WriteSmoothing: Improving Lifetime of Non-volatile Caches Using Intra-set Wear-leveling

Sparsh Mittal, Jeffrey S. Vetter and Dong Li
Future Technologies Group, Oak Ridge National Laboratory
Oak Rige, Tennessee, USA 37830
{mittals,vetter,lid1}@ornl.gov

ABSTRACT

Driven by the trends of increasing core-count and bandwidth-wall problem, the size of last level caches (LLCs) has greatly increased. Since SRAM consumes high leakage power, researchers have explored use of non-volatile memories (NVMs) for designing caches as they provide high density and consume low leakage power. However, since NVMs have low write-endurance and the existing cache management policies are write variation-unaware, effective wear-leveling techniques are required for achieving reasonable cache lifetimes using NVMs. We present WriteSmoothing, a technique for mitigating intra-set write variation in NVM caches. WriteSmoothing logically divides the cache-sets into multiple modules. For each module, WriteSmoothing collectively records number of writes in each way for any of the sets. It then periodically makes most frequently written ways in a module unavailable to shift the write-pressure to other ways in the sets of the module. Extensive simulation results have shown that on average, for single and dual-core system configurations, WriteSmoothing improves cache lifetime by 2.17× and 2.75×, respectively. Also, its implementation overhead is small and it works well for a wide range of algorithm and system parameters.

Categories and Subject Descriptors

B.3.2 [**Hardware**]: MEMORY STRUCTURES—*Cache memories*; B.8.m [**Hardware**]: PERFORMANCE AND RELIABILITY—*Miscellaneous*

Keywords

Non-volatile memory; device lifetime; cache memory; intra-set write variation; wear-leveling; write endurance

1. INTRODUCTION

To meet the demands of increasing number of on-chip cores and circumvent the problem of memory bandwidth-wall, modern processors are using very large last level caches. For example, Intel's Itanium 9560 processor uses 32MB last level cache (LLCs) [1]. Conventionally, processor caches have been designed using SRAM since it provides low access latency and has high write endurance. However, SRAM has large leakage power consumption and hence, large LLCs designed with SRAM consume huge amount of power and chip area [13]. For example, the leakage power of the LLCs contributes to 63% and 56% of the total leakage power in Xeon Tulsa and Core 2 Penryn processors respectively, which corresponds to 30% and 20% of the total power in these processors [11].

To address this issue, researchers have explored use of non-volatile memories, such as resistive RAM (ReRAM), spin transfer torque RAM (STT-RAM) and phase change memory (PCM) for designing on-chip caches [6,9,18]. NVMs provide high density and scalability, consume very low leakage power and intrinsically avoid the need of refresh operations for maintaining data-integrity (unlike embedded DRAM devices) [12]. A crucial limitation of NVMs, however, is that their write endurance is orders of magnitude smaller than that of SRAM and DRAM. This low endurance value may lead to very small device lifetimes. For example, while the write endurance of SRAM and DRAM is in excess of 10^{15}, the write endurance of ReRAM, STT-RAM and PCM are only 10^{11}, 4×10^{12} and 10^{8}, respectively [8,10,16,18].

Further, the conventional cache management policies have been designed for optimizing performance and energy-efficiency, and they do not take the write endurance into account. In an attempt to leverage temporal locality, they may significantly increase the number of writes on a few cache blocks. This may cause those blocks to fail much earlier than the anticipated lifetime assuming uniform write distribution. As an example, conventional selective-way based cache reconfiguration approaches (e.g. [14]) only control the *number* of turned-off ways and do not control *which* ways will be selected for being turned-off. Thus, they are likely to keep the same ways turned-on (or turned-off) during the entire execution of the program which may exacerbate the problem of limited write-endurance. Similarly, with LRU (least-recently used) replacement policy, the most recently used data are expected to be repeatedly accessed and hence, the number of writes to the physical blocks which store these data is expected to increase much more than those in the remaining ways. To illustrate this, we take the example of povray benchmark from SPEC06 suite and execute it with a ReRAM L2 cache (the simulation parameters are shown in Section 5). We observe that due to write-variation, the cache lifetime is observed to be only **2.3 days**, although assuming an ideal uniform write-distribution to all L2 cache blocks, the lifetime would be **38.7 years**. This clearly shows the need of a wear-leveling technique.

In this paper, we present WriteSmoothing, a technique for improving cache lifetime by mitigating intra-set write variation. WriteSmoothing logically divides the cache-sets into multiple modules, for example, in a cache with 4096 sets and 32 modules, each module contains 128 sets. WriteSmoothing works on the following key

GLSVLSI'14, May 21–23, 2014, Houston, Texas, USA.
Copyright 2014 ACM 978-1-4503-2816-6/14/05 ...$15.00.
http://dx.doi.org/10.1145/2591513.2591525.

idea: if the intra-set write variation in a module is larger than a threshold, then the most heavily written cache ways can be temporarily made "unavailable" which will shift the write-pressure on the remaining ways. Different cache ways are made unavailable in rotation and this leads to wear-leveling which improves the cache lifetime (Section 3). WriteSmoothing does not require static profiling or modification of program binary and its overhead is very small (Section 4).

We perform exhaustive simulations using an x86-64 simulator and benchmarks from SPEC CPU2006 suite and HPC (high performance computing) field (Section 5). Results have shown that WriteSmoothing is effective in reducing the intra-set write variation which leads to increase in cache lifetime (Section 6). For single and dual-core systems, the average improvement in cache lifetime are 2.17× and 2.75×, respectively. Also, WriteSmoothing has very small effect on performance and energy efficiency. Additional experiments show that WriteSmoothing works well for different system and algorithm parameters.

2. BACKGROUND AND RELATED WORK

Improvement in lifetime of NVM caches can be obtained by reducing the number of writes and uniformly distributing them over different blocks (wear-leveling). The reduction in number of writes is achieved by using additional levels of caches [2] or write coalescing buffers [17] and avoiding redundant writes [9, 20]. These approaches are orthogonal and complementary to our technique, and hence, can be synergistically integrated with it.

Since caches show both inter-set and intra-set write variation [13], wear-leveling can be performed at the level of inter-set [5, 18] or intra-set [18]. In this paper, we propose an intra-set wear-leveling technique. Intra-set write variation increases with increasing associativity and it can be larger than inter-set write variation for some workloads hence, addressing it is extremely important ([18], also see Section 6). Further, WriteSmoothing can be integrated with the techniques for mitigating inter-set write variation and error correction/detection to improve the cache lifetime even further.

Wang et al. [18] propose an intra-set wear-leveling technique, which works by periodically flushing the block seeing a write-hit to change the cache block location of the hot data-item. A limitation of this technique is that it does not detect the write-variation present in the application and thus, it may blindly flush the cache. This is especially harmful for applications which have small write-variation but high write intensity. Moreover, while attempting to uniformly spread the writes to the cache, it may increase the writes on the main memory leading to contention, energy loss and endurance issues in main memory.

3. SYSTEM ARCHITECTURE

Background and Notations: We logically divide the cache-sets into multiple (e.g. 32) "modules", where each module contains several sets. We collect the number of cache writes for each way at the granularity of a single module. For example, if the cache has 32 modules, 4096 sets and 8 ways, then a group of 128 sets form one module and a total of 32×8 counters are used. The 8 counters for module 0 record the number of writes in each way for any of the sets numbering 0 to 127 and so on. We term the way of each module as a "sub-way".

Let S, A, B and T denote the number of cache sets, associativity, block-size and tag-size, respectively. In this paper, we assume, B = 64B and T = 40bits. Also, let $w_{i,j}$ denote the number of writes on any block at set i and way-index j. Further, let W_{avg} denote the average number of writes on all the blocks. Then, the coefficient of intra-set write variation (IntraV) for the entire cache is defined as follows [18],

$$IntraV = \frac{100}{S \cdot W_{avg}} \sum_{i=1}^{S} \sqrt{\frac{\sum_{j=1}^{A} \left(w_{i,j} - \sum_{r=1}^{A} w_{i,r}/A \right)^2}{A-1}} \quad (1)$$

Note that compared to [18], we express IntraV as percentage and hence, multiply the value by 100. Similarly, we can define the IntraV for each module (called ModuleIntraV). Let M denote the number of cache-modules and $e_{m,j}$ denote the writes on sub-way j of a module m. Let $E_{m,avg}$ denote the average writes on all the sub-ways of module m. Then, we have

$$ModuleIntraV[m] = \frac{100}{E_{m,avg}} \sqrt{\frac{\sum_{j=1}^{A} \left(e_{m,j} - \sum_{r=1}^{A} e_{m,r}/A \right)^2}{A-1}} \quad (2)$$

3.1 Key Idea

WriteSmoothing works on the idea that if the ModuleIntraV[m] for a module m is greater than a threshold λ, then the data in the most-frequently written (MFW) way can be transferred to that in the least-frequently written way and the MFW way can be temporarily made unavailable, which helps in shifting the future write-pressure to the remaining blocks. The MFW way is expected to store hot data, and thus, future writes are expected to be redirected especially to the least-frequently written way. If desired, the unavailable way can be turned-off to save leakage energy, however, we do not implement this in our experiments since NVMs consume negligible amount of leakage power. Although WriteSmoothing works to directly reduce *intra-set* write-variation, by virtue of working at the granularity of cache module, it accounts for *inter-set* write-variation also.

In a module, WriteSmoothing makes the same physical cache-way unavailable for all the sets, although in different sets of a module, the actual position of the most frequently written way may be different. To address this issue, we can minimize the number of cache sets in a module (i.e., $M = S$), such that each cache set has a set of counters to track each way. However, this incurs a large profiling overhead. Hence, the choice of M provides a balance between profiling overhead and accuracy. We study the sensitivity of wear-leveling to the choice of M in Section 6.2. The general conclusion based on our study is that for a reasonably large value of M (e.g. 32 or 64), its effect on wear-leveling is small.

3.2 Algorithm Description

Algorithm 1 shows the pseudo-code for WriteSmoothing, which runs after K cycles (e.g. K = 5 million) and can be a kernel module. The algorithm works as follows. For each module, ModuleIntraV is computed. Wear-leveling for any module is only performed if the ModuleIntraV is greater than λ. This helps in minimizing the algorithm overhead for workloads with small write-variation. The algorithm searches for the sub-way with the highest and the lowest $nWrite$ values and transfers the data from W_{max} sub-way to W_{min} sub-way. The TransferDataBetweenSubWays function copies valid data from the W_{max} sub-way to the W_{min} sub-way. If this data-item is clean, it is only copied if the destination data-item is invalid, otherwise it is flushed. If this data-item is dirty, it is copied regardless of the state of destination data-item. The reason for this is that the data-item at W_{max} location is expected to be hot and hence, keeping it in cache is likely to be beneficial. The Make-

Algorithm 1: Algorithm for WriteSmoothing

```
1  Let $IsAvailable[0:M-1][0:A-1]$ show whether a particular sub-way is available
2  Let $nWrite[0:M-1][0:A-1]$ denote the writes on each sub-way
3  Let $ModuleIntraV[0:M-1]$ denote the IntraV for each module (calculated from $nWrite$ values)
4  for each module $m$ do
5      Let $NumUnavailable[m]$ show the total number of unavailable ways in module $m$
6      if $ModuleIntraV[0:M-1] > \lambda$ then
7          Let $W_{max}$ be the sub-way with the highest number of writes where $IsAvailable[m][W_{max}]$ is TRUE
8          Let $W_{min}$ be the sub-way with the least number of writes
9          TransferDataBetweenSubWays($m$,$W_{max}$, $W_{min}$)
10         MakeUnavailable($m$, $W_{max}$)
11         /* If the number of unavailable ways in $m$ has become larger than $Z$, make the least written way available */
12         if $NumUnavailable[m] > Z$ then
13             Let $W_{low}$ be the sub-way with the least number of writes among all sub-ways $w$, such that $W_{low} \neq W_{min}$ and $IsAvailable[m][W_{max}]$ is FALSE
14             MakeAvailable($m$, $W_{low}$)
15         end
16     else
17         if $NumUnavailable[m] > 0$ then
18             Let $W_{low}$ be the sub-way with the least number of writes among all sub-ways $w$, such that $IsAvailable[m][W_{max}]$ is FALSE
19             MakeAvailable($m$, $W_{low}$)
20         end
21     end
22 end
```

Unavailable function write-backs the dirty data and flushes clean valid data of a sub-way and marks it as unavailable for the next interval. The MakeAvailable function simply marks a sub-way as available for the next interval.

If the number of unavailable ways in a module increases greater than Z, a single least-frequently written way is made available to keep the performance loss small. Note that at any point of time, the number of unavailable ways in different modules can be different. This is an important feature of our technique which helps in accounting for inter-set write variation and also keeping the performance loss small.

In processors with higher number of cores, not all the cores may run applications at the same time. Also, with private LLC or partitioned shared LLC, the cache space of each application is optimized for performance by exploiting temporal locality. In these scenarios, the write-variation is expected to be high and hence, we expect that with increasing number of cores, the benefits of WriteSmoothing will increase further.

4. IMPLEMENTATION AND OVERHEAD ASSESSMENT

Storage Overhead: We assume 40-bit counters for recording $nWrite$ values. Also, the data transfer between sub-ways is performed using a temporary buffer, as used in previous works [19] which has 128 registers, each 64B wide. These registers can also be used as intermediate storage for computing ModuleIntraV, since this happens in series (and not in parallel) with data-transfer. Thus, the percentage overhead of WriteSmoothing implementation compared to the L2 cache can be computed as

$$Overhead = \frac{(M \times A \times 40) + (128 \times 64 \times 8)}{S \times A \times (B + T)} \times 100 \qquad (3)$$

As an example, for a 4MB, 16-way cache with 32 modules, this $Overhead$ is only 0.24% of the L2 cache. Assuming some additional logic for computation and data-transfer, we conservatively take 0.5% as the upper bound of the overhead of WriteSmoothing, which is very small.

Latency and Energy Overhead: We assume that for each module, computing ModuleIntraV takes 40 cycles. For each module which undergoes wear-leveling, 60 cycles are consumed for selecting W_{max}. In Section 6.2 we also conduct experiments assuming $4\times$ higher overhead of computation and data-transfer and observe that the performance and energy loss of WriteSmoothing still remains small. Transfer of data takes $L_W + 4$ cycles, where L_W is the write latency of L2 NVM cache (shown in Table 1) and 2 cycle each is consumed in writing 64B data to and from the buffer over a 32B-wide bus. The extra writes due to algorithm execution are accounted in the number of L2 writes, used for computing energy, lifetime etc. Note that since the algorithm runs after a few million cycles, its overhead is easily amortized over the interval-length. Moreover, a small increase in latency of LLC is easily hidden by the instruction-level parallelism (ILP). Thus, WriteSmoothing has minimal effect on performance, as confirmed by our experiments (see Section 6.1).

5. EXPERIMENTAL METHODOLOGY

Simulation Infrastructure: We use interval-core model in Sniper x86-64 multicore simulator [3]. The frequency of processor is 2GHz. L1 I/D caches are 32KB 4-way LRU caches and are private to each core. L2 cache is shared among cores and its parameters are shown in Table 1, which are obtained using NVsim tool [7]. In this paper, we assume a ReRAM L2 cache, and based on it, WriteSmoothing can be easily applied to caches designed with other NVMs. Due to their properties, NVMs are more suitable to be used as last level caches and not first level caches. For this reason, in this paper, we assume that NVM is used for designing the L2 cache and apply WriteSmoothing algorithm in the L2 cache. The latency of main memory is 220 cycles. The peak memory bandwidth for single and dual-core systems are 10 and 15GB/s, respectively and contention is also modeled.

Table 1: Parameters for ReRAM L2 Cache

	2MB	4MB	8MB	16MB
Hit latency (ns)	4.33	4.13	4.74	6.21
Miss latency (ns)	1.47	1.44	1.55	1.81
Write latency (ns)	21.72	21.55	21.87	23.09
Hit Energy (nJ)	0.524	0.547	0.646	0.679
Miss Energy (nJ)	0.204	0.188	0.194	0.200
Write Energy (nJ)	0.834	0.851	0.925	0.967
Leakage Power (W)	0.204	0.325	0.785	1.118

Workloads: All 29 SPEC CPU2006 benchmarks with *ref* inputs and 5 benchmarks from HPC field (shown as italics in Table 2) are taken as single-core workloads. Using these, 17 dual-core multiprogrammed workloads are randomly created such that each benchmark is used exactly once. These workloads are shown in Table 2.

Evaluation Metrics: We show the results on 1.) Relative lifetime 2.) IntraV 3.) Weighted speedup (called relative performance) [14] 4.) Percentage energy loss and 5.) Absolute increase in MPKI (miss per kilo-instructions). The lifetime is defined as the inverse of the maximum number of writes on any block. We model

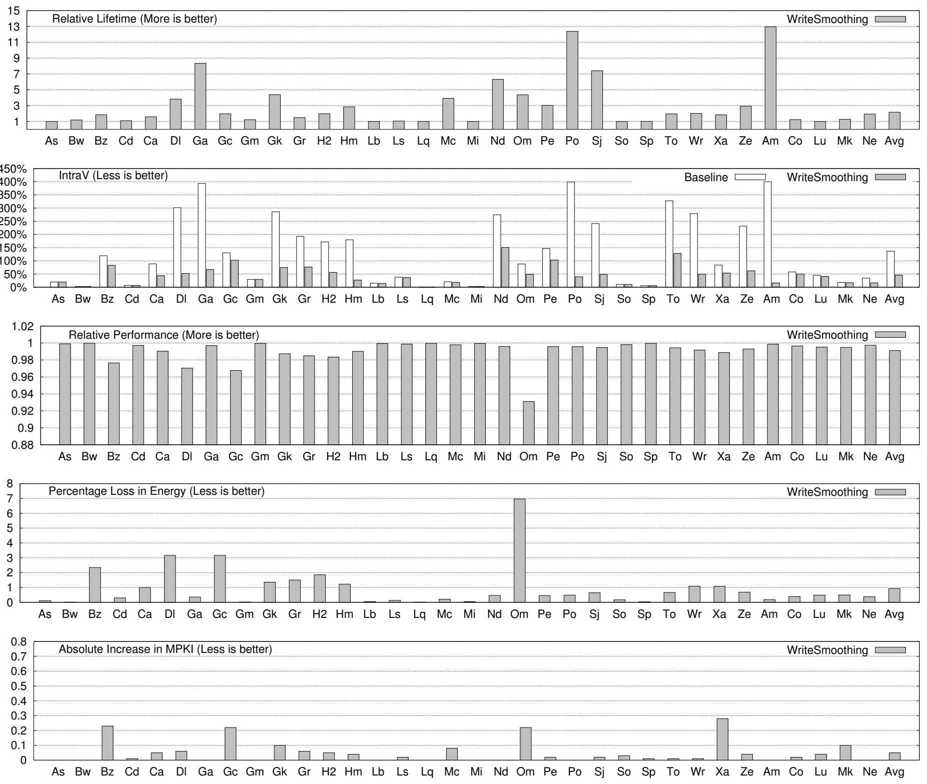

Figure 1: WriteSmoothing Results for Single-core System

Table 2: Workloads Used in the Paper

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), *Am(amg2013)*
Co(CoMD), Lu(LULESH), Mk(MCCK), Ne(Nekbone)
Dual-core workloads (Using acronyms shown above)
AsDl, GcGa, BzXa, LsLb, GkNe, OmGr, NdCd, CaTo
SpSo, LqPo, SjWr, GmMk, PeZe, HmH2, BwMi, McLu, CoAm

the energy of L2 cache, which is computed using parameters from Table 1 and includes the contribution of extra writes due to algorithm execution (see Section 4). The energy consumed by the counters is orders of magnitude smaller than that consumed by the L2 cache and hence, is ignored. For dual-core system, we have also computed the fair speedup [15] and have found the fair speedup to be almost the same as weighted speedup. Thus, WriteSmoothing does not cause unfairness. For brevity, we omit these results. Speedup values are averaged using geometric mean and the remaining metrics are averaged using arithmetic mean [15]. Simulations are performed till each core runs 500M instructions. In dual-core system, the program which finishes earlier is allowed to run, but its IPC is only recorded for the first 500M instructions. Remaining metrics are computed for the entire simulation (following well-established simulation methodology [4, 14, 15]).

6. EXPERIMENTAL RESULTS

6.1 Main Results

Figure 1 and 2 show the results for single and dual-core, respectively, which are obtained using the following parameter values: Z = 3, K = 5M cycles, λ =15%, 16-way set-associativity, 4MB L2 with M = 32 for single-core system and 8MB L2 with M = 64 for dual-core system. Our baseline is a cache which uses LRU replacement policy but does not use any wear-leveling technique.

We now analyze the results. Firstly, for some workloads, baseline IntraV can be as high as 400%, for example Ga (gamess), Po (povray), Am(amg2013). Also, we observe that on average, for single and dual-core system, 87.8% and 89.3% of the write accesses happen to just the MRU way of the 16-way cache (figure omitted for brevity). This highlights the need of using an intra-set wear-leveling technique for achieving reasonable cache lifetime. On average, for the single and the dual-core systems, improvement in lifetime are 2.17× and 2.75×, respectively. For some workloads, the improvement in lifetime is more than 10×, for example, Po, Am and LqPo (libquantum-povray). For a few other workloads, such as Ga, Sj (sjeng), GcGa (gcc-gamess), CoAm (CoMD-amg2013), the improvement in lifetime is more than 7×. This shows the effectiveness of WriteSmoothing.

WriteSmoothing reduces the IntraV from 136.6% to 45.7% for single-core system, and from 136.4% to 49.2% for dual-core system. As seen from the figure on relative lifetime and IntraV, the improvement in lifetime achieved depends on the intra-set variation present in the original application. By virtue of computing ModuleIntraV, WriteSmoothing performs shifting and incurs its overhead only when the intra-set write variation in original application is high. Thus, for applications such as Lb (lbm), Lq, Sp (sphinx), Mi (milc), LsLb (leslie3d-lbm) etc., WriteSmoothing does not incur performance or energy overhead. This feature is especially beneficial for workloads such as Lb, which have very high write intensity but low intra-set write variation. For the single and the dual-core systems, relative performance values are 0.99× and 0.99×, respectively. The reason WriteSmoothing maintains the performance

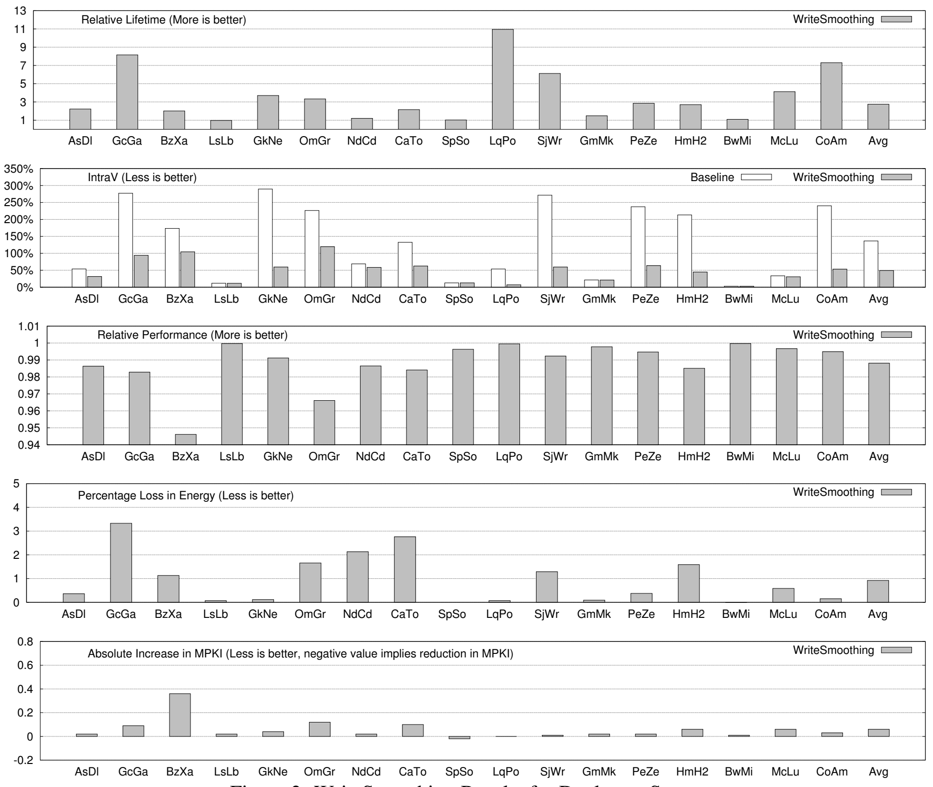

Figure 2: WriteSmoothing Results for Dual-core System

close to 1× is that it makes cache ways unavailable only when the intra-set write-variation is high, which happens when the application does not fully utilize the cache and a few ways remain unused. In such a case, making some ways unavailable does not significantly harm the performance. Only Om (omnetpp) and BzXa (bzip2-xalan) show relative performance less than 0.95×, since these workloads are very sensitive to L2 cache performance.

For the single and the dual-core systems, on average, increase in MPKI are 0.05 and 0.06, respectively and loss in energy are 0.93% and 0.92%, respectively. These values are very small and thus, WriteSmoothing does not increase the DRAM traffic appreciably, which is a significant improvement over previous data-invalidation based techniques (e.g. [5,18]), which lead to increased DRAM traffic. Our wear-leveling approach also provides the benefit of write density minimization [13] which can help in lowering the chip-temperature and cooling cost. Moreover, NVMs offer high density and low leakage power compared to SRAM and WriteSmoothing addresses the crucial limitation of NVMs, namely their limited write endurance. For these reasons, a small increase in miss-rate and energy may be acceptable. Further, as we show in Section 6.2, by changing the algorithm parameters (viz. M, λ, K and Z), a designer can strike a balance between acceptable algorithm implementation overhead and desired improvement in lifetime.

6.2 Parameter Sensitivity Results

We now evaluate the sensitivity of WriteSmoothing for different parameters. The results are summarized in Table 3. Except the parameter mentioned, the values of all parameters are same as used in the default case. For comparison purposes, the value with default case is also shown.

Change in Number of Modules (M): On increasing M, the granularity of wear-leveling is also increased leading to higher improvement in lifetime. However, this also leads to a small increase in the algorithm implementation overhead. Opposite is seen on reducing the number of modules.

Change in Threshold λ: Reducing λ increases the aggressiveness of wear-leveling, which increases the improvement in the lifetime at the cost of a small increase in energy loss, and vice-versa.

Change in Interval Size (K): Smaller value of interval size leads to more frequent wear-leveling which improves the cache lifetime, although it leads to a small increase in energy loss due to more frequent data-transfer. At 10M cycle interval size, the opportunity of wear-leveling is missed, although due to reduced data-transfer, the energy loss is also reduced.

Change in Maximum Unavailable Ways (Z): Change in Z does not monotonically affect the improvement in lifetime, since different applications have different associativity requirements and cache usage intensity. Change in Z has very small effect on the energy loss and increasing Z increases the energy loss since the effective cache associativity is reduced. Considering these factors, for a 16-way cache, a value of $Z = 3$ or $Z = 4$ is suitable.

Higher algorithm overhead: We evaluate WriteSmoothing assuming a latency overhead which is 4× that of shown in Section 4 (i.e. 160 cycles for computing ModuleIntraV and 240 cycles for selecting W_{max}). As shown in Table 3, the relative performance still remains 0.99× and energy loss is small (close to 1%). This confirms that, due to the reasons mentioned in Section 4, overhead of WriteSmoothing is quite small.

Change in Associativity (A): For a fixed capacity, a cache with lower associativity has higher miss-rate and smaller number of replacement candidates, which reduces the non-uniform distribution of writes, leading to smaller value of IntraV. This is evident from the values of IntraV for baseline cache and can also be understood by considering the extreme cases, viz. a direct-mapped cache and

Table 3: Parameter Sensitivity Results (Rel. = Relative, LT. = Lifetime, WrSm = WriteSmoothing, Perf. = Performance, Energy loss values are in percentage.). Default values are shown in Section 6.1.

	Rel. LT.	IntraV Base	IntraV WrSm	Rel. Perf.	Energy Loss	Δ MPKI
Single-core System						
Default	2.17	136.6	45.7	0.99	0.93	0.05
M =8	1.80	136.6	50.2	0.99	0.83	0.04
M =16	1.94	136.6	47.7	0.99	0.85	0.04
M =64	2.49	136.6	43.6	0.99	1.03	0.06
M =128	2.61	136.6	41.3	0.99	1.15	0.06
λ =10%	2.26	136.6	42.8	0.99	1.26	0.08
λ =20%	2.00	136.6	48.7	0.99	0.72	0.04
K =3M	2.18	136.6	44.3	0.99	1.06	0.05
K =10M	2.03	136.6	51.0	0.99	0.74	0.05
Z =2	2.07	136.6	48.2	0.99	0.83	0.04
Z =4	2.21	136.6	44.6	0.99	1.00	0.05
$\Uparrow$ Overhead	2.13	136.6	45.3	0.99	1.07	0.05
8-way	1.72	107.1	34.4	0.99	1.14	0.07
32-way	2.37	167.7	60.7	0.99	0.73	0.05
2MB L2	1.82	103.7	37.6	1.00	0.46	0.02
8MB L2	2.58	175.4	53.8	0.98	2.00	0.10
Two-core System						
Default	2.75	136.4	49.2	0.99	0.92	0.06
M =16	2.20	136.4	53.1	0.99	0.82	0.05
M =32	2.25	136.4	51.1	0.99	0.87	0.05
M =128	3.25	136.4	47.3	0.99	1.02	0.06
M =256	3.85	136.4	45.1	0.98	1.19	0.07
λ =10%	3.25	136.4	45.3	0.98	1.22	0.08
λ =20%	2.37	136.4	52.6	0.99	0.76	0.04
K =3M	2.75	136.4	49.2	0.99	1.00	0.06
K =10M	2.63	136.4	51.9	0.99	0.79	0.05
Z =2	2.61	136.4	51.0	0.99	0.85	0.05
Z =4	2.68	136.4	49.0	0.99	0.98	0.06
$\Uparrow$ Overhead	2.75	136.4	49.1	0.99	1.09	0.06
8-way	2.11	106.6	38.9	0.98	2.14	0.08
32-way	3.80	170.0	61.6	0.99	0.98	0.05
4MB L2	2.22	100.9	42.2	0.99	1.16	0.05
16MB L2	2.99	171.0	58.1	0.98	1.80	0.08

a fully-associative cache. WriteSmoothing works well for all associativity values and improves lifetime in proportion to the write-variation present in the baseline. For dual-core system with 32-way cache, the lifetime improvement is 3.8×.

Change in Cache Capacity: Since applications have fixed working set size, an increase in cache size improves the hit-rate and thus, only a few blocks are repeatedly accessed and cache evictions are reduced. This leads to higher write-variation, as evident from IntraV values. Depending on IntraV, WriteSmoothing provides large improvement in lifetime, with only small loss in performance and energy.

For all the above parameters, relative performance is greater than 0.97× and increase in MPKI is less than 0.11, which confirm that WriteSmoothing works well for a wide range of system and algorithm parameters.

7. CONCLUSION

Addressing the limitations posed by low write-endurance of NVMs is essential for making them a universal memory solution. In this paper, we presented WriteSmoothing, a technique for improving lifetime of non-volatile caches by minimizing intra-set write variation. Exhaustive evaluation over different benchmarks, algorithm and system parameters have shown that WriteSmoothing is effective in improving cache lifetime and incurs very small loss in performance and energy. Our future work will focus on integrating WriteSmoothing with write-minimization techniques to improve the cache lifetime even further.

8. REFERENCES

[1] `http://download.intel.com/newsroom/archive/Intel-Itanium-processor-9500_ProductBrief.pdf`.

[2] J. Ahn and K. Choi. Lower-bits cache for low power STT-RAM caches. In *ISCAS*, pages 480–483, 2012.

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

[4] M. Chaudhuri. Pseudo-LIFO: the foundation of a new family of replacement policies for last-level caches. In *MICRO*, 2009.

[5] Y. Chen et al. On-chip caches built on multilevel spin-transfer torque RAM cells and its optimizations. *J. Emerg. Technol. Comput. Syst.*, 9(2):16:1–16:22, 2013.

[6] X. Dong et al. Circuit and microarchitecture evaluation of 3D stacking magnetic RAM (MRAM) as a universal memory replacement. In *DAC*, pages 554–559, 2008.

[7] X. Dong et al. NVsim: A circuit-level performance, energy, and area model for emerging nonvolatile memory. *IEEE TCAD*, 31(7):994–1007, 2012.

[8] Y. Huai. Spin-transfer torque MRAM (STT-MRAM): Challenges and prospects. *AAPPS Bulletin*, 2008.

[9] Y. Joo et al. Energy-and endurance-aware design of phase change memory caches. In *DATE*, pages 136–141, 2010.

[10] Y.-B. Kim et al. Bi-layered RRAM with unlimited endurance and extremely uniform switching. In *VLSIT*, pages 52–53. IEEE, 2011.

[11] S. Li et al. CACTI-P: Architecture-level modeling for SRAM-based structures with advanced leakage reduction techniques. In *ICCAD*, pages 694–701, 2011.

[12] S. Mittal. A Cache Reconfiguration Approach for Saving Leakage and Refresh Energy in Embedded DRAM Caches. Technical report, Iowa State University, USA, 2013.

[13] S. Mittal. A survey of architectural techniques for improving cache power efficiency. *Sustainable Computing: Informatics and Systems*, 2013.

[14] S. Mittal et al. FlexiWay: A Cache Energy Saving Technique Using Fine-grained Cache Reconfiguration. In *IEEE ICCD*, pages 100–107, 2013.

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

[16] M. K. Qureshi et al. *Phase change memory: From devices to systems*, volume 6. Morgan & Claypool Publishers, 2011.

[17] G. Sun et al. A novel architecture of the 3D stacked MRAM L2 cache for CMPs. In *HPCA*, pages 239–249, 2009.

[18] J. Wang et al. i^{2}WAP: Improving non-volatile cache lifetime by reducing inter-and intra-set write variations. In *HPCA*, pages 234–245, 2013.

[19] X. Wu et al. Hybrid cache architecture with disparate memory technologies. In *ISCA*, pages 34–45, 2009.

[20] P. Zhou et al. Energy reduction for STT-RAM using early write termination. In *ICCAD*, pages 264–268, 2009.