Journal of
Low Power Electronics and Applications

Article

A Technique for Improving Lifetime of Non-Volatile Caches Using Write-Minimization

Sparsh Mittal [1,*] and Jeffrey Vetter [1,2]

Received: 9 November 2015; Accepted: 7 January 2016; Published: 18 January 2016
Academic Editor: Swaroop Ghosh

1 Oak Ridge National Laboratory, Oak Ridge, TN 37830, USA
2 Georgia Institute of Technology, Atlanta, GA 30332, USA
* Correspondence: mittals@ornl.gov; Tel.: +1 865-574-8531

Abstract: While non-volatile memories (NVMs) provide high-density and low-leakage, they also have low write-endurance. This, along with the write-variation introduced by the cache management policies, can lead to very small cache lifetime. In this paper, we propose ENLIVE, a technique for ENhancing the LIfetime of non-Volatile cachEs. Our technique uses a small SRAM (static random access memory) storage, called HotStore. ENLIVE detects frequently written blocks and transfers them to the HotStore so that they can be accessed with smaller latency and energy. This also reduces the number of writes to the NVM cache which improves its lifetime. We present microarchitectural schemes for managing the HotStore. Simulations have been performed using an x86-64 simulator and benchmarks from SPEC2006 suite. We observe that ENLIVE provides higher improvement in lifetime and better performance and energy efficiency than two state-of-the-art techniques for improving NVM cache lifetime. ENLIVE provides 8.47×, 14.67× and 15.79× improvement in lifetime or two, four and eight core systems, respectively. In addition, it works well for a range of system and algorithm parameters and incurs only small overhead.

Keywords: non-volatile memory (NVM); SRAM; cache; write-endurance; temporal locality; write minimization; device lifetime

1. Introduction

Recent trends of chip-miniaturization and CMOS (complementary metal-oxide semiconductor) scaling have led to a large increase in the number of cores on a chip [1]. To feed data to these cores and avoid off-chip accesses, the size of last level caches has significantly increased, for example, Intel's Xeon E7-8870 processor has 30 MB last level cache (LLC) [1]. Conventionally, caches have been designed using SRAM (static random access memory), since it provides high performance and write endurance. However, SRAM also has low density and high leakage energy which leads to increased energy consumption and temperature of the chip. With ongoing scaling of operating voltage, the critical charge required to flip the value stored in a memory cell has been decreasing [2], and this poses a serious concern for reliability of charge-based memories such as SRAM.

Non-volatile memories (NVMs) hold the promise of providing a low-leakage, high-density alternative to SRAM for designing on-chip caches [3–6]. NVMs such as spin transfer torque RAM (STT-RAM) and resistive RAM (ReRAM) have several attractive features, such as read latency comparable to that of SRAM, high-density and CMOS compatibility [7,8]. Further, NVMs are expected to scale to much smaller feature sizes than the charge-based memories since they rely on resistivity rather than charge as the information carrier.

A crucial limitation of NVMs, however, is that their write endurance is orders of magnitude lower than that of SRAM and DRAM (dynamic random access memory). For example, while the write-endurance of SRAM and DRAM are in the range of 10^{16}, this value for ReRAM is only 10^{11} [9]. For STT-RAM, although a write-endurance value of 10^{15} has been predicted [10], the best write-endurance test so far shows only 4×10^{12} [5,11]. Process variation may further reduce these values by an order of magnitude [12].

Further, existing cache management techniques have been designed for optimizing performance and they do not take the write-endurance of device into account. Hence, due to the temporal locality of program access, they may greatly increase the number of accesses to a few blocks of the cache. Due to failure of these blocks, the actual device lifetime may be much smaller than what is expected assuming uniform distribution of writes. Figure 1 shows an example of this where the number of writes to different cache blocks are plotted for SpPo (sphinx-povray) workload (more details of experimental setup are provided in Section 4). Clearly, the write-variation across different cache bocks is very large: the maximum write on a block (72349) is 1540 times the average write on any block (47).

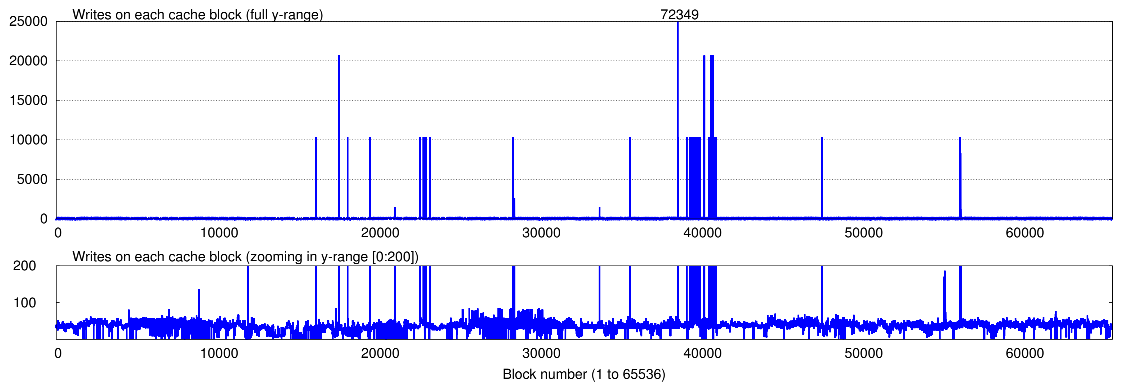

Figure 1. Number of writes on cache blocks for SpPo (sphinx-povray) workload. The top figure shows the full y-range (number of writes), while the bottom figure shows only the range [0:200]. Notice that the write-variation is very large, while the average write-count per block is only 47, the maximum write-count is 72,349.

Thus, limited endurance of NVMs, along with the large write-variation introduced by the existing cache management techniques, may cause the endurance limit to be reached in a very short duration of time. This leads to hard errors in the cache [2] and limits its lifetime significantly. This vulnerability can also be exploited by any attacker by running a code which writes repeatedly to a single block to make it reach its endurance limit (called repeated address attack) for causing device failure. Thus, effective architectural techniques are required for managing NVM caches for making them a universal memory solution.

In this paper, we present ENLIVE, a technique for ENhancing the LIfetime of non-Volatile cachEs. ENLIVE uses a small (e.g., 128 entry) SRAM storage, called HotStore. To reduce the number of writes to the cache, ENLIVE migrates the frequently used blocks into the HotStore, so that the future accesses can be served from the HotStore (Section 2). This improves the performance and energy efficiency and also reduces the number of writes to the NVM cache, which translates into enhanced cache lifetime. We also discuss the architectural mechanism to manage the HotStore and show that the storage overhead of ENLIVE is less than 2% of the L2 cache (Section 3), which is small. In this paper, we assume a ReRAM cache, and based on the explanation, ENLIVE can be easily applied to any NVM cache. In the remainder of this paper, for sake of convenience, we use the words ReRAM and NVM interchangeably.

Microarchitectural simulations have been performed using Sniper simulator and workloads from SPEC2006 suite and HPC field (Section 4). In addition, ENLIVE has been compared with two recently proposed techniques for improving lifetime of NVM caches, namely PoLF (probabilistic line-flush) [5] and LastingNVCache [13] (refer Section 5.1). The results have shown that, compared to other techniques, ENLIVE provides larger improvement in cache lifetime and performance, with a smaller energy loss (Section 5.2). For two, four and eight core systems, the average improvement in lifetime on using ENLIVE are 8.47×, 14.67× and 15.79×, respectively. By comparison, the average lifetime improvement with LastingNVCache (which, on average, performs better than PoLF) for 2, 4 and 8 core systems is 6.81×, 8.76× and 11.87×, respectively. Additional results show that ENLIVE works well for a wide-range of system and algorithm parameters (Section 5.3).

The rest of the paper is organized as follows. Section 2 discusses the methodology. Section 3 presents the salient features of ENLIVE, evaluates its implementation overhead and discusses the application of ENLIVE for mitigating security threats in NVM caches. Section 4 discusses the experimental platform, workloads and evaluation metrics. Section 5 presents the experimental results. Section 6 discusses related work on NVM cache and lifetime improvement techniques. Finally, Section 7 concludes this paper.

2. Methodology

Notations: In this paper, the LLC is assumed to be an L2 cache. Let S, W, D and G denote the number of L2 sets, associativity, block-size and tag-size, respectively. In this paper, we take $D = 512$ bits (64 bytes) and $G = 40$ bits. Let β denote the ratio of the number of entries in the HotStore and the number of L2 sets. Thus, the number of entries in HotStore is βS. The typical values of β are 1/32, 1/16, 1/8 *etc.*

2.1. Key Idea

It is well-known that due to temporal locality of cache access, a few blocks see much more number of cache writes than the remaining blocks. This leads to two issues in NVM cache management. First, NVM write latency and energy are much higher than read latency and energy (refer Table 1), respectively, and, thus, these writes degrade the performance and energy-efficiency of NVM cache. Second, and more importantly, frequent writes to few blocks lead to failure of these blocks much earlier than what is expected assuming uniformly distributed writes.

ENLIVE works on the key idea that the frequently written blocks can be stored in a small SRAM storage, called HotStore. Thus, future writes to those blocks can be served from the HotStore, instead of from the NVM cache. Due to the small size of SRAM storage, along with smaller access latency of SRAM itself, these blocks can be accessed with small latency, which improves the performance by making the common-case fast. In addition, since the write-endurance of SRAM is very high, the SRAM storage absorbs most of the writes, and, thus, the writes to NVM cache are significantly reduced. This leads to improvement in the lifetime of the NVM cache. The overall organization of cache hierarchy with addition of HotStore is illustrated in Figure 2.

The HotStore only stores the data of a block and does not require a separate tag-field, since the tag directory of L2 cache itself is used. The benefit of this is that on a cache access, the tag-matching is done only on the L2 tag directory, as in the normal cache. Thus, from the point of view of processor-core, all the tags and corresponding cache blocks are always stored in the L2 cache, although internally, they may be stored in either the main NVM storage or the HotStore. Hence, a fully-associative search in the HotStore is not required. This fact enables use of a relatively large number of entries in the HotStore. From any set, only a single block can be stored in the HotStore and hence, when required, this block can be directly accessed as shown in Figure 3. In addition, when a block, which is evicted, is in the HotStore, the newly arrived data-item is placed in the HotStore itself.

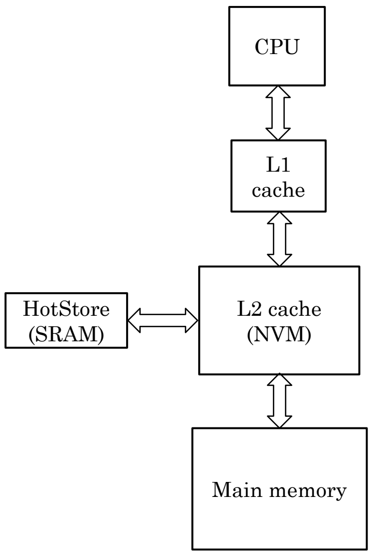

Figure 2. Illustration of the cache hierarchy with addition of HotStore.

2.2. An Example of How HotStore Works

Figure 3 shows an example of how HotStore works to illustrate the basic idea. Initially, the HotStore is empty and a set with set-index 38 has data in all of its ways. On a write-command to block "B", its write-counter exceeds the threshold (2), and, hence, it is inserted into HotStore. Next, "read C" command is issued, and, hence, the element "C" is read from L2 cache. Next, "write B" command is issued, and since B is stored in HotStore, it is written in HotStore and not in L2 cache. Next, "read B" command is issued and the data are supplied from HotStore and not from L2 cache. Section 2.3 explains the implementation and working of HotStore in full detail.

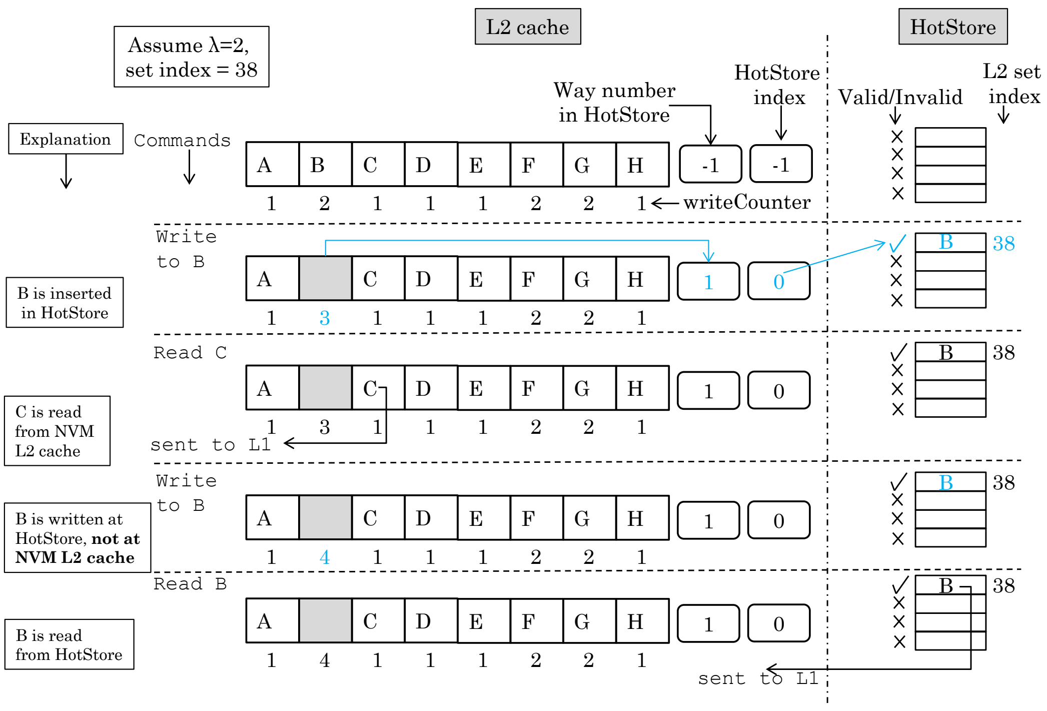

Figure 3. Example of how the ENLIVE technique works for an arbitrary set-index 38 (changed items are marked in color).

2.3. Implementation and Optimization

To ensure that the HotStore latency is small, its size needs to be small. Due to the limited and small size of the storage, effective cache management is required to ensure that only the most frequently written blocks reside in the HotStore and the cold blocks are soon evicted. Algorithm 1 shows the procedure for handling a cache write and managing the HotStore. If the block is already in the HotStore (Lines 1-3), a read or write is performed from the HotStore. Regardless of whether the block is in HotStore or L2 cache, on a write, the corresponding write-counter in L2 cache is incremented and on any cache read/write access, the information in LRU replacement policy of L2 cache is updated (not shown in Algorithm 1). The insertion into and eviction from HotStore are handled as follows.

Algorithm 1: Algorithm for handling a write to an arbitrary block "blockB" and managing the HotStore

```
1  if isInHotStore[blockB] == TRUE then
2      /* Block is already in HotStore so write there, no writes to NVM cache            */
3      writeToHotStore(blockB)
4  else if writeCounterBlockB $>$ $\lambda$ AND isInHotStoreFromThisSet[setOfBlockB] == FALSE then
5      /* This is a hot block and no block from this set is in HotStore.                 */
6      if curSizeOfHotStore $<$ maxSizeOfHotStore then
7          /* HotStore has free space, so this block can be inserted                     */
8          isInHotStore[blockB] $\leftarrow$ TRUE
9          isInHotStoreFromThisSet[setOfBlockB] $\leftarrow$ TRUE
10         insertInHotStore(blockB, setOfBlockB)
11     else
12         /* No free space, see if a block can be evicted                               */
13         replacementCandidate $\leftarrow$ get_ReplacementCandidateOfHotStore()
14         if writeCounterBlockB $>$ writeCounter(replacementCandidate) then
15             /* replacementCandidate can be evicted.  Copy it back to NVM cache        */
16             setOfReplCandidate $\leftarrow$ get_setnumber(replacementCandidate)
17             isInHotStoreFromThisSet[setOfReplCandidate] $\leftarrow$ FALSE
18             isInHotStore[replacementCandidate] $\leftarrow$ FALSE
19             copyFromHotStoreToNVMCache(replacementCandidate)
20             /* Now transfer blockB to HotStore                                        */
21             isInHotStore[blockB] $\leftarrow$ TRUE
22             isInHotStoreFromThisSet[setOfBlockB] $\leftarrow$ TRUE
23             insertInHotStore(blockB, setOfBlockB)
24         else
25             /* This block cannot be inserted to HotStore.  Write it to NVM cache      */
26             writeToNVMcache(blockB)
27         end
28     end
29 else if writeCounterBlockB $>$ $\lambda$ AND isInHotStoreFromThisSet[setOfBlockB] == TRUE then
30     /* A block from the same set is already in the HotStore, see which is hotter.     */
31     blockFromThisSetInHotStore $\leftarrow$ get_BlockFromThisSetInHotStore(setOfBlockB)
32     if writeCounterBlockB $>$ writeCounter(blockFromThisSetInHotStore) then
33         /* Evict blockFromThisSetInHotStore and copy it back to NVM cache             */
34         isInHotStore[blockFromThisSetInHotStore] $\leftarrow$ FALSE
35         copyFromHotStoreToNVMCache(blockFromThisSetInHotStore)
36         /* Now transfer blockB to HotStore                                            */
37         isInHotStore[blockB] $\leftarrow$ TRUE
38         insertInHotStore(blockB, setOfBlockB)
39     else
40         /* This block cannot be inserted to HotStore.  Just write to NVM cache        */
41         writeToNVMcache(blockB)
42     end
43 else
44     /* no need of inserting in HotStore, since this is not a hot block                */
45     writeToNVMcache(blockB)
46 end
```

2.3.1. Insertion into HotStore

We use a threshold λ. A cache block which sees more than λ writes is a candidate for promotion to HotStore (Lines 4–42), depending on other factors. A block with less than λ writes is considered a cold block and need not be inserted in the HotStore (Lines 43–45). Due to the temporal locality, only a few MRU blocks in a set are expected to see most of the access [14]. Hence, we allow only one block from a set to be stored in the HotStore at a time which also simplifies the management of HotStore. If free space exists in the HotStore, a hot block can be inserted (Lines 6–10), otherwise, a replacement candidate is searched (Line 13) as discussed below.

2.3.2. Eviction from HotStore

If the number of writes to the replacement candidate are less than the incoming block (Line 14), it is assumed that the incoming block is more write-intensive, and, hence, should get priority over the replacement candidate. In such a case, the replacement candidate is copied back to the NVM cache (Lines 16–19), and the new block is stored in the HotStore (Lines 21–23). A similar situation also appears if a block from the same set as the incoming set is already in the HotStore (Lines 29–42). Since only one block from a set can reside in HotStore at any time, the write-counter values of incoming and existing blocks are compared and the one with largest value is kept in the HotStore. If the incoming block is less write-intensive, it is not inserted into HotStore (Lines 24–26 and Lines 39–41).

2.3.3. Replacement Policy for HotStore

To select a suitable replacement candidate, HotStore also uses a replacement policy. An effective replacement policy will keep only the hottest blocks in the HotStore and thus minimize frequent evictions from HotStore. We have experimented with two replacement policies.

1. LNW (least number of writes), which selects the block with the least number of writes to it as the replacement candidate. It uses the same write-counters as the L2 cache. We assume four-cycle extra latency of this replacement policy, although it is noteworthy that the selection of a replacement candidate can be done off the critical path.
2. NRU (not recently used) [15], is an approximation of LRU (least recently used) replacement policy and is commonly used in commercial microprocessors [15]. NRU requires only one bit of storage for each HotStore entry, compared to the LRU which requires $\log(\beta S)$ bits for each entry.

HotStore is intended to keep the most heavily written blocks and hence, LNW aligns more closely with the purpose of the HotStore, although it also has higher implementation overhead. For this reason, we use LNW in our main results and show the results with NRU in the section on parameter sensitivity study (Section 5.3). Our results show that in general, LNW replacement policy provides higher improvement in lifetime compared to the NRU replacement policy.

3. Salient Features and Applications of ENLIVE

ENLIVE has several salient features. It does not increase the miss-rate, and, thus, unlike previous NVM cache lifetime improvement techniques (e.g., [4,5,16]) it does not cause extra writes to main memory. In addition, it does not require offline profiling or compiler analysis (unlike [17]) or use of large prediction tables (unlike [18]). In what follows, we first show that the implementation overhead of ENLIVE is small (Section 3.1). We then show that ENLIVE can be very effective in mitigating security threats to NVM caches (see Section 3.2), such as those posed by repeated address attack.

3.1. Overhead Assessment

ENLIVE uses the following extra storage.

(E_1) The HotStore has βS-entries, each of which uses D-bits for data storage, one-bit for valid/invalid information and $\log S$-bits for recording which set a particular entry belongs to (note that for all log values, the base is two).
(E_2) For each cache set, $\log W$ bit storage is required to record the way number of the block from the set which is in HotStore and $\log(\beta S)$-bit storage is required to record the index in the HotStore where this block is stored.
(E_3) We assume that for each block, eight bits are used for the write-counter. In a single "generation", a block is unlikely to get larger than 2^8 writes. If, for a workload, a block gets more than this number of writes, then we allow the counter to get saturated; thus, overflow does not happen.
(E_4) NRU requires (βS) bit additional storage.

Thus, the storage overhead (Θ) of HotStore as a percentage of L2 cache is as follows:

$$\Theta = \frac{E_1 + E_2 + E_3 + E_4}{StorageOfL2} \times 100 \quad (1)$$

$$= \frac{\beta S \cdot (D + 1 + \log S) + S \cdot (\log W + \log(\beta S)) + S \cdot W \cdot 8 + (\beta S)}{S \cdot W \cdot (D + G)} \times 100. \quad (2)$$

Assuming a 16-way, 4 MB L2 cache and $\beta = 1/16$, the relative overhead (Θ) of ENLIVE is nearly 1.96% of the L2 cache, which is very small. By using smaller value of β (e.g., 1/32), a designer can further reduce this overhead, although it also reduces the lifetime improvement obtained (refer Section 5.3.3).

3.2. Application of ENLIVE for Mitigating Security Threat

Due to their limited write-endurance, NVMs can be easily attacked and worn-out using repeated address attack (RAA). Using a simple RAA, a malicious attacker can write a cache block repeatedly which may lead to write endurance failure. Further, it is also conceivable that a greedy-user may run attack-like codes a few months before the warranty of his/her computing system expires, so as to get a new system before the warranty expiration period. An endurance-unaware cache can be an easy target of such attacks and hence, conventional cache management policies leave a serious security vulnerability for NVM caches with limited write endurance. To show the extent of vulnerability, we present the following example.

Assume that L1 is designed with SRAM and its associativity is W_{L1}. Assume an attack-like write-sequence which circularly writes to W_{L1}+1 data blocks which are mapped to the same L1 set. Since this set can only hold W_{L1} blocks, every L1 write will lead to writebacks in the same L2 cache set. Assume that due to this, after every 200 cycles, the same L2 block is written again. In this case, the time required to fail this block is:

$$(WriteEndurance \times CyclePerWrite)/(CyclesPerSecond) \quad (3)$$

For write endurance of 10^{11} and 2GHz frequency, we obtain the time to fail as: 10,000 s or 2.8 h only. Clearly, in absence of any policy for protection from attacks, the L2 cache can be made to fail in less than 3 h. In addition, due to process variation, the write-endurance of weakest block may be smaller than 10^{11}.

ENLIVE can be useful in mitigating such security threats. ENLIVE keeps the most heavily written blocks in an SRAM storage, which has very high write-endurance and thus, the writes targeted for an NVM block are channeled to the SRAM storage. Furthermore, since ENLIVE dynamically changes the block which is actually stored in HotStore, alternate or multiple write-attacks can also be tolerated since other blocks which see large number of writes will be stored in the HotStore.

Further, the threshold λ can be randomly varied within a predetermined range which makes it difficult for attackers to predict the new block-location which will be stored in the HotStore. For this, note that a change in value of λ does not affect program correctness but only changes the number of writes absorbed by HotStore. Thus, assume that λ can be varied between range [2,4]. Initially, its value is two and thus, a block receiving more than two writes is promoted to HotStore. Now, on changing λ to three, a block which sees more than three writes is promoted to HotStore. Thus, some of the blocks, which were previously promoted to HotStore, may not get promoted to HotStore with this change value of λ. However, the attacker is unaware of this fact since only cache controller is aware of λ value and can change it. Thus, even if the attacker knows that a HotStore is being used, he/she does not know which blocks are stored in HotStore and hence, he/she cannot design an attack which only targets the blocks not stored in HotStore. Finally, as we show in the results section, use of HotStore can extend the lifetime of cache by an order of magnitude and, in this amount of time, any abnormal attack can be easily identified.

Note that although intra-set wear-leveling techniques (e.g., [5,13,19–21]) can also be used for mitigating repeated address attack to NVM, ENLIVE offers a distinct advantage over them. An intra-set wear-leveling technique only performs uniform-distribution of writes to a set and does not reduce the total number of writes to a set. Thus, although it can prolong the time taken for the first block in a set to fail, it cannot prolong the time required for all the blocks to fail. Hence, an attacker can continue to target different blocks in a set and finally, all the blocks in a set will fail in the same time as in the conventional cache, even if an intra-set wear-leveling technique is used. By comparison, ENLIVE performs write-minimization since the SRAM HotStore absorbs a large fraction of the writes and thus, it can increase the time required in both, making a single block fail and making all the blocks fail. Clearly, ENLIVE is more effective in mitigating security threats to NVM caches.

4. Evaluation Methodology

4.1. Simulation Platform

We perform simulations using Sniper x86-64 multi-core simulator [22]. The processor frequency is 2 GHz. L1 I/D caches are four-way 32 KB caches with two-cycle latency and are private to each core. They are assumed to be designed using SRAM for performance reasons. L2 cache is shared among cores, and its parameters are obtained using DESTINY tool [23,24] and are shown in Table 1. Here, we have assumed 32 nm process, write energy-delay product (EDP) optimized cache design, 16-way associativity and sequential cache access. All caches use LRU, write-back, write-allocate policy and L2 cache is inclusive of L1 caches. The parameters for SRAM HotStore are also obtained using DESTINY and they are shown in Table 2. Note that the parameters for L2 and HotStore are obtained using "cache" and "RAM" as the design target in the DESTINY, respectively. Main memory latency is 220 cycles. Memory bandwidth for two, four and eight-core systems is 15, 25 and 35 GB/s, respectively.

Table 1. Parameters for resistive RAM (ReRAM) L2 cache.

L2 size	2 MB	4 MB	8 MB	16 MB	32 MB
Hit Latency (ns)	5.059	5.120	5.904	5.974	8.100
Miss Latency (ns)	1.732	1.653	1.680	1.738	2.205
Write Latency (ns)	22.105	22.175	22.665	22.530	22.141
Hit Energy (nJ)	0.542	0.537	0.602	0.662	0.709
Miss Energy (nJ)	0.232	0.187	0.188	0.190	0.199
Write Energy (nJ)	0.876	0.827	0.882	0.957	1.020
Leakage Power (W)	0.019	0.037	0.083	0.123	0.197

Table 2. Parameters for static RAM (SRAM) HotStore.

Number of HotStore Entries	128	256	512	1024	2048
Read Latency (ns)	0.043	0.066	0.115	0.154	0.184
Write Latency (ns)	0.035	0.057	0.106	0.136	0.150
Read Energy (nJ)	0.034	0.037	0.038	0.054	0.072
Write Energy (nJ)	0.033	0.036	0.037	0.052	0.070
Leakage Power (mW)	3.35	4.36	5.17	8.68	17.35

4.2. Workloads

We use all 29 benchmarks from SPEC (Standard Performance Evaluation Corporation) CPU2006 suite with *ref* inputs and six benchmarks from HPC field (shown in italics in Table 3). Using these, we randomly create 18, nine and five multiprogrammed workloads for dual, quad and eight-core systems, such that each benchmark is used exactly once (except for completing the left-over group). These workloads are shown in Table 3.

Table 3. Workloads used in the experiments.

Benchmarks and Their Acronyms
As(astar), Bw(bwaves), Bz(bzip2), Cd(cactusADM)
Ca(calculix), Dl(dealII), Ga(gamess), Gc(gcc)
Gm(gemsFDTD), Gk(gobmk), Gr(gromacs), H2(h264ref)
Hm(hmmer), Lb(lbm), Ls(leslie3d), Lq(libquantum),
Mc(mcf), Mi(milc), Nd(namd), Om(omnetpp)
Pe(perlbench), Po(povray), Sj(sjeng), So(soplex)
Sp(sphinx), To(tonto), Wr(wrf), Xa(xalancbmk)
Ze(zeusmp), *Co(CoMD), Lu(lulesh), Mk(mcck)*
Ne(nekbone), Am(amg2013), Xb(xsbench)
2-core workloads (Using acronyms shown above)
SjGa, HmWr, AsDl, GcBw, GmGr, SoXa, BzMc
OmLb, NdCd, CaTo, SpPo, LqMi, LsZe, GkH2
PePo, NeLu, MkXb, CoAm
4-core workloads (Using acronyms shown above)
AsGaXaLu, GcBzGrTo, CaWrMkMi, LqCoMcLs, BwSoSjH2
PeZeHmDl, GkPoGmNd, LbOmCdSp, AmXbNeBw
8-core workloads (Using acronyms shown above)
GmCdNeLuToZeGkCo, AsGcCaLqLsPeGkLb
AmGaBzWrCoSoZePo, OmXbXaGrMkMcSjHm
MiBwH2DlNdSpGaGc

4.3. Evaluation Metrics

Our baseline is an LRU-managed ReRAM L2 cache which does not use any technique for write-minimization. We model the energy of L2, main memory and HotStore. We ignore the energy overhead of counters, since it is several orders of magnitude smaller compared to the energy of (L2 + main memory + HotStore). The parameters for both L2 cache and HotStore are shown above and the dynamic energy and leakage power of main memory are taken as 0.18 W and 70 nJ/access, respectively [14]. We show the results on the following metrics:

1. Relative cache lifetime where the lifetime is defined as the inverse of maximum writes on any cache block [4,13]
2. Weighted speedup [14] (called relative performance)
3. Percentage energy loss
4. Absolute increase in MPKI (miss-per-kilo-instructions)

To provide additional insights for ENLIVE, we also show the results on percentage decrease in WPKI (writes per kilo instructions) to NVM cache and the number of writes that are served from the HotStore (termed as nWriteToHotStore). The higher these values, the higher is the efficacy of HotStore in avoiding the writes to NVM.

We fast-forward the benchmarks for 10 B instructions and simulate each workload until the slowest application executes nInst instructions, where nInst is 200 M for dual-core workloads and 150 M instructions for quad and eight-core workloads. This helps in keeping the simulation turnaround time manageable since we simulate a large number of workloads, system configurations (dual, quad and eight-core system), algorithm parameters and three techniques (*viz.* ENLIVE, PoLF and LastingNVCache) along with the baseline (refer Section 5.1 for a background on PoLF and LastingNVCache). The early-completing benchmarks are allowed to run but their IPC (instruction per cycle) is recorded only for the first nInst instructions, following well-established simulation methodology [14]. Remaining metrics are computed for the entire execution, since they are system-wide metrics (while IPC is a per-core metric).

Note that since the last completing benchmark may take significantly longer time than the first one, the total number of instructions executed by the workloads is much more than nInst times the number of cores. Relative lifetime and weighted speedup values are averaged using geometric mean while the remaining metrics are averaged using arithmetic mean, since their values can be zero or negative. We have also computed fair speedup [14] and observed that their values are nearly same as weighted speedup and thus, ENLIVE does not cause unfairness. These results are omitted for brevity.

5. Results and Analysis

This section presents results on experimental evaluation of ENLIVE. We have compared ENLIVE with two recently proposed techniques for improving NVM cache lifetime named PoLF (Probabilistic line flush) [5] and LastingNVCache [13]. We first present a background on these techniques.

5.1. Comparison with Other Techniques

In PoLF, after a fixed number of write hits (called flush threshold or FT) in the entire cache, a write-operation is skipped; instead, the data item is directly written-back to memory and the cache-block is invalidated, without updating the LRU-age information. In LastingNVCache, such a flushing operation is performed after a fixed number of write-hits to that particular block itself. While PoLF selects hot-data in probabilistic manner, LastingNVCache does not work by probabilistic manner, rather, it actually records the writes to each block. For both of them, lifetime improvement is achieved by the fact that based on LRU replacement policy, the hot data from flushed block will be loaded in another cold block leading to intra-set wear-leveling. The latency values of incrementing the write-counter and comparison with the threshold are taken as one and two cycles, respectively.

Choice of flush threshold: Both PoLF and LastingNVCache work by data-invalidation, while ENLIVE works by data-movement between cache and HotStore. Thus, unlike the former two techniques, ENLIVE does not increase the accesses to main memory or cause large energy losses. Since large energy losses degrade system performance and reliability, and also achieving fair and meaningful comparison, we choose the flush threshold values for PoLF and LastingNVCache in the following manner. We test with even values of FT and find one FT each for dual, quad and eight-core systems separately which achieves highest possible lifetime improvement, while keeping the average energy loss across workloads less than 10%. Note that for every watt of power dissipated in computing system, an additional 0.5 to 1 watt of power is consumed by the cooling system also and very aggressive threshold values may significantly increase the writes to main memory which itself may be designed using NVM, thus creating endurance and contention issue in main memory; for these reasons, larger energy loss may be unacceptable. The exact values of FT which are obtained are shown in Table 4. As we show in the results section, with smaller energy loss (less than 3% on average), ENLIVE provides larger lifetime improvement than both PoLF and LastingNVCache.

Table 4. Flush threshold (FT) values for probabilistic line flush (PoLF) and Lasting Non-volatile(NV)Cache.

	Dual-Core	Quad-Core	Eight-Core
PoLF	20	20	26
LastingNVCache	14	14	16

5.2. Results with Default Parameters

Figures 4–6 show the results, which are obtained using the following parameters, LNW replacement policy for HotStore, $\lambda = 2$, $\beta = 1/16$, 16-way associativity for L2, 4 MB L2 for two core system, 8 MB L2 for four core system and 16 MB L2 for eight core-system, respectively. This range of LLC sizes and number of cores to LLC size ratio are common in commercial processors [25,26]. Per-workload figures for increase in MPKI and nWriteToHotStore are omitted for brevity; their average values are discussed below. We now analyze the results.

5.2.1. Results on Lifetime Improvement

For two, four and eight core systems, ENLIVE improves the cache lifetime by 8.47×, 14.67× and 15.79×, respectively. For several workloads, ENLIVE improves lifetime by more than 30×, e.g., SpPo, LqCoMcLs, OmXbXaGrMkMcSjHm, *etc.* The improvement in lifetime achieved with a workload also depends on the write-variation present in the original workload. If the write-variation is high, most writes happen to only a few blocks and when those blocks are stored in the HotStore, the number of writes to the NVM cache are significantly reduced. Conversely, for workloads with low write-variation, large reduction in writes to cache cannot be achieved by merely storing few blocks in HotStore since its size is fixed and small, and, hence, the improvement obtained in cache lifetime is small.

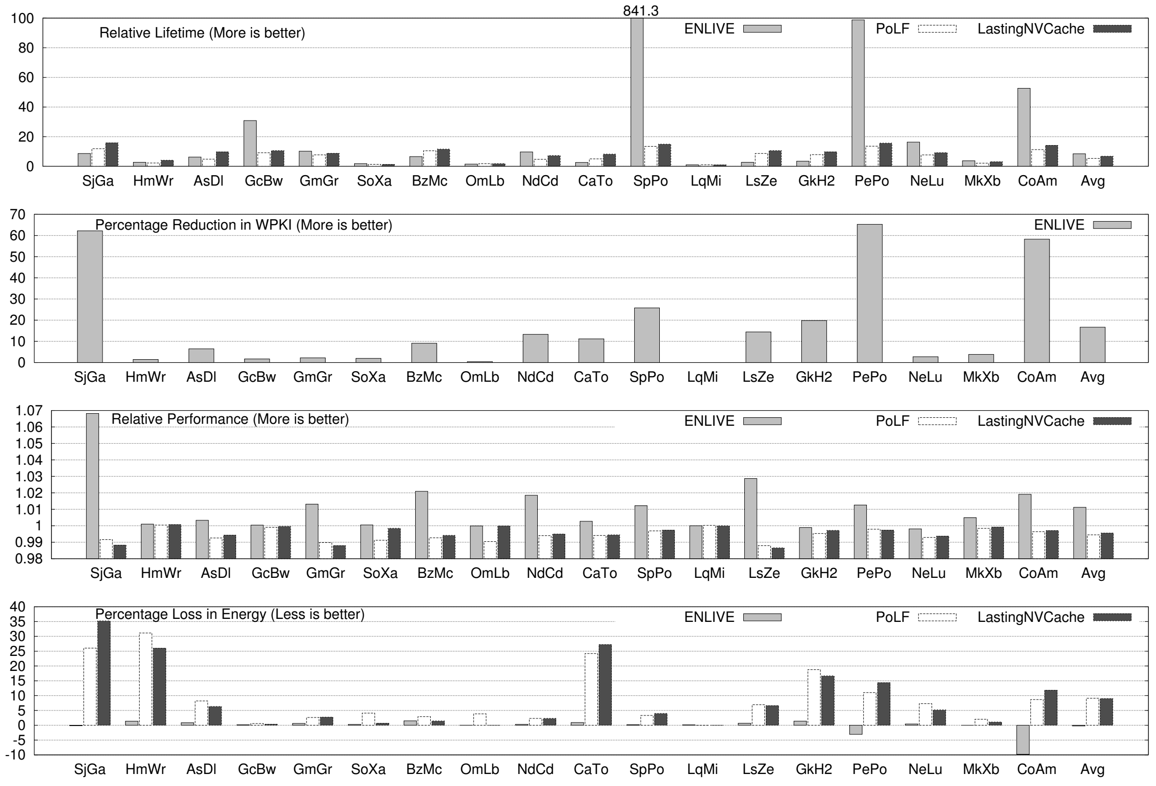

Figure 4. ENLIVE Results for two-core System.

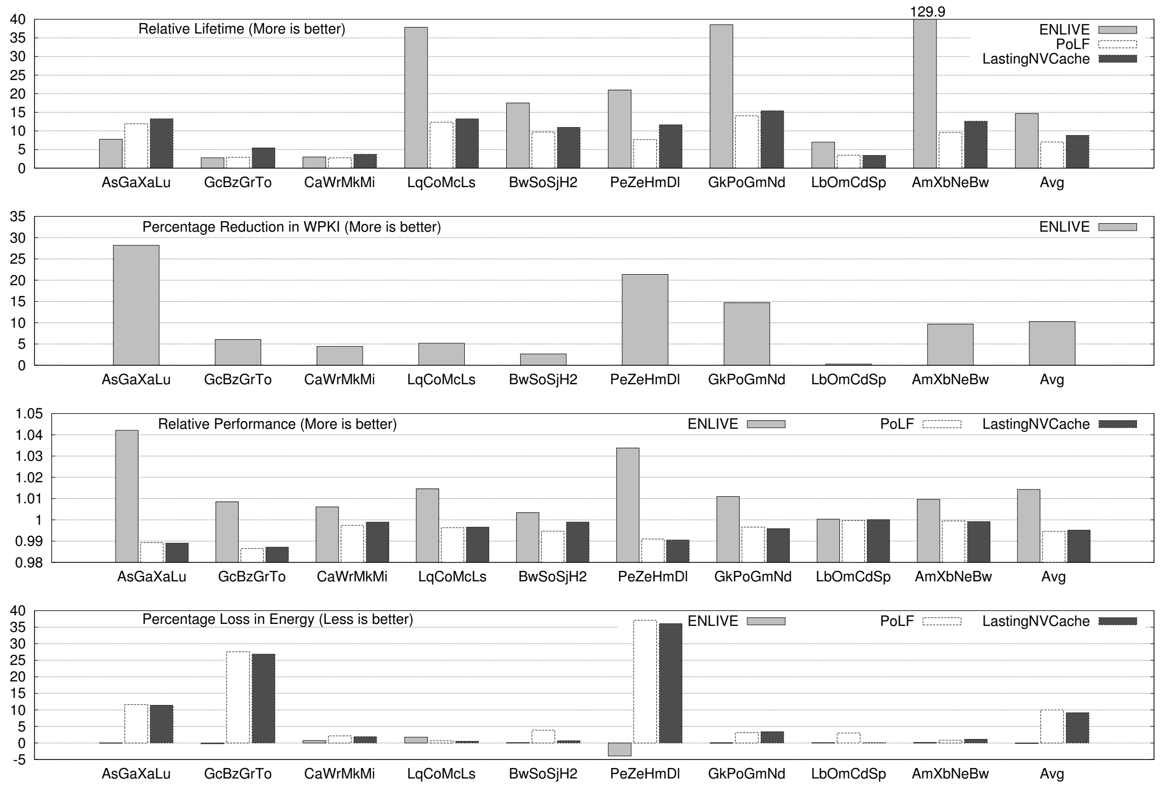

Figure 5. ENLIVE Results for four-core System.

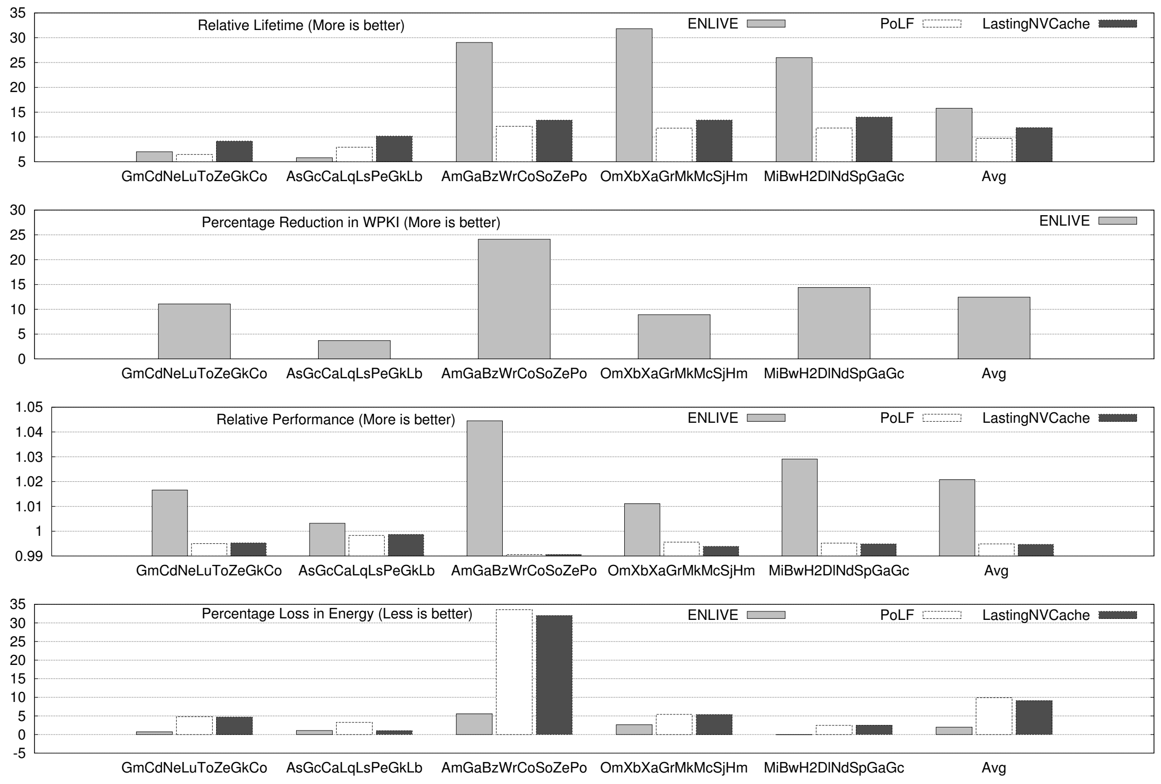

Figure 6. ENLIVE Results for eight-core System.

The lifetime improvement for two, four and eight-core system with PoLF is 5.35×, 7.02× and 9.72×, respectively and for LastingNVCache, these values are 6.81×, 8.76× and 11.87×, respectively.

Clearly, LastingNVCache provides higher lifetime improvement than PoLF, although it is still less than that provided by ENLIVE. This can be easily understood from the fact that ENLIVE actually reduces the writes to NVM, while other techniques only uniformly distribute the writes to different cache blocks.

5.2.2. Results on MPKI, Performance and Energy

ENLIVE uses in-cache data-movement, and, hence, it does not increase MPKI. The average values of increase in MPKI on using PoLF and LastingNVCache are shown in Table 5. PoLF and LastingNVCache cause a small performance loss, while ENLIVE provides a small improvement in performance. For 2 and 4-core system, ENLIVE provides a small saving in energy and for 8-core system, it incurs 2% energy loss. By comparison PoLF and LastingNVCache cause large energy loss (recall from Section 5.1 that an energy loss bound of 10% was used for them).

Table 5. Results on miss per kilo instruction (MPKI) increase with PoLF and LastingNVCache.

	Dual-Core	**Quad-Core**	**Eight-Core**
PoLF	0.26	0.19	0.20
LastingNVCache	0.16	0.13	0.17

Notice that our energy model includes the energy consumption of main memory, and, thus, while trying to improve cache lifetime, ENLIVE does not increase the energy consumption of main memory. Since most accesses are supplied from the HotStore, which is smaller and faster than the NVM L2 cache, ENLIVE slightly improves performance. However, HotStore is designed using SRAM which has higher leakage energy, and migrations to and from HotStore also consume energy, the energy advantage of HotStore is slightly nullified. Overall, it can be concluded that unlike other techniques, ENLIVE does not harm performance or energy efficiency.

5.2.3. Results on WPKI and nWriteToHotStore

PoLF and LastingNVCache only perform wear-leveling and hence, they do not reduce the WPKI. For ENLIVE, average values of percentage reduction in WPKI and nWriteToHotStore are shown in Table 6. Note that the HotStore stores only βS blocks, while the L2 cache stores WS, thus HotStore stores only $(\beta/W) \times 100$ percentage of L2 blocks. For $\beta = 1/16$ and $W = 16$, this equals 0.39% of L2 blocks. Thus, with less than 0.4% of extra data storage, ENLIVE can reduce the WPKI of cache by more than 10%.

Table 6. Results on reduction in write per kilo instruction (WPKI) and nWriteToHotStore with ENLIVE.

	Dual-Core	**Quad-Core**	**Eight-Core**
Percentage reduction in WPKI	16.7%	10.3%	12.4%
nWriteToHotStore	318 K	594 K	1785 K

Figure 7 shows the WPKI value (*i.e.*, write intensity) for baseline execution. From these values, the correlation between lifetime improvement achieved for an application and its write intensity can be seen. For example, HmWr and CaTo workloads have low WPKI (1.74 and 1.95, respectively) and hence, the lifetime improvement achieved for them with ENLIVE is also small, e.g., 2.69× and 2.60×, respectively. On the other hand, the WPKI of LqCoMcLs and BwSoSjH2 is 17.72 and 11.49, and due to this large write-intensity, the lifetime improvement achieved for them with ENLIVE is also large, e.g., 37.85× and 17.50×, respectively.

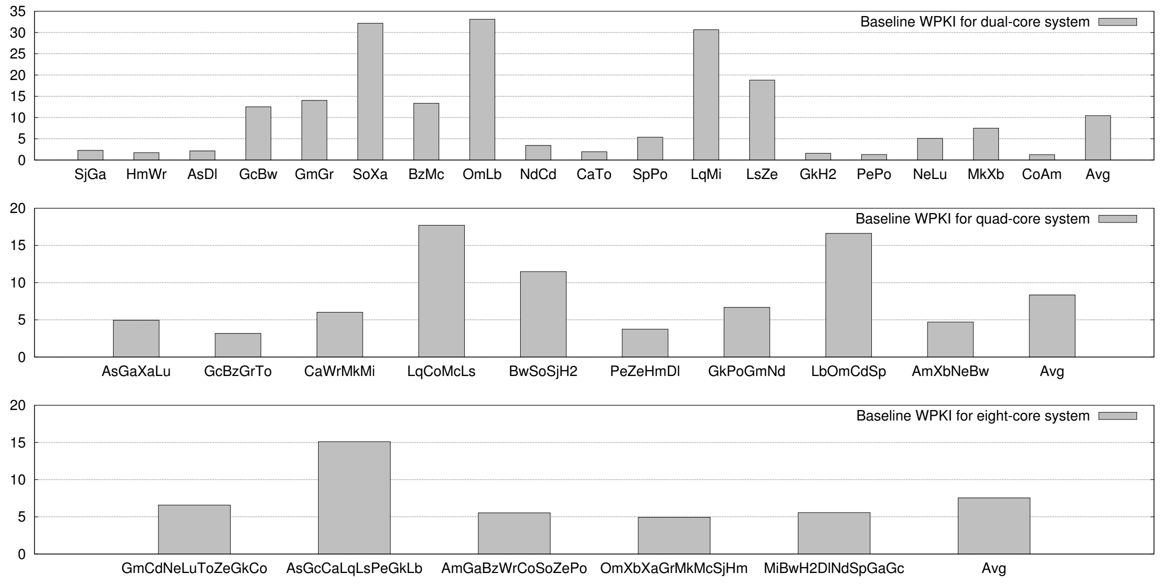

Figure 7. WPKI (*i.e.*, write-intensity) value for baseline execution.

5.2.4. Comments on Pros and Cons of Each Technique

PoLF flushes a block in probabilistic manner and hence, it may not always select a hot block. For this reason, PoLF provides smaller lifetime improvement and also incurs higher performance and energy loss than other two techniques. In addition, for a workload with high write-intensity but low write-variation, it may lead to unnecessary flushing of data without achieving corresponding improvement in lifetime. By comparison, both LastingNVCache and ENLIVE use counters for measuring write-intensity and thus can detect the hot-block more accurately.

Both PoLF and LastingNVCache use data-invalidation and one of their common limitations is that in an effort to reduce write-variation in cache, they may increase the writes to main memory. Since the main memory itself may be designed using NVM, these techniques may cause endurance issues in the main memory. By contrast, ENLIVE uses in-cache data-movement, and, hence, it does not degrade performance or energy efficiency or cause endurance or contention issue in the main memory.

The advantage of PoLF is that it only uses a single global counter for recording the number of writes, and hence, it incurs the smallest implementation overhead. Both LastingNVCache and ENLIVE use per-block counters. Further, the advantage of PoLF and LastingNVCache over ENLIVE is that they do not use any extra storage and thus, do not incur the area overhead of SRAM HotStore. However, as shown above, the overhead of ENLIVE is still very small and it performs better than other techniques on all metrics and hence, its small overhead is easily justified.

5.3. Parameter Sensitivity Results

We now focus exclusively on ENLIVE and study its sensitivity for different parameters. Each time, only one parameter is changed from the default parameters and the results are summarized in Table 7.

5.3.1. HotStore Replacement Policy

Compared to the LNW replacement policy, the NRU replacement policy gives smaller improvement in lifetime, which is expected since NRU accounts for only the recency of access and not the the magnitude of writes.

Table 7. Parameter Sensitivity Results. Default parameters are shown in Section 5.2.

	Relative Lifetime	% WPKI Decrease	Relative Performance	% Energy Loss	nWriteToHotStore
		Dual-core System			
Default	8.47	16.7	1.01	–0.24	318 K
NRU	8.25	16.4	1.01	–0.24	314 K
$\lambda = 3$	8.44	16.6	1.01	–0.24	317 K
$\lambda = 4$	8.54	16.6	1.01	–0.24	317 K
$\beta = 1/32$	6.88	13.3	1.01	–0.25	235 K
$\beta = 1/8$	9.06	19.9	1.02	–0.15	424 K
8-way	11.43	20.0	1.02	–0.03	420 K
32-way	6.11	13.1	1.01	–0.47	232 K
2 MB	5.45	13.0	1.01	–0.42	239 K
8 MB	15.19	21.7	1.02	0.00	398 K
		Quad-core System			
Default	14.67	10.3	1.01	–0.15	594 K
NRU	14.21	10.1	1.01	–0.16	580 K
$\lambda = 3$	14.37	10.2	1.01	–0.15	593 K
$\lambda = 4$	14.32	10.2	1.01	–0.15	591 K
$\beta = 1/32$	11.55	8.0	1.01	–0.13	480 K
$\beta = 1/8$	14.65	12.0	1.02	–0.01	683 K
8-way	15.06	11.8	1.02	0.04	685 K
32-way	7.80	8.1	1.01	–0.21	477 K
4 MB	8.20	7.5	1.01	0.55	507 K
16 MB	19.55	12.0	1.02	–0.23	668 K
		Eight-core System			
Default	15.79	12.4	1.02	1.99	1785 K
NRU	15.78	12.2	1.02	1.98	1751 K
$\lambda = 3$	16.67	12.4	1.02	1.99	1784 K
$\lambda = 4$	16.69	12.4	1.02	1.99	1783 K
$\beta = 1/32$	14.87	9.7	1.02	1.46	1410 K
$\beta = 1/8$	17.38	14.5	1.02	2.25	2086 K
8-way	25.79	14.4	1.02	1.95	2093 K
32-way	13.68	9.8	1.02	1.37	1403 K
8 MB	12.51	8.9	1.02	0.69	1506 K
32 MB	21.86	15.6	1.03	1.89	2036 K

5.3.2. Threshold (λ)

For small value of λ, ENLIVE aggressively promotes the blocks to HotStore, which works to reduce the number of writes to NVM. However, it also has the disadvantage of creating contention for HotStore leading to frequent eviction of blocks from HotStore which are written-back to L2 cache. Thus, the actual improvement in lifetime obtained depends on the trade-off between these two factors. Due to the mutual effect of these two parameters, the lifetime improvement does not change monotonically with λ. Still, a value of λ = 2, 3 or 4 is found to be reasonable.

5.3.3. Number of HotStore Entries (Corresponding to β)

On increasing β to 1/8, the number of entries in HotStore are increased and, with this, the improvement in lifetime also increases, since a higher number of hot blocks can be accommodated in the HotStore. The exact amount of improvement depends on the write-variation present in the workloads. Larger sized HotStore also dissipates higher leakage energy, which reflects in increased energy loss.

5.3.4. L2 Cache Associativity

For a fixed cache size and fixed β value, on reducing the cache associativity by half, the number of L2 sets and entries in HotStore are both increased by a factor of two. With a 16-way cache, only one out of 16 ways can be inserted in the HotStore, while with eight-way cache, one out of eight ways can be inserted, and, hence, the contention for HotStore is reduced. This increases the efficacy of HotStore in capturing hot blocks. This reflects in increased improvement in lifetime. Opposite is seen with 32-way cache, since now only one out of 32 blocks can be inserted in the HotStore at any time, which reduces the improvement in lifetime.

5.3.5. L2 Cache Size

With increasing cache size, cache hit-rate increases since applications have fixed working set size. This also increases the write-variation since only few blocks see repeated access. Hence, with increasing cache size, the effectiveness of ENLIVE in capturing hot blocks also increases, which reflects in higher improvement in cache lifetime.

The results shown in this section confirm that ENLIVE works well for a wide range of parameters. In addition, ENLIVE provides tunable knobs for trading-off acceptable implementation overhead and desired improvement in lifetime.

6. Related Work

In this section, we discuss relevant research work.

6.1. SRAM Limitations and Emerging Technologies

As the number of cores on a chip increases and key applications become even more data intensive [27], the requirement of cache and memory capacity is also on rise. In addition, since the power budget is limited, power consumption is now becoming the primary constraint in designing computer systems. Due to its low density and high leakage power, SRAM caches consume a significant fraction of chip area and power budget [28]. These elevated levels of power consumption may increase the chip-design complexity and exhaust the maximum capability of conventional cooling technologies. Although it is possible to use architectural mechanisms to reduce power consumption of SRAM caches [14,28], the savings provided by these techniques is not sufficient to meet the power budget targets of future computing systems.

To mitigate these challenges, researchers have explored several alternative low-leakage technologies, such as eDRAM [10,29], die-stacked DRAM [30] and NVM [5,21], *etc.* While eDRAM has high write-endurance, its main limitation is the requirement of refresh operations to maintain data integrity. Further, its retention period is in the range of tens of microseconds [29] and hence, a large fraction of energy spent in eDRAM caches is in the form of refresh energy. Similarly, die-stacked DRAM caches also require refresh operations and may create thermal and reliability issues due to higher operation temperatures in die-stacked chips. As for NVMs, although their write latency and energy are higher than that of SRAM, it has been shown that their near-zero leakage and high capacity can generally compensate for the overhead of writes [29]. Thus, addressing the limited write-endurance of NVMs remains a crucial research challenge for enabling their wide-spread adoption. For this reason, we have proposed a technique for improving lifetime of NVMs.

6.2. Techniques for Improving Lifetime of NVM Caches

Researchers have proposed several lifetime enhancement techniques for NVMs, which can be classified as write-minimization or wear-leveling. The write-minimization techniques, aim to reduce the number of writes to NVMs [31,32], while the wear-leveling techniques aim to uniformly distribute the writes to different blocks in the NVM cache [4,5,13,16,19,20]. Based on their granularity, wear-leveling techniques can be further classified as inter-set level [4,5,16] and intra-set level [5,13,19,20]. ENLIVE reduces the writes to hot blocks and thus, in addition to write-minimization, it also implicitly performs wear-leveling.

Some techniques perform a read-before-write operation to identify the changed bits or use additional flags to record the changed data words [33]. Using this, the redundant writes can be avoided, since only those bits or words whose values have changed can be written to the NVM cache. However, read-before-write schemes are effective mainly for PCM, since its write latency/energy are significantly higher (e.g., six to 10 times) than that its read latency/energy. By comparison, for ReRAM and STT-RAM, the write latency/energy are only two to four times that of read latency/energy and hence, the performance overhead of an extra read operation before the write operation becomes high. For this reason, read-before-write schemes are less suitable for caches, which are expected to be designed using STT-RAM and ReRAM, and not PCM.

Similarly, some techniques use extra flag bits to denote certain data-patterns (e.g., all-zero) and when the incoming data has such patterns, the write can be avoided by setting the flag to ON and later constructing the data using the flag value [34]. ENLIVE performs write-minimization at cache-access level and can be synergistically integrated with the above mentioned bit-level write-minimization techniques to further reduce the writes to the NVM cache.

6.3. Comparison with NVM-Only Cache Designs

Several techniques have been proposed for improving lifetime of NVM only caches, as shown above. However, since the write endurance of NVMs is orders of magnitude smaller than that of SRAMs, in case of high write traffic, NVM-only caches may show small lifetime since even after wear-leveling and write minimization, the worst-writes on a block may exceed the NVM write endurance. Further, as we have shown in Section 3.2, an NVM-only cache is much more vulnerable to a write-attack than an NVM cache with an SRAM HotStore (as used by ENLIVE technique).

In Section 5.1, we have qualitatively and quantitatively compared ENLIVE with two recently proposed techniques for NVM caches. In what follows, we qualitatively compare ENLIVE with two other techniques which use additional SRAM storage.

Sun *et al.* [31] use write-buffers for hiding long latencies of STT-RAM banks. A write-buffer temporarily buffers incoming writes to L2 cache. It is well-known that due to filtering by first-level cache, the locality of cache access is significantly reduced at the last level cache [14] and hence, the write-buffer cannot fully capture the hot blocks and may be polluted by blocks which are written only once. Thus, its effectiveness in reducing the number of writes to NVM cache is limited. By comparison, in ENLIVE technique, only the hot blocks are moved to the HotStore and they are later served from there and thus, it reduces the number of writes to the NVM cache. In addition, ENLIVE aims at improving the lifetime of NVM caches and not on the performance, which is the focus of the work by [31].

Ahn *et al.* [32] observe that, in many applications, computed values are varied within small ranges and hence, the upper bits of data are not changed as frequently as the lower bits. Based on this, they propose use of a small lower-bit cache (LBC) designed with SRAM which is placed between the L1 cache and the STT-RAM L2 cache. LBC aims to coalesce write-requests and also hides frequent value changes from the L2 cache. LBC stores the lower half of every word in cache blocks written by the L1 data cache. On a hit in LBC, the data to be provided for the L1 cache are a combination of upper bits from the L2 cache and lower bits from the LBC, since the lower bits in L2 may not be up-to-date. However, in cases when upper bits change frequently, this technique will incur large overhead. In addition, for the reasons mentioned above, an LBC may not capture the hot blocks.

6.4. Comparison with SRAM-NVM Hybrid Cache Designs

Some researchers propose way-based SRAM-NVM hybrid caches [31,35,36], where a few ways are designed using SRAM and the remaining ways are designed using NVM. The management policies for these caches aim to keep the write-intensive blocks in SRAM to improve the lifetime of the cache by reducing the number of writes to NVM. The limitation of these hybrid-cache designs, however, is that designing a few way using SRAM while others using NVM in the same cache may increase the design complexity and cost. By comparison, in ENLIVE, the SRAM HotStore can be designed separately from the cache.

In addition, in a hybrid cache with (say) one SRAM way, the number of SRAM sets become equal to the number of LLC sets. This may lead to wastage of SRAM space, since not all the LLC sets are expected to store hot-data. In contrast, the number of HotStore entries is much smaller than the number of LLC sets (e.g., 1/16 times the number of LLC sets). ENLIVE allows adapting the size of HotStore based on the write-variation present in the target workloads or the desired improvement in lifetime. The limitation of ENLIVE, however, is that the data in SRAM HotStore is also stored in the NVM LLC cache; by comparison, hybrid caches in general do not incur such redundancy. However, note that due to small size of HotStore, the redundancy is also small.

6.5. Techniques for Improving Lifetime of NVM Main Memory

Some researchers propose hybrid DRAM-PCM main memory design which aim to leverage the high performance and write-endurance of DRAM along with high density and low leakage power of PCM [37]. However, several key differences in operational mechanisms between caches and main memory make the solutions proposed for main memory inadequate for caches. In addition to inter-set write-variation, caches also show intra-set write-variation, which presents both opportunities and challenges. In addition, the techniques proposed for main memory typically have high latency overhead, which is acceptable at main memory level but is unacceptable at the level of on-chip caches.

7. Conclusions

Under ongoing core-scaling, NVMs offers a practical means of scaling cache capacity in an energy-efficient manner. NVMs are potential candidates for replacing SRAM for the design of last level caches due to their high density and low leakage power consumption. However, their limited write endurance and high write latency and energy present a crucial bottleneck in their widespread use. In this paper, we presented ENLIVE, a technique for improving the lifetime of NVM caches by minimizing the number of writes. Microarchitectural simulations show that ENLIVE outperforms two recently proposed techniques and works well for different system configurations.

Our future work will focus on synergistically integrating ENLIVE with write-reduction mechanisms (such as cache compression and bit-level write reduction) and wear-leveling mechanisms to improve the cache lifetime even further. To aggressively improve application performance with NVM caches, we also plan to study use of prefetching [38] in NVM caches while ensuring that extra writes do not degrade NVM lifetime. In recent years, the size of GPU caches has been increasing. Since power consumption is becoming first order design constraint in GPUs as well, our future efforts will focus on studying use of NVMs for designing GPU caches. Furthermore, recent research has highlighted sources of soft-errors in NVMs [2] and a part of our future work will focus on mitigating these errors in NVM caches.

Acknowledgments: Support for this work was provided by the U.S. Department of Energy, Office of Science, Advanced Scientific Computing Research.

Author Contributions: Both authors discussed the idea. Sparsh Mittal performed experiments and Jeffrey Vetter supervised the project. Both authors discussed the results. Sparsh Mittal developed the manuscript and both authors approve it.

Conflicts of Interest: The authors declare no conflict of interest.

References

1. Intel. Available online: http://ark.intel.com/products/53580/ (accessed on 13 January 2016).

2. Mittal, S.; Vetter, J. A Survey of Techniques for Modeling and Improving Reliability of Computing Systems. *IEEE Trans. Parallel Distrib. Syst. (TPDS)* **2015**, *PP*, doi:10.1109/TPDS.2015.2426179.
3. Motaman, S.; Iyengar, A.; Ghosh, S. Synergistic circuit and system design for energy-efficient and robust domain wall caches. In Proceedings of the International Symposium on Low Power Electronics and Design, La Jolla, CA, USA, 11–13 August 2014; pp. 195–200.
4. Chen, Y.; Wong, W.F.; Li, H.; Koh, C.K.; Zhang, Y.; Wen, W. On-chip caches built on multilevel spin-transfer torque RAM cells and its optimizations. *J. Emerg. Technol. Comput. Syst.* **2013**, *9*, 16:1–16:22.
5. Wang, J.; Dong, X.; Xie, Y.; Jouppi, N.P. Endurance-aware cache line management for non-volatile caches. *ACM Trans. Archit. Code Optim. (TACO)* **2014**, *11*, doi:10.1145/2579671.
6. Ghosh, S. Design methodologies for high density domain wall memory. In Proceedings of the International Symposium on Nanoscale Architectures (NANOARCH), Brooklyn, NY, USA, 15–17 July 2013; pp. 30–31.
7. Mittal, S.; Vetter, J.S. A Survey of Software Techniques for Using Non-Volatile Memories for Storage and Main Memory Systems. *IEEE Trans. Parallel Distrib. Syst.* **2015**, *PP*, doi:10.1109/TPDS.2015.2442980.
8. Vetter, J.; Mittal, S. Opportunities for Nonvolatile Memory Systems in Extreme-Scale High Performance Computing. *Comput. Sci. Eng. (CiSE)* **2015**, *17*, 73–82.
9. Kim, Y.B.; Lee, S.R.; Lee, D.; Lee, C.B.; Chang, M.; Hur, J.H.; Lee, M.J.; Park, G.S.; Kim, C.J.; Chung, U.I.; *et al.* Bi-layered RRAM with unlimited endurance and extremely uniform switching. In Proceedings of the IEEE Symposium on VLSI Technology (VLSIT), Honolulu, HI, USA, 14–16 June 2011; pp. 52–53.
10. Mittal, S.; Vetter, J.S.; Li, D. A Survey Of Architectural Approaches for Managing Embedded DRAM and Non-volatile On-chip Caches. *IEEE Trans. Parallel Distrib. Syst. (TPDS)* **2015**, *26*, 1524–1537.
11. Huai, Y. Spin-transfer torque MRAM (STT-MRAM): Challenges and prospects. *AAPPS Bull.* **2008**, *18*, 33–40.
12. Mittal, S. A Survey Of Architectural Techniques for Managing Process Variation. *ACM Comput. Surv.* **2016**, in press.
13. Mittal, S.; Vetter, J.S.; Li, D. LastingNVCache: A Technique for Improving the Lifetime of Non-volatile Caches. In Proceedings of the IEEE Computer Society Annual Symposium on VLSI (ISVLSI), Tampa, FL, USA, 9–11 July 2014.
14. Mittal, S.; Zhang, Z.; Vetter, J. FlexiWay: A Cache Energy Saving Technique Using Fine-grained Cache Reconfiguration. In Proceedings of the 31st IEEE International Conference on Computer Design (ICCD), Asheville, NC, USA, 6–9 October 2013; pp. 100–107.
15. Jaleel, A.; Theobald, K.B.; Steely, S.C., Jr.; Emer, J. High performance cache replacement using re-reference interval prediction (RRIP). *ACM Sigarch Comput. Archit. News* **2010**, *38,* 60–71.
16. Mittal, S. Using Cache-coloring to Mitigate Inter-set Write Variation in Non-volatile Caches. In *Technical Report*; Iowa State University: Ames, IA, USA, 2013.
17. Li, Y.; Jones, A.K. Cross-Layer Techniques for Optimizing Systems Utilizing Memories with Asymmetric Access Characteristics. In Proceedings of the IEEE Computer Society Annual Symposium on VLSI (ISVLSI), Amherst, MA, USA, 19–21 August 2012; pp. 404–409.
18. Quan, B.; Zhang, T.; Chen, T.; Wu, J. Prediction Table based Management Policy for STT-RAM and SRAM Hybrid Cache. In Proceedings of the 7th International Conference on Computing and Convergence Technology (ICCCT), Seoul, South Korea, 3–5 December 2012; pp. 1092–1097.
19. Mittal, S.; Vetter, J.S. EqualWrites: Reducing Intra-set Write Variations for Enhancing Lifetime of Non-volatile Caches. *IEEE Trans. VLSI Syst.* **2016**, *24*, 103–114.
20. Mittal, S.; Vetter, J.S. EqualChance: Addressing Intra-set Write Variation to Increase Lifetime of Non-volatile Caches. In Proceedings of the 2nd USENIX Workshop on Interactions of NVM/Flash with Operating Systems and Workloads (INFLOW), Broomfield, CO, USA, 5 October 2014.
21. Mittal, S.; Vetter, J.S.; Li, D. WriteSmoothing: Improving Lifetime of Non-volatile Caches Using Intra-set Wear-leveling. In Proceedings of the ACM Great Lakes Symposium on VLSI (GLSVLSI), Houston, TX, USA, 21–23 May 2014.
22. Carlson, T.E.; Heirman, W.; Eeckhout, L. Sniper: Exploring the Level of Abstraction for Scalable and Accurate Parallel Multi-Core Simulations. In Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis (SC), Seattle, WA, USA, 12–18 November 2011.
23. Poremba, M.; Mittal, S.; Li, D.; Vetter, J.S.; Xie, Y. DESTINY: A Tool for Modeling Emerging 3D NVM and eDRAM caches. In Proceedings of the Design Automation and Test in Europe, Grenoble, France, 9–13 March 2015.

24. Mittal, S.; Poremba, M.; Vetter, J.; Xie, Y. Exploring Design Space of 3D NVM and eDRAM Caches Using DESTINY Tool. In *Technical Report ORNL/TM-2014/636*; Oak Ridge National Laboratory: Oak Ridge, TN, USA, 2014.
25. Intel. Available online: http://ark.intel.com/products/75122/Intel-Core-i7-4770-Processor-8M-Cache-up-to-3_90-GHz (accessed on 13 January 2016).
26. AMD. Available online: http://www.cpu-world.com/CPUs/Bulldozer/AMD-Opteron%206328%20-%20OS6328WKT8GHK.html (accessed on 13 January 2016).
27. Ahuja, V.; Farrens, M.; Ghosal, D. Cache-aware affinitization on commodity multicores for high-speed network flows. In Proceedings of the Symposium on Architectures for networking and communications systems, Austin, TX, USA, 29–30 October 2012; pp. 39–48.
28. Mittal, S. A Survey of Architectural Techniques For Improving Cache Power Efficiency. *Elsevier Sustain. Comput.: Inform. Syst.* **2014**, *4*, 33–43.
29. Chang, M.T.; Rosenfeld, P.; Lu, S.L.; Jacob, B. Technology Comparison for Large Last-Level Caches (L^{3}Cs): Low-Leakage SRAM, Low Write-Energy STT-RAM, and Refresh-Optimized eDRAM. In Proceedings of the International Symposium on High-Performance Computer Architecture (HPCA), Shenzhen, China, 23–27 February 2013.
30. Mittal, S.; Vetter, J. A Survey Of Techniques for Architecting DRAM Caches. *IEEE TPDS* **2015**, *PP*, doi:10.1109/TPDS.2015.2461155.
31. Sun, G.; Dong, X.; Xie, Y.; Li, J.; Chen, Y. A novel architecture of the 3D stacked MRAM L2 cache for CMPs. In Proceedings of the 15th IEEE International Symposium on High Performance Computer Architecture (HPCA), Raleigh, NC, USA, 14–18 February 2009; pp. 239–249.
32. Ahn, J.; Choi, K. Lower-bits cache for low power STT-RAM caches. In Proceedings of the IEEE International Symposium on Circuits and Systems (ISCAS), Seoul, South Korea, 20–23 May 2012; pp. 480–483.
33. Park, S.P.; Gupta, S.; Mojumder, N.; Raghunathan, A.; Roy, K. Future cache design using STT MRAMs for improved energy efficiency: Devices, circuits and architecture. In Proceedings of the 49th Annual Design Automation Conference (DAC'12), San Francisco, CA, USA, 3–7 June 2012; pp. 492–497.
34. Jung, J.; Nakata, Y.; Yoshimoto, M.; Kawaguchi, H. Energy-efficient Spin-Transfer Torque RAM cache exploiting additional all-zero-data flags. In Proceedings of the 2013 14th International Symposium on Quality Electronic Design (ISQED), Santa Clara, CA, USA, 4–6 March 2013; pp. 216–222.
35. Mittal, S.; Vetter, J.S. AYUSH: A Technique for Extending Lifetime of SRAM-NVM Hybrid Caches. *IEEE Comput. Archit. Lett. (CAL)* **2015**, *15*, 115–118.
36. Li, Q.; Zhao, M.; Xue, C.J.; He, Y. Compiler-assisted preferred caching for embedded systems with STT-RAM based hybrid cache. *ACM Sigplan Not.* **2012**, *47*, 109–118.
37. Mittal, S. A Survey of Power Management Techniques for Phase Change Memory. *Int. J. Comput. Aided Eng. Technol. (IJCAET)* **2016**, in press.
38. Mehta, S.; Fang, Z.; Zhai, A.; Yew, P.C. Multi-stage coordinated prefetching for present-day processors. In Proceedings of the International Conference on Supercomputing, Muenchen, Germany, 10–13 June 2014; pp. 73–82.