Using Logging-on-Write to Improve Non-Volatile Memory Checkpoints via Processing-in-Memory

Kleber Kruger
Institute of Computing
State University of Campinas
Campinas, Brazil
kleber.kruger@ufms.br

Ricardo Pannain
Institute of Computing
State University of Campinas
Campinas, Brazil
pannain@ic.unicamp.br

Rodolfo Azevedo
Institute of Computing
State University of Campinas
Campinas, Brazil
rodolfo@ic.unicamp.br

***Abstract*—NVM architectures must keep consistent data in case of failures, a property called crash consistency. A common way to do so is by checkpoint mechanisms. However, most of the strategies developed have performance and usability problems. Among main limitations are non-software-transparent strategies, the addition of logging operations in the critical execution path, and increased writes to NVM, resulting in significant bandwidth usage between processor and memory. DONUTS solves these problems by a hardware mechanism that provides crash consistency via checkpoints integrated into cache replacement policy using Processing-in-Memory to perform logging operations. Its approach reduces writes from the processor's memory controller and the NVM external bandwidth usage but generates unnecessary log entries. This paper expands DONUTS to a multi-core scenario evaluating two processing-in-memory strategies. The first logs during read operations, and the second uses a new lazy strategy to log data exclusively on write operations when these operations are effectively needed. Finally, we compare runtime performance, log rate, energy consumption, and memory space. Results show that our new logging-on-write strategy maintained the DONUTS runtime performance but reduced energy consumption in multi-core applications by around 33% due to an average reduction of 42% in log operations. Also, the new strategy generates a checkpoint size 5x smaller than the previous system, maximizing the use of NVM. Compared to other systems, DONUTS presented an average overhead of 1% to 3% against up to 7% of previous software-transparent better-performing projects.**

***Index Terms*—processing-in-memory, in-memory computing, non-volatile memory, checkpointing, crash consistency.**

I. Introduction

Non-Volatile Memory (NVM) is a technology emerging as an alternative to DRAM [1]. It provides fast access latency, lower power consumption, and higher storage density than current technology. Hence, several studies [2]–[4] have explored its use as a new component in memory and storage hierarchy, replacing or using it in addition to DRAM. With NVM, applications can access persistent data directly in main memory by load/store instructions, avoiding data serialization/deserialization and paging in/out slower storage devices [5]. Examples of non-volatile memory include Phase Change Memory (PCM) [6], Spin-Torque Transfer RAM (STT-RAM) [7], Resistive RAM (ReRAM) [8], and 3D XPoint [9].

Despite advantages, NVM architectures need to ensure data consistency in case of failures, a property known as crash consistency [10]. An example of failure occurs when an atomic update to a data structure modifies two cache lines. If the system crashes after exclusively one cache line reaches persistent main memory, the data structure turns inconsistent because of the partial update in NVM [11]. This behavior is not a problem on volatile memory systems due the main memory is reset upon a system restart. However, the inconsistent data state remains on NVM-based systems even after a restart. Therefore, NVM-based architectures must ensure recovery to a consistent version, also called a checkpoint [12]. Thus, these systems need to perform persistent-data updates by atomic transactions in NVM, i.e., all writes in an update are successfully committed, or none performs. This property is also named atomic durability [13].

Projecting crash consistency mechanisms has been a challenge for NVM architectures since most strategies can lead to three problems: 1) non-software-transparent approaches, which restrict the NVM usage to applications based on transac-

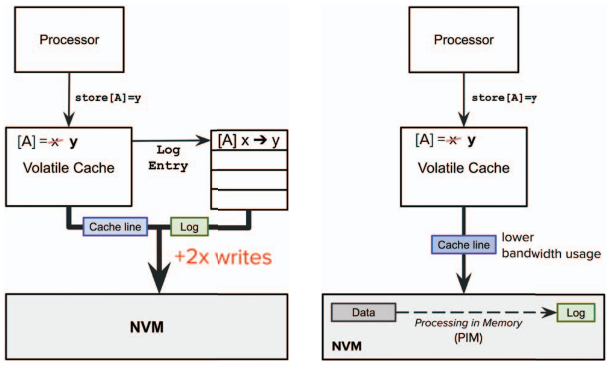
```
void updateNode(int old_value, int new_value)
{
  NVHeap *nv = NVHOpen("foo.nvheap");
  NVList::VPtr a = nv->GetRoot<NVList::NVPtr>();
  AtomicBegin {
    while(a != nullptr) {
      if(a->get_value() == old_value)
        a->set_value(new_value);
      a = a->get_next();
    }
  } AtomicEnd;
}
```

(a) Software-based

```
void updateNode(int old_value, int new_value)
{
    struct List *a = root_pointer;
    while(a != nullptr) {
        if(a->value == old_value)
            a->value = new_value;
        a = a->next;
    }
}
```

(b) Software-transparent

Fig. 1: Examples of software-based and software-transparent implementations. The second does not require changes to the application's source code.



(a) Other mechanisms (b) DONUTS

Fig. 2: Traditional crash consistency mechanisms compared to DONUTS logging scheme via Processing-in-Memory.

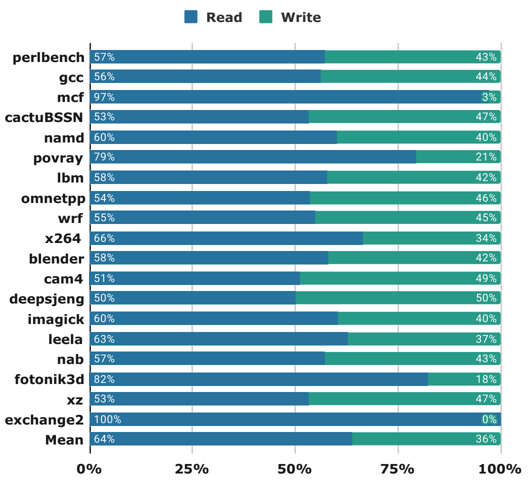


Fig. 3: Read/write rates of SPEC CPU2017 applications.

tional memory models. As shown in Figure 1, software-based approaches need programmers to rewrite the application's source code to adapt to a new programming model, making it a costly and error-prone task [3]; 2) some mechanisms [11], [13], [14] generate persistence operations on the critical execution path (e.g., via software write-back instructions, such as `clflush`, `clwb`, e `dccvap`), degrading performance; and 3) due to logging operations, crash consistency mechanisms tend significantly to increase the number of writes to NVM, as shown Figure 2a. So, in addition to conventional writes from an application, also it is necessary to write recovery data in specific NVM regions. These logging operations executed by the memory controller increase the bandwidth usage on the bus between processor and memory, causing performance degradation in high-demand memory access applications. DONUTS [15] solves this problem using Processing-in-Memory[1] (PIM) to perform logging operations in the background, reducing data movement between processor and memory (Figure 2b), and consequently, the NVM's external bandwidth usage. However, its logging-on-read strategy generates excessive entries since some data loaded from memory is not modified during a specific part of the program, generating unnecessary logging operations and increasing the checkpoint memory space.

This work expands DONUTS [15] by implementing a new PIM logging strategy (LoW) capable of performing exclusively on modified data, reducing the number of unnecessary logs generated by the original method (we named it LoR). Furthermore, our strategy presents a different behavior than LoR since it delays log writes to the commit phase on an epoch, unlike LoR, which performs logging during the epoch execution. Figure 3 shows the read/write rate of SPEC CPU 2017 applications observing that the amount of read access is almost twice the write access rate. We also note that our PIM implementation model is compatible with any resistance-based NVM technology by adopting the already established DDR standard in the industry. Results show that our proposed strategy reduced logging by 42%, decreasing 33% of energy consumption in multicore applications while maintaining the same DONUTS performance. Compared with another better-performing system, our proposal presented a runtime overhead of 1% to 3% against up to 7% of PiCL [2].

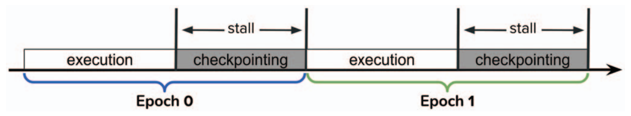
This paper is organized as follows: Section II explains background concepts, such as terms about crash consistency mechanisms, logging techniques, and PIM definitions. Section III describes PIM logging strategies and its implementation in the DONUTS mechanism, with their evaluation presented in Section IV. Finally, Section V shows related works and Section VI concludes this paper.

II. Background

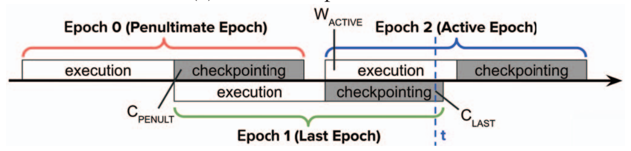
A. Crash Consistency

The principle of crash consistency techniques for NVM systems is periodically saving consistent memory data (plus CPU state) to have a checkpoint when a crash occurs. A checkpoint keeps a consistent memory snapshot, and an epoch is the time interval between two checkpoints [12]. An epoch comprises three phases: execution, commit, and persistence. During execution, applications can read and write data in volatile caches by load/store operations. On ending an epoch, the system commits all modified (dirty) data, creating a consistent snapshot (but not durable yet) of the epoch in execution. This process allows the running epoch to finish, and a new one can start. However, the system must persist the committed data to NVM, turning it durable after the persistence phase. The epoch length depends on checkpointing mechanisms strategy, which usually uses a periodic time or interval of instructions. DONUTS establishes dynamic epochs by cache thresholds or timeout events.

[1]Processing-in-Memory (PIM) is also known in the literature by the terms: Near-Data Processing (NDP), Near-Memory Processing (NMP), Near-Memory Computing (NMC), or in the case of storage devices, In-Storage Processing (ISP).

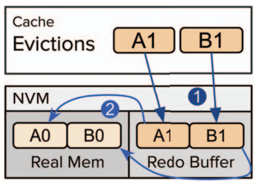


(a) Alternate epochs model

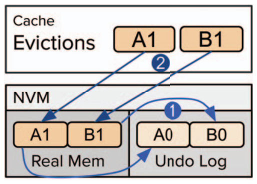


(b) ThyNVM overlapping epochs model

Fig. 4: Checkpointing scheme with (a) alternate phases and other with (b) phase overlapping.



(a) Redo Logging



(b) Undo Logging

Fig. 5: Redo-logging and undo-logging schemes.

Crash consistency systems need to overlap epochs to obtain viable performance. According to ThyNVM [12], switching between execution and checkpointing (commit + persistence) phases without overlapping can incur significant performance degradation, with checkpointing consuming up to 35.4% of all execution time in memory-intensive workload applications. This problem is even more costly in nowadays systems with large-capacity caches. Thus, instead of alternating between phases as shown in Figure 4a, modern crash consistency mechanisms [2]–[4] overlap them to remove persistence from the critical path and prevent stalls, mitigating performance degradation [16]. Figure 4b shows an example of ThyNVM. The active working copy $\mathtt{W}_{\mathtt{active}}$ corresponds to a region where the running data is updated. While executing the current epoch $\mathtt{W}_{\mathtt{active}}$, the previous $\mathtt{C}_{\mathtt{last}}$ is persisted to NVM. However, a crash during $\mathtt{C}_{\mathtt{last}}$ persistence could leave inconsistent checkpoint data. Therefore, ThyNVM maintains the penultimate checkpointing $\mathtt{C}_{\mathtt{penult}}$ until $\mathtt{C}_{\mathtt{last}}$ is finished. So, this scheme overlaps three epochs: while the current one (epoch 2) is running, the previous (epoch 1) is checkpointing, and the penultimate (epoch 0) corresponds to a safe epoch — i.e., an epoch recoverable in a system crash. In a fully asynchronous model, previous epochs can be committed and queued in order to persist to NVM during an epoch in execution.

B. *Write-Ahead Logging*

The most common strategy to ensure crash consistency in NVM systems is Write-ahead Logging (WAL), subdivided into undo-logging or redo-logging techniques. In a redo-logging scheme (Figure 5a), cache evictions are temporarily held in a redo-buffer to preserve main-memory consistency and later written to original memory address regions. Like a CPU write-buffer, the NVM redo-buffer is snooped on every memory access to avoid returning outdated data [2]. So, redo-logging requires additional mapping to find the latest data consistent version because it writes indirectly since the updated data is first stored at a redo-buffer region and later updated in its default locations. A second way is an undo-logging scheme (Figure 5b), that on a cache eviction, the system first reads undo-data (original data before modifications) from its default memory address. Then, it persists in an NVM undo-buffer and finally writes evictions locally to memory. This step is called the read-log-modify access sequence. For example, suppose a system crash or loses power. The system rollbacks these writes by applying undo-buffer entries to recover a valid checkpoint. In summary, redo-logging allows recovery to a consistent state redoing pending operations, while undo-logging recovers by returning incomplete updates to consistent state values.

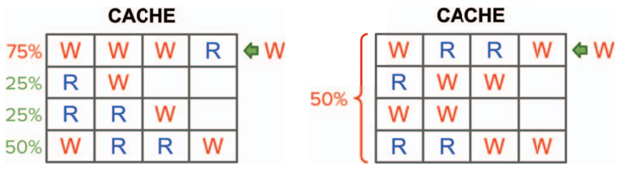
C. *Processing-in-Memory*

Processing-in-Memory allows memory devices to run simple processing operations, making these systems able to compute data in a cache, main memory, or local external storage. This strategy reduces the cost of moving data between processor and memory [17]. Meantime, building PIM architectures requires at least two challenges. First, programmers must identify which parts of an application are processable in memory, and architects must understand restrictions imposed by PIM logic design specifications. Second, after identifying PIM opportunities and designing their architectures, programmers need a way to extract PIM benefits without having to resort to complex programming models [18]. Examples where PIM is employed are: bulk copying and data initialization [19], [20], bitwise operations on bulk data [21]–[24] and simple arithmetic operations (e.g., addition, multiplication, implication) [20], [25]–[27].

In DONUTS [15], PIM is used for logging operations, avoiding commands, and data movement between the processor's memory controller and NVM. In traditional schemes, memory is passive, i.e., just obeying commands from the memory controller. For example, to perform a copy operation from A to B address, the memory controller sends signals to memory to activate rows and columns of the cell array at address A, moving data to row buffers. Subsequently, it sends new commands to activate rows and columns of the cell array at address B to copy the content. Noting that both data and the destination are already in the main memory, DONUTS performs undo-logging without the immediate need for the memory controller, taking advantage of the access command to trigger a PIM copy operation from traditional banks to specific log banks.

D. *DONUTS*

DONUTS [15] is based on overlapping epochs to allow an epoch in the persistence phase not to block another in execution, making them independent (decoupled) and without



(a) Associative set threshold (b) Cache capacity threshold

Fig. 6: DONUTS thresholds (75% for the associative set and 50% for the cache capacity). R denotes clean blocks and W dirty blocks. Both figures exemplify a situation where a new cache-block allocation triggers an epoch commit event.

TABLE I: Memory controller command list.

CMD	Description
NOP	ignores all inputs
ACT	activates a row in a particular bank
RD	initiates a read burst to an active row
WR	initiates a write burst to an active row
LRD	creates a new log entry and a read burst to an active row
LWR	creates a new log entry and a write burst to an active row
LOG	mounts the physical address metadata
PRE	closes a row in a particular bank
CKP	closes log row buffers and creates a checkpoint

the need for synchronization between phases. A commit event occurs under three conditions: a) on reaching an associative set threshold, b) on reaching the cache capacity threshold, and c) on reaching a specified timeout.

The epoch dynamic behavior results from the DONUTS cache replacement policy, based on a conventional Least Recently Used (LRU), with a slight modification so that cache block evictions victimize only clean blocks. Figure 6a shows a checkpoint example on case A, with an associative set threshold defined to 75%. When a set reaches the threshold, a commit event generates a checkpoint of the current epoch, persisting modified blocks in the background and releasing the system to start a new epoch (SystemID++). In order to track the epoch of data updates, DONUTS adds a field EpochID (EID) in each cache block. Another commit event example occurs when the cache reaches the capacity threshold (case b), defined in 50% as shown in Figure 6b. Although all associative sets did not exceed the defined threshold (75%), more than half of the cache blocks were dirty, causing a checkpoint. A third possible checkpoint event occurs when the cache exceeds a timeout between the current time and the last persisted checkpoint. This third case prevents the system from keeping lagged checkpoints.

III. Logging via Processing-in-Memory

DONUTS implements logging operations by slightly modifying NVM circuits and the DDRx protocol. In each memory module chip, one log bank is added in addition to conventional banks. These banks are visible only to checkpoint routines and act outside the application's address space. Table I shows a commands list sent by the memory controller to DIMM. First, the read/write command with early log (LRD/LWR) copy original data from conventional banks to on-chip log banks in the background before each read operation. Next, the LOG command creates a log entry composed of data plus its original physical address (metadata - formed by concatenating bits from the row and column sent during activation plus the read/write operation). Finally, CKP closes the log-banks row buffers and marks the current epoch checkpoint as finished.

Figure 7 details command sequence steps along with the latency times of each operation (represented by labeled arrows). Black circles show an observation point to highlight. At point 1, upon receiving an ACT command, the memory system activates a word line, allowing sense amplifiers to interpret the cell contents of the selected row and temporarily store it in a row buffer. This cost is a Row Address Strobe (tRAS) latency, being only necessary when the requested data is not in the already activated line. Once a line is active, the memory controller can send read/write commands (point 2), selecting a column from the opened line and sending the read value to the in/out buffer (case a read) or receiving from in/out buffer a value to be written to cell (case a write). This latency is called Column Access Strobe ($\texttt{tCAS}^{\texttt{data}}$).

Unlike RD and WR commands, LRD and LWR simultaneously log data, i.e., copy selected cell content to log buffer. Note that in this step, the system can build the data address since it stored row + column bits in a temporary register (shown at point 3) during the activation command and the LRD (or LWR) command, respectively. After this sequence, on a read operation, for instance, the NVM has the requested data in both output and log buffers. So, it is possible to send output data (point 4) while creating a log entry (joining data to metadata) and subsequently write it to the log bank in the background (point 5) after a LOG command. In order to optimize the logging procedure, the system keeps log bank row buffers open until their capacity limit or a CKP command is received. This sequence occurs for all memory accesses within an epoch until a commit event occurs (shown at point 6). In this case, outstanding log entries in row buffers must be written to NVM cells, making the checkpoint persistent. As row buffers are open, a write operation in the background results in a page hit. This latency corresponds to CAS device latency (tCAS).

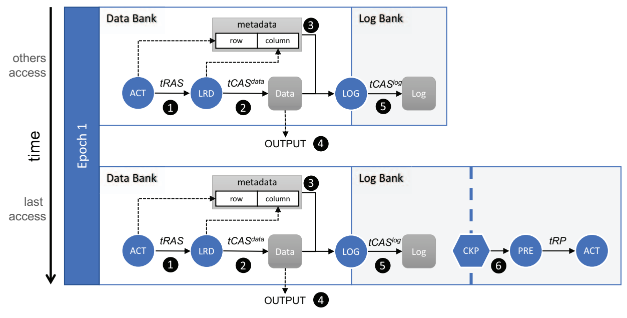
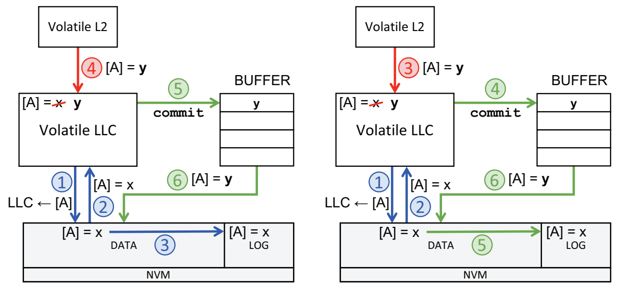


Fig. 7: Command sequence during an epoch on running.



(a) Logging-on-Read (b) Logging-on-Write

Fig. 8: DONUTS logging strategies.

A. Logging Strategies

DONUTS [15] is a crash consistency mechanism based on undo-logging. So, unlike redo-logging, no translation tables are needed because cache evictions are written to default memory locations. Figure 8 shows the original DONUTS Logging-on-Read (LoR) strategy and our new implementation, classified as Log-on-Write (LoW). In the first one, logs are performed from read operations, taking advantage of opening the memory row buffers. Since the crash consistency mechanism logs on the same read operation, the data copy from conventional to log banks generates no additional latency because the log row buffers are kept open. So, an additional latency ($\mathtt{CAS}_{\mathtt{log}}$) only occurs in log-row-buffer overflows. This strategy dilutes log operations during the execution of the current epoch. However, it generates unnecessary log operations since the data loaded from memory may not be modified in the current epoch. In contrast, the LoW strategy logs on write access, taking advantage of opening lines during write access to copy the data not yet modified from conventional to log bank. This process takes advantage of the read-modify-write nature of the DDRx protocol because when writing to a memory cell, the data is first copied to the row-buffer in the row opening process, modified, and only written back to the opened cells in the array.

B. Implementation

Our logging strategy was added to DONUTS design and implemented in Sniper simulator 7.4. In the memory controller, we implement the logging strategies described in Figure 8. Thus, after sending data from the main memory to Last Level Cache (LLC), the NVM controller makes a PIM logging operation, simulating data copying from conventional to log banks. This background operation takes advantage of opened row buffers in log banks, generating no additional cost except in buffer overflow or after a checkpoint commit. In these cases, data in the row buffers persist in cell arrays. The row buffers size and the persistence cost latency from row buffers to NVM cells (determined by `CAS` device latency) are configurable. Except for these modifications, no other component was changed.

Beyond DONUTS, we also carefully implemented PiCL [2] and NVOverlay [28] in the same simulator according to

TABLE II: Configuration system.

Processor	2 GHz
Number of cores	2-16
L1 D/I Cache	Private 32KB, 4-way, 64B block; 1 cycles hit
L2 Cache	Private 256KB, 8-way, 64B block; 4 cycles hit
L3 Cache	Shared 2MB, 8-way, 64B block; 30 cycles hit
NVM latencies	20ns read, 200ns write (STT-RAM)
DONUTS parameters	Thresholds = 100% associative set, 75% cache, 50ms timeout; 4KB log row buffer
PiCL parameters	32 entries on-chip undo logging; `acs-gap` = 3

its specifications. Thus, the MESI cache coherence policy was modified to add the respective mechanism's commit and checkpoint events. Also, to implement NVOverlay we add the triggers in the cache coherence protocol to version the logs according to the front-end system called Coherent Snapshot Tracking (CST), which after versioning them sends them to a Multi-snapshot NVM Mapping (MNM), back-end component responsible for writing logs in the background on NVM. All implementations are available at: https://github.com/kleberkruger/donuts.

IV. Evaluation

In order to evaluate LoW performance, we executed experiments using SPEC CPU2017 (single-core), PARSEC, and Splash2 (multi-core benchmarks integrated into Sniper Simulator). The experiments ran one billion instructions for each application, and system configuration parameters were similar to those defined in PiCL [2], except by NVM latency parameters updated to STT-RAM technology [29], as shown in Table II. Our first test set considered DONUTS performance in single-core and multi-core models ranging from 2 to 16 cores. The aim was to evaluate the runtime overhead between LoR and LoW strategies and compare them to other software-transparent projects. The following tests observed both DONUTS logging strategies regarding log reduction rate, memory space, and energy consumption.

A. Runtime Performance

Our first tests evaluated the DONUTS performance by comparing it to PiCL [2] and NVOverlay [28], the better-performing crash consistency mechanisms so far. As shown in Table II, we configured DONUTS with 4K log row buffer, associative-set, and cache-capacity thresholds at 100% and 75%, respectively, and timeout for checkpointing of 50ms. In PiCL, the gap between epochs was defined at 30 million instructions, and the `acs-gap` parameter (window size for pending epochs persistence) was assigned a value of three according to original paper settings.

1) Single-core Performance: Figure 9 shows the single-core performance of DONUTS using LoR and LoW strategies compared to PiCL. NVOverlay was not evaluated because its design is restricted to multi-core architectures. On average, the PiCL runtime overhead was 4.6% against 2.8% of DONUTS

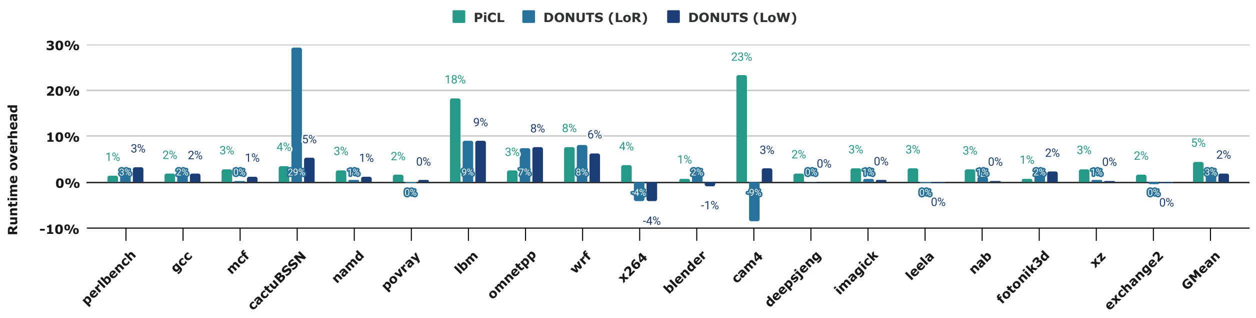
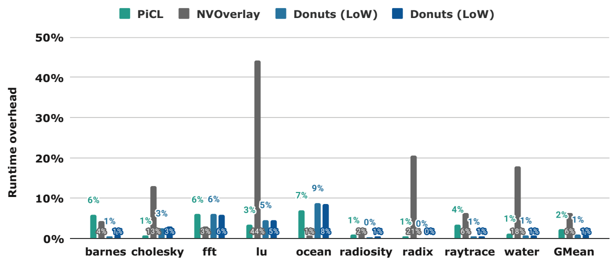
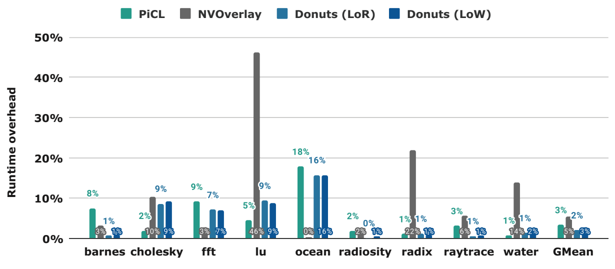
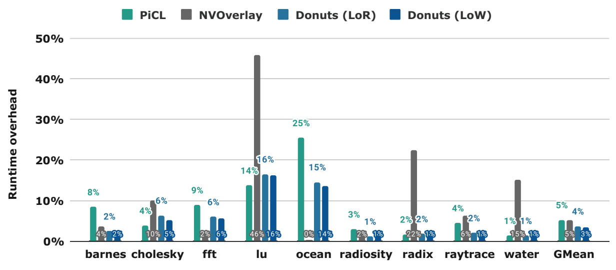


Fig. 9: Single-core performance of DONUTS and PiCL.


(a) 2-cores


(b) 4-cores


(c) 8-cores

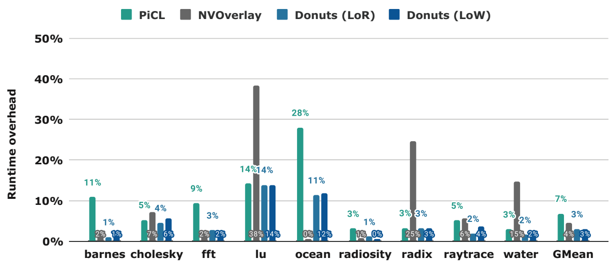
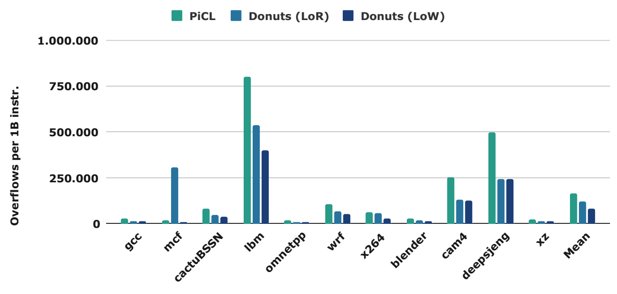

(d) 16-cores

Fig. 10: Multi-core performance of DONUTS, PiCL and NVOverlay.

LoW and 2% on LoR strategy. The `cactuBSSN` and `cam4` applications had significantly varied overhead between both DONUTS logging strategies. In cactuBSSN, the LoR strategy log flow generated an average memory bandwidth usage of 17% (versus 10% for LoW) and a log overhead of 29%. On the other hand, on `cam4`, the LoW strategy got a low log reduction rate (3.58%), justifying a better LoR performance. Meantime, LoR and LoW strategies presented a similar performance in most applications. This result occurs because although the LoR strategy generates more significant log writes to NVM, their logs are dissolved during the running epoch. In contrast, the LoW strategy concentrates logs on the persistence phase, causing queue delay increases during writes to log banks. However, DONUTS performs these operations outside the critical execution path due to overlapping epochs and writes to log banks in the background. Finally, considering average bandwidth usage, both DONUTS strategies maintained the same average of the baseline system while PiCL increased by 3%. Therefore, DONUTS presented 8% of average bandwidth usage, and PiCL obtained a result 1.37x higher than DONUTS.

2) *Multi-core Performance:* Figure 10 shows PiCL, NVOverlay and DONUTS runtime overheads on multi-core scenarios ranging from 2 to 16 cores. Both DONUTS versions obtained close runtime overheads, reaching approximately up to 3.7% and 3.3% in the LoW and LoR strategies, respectively. In contrast, due to increased memory bandwidth usage caused by intensified writes from on-chip undo buffer on multi-core architectures, when scaling to 8-16 cores, PiCL had a runtime overhead of 5.3% and 6.8%, respectively. For example, the bandwidth usage in the ocean application went from 43% in 2-cores, to 76% in 4-cores, 87% in 8-cores, and 98% in a 16-cores architecture. On the other hand, NVOverlay presented worst performance than PiCL on architectures with 2-4 cores (2.3% vs 6.4%, and 3.4% vs 5.5% of overhead),


(a) SPEC 2017 single-core applications.

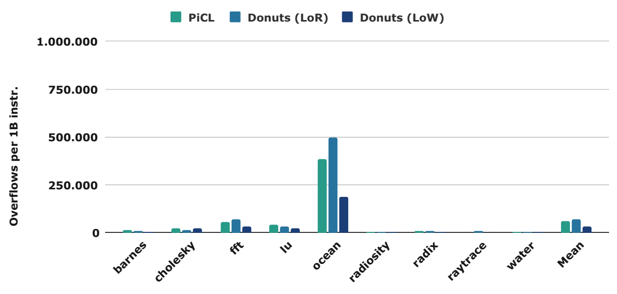
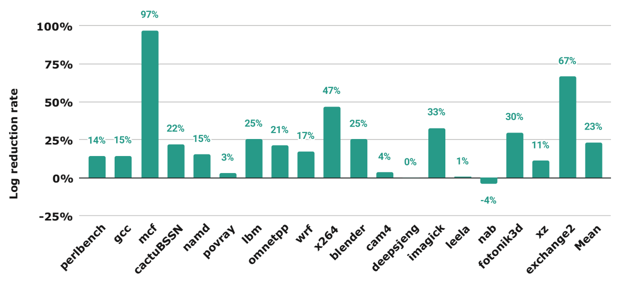

(b) Splash2 multi-core applications (8 cores).

Fig. 11: Number of buffer overflows per 1 billion instructions.


(a) SPEC 2017 single-core applications.

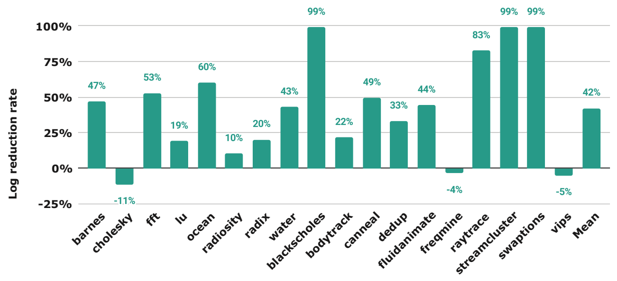

(b) Splash2 and Parsec multi-core applications (8 cores).

Fig. 12: Logging reduction rate of LoW to LoR.

similar in 8 cores (5.2% vs 5.1%), and superior when scaled to 16 cores (6.8% vs 4.5%). This performance is because the increasing of cores allows more versioned domains (VDs) components, which are used to synchronize versions of logs sent from its front-end to back-end component. With greater efficiency of VDs, more data is merged, reducing write-amplification (redundant log writes). Furthermore, its average runtime performance was worse in all tests compared to DONUTS. In addition, NVOverlay presented higher overheads in `lu`, `radix` and `water` applications, since the behavior of consecutive writes to a memory address intensifies data versioning, generating a significant increase in the logs (763%, 824% and 492%, respectively).

3) Buffer Overflows: Runtime overheads can be explained by additional latency in logging operation when on-chip undo buffer (PiCL) or log row buffer (DONUTS) overflows. In the first case, no more entry is available in the on-chip undo buffer, needing to persist its log entries in the NVM. In the second, DONUTS overflowed the log row buffer size, being necessary to write it in the log bank cell array. $\texttt{CAS}_{\texttt{log}}$ determines the cost of this operation. Figure 11 exhibits an average of buffer overflows per 1 billion instructions for each application. In applications where PiCL presented higher overheads compared to DONUTS (such as `lbm` and `cam4` from SPEC CPU2017, and `ocean` from Splash2), the number of additional latencies generated by the on-chip undo buffer was about 2x higher than DONUTS LoW. On the other hand, in `mcf` and `ocean` applications, the amount of buffer overflows in DONUTS LoR was higher than PiCL because both applications intensify read accesses (97% and 73%, respectively). However, this behavior does not reflect higher runtime overhead since, as applications do not modify cache data, the DONUTS dynamic epochs result in longer intervals due to the checkpoint events frequency depending on the overflow of dirty blocks in the cache.

B. Logging Reduction Rate and Checkpoint Size

The second test set analyzed logging reduction rate and memory space aspects between both DONUTS logging strategies. Figure 12 shows that the LoW strategy obtained an average log reduction of 23% on SPEC CPU2017 applications and 42% on multi-core applications of the Parsec and Splash2 benchmarks. In total, LoW strategy showed improvements in about 87% of applications, with 10% of them (`mcf`, `blackscholes`, `streamcluster` and `swaptions`) leading reduction rate above 97%, given intensive read access behavior of these applications. In the `streamcluster` application, for example, the read memory access rate is 98.9%. In another 10% of the applications, LoR obtained a better rate. In `cholesky` and `vips`, where LoW log rate was 11% and 5% higher than in LoR, 53% and 50% of the accesses to memory was for write operations, respectively.

Figure 13 shows memory space (in megabytes) spent creating checkpoints in Splash2 applications. The LoR strategy consumes 16 MB of memory in `fft` application, while LoW

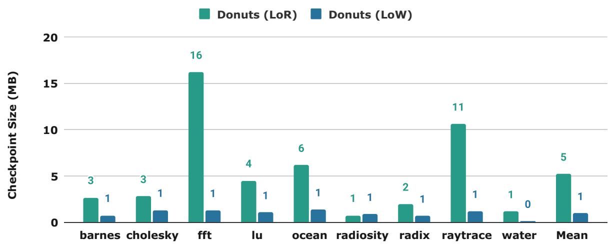
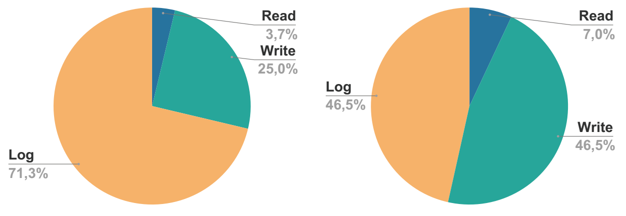
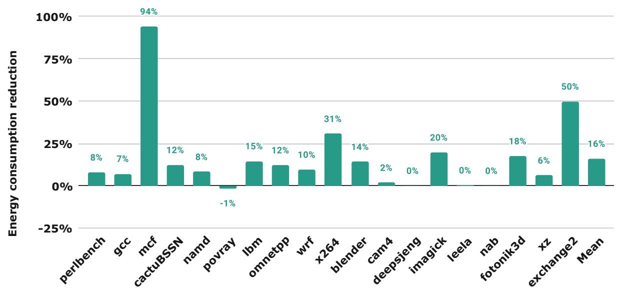


Fig. 13: DONUTS checkpoint size.

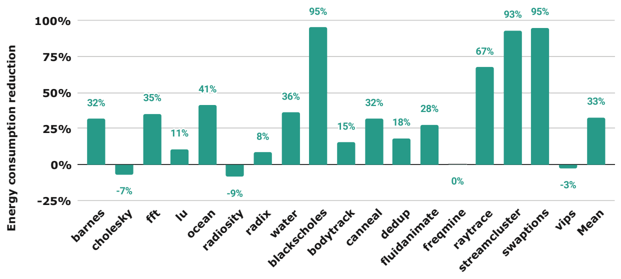


(a) Logging-on-Read (b) Logging-on-Write

Fig. 15: Energy consumption per operation.



(a) SPEC 2017 single-core applications.



(b) Splash2 and Parsec multi-core applications (8 cores).

Fig. 14: Energy consumption reduction of LoR $\rightarrow$ LoW.

consumes a maximum of 1.5 MB. The average size required for an LoR checkpoint is 5.2 MB versus 0.9 MB for LoW.

C. Energy Consumption

We used NVMExplorer [29] to generate accurate energy consumption estimates, whose tool contains updated latency and power consumption parameters of the leading current technologies in the literature. In general, write operations on NVM use 10x to 450x more energy than read, depending on the technology employed.

Figure 14 shows the energy consumption reduction of crash consistency mechanisms using as parameter STT-RAM memory. The columns present the energy reduction rate of the LoW strategy compared to LoR. In the `mcf` application of Figure 14a, for example, LoW reduced energy consumption by 94% due to 97% log reduction (shown in Figure 12) which, consequently, would generate costly write operations to NVM. On the other hand, negative values seen in `cholesky`, `radiosity` and `vips` applications mean higher energy consumption in LoW. However, on average, the energy consumption reduction for SPEC CPU2017 single-core applications was 16%, and in multi-core applications, whose average log reduction rate was higher, energy consumption was reduced by 33%.

Figure 15 displays the operations with the greatest impact on energy consumption when running the Parsec benchmark application suite on SST-RAM memory. In the LoR strategy, logging operations were responsible for 71.3% of energy consumption, reducing to 46.5% in the LoW strategy. By reducing the total energy cost, reading operations on energy consumption represent a more significant cost, growing from 3.7% to 7%.

V. Related Works

Existing works in the literature have proposed different methods to provide crash consistency. One way to categorize them is to separate into software or hardware solutions. The first support fault consistency by compilers or software libraries based on Persistent Transactional Memory (PTM) programming models, while hardware solutions modify architecture design to provide consistency on persistent memories. Hardware solutions have been more attractive alternatives, as they do not add new instructions to applications, generating less overhead. Another way to categorize these mechanisms is by the crash consistency technique adopted. The most common is by logging, which subdivides into redo-logging [3], [5], [10], [12], [14], [30]–[35], undo-logging [2], [11], [13], [15], [36], or a hybrid solution of both [37], [38]. Furthermore, other works have implemented shadow-paging [1], [16], [39], logging-structured [40] and out-of-placing [4] updates. Finally, it is also possible to categorize them according to their programming model (software-based or software-transparent). Table III summarizes some of the crash consistency mechanisms proposed in the literature.

Software-based: Software solutions [10], [14], [34], [36], [41]–[44] provide crash consistency by instructions for imposing fence and synchronizing read/write operations (`mfence` e `sfence`) and by instructions to force write-backs to cache lines (`clflush`, `clwb`, e `dccvap`). In addition to implementing a PTM, Mnemosyne defers checkpointing and log truncation to eliminate them from the transaction's critical path. Later, Atlas [42] guaranteed atomicity of the outer

TABLE III: Summary of some crash consistency mechanisms.
*** Note:** S.T.: software-transparent; PIM: processing-in-memory.

Project	**Type**	**Technique**	**S.T.**	**PIM**
ATOM [11]	hardware	undo-logging		
BPFS [39]	software	shadow-paging		
BPPM [30]	software	redo-logging		
CCHL [5]	hardware	redo-logging		
DCT [36]	hardware	undo-logging		
DHTM [31]	hardware	redo-logging		
DONUTS [15]	hardware	undo-logging	✓	✓
Dual-Page [1]	hardware	shadow paging	✓	
DudeTM [10]	software	redo-logging		
FWB [37]	hardware	undo+redo-logging		
HOOP [4]	hardware	out-of-place	✓	
LOC [32]	hardware	redo-logging		
LSNVMM [40]	software	log-structured		
Mnemosyne [14]	software	redo-logging		
MorLog [38]	hardware	undo+redo-logging		
NICO [3]	hardware	redo-logging	✓	
NV-Heaps [41]	software	redo-logging		
NVOverlay [28]	hardware	shadow-paging	✓	
PiCL [2]	hardware	undo-logging	✓	
Proteus [13]	hardware	undo-logging		
ReDU [33]	hardware	redo-logging		
SoftWrAP [34]	software	redo-logging		
SSP [16]	hardware	shadow paging		
ThyNVM [12]	hardware	redo-logging	✓	
WrAP [35]	hardware	redo-logging		

critical sections by extending them through semantic locks. NVThreads [43] was based on Atlas to provide an immediate replacement for pthreads. On the other hand, SFR [44] provided persistence in thread regions delimited by sync operations. Recent researches [45], [46] have improved PTM scalability with better concurrency control protocols and hybrid logging schemes in DRAM + NVM.

Hardware-based: Hardware strategies mitigate the runtime overhead of software solutions. However, despite being hardware, most of them [4], [5], [11], [13], [16], [31]–[33], [35]–[38] are not transparent to software, depending on programming models PTM-based to indicate atomic transactions, being hardware restricted to handle logs and checkpoint metadata. In contrast, other works [1]–[4], [12] proposed software-transparent solutions. ThyNVM, for example, adopts a synchronized overlapping epochs model, but the need for synchronization affects system performance, especially in multicore scenarios. PiCL establishes a concept of multi-undo-logging, which decouples system epochs and allows asynchronous persistence outside the critical execution path, but its model generates excessive writes in the NVM. NICO [3] designed a lightweight checkpointing scheme that needs to flush and modify a smaller amount of data when creating a consistent snapshot of persistent memory data. However, like PiCL, it also increases the traffic between processor and memory. DONUTS [15] distinguishes them by exploiting Processing-in-Memory, but its LoR strategy creates unnecessary entries.

VI. Conclusion

Although recent works have proposed alternatives to reduce runtime overhead, logging operations tend to multiply the number of writes to NVM and significantly increase data movement between processor and memory. DONUTS proposes a logging strategy via PIM that reduces the external bandwidth usage but generates an excessive amount of unnecessary logs, negatively impacting the internal bandwidth, energy consumption, and the device's useful lifetime. On the other hand, our strategy maintains DONUTS performance while reducing log writes by up to 42% on average, reaching up to 99% in specific applications with intensive read access. Consequently, as NVM power/energy consumption is a function of the number of reads/writes, our logging strategy positively impacts the energy consumption, showing an average reduction of 32% in multi-core applications.

Acknowledgment

This project is supported by FAPESP (2013/08293-7, 2019/26702-8), CNPq (438445/2018-0, 312088/2020-5), CAPES (Finance Code 001), and FAEPEX.

References

[1] S. Wu, F. Zhou, X. Gao, H. Jin, and J. Ren, "Dual-page checkpointing: An architectural approach to efficient data persistence for in-memory applications," *ACM Trans. Archit. Code Optim.*, vol. 15, no. 4, Jan. 2019.

[2] T. M. Nguyen and D. Wentzlaff, "Picl: A software-transparent, persistent cache log for nonvolatile main memory," in *2018 51st Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, 2018, pp. 507–519.

[3] X. Wei, D. Feng, W. Tong, J. LIU, and L. Ye, "Nico: Reducing software-transparent crash consistency cost for persistent memory," *IEEE Transactions on Computers*, vol. 68, no. 9, pp. 1313–1324, 2019.

[4] M. Cai, C. C. Coats, and J. Huang, "Hoop: Efficient hardware-assisted out-of-place update for non-volatile memory," in *2020 ACM/IEEE 47th Annual International Symposium on Computer Architecture (ISCA)*, 2020, pp. 584–596.

[5] X. Wei, D. Feng, W. Tong, J. Liu, C. Wang, and L. Ye, "Cchl: Compression-consolidation hardware logging for efficient failure-atomic persistent memory updates," in *49th International Conference on Parallel Processing - ICPP*, ser. ICPP '20. New York, NY, USA: Association for Computing Machinery, 2020.

[6] Q. Wu, F. Sun, W. Xu, and T. Zhang, "Using multilevel phase change memory to build data storage: A time-aware system design perspective," *IEEE Transactions on Computers*, vol. 62, no. 10, pp. 2083–2095, 2013.

[7] H. Noguchi, K. Ikegami, K. Kushida, K. Abe, S. Itai, S. Takaya, N. Shimomura, J. Ito, A. Kawasumi, H. Hara, and S. Fujita, "7.5 a 3.3ns-access-time 71.2μw/mhz 1mb embedded stt-mram using physically eliminated read-disturb scheme and normally-off memory architecture," in *2015 IEEE International Solid-State Circuits Conference - (ISSCC) Digest of Technical Papers*, 2015, pp. 1–3.

[8] R. Fackenthal, M. Kitagawa, W. Otsuka, K. Prall, D. Mills, K. Tsutsui, J. Javanifard, K. Tedrow, T. Tsushima, Y. Shibahara, and G. Hush, "19.7 a 16gb reram with 200mb/s write and 1gb/s read in 27nm technology," in *2014 IEEE International Solid-State Circuits Conference Digest of Technical Papers (ISSCC)*, 2014, pp. 338–339.

[9] Intel. (2015, Jul.) Intel and micron produce breakthrough memory technology. [Online]. Available: https://newsroom.intel.com/news-releases/intel-and-micron-produce-breakthrough-memory-technology/

[10] M. Liu, M. Zhang, K. Chen, X. Qian, Y. Wu, W. Zheng, and J. Ren, "Dudetm: Building durable transactions with decoupling for persistent memory," in *International Conference on Architectural Support for Programming Languages and Operating Systems - ASPLOS*, vol. Part F127193, 2017, pp. 329–343.

[11] A. Joshi, V. Nagarajan, S. Viglas, and M. Cintra, "Atom: Atomic durability in non-volatile memory through hardware logging," in *2017 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, 2017, pp. 361–372.

[12] J. Ren, J. Zhao, S. Khan, J. Choi, Y. Wu, and O. Mutiu, "Thynvm: Enabling software-transparent crash consistency in persistent memory systems," in *2015 48th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, 2015, pp. 672–685.

[13] S. Shin, S. K. Tirukkovalluri, J. Tuck, and Y. Solihin, "Proteus: A flexible and fast software supported hardware logging approach for nvm," in *2017 50th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, 2017, pp. 178–190.

[14] H. Volos, A. J. Tack, and M. M. Swift, "Mnemosyne: Lightweight persistent memory," *SIGARCH Comput. Archit. News*, vol. 39, no. 1, p. 91–104, mar 2011.

[15] K. Kruger, R. Pannain, and R. Azevedo, "Donuts: An efficient method for checkpointing in non-volatile memories," *Concurrency and Computation: Practice and Experience*, p. e7574, 2022.

[16] Y. Ni, J. Zhao, H. Litz, D. Bittman, and E. Miller, "Ssp: Eliminating redundantwrites in failure-atomic nvrams via shadow sub-paging," in *Proceedings of the Annual International Symposium on Microarchitecture, MICRO*, 2019, pp. 836–848.

[17] M. Gokhale, B. Holmes, and K. Iobst, "Processing in memory: the terasys massively parallel pim array," *Computer*, vol. 28, no. 4, pp. 23–31, 1995.

[18] S. Ghose, A. Boroumand, J. S. Kim, J. Gómez-Luna, and O. Mutlu, "Processing-in-memory: A workload-driven perspective," *IBM Journal of Research and Development*, vol. 63, no. 6, pp. 3:1–3:19, 2019.

[19] K. K. Chang, P. J. Nair, D. Lee, S. Ghose, M. K. Qureshi, and O. Mutlu, "Low-cost inter-linked subarrays (lisa): Enabling fast inter-subarray data movement in dram," in *2016 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, 2016, pp. 568–580.

[20] S. Aga, S. Jeloka, A. Subramaniyan, S. Narayanasamy, D. Blaauw, and R. Das, "Compute caches," in *2017 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, 2017, pp. 481–492.

[21] S. Li, C. Xu, Q. Zou, J. Zhao, Y. Lu, and Y. Xie, "Pinatubo: A processing-in-memory architecture for bulk bitwise operations in emerging non-volatile memories," in *2016 53nd ACM/EDAC/IEEE Design Automation Conference (DAC)*, 2016, pp. 1–6.

[22] S. Angizi, Z. He, and D. Fan, "Pima-logic: A novel processing-in-memory architecture for highly flexible and energy-efficient logic computation," in *2018 55th ACM/ESDA/IEEE Design Automation Conference (DAC)*, 2018, pp. 1–6.

[23] S. Angizi, Z. He, A. S. Rakin, and D. Fan, "Cmp-pim: An energy-efficient comparator-based processing-in-memory neural network accelerator," in *Proceedings of the 55th Annual Design Automation Conference*, ser. DAC '18. New York, NY, USA: Association for Computing Machinery, 2018.

[24] S. Angizi, J. Sun, W. Zhang, and D. Fan, "Aligns: A processing-in-memory accelerator for dna short read alignment leveraging sot-mram," in *Proceedings of the 56th Annual Design Automation Conference 2019*, ser. DAC '19. New York, NY, USA: Association for Computing Machinery, 2019.

[25] S. Hamdioui, L. Xie, H. A. D. Nguyen, M. Taouil, K. Bertels, H. Corporaal, H. Jiao, F. Catthoor, D. Wouters, L. Eike, and J. van Lunteren, "Memristor based computation-in-memory architecture for data-intensive applications," in *Proceedings of the 2015 Design, Automation & Test in Europe Conference & Exhibition*, ser. DATE '15. San Jose, CA, USA: EDA Consortium, 2015, p. 1718–1725.

[26] P.-E. Gaillardon, L. Amarú, A. Siemon, E. Linn, R. Waser, A. Chattopadhyay, and G. De Micheli, "The programmable logic-in-memory (plim) computer," in *Proceedings of the 2016 Conference on Design, Automation & Test in Europe*, ser. DATE '16. San Jose, CA, USA: EDA Consortium, 2016, p. 427–432.

[27] S. Li, D. Niu, K. T. Malladi, H. Zheng, B. Brennan, and Y. Xie, "Drisa: A dram-based reconfigurable in-situ accelerator," in *Proceedings of the 50th Annual IEEE/ACM International Symposium on Microarchitecture*, ser. MICRO-50 '17. New York, NY, USA: Association for Computing Machinery, 2017, p. 288–301.

[28] Z. Wang, C.-H. Choo, M. A. Kozuch, T. C. Mowry, G. Pekhimenko, V. Seshadri, and D. Skarlatos, "Nvoverlay: Enabling efficient and scalable high-frequency snapshotting to nvm," in *2021 ACM/IEEE 48th Annual International Symposium on Computer Architecture (ISCA)*, 2021, pp. 498–511.

[29] L. Pentecost, A. Hankin, M. Donato, M. Hempstead, G.-Y. Wei, and D. Brooks, "Nvmexplorer: A framework for cross-stack comparisons of embedded non-volatile memories," in *2022 IEEE International Symposium on High-Performance Computer Architecture*, 2022, pp. 938–956.

[30] Y. Lu, J. Shu, and L. Sun, "Blurred persistence in transactional persistent memory," in *2015 31st Symposium on Mass Storage Systems and Technologies (MSST)*, 2015, pp. 1–13.

[31] A. Joshi, V. Nagarajan, M. Cintra, and S. Viglas, "Dhtm: Durable hardware transactional memory," in *Proceedings of the 45th Annual International Symposium on Computer Architecture*, ser. ISCA '18. IEEE Press, 2018, p. 452–465.

[32] Y. Lu, J. Shu, L. Sun, and O. Mutlu, "Loose-ordering consistency for persistent memory," in *2014 IEEE 32nd International Conference on Computer Design (ICCD)*, 2014, pp. 216–223.

[33] J. Jeong, C. Park, J. Huh, and S. Maeng, "Efficient hardware-assisted logging with asynchronous and direct-update for persistent memory," in *Proceedings of the Annual International Symposium on Microarchitecture, MICRO*, vol. 2018-October, 2018, pp. 520–532.

[34] E. Giles, K. Doshi, and P. Varman, "Softwrap: A lightweight framework for transactional support of storage class memory," in *IEEE Symposium on Mass Storage Systems and Technologies*, vol. 2015-August, 2015.

[35] K. Doshi, E. Giles, and P. Varman, "Atomic persistence for scm with a non-intrusive backend controller," in *International Symposium on High-Performance Computer Architecture*, vol. 2016-April, 2016, pp. 77–89.

[36] A. Kolli, S. Pelley, A. Saidi, P. Chen, and T. Wenisch, "High-performance transactions for persistent memories," in *International Conference on Architectural Support for Programming Languages and Operating Systems - ASPLOS*, vol. 02-06-April-2016, 2016, pp. 399–411.

[37] M. Ogleari, E. Miller, and J. Zhao, "Steal but no force: Efficient hardware undo+redo logging for persistent memory systems," in *International Symposium on High-Performance Computer Architecture*, vol. 2018-Feb, 2018, pp. 336–349.

[38] X. Wei, D. Feng, W. Tong, J. Liu, and L. Ye, "Morlog: Morphable hardware logging for atomic persistence in non-volatile main memory," in *2020 ACM/IEEE 47th Annual International Symposium on Computer Architecture (ISCA)*, 2020, pp. 610–623.

[39] J. Condit, E. B. Nightingale, C. Frost, E. Ipek, B. Lee, D. Burger, and D. Coetzee, "Better i/o through byte-addressable, persistent memory," in *Proceedings of the ACM SIGOPS 22nd Symposium on Operating Systems Principles*, ser. SOSP '09. New York, NY, USA: Association for Computing Machinery, 2009, p. 133–146.

[40] Q. Hu, J. Ren, A. Badam, J. Shu, and T. Moscibroda, "Log-structured non-volatile main memory," in *Proceedings of the 2017 USENIX Conference on Usenix Annual Technical Conference*, ser. USENIX ATC '17. USA: USENIX Association, 2017, p. 703–717.

[41] J. Coburn, A. M. Caulfield, A. Akel, L. M. Grupp, R. K. Gupta, R. Jhala, and S. Swanson, "Nv-heaps: Making persistent objects fast and safe with next-generation, non-volatile memories," *SIGARCH Comput. Archit. News*, vol. 39, no. 1, p. 105–118, Mar. 2011.

[42] D. R. Chakrabarti, H.-J. Boehm, and K. Bhandari, "Atlas: Leveraging locks for non-volatile memory consistency," *SIGPLAN Not.*, vol. 49, no. 10, p. 433–452, Oct. 2014.

[43] T. C.-H. Hsu, H. Brügner, I. Roy, K. Keeton, and P. Eugster, "Nvthreads: Practical persistence for multi-threaded applications," in *Proceedings of the Twelfth European Conference on Computer Systems*, ser. EuroSys '17. New York, NY, USA: Association for Computing Machinery, 2017, p. 468–482.

[44] V. Gogte, S. Diestelhorst, W. Wang, S. Narayanasamy, P. M. Chen, and T. F. Wenisch, "Persistency for synchronization-free regions," *SIGPLAN Not.*, vol. 53, no. 4, p. 46–61, Jun. 2018.

[45] J. Gu, Q. Yu, X. Wang, Z. Wang, B. Zang, H. Guan, and H. Chen, "Pisces: A scalable and efficient persistent transactional memory," in *Proceedings of the 2019 USENIX Conference on Usenix Annual Technical Conference*, ser. USENIX ATC '19. USA: USENIX Association, 2019, p. 913–928.

[46] R. M. Krishnan, J. Kim, A. Mathew, X. Fu, A. Demeri, C. Min, and S. Kannan, "Durable transactional memory can scale with timestone," in *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems*, ser. ASPLOS '20. New York, NY, USA: Association for Computing Machinery, 2020, p. 335–349.