Logging in Persistent Memory: to Cache, or Not to Cache?

Mengjie Li, Matheus Ogleari, Jishen Zhao*,†
*University of California, Santa Cruz, †University of California, San Diego

1 Overview

Persistent memory is a new class of memory that functions as a hybrid of traditional storage systems and main memory. It combines the benefits of both the data persistence property of storage and the fast load/store interface of memory. In order to maintain data persistence in memory, a widely used mechanism is logging – in addition to updating the original data structures as in traditional memory systems, persistent memory systems also log the update. Most previous persistent memory studies suggest that log updates should bypass the cache hierarchy because the log is only used for system recovery and will not be reused during program execution. Caching the log only contaminates critical cache resources, leading to performance degradation. However, our study shows that current cache bypassing schemes (e.g., cache bypassing instructions and write-combining buffers) are sub-optimal for accommodating log writes. Making the log uncacheable can degrade persistent memory system performance worse than with cacheable logging. This presentation outlines our observations of the trade-offs between cacheable and uncacheable logging, based on our experimental results. We also analyze the reasons that lead to such trade-offs.

2 Persistent Memory

Traditional systems maintain persistent data in slow storage devices (e.g., disks and flash) via block-level I/Os. Byte-addressable nonvolatile RAM (NVRAM) technologies enable new persistent memory techniques that combine fast load/store interface of memory with data recoverability of storage. As demonstrated in prior studies, doing so can improve the performance and energy efficiency of storage workloads, such as file systems, databases, and key-value stores, by an order of magnitude. Yet conventional memory system management schemes do not support data persistence. Persistence cannot be supported by simply replacing volatile DRAM with an NVRAM device. For example, a power outage can occur while an application is inserting a node to a linked list. In this case, the processor caches and memory controllers can reorder the write requests, writing the pointer into NVRAM before writing the values of the new node. The linked list would thus lose consistency due to dangling pointers if values of the new node remaining in processor caches are lost due to the power outage. This would lead to unrecoverable data corruption. To avoid such inconsistency problems, most persistent memory designs maintain persistence in the memory by borrowing data persistence mechanisms used in traditional databases and file systems [4, 5].

3 Should We Cache Log Updates?

Logging in persistent memory. One of widely used persistence mechanisms is logging. With logging, persistent memory systems can maintain two data versions: one in the original data structure, the other in the log. Applications update both versions but not at the same time. As a result, system failures can corrupt one of the versions, but never both. Using our linked list example: persistent memory systems can first write the values of the new node into a log and then update the original linked list (nodes and pointers) after the log updates arrive at NVRAM. Therefore, if the system loses power before logging is complete, the original linked list is intact. Otherwise, if system failures corrupt the linked list, the log updates already committed can be used to recover the original data structure.

Problems with Caching the Log. Most previous studies suggest that log updates should not be cached in the processor caches. The reason is straightforward: caches are meant to utilize memory access locality and exploit data reuse. Yet the log is only reused during system recovery. Therefore, it has low temporal locality and is thus less likely to be re-accessed during program execution. Furthermore, because processor caches have limited capacity, caching the log can pollute the cache, forcing the cache hierarchy to evict useful working data with high locality. Therefore, it appears beneficial to make the log uncacheable to avoid cache pollution.

Problems with Uncacheable Log. Unfortunately, existing cache bypassing schemes are sub-optimal with persistent memory systems. Commodity processors offer special instructions and hardware components to accommodate uncacheable writes. For example, x86 processors offer uncacheable write instructions, such as `movnti` and `movntq`, that can invoked through inline functions (`__asm__()`) or intrinsic functions (e.g., `_mm_stream_si64`) [2]. These instructions require more cycles to complete than cacheable write instructions (e.g., `movl` and `movq`) due to (i) additional data movement across registers and (ii) longer latency of writing to memory than to caches. Furthermore, uncacheable writes are typically buffered in the processor to exploit write coalescing. For example, Intel's x86 processors provide write-combining buffers, which are four to six cache line-sized entries per core, to coalesce uncacheable stores. The uncacheable data are stored in this write-combining buffer in a FIFO manner. If any buffer update propagates to memory when the entry is not full or does not contain new data, the uncacheable store become a partial write. Partial writes are inefficient because it under-utilizes both write-combining buffer and memory bus resources. Therefore, log updates at a granularity smaller than a cache line generates partial writes to the write-combining buffers, leading to this under-utilization of resources.

4 Experimental Studies

We compare the performance of cacheable and uncacheable logging in persistent memory systems with a microbenchmark shown in Figure 1. The benchmark uses a loop to assign random strings to the elements in an array of strings. We ensure persistence of the array updates in each iteration in the loop by logging the index and values of the array element to be updated (i.e., we employ redo logging). Figure 1 shows two versions of the benchmark: one using intrinsic functions to invoke `movnti` to perform uncacheable log updates; the other caches log updates.

```
//Uncacheable log
for (i = 0; i < array_size; ++i) {
    value = random_string;
    key   = i;

    // Log updates
    // Intrinsic functions to invoke movnti
    _mm_stream_si32(&log[2 * i], key);
    _mm_stream_si32(&log[2 * i + 1], value);
    asm volatile ("sfence");

    array[i] = value;
}
```

```
//Cacheable log
for (i = 0; i < array_size; ++i) {
    value = random_string;
    key   = i;

    // Log updates
    log[2 * i] = key;
    log[2 * i + 1] = value;
    asm volatile ("sfence");

    array[i] = value;
}
```

Figure 1. Pseudocode examples with uncacheable log and cacheable log, respectively.

We use intrinsic functions to invoke `movnti` because compared to inline assembly functions, the compiler has an innate knowledge of the intrinsic function and can therefore better integrate it and optimize it for the situation.

We run our benchmark on a Dell OptiPlex 7040 Tower Desktop computer, with 4-core 3.4GHz Intel Core-i7 CPU and an 8MB L3 cache. We construct the array to be 8MB in size so the working data can fit in the last-level cache (LLC) without logging. The log stores both array index and data in the updates, making it larger than the LLC. We run each version of our benchmark for 20 times and only collect statistics during loop iterations. We observed that the performance deviation is within 1%. Therefore, we only report the average performance numbers. We use Linux *perf* [1], which is a kernel tool that can instrument CPU hardware counters, to profile processor performance statistics. We use `rdtsc` [3], the time stamp counter available on x86 processors to obtain processor timing statistics. Our experimental results do not consider NVRAM's longer latency than DRAM. However, with longer latency, the aforementioned resource under-utilization and write latency issues of uncacheable log updates can be even worse.

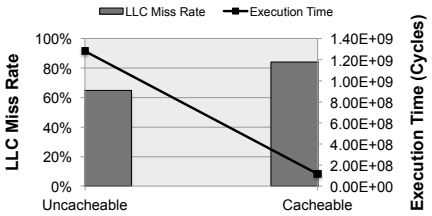

Figure 2. LLC miss rate (bars) and execution time (line).

4.1 Cache Pollution with Cacheable Log

Figure 2 compares the LLC miss rate and execution time between the two versions of our benchmark. Apparently, uncacheable log substantially reduces LLC miss rate compared with caching the log in the processor. This is expected because caching the log stresses the cache capacity by contention with the original data array. As a result, original data and log updates yields substantial cache thrashing. We observe that the log updates frequently break the spatial locality of working data updates in caches.

4.2 Inefficiencies of Uncacheable Log

Surprisingly, the cacheable log version of benchmark does not appear to degrade system performance. Instead, it yields much lower execution time compared with the uncacheable log version of benchmark as shown in Figure 2. We further analyze the reasons behind this performance observation.

Additional register access introduced by movnti. When we further investigate the assembly-level execution of our benchmarks, we notice that the intrinsic function that invokes `movnti` takes more cycles to execute than a cacheable store. One reason is that `movnti` instruction requires a general-purpose register as one of its arguments and therefore generates extra data movements in the register file. The additional latency caused by this additional register access is not negligible in a program contains a large number of uncacheable writes.

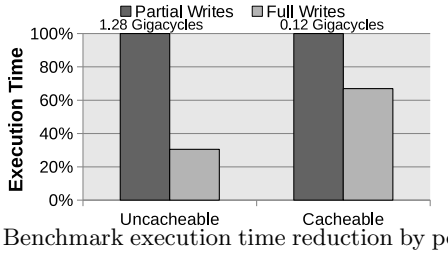

Figure 3. Benchmark execution time reduction by performing full writes in write-combining buffers.

Partial writes in write-combining buffers. Partial writes in write-combining buffers is another reason for performance degradation with the uncacheable log. Figure 3 shows our evaluation of the partial vs. full writes' effect on write-combining buffers. The *y-axis* is benchmark execution time (cycles) normalized to the benchmark iteration configuration that generates partial writes to write-combining buffers. Because partial writes will write less data than full writes in each iteration, processing the same amount of working data requires more iterations. Each iteration needs to execute an `sfence` instruction (Figure 1), which can block subsequent stores. Frequently invoking `sfence` can introduce long latency during program execution. Reducing the number of iterations can effectively decrease the number of sfence instructions. As a result, with partial writes in write-combining buffers, our benchmark with cacheable log has 11.13× speedup over the version with uncacheable log. Yet with full writes in write-combining buffers, the speedup is reduced to 5.06×.

5 Conclusion

Most prior log-based persistent memory designs make log updates uncacheable. This is reasonable given that log is used during system recovery and can contaminate CPU caches. However, our study and analysis show that caching log updates leads to better persistent memory system performance. This is because existing cache bypassing schemes are sub-optimal in accommodating persistent memory access.

References

[1] Perf wiki. http://perf.wiki.kernel.org/.

[2] Intel. Intel architecture instruction set extensions programming reference, 2016.

[3] G. Paoloni. Intel, how to benchmark code execution times on Intel IA-32 and IA-64 instruction set architectures. In *Intel White Paper*, 2010.

[4] H. Volos, A. J. Tack, and M. M. Swift. Mnemosyne: Lightweight persistent memory. In *Proceedings of the Sixteenth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS XVI, pages 91–104, New York, NY, USA, 2011. ACM.

[5] J. Xu and S. Swanson. NOVA: A log-structured file system for hybrid volatile/non-volatile main memories. In *14th USENIX Conference on File and Storage Technologies (FAST 16)*, pages 323–338, 2016.