Energy Aware Persistence: Reducing the Energy Overheads of Persistent Memory

Sudarsun Kannan, Moinudin Qureshi, Ada Gavrilovska, and Karsten Schwan

Abstract—Next generation byte addressable nonvolatile memory (NVM) technologies like PCM are attractive for end-user devices as they offer memory scalability as well as fast persistent storage. In such environments, NVM's limitations of slow writes and high write energy are magnified for applications that need atomic, consistent, isolated and durable (ACID) updates. This is because, for satisfying correctness (ACI), application state must be frequently flushed from all intermediate buffers, including processor cache, and to support durability (D) guarantees, that state must be logged. This increases NVM access and more importantly results in additional CPU instructions. This paper proposes Energy Aware Persistence (EAP). To develop EAP, we first show that the energy related overheads for maintaining durability are significant. We then propose energy-efficient durability principles that mitigate those costs, an example being flexible logging that switch between performance and energy-efficient modes and a memory management technique that trades capacity for energy. Finally, we propose relaxed durability (ACI-RD) mechanism used under critical low energy conditions that do not affect correctness. The initial results for several realistic applications and benchmark show up to 2x reduction in CPU and NVM energy usage relative to a traditional ACID-based persistence.

Index Terms—NVM, heap-based persistence, ACID, energy overheads, logging

1 Introduction

Industry predicts future byte addressable, nonvolatile memory (NVM) technologies like phase change memory (PCM) to have 100× lower access latency compared to that of Flash/SSD devices and to scale to four-eight times the density of DRAM, without consuming refresh power. The promised outcomes include: (1) larger memory capacities [1], at (2) lower energy usage [2] compared to DRAM, coupled with (3) faster persistent data storage and access than Flash/SSD. Given these properties, end-user devices with limited DRAM capacities and slow flash storage [2] are an attractive target for dual-use of device-level NVM, to gain both high memory capacity (volatile heap) and fast data persistence (storage). Using NVM for memory-based persistent (NVMemPersist) storage, however, requires systems to provide applications with ACID guarantees, comprised of two components: (1) a correctness component (ACI of ACID), and (2) a durability component (D). The correctness component ensures correctness by ordering and flushing the application state and its associated metadata (e.g., persistent allocations), thereby, increases processor stalls with barriers, fence, and flush operations. The durability component logs application data and metadata state to a persistent log in NVM (fastest persistent storage) during a transaction, and when completed, the log can be committed to the actual data location. If a failure happens before a transaction is complete, the log is used to undo/redo changes. These persistence components cause undesirable increases in NVM write energy, and more importantly, CPU energy arising from the software stack enforcing ACID properties.

To explain the difference between metadata and data persistence, Fig. 1 shows an example for persisting a key-value pair in a persistent hashtable. The steps include, (1) allocating a key-value pair with persistent-aware library allocator, (2) logging the allocator metadata required for NVMemPersist, (3) updating the key-value data, and finally, (4) commit. The commit involves (a) first logging the key-value data (known as record), followed by (b) logging the record header which contains a record pointer, size, and index. After a failure, the library metadata log is used to restore application's persistent heap, whereas, the application metadata (record headers) restores data to a consistent state before failure. The energy cost of small fixed size metadata (16 bytes) is typically lesser compared to the data.

Prior NVMemPersist work such as [3], [4], [5], [6] has sought to reduce the ACID-related costs by proposing performance centric models that do not differentiate between updates required for correctness from durability. Pelly et al. [5] propose a consistency model that enables reordering commit sequence from CPU to cache (ordered from cache to NVM) for achieving higher concurrency, whereas Lu et al. [6] proposes a loose ordering consistency (LOC) protocol with an eager commit for relaxing intra-transaction ordering. These eager/relaxed schemes do not reduce the total CPU instructions or NVM accesses and may not solve energy-related problems.

To reduce the energy overheads of persistence, this paper proposes an *energy-aware persistence* (EAP) approach for efficient use of NVM in end-user devices such as smartphones, tablets and laptops. EAP's design decisions made are based on (1) a detailed analysis of the energy implications of the ACID components associated with application persistence. To the best of our knowledge, these analyses are the first (a) to identify that durability-related costs are the most significant contributor to persistence-related energy cost, and (b) to quantitatively demonstrate the significance of the CPU-related versus solely NVM-related costs in the overall ACID energy overheads. EAP leverages these observations to incorporate (2) energy-efficient durability alternatives that trade performance or memory capacity in a controlled manner, to meet the constrained energy budgets of end-user devices. When such optimizations are insufficient, under critically-low energy budgets, EAP (3) further improves energy usage by using a novel relaxed (not delayed) durability model—ACI-RD—that does not affect application correctness with its dynamic awareness to an energy budget, illustrated in Fig. 2. In the traditional ACID design, application data and metadata are always ordered, flushed and logged irrespective of available energy. In contrast, EAP uses traditional ACID (Epoch 1,4) only when the energy availability is sufficient, and when critically low, the data durability is relaxed (Epoch 2,3), but not the metadata that is required for correctness. We discuss the design in Section 3.

2 Deconstructing the Energy Overheads

To understand the energy overheads of persistence, we formulate and use a simple energy model (shown below) to decipher NVM and CPU energy for each ACID component. Briefly, the total energy consumed by an application is the sum of CPU and NVM energy consumed without ACID, plus the sum of data and metadata logging, and flush (flush includes cache flush, fence and barriers)

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

where, E_{APP} denotes application energy use without ACID, $E_{flush}, E_{datlog}, E_{metalog}$ – denotes energy from flush, application data logging, and metadata (application and library) logging respectively. EA_{CPU}, EA_{NVM} denotes CPU, NVM energy from ACID, and their sum EA. Energy-aware optimizations, therefore, must address these components and their joint operation.

• *The authors are with the Georgia Institute of Technology, Atlanta, GA. E-mail: sudarsun@gatech.edu, moin@ece.gatech.edu, {ada, schwan}@cc.gatech.edu.*

Manuscript received 19 Apr. 2015; revised 1 July 2015; accepted 3 Aug. 2015. Date of publication 24 Aug. 2015; date of current version 5 Jan. 2017.
For information on obtaining reprints of this article, please send e-mail to: reprints@ieee.org, and reference the Digital Object Identifier below.
Digital Object Identifier no. 10.1109/LCA.2015.2472410

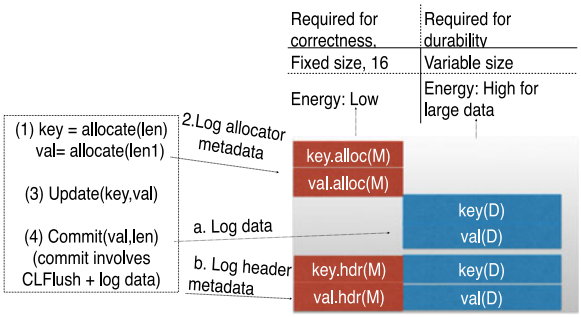

Fig. 1. Hashtable data and metadata logging. M, D indicates metadata and data respectively.

2.1 Component-Level Energy Analysis

We assume all NVMemPersist benchmarks use only NVM (instead of a DRAM-NVM hybrid model) with processor cache [7] and analyze ACID's component-level CPU and NVM energy use. We plan to analyze the energy cost for hybrid architecture in the future. While our energy analysis is for PCM-based NVMs (40× higher write energy than DRAM), the principles for reducing durability energy cost proposed in this paper is applicable for other competing technologies such as STT-RAM (5×-10× higher write energy state than DRAM).

Hardware basis. We use an X86-based Haswell desktop-based system running the 3.9.0 Linux kernel, with 32 KB L1, 4 MB LLC write back cache, Intel 520 120 GB flash memory, and 4 GB DDR3 DRAM out of which 2 GB used for NVM. For emulating the slow-read writes seen in NVMs, we use hardware counters to delay application load/stores with the delay model described in [8]. For analysis, we manually enable each ACID-component, and measure the increase in CPU energy and NVM read/writes using H/W counter, and estimate the NVM read/write energy using [2]. For EAP's dynamic component-level energy optimizations (discussed in Section 3), unlike the manual analysis, to estimate the increase in NVM writes and energy, we use PIN instrumentation only for ACID-component implementation which adds ~1-2 percent noise.

Benchmarks. There are not many publicly available NVMemPersist benchmarks. We use the following persistent benchmarks similar to prior research [6]. The benchmarks exhibit high variance in-terms of NVM access and cache miss rates. *1. Binary tree*, implemented as an extension of [9] with 500 K inserts, lookup and deletion. Note that binary trees are cache and memory inefficient compared to other tree structures. For logging support, we extend and optimize the open source NVM-based logging [9] with ideas discussed in [10], *2. B-Tree*, extensively used in applications such as databases and application caches. We modify the B-Tree to provide heap-based persistence and use the same workload used for binary tree, *3. SQLite*, a well-known traditional query-based transaction database [11] used extensively in end-user platforms. We use the SQLite benchmark suite [12] with 500 K transactions, and

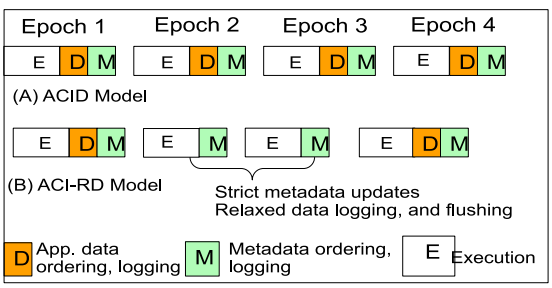

Fig. 2. (A) Traditional ACID, (B) ACID-RD: Data logging relaxed for critically low energy Epoch 2,3.

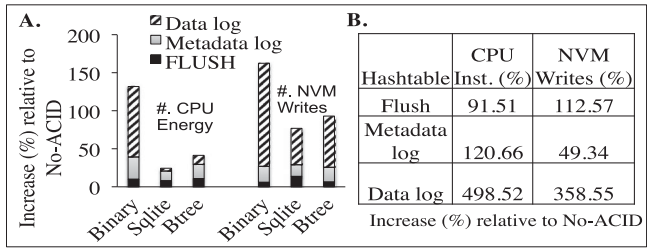

Fig. 3. ACID component analysis.

finally, *4. Hashtable*, used in key-value store-based in-memory caches. Hashtable have high persistence overhead because for each key/value update, at least three persistent allocations (the hash entry, key, value) is required, and for every key-value update, the hash entry, data, and hash structure metadata should be updated.

Fig. 3A and the table in Fig. 3B (extreme case—hashtable) show the increased energy and NVM writes for the benchmarks. The ACID-related components are indicated in the graph, with (1) FLUSH referring to cache flushing, fencing, and memory barriers, (2) Data log referring to application data durability (logging), and finally, (3) the metadata logging required for durability and application correctness. ACID-based persistence increases CPU energy and NVM writes, and more importantly, the durability component contributes up to 92 percent CPU energy and 135 percent NVM writes (358 percent for hashtable) of the overall cost. Interestingly, the metadata cost adds up to 36 percent arising from frequent allocation/free of persistent data that must be logged for correctness.

2.2 Deciphering Durability Costs

We next analyze the source of durability component cost and use the resulting insights to formulate a set of EAP principles for reducing energy use.

2.2.1 Logging Methods

Prior research on NVMemPersist have used either (1) UNDO (journal) logging [10] or (2) Write-ahead logging (WAL) [13]. Also, to exploit byte addressability, NVMemPersist use word or object based logging unlike disk-based systems that use page-based logging. Briefly, UNDO works by first copying the old data to a journal, committing the updates in-place in the original data location, and if transaction fails, reverts using journal data. In WAL, all data updates are appended to a log, and when the log buffer runs out, the log contents are checkpointed to the original persistent location. WAL maintains a log index to locate and fetch the newest copy of data, whereas in UNDO, reads are always serviced from original data location. WAL provides sequential writes avoiding random writes which provides significant benefits for disk-based systems but gains are limited for memory-based systems. As Narayanan and Hodson [7] point out, although WAL in NVMemPersist-based applications can significantly improve performance and concurrency for large multicore systems with write-intensive workload, for read intensive workloads, the improvement is negligible or no improvement at all. This is because, servicing read operation requires looking up WAL index to locate the data with multiple versions in log unlike UNDO. Also, when updates are large, the fixed size logs buffer have to be frequently checkpointed. Checkpoints require parsing the log records sequentially, and copying multiple versions of same data to original location. While WAL makes updates faster by only appending to log, eventually all the logs should be committed, and hence does not change the total CPU instructions or NVM access. Importantly, a key NVM-Heap specific issue is that, unlike UNDO, after an update of data, every load/store to same data must be redirected by S/W to a log instead of its original location. This not only requires significant application code changes, but also results in additional CPU instructions.

A.

Write pattern	WAL	UNDO
Sequential	7849	6199
Large - Seq	7886	7326
Large- Rand	2718	4728
Batched	44247	82918

In OPS/Sec

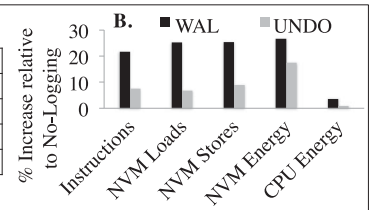

Fig. 4. SQLite bench—A. OPS/sec comparison, B. WAL, UNDO energy & NVM writes relative to baseline.

To verify with a NVMemPersist workload, we extend SQLite's in-memory database (with no logging) feature with Intel's memory persistence logging library [9]. We run the SQLite benchmark with UNDO, REDO logging discussed in [10], [13]. The table in Fig. 4A shows that, while WAL provides higher throughput (OPS/sec) for small sequential and random updates, the performance drops for large updates. Fig. 4B shows the relative increase in CPU instructions, NVM writes, and energy for WAL relative to UNDO logging for complete benchmark that includes all access patterns.

2.2.2 Metadata Durability Cost

The metadata durability cost can be significant mainly from persistent NVM allocation/de-allocations. Specifically, most modern allocators use complex data structures, and aggressive garbage collection to reduce fragmentation, which increases the metadata-related energy. While Kannan et al. [14] propose an NVM-write-aware allocator placing complex data structures into DRAM and reduce NVM access, the CPU energy overheads do not change, and with increasing $metadatasize/datasize$ ratio, the CPU instructions and NVM accesses also increase.

3 Energy Efficient Durability Principles

We next discuss energy-efficient durability principles as a first step towards reducing application energy usage by trading off performance and capacity.

Flexible support for energy efficient logging. As analyzed earlier, performance and energy-usage differs across NVMemPersist logging methods (WAL and UNDO). Therefore, we propose an application transparent mechanism that switches from performance mode (WAL) to energy-efficient mode (UNDO) when energy-jbudget is constrained. Applications start with the default performance (WAL) mode. After every epoch (fixed time interval), the logging library measures the available energy budget and usage. When energy use from durability is high, the energy-efficient mode (UNDO) is enabled for all subsequent transactions. A restriction enforced is that for transactions with a dirty (uncommitted) WAL log, UNDO logging cannot be used.

Trading capacity for energy. To reduce the metadata durability-related energy cost of persistent allocators, we propose (1) an adaptive mechanism that reduces frequent OS allocation requests by mapping/reserving larger memory regions when the energy budget is constrained. The size of mapped regions is limited by available capacity. While reserving larger NVM space can increase fragmentation, by trading off capacity, the work done (and energy consumption) can be reduced. (2) Next, the adaptive allocator also delays the frequency of garbage collection that perform coalesce/merge/free under energy constrained mode. Our analysis of SQLite benchmark shows around 12 percent, and 9.6 percent reduction in instructions and NVM writes with such optimization, and less than 280 MB additional capacity.

3.1 Relaxing Durability—ACI-RD

When energy-level is critical, efficient durability is not sufficient. In other words, always using a strong ACID approach in end-user devices can substantially drain battery power and prevent application running to completion. Hence, for energy critical conditions, we propose the ACI-RD (RD—relaxed durability), its key idea being to reduce the frequency/increase the interval between application data durability updates, without delaying the metadata updates critical for correctness. Specifically, in the event of failure, the up-to-date metadata can be used to UNDO all changes after the last consistent log update (previous epoch), and restore application to the last epoch state (an approach similar to checkpoint/recovery), but used only when energy budget is critical. To enable this, EAP maintains a consistent log with last consistent state of both metadata and data, and an UNDO log with just the consistent and durable metadata for recovering from failure. The UNDO log is required for garbage collection, and also avoids complete sequential scan of consistent log. Fig. 5 shows ACI-RD steps without the allocator log, (1) In the ACID epoch, when energy is not a constraint, data, metadata for A,B are updated to consistent log. Next, (2) when energy becomes a constraint, for updates to A,B with A1, B1, ACID-RD epoch is enabled, and only A1, B1 metadata is written to UNDO log. Step (3) shows original data state after (1), (2), and (4) shows failure. Finally, during recovery, (5) A1, B1 are reverted with the UNDO log, and A,B is recovered from consistent log. The result of this mechanism is a trade-off between durability (D in ACID) and energy, without compromising application correctness. We realize this approach with an epoch-based durability, where, given an energy budget (E), the application execution is divided into intervals of t_{msec}, called 'Epochs', with per-epoch energy budget E_{epoch}. At the end of an epoch, the increase in energy usage relative to its budget ΔE_{epoch} is estimated, in addition to the increase in the number of transactions performed by the application. E_{epoch_i} denotes the energy consumed at epoch i. Currently, we assume E_{budget} is known from application that only commits metadata for correctness (or) from earlier runs, but we plan to explore more online models. $\Delta E_{epoch} = (E_{epoch_i} - E_{budget})/E_{budget}$.

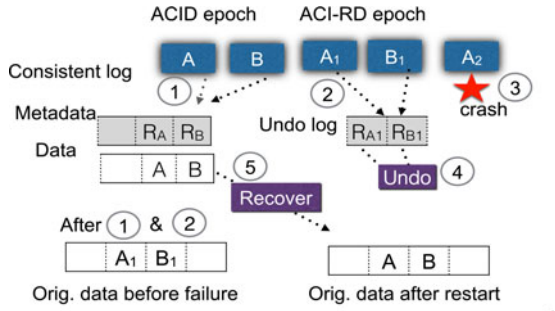

Fig. 5. ACID-RD steps.

3.2 Impact of EAP

To evaluate the overall impact of EAP, we first apply the energy-efficient optimizations (flexible logging, and trading capacity for energy), and combine them with relaxed durability (ACI-RD) based on the available energy budget. We use the same benchmarks and system setup, and CPU and NVM energy estimation methods discussed in Section 2. The epoch interval is set to 1 Sec

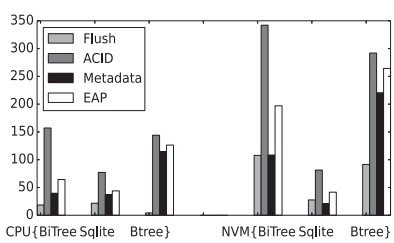

Fig. 6. Increase in CPU, NVM energy (percent) relative to No-ACID.

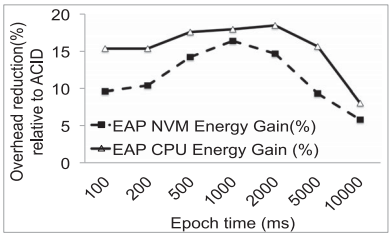

Fig. 7. Energy gains versus Epoch interval for B-tree.

and applications runtime vary from 40-60 Sec with energy budget of a no-ACID execution as baseline. Hence all optimizations are always applied. We compare the proposed EAP approach against (1) FLUSH (only CLFlush, fence and barriers), (2) traditional ACID, (3) Metadata, which logs only the metadata. Fig. 6 shows the implications of persistence support on increase in CPU, and NVM energy consumption relative to a No-ACID baseline (y-axis in percent). EAP provides $\sim 2\times$ reduction in CPU and NVM energy cost relative to the ACID approach for memory and cache inefficient Binary tree, 45 percent over transaction intensive SQLite, and 18 percent over cache and memory efficient B-tree—lower gains are mainly due to significant library metadata allocator cost for new node insert, and delete operations although the data log cost relatively is lesser than Binary tree. Finally, for memory- and persistent-intensive Hashtable (not shown in this graph), we observed up to $2.5\times$ energy-reduction.

Impact of Epoch interval. The epoch interval is used for the runtime energy profiling and initiating EAP's energy efficient durability and ACI-RD. Fig. 7 shows the impact of the epoch interval (x-axis in milliseconds) on reducing the energy overheads for memory- and cache-efficient B-tree compared against ACID. Maximum gains are achieved in the interval range of 700-1,000 ms range, whereas decreasing the interval to 100 ms increases profiling noise, and logging reconfiguration (ACID—ACI-RD) overhead. Increasing the interval increases ACID cost. When using ACI-RD with 700 ms epoch, 8-10 percent of data is not made durable by EAP (a trend similar to the epoch interval impact).

3.3 Conclusions and Future Work

This paper analyzes the energy overhead of persistence and identifies that durability costs are the most significant contributor to energy usage. Further, the paper proposes energy-aware persistence to reduce logging energy usage with efficient durability and a novel relaxed durability (ACI-RD). While the results with persistent benchmarks are promising, we plan to study more persistent applications, and the durability overhead of OS, and an extensive energy model.

References

[1] M. K. Qureshi, V. Srinivasan, and J. A. Rivers, "Scalable high performance main memory system using phase-change memory technology," in *Proc. 36th Annu. Int. Symp. Comput. Archit.*, 2009, pp. 24–33.
[2] B. C. Lee, E. Ipek, O. Mutlu, and D. Burger, "Architecting phase change memory as a scalable dram alternative," in *Proc. 36th Annu. Int. Symp. Comput. Archit.*, 2009, pp. 2–13.
[3] 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," in *Proc. 46th Annu. IEEE/ACM Int. Symp. Microarchit.*, 2013, pp. 421–432.
[4] J. Condit, E. B. Nightingale, C. Frost, et al., "Better I/O through byte-addressable, persistent memory," in *Proc. ACM SIGOPS 22nd Symp. Operating Syst. Principles*, 2009, pp. 133–146.
[5] S. Pelley, P. M. Chen, and T. F. Wenisch, "Memory persistency," in *Proc. 41st Annu. Int. Symp. Comput. Archit.*, 2014, pp. 265–276.
[6] Y. Lu, J. Shu, L. Sun, and O. Mutlu, "Loose-ordering consistency for persistent memory," in *Proc. IEEE 32nd Int. Conf. Comput. Des.*, 2014, pp. 216–223.
[7] D. Narayanan and O. Hodson, "Whole-system persistence," in *Proc. 17th Int. Conf. Archit. Support Program. Lang. Oper. Syst.*, 2012, pp. 401–410.
[8] S. R. Dulloor, S. Kumar, A. Keshavamurthy, et al., "System software for persistent memory," in *Proc. 9th Eur. Conf. Comput. Syst.* 2014, pp. 15:1–15:15.
[9] Intel. Logging library [Online]. Available: https://github.com/pmem/nvml
[10] J. Coburn, A. M. Caulfield, A. Akel, et al., "NV-Heaps: Making persistent objects fast and safe with next-generation, non-volatile memories," in *Proc. 16th Int. Conf. Archit. Support Program. Lang. Oper. Syst.*, 2011, pp. 105–118.
[11] SQLite [Online]. Available: http://www.sqlite.org
[12] Google. LevelDb [Online]. Available: http://leveldb.org/
[13] H. Volos, A. J. Tack, and M. M. Swift, "Mnemosyne: Lightweight persistent memory," in *Proc. 16th Int. Conf. Archit. Support Programm. Lang. Oper. Syst.*, 2011, pp. 91–104.
[14] S. Kannan, A. Gavrilovska, and K. Schwan, "Reducing the cost of persistence for nonvolatile heaps in end user devices," in *Proc. IEEE 20th Int. Symp. High-Perform. Comput. Archit.*, 2014, pp. 512–523.