Energy Aware Persistence: Reducing Energy Overheads of Memory-based Persistence in NVMs

Sudarsun Kannan
College of Computing,
Georgia Tech
sudarsun@gatech.edu

Moinuddin Qureshi
School of ECE, Georgia Tech
moin@ece.gatech.edu

Ada Gavrilovska
College of Computing,
Georgia Tech
ada@cc.gatech.edu

Karsten Schwan
College of Computing,
Georgia Tech

ABSTRACT

Next generation byte addressable nonvolatile memories (NVMs) such as PCM, Memristor, and 3D X-Point are attractive solutions for mobile and other end-user devices, as they offer memory scalability as well as fast persistent storage. However, NVM's limitations of slow writes and high write energy are magnified for applications that require atomic, consistent, isolated and durable (ACID) persistence. For maintaining ACID persistence guarantees, applications not only need to do extra writes to NVM but also need to execute a significant number of additional CPU instructions for performing NVM writes in a transactional manner. Our analysis shows that maintaining persistence with ACID guarantees increases CPU energy up to 7.3x and NVM energy up to 5.1x compared to a baseline with no ACID guarantees. For computing platforms such as mobile devices, where energy consumption is a critical factor, it is important that the energy cost of persistence is reduced.

To address the energy overheads of persistence with ACID guarantees, we develop novel energy-aware persistence (EAP) principles that identify data durability (logging) as the dominant factor in energy increase. Next, for low energy states, we formulate energy efficient durability techniques that include a mechanism to switch between performance and energy efficient logging modes, support for NVM group commit, and a memory management method that reduces energy by trading capacity via less frequent garbage collection. For critical energy states, we propose a relaxed durability mechanism – ACI-RD – that relaxes data logging without affecting the correctness of an application. Finally, we evaluate EAP's principles with real applications and benchmarks. Our experimental results demonstrate up to 2x reduction in CPU and 2.4x reduction in NVM energy usage compared to the traditional ACID persistence.

1. INTRODUCTION

Industry announcements claim future byte addressable, nonvolatile memory (NVM) technologies like phase change memory (PCM) and 3D Xpoint [1] to have 100x lower access latency compared to that of Flash/SSD devices and to scale to 4-8 times the density of DRAM, without consuming refresh power. The promised outcomes include larger memory capacities [37] at lower energy usage [29] compared to DRAM, coupled with faster persistent data storage and access than Flash/SSD. Given this, current end-user devices such as smartphones, tablets, and laptops with limited DRAM capacity and slow flash storage [27] are an important target for NVM. Particularly attractive are the dual benefits of using NVM for higher memory capacity (volatile heap) and faster data persistence [29, 37, 23].

However, using NVM for persistence requires fail-safe guarantees from application or device failures. For example, when saving a user id (key) and password (value) into a persistent key-value store, a failure after updating the key, but before saving the password can result in an undesired state. To guard against such problems, applications must satisfy atomicity (A), consistency (C), isolation (I), and durability (D), also commonly referred to as ACID. In ACID, 'A' requires either all or none of the operations to complete in a transaction, whereas 'C' requires each update to convert persistent data from one consistent state to another. 'I' ensures concurrent updates are invisible to each other, and is realized through race free mechanisms and synchronization, and finally, 'D' requires that any changes to application data committed before a crash can be recovered, usually achieved by logging the updates.

Prior work has focused mainly on optimizing the performance of persistent storage that satisfies ACID properties by either treating NVM as a fast disk [36, 14, 12] or by using NVM as heap [39, 11, 35], also commonly referred to as memory-based persistence (hereafter simply **memory persistence**). In an NVM-as-a-disk approach, applications rely on the filesystem for ACID guarantees, whereas in memory persistence, applications use persistent allocators to allocate and update heap objects inside a transaction before committing them. The benefits of memory persistence over disk-based persistence has been widely studied for a broad range of applications such as key-value stores, persistent in-memory data structures, object-based storage, NoSQL and SQL database.

When using memory persistence, the correctness of an application is maintained by frequently flushing the application data and the related metadata, whereas durability is achieved by logging them. The metadata refers to information such as the state of a persistent allocator, and the data log headers. Consequently, supporting memory persistence not only increases the energy cost due to increase in NVM accesses but also increases the CPU instructions, hence resulting in a higher CPU energy use. Our analysis shows that memory persistence with ACID guarantees incurs a CPU energy increase of up to 7.3x and NVM energy increases of up to

PACT '16, September 11-15, 2016, Haifa, Israel
© 2016 ACM. ISBN 978-1-4503-4121-9/16/09...$15.00
DOI: http://dx.doi.org/10.1145/2967938.2967953

5.1x compared to a baseline that does not maintain persistence. For computing platforms (such as mobile devices) where energy consumption is a critical factor, it is important that the energy cost of persistence is reduced.

To address this, we propose **energy-aware persistence** (EAP) as an essential element for deploying NVM in end-user devices. Prior work has sought to reduce the cost of consistent and durable memory persistence [42, 12, 35, 30] using performance-centric solutions. These solutions optimize the transactional component of the memory persistence without considering whether such optimizations address the durability- or correctness-related persistent costs. In contrast, EAP analyzes the energy implications of the correctness and durability software components in detail. Our analysis is the first to identify that durability (logging) costs are the most significant contributor to energy usage in an memory persistence, and to quantitatively demonstrate the significance of both NVM and CPU-related energy costs in the overall energy overheads of supporting persistence. EAP's energy reduction methods are applicable for other non-NVM storage mediums too. EAP leverages these observations to reduce the durability-related energy cost of ACID. The EAP design comprises of two parts, (i) energy efficient durability for low-but-not-critical energy states, and (ii) relaxed durability (ACI-RD) for critically low energy states. We next briefly introduce the key ideas of energy efficient durability and ACI-RD.

Energy efficient durability. Durability methods can be broadly classified into REDO and UNDO logging. In REDO, the application updates are first written to a log, and when the log is full, the log contents are committed to the original data location. This method is also referred to as Write-Ahead Logging (WAL). In contrast, for UNDO, the updates are committed in-place by first taking a backup of the original data. Most state-of-the-art memory persistence designs use WAL because UNDO doubles the required NVM updates in a transaction. However, as we show in Section 3, WAL is performance-efficient but consumes higher energy compared to the UNDO logging. We also show that persistent memory allocation and garbage collection have to be logged, and add a significant CPU and NVM energy cost. To address these issues, we first design energy efficient durability. The energy efficient durability method provides a mechanism to switch automatically between performance-efficient WAL and energy-efficient UNDO logging modes. Further, to reduce the allocation and garbage collection cost, the energy efficient durability reduces frequent allocation and garbage collection, and thereby trades additional NVM capacity use with lower energy cost.

Relaxed durability (ACI-RD). The energy efficient durability method is not sufficient when the energy availability is critical. For such energy critical states, we design ACI-RD, that relaxes the strict application data logging (D of ACID) requirements of a traditional ACID model, without impacting the correctness (ACI) of an application. Although ACI-RD weakens the durability guarantees and increases the application recovery time after a failure, we show that by careful design, energy consumption can be significantly reduced without weakening the correctness property of ACID. Figure 1 provides a high-level timing diagram comparing a traditional ACID vs. our ACI-RD design. In the ACID case, after application execution, both application data, and metadata are ordered, flushed and logged. In contrast, with EAP, when energy is critically low, data durability is relaxed for application data, but not for its metadata, thus maintaining the ACI properties essential for correct operation after a failure.

EAP uses an epoch-based execution model, where the target energy level at the start of each epoch drives the choices of suitable alternatives by dynamically measuring the durability-related energy

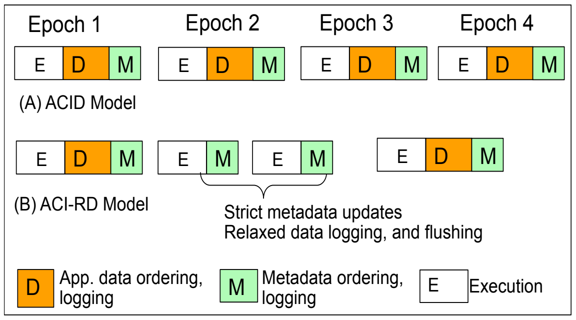


Figure 1: (A) Traditional ACID epoch execution, (B) ACID-RD: Data logging relaxed for critically low energy Epoch 2,3

of each ACID stack component. If the energy budget is low-but-not critical, EAP just uses the efficient durability, but when the energy is critically low, EAP combines the efficient durability methods with ACID-RD by relaxing the durability guarantees.

We implement EAP's software-based efficient durability and ACI-RD methods by extending Intel's NVML [21] open source persistence library. Our evaluations using different memory persistence benchmarks and applications show that EAP's efficient durability for low energy state reduces NVM and CPU energy usage by up to 28% and 41%, respectively. By combining efficient durability with ACI-RD for critical energy states, EAP achieves up to 2.1x and 2.4x CPU and NVM energy reduction, respectively.

In summary, this paper makes the following contributions:

- *Durability cost.* We quantify data durability as the key source of energy bottlenecks for applications with memory persistence with ACID guarantees (Section 3).
- *Energy efficient durability.* To reduce durability-related energy overheads, we propose energy efficient principles that trade off performance and capacity for energy, support flexible logging, and NVM group commit methods when the available budget is low (Section 4).
- *Epoch-based relaxed durability (ACI-RD).* For critically low energy budgets, we design a novel ACI-RD model that relaxes data durability based on the energy usage in each epoch, without affecting application correctness (Section 5).
- *Experimental evaluation.* We evaluate EAP with realistic benchmarks and applications and demonstrate the substantial benefits of EAP (Section 6).

In the rest of the paper, Section 2 discusses the background of the memory-based persistence and the related work, Section 3 analyzes the energy overheads of different memory persistence software components such as cache flush and logging required for providing memory persistence. Section 4 discusses the energy efficient durability principles for low-but-not-critical energy budget. Section 5 describes the ACI-RD design and implementation details for critically low energy budget. Section 6 presents the evaluation methodology, and the energy benefits and implications of the proposed EAP design, followed by the conclusion in Section 7

2. BACKGROUND AND RELATED WORK

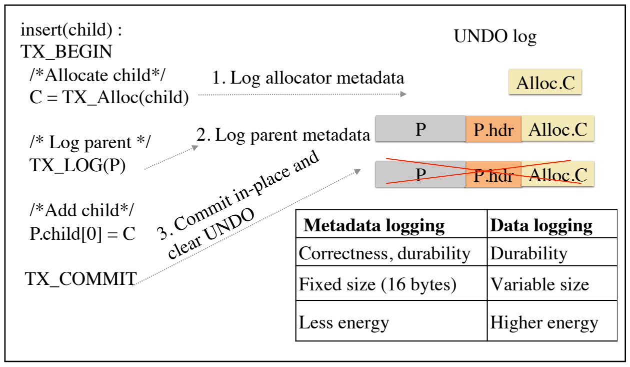
Byte-addressable NVMs. NVMs such as PCM are byte-addressable persistent devices expected to be 100x faster (read-write performance) compared to current SSDs [12, 14, 8]. Further, NVM can scale 2x-4x higher density than DRAM [1] as they can store multiple bits per cell with no refresh power, with known limitations imposed by an endurance of a few million writes per cell. These attributes make NVM a suitable candidate for replacing SSDs. Additionally, NVM can be extended as a memory placed in parallel with DRAM, connected via the memory bus. NVMs offer load/store access interface that can potentially avoid POSIX-based block access supported in current storage devices. NVMs' read latency is comparable to DRAM latency, but the write latency is 5x-10x slower due to high SET times [28], and more importantly, the active write energy is 15-40x higher than DRAM [41, 28].

NVM usage models. One usage model is to treat DRAM as a cache for NVM, while also providing a file system interface, such as in [37]. However, with this usage mode, supporting persistence with ACID guarantees can be expensive because the data needs to be flushed and moved to multiple levels (processor cache to DRAM to NVM) before committing it to NVM. Moneta [8], BPFS [12], and PMFS [14] address this by designing a new filesystem for NVM that does not use DRAM as a cache, and provides direct access to NVM. Although the I/O performance improves with these filesystems, the use of the block-based POSIX I/O operations such as *read(), write()* require frequent user-to-kernel transitions.

An alternative usage model is to provide applications with a byte-level access to NVM. This reduces the user-to-kernel transition and the kernel-level filesystem cost. POSIX already provides a byte-level access to a persistent storage device via memory-mapped (*mmap()*) interface. But *mmap()* is coarse grained and inflexible [11, 35, 23, 25], and it does not provide ACID guarantees [14]. Prior research addresses these issues by designing a persistent object store [39, 11, 21] for NVM that provides a byte addressable load-store interface. The object store also provides a persistent memory allocator and transactional persistence support. In EAP, **we use the persistent object store model** and address the energy overheads of persistence.

Durability and consistency. The processor cache can buffer writes and significantly reduce the impact of high NVM write latency. Current OSes and architectures use a write-back cache model in which the cache lines can be evicted in any order. However, for guaranteeing ACID properties in persistent storage, writes to NVM must be ordered for preserving the application correctness [35, 30]. For example, committing the metadata before committing the original data can be dangerous. The prior research uses a write-through cache model [39] with an epoch-based cache eviction [12] supported by memory barriers or introduces a persistent processor cache. Besides the write ordering issues, data durability can be affected due to a power failure that destroys the cached data, resulting in a non-deterministic state. Although applications can flush the cache to reduce non-recoverable failures, for ACID guarantees, other mechanisms such as logging (e.g., UNDO or REDO) are required. However, frequent cache flushes and logging are expensive.

Reducing ACID overheads on performance. Recent research such as [42, 23, 35, 43, 30] have proposed both hardware and software methods to reduce NVM overheads while preserving ACID. They have also considered relaxed consistency models [35, 30] that do not impact application performance. A recent research, persistent hardware transactional memory (PHTM) [7] analyzes the benefits of supporting persistence for the transactional memory hardware when providing strict ACID guarantees. PHTM's hardware extension improves parallelism and transaction performance resulting in application speedup. However, even with the hardware extension, results show up to 5x slowdown for supporting strict ACID guarantees compared to no ACID support. In contrast, EAP aims to reduce the number of CPU instructions, NVM accesses, and energy, by optimizing and relaxing strict ACID guarantees without impacting the correctness of an application. More importantly, EAP does not require hardware changes. We believe its software methods are complementary to the hardware extensions.



```
insert(child) :
TX_BEGIN
 /*Allocate child*/
 C = TX_Alloc(child)

 /* Log parent */
 TX_LOG(P)

 /*Add child*/
 P.child[0] = C

 TX_COMMIT
```

Metadata logging	Data logging
Correctness, durability	Durability
Fixed size (16 bytes)	Variable size
Less energy	Higher energy

Figure 2: Software NVM memory persistence. The figure shows steps for inserting a child node to a persistent B-tree inside a transaction, including allocator metadata and data logging.

NVM-based memory persistence. Recent research has extensively explored memory persistence for NVMs with ACID guarantees [39, 11]. Support for memory persistence requires application-level changes, and the objects made persistent have to be logged either with a UNDO log or with a write-ahead log (WAL). In contrast, the disk-based persistence used in prior database research [36, 6] log the entire page even when a word in the page changes.

To provide a background of how memory persistence works, Figure 2 shows an example of inserting a new node to a persistent B-tree inside a software transaction. First, a new child node is allocated using the persistent memory allocator and the allocator metadata (Alloc.C) for this child node is logged. Second, before adding the child node to the parent, the parent is backed into an UNDO log. The backup involves first writing the parent node (P) – considered as an application data – to the UNDO log, followed by a header of the application data (P.hdr) – considered as the application metadata. After UNDO logging, the parent pointer to the child node is updated, and when the transaction successfully commits, the UNDO logs (application data and metadata) are cleared. In the event of a failure before a transaction commit, the application metadata (P.hdr) is used to UNDO any data changes and restore the parent (P) to a state before the child node was added, using the UNDO log. The allocator metadata (Alloc.C) is used for the garbage collection of the allocated memory for the child node. Note that the metadata (Alloc.C and P.hdr) is required for correctness of the application and is typically much smaller (16B) than the data, thus requiring much lower energy costs during updates. We target applications which, as in this example, require memory persistence, and we focus on reducing the energy costs associated with providing ACID requirements in memory persistence.

Memory persistence libraries. Prior work has developed memory persistence libraries that include Mnemosyne [39], NV-Heaps [11], NVML [21], and Atlas [9]. Both Mnemosyne and NV-Heaps use transactional memory persistence, and differ in the granularity of logging (word versus object). Atlas, a recent work, differs from prior work by providing semantics for locking-based persistence in contrast to transactional memory persistence. While implementations such as NV-Heaps and Atlas are not openly available, Mnemosyne's legacy user-level Intel STM libraries are no longer functional. Also, Mnemosyne's OS component suffers significant cache overhead [24]. Hence, in this work, we use the NVML library from Intel for its comprehensive memory persistence implementation, SNIA compatibility, and architecture-specific optimizations. Other persistent allocators such as NVMAlloc [31], nvm_malloc [38], and NVMMalloc [40] only provide ACID guarantees for the allocator, but not for the transactional updates important for persistence. We also demonstrate EAP's generic principles for other well-known applications such as SQLite [5], and Snappy compression [16].

Relaxing persistence. To reduce persistence overheads, Pelly et al. [35] propose a consistency model that allows reordering between one or more independent transactions for better concurrency. Reordering is done only for writes to the cache, whereas writes to the NVM from the cache are ordered. This approach constitutes a variation of the epoch-based consistency model that requires hardware and software changes for reordering writes. Y. Lu et al. [30] propose a loose ordering consistency (LOC) protocol for relaxing intra- and inter-transaction ordering. In their approach, the log area is divided into blocks of 64 bytes with one metadata header for seven blocks. Committing log data in block-groups avoids repeated metadata update. After a failure, the consistency is validated using the group-level metadata. LOC assumes the presence of a nonvolatile cache. FIRM [43] improves persistent storage performance by adding intelligence to the memory controller for prioritizing persistent data updates over non-persistent data updates. All of the above proposals require hardware and software modification. A recent software-only solution, NVRAMDB [36], uses NVM as a disk for page-based persistence and proposes performance optimizations specifically for databases. NVRAMDB uses a batched/group commit that first copies the original data to a UNDO rollback/recovery log, then buffers multiple transactions in a DRAM buffer, and finally, it commits all the buffered data to NVM. Although this approach improves the throughput, it does not change the total data logged to NVM. In fact, by buffering transactions in DRAM and then copying to NVM, this solution consumes additional CPU and DRAM energy. In contrast, we propose a group commit update customized for memory persistence that reduces energy use.

EAP versus prior work. EAP differs from the work reviewed above [35, 30, 43, 36] in several ways. (1) EAP identifies and addresses the ACID software components that increase energy. (2) Other than NVM support, EAP does not require additional hardware changes. (3) Prior work principally seeks improvements in application performance, a case in point being the lazy asynchronous, or relaxed atomicity and ordering in [30] and [35]. Using such methods changes when certain actions are taken, but they do not reduce the total CPU instructions or NVM accesses. As a result, prior research does not directly reduce the persistence-related energy consumption. (4) EAP's relaxed durability model is designed for memory persistence where it is important to classify data and metadata for maintaining correctness of the heap and the application state, unlike [35]. Finally, (5) our mechanisms are driven by the current energy availability, with flexibility to switch between the performance and energy efficient modes.

3. DECONSTRUCTING ACID ENERGY COST

To understand the energy costs of memory persistence, we analyze the CPU and NVM energy usage of correctness and durability ACID components. We use the analysis to formulate EAP's energy efficient durability principles and ACI-RD design. All our memory persistence applications use only the NVM [32] as opposed to a DRAM-NVM hybrid model. We report the increase in CPU instructions, NVM writes, and CPU and NVM energy. For NVM energy, we use the PCM energy prediction values as discussed in [28] with a 36x higher active write energy compared to DRAM. Although we use PCM for analysis, EAP's software principles are generic with its main focus toward reducing the necessary parameters such as CPU and NVM use. To validate the benefits of EAP's principles for other competing NVM technologies such as STT-RAM (only 5-10x higher write energy than DRAM), in Section 6, we run the same applications only on DRAM, and measure the full system energy using a power meter.

3.1 Component-level energy analysis

For memory persistence with ACID guarantees, multiple user-level and system-level software components are required. The user-level components include the actual application code, the persistent allocation and garbage collection component, and the transaction component with logging support. System-level components can be a persistent filesystem or persistent memory management mechanisms [11, 23], each maintaining ACID properties for their internal system-level persistent data and metadata structures.

Model. The total energy consumed by a persistent application is the sum of energy used by each of the ACID components that order, flush, fence, and log their states. Our energy-aware optimizations, therefore, address these components and their joint operation. Stated more precisely and focusing on the major contributors of energy consumption – CPU and NVM, the following simple equations denote the total energy used by an application.

$$\left.\begin{aligned} E_{total} &= E_{APP} + E_{datlog} + E_{metalog} + E_{flush} \\ EA = E_{datlog} + E_{metalog} + E_{flush} &= E_{total} - E_{APP} \\ EA &= EA_{CPU} + EA_{NVM} \end{aligned}\right\} \quad (1)$$

E_{APP} denotes energy without ACID, E_{flush} - the energy from cache flush, fence and drain, E_{datlog} - the energy from data logging, $E_{metalog}$ - the energy from metadata logging, and EA - the energy for maintaining ACID guarantees.

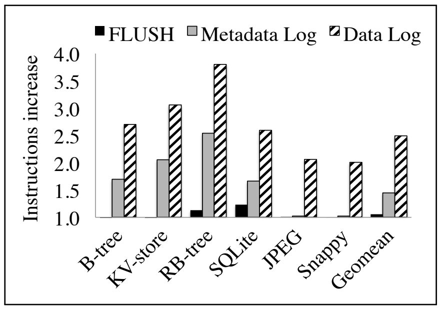
NVM emulation and analysis approach. We use an x86 Haswell desktop system running Linux 3.9.4 kernel. The system has a 32KB L1 and 4MB LLC write-back cache, Intel 520 120GB flash memory, and 4GB of DDR3-based DRAM, of which we use 2GB for NVM by mounting the PMFS filesystem [14]. Because byte addressable NVMs such as PCM are not commercially available, to emulate NVM's read and write latency, we inject software delays similar to most of the prior NVM research [39, 11]. Our emulator periodically gets the total load and store cache misses (every 100ms) using the RDTSCP synchronous instruction as a timer with +-20ns accuracy. It uses the delay model proposed by Dulloor et al. [14] to emulate 100ns load and 400ns store latency. For the CPU and NVM energy, we use the RAPL support [18] to first measure the CPU and DRAM energy consumption, and then use the LLC cache misses due to load and store instructions to estimate the NVM energy usage based on the read and write energy values discussed in [28]. In Section 6, we describe the details of our dynamic energy estimation method. We focus our analysis on the fundamental elements necessary for performance and energy such as the increase in CPU instructions and NVM accesses from memory persistence with ACID guarantees. In Section 6, we show that our analysis and solutions are applicable even when the NVM latency and energy cost are same as DRAM.

We extend and optimize the SNIA standard-based [4] NVML library from Intel [21]. NVML already supports transactional object persistence, persistent memory allocations, logging, and persistent barrier. The persistent objects are stored in a large memory-mapped region managed by the PMFS filesystem. NVML provides an API for an UNDO log, as shown in Figure 2, and rolls back failed transactions. We also extend NVML with a support for WAL-based logging. Further, each persistent object has a different virtual address but one unique identifier across application restarts. The unique identifier is used for loading the object from a persistent region.

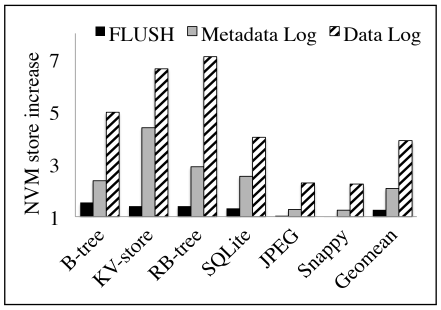
Applications. Table 1 shows the end-user-centric persistence bench-

Applications	Description	Workload
RB-tree [21]	Red-black trees (cache & memory inefficient)	500K random insert, read, delete operations
B-tree [21]	Persistence-friendly, balance search tree used in databases, software cache etc.	Same as RB-tree
KV-store	Simple key-value store using persistent hashmap	Same as RB-tree
SQLite [5]	Database used extensively in end-user devices	500K operations of SQLite benchmark [15]
Snappy [16]	Fast compression library used in chrome and other Google products. We ported it with memory persistence	1.5GB of image, video, audio, document files
JPEG [3]	Well know image conversion service that we ported with memory-based persistence	40K JPEG images converted to bitmap format

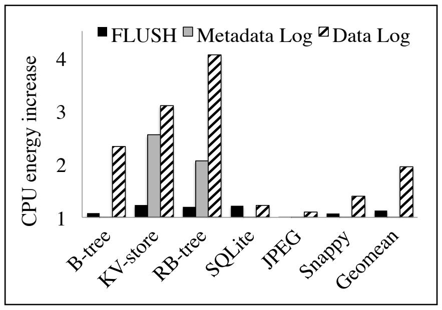
Table 1: Applications.



(a) CPU instructions increase.



(b) NVM stores increase.



(c) CPU energy increase.

Figure 3: ACID cost analysis. Bars show cost of each component. Y-axis shows increase factor relative to No-ACID as the baseline.

marks [30] and applications used in prior studies [30, 23].
(1) B-tree (balance search tree) is a well-known persistence-efficient data structure extensively used in databases, large graph libraries, and also for memory management [21]. B-tree has an O(log n) update or search worst case complexity, and O(n) space complexity. Each B-tree node can have one or many child nodes which reduce the depth of the B-tree. The parent and child nodes can fit inside one cache line reducing the overall cache misses. However, node additions and deletions require re-balancing the tree. Consequently, a memory persistence implementation has to log and commit the changes inside a transaction, and the cost of persistence increases the energy cost significantly.
(2) RB-tree (red-black tree) is extensively used for in-memory data structures inside the OS with an O(log n) insert and O(n) worst-case space complexity. RB-trees are both memory and persistence inefficient because every update to a tree node results in flushing and logging both the parent node and its two child pointers.
(3) A key-value store (KV-store), like those used as a data cache in both end-user and server platforms, is designed using a persistent hashtable. For adding a KV-store entry, first, a hash entry, a key, and its value are allocated using a persistent memory allocator, and the allocations are logged (metadata logging). Next, the data of the key, value and the hash entry pointers are updated and logged. Hence, the persistent KV-store also has a high ACID persistence cost.
(4) The SQLite database is widely used in end-user platforms. It supports both write-ahead logging (WAL), journaling and rollback (referred to as UNDO in this paper). We use the existing in-memory database and logging feature of SQLite with page-based (as opposed to an object-based) persistence to avoid significant code changes to the application.
Finally, we use (5) Google's Snappy object compression library, and (6) JPEG – an image conversion library widely used across different OSes.

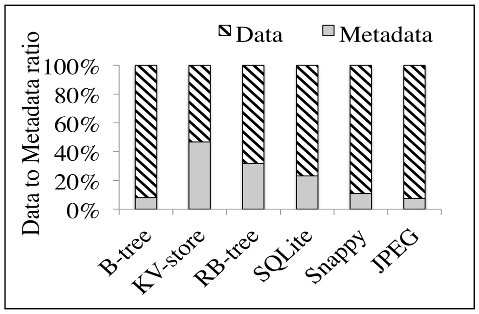
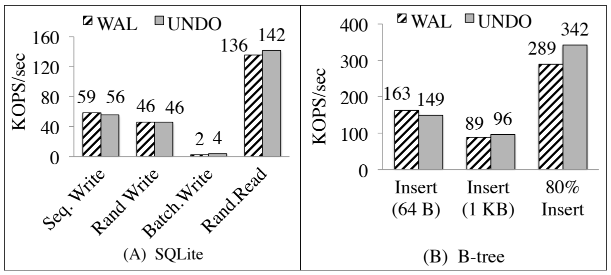


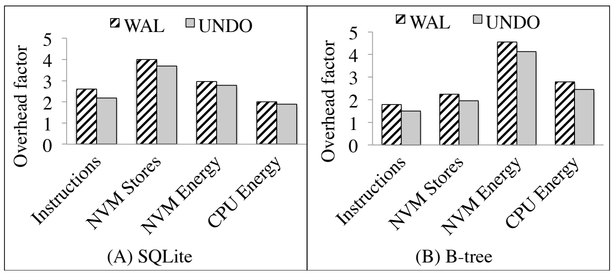
Figure 5: Total data vs. metadata log size ratio.

Analysis. Figures 3.(a)-(c) show the increase in CPU instructions, NVM writes, and CPU energy for applications listed on the x-axis compared to the No-ACID approach that does not offer correctness or durability guarantees. The FLUSH bar refers to persistent barrier cost that includes cache flush and memory ordering (fence and drain operations) instructions [4]. Metadata and Data Log refer to the application and allocator metadata, and data logging cost, respectively.

As the graphs show, supporting ACID for NVM memory persistence increases CPU instructions, NVM accesses, and energy significantly, with durability (metadata and data logging) dominating the cost. The overall CPU energy increases by up to 6.1x for RB-tree of which the data logging alone adds 4.1x overheads. The NVM store increase also shows a similar trend but with even higher overheads. For instance, both RB-tree and KV-store incur 7x more NVM writes. It is important to note that, for KV-store, the metadata to data size ratio is higher compared to all the other application benchmarks as shown in Figure 5. This is primarily due to higher persistent memory allocation and garbage collection cost including the application metadata cost for the reasons we discussed earlier. Therefore, the metadata-related CPU energy and NVM stores in-



(a) WAL vs. UNDO Kilo operations/second for access patterns.



(b) WAL vs. UNDO Instruction, CPU energy, NVM access. Y-axis is overhead increase factor relative to No-ACID.

Figure 4: WAL vs. UNDO performance and energy comparison.

crease by 2.5x and 3.72x, respectively. Snappy and JPEG have relatively lower metadata logging cost as they process less than 50K files. In contrast, the data logging cost is high mainly from logging large multimedia objects.

3.2 Deciphering durability costs

We next analyze the sources of application data and metadata logging costs. We use the resulting insights to formulate a set of EAP principles for reducing energy usage.

Logging methods. Prior research on NVM memory persistence have used either (1) UNDO (refers to journaling) logging [11] or (2) write-ahead logging (WAL) [39]. Also, to exploit byte addressing capability, the memory persistence research use word or object-based logging unlike page-based logging in disk-based systems. We next discuss the energy implications of such logging methods.

UNDO vs. WAL logging for memory persistence. When using UNDO logging, the original data is backed to a journal (log) in the NVM before modifying the data in-place. After a transaction commits successfully, the log is discarded, or else, if the transaction aborts or if the system fails, the backed up log is used to revert the intermediate updates. Although UNDO requires double writes to NVM for each transaction – first to UNDO journal, and then to actual data address, it allows in-place writes and read-after-writes. In contrast, WAL reduces the double write bottleneck by first appending updates directly to the log, and when the log space runs out, it checkpoints the log contents to the original data location. However, because updates are not in-place, subsequent writes and read-after-writes inside a transaction have to be redirected to a log. To read from the log, WAL maintains an index to locate and fetch the latest version. WAL, therefore, sequentially appends writes, which can improve performance and concurrency for large multi-core systems with write-intensive workloads. However, for read-intensive workloads, access redirections can be expensive in terms of CPU instructions and NVM accesses. Additionally, sequential updates are significantly beneficial for disks, but the gains are limited for NVMs [32, 14]. Redirecting every access to a log can result in significant code changes, eliminating the use of the byte-addressable load-store interface.

As discussed by [32, 14], another drawback of WAL for large data updates is that the fixed size log buffers have to be frequently truncated, and their contents have to be checkpointed to the original data location. Checkpoints require parsing the log records sequentially and copying multiple versions of the same data to the original location. Although the updates are faster with WAL log appends, eventually all log entries should be committed, and hence this does not change the total CPU instructions or NVM access. These issues are relevant for page-based logging mechanisms too [33, 10].

Analysis. We analyze the performance and energy impact of UNDO and WAL logging for the B-tree benchmark that uses NVML's object-based memory persistence, and for SQLite that uses page-based persistence. The system setup and NVM emulation are same as discussed earlier. Figure 4a compares the performance (throughput) of WAL and UNDO for B-tree and SQLite. The y-axis represents thousands of operations per second, and the x-axis shows different access patterns. For small sequential, write-intensive workload as in the case of SQLite and All-insert for B-tree, WAL performs marginally better than UNDO. However, for the read-intensive workload (20% writes, 80% reads) and large data updates for B-tree, the UNDO performance is better, and large SQLite updates show similar trends. Next, regarding the energy use, Figure 4b shows the increase in CPU instructions, NVM writes, and energy for SQLite and B-tree when using WAL and UNDO. The y-axis shows the increase factor compared to the No-ACID (and no logging). The y-axis values are for an entire benchmark (cumulative) run with different access patterns. We observe that, compared to WAL, UNDO logging reduces CPU instructions, NVM writes and energy usage for both B-tree and SQLite. *In short, the performance and energy of logging methods vary based on the workload and implementation. Hence, a mechanism that can switch between energy and performance modes is important.*

Metadata durability cost. Concerning the energy overheads associated with the metadata persistence, the persistent memory allocators' state has to be logged for correctness and durability, which can increase the energy use for applications that frequently allocate and deallocate data structures (e.g., KV-store, B-tree). Our prior research [23] proposes an NVM write-aware allocator that reduces NVM writes by placing complex allocator data structures in DRAM and just maintaining a log of all allocations in the NVM. However, even with this approach, for small and frequent NVM allocations, logging and flushing the allocator state can become expensive, especially when the $metadata/data$ ratio increases. As a result, the CPU instructions, NVM accesses, and metadata-related energy cost increase.

4. ENERGY EFFICIENT ACID PRINCIPLES

Based on the insights gained from the analysis of memory persistence ACID overheads, we next formulate a set of energy efficient durability (logging) principles. The principles discussed are the first step towards energy reduction under low-but-not-critical energy budgets for end-user devices. In the next section, we propose a relaxed durability model (ACI-RD) for critically low energy state.

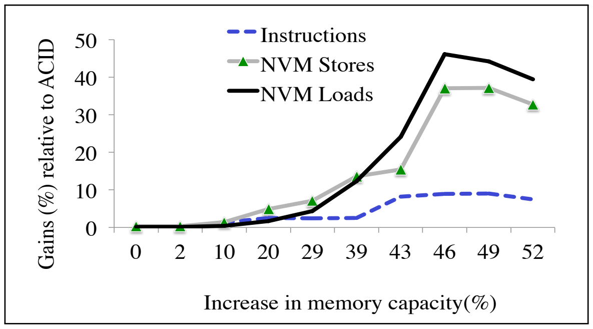


Figure 6: B-tree: y-axis shows the gains relative to ACID by trading memory capacity (x-axis in %).

Instructions	NVM Loads	NVM Stores
11.86%	9.28%	10.68%

Table 2: SQLite gains from trading 35% higher capacity.

4.1 Flexible logging

As discussed in Section 3, although WAL provides marginally higher throughput compared to the UNDO log method for small updates and write-intensive workloads, it increases CPU and NVM energy usage for large and read-intensive workloads.

Key idea. Motivated by these observations, EAP provides a dynamic logging mechanism that transparently switches to an energy-aware logging mode (WAL to UNDO, and vice versa). When energy is not a constraint, applications start with WAL as a default logging mode. At fixed time intervals (epochs), the logging library measures the available energy budget. When energy availability becomes limited, the library switches to UNDO logging. The dynamic switch from WAL to UNDO is initiated only for new transactions because splitting the log for a dirty and uncommitted transaction across WAL and UNDO is suboptimal for logging and recovery. Additionally, all the pending WAL transactions (including nested transactions) are committed before the switch. These restrictions simplify code changes required for SQLite to support EAP's energy-aware methods. We evaluate the additional energy overheads of frequent switching due to small epochs in Section 6.

4.2 Gain energy by trading capacity

We next discuss the principles for reducing persistent memory management energy costs by trading capacity when energy availability is limited.

Reduce allocator work. Modern DRAM and persistent memory allocators strive to provide fast allocation and reduce memory fragmentation. They maintain memory object slabs of different sizes (generally in powers of two), and service request from the slabs. To align requests to the nearest slab, allocators frequently perform complex operations such as merging multiple small objects. When ineffective, they request the OS (via *mmap()* or *break()*) to allocate a batch of pages, and these operations consume significant CPU energy. Requests for a smaller batch result in frequent OS requests, whereas larger batch increases memory fragmentation. The persistent allocator state must also be persisted, which adds to the energy cost. Hence, to reduce energy (and instructions and NVM accesses) usage, EAP's energy efficient durability trades off capacity by using large OS allocations (64MB) only when there is an energy constraint. Consequently, complex merge operations are avoided, reducing CPU work and energy.

Reduce garbage collection overheads. More than the memory allocation, the cost of persistent memory garbage collection is even higher. Most garbage collection methods use a 'mark and sweep' approach (mark, and then delete objects). Prior research have analyzed the performance overheads of DRAM-based garbage collection in end-user devices [34] and server machines [19]. For NVMs, the allocator-related persistence cost is even higher because the allocator has to persist (and log) the allocator state for deletion (*free()*) before requesting the OS to release memory pages using an expensive *munmap()* call. Hence, when the energy is a constraint, EAP delays the garbage collection, and therefore, trades NVM capacity for reduced energy consumption, without affecting the correctness. Garbage collection is delayed only until the system swapping threshold is reached. We modify the persistent memory allocator to mark the objects for deletion but to free them only when the available free NVM capacity is below a threshold. We extend this to the OS-level garbage collection for delaying the release of application heap pages by setting a special one-bit page flag.

Analysis. Figure 6 shows the combined effects of energy efficient allocation and delayed garbage collection. The results show reduced NVM load-store accesses and CPU instructions compared to the memory persistence with traditional ACID approach (y-axis in %). The x-axis indicates the increase in NVM usage as a trade-off for energy. Trading off capacity reduces allocator and garbage collection cost thereby reducing CPU instructions and NVM access. It is important to note that the reduction in NVM accesses is higher compared to the decrease in CPU instructions because recycling objects requires several expensive *memset()*, *memcopy()*, and FLUSH operations. Beyond 45% increase in the capacity, the gains reduce because, after reaching the swap threshold, both the persistent memory allocator and the OS have to release memory aggressively. These actions increase the overall work done by both the allocator and the OS, thereby increasing the total CPU instructions and NVM accesses. Furthermore, a subtle but important reason is that the library allocator maintains all objects as nodes in a B-tree, and their lookup or update time increases as we delay the garbage collection. We observe similar trends for SQLite (see Table 2), but the benefits are lower due to a custom page-management.

4.3 Memory persistence group commit

Transactional updates with persistence barriers can be expensive. Grouping smaller transactions into a larger one, referred to as group commit protocol in databases and filesystems [17] is a well-known technique. NVRAMDB mentioned in Section 2 uses a group commit protocol that buffers updates in DRAM and lazily commits logs and data to NVM [36]. However, NVRAMDB is a performance-centric approach and lacks energy awareness. It increases CPU instructions and accesses to DRAM and NVM by adding a DRAM buffer. Instead, we propose an energy-aware group commit protocol for memory persistence.

Key idea. In a traditional ACID design, a persistence barrier (FLUSH for application data including log [36]) is applied twice, once to the UNDO log before an update, and once to the original data after an in-place update. In contrast, in EAP, we apply barriers only twice in a group of transactions instead of every transaction. The first barrier is used when an object is modified for the first time in a group, and the next barrier when the entire group commits. Also, unlike NVRAMDB [36], our design does not buffer the log updates in DRAM but creates a clean, separate undo log when an object is updated for the first time in a transaction group. For subsequent transactions, logging and in-place updates happen without persistence barriers. Figure 7 shows the pseudocode for inserting a node in B-tree. Before the parent node pointers are updated, an UNDO log is created for every child (update or delete operations). For a

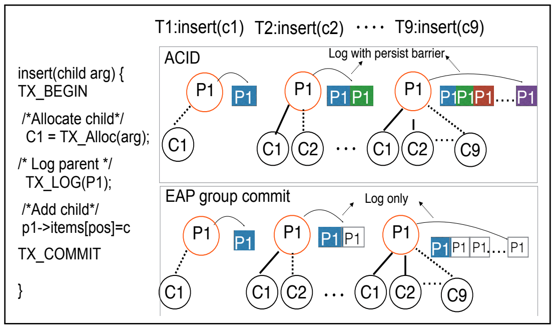


Figure 7: Memory persistence group commit. Circles P, C represent B-tree parent and child nodes. Shaded and non-shaded P1 squares indicate UNDO log with and without persistence barriers, respectively.

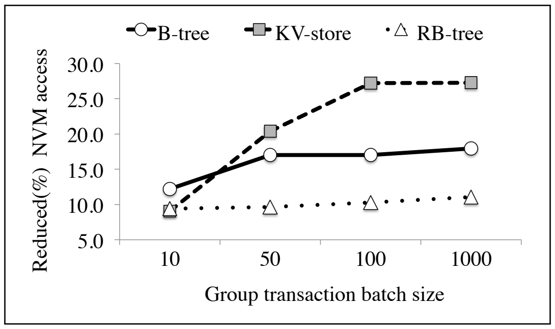


Figure 8: Impact of group commit batch size. Higher object reaccess reduces NVM access.

transaction batch size nine in this example, the persistence barrier for the parent node (P1) is only applied for T1 and T9. If a failure or abort occurs before the group commit, all the updates inside a batch are reverted using the UNDO log. This guarantee of "all or none" is same as the prior group commit approach. We implement this by adding a single bit to the object's allocator metadata and setting the flag first time an object is modified inside a group and resetting the flag when the entire group commits. Figure 8 shows the impact of transaction batch size in the x-axis. Intuitively, the benefits are higher for objects that are repeatedly accessed in the same group. B-tree benefits by reducing the redundant barriers for the parent node when multiple new child nodes are inserted, whereas in the KV-store, repeated barriers for parent hashtable structure is avoided. The energy impact is minimal for RB-tree with fewer reaccess. Hence limiting the group transaction size to increase reaccess per batch is important.

Atomic commits. New persistent memory-specific x86 atomic instructions such as CLWB for cache line write-back without invalidation but ordered store fences, PCOMMIT for non-temporal stores, and CLFLUSHOPT for optimized cache line flush [20] can reduce CPU instruction and NVM access cost by avoiding the need to log values smaller than 64 bytes. This can significantly reduce the metadata logging cost and the data logging cost for small updates.

5. RELAXED DURABILITY – ACI-RD

When the energy budget is critically low, the efficient durability principles discussed in the previous section are not sufficient. In other words, always using a strict ACID approach for end-user devices can substantially drain the battery power preventing an application from running to completion. This impacts the target memory

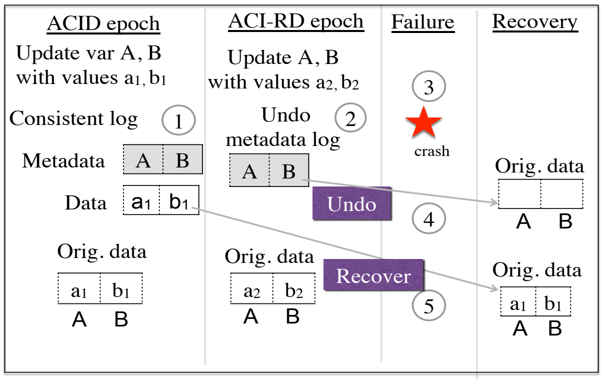


Figure 9: ACI-RD steps. ACI-RD used for critical low energy state.

persistence application, as well as other background applications.

Key Idea. In memory persistence, all application state, including object data, its metadata (object headers), and library metadata (persistent memory allocator state) is logged to a consistent log. In the group commit approach discussed earlier, the metadata and data logs are written for each transaction, and only persistence barriers are relaxed in a group. For critical energy states, when such optimizations are insufficient, we propose a relaxed durability model – ACI-RD. ACI-RD reduces the frequency of logging by increasing the interval between data logging, but without delaying the application, allocator, and the library metadata logging, critical for the correctness of an application. When ACI-RD is enabled, the metadata is updated to a separate UNDO log used only during ACI-RD. Further, an application can transition from a strict ACID phase to ACI-RD phase, and vice versa, depending on the available energy. We refer to each phase as an 'epoch', corresponding to a fixed time interval during which either an ACID or ACI-RD is used. As discussed in Section 2, prior performance-centric delayed durability methods reduce logging cost by buffering log writes in DRAM and eventually writing them to the NVM. However, they do not reduce the overall CPU instructions, NVM writes and the energy usage. In contrast, EAP, to address critical energy state, writes only the metadata log to UNDO thereby reducing NVM writes, CPU instructions, and energy, without compromising the correctness.

Why to use a separate ACI-RD metadata UNDO log? The use of a separate UNDO log for metadata updates during an ACI-RD epoch is critical for application correctness. In the event of a failure, all metadata updates in the UNDO log are used to revert updates during an ACI-RD epoch, and restore the last checkpoint state where both data and metadata were made durable in a consistent log. Specifically, for memory persistence, the UNDO log contains both application- and library-related metadata and provides information for garbage collection of memory allocated in an ACI-RD epoch. Further, the UNDO log clearly segregates the consistent updates in the ACID epoch from the relaxed updates in the ACI-RD epoch. The separate log avoids the need to parse the entire log sequentially and classify the ACID and ACI-RD updates.

Transition from ACI-RD to ACID. During the transition to ACID, EAP first enables load and store fences, flushes all data updates from the cache lines, followed by all metadata updates, and finally, issues a load and store fence, so as to guarantee that all updates from the previous ACI-RD epoch are complete. This provides the same correctness guarantees as ACID. In-flight errors can exist in both ACID or ACI-RD, and can be avoided if the future hardware provides an acknowledgment upon a successful write [8].

ACID-RD steps. Figure 9 shows the update and recovery steps for two variables A and B, for both ACID and ACI-RD epochs. When

```
for each epoch do
    if energy_save_mode = true then
        /*EAP-ACID (efficient durability)*/
        Switch to undo logging
        Apply EAP batch allocation
        Apply EAP delayed garbage collection
        Apply group commit transactions
        Find $\Delta Trans_{epoch}$ from Equation 3
        if $\Delta Trans_{epoch} <= 0$ then
            continue;
        end
        if $energy\_critical = true$ then
            if $commit.size < cacheline.size$ then
                atomic commit;
            end
            if $\Delta Trans_{epoch} > 0 \,||\, commit.size > maxsize$ then
                /* ACI-RD epoch */
                Apply fence
                Update data in-place, FLUSH, drain
                Log metadata to UNDO log
                For transaction commit, return special code
            end
        end
    end
end
```

Algorithm 1: EAP efficient durability and ACI-RD steps.

the variables A and B are updated with values a_1 and b_1 in an ACID epoch with no energy constraints in $\textcircled{1}$, both data (a_1, b_1), and the metadata (address of A, B, and size of update) are written to a consistent log. When the energy becomes a constraint, in $\textcircled{2}$, ACI-RD is enabled. In this case, for updates of the variables A and B with values a_2 and b_2 only the metadata (A, B address, and update size) is written to the ACI-RD UNDO log. The data is updated in-place with FLUSH or atomic updates, if applicable. If a failure happens as in $\textcircled{3}$, first the updates to A and B are reverted using the metadata log as shown in $\textcircled{4}$, and then the consistent data log is used to restore the previous values a_1 and b_1. The result of this mechanism is a trade-off between durability (D in ACID) and energy, without compromising application correctness.

Intuitively, as the ACI-RD epoch time interval increases, the size of the total data that is not made durable increases. Hence, a failure during the ACI-RD epoch increases restart cost. More formally, as shown in Equation 2, after a failure in the ACI-RD epoch, the restart time (R_{ACI-RD_t}) is approximately the sum of the time to undo all updates in the ACI-RD epoch ($UNDOt$), the time to recover data from the consistent log, and the time to re-execute the 'lost' transactions. The time to recover from a consistent log is equal to the restart time (R_{ACID_t}). The time to re-execute the 'lost' transactions is the product of total relaxed transactions ($N_{Trans_{epoch}}$) and the average per-transaction time ($Trans_t$). $UNDOt$ directly depends on $N_{Trans_{epoch}}$, and average UNDO time per transaction ($UNDO_{Trans_t}$).

$$\left.\begin{aligned} R_{ACI-RD_t} &\approx R_{ACID_t} + UNDO_t + (N_{Trans_{epoch}} * Trans_t) \\ UNDO_t &\approx N_{Trans_{epoch}} * UNDO_{Trans_t} \end{aligned}\right\} \quad (2)$$

$$\left.\begin{aligned} \Delta Trans_{epoch} &= Trans_{epoch_i} - Trans_{epoch_{i-1}} \\ \Delta E_{epoch} &= (E_{epoch_i} - E_{budget})/E_{budget} \end{aligned}\right\} \quad (3)$$

To implement the ACI-RD mechanism, given an energy budget E, the application execution is divided into per-epoch intervals of time t_{msec} with per-epoch energy budget E_{budget}. We assume the value of E_{budget} is known and is equal to the per-epoch energy of the metadata only approach in which only the metadata required for correctness is logged. We estimate this budget by sampling an epoch at runtime [13]. After the end of each epoch, EAP estimates the increase in the epoch's energy usage (ΔE_{epoch}) and the increase in the number of transactions ($\Delta Trans_{epoch}$) relative to the current and previous epoch transactions $Trans_{epoch_i}$ and $Trans_{epoch_{i-1}}$. As discussed in Section 3, the total energy for persistent applications with ACID guarantees is a factor of application execution, data logging, cache flush, and metadata transactions. We use this simple estimation model to avoid the overheads (including energy) of a complicated model.

EAP execution. Algorithm 1 shows EAP's sequence of steps for reducing energy usage. EAP's efficient and relaxed durability are activated only when the energy saving mode is enabled – a feature available in most end-user devices. For low energy budget, EAP applies the energy efficient durability principles described in Section 4 – switching from WAL to UNDO logging, group commit, batched allocation, and relaxed garbage collection – while still maintaining ACID guarantees. We refer to this mode as *EAP-ACID*. In the next epoch, the EAP runtime checks if EAP-ACID is sufficient to meet the per-epoch energy budget, and if not, ACI-RD is activated. These steps are repeated for subsequent epochs until the energy budget is satisfied. Note that unlike checkpointing-recovery protocols with fixed checkpoint interval [26], in a transactional application, the logging frequency depends on the number of transactions executed by the application. EAP uses the energy budget to tailor/relax logging without affecting the application correctness. For ACI-RD epochs, atomic commits can be used for the metadata log, as shown in Algorithm 1. Further, durability for objects larger than a threshold size is relaxed as they consume significantly higher energy.

Application notification and restarts. ACI-RD is a feature of EAP that either a user or a developer can enable. When ACI-RD is enabled for critically low energy budget, our implementation notifies the application about the completion of a transaction with a special return code on commits. If a system with ACI-RD epoch encounters failure, then the application re-executes all ACI-RD transaction for the failed epoch upon restart.

6. EAP EVALUATION

The key goal of our evaluation is to understand the impact of the proposed energy efficient durability (EAP-ACID), relaxed durability (ACI-RD), and the overall benefits of EAP that combine EAP-ACID and ACI-RD. We also analyze the implications of ACI-RD on restart time after a failure, and the overall impact on the system energy.

Methodology. In Section 3, we described our experimental platform details, NVM emulation with software delay, and the use of the RAPL [2] hardware counters to estimate the NVM and the CPU energy. We dynamically query the performance and energy hardware counters every 100ms (minimum frequency to measure energy) to measure the increase in CPU instructions, NVM accesses, and CPU energy. We estimate the NVM energy usage using the load-store cache misses [39, 22]. The latency and energy values are based on a PCM-based NVM [28], but EAP's principles for reducing energy are relevant for other NVM technologies such as STT-RAM. We also validate this by running the applications in a DRAM-based system and measuring the overall system energy using a power meter.

Baselines. We evaluate and compare EAP, a system which combines EAP-ACID and ACI-RD, against four different persistence

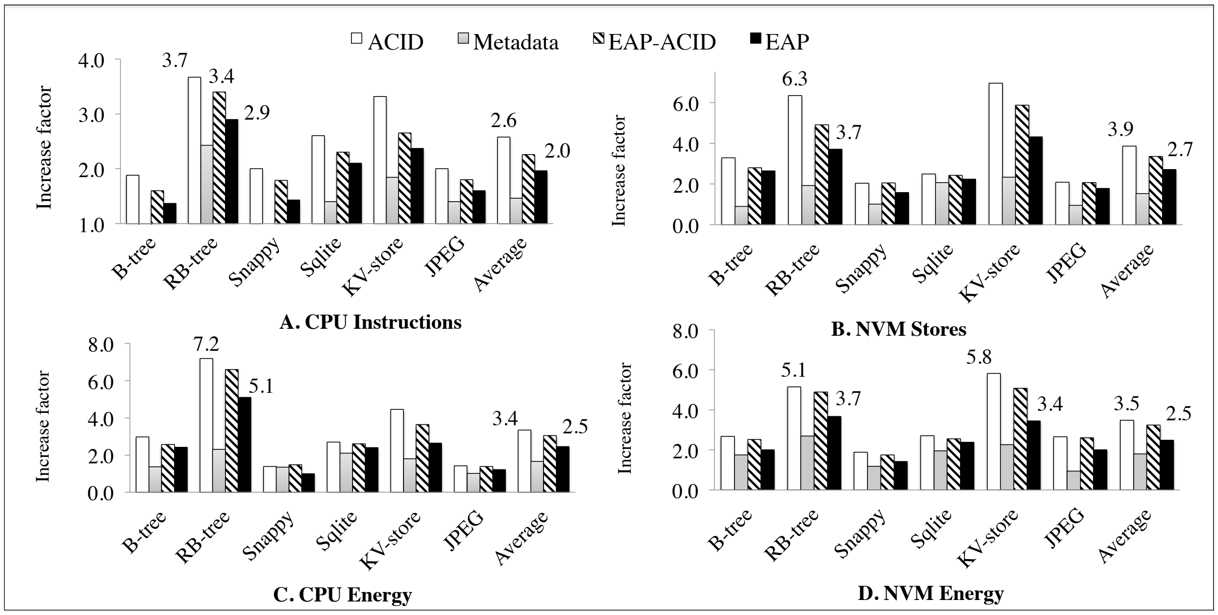


Figure 10: EAP impact on CPU instructions, NVM writes, CPU and NVM energy. Y-axis in the graphs denotes increase factors(x) relative to using only FLUSH. EAP bars include EAP-ACID and ACI-RD.

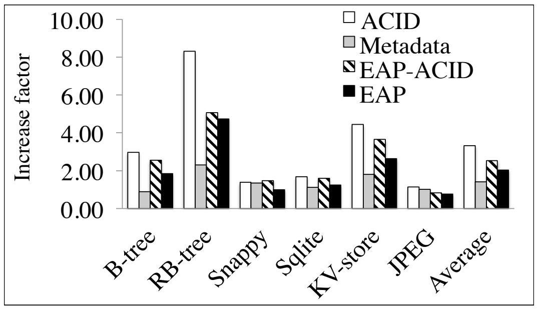


Figure 11: EAP runtime impact.

methods. First, the baseline method – FLUSH – that performs only flush, fence, and orders data in the cache without satisfying durability and correctness. Second, the traditional ACID approach with full data and metadata logging. Third, the Metadata-only method that provides durability only for the allocation and library metadata without the data durability. This method cannot recover any data after a failure and only validates the correctness, and hence, is not useful for most applications. Fourth, the EAP-ACID, which provides efficient durability via flexible logging, energy-aware allocation and garbage collection and the group commit mechanism. Note that unlike EAP, this approach does not relax durability and is applied for low-but-not-critical energy states.

6.1 Reduced energy use with EAP

Figures 10.(A)-(D) analyze the implications of persistence on increase in CPU instructions, NVM writes, CPU energy, and NVM energy, respectively. The y-axis shows the overheads as factor increase relative to the FLUSH approach as a baseline. The experiments use the applications and benchmarks introduced in Section 3. Application runtimes vary between 28 and 64 seconds. In EAP-ACID and EAP, the epoch time is set to 400ms, and we later discuss the epoch interval sensitivity. All the three benchmarks – B-tree, RB-tree, and KV-store – perform insert, find, delete and overwrite operations with 128-byte values.

Analysis. Although B-tree is persistence friendly, ACID increases CPU instructions by 1.9x and NVM accesses by 3.28x compared to the baseline. EAP-ACID with its energy efficient optimization reduces instructions and NVM accesses by 30% and 48% respectively compared to ACID. These benefits are mainly from the memory persistence-based group commit that reduces multiple persistent barriers for the same parent when child nodes are inserted, and from trading capacity by reducing the overhead of garbage collection. In contrast, EAP, by using ACI-RD with 400ms epoch intervals further reduces CPU and NVM energy by 61% and 67%, respectively.

RB-tree is a persistence inefficient data structure and shows the highest the ACID cost. When using only EAP-ACID, the energy reduction gains are minimal (~28% reduction in the CPU energy), most of which are from reducing the garbage collection cost and avoiding redundant log updates using the group commit. In contrast, with EAP, the logging is relaxed for RB-tree updates. Specifically, relaxing a log update for a parent node in the RB-tree avoids the logging cost for two of its child nodes. Consequently, this reduces instructions and NVM access by 80% and 2.6x, respectively. The Metadata-only approach offers higher gains but is less useful. For the KV-store, EAP reduces the high CPU and NVM energy costs of ACID by 2x and 2.5x, respectively. EAP still exhibits high overheads because of the allocator cost. Redesigning the KV-store data structure can reduce such cost.

For Snappy [16], which uses an input workload of 50K files (total 1.5 GB), and file sizes that vary between 10KB-140MB, the metadata logging cost is negligible. However, persisting large files with FLUSH is expensive, and the data logging is even more expensive. EAP relaxes logging for large data updates (see Algorithm 1), which provides considerable improvement. For applications such as Snappy, strict ACID is unnecessary because only files for which compression failed require re-compression. EAP provides a transparent support without depending on the application developer. JPEG image conversion [3] shows similar trends with only 40K transactions but incurs higher number of NVM store accesses because the output files can be larger (up to 30%) adding to

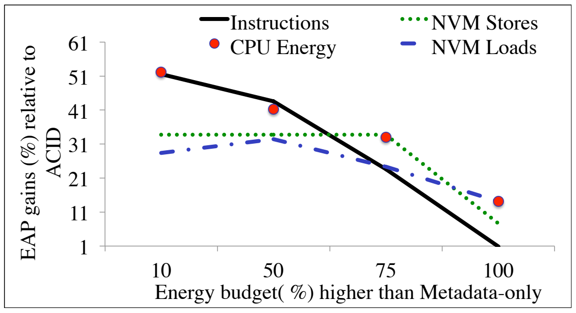


Figure 12: Impact of energy budget.

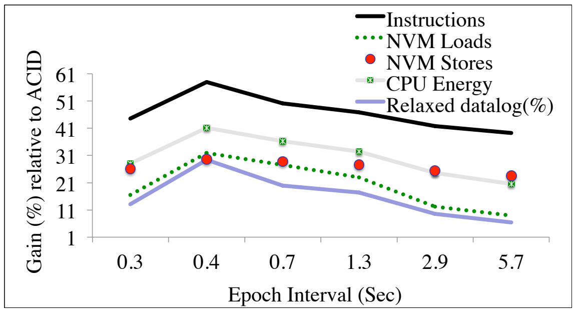


Figure 13: Impact of epoch interval.

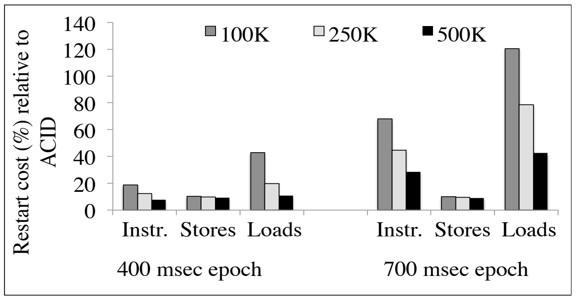


Figure 14: Epoch interval vs. restart cost. Load and stores represent NVM load stores.

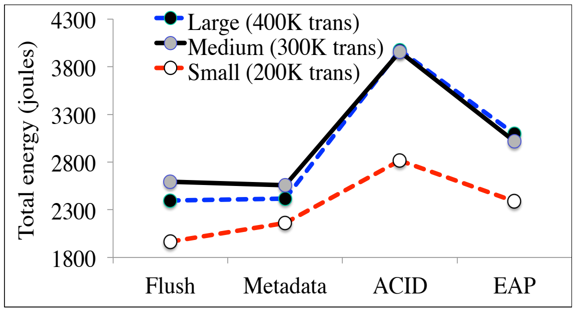


Figure 15: Overall system energy improvements with EAP. Y-axis indicates cumulative energy from running all applications.

the logging cost. With SQLite, EAP-ACID's flexible logging and allocator optimization reduce instructions (16%) and NVM writes (12%), compared to ACID with WAL-based logging. ACI-RD reduces CPU and NVM energy by additional 26-28%. Furthermore, using a page-based persistence limits the gain [33], and redesigning SQLite for memory persistence in the future can improve the benefits. Figure 11 shows the runtime impact of EAP. As shown, EAP, by combining efficient durability with ACI-RD, reduces CPU instructions and NVM writes, thereby improving application runtime. The runtime results show the same trends as the other system parameters.

6.2 EAP parameter sensitivity

We next use the persistence-efficient B-tree benchmark to analyze the sensitivity of the energy gains and the restart cost for EAP, by varying the energy budget and the epoch interval.

Energy budget. Figure 12 shows the impact of the energy budget on the gains that can be achieved with EAP. The x-axis shows the increase (%) in the budget compared to the energy required for the Metadata-only case. The y-axis shows the gains compared to strict ACID. Intuitively, when the budget is high, the number of ACI-RD epochs decreases along with EAP's energy gains. As observed, for 0 to 45% increase in the budget, the CPU instruction and energy reduction gains vary from 56% to 40%, and the NVM accesses vary from 36% to 31%. These results show (1) the significant difference between the ACID and the Metadata-only case, and (2) the need to use both efficient durability (EAP-ACID) and ACI-RD for most epochs when the energy budget is low. When increasing the budget beyond 50%, the number of ACI-RD epochs decreases, and at 90% EAP is turned off to use the strict ACID.

Epoch interval. Figure 13 shows the sensitivity to the epoch interval duration. Note that after each epoch interval, EAP's Algorithm 1 is used to decide if EAP-ACID or ACI-RD should be applied. In addition to the reduction in CPU instructions and NVM accesses, we also show the percentage of the total data log size that is relaxed, compared to the ACID case. We observe the following. First, for B-tree, EAP achieves maximum gains for epoch intervals in the 400-700ms range (10-50K B-tree transactions). Reducing the epoch interval below 400ms or increasing it above one second reduces the EAP's gains. The reduction is because for short execution intervals, the overhead of switching from ACID to EAP or from EAP to ACID dominates the benefits achieved from EAP. The costs include switching to an energy efficient logging mode and EAP's software budget profiling cost, which increase CPU instructions by 10%. When increasing the epoch interval beyond one second, EAP fails to apply ACI-RD when the energy budget exceeds the epoch budget, thus incurring higher data durability cost, as shown in the figure. *These results confirm the high data logging cost, and show that even when relaxing the data logging interval to 400ms, reductions of up to ~60% CPU instructions and ~40% NVM accesses can be achieved with EAP.*

Restart costs. As discussed in Section 5 and Equation 2, the restart cost is dependent on the epoch interval for which data durability is relaxed. For brevity, in Figure 14, we show only the impact of epoch intervals of 400ms and 700ms. The y-axis shows the increase in restart cost compared to strict ACID for 100K, 250K, and 500K transactions. For a 400ms epoch interval with 100K transactions, around 5K transactions are relaxed. For restarts after a failure, the CPU instructions and NVM accesses increase by only 19% and 10%, respectively. For 500K transactions, the CPU instructions and NVM accesses overheads further reduce mainly because the restart cost from loading the UNDO log and other initialization costs amortize. For ACI-RD epoch of 700ms, the additional UNDO operations increase the restart costs. The results show that using an optimal epoch interval (400ms in this case) is important to reduce the increase in restart cost due to ACI-RD.

It is also important to note that the overall duration of a restart is 59x lower than the total application execution time, and the energy benefits of EAP are significantly higher compared to the increase in the restart cost. The results show the sensitivity of EAP's gain towards the epoch interval and energy budget.

6.3 Memory technology independent gains

To validate EAP's principles independent of the NVM technology, we run all applications on DRAM for storage and computation, without emulating PCM-based NVM. We measure the overall system energy consumption by connecting the power source to a power meter. In Figure 15, the y-axis shows the overall energy usage (in joules) when running all applications (B-tree, RB-tree, KW-store, JPEG, and Snappy). The number of total transactions per application is varied (large, medium, and small) by changing the problem or the input data size. The x-axis indicates different persistence methods. For EAP, we use a 400ms epoch interval – the best case in the previous evaluation result. As seen, the overall energy increase from strict ACID memory persistence shows same trends as in the earlier NVM and CPU energy results. For large and medium transactions, ACID increases energy use by 2x compared to the FLUSH only approach. EAP, by combining EAP-ACID and ACI-RD, reduces energy consumption by ~34% compared to the ACID approach. *The results validate EAP's generic energy reduction principles, regardless of the persistent memory technology.*

Summary and Discussion. In summary, EAP, by combining EAP-ACID and ACI-RD provides up to 2.4x reduction in NVM energy and 2.1x reduction in CPU energy for data structures that are not persistence friendly (RB-tree), and up to 64% and 40% CPU and NVM energy gains for persistence-efficient applications (B-tree). Although EAP-ACID is effective for typical systems, for systems with critically low energy budget, ACI-RD may be required to ensure application completion. EAP does not compromise correctness. Support for multi-cache line commits and hardware-based energy profiling can reduce metadata and software energy cost.

7. CONCLUSIONS AND FUTURE WORK

This paper analyzes the energy overheads of persistence, and identifies durability-related costs as the most significant contributor to energy usage. To reduce durability energy costs, we propose energy-aware persistence (EAP). EAP first designs energy-efficient durability principles that under low but not critical energy budgets include flexible logging to trade off a small fraction of performance with energy, delayed garbage collection to trade NVM capacity with energy, and use of an NVM group commit method. For critically low energy conditions, EAP proposes a relaxed durability (ACI-RD) design that reduces energy significantly by relaxing data logging, but without impacting correctness. EAP's evaluation using benchmarks and applications shows a reduction of up to 2.1x in CPU energy and 2.4x in NVM energy, compared to a strict ACID approach. An interesting outcome of this research is that, for reducing persistence cost, it is important to reduce both the CPU and NVM energy. As a future work, we plan to understand the failure modes of end-user devices and to support alternative durability models.

8. ACKNOWLEDGMENT

The authors would like to thank all the anonymous reviewers and our shepherds, Professor Yan Solihin and Professor Larry Rudolph, for their valuable feedback in improving the quality of the paper. We thank the immense contribution of Professor Karsten Schwan whom we will greatly miss. This work was supported in part by the Intel URO program on software for persistent memories, and by C-FAR, one of the six SRC STARnet Centers, sponsored by MARCO and DARPA.

9. REFERENCES

[1] Intel-Micron Memory 3D XPoint. http://intel.ly/1eICR0a.

[2] Intel RAPL driver. http://lwn.net/Articles/545745/.

[3] JPEG library. http://libjpeg.sourceforge.net/.

[4] SNIA specification. http://tinyurl.com/nktmrby.

[5] SQLite. http://www.sqlite.org.

[6] J. Arulraj, A. Pavlo, and S. R. Dulloor. Let's talk about storage & recovery methods for non-volatile memory database systems. In *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data*, SIGMOD '15, pages 707–722, New York, NY, USA, 2015. ACM.

[7] H. Avni, E. Levy, and A. Mendelson. *Hardware Transactions in Nonvolatile Memory*, pages 617–630. Springer Berlin Heidelberg, Berlin, Heidelberg, 2015.

[8] A. M. Caulfield, A. De, J. Coburn, T. I. Mollow, R. K. Gupta, and S. Swanson. Moneta: A high-performance storage array architecture for next-generation, non-volatile memories. In *MICRO 2010*, pages 385–395.

[9] D. R. Chakrabarti, H.-J. Boehm, and K. Bhandari. Atlas: Leveraging locks for non-volatile memory consistency. In *Proceedings of the 2014 ACM International Conference on Object Oriented Programming Systems Languages & Applications*, OOPSLA '14, pages 433–452, New York, NY, USA, 2014. ACM.

[10] J. Coburn, T. Bunker, M. Schwarz, R. Gupta, and S. Swanson. From aries to mars: Transaction support for next-generation, solid-state drives. In *SOSP 2013*, pages 197–212.

[11] 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. In *ASPLOS, 2011*, pages 105–118.

[12] 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 *SOSP, 2009*, pages 133–146.

[13] I. Constandache, S. Gaonkar, M. Sayler, R. Choudhury, and L. Cox. Enloc: Energy-efficient localization for mobile phones. In *INFOCOM 2009*, pages 2716–2720, 2009.

[14] S. R. Dulloor, S. Kumar, A. Keshavamurthy, P. Lantz, D. Reddy, R. Sankaran, and J. Jackson. System software for persistent memory. In *EUROSYS, 2014*, pages 15:1–15:15.

[15] Google. LevelDb. http://leveldb.org/.

[16] Google. Snappy Compession. http://tinyurl.com/ku899co.

[17] R. Hagmann. Reimplementing the cedar file system using logging and group commit. In *Proceedings of the Eleventh ACM Symposium on Operating Systems Principles*, SOSP '87, pages 155–162, New York, NY, USA, 1987. ACM.

[18] M. Hähnel, B. Döbel, M. Völp, and H. Härtig. Measuring energy consumption for short code paths using rapl. *SIGMETRICS Perform. Eval. Rev.*, 40(3):13–17, Jan. 2012.

[19] M. Hertz and E. D. Berger. Quantifying the performance of garbage collection vs. explicit memory management. In *OOPSLA, 2015*, pages 313–326.

[20] Intel. Intel Development Manual. http://intel.ly/1CdHj1r.

[21] Intel. Logging library. https://github.com/pmem/nvml.

[22] Intel. PMFS: Persistent memory file system. github.com/linux-pmfs.

[23] S. Kannan, A. Gavrilovska, and K. Schwan. Reducing the cost of persistence for nonvolatile heaps in end user devices. In *HPCA, 2014*, pages 512–523.

[24] S. Kannan, A. Gavrilovska, and K. Schwan. pvm: Persistent virtual memory for efficient capacity scaling and object storage. In *Proceedings of the Eleventh European Conference on Computer Systems*, EuroSys '16, pages 13:1–13:16, New York, NY, USA, 2016. ACM.

[25] S. Kannan, A. Gavrilovska, K. Schwan, and S. Kumar. Nvm heaps for accelerating browser-based applications. In *Proceedings of the 1st Workshop on Interactions of NVM/FLASH with Operating Systems and Workloads*, INFLOW '13, pages 8:1–8:8, New York, NY, USA, 2013. ACM.

[26] S. Kannan, A. Gavrilovska, K. Schwan, and D. Milojicic. Optimizing checkpoints using nvm as virtual memory. In *Parallel Distributed Processing (IPDPS), 2013 IEEE 27th International Symposium on*, pages 29–40, May 2013.

[27] H. Kim, M. Ryu, and U. Ramachandran. What is a good buffer cache replacement scheme for mobile flash storage? *SIGMETRICS Perform. Eval. Rev.*, 40(1):235–246, June 2012.

[28] B. C. Lee, E. Ipek, O. Mutlu, and D. Burger. Architecting phase change memory as a scalable dram alternative. In *ISCA, 2009*, pages 2–13.

[29] R.-S. Liu, D.-Y. Shen, C.-L. Yang, S.-C. Yu, and C.-Y. M. Wang. Nvm duet: Unified working memory and persistent store architecture. In *ASPLOS, 2014*, pages 455–470.

[30] Y. Lu, J. Shu, L. Sun, and O. Mutlu. Loose-ordering consistency for persistent memory. In *ICCD, 2014*, pages 216–223.

[31] I. Moraru, D. G. Andersen, M. Kaminsky, N. Tolia, P. Ranganathan, and N. Binkert. Consistent, durable, and safe memory management for byte-addressable non volatile main memory. In *Proceedings of the First ACM SIGOPS Conference on Timely Results in Operating Systems*, TRIOS '13, pages 1:1–1:17, New York, NY, USA, 2013. ACM.

[32] D. Narayanan and O. Hodson. Whole-system persistence. In *ASPLOS, 2012*, pages 401–410.

[33] G. Oh, S. Kim, S. Lee, and B. Moon. Sqlite optimization with phase change memory for mobile applications. *PVLDB*, 8(12):1454–1465, 2015.

[34] K. Paul and T. K. Kundu. Android on mobile devices: An energy perspective. In *CIT 2010*, pages 2421–2426.

[35] S. Pelley, P. M. Chen, and T. F. Wenisch. Memory persistency. In *ISCA, 2014*, pages 265–276.

[36] S. Pelley, T. F. Wenisch, B. T. Gold, and B. Bridge. Storage management in the nvram era. *Proc. VLDB Endow.*, 7(2):121–132, Oct. 2013.

[37] M. K. Qureshi, V. Srinivasan, and J. A. Rivers. Scalable high performance main memory system using phase-change memory technology. *SIGARCH Comput. Archit. News*, 37(3):24–33, June 2009.

[38] D. Schwalb, T. Berning, M. Faust, M. Dreseler, and H. Plattner. nvm malloc: Memory allocation for nvram. In *VLDB*, 2015.

[39] H. Volos, A. J. Tack, and M. M. Swift. Mnemosyne: Lightweight persistent memory. In *ASPLOS, 2011*, pages 91–104.

[40] C. Wang, S. S. Vazhkudai, X. Ma, F. Meng, Y. Kim, and C. Engelmann. Nvmalloc: Exposing an aggregate ssd store as a memory partition in extreme-scale machines. In *Parallel Distributed Processing Symposium (IPDPS), 2012 IEEE 26th International*, pages 957–968, May 2012.

[41] H. Yoon. Row buffer locality aware caching policies for hybrid memories. In *ICCD, 2012*, pages 337–344.

[42] J. Zhao, S. Li, D. H. Yoon, Y. Xie, and N. P. Jouppi. Kiln: Closing the performance gap between systems with and without persistence support. MICRO-46, pages 421–432, 2013.

[43] J. Zhao, O. Mutlu, and Y. Xie. Firm: Fair and high-performance memory control for persistent memory systems. In *MICRO-47, 2014*, pages 153–165.