Addressing Inter-set Write-Variation for Improving Lifetime of Non-Volatile Caches

Sparsh Mittal and Jeffrey S. Vetter
Oak Ridge National Laboratory, TN, USA
{mittals,vetter}@ornl.gov

I. Introduction

It is well-known that the number of accesses to different cache-sets is non-uniform[1] [2]. An example of this is shown in Figure 1, where we plot the number of accesses to different sets in a 4MB cache when the SPEC2006 povray benchmark is run for 1B instructions. Clearly, most of the sets experience zero or near-zero writes, while a few sets experience as high as 166726 writes. This fact along with the small write-endurance of NVMs may lead to very small lifetime of NVM caches.

To address this, we propose a technique which minimizes inter-set write variation in NVM caches for improving its lifetime. Our technique uses cache coloring scheme [3] to add a software-controlled mapping layer between groups of physical pages (called memory regions) and cache sets. Periodically, the number of writes to different colors of the cache is computed and based on this result, the mapping of a few colors is changed to channel the write traffic to least utilized cache colors. This change helps to achieve wear-leveling.

II. Related Work

Wang et al. [4] present a technique for addressing inter-set write-variation in NVM caches. Their technique periodically swaps the cache sets in order to distribute the write-traffic uniformly. The technique proposed by Chen et al. [5] uses a register called remap register. At the end of each interval, their technique updates this register and uses XOR between this register and the set-index bit of cache address. Using this technique, the set-index of all addresses in the cache is changed, which helps in achieving wear-leveling.

III. Methodology

Key Idea: Our key idea is that physical addresses which are heavily (frequently) written should be periodically mapped to different sets of the cache, so that the number of writes to a single (or few) set(s) does not increase much more than the average value. To keep the profiling overhead small, we use the granularity of cache remapping as a single cache color.

Cache Coloring Scheme: To enable flexible mapping of physical addresses to cache sets, we use cache coloring scheme [3]. In this scheme, the cache is logically divided into N portions, called cache colors, where N is given by

$$N = \frac{CacheSize}{PageSize \times CacheAssociativity} \quad (1)$$

[1]This work has been published as a chapter in the book [1].

Second, the physical pages are divided into N *memory regions* based on the least significant bits of their physical page number. Cache coloring works by mapping a memory region (and hence, all physical pages in that region) to a unique color in the cache. A small mapping table is used which stores the cache color assigned to each memory region. By changing the mapping between physical pages and cache colors, a particular memory region can be mapped to the desired cache color. Thus, the cache can be reconfigured at the granularity of a single cache color.

Algorithm Details: The algorithm runs, after every K writes, if at least 3M cycles have elapsed since the last algorithm execution. This is done to ensure that the algorithm does not run too frequently for applications with very high write counts. If 3M cycles have not elapsed, the algorithm waits till 3M cycles have elapsed, and checks for the possible reconfiguration after next K writes. The symbols used in the description of the algorithm are shown in Table I.

TABLE I: Symbols used in the algorithm

Symbol	Meaning
Region[i]	Region number which is mapped to color i
nWriteGlobal[i]	Writes to color i till now
nWriteLI[i]	Writes to color i in last interval
AVG	Average of nWriteLI[i] for all colors
SDW	Standard deviation of nWriteLI[i]
nHigher	Number of colors with nWriteLI[i] $>$ AVG
λ	A threshold ($\leq$ N/2)
β	Another threshold
L1, L2	Two lists

MAX(A,B) function returns the maximum value out of A and B. Also, if (r1, c1) and (r2, c2) be the region-to-color mapping for two colors c1 and c2, then we define *swapping* their region-to-color mapping as obtaining a mapping (r1, c2) and (r2, c1). If c1 = c2, swapping function does not perform any action. The algorithm works as follows.

1) If SDW $<$ β, the variation in write-count is small. Return.
2) Maintain two lists. In one list L1, sort the colors based on decreasing value of nWriteLI[i]. In another list L2, sort the colors based on increasing value of nWriteGlobal[i].
3) Let nColorToSwap = MAX(nHigher,λ)
4) For all k, $1 \leq k \leq$ nColorToSwap {
   Swap the region-to-color mapping of color at location L1[k] and L2[k]. }

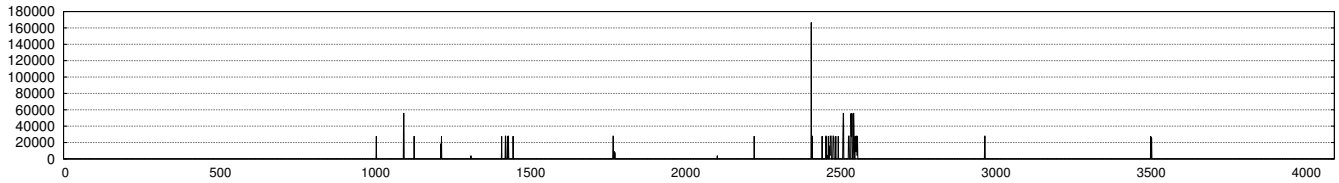

Fig. 1: Writes to the different cache sets for Povray benchmark

Explanation: If the write variation in the last interval is small, remapping is not performed. The algorithm attemps to reduce the write-pressure on those colors which were heavily used in the last interval. Through periodic remapping, this write-traffic is channeled to those colors which have seen the least number of writes till now. We maintain two lists: L1 and L2. L1 is sorted based on writes in the last interval and L2 is sorted based on accumulated writes till now. The value of nColorToSwap decides the number of maximum colors to be swapped in an interval. λ shows the fixed upper limit on nColorToSwap. Intuitively, reducing the write traffic of a color is expected to be useful only if the writes to it are higher than the average. When the mapping of a region to color is changed, the dirty blocks are written back to memory and the clean blocks are discarded.

Features and Limitations: Previous techniques which perform cache remapping at set-level [4], [5], do not collect per-set statistics, since that would incur large overhead. Thus, they blindly remap the whole cache periodically, which causes large flushing overhead. Our technique performs the mapping at coarse level to facilitate collecting profiling information with small overhead. As an example, a 16-way, 4MB cache has only 64 colors but 4096 sets. By changing the value of λ, β and algorithm interval, a trade-off can be achieved between performance loss and improvement in endurance. Since the algorithm runs after a large interval, its overhead is easily amortized over the interval length. Unlike [5], our technique uses number of writes to determine the interval length and not the number of cycles. Using this, our technique easily adapts itself based on the write-intensive nature of a program. By virtue of using dynamic profiling, it is suitable for use in product systems. Our cache coloring scheme does not require a change in underlying virtual address to physical address mapping, and thus can be implemented with little overhead. Also, the size of mapping table is extremely small.

The limitation of our technique is that it performs management only at level of cache color, which has several sets. For our experiments, a single cache color has 64 sets, and it may be possible that write-variation may occur within a single cache color. However, this fine-grain information is lost in the coarse grain migration.

IV. Experimental Methodology and Results

We use the interval core model in the Sniper x86-64 simulator. We use a 4MB, 16-way STT-RAM L2 cache and take its parameters from [6], assuming a retention time of 1 second. We use all 29 SPEC CPU2006 benchmarks with *ref* inputs (first three letters are used as acronyms) and simulate for 400M instructions.

Figure 2 shows the results on relative cache lifetime, where lifetime is defined as the inverse of the maximum number of writes on any cache block. We take $\beta = 75$ and $\lambda = N/4$. The average improvement in cache lifetime is $4.20\times$. The maximum amount of improvement in lifetime which is achieved depends on the write variation which was originally present. For this reason, several benchmarks, such as povray and sjeng, achieve a large improvement. We have also evaluated the performance of our technique and have found the average normalized IPC to be $0.96\times$.

Future work will focus on integrating our technique with error-correction/detection techniques to improve the cache lifetime even further.

References

[1] S. Mittal, *Architectural Techniques For Managing Non-volatile Caches*. Germany: Lambert Academic Publishing (LAP), 2013. [Online]. Available: https://www.morebooks.de/store/gb/book/architectural-techniques-for-managing-non-volatile-caches/isbn/978-3-659-46253-5

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

[3] S. Mittal *et al.*, "Palette: A cache leakage energy saving technique for green computing," in *HPC: Transition Towards Exascale Processing*, ser. Advances in Parallel Computing. IOS Press, 2013.

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

[5] Y. Chen *et al.*, "On-chip caches built on multilevel spin-transfer torque RAM cells and its optimizations," *ACM JETC*, vol. 9, no. 2, 2013.

[6] A. Jog *et al.*, "Cache revive: architecting volatile STT-RAM caches for enhanced performance in CMPs," in *DAC*, 2012, pp. 243–252.

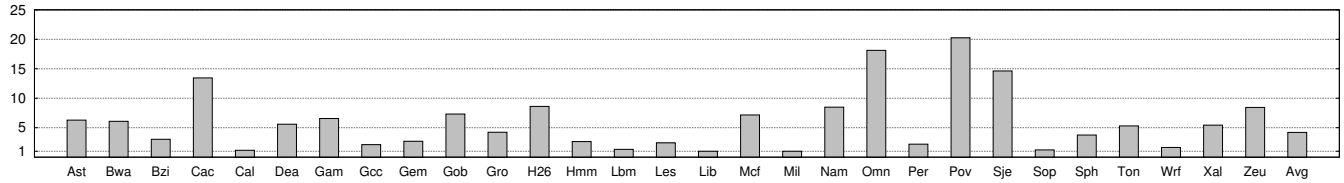

Fig. 2: Results on relative lifetime on using our technique