Adaptive Software Caching for Efficient NVRAM Data Persistence

Pengcheng Li[1] and Dhruva R. Chakrabarti[2]

[1] University of Rochester & Hewlett Packard Labs
[2] Hewlett Packard Labs

Abstract. Persistent memory is getting increasingly popular. However, the existence of transient CPU caches brings a serious performance issue for utilization of persistence. In particular, cache lines have to be flushed frequently to guarantee consistent, persistent program states. In this paper, we optimize data persistence by proposing a software cache. The software cache first buffers lines that need to be flushed, and then flushes them out at an appropriate later time. The software cache supports adaptive selection of the best cache size at run-time.

1 Introduction

Persistent memory or non-volatile memory (NVRAM) technologies, such as memristors and phase change memory (PCM), are increasingly popular. Persistent memory is byte-addressable and directly accessible (i.e., without DRAM buffers) with CPU loads and stores. Data in NVRAM will not be erased if the creator process does not clean it. It enables data reuse across system restarts and of course process restarts. This in-memory durability model can greatly change the programming paradigm for many applications [1].

A problem of NVRAM data persistence is the transient memories in current computer architectures, such as CPU caches. At any point of program execution, some of the updates to persistent memory may only reside in CPU caches and have not yet propagated to NVRAM. If there is a failure at this point of execution, the program state in NVRAM may not be consistent thus preventing full recovery.

Consistent persistent states are guaranteed by forcing all data out of caches to persistent memory in the event of any tolerated failure. In `Atlas` [1] programming model, a *failure-atomic section (FASE)* foresees a failure. A FASE is a code segment that changes program invariants. Either all or none of the updates in a FASE are visible in NVRAM. Therefore, persistent data is guaranteed to be consistent at the end of a FASE.

In this paper, we propose a software cache to reduce the number of cache line flushes. Its purpose is to cache the data writes and combine multiple writes into a single cache flush at the time of eviction. We flush all cache lines in the software cache at the end of a FASE. In addition, we develop a reuse-based locality theory that allows us to optimize it by choosing the best cache size.

For the purpose of disambiguity, if without explicit clarification of hardware cache, "cache" in this paper refers to the software cache.

2 Software Cache

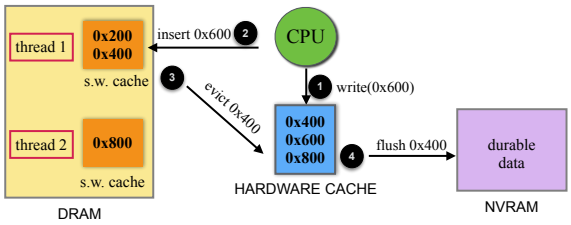

Fig. 1: Illustration of the software cache. The software cache has two cache lines and is full. Thread 1 writes a new cache line 0x600. 0x400 is evicted from the software cache and flushed out of the hardware cache.

The software cache is a per-thread in-memory local store. It is in control of determining when to flush a cache line to persistent memory. Figure 1 shows its basic execution model. When writing a value to persistent memory, CPU, instead of immediately flushing the corresponding cache line, forwards the cache line address to the software cache to buffer the write. In a parallel program, per-thread caching provides isolation and good scalability. The isolation is important. Each thread independently manipulates its own cache, without interference from others. Scalability is good because the implementation does not require locking.

Each thread combines cache line flushes if the coming cache line is already in its local store. Otherwise, it would replace a stale entry, when its local store is full, with the new cache line. A thread issues a command to the hardware cache to force data of the stale cache line out to NVRAM. Figure 1 shows that the local store of thread 1 is full, and after inserting the new cache line 0x600, thread 1 instructs the hardware cache to force 0x400 out to the NVRAM storage.

The software cache is placed in the faster DRAM, rather than NVRAM. We use Least-Recently-Used (LRU) replacement policy. Traditionally, hardware cache has been optimized for fast reads. For persistent memory, the software cache only stores modified data.

3 Adaptive Write Caching

It would be beneficial if cache capacity is workload-aware. Overly large cache size incurs a long CPU stall at the end of a FASE, when CPU resources are wasted. Too small cache size would cause too many cache line flushes.

We use the miss ratio curve (MRC). MRC shows cache miss ratios over different cache sizes. We choose the size online adaptively, which has relatively small cache miss ratio based on MRC and is not very large to stall CPU too long, as the software cache size. The number of misses is the number of cache line flushes in the software cache.

In this section, we present a reuse-based locality theory to derive MRC. We consider an execution as a sequence of data accesses (writes). A logical time is

assigned to each data access. A time window is designated by two data accesses and includes all intervening accesses. The length of a window is the number of accesses it contains.

The reuse locality is measured by the number of data reuses in a time window. Counting the number of intra-window reuses is the same as counting number of reuse intervals that fall within the window. We define the following:

Definition 1 Reuse interval and Intra-window reuse *The time interval between a data access and its next access to the same datum is defined as a reuse interval. If a reuse interval is enclosed within a window, we say that the window has an intra-window reuse.*

Different windows may contain different numbers of reuses. We define the timescale reuse *reuse(k)* as the average number of intra-window reuses of all windows of length k. We call the length k the timescale parameter. Given any trace, *reuse(k)* is uniquely defined.

From Reuse to Cache Hit Ratio At any moment t in fully associative LRU cache, the content consists of data referenced by previous k accesses for some k. *reuse(k)* is the average number of reuses in each k consecutive data accesses. It follows that on average, there are *k-reuse(k)* distinct data in these accesses. The next access is a hit if it is a reuse; otherwise, it is a miss. The difference, $reuse'(k)$=*reuse(k+1)* - *reuse(k)*, shows the average portion of times that the next access is a reuse. Hence, the hit ratio of cache of size (k - *reuse(k)*) is the derivative of *reuse(k)* at k, as shown in Eq. 1.

$$hr(c) = reuse'(k) = reuse(k+1) - reuse(k) \tag{1}$$

where $c = k - reuse(k)$. To illustrate, consider an example pattern "*abab...*" that is is infinitely repeating. The following table shows discrete values of *reuse(k)* and hit ratio, where c denotes cache size, i.e., *k-reuse(k)*.

k	$reuse(k)$	c	hr
1	0	1	0
2	0	2	1
3	1	2	1
4	2	2	1

Reuse vs. Footprint Locality Footprint $fp(k)$ is the average number of distinct data accesses in all windows of length k [4]. Hence, it is obvious that $fp(k)$ plus $reuse(k)$ is k. Xiang et al. showed that the miss ratio is the derivative of footprint [4]. Inspired by their work, we can prove that the derivative of reuse is the hit ratio theoretically. The result is mathematically derivable from footprint, so it is not new. However, the formulation is new and has not been considered in past work. The new derivation gives a new linear-time algorithm to calculate cache performance, which we refer to [3]. In addition, it is the first mathematical connection between the theory of locality [4] (data caching) and the theory of liveness [2, 3] (memory allocation).

4 Preliminary Results

We implemented the software cache in `Atlas` [1], and used an emulator to use DRAM to simulate NVRAM. The emulator system is a machine shipped with 60 Intel Xeon E7-4890 cores at 2.8GHz, running Linux kernel 3.10. We tested SPLASH2 benchmark suite for single-threaded runs. We chose the best cache size online once we have MRC. We compared our approach, **SC**, with three alternatives:

- **AT**: the table approach used in the state-of-the-art `Atlas` [1].
- **ER**: the eager approach, which flushes cache lines instantly every time a persistent store happens.
- **LA**: the lazy approach, which flushes all cache lines at the end of a FASE.

Benchmarks	ER	LA	AT	SC	AT/SC	SC/LA
barnes	1.00000	0.00295	0.08206	0.00391	20.987×	1.325×
fmm	1.00000	0.00246	0.01683	0.00328	5.131×	1.333×
ocean	1.00000	0.09203	0.40290	0.16467	2.447×	1.789×
raytrace	1.00000	0.07140	0.13952	0.07918	1.762×	1.108×
volrend	1.00000	0.00219	0.03189	0.00219	14.561×	1×
water-nsquared	1.00000	0.00107	0.05334	0.00411	12.978×	3.748×
water-spatial	1.00000	0.00103	0.07122	0.00157	45.363×	1.524×
average	1.00000	0.02473	0.11396	0.03698	14.747×	1.893×

Table 1: The data flush ratios of different techniques.

Table 1 shows the write-back ratios of the four techniques. SC outperforms AT by 15× significantly, as a result of selection of the best cache size. As profiled, these sizes are all different and hence workload-aware. Moreover, SC achieves the best for *volrend*. We also measured performance in execution time for ER, AT, and SC. Over ER, the speedup of SC ranges from 1.4× to 34.2×, with an average of 9.6×. The average speedup over AT is 2.1×. As tested, the online overhead of MRC computation is negligible. LA reaches the lowest possible, 16%, since it maximally combines data flushes. However, since all cache lines are written back at the end of a FASE, CPU resources are wasted and hence performance is extremely bad. For example, for *volrend*, LA is slower than AT by 17.8× in running time.

References

1. Dhruva R. Chakrabarti, Hans-J. Boehm, and Kumud Bhandari. Atlas: Leveraging locks for non-volatile memory consistency. In *Proceedings of OOPSLA*, 2014.
2. Pengcheng Li, Chen Ding, and Hao Luo. Modeling heap data growth using average liveness. In *Proceedings of ISMM*, 2014.
3. Pengcheng Li, Hao Luo, and Chen Ding. Rethinking a heap hierarchy as a cache hierarchy: A higher-order theory of memory demand (hotm). In *Proceedings of ISMM*, 2016.
4. Xiaoya Xiang, Chen Ding, Hao Luo, and Bin Bao. HOTL: a higher order theory of locality. In *Proceedings of ASPLOS*, 2013.