apply.

[55] D. Tiwari, S. Boboila, S. S. Vazhkudai, Y. Kim, X. Ma, P. Desnoyers, and Y. Solihin, “Active flash: towards energy-efficient, in-situ data analytics on extreme-scale machines.,” in *FAST*, pp. 119–132, 2013.

[56] A. De, M. Gokhale, R. Gupta, and S. Swanson, “Minerva: Accelerating data analysis in next-generation ssds,” in *Field-Programmable Custom Computing Machines (FCCM), 2013 IEEE 21st Annual International Symposium on*, pp. 9–16, IEEE, 2013.

[57] Marvell, “High performance pcie ssd controllers,” *https://www.marvell.com/storage/ssd/88ss1092-93/*, 2017.

[58] A. M. Caulfield, L. M. Grupp, and S. Swanson, “Gordon: An improved architecture for data-intensive applications,” *IEEE micro*, vol. 30, no. 1, 2010.

[59] J. Ahn, D. Kwon, Y. Kim, M. Ajdari, J. Lee, and J. Kim, “Dcs: a fast and scalable device-centric server architecture,” in *Proceedings of the 48th International Symposium on Microarchitecture*, pp. 559–571, ACM, 2015.

[60] B. C. Lee, E. Ipek, O. Mutlu, and D. Burger, “Architecting phase change memory as a scalable dram alternative,” in *ACM SIGARCH Computer Architecture News*, vol. 37, pp. 2–13, ACM, 2009.

[61] M. K. Qureshi, V. Srinivasan, and J. A. Rivers, “Scalable high performance main memory system using phase-change memory technology,” in *ACM SIGARCH Computer Architecture News*, vol. 37, pp. 24–33, ACM, 2009.

[62] G. Dhiman, R. Ayoub, and T. Rosing, “Pdram: A hybrid pram and dram main memory system,” in *2009 46th ACM/IEEE Design Automation Conference*, pp. 664–669, IEEE, 2009.

[63] L. E. Ramos, E. Gorbatov, and R. Bianchini, “Page placement in hybrid memory systems,” in *Proceedings of the international conference on Supercomputing*, pp. 85–95, ACM, 2011.

[64] J. Izraelevitz, J. Yang, L. Zhang, J. Kim, X. Liu, A. Memaripour, Y. J. Soh, Z. Wang, Y. Xu, S. R. Dulloor, *et al.*, “Basic performance measurements of the intel optane dc persistent memory module,” *arXiv preprint arXiv:1903.05714*, 2019.

[65] G. Sun, D. Niu, J. Ouyang, and Y. Xie, “A frequent-value based pram memory architecture,” in *16th Asia and South Pacific Design Automation Conference (ASP-DAC 2011)*, pp. 211–216, IEEE, 2011.

[66] M. K. Qureshi, M. M. Franceschini, and L. A. Lastras-Montano, “Improving read performance of phase change memories via write cancellation and write pausing,” in *HPCA-16 2010 The Sixteenth International Symposium on High-Performance Computer Architecture*, pp. 1–11, IEEE, 2010.

[67] W. Zhou, D. Feng, Y. Hua, J. Liu, F. Huang, and Y. Chen, “An efficient parallel scheduling scheme on multi-partition pcm architecture,” in *Proceedings of the 2016 International Symposium on Low Power Electronics and Design*, pp. 344–349, ACM, 2016.

[68] M. K. Qureshi, J. Karidis, M. Franceschini, V. Srinivasan, L. Lastras, and B. Abali, “Enhancing lifetime and security of pcm-based main memory with start-gap wear leveling,” in *Proceedings of the 42nd annual IEEE/ACM international symposium on microarchitecture*, pp. 14–23, ACM, 2009.

[69] M. Jung, “Nearzero: An integration of phase change memory with multi-core coprocessor,” *IEEE Computer Architecture Letters*, vol. 16, no. 2, pp. 136–140, 2017.

Authorized licensed use limited to: Xiamen University. Downloaded on July 22,2021 at 07:14:30 UTC from IEEE Xplore. Restrictions apply.