Non-Volatile Memory Lecture Notes

Lecture Dates: 3/10/2015, 3/12/2015

Types of NVRAM:
- Flash Memory
- Phase Change Memory
- STTR-RAM
- MRAM
- NV-DIMM (DRAM with backup power) *Sidenote: a company called Pure Storage was using NV-DIMM a couple of years ago.

Common Features/Characteristics of NVRAM:
- Often asymmetric read and write performance, with reads faster
- Erase count is limited – not as severe as flash memory
- Power savings is important
- Word-addressable

Contrast with DRAM (main memory in typical systems today):
- volatile
- lose data when the power is off

SRAM – cache memory, faster and smaller than DRAM, often on-chip with CPU

NVRAM interacts with:
- Main memory (memory)
- PCI-E Bus (memory storage)
- Storage Connection (SAS), usually can be shared by many computing nodes in a different location
- Storage Systems – can be a part of storage systems

NVRAM for Main memory:
- HPC Application – Checkpointing
- Database Application – Online transactions – either carry out of don’t, can roll back if something goes wrong between updates.
In today’s typical systems, DRAM is used as a memory buffer and for buffer management. If input/output is large, output eventually gets written into storage. Typical systems use paging, where some pages are brought into memory, some are flushed out to storage, and many are already in storage. Every 30 seconds, dirty pages (that have been newly written to DRAM or have been updated but not flushed to storage) are flushed out to storage. DRAM is also used as a read buffer for reads coming out of storage.

We need a mechanism to tell how long to keep dirty pages (new or modified pages) in DRAM before flashing them out to storage. We use two mechanisms to do this:

1. pdflash thread
   Every 5 seconds, the thread wakes up to check (scan the pages to see) whether each dirty page has been in memory more than 30 seconds or not. Dirty pages that have been in memory 30 seconds or more are flushed to storage.

2. A macro, can call it “dirty-ratio”
   Idea: don’t allow too many dirty pages in cache, depending on workload (say it is more write-intensive).
   - If dirty page ratio is greater than say 10%, force a flash out to storage. (This is called the high watermark.)
   - A second, low watermark is used to determine when to stop flashing out to storage. If it is 5% and the high watermark is 10%, then when the high watermark is reached, data is flushed out to storage until the dirty page ratio is only 5%.

**Cache Policy and LRU 2Q (Least Recently Used Two Queues)**
Cache policies help to decide which data is cached in memory when it is written into or read out of storage. Also, these policies help to decide which data to evict from memory when the cache becomes full.
Background:
If there is a cache miss, insert the data at the MRU (most recently used) position in the LRU cache. If there is a cache hit, migrate the data from its current position to the MRU position, and push back toward the tail. Evict from the LRU position when the LRU cache is full. This is implemented using a linked list. A problem is that this is expensive on cache hits because it requires deleting a node and inserting to the head of the structure. To minimize this expense, we use a Clock queue, where each position (node) around the clock represents a position in the queue. Each node has a bit that tells whether a page has been frequently referenced or not. 0 means it has been referenced lately, and 1 means it has not. A pointer (the hand of the clock) moves around the clock. If it points to 1, it can evict this page from the queue. If it points to 0, it can change the bit to 1 when moving the pointer to the next position. If there is a cache hit, simply change this 1 bit to a 0.

LRU 2Q:
The LRU 2Q policy is used by the current Unix OS and uses a FIFO (first in, first out) Queue and a Clock Queue. The FIFO queue is an inactive list. When a page is first referenced, it is put in the FIFO queue. If it is never referenced again, it will get pushed out as new pages come in. If the page is referenced again, it is moved to the Clock queue (which is an active list that holds pages referenced at least twice). If the cache is full, the Clock queue is checked first for a victim. Also, the FIFO list occupies about 1/3 of the cache buffer while the Clock queue occupies the remaining 2/3.

All caching algorithms somewhat rely on history, because we do not know future accesses. The most popular algorithm is LRU (as discussed above), which is based on FIFO and solely relies on recency of accesses. The other idea, from the H-ARC paper, is to use frequency of accesses.

Sometimes caching algorithms just cache 10% of the most frequently accessed data, or they can use window size (only look so far back in history, because patterns can change over time). However, this may be expensive to implement. When designing caching algorithms, it is best to first focus on what gives the best results. Then one may design a way to approximate this so that it is not as expensive to implement. For example, the Clock queue is less expensive than LRU, but provides a useful approximation to it.

In the current system using DRAM as main memory, caching algorithms are designed for read hit ratio (at the page granularity).
- R/R (read after read): This is a read hit.
- R/W (write after read): This is not a cache hit because the data is being modified.
- W/R (read after write): This is a hit since the write flashes out before it is read.
- W/W (write after write): This has no effect (is not a hit), since the data is being modified by the second write.

Using NVRAM in place of DRAM
Question: Given the space of NVRAM, how do we partition it into clean and dirty pages? This answer will depend on the application. Is the application read-intensive or write-intensive? What is the ratio of writes after writes? NVRAM

First, consider what counts as a cache hit in NVRAM:
- R/R (read on read): read hit
- R/W (write on read): still not a hit, because we still have to write the same number of times to storage
- W/R (read on write): read hit
- W/W (write on write): This is now a “write hit,” even though it still has to write to storage eventually. With NVRAM, however, we do not have to flush the first write to storage right away. Thus, modifying a write saves one write to storage. We can interpret this as a write hit.

If an SSD is used for storage, using NVRAM for memory makes even more sense because saving writes is important. Also, some argue that even if data is in NVRAM and the system crashes, we may not be able to access the data that was in NVRAM. Thus, it may be reasonable to still flush out periodically, but less often (say every five minutes). If we are designing for shared storage, in which there are multiple clients, a synchronization issue emerges. More updates, such as in a transactional database, become a large problem. If our assumptions about the system change (i.e. to include shared storage for multiple clients with the shared storage using NVRAM and clients using DRAM), then our actions may change. If a copy of data is sent to a client, the client will most likely cache this. Then do we really need to cache it in NVRAM? The I/O trace coming into each client’s DRAM will be different than the I/O trace coming out from the system’s NVRAM and going to a client’s DRAM. Also, our clients may have mixed NVRAM, DRAM, or NVRAM+DRAM.

ARC (Adaptive Replacement Cache), '02, used by IBM and ZFS
ARC is an NVRAM-based cache policy that holds pages in cache if they have been referenced recently or frequently. This algorithm was developed because most NVRAM-based cache policies focus on management of clean and dirty pages, but there were none that used the recency and frequency of pages in a clever way so as not to be computationally expensive.

Two factors are considered:
- Recency: covered by a LRU queue
- Frequency: Hard to implement unless count frequency for each page, but this is very expensive with a moving window counting frequencies. Instead, it is implemented as another LRU queue. Pages in the recency queue that are referenced are moved to the frequency queue.

Additionally, there are two ghost caches (one for recency and one for frequency) whose combined size is equal to the size of the real recency and frequency caches. These ghost caches hold only the metadata of pages for those pages that have been most recently evicted. Consider Figure 2, which shows the partitioning of the ARC algorithm. There is always a current pointer which shows where the partition boundary is in NVRAM, and an ideal pointer that the algorithm uses to determine which partitioning would be better. For example, if there is a hit in n4 (the frequency ghost cache), then the pointer will move left
because data is migrated into n2, the frequency LRU. If the pointer needs to be moved for a more ideal partitioning and the cache is full, then some page(s) must be evicted. Evicted pages always migrate from their position in the real cache to the MRU position in their respective ghost cache. The ghost caches push pages back until the LRU position is filled. If a ghost cache is full, adding another page simply pushes a page out of the LRU position.

Figure 2: ARC Partition Layout

H-ARC (Hierarchical Adaptive Replacement Cache)
H-ARC uses the idea of ARC to partition NVRAM based on both clean versus dirty pages and recency versus frequency. Thus, the ARC process, as in Figure 2, is applied twice. As we see in Figure 3, H-ARC first partitions the data into a clean cache and a dirty cache, with corresponding ghost caches. Then the process is applied to each side of the partition (dirty ARC and clean ARC), where each is divided into a recency cache and frequency cache (with corresponding ghost caches). Thus, the recency and frequency ghost caches combine to form the dirty and clean ghost caches. H-ARC assumes that we do not flash out to storage whenever there is a write. Thus, write-after-write counts as a cache hit.
Combining DRAM and NVRAM (things to consider)

We can argue that DRAM is best for holding clean pages, while NVRAM is best for holding dirty pages (since these pages can be updated many times and not flashed to storage, saving writes to storage).

If we use DRAM and NVRAM, say DRAM is of size $S_1$ and NVRAM is of size $S_2$, several issues may be considered:

- We could have that $S_2 < S_1$, because NVRAM is more expensive.
- NVRAM is slower than DRAM (2-3x slower for reads, 5-7x slower for writes), which can be a performance issue.
- Erase count issue: NVRAM has a limited erase count. In the extreme case, everything is modified on DRAM side.
- There is also a power saving issue. How best can one manage pages over both NVRAM and DRAM to make most efficient use of power.

While each of these last three issues (performance, erase count, and power) has been studied separately, nobody is studying these issues in tandem. This is most likely necessary since solving one of these problems optimally affects the other problems negatively.

If we assume that DRAM and NVRAM are the same speed for reads and writes, then we can consider sending most writes to NVRAM and most reads to DRAM to account for the reliability issue of DRAM. Reads-after-writes (W/R) could go to either. But the question is how to evaluate whether a page will be written again or should move to DRAM because
there are more reads after a page is written. Some sort of knowledge about the workload must be used, such as keeping track of recent history through a moving window. Also, nobody has taken into account the locality of pages. Additionally, if there are a lot of clean pages, some of them must have to occupy NVRAM in addition to DRAM. This may affect performance significantly.