LastingNVCache: A Technique for Improving the Lifetime of Non-volatile Caches

Sparsh Mittal[⋆], Jeffrey S. Vetter[⋆§], Dong Li[⋆]
[⋆]*Oak Ridge National Laboratory*
[§]*Georgia Institute of Technology*
Oak Ridge, Tennessee, USA
Email: {mittals,vetter,lid1}@ornl.gov

***Abstract*—Use of non-volatile memory (NVM) devices such as resistive RAM (ReRAM) and spin transfer torque RAM (STT-RAM) for designing on-chip caches holds the promise of providing a high-density, low-leakage alternative to SRAM. However, low write endurance of NVMs, along with the write-variation introduced by existing cache management schemes significantly limits the lifetime of NVM caches. We present LastingNVCache, a technique for improving the cache lifetime by mitigating the intra-set write variation. LastingNVCache works on the key idea that by periodically flushing a frequently-written data-item, next time the block can be made to load into a cold block in the set. Through this, the future writes to that data-item can be redirected from a hot block to a cold block, which leads to improvement in the cache lifetime. Microarchitectural simulations have shown that LastingNVCache provides 6.36X, 9.79X, and 10.94X improvement in lifetime for single, dual and quad-core systems, respectively. Also, its implementation overhead is small and it outperforms a recently proposed technique for improving lifetime of NVM caches.**

***Keywords*-Non-volatile memory (NVM), microarchitectural technique, device lifetime, write-endurance, wear-leveling, intra-set write variation.**

I. Introduction

Recent trends of increasing system core-count and memory bandwidth-wall problem have led to a large increase in the size of last level caches (LLCs), for example, Intel's Enterprise Xeon processor uses 30 MB LLC [1]. Conventionally, SRAM has been used for designing LLCs because it provides fast access speed and high write endurance. However, SRAM consumes high leakage power and has low density, and hence, SRAM LLCs contribute significantly to the chip power and area budget. For example, L2 cache consumes 25% of the total processor power in the Niagara-2 processor [2]. As cache capacity requirement increases, the increased power consumption will necessitate expensive cooling solutions (e.g. liquid cooling) and may also severely restrict further performance scaling.

To address this issue, researchers have explored non-volatile memory, such as ReRAM (resistive RAM), STT-RAM (spin transfer torque RAM) and PCM (phase change RAM) for designing on-chip caches [3–7]. NVMs provide high density, low-leakage power and non-volatile operation. As an example, the size of an SRAM cell ranges from 125–$200F^2$, while that of a ReRAM cell ranges from 4–$10F^2$ [8–10]. However, a critical limitation of NVMs is that their write endurance value is orders of magnitude smaller than that of SRAM and DRAM. Specifically, while the write endurance of SRAM and DRAM is more than 10^{15} [8], for ReRAM, STT-RAM and PCM, these values are 10^{11}, 4×10^{12} and 10^8, respectively [8, 11–13]. Due to process variations, these values may be further reduced by an order of magnitude [14].

Further, since existing cache management policies are write-variation unaware, the large amount of write variation introduced by them may significantly reduce the cache lifetime compared to the expected lifetime assuming uniform write distribution. For example, LRU (least recently used) replacement policy keeps the recently accessed data in the cache to leverage temporal locality, however, this may significantly increase the number of writes to the blocks storing those data-items.

In this paper, we propose **LastingNVCache**, a microarchitectural technique for improving cache lifetime by mitigating the intra-set write variation in NVM caches. LastingNVCache works on the key idea that if a frequently-written data-item is periodically flushed without updating its LRU-age information, the next time it will be loaded into a cold block in the set. Through this, the future writes to that data-item can be redirected from a hot block to a cold block, which leads to intra-set wear-leveling. Thus, by uniformly distributing the write-pressure, the worst-case writes on any block can be reduced which leads to improvement in the lifetime of the cache (Section III).

The storage requirement of LastingNVCache is less than 0.8% of the L2 cache size, which is very small. Also, it does not require changing the set-decoding (unlike [4]), or extra swap-buffers for in-cache data movement (unlike [6, 15]), or compiler analysis (as in [16]) or including tag bits as part of set-index (unlike [3]). In this paper, we assume a ReRAM cache and based on the explanation, LastingNVCache can be easily applied to LLCs designed with other NVMs. For sake of convenience, we henceforth use the terms ReRAM and NVM interchangeably.

Microarchitectural simulations have been performed using an x86-64 simulator and benchmarks from SPEC2006 suite and HPC (high-performance computing) field IV. Also, we compare LastingNVCache with a recently proposed technique for improving lifetime of NVM caches, namely PoLF

978-1-4799-3765-3 2014

DOI 10.1109/ISVLSI.2014.69

[11]. Results have shown that LastingNVCache provides higher improvement in lifetime than PoLF and also incurs smaller loss in performance and energy. For single, dual and quad-core systems, the average improvement in lifetime on using LastingNVCache is 6.36×, 9.79×, and 10.94×, respectively (Section V). Additional experiments confirm that LastingNVCache works well for a wide range of system and algorithm parameters.

II. Background and Related Work

The raw lifetime of cache is determined by the first memory-cell that wears out and thus, the lifetime of cache can be maximized by minimizing the worst-case write count to a cache line. This can be achieved by either minimizing the number of writes to the cache or by uniformly distributing them over cache (called wear-leveling). Some write-minimization techniques reduce the number of writes at cache-access level by using buffers or additional level of caches [7, 17], while others avoid redundant writes at bit-level [5, 18]. Our technique uses wear-leveling and can be synergistically integrated with write-minimization techniques.

Based on the granularity of wear-leveling, different techniques can be divided into set-level (or cache color-level) [3, 4, 11], way-level [11, 15] or memory-cell level [5]. Our technique uses way-level wear-leveling, which unlike set-level and color-level wear-leveling, does not require changing the set-decoding of the cache or flushing the contents of cache on reconfiguration and thus, incurs smaller overhead.

Wang et al. [11] propose an intra-set wear-leveling technique named PoLF (probabilistic set-line flush). In PoLF, after a fixed number of write hits (called flush threshold 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. Probabilistically, the flushed block is expected to be hot and hence, the hot data-item will be loaded in another cold block which leads to intra-set wear-leveling. We both qualitatively and quantitatively compare our technique with PoLF in Sections III and V.

III. Methodology

Notations: Our notation is as follows. N shows the number of cores. For L2 cache, S, A, B and G denote the number of sets, associativity, cache block (line) size and tag size, respectively. In this paper, we assume B = 64 bytes and G = 40 bits. We use the terms ‘block’ and ‘line’ synonymously and similarly for ‘flushing’ and ‘invalidation’.

A. Main Idea

Caches work by exploiting the temporal locality principle, by which frequently accessed data are kept in the cache to improve the hit-rate. However, if few cache blocks are repeatedly written, the number of writes to them may become much larger than those to the remaining blocks in the set.

LastingNVCache works on the key idea that if after a fixed number of writes to a cache block, the data-item stored in that block is invalidated, then the storage location of the hot data-item in the set can be changed, and thus, the future writes can be redirected to another ‘cold’ block-location. This helps in achieving wear-leveling which improves the lifetime of the cache.

B. Implementation Details

Algorithm 1 shows the working of LastingNVCache. We now describe it in detail.

Algorithm 1: LastingNVCache: algorithm for handling a write-hit in set-index i

```
Let $k$ be the index of the write-hit block
Increment nWrite[$i$][$k$] by 1
if nWrite[$i$][$k$] == $\Phi$ then
    Write-back incoming data and invalidate cacheData[$i$][$k$]
    for all blocks $j$ in set $i$ do
        if $j \neq k$ AND nWrite[$i$][$j$]>0 then
            Reduce nWrite[$i$][$j$] by $\lambda$
        end
    end
else
    Write incoming data to cacheData[$i$][$k$], mark dirty and update
    LRU-age information
end
```

We use a parameter Φ, which denotes the flushing threshold. With each cache block, we use a counter (termed nWrite), which records the number of writes to the block in its current generation, i.e. from the point of time when the existing data-item was stored in that block-location. On each write-hit to a block, its nWrite counter is incremented by 1. When on a write, the nWrite value reaches Φ, the write is skipped and the data-item is directly written back to memory, without updating the LRU information of the block. Thus, any subsequent miss will invalidate the actual LRU block and store the hot data in it, and not in the above mentioned invalidated block. Effectively, the cache-block invalidation induced by LastingNVCache works to change the location of hot block and thus, distribute the write-pressure uniformly.

In LastingNVCache, before a block is invalidated, writes to it may not happen in a short window of time. This is because the write counters accumulate writes over a complete generation of a block. In such a case, when a block is actually invalidated, it may not “currently” be a hot block. To address this, and actually capture the temporal locality, we proceed as follows. When a cache block is invalidated, the write-counter of all the other blocks in the set is reduced by λ, which is a tunable parameter. Typical values of λ are 0, 1 and 2. By using $\lambda > 0$, we ensure that all or most of the write operations to a block have taken place in a ‘recent’ window of time. This is because, if another block

Table I: The behavior of a 4-way cache set for an access pattern under LRU, PoLF and LastingNVCache schemes. A to F are valid data-items, 'X' shows invalid data-item. The numbers after data-items show LRU-age, 0 being the MRU and 3 being the LRU. Cache block being written is marked in **bold**. Note that LastingNVCache distributes write more uniformly than PoLF (see '# writes' row).

	LRU Policy		PoLF Scheme (FT=3)		LastingNVCache Scheme (Φ=3, λ=0)	
Command	(Data LRU-age)$_4$	Result	(Data LRU-age)$_4$	Result	(Data LRU-age)$_4$	Result
Initial status	A 0 B 1 C 2 D 3		A 0 B 1 C 2 D 3		A 0 B 1 C 2 D 3	
Write B	A 1 **B** 0 C 2 D 3	Hit	A 1 **B** 0 C 2 D 3	Hit	A 1 **B** 0 C 2 D 3	Hit
Write B	A 1 **B** 0 C 2 D 3	Hit	A 1 **B** 0 C 2 D 3	Hit	A 1 **B** 0 C 2 D 3	Hit
Read E	A 2 B 1 C 3 **E** 0	Miss	A 2 B 1 C 3 **E** 0	Miss	A 2 B 1 C 3 **E** 0	Miss
Write E	A 2 B 1 C 3 **E** 0	Hit	A 2 B 1 C 3 X 0	Hit, Flush E	A 2 B 1 C 3 **E** 0	Hit
Read F	A 3 B 2 **F** 0 E 1	Miss	A 3 B 2 **F** 0 X 1	Miss	A 3 B 2 **F** 0 E 1	Miss
Write B	A 3 **B** 0 F 1 E 2	Hit	A 3 **B** 0 F 1 X 2	Hit	A 3 X 2 F 0 E 1	Hit, Flush B
Write B	A 3 **B** 0 F 1 E 2	Hit	A 3 **B** 0 F 1 X 2	Hit	**B** 0 X 3 F 1 E 2	Miss
Read F	A 3 B 1 **F** 0 E 2	Hit	A 3 B 1 F 0 X 2	Hit	B 1 X 3 F 0 E 2	Hit
Write B	A 3 **B** 0 F 1 E 2	Hit	A 3 X 1 F 0 X 2	Hit, Flush B	**B** 0 X 3 F 1 E 2	Hit
Write B	A 3 **B** 0 F 1 E 2	Hit	**B** 0 X 2 F 1 X 3	Miss	X 0 X 3 F 1 E 2	Hit, Flush B
# writes	0 6 1 2		1 4 1 1		2 2 1 2	
Summary	8 hits, 2 misses, 9 writes		7 hits, 3 misses, 7 writes, 2 flushes		7 hits, 3 misses, 7 writes, 2 flushes	

in the set has been recently flushed, then the write-counter of this block would be reduced by λ, and hence, more writes need to happen to this block to reach the Φ limit. By virtue of this, we ensure that we invalidate based on the number of writes *relative* to other blocks (which actually indicates high intra-set write variation) and not based on the *absolute* number of writes alone. A value of $\lambda = 0$ indicates that the writes to a block are accounted over its one complete generation, irrespective of the writes to other blocks in the set. The higher the value of λ, the higher are chances that a block being invalidated has seen large number of writes in recent interval. We have chosen to study these values of λ since they are reasonable and small and thus help us in exercising moderate control in aggressiveness of cache flushing and wear-leveling.

Table I compares it with LRU and PoLF scheme for an example access pattern. It can be easily understood based on the explanation provided above and in Section II.

C. Salient Features

LastingNVCache provides several key advantages over PoLF. LastingNVCache records the number of writes to each cache block in its current generation and flushes a block only if the block alone has accumulated Φ number of writes. If before that, the block is evicted, another block will be installed which will have nWrite initialized to 0 and its flushing will be postponed. By contrast, PoLF blindly flushes 1/FT fraction of write hits and hence, with PoLF, even a newly-installed block may be flushed (see Table I) since PoLF flushes in a probabilistic manner. Also, there is no guarantee that the block chosen for invalidation currently stores a hot data-item. Due to this, for workloads which have high write-intensity but low write-variation, PoLF may lead to unnecessarily large number of invalidations leading to performance and energy loss.

D. Overhead Assessment

LastingNVCache does not record all the writes on a block, rather, it only records writes in a single generation and thus, its storage requirement is small. For each cache block, we use $\lceil \log_2(\Phi) \rceil$ bits to store the number of writes on it. Thus, the percentage overhead of LastingNVCache, compared to the L2 cache can be computed as

$$Overhead = \frac{S \times A \times \log_2(\Phi)}{S \times A \times (B + G)} \times 100 \qquad (1)$$

For $\Phi = 16$, we obtain $Overhead$ = 0.72%, and thus, the storage overhead of LastingNVCache is small.

For both PoLF and LastingNVCache, we assume that on each write-hit, comparison of write-counter with threshold value takes 3 cycles, since the write-counter has only few (e.g. 4-5) bits. Note that due to instruction-level parallelism (ILP), a small increase in latency of LLC can be easily hidden, as confirmed by the results. For LastingNVCache, when a data-item is flushed, the latency of reducing the write-counters of remaining blocks is hidden by the latency of write-back operation. The performance overhead of LastingNVCache comes from extra writebacks to the memory, which we model in our experiments. For further optimization, we assume that the overhead of writebacks can be hidden by using write-back buffers and MSHR (miss-status holding registers) techniques. The counters used for measuring the number of writes are not designed NVM and hence, they do not have write-endurance issues. If required, to minimize the dynamic power consumption of counters, Gray counters can be utilized.

IV. Experimental Methodology

Simulation Platform: We use Sniper x86-64 simulator [19] for performing microarchitectural simulation. The processor frequency is 2GHz. Both L1-I and L1-D caches are 4-way 32KB caches with 2 cycle latency. We assume that L1 caches are designed using SRAM for performance reasons. L2 cache parameters are shown in Table II, which are obtained using NVSim [10], assuming 32nm process, write energy-delay product (EDP) optimized cache design and sequential cache access. All caches use LRU, write-back, write-allocate policy and L2 cache is inclusive of L1 caches. L1 caches are private to each core and L2 cache is shared among cores. Main memory latency is 220 cycles. Memory bandwidth for 1-core, 2-core and 4-core systems is 10, 15 and 25GB/s, respectively.

Table II: Parameters for 16-way ReRAM L2 cache

	2MB	4MB	8MB	16MB	32MB
Hit Latency (ns)	5.059	5.120	5.904	5.974	8.100
Miss Latency (ns)	1.732	1.653	1.680	1.738	2.025
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

Workloads: As single-core workloads, we use all 29 benchmarks from SPEC CPU2006 suite with *ref* inputs and 6 benchmarks from HPC field (shown in italics in Table III). Using these, we create 18 dual-core and 9 quad-core multiprogrammed workloads such that each benchmark is used exactly once (except for completing the left-over group). Table III shows these workloads.

Table III: Workloads used in the experiments

Single-core workloads 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)
Dual-core workloads (Using acronyms shown above)
AsDl, GcBw, GmGr, SoXa, BzMc, OmLb, NdCd, CaTo, SpPo
LqMi, SjWr, LsZe, HmGa, GkH2, PePo, NeLu, MkXb, CoAm
Quad-core workloads (Using acronyms shown above)
AsGaXaLu, GcBzGrTo, CaWrMkMi, LqCoMcBw
LsSoSjH2, PeZeHmDl, GkPoGmNd, LbOmCdSp, AmXbNeGa

Evaluation Metrics: Our baseline is a ReRAM L2 cache, which uses LRU replacement policy, but does not use any wear-leveling technique. We model the energy of L2 and main memory. The leakage power and dynamic energy of main memory are taken as 0.18W and 70nJ/access, respectively [20] and the energy parameters for L2 are shown in Table II. We ignore the overhead of counters and buffers, since it is several orders of magnitude smaller compared to the memory subsystem (L2 + main memory), as also confirmed by previous works [21]. We show the results on **a)** relative cache lifetime where the lifetime is defined as the inverse of maximum writes on any cache block **b)** coefficient of intra-set write-variation, termed as IntraV [11], **c)** percentage energy loss **d)** weighted speedup [22], **e)** absolute increase in MPKI (miss-per-kilo-instructions) [20] and **f)** the number of flush operations (nFlush).

We fast-forward the benchmarks for 10B instructions and simulate each workload till the slowest application executes 300M instructions. In multi-core workloads, the benchmarks which complete early are allowed to run but their IPC is recorded only for the first 300M instructions, following well-established simulation methodology [22, 23]. Remaining metrics are computed for the entire simulation, since they are system-wide metrics (while IPC is a per-core metric) [20]. Relative lifetime and speedup values are averaged using geometric mean and the remaining metrics are averaged using arithmetic mean. We henceforth refer weighted speedup as the relative performance. For dual and quad-core systems, we have also computed fair speedup [20, 22] and observed their values to be nearly same as weighted speedup and thus, LastingNVCache does not cause unfairness. For brevity, we omit these results.

V. Results and Analysis

A. Main Results

Figures 1, 2 and 3 show the results. Here, the size of L2 for N =1, 2, and 4 are 4MB, 8MB and 16MB, respectively and Φ (for LastingNVCache) and FT (for PoLF) are both 16. For MPKI increase and nFlush, we omit the per-workload figures for brevity and only state the average. The value for nFlush for N = 1, 2 and 4 for LastingNVCache (resp. PoLF) are 25K (resp. 95K), 69K (resp. 219K) and 163K (resp. 533K), respectively. Increase in MPKI for N = 1, 2 and 4 for LastingNVCache (resp. PoLF) are 0.14 (resp. 0.30), 0.15 (resp. 0.31) and 0.15 (resp. 0.29) respectively. We now analyze the results.

For all configurations, LastingNVCache achieves higher improvement in lifetime than PoLF with much smaller number of invalidations and also smaller performance and energy loss. For some workloads, LastingNVCache improves the lifetime by more than 10×, for example, Po, Ga, Nd, Sj, BzMc, CoAm, AsGaXaLu, GkPoGmNd etc. As evident from figures, for N = 1, 2 and 4, LastingNVCache (resp. PoLF) reduces the IntraV from 139.6% to 38.1% (resp. 52.6%), from 137.0% to 36.0% (resp. 44.2%) and from 122.7% to 28.6% (resp. 34.2%), respectively. Clearly, LastingNVCache reduces intra-set write variation more effectively with smaller number of invalidations.

The lifetime enhancement achieved with a workload also depends on the write-variation originally present in the baseline. Thus, the highest amount of lifetime improvement is achieved for workloads which have highest amount of write-variation and vice-versa. Hence, for workloads such as Lq, Mi, Xb etc. which have small write variation, the improvement in lifetime can be achieved by other methods such as write-minimization (see Section II). For a few workloads, the performance shows very small improvement (instead of loss) on using LastingNVCache and PoLF, for example, for Hm, the relative performance with both LastingNVCache and PoLF are 1.002×. This happens because writebacks occur eagerly on cache flushing, and hence, they are less likely to stall the processor later on, as also observed by the previous works [21, 24].

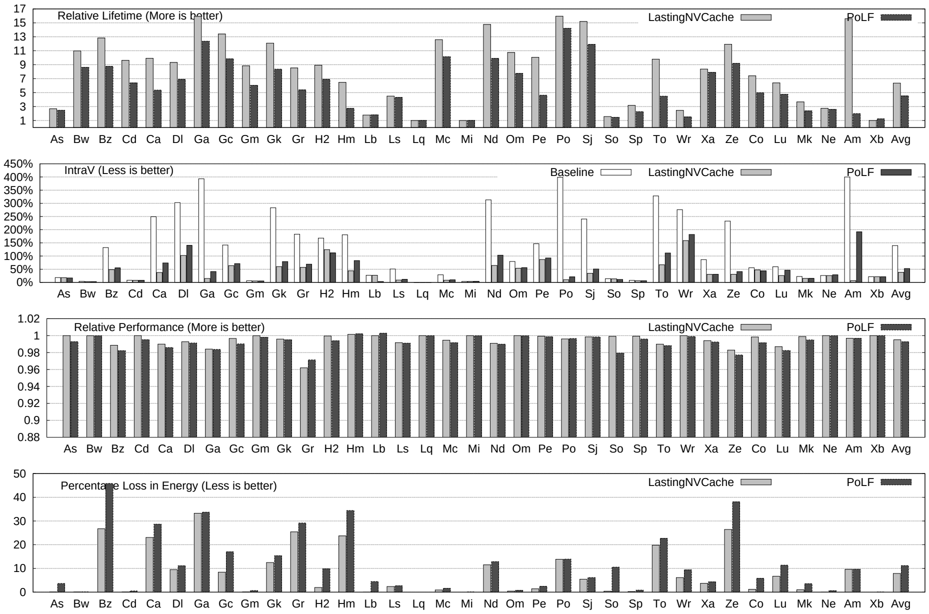

Figure 1: Results for Single-core System

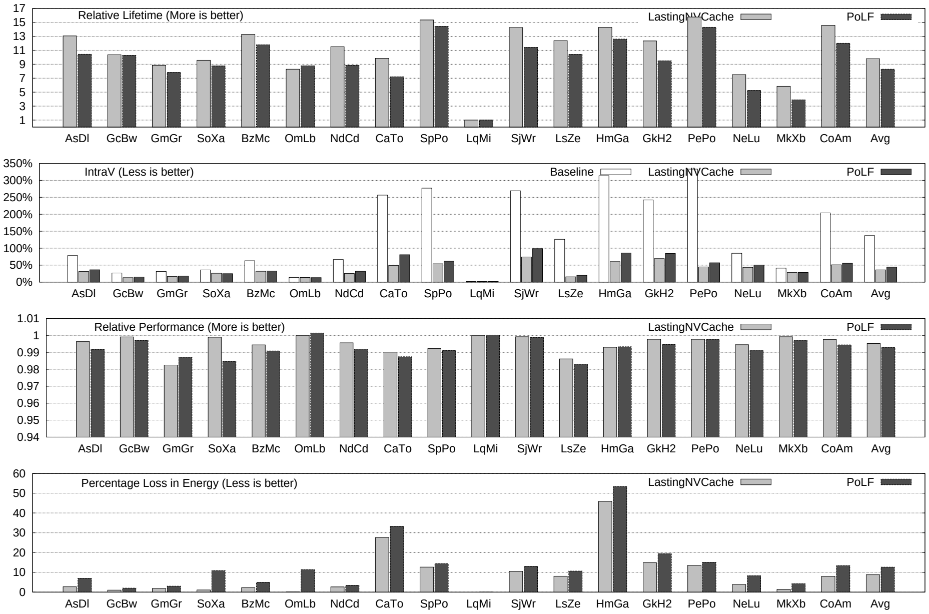

Figure 2: Results for Dual-core System

On average, PoLF flushes more than 3× the number of blocks as LastingNVCache and increases MPKI by nearly the double. For workloads such as Lb, which have *high write-intensity but low write-variation*, the advantage of LastingNVCache can be clearly seen. For these workloads, PoLF flushes much more number of blocks for achieving nearly the same improvement in lifetime as LastingNVCache. This clearly shows the advantage of LastingNVCache. The advantage of PoLF is that it only requires one global counter.

Both LastingNVCache and PoLF incur loss in energy due to flushing operations which increase the number of off-chip accesses. However, note that due to their high-density and low-leakage, NVMs facilitate use of larger sized LLCs than SRAM, which in general leads to better energy efficiency. Also, write-density minimization comes as a side-benefit of wear-leveling, which leads to reduced thermal density and chip-temperature. Hence, a small loss in energy due to the use of LastingNVCache may be acceptable, since our technique addresses the most crucial bottleneck

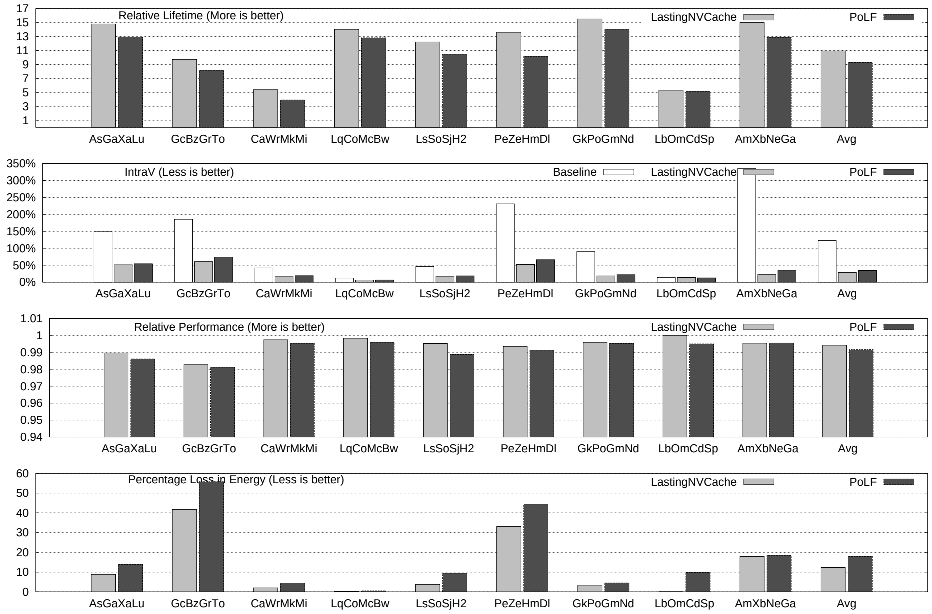

Figure 3: Results for Quad-core System

of NVMs, namely the small device-lifetime due to limited write-endurance. Further, as we show below and in Section V-B, by choosing a suitable value of Φ and λ, a designer can trade-off the desired improvement in performance and acceptable performance and energy loss.

Table IV shows the results with LastingNVCache and PoLF for different flushing threshold values. The higher the value of Φ (or FT), the smaller is the improvement in lifetime and the energy overhead of the technique. Conversely, for smaller value of flushing thresholds, higher improvement in lifetime is obtained at the cost of higher energy loss. Also note that LastingNVCache provides better results than PoLF on all configurations and parameters evaluated.

Table IV: Comparison of LastingNVCache (LNC) and PoLF (Rel. = relative, LfT = Lifetime, Perf. = performance.)

		Rel. LfT	Energy Loss %	Rel. Perf.	nFlush	Δ MPKI
Single-core System						
LNC	Φ=12	6.65	10.65	0.99	34K	0.20
PoLF	FT=12	4.83	14.84	0.99	127K	0.39
LNC	Φ=20	6.10	5.94	1.00	19K	0.11
PoLF	FT=20	4.29	8.98	0.99	76K	0.24
Dual-core System						
LNC	Φ=12	10.08	12.03	0.99	95K	0.21
PoLF	FT=12	8.74	16.88	0.99	285K	0.42
LNC	Φ=20	9.55	6.87	1.00	51K	0.11
PoLF	FT=20	8.32	10.12	0.99	171K	0.25
Quad-core System						
LNC	Φ=12	11.23	16.4	0.99	225K	0.20
PoLF	FT=12	9.93	23.66	0.99	711K	0.38
LNC	Φ=20	10.62	9.34	1.00	125K	0.11
PoLF	FT=20	8.99	14.41	0.99	427K	0.23

B. Parameter Sensitivity Results

We now focus exclusively on LastingNVCache and evaluate it for different parameters. Each time we only change one parameter compared to those mentioned in Section V-A and summarize the results in Table V.

Table V: LastingNVCache Parameter Sensitivity Study

	Rel. LfT	IntraV Base	IntraV LNC	Energy Loss %	Rel. Perf.	nFlush
Single-core System						
Default	6.36	139.6	38.1	7.88	1.00	25K
λ = 0	6.43	139.6	37.6	8.26	1.00	26K
λ = 2	6.35	139.6	38.7	7.64	1.00	24K
8-way	4.17	110.3	32.3	8.13	1.00	25K
32-way	8.54	170.4	43.0	6.95	1.00	24K
2MB	4.61	108.2	26.7	7.33	1.00	23K
8MB	7.65	179.3	60.6	7.38	0.99	28K
Dual-core System						
Default	9.79	137.0	36.0	8.76	1.00	69K
λ = 0	9.92	137.0	35.4	9.10	1.00	70K
λ = 2	9.82	137.0	36.6	8.56	1.00	68K
8-way	5.79	106.1	30.3	8.97	1.00	69K
32-way	15.77	174.4	41.2	8.27	1.00	66K
4MB	6.86	100.2	23.0	8.85	1.00	63K
16MB	11.04	171.6	53.1	9.79	1.00	83K
Quad-core System						
Default	10.87	122.7	28.8	12.05	0.99	161K
λ = 0	10.92	122.7	28.4	12.79	0.99	167K
λ = 2	10.87	122.7	28.8	12.05	0.99	161K
8-way	5.91	96.6	25.1	12.28	0.99	164K
32-way	19.96	152.9	31.1	11.31	0.99	155K
8MB	9.17	97.8	22.1	10.16	0.99	138K
32MB	12.83	154.1	39.7	12.39	0.99	200K

Change in λ: On decreasing λ, the lifetime enhancement is slightly increased at the cost of a small increase in the loss in performance and energy. This is expected, since for smaller λ, the aggressiveness of cache flushing and wear-leveling is increased.

Change in associativity: With increasing cache associativity, the intra-set write-variation also increases, as evident from the value of IntraV in Table V. Clearly, LastingNVCache provides large improvement in lifetime, in proportion to IntraV and is especially important for the caches of large associativity.

Change in cache size: With increasing cache size, the hit-rate also increases since workloads have fixed working set size. This increases the IntraV, since a few blocks see repeated hits. Hence, the lifetime improvement provided by LastingNVCache also increases. Since future systems are expected to have large LLCs, the importance of LastingNVCache will grow even further in next-generation systems.

Clearly, LastingNVCache works well for a wide range of system and algorithm parameters.

VI. Conclusion

NVM devices hold the promise of being used as a universal memory solutions in the future computing systems. However, some of their limitations such as low write-endurance present a crucial bottleneck in their use for designing on-chip caches. In this paper, we presented a technique for improving the lifetime of NVM caches by mitigating the intra-set write-variation. With only small implementation overhead, LastingNVCache provides large improvement in cache lifetime and also outperforms a recently proposed technique. Our future work includes integrating LastingNVCache with the techniques for mitigating inter-set write-variation for further improving the cache lifetime.

Acknowledgment

Support for this work was provided by U.S. Department of Energy, Office of Science, Advanced Scientific Computing Research. The work was performed at the Oak Ridge National Laboratory, which is managed by UT-Battelle, LLC under Contract No. DE-AC05-00OR22725 to the U.S. Government. Accordingly, the U.S. Government retains a non-exclusive, royalty-free license to publish or reproduce the published form of this contribution, or allow others to do so, for U.S. Government purposes.

References

[1] Intel, http://ark.intel.com/products/53580/.

[2] S. Li *et al.*, "McPAT: an integrated power, area, and timing modeling framework for multicore and many-core architectures," in *MICRO*, 2009, pp. 469–480.

[3] Y. Chen *et al.*, "On-chip caches built on multilevel spin-transfer torque RAM cells and its optimizations," *J. Emerg. Technol. Comput. Syst.*, vol. 9, no. 2, pp. 16:1–16:22, 2013.

[4] S. Mittal, "Using cache-coloring to mitigate inter-set write variation in non-volatile caches," Iowa State University, Tech. Rep., 2013.

[5] Y. Joo *et al.*, "Energy-and endurance-aware design of phase change memory caches," in *DATE*, 2010, pp. 136–141.

[6] X. Wu *et al.*, "Hybrid cache architecture with disparate memory technologies," in *ISCA*, 2009, pp. 34–45.

[7] G. Sun *et al.*, "A novel architecture of the 3D stacked MRAM L2 cache for CMPs," in *HPCA*, 2009.

[8] M. Qureshi *et al.*, *Phase change memory: From devices to systems*. Morgan & Claypool Publishers, 2011, vol. 6, no. 4.

[9] S.-S. Sheu *et al.*, "A 4Mb embedded SLC Resistive-RAM macro with 7.2 ns read-write random-access time and 160ns MLC-access capability," in *ISSCC*, 2011.

[10] X. Dong *et al.*, "NVSim: A circuit-level performance, energy, and area model for emerging nonvolatile memory," *IEEE TCAD*, 2012.

[11] J. Wang *et al.*, "i^{2}WAP: Improving non-volatile cache lifetime by reducing inter-and intra-set write variations," in *HPCA*, 2013.

[12] Y.-B. Kim *et al.*, "Bi-layered RRAM with unlimited endurance and extremely uniform switching," in *VLSIT*. IEEE, 2011, pp. 52–53.

[13] Y. Huai, "Spin-transfer torque MRAM (STT-MRAM): Challenges and prospects," *AAPPS Bulletin*, vol. 18, no. 6, pp. 33–40, 2008.

[14] W. Zhang and T. Li, "Characterizing and mitigating the impact of process variations on phase change based memory systems," in *MICRO*, 2009, pp. 2–13.

[15] S. Mittal *et al.*, "WriteSmoothing: Improving Lifetime of Non-volatile Caches Using Intra-set Wear-leveling ," in *ACM GLSVLSI*, 2014.

[16] Q. Li *et al.*, "Compiler-assisted preferred caching for embedded systems with STT-RAM based hybrid cache," *ACM SIGPLAN Notices*, pp. 109–118, 2012.

[17] J. Ahn and K. Choi, "Lower-bits cache for low power STT-RAM caches," in *ISCAS*, 2012, pp. 480–483.

[18] P. Zhou *et al.*, "Energy reduction for STT-RAM using early write termination," in *ICCAD*, 2009, pp. 264–268.

[19] T. E. Carlson *et al.*, "Sniper: Exploring the level of abstraction for scalable and accurate parallel multi-core simulations," in *SC*, 2011.

[20] S. Mittal *et al.*, "FlexiWay: A Cache Energy Saving Technique Using Fine-grained Cache Reconfiguration," in *ICCD*, 2013.

[21] S. Kaxiras *et al.*, "Cache decay: exploiting generational behavior to reduce cache leakage power," in *ISCA*, 2001.

[22] S. Mittal *et al.*, "MASTER: A Multicore Cache Energy Saving Technique using Dynamic Cache Reconfiguration," *IEEE TVLSI*, 2013.

[23] R. Manikantan *et al.*, "Probabilistic shared cache management (PriSM)," in *ISCA*, 2012, pp. 428–439.

[24] H.-H. S. Lee *et al.*, "Eager writeback-a technique for improving bandwidth utilization," in *MICRO*, 2000.