Implications of CPU Caching on Byte-addressable Non-Volatile Memory Programming

Kumud Bhandari, Dhruva R. Chakrabarti, Hans-J. Boehm

HP Laboratories
HPL-2012-236

Abstract:

Byte-addressable non-volatile memory may usher in a new era of computing where in-memory data structures are persistent and can be reused directly across machine restarts. In this context, we study the implications of different CPU caching modes and show how they affect both programmability and performance of a program.

External Posting Date: December 6, 2012 [Fulltext] Approved for External Publication
Internal Posting Date: December 6, 2012 [Fulltext]

Implications of CPU Caching on Byte-addressable Non-Volatile Memory Programming

Kumud Bhandari *
Rice University and HP Labs, USA
kumud.bhandari@hp.com

Dhruva R. Chakrabarti, Hans-J. Boehm
HP Labs, USA
{dhruva.chakrabarti, hans.boehm}@hp.com

Abstract

Byte-addressable non-volatile memory may usher in a new era of computing where in-memory data structures are persistent and can be reused directly across machine restarts. However, sudden failures complicate matters because a program state may partially reside in volatile buffers and caches as opposed to primary non-volatile memory. We study the implications of different CPU caching modes and show that a particular choice affects both programmability and performance of a program.

1. Introduction and Problem Definition

New non-volatile memory (NVRAM) technologies such as memristors and phase change memory (PCM) offer a byte-addressable interface and memory access latencies that are comparable to those of DRAM. These NVRAM devices are expected to be connected as memory and accessed using regular CPU loads and stores. The result is fast persistence of program objects and potentially these in-memory data structures that are already maintained by the application can be reused directly.

But such in-memory persistent data are useful only if they are consistent; in particular their invariants must be preserved. Even in the presence of NVRAM, there will be volatile buffers and caches in the memory hierarchy, simply because of the performance advantages they provide. This implies that during program execution, some of the state may reside in volatile structures and the rest in NVRAM. Figure 1 shows a simplified version of the architectural model that we assume; it is in part based on the Intel x86-64 architecture [1]. The CPU core may temporarily keep each store to memory in a store buffer. The store buffer improves performance by hiding the latency of cache and memory accesses. Certain instructions such as memory fences result in draining the store buffer. In the write combining (WC) caching mode, writes are retained in the WC buffer temporarily, improving performance. The WC buffer does not participate in the cache coherence protocol, and thus the WC mode provides very weak memory ordering guarantees. As shown in figure 1, we assume that DRAM and NVRAM may co-exist in the same memory system.

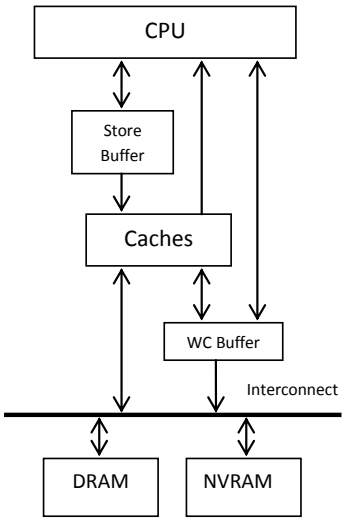

Figure 1. Simplified Architectural Model

As figure 1 shows, even in the presence of NVRAM, volatility remains an important part of the memory hierarchy. If the program crashes because of a hardware or power failure, only the state present in NVRAM at the time of the crash will survive a restart. Any state that was present in volatile structures at the time of the crash is lost. Hence, the challenge is to ensure that at any point of program execution, there is enough

* Work done during a summer internship at HP Labs.

information on NVRAM to reconstruct a consistent state of the program data structures.

It turns out that constraining the order in which writes become visible on NVRAM is at the core of maintaining consistency. Consider, for example, a common programming idiom where a persistent memory location N is allocated, initialized, and published by assigning the allocated address to a global persistent pointer p. If the assignment to the global pointer becomes visible in NVRAM before the initialization (presumably because the latter is cached and has not made its way to NVRAM)[1] and the program crashes at that very point, a post-restart dereference of the persistent pointer will read uninitialized data. Assuming write-back (WB) caching mode, this can be avoided by inserting cache-line flushes for the freshly allocated persistent locations N before the assignment to the global persistent pointer p. This ensures that the initialization and publication are visible in this order on NVRAM. Higher-level guarantees such as transactional semantics can be built on top of this low-level visibility constraint.

However, reasoning in terms of low-level interfaces, such as cache-line flushes, is error-prone[2]. Additionally, insertion of such cache flush instructions at appropriate program points requires recompilation and precludes the use of existing code. In this paper, we explore the viability of write-through (WT) caching mode for persistent data. Using WT mode on Intel x86-64 [1], all CPU stores result in writes to all levels of caches and through to system memory. Reads continue getting the benefit of caching. Though an individual WT CPU store may be slower than a corresponding WB one, expensive cache flushes are no longer required, and unmodified legacy code may be reused in some contexts.

Our contributions are 2-fold. We show that the overall performance of arbitrary applications is comparable between WB and WT, but optimizations enabled by particular programming idioms may allow WT to achieve significant speedups over WB. Since programmability benefits with WT are a given in any application, using WT selectively for persistent data structures appears to be a viable alternative.

[1] Note that hardware or compiler reordering may have a similar effect.

[2] This is akin to memory fence based multithreaded programming, not a very successful programming paradigm.

2. Description of our Approach

We assume a programming environment where failure-atomic sections of code are used to transition data structures from one consistent state to another. In a multithreaded program, such a section of code provides transactional guarantees of failure-atomicity, consistency, isolation, and durability. Failure-atomicity implies an all-or-nothing behavior for visibility of updates to NVRAM. Publication safety [2] is honored so that if a failure-atomic section completes, the effects of all operations on persistent data executed within or before that code section must have reached NVRAM.

```
1: node = nvm_alloc(sizeof(node_t));
2: node->value = val;
3: node->next = head;
4: atomically {head = node;}
```

Figure 2. Typical allocation, initialization, publication sequence

While a programmer reasons in terms of transactional semantics, the underlying implementation may use a write-ahead log to track accesses to persistent locations to support failure-atomicity [3, 4]. Once the transactional region commits, the logs have to be flushed to NVRAM before the persistent user locations are written and flushed out. Once all of this is successfully performed, the log entries can be discarded.

Consider figure 2 where a node of a list is allocated, initialized, and inserted at the head. As shown, most of the code sequence can be non-transactional. This allows reuse of existing library code, such as for creating a complex data structure, and inserting that data structure into client objects atomically. This is similar to our previous publication example.

Since failure-atomicity is not implied for non-transactional code, log entries are not required outside transactions. But to support publication safety, the implementation must make sure that all updates made before a transaction reach NVRAM at or before the commit point of that transaction. This ensures that restart code that sees a new head value will also see the corresponding node fields.

If WB caching mode is used, and conventional single-line cache flush instructions are used, the non-transactional persistent updates must still be tracked (or directly flushed) so that the relevant addresses can be flushed out at or before the commit point of the succeed-

ing transaction. This incurs costs for non-transactional code, and prevents reuse of existing library code to access persistent data, even outside transactions.

Now consider using WT caching mode for figure 2. Since any write will be written all the way down to NVRAM, the non-transactional code does not have to be tracked at all. The transactional code will still require logging in order to support failure-atomicity. Visibility constraints can be enforced without added code, and hence legacy code can be reused without modification outside of transactions.

3. Implementation and Experimental Results

3.1 Methodology

We explored performance tradeoffs of different caching modes available on Intel x86-64 [1]. In addition to WB and WT, we experimented with two additional caching modes using Linux 2.6.32: uncacheable (UC, bypasses cache for read and writes), and write combining (WC, same as UC except writes are combined using a write-combining buffer for efficiency). We implemented system calls such that user level applications could request changes to the caching mode of a memory region mapped to persistent storage (defined by a starting and an ending virtual address) at page-level granularity. Although WT is available as an option in Intel x86-64 architecture, recent versions of Linux do not support this mode by default. Hence, we modified the Linux kernel initialization code to enable WT support. We give some details in the Appendix.

We simulated NVRAM with DRAM using Linux RAM disk and successfully performed crash/recovery testing.

3.2 Benchmarks

In order to study the performance characteristics of different caching modes in Intel x86-64, we experimented with different flavors of a synthetic benchmark and 3 persistent data structures. The synthetic benchmark linearly traverses an array within a loop, spanning several memory pages, performing: R (read), or W (update to each element), or RW (read and update to each element). The array takes about 800KB. Transactions are not used in the synthetic benchmark. The persistent data structures used are a multithreaded queue, a failure-atomic version of Christopher Clark's hashtable [6] using transactions, and a multithreaded copy-on-write array-based list[3] (henceforth called cow_al). In queue, two threads insert and remove 100,000 elements making it a write-intensive benchmark. In hashtable, the program inserts and removes 4000 elements with traversal in between, making it read-write balanced. cow_al maintains an internal array that, once created, is never modified. For every mutating operation, a fresh copy of the array is created, modified, and a pointer atomically switched. To allow multiple writers, a version counter is maintained — if it has changed since the mutation started, a fresh copy of the internal array is made with the updated version and the process repeats until the atomic update is successful (with the original version). A read operation, such as a query, obtains a handle to the internal array and proceeds, without any cross-thread interference or synchronization. Our cow_al driver maintains 2 threads, each of which repeatedly performs a mutation followed by a traversal of the list such that each thread makes a total of around 1,000,000 writes to persistent locations. A distinguishing characteristic of this benchmark is that most of the code in this benchmark is non-transactional, the only transactional one being for the atomic pointer switch. All reported results are averages over 4 runs and are obtained on a quad-core Intel(R) Xeon(R) E5620 with 12M cache and running at 2.4GHz.

[3] This benchmark is inspired by Java's CopyOnWriteArrayList.

3.3 Results

The first three sets of columns in figure 3 show the runtimes for the synthetic loop benchmark in each of the described flavors – note that no visibility constraints were added here. It is evident that WT has read characteristics (R) of WB and write characteristics (W) of UC. The latter indicates that the write-combining buffer is not used and hence an mfence is not required after a WT store to enforce ordering[4]. Likewise, WC has read characteristics (R) of UC and write characteristics (W) of WB. As expected, stores in WT mode are expensive. But given that stores to persistent data in WB mode require expensive cache line flushes, the question is whether WT mode is attractive when ensuring correct visibility semantics on NVRAM. In our experiments, we observed that a cache line flush takes around 300 cycles to complete. Interestingly, our results appear to indicate that

[4] Volume 3A, Section 11.3 of Intel64 Architecture Software Developer's Manual [1] mentions that write combining is allowed in WT. However, our experimental results indicate that the Intel x86-64 machine we have does not use it.

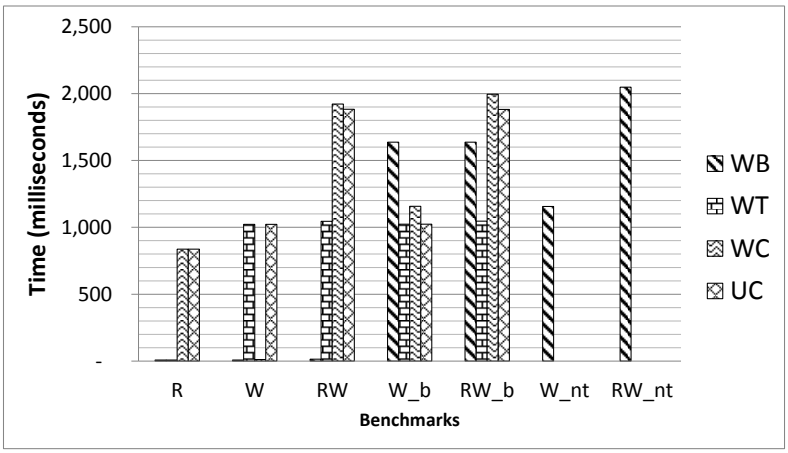

Figure 3. Runtimes for synthetic benchmark flavors

a cache line flush is equally expensive for clean, dirty, and invalid lines in `Intel x86-64`.

Next, assuming that the array resides in persistent memory, we added sufficient instructions to ensure that stores become visible in order. The `R` benchmark stays unchanged as it does not contain stores. The modified `W` and `RW` benchmarks are denoted by adding `_b` to their corresponding names, i.e. `W_b` and `RW_b`. In `WB` mode, the visibility constraint is imposed by a `FLUSH` [7] that, on `Intel x86-64`, comprises a memory fence (`mfence`), followed by a cache line flush of the relevant address (`clflush`), followed by another `mfence`. The first `mfence` ensures that the store buffer is emptied before issuing the `clflush`. The memory fences also ensure the correct instruction execution order. Under `UC` and `WT` modes, we insert a directive `asm volatile ("" ::: "memory")` following a store to prevent compiler instruction re-ordering. Since the `Intel x86-64` architecture follows a `TSO` memory model, a separate memory fence is not required[5]. In case of `WC` mode, `W_b` and `RW_b` require a hardware memory fence (`mfence`) after stores so that the write-combining buffer is flushed. Lastly, in `WB` mode, we experimented with `movntq` (move with a non-temporal hint) followed by `mfence` to achieve visibility and ordering guarantees of memory stores. We denote these benchmarks by adding `_nt` to their original names (i.e. `W_nt` and `RW_nt`). The aim is to understand any differences in the performance characteristics between using `movntq` and `WC` mode.

In figure 3, the last four sets of columns show runtimes for `W_b`, `RW_b`, `W_nt`, and `RW_nt`. Results for both `W_b` and `RW_b` show that employing a `WT` mode is around 50% faster than `WB` mode. Furthermore, `W_b` under `WT` mode runs around 12% faster than `W_nt`. The memory pages in `RW_nt` are mapped in `WB` mode and hence this benchmark suffers from cache line eviction each time a write to the line is performed (as outlined[6] in Volume 1, Section 10.4.6.2 of [1]), which slows down reads. As a result, `RW_b` under `WT` mode runs 2x faster than both `RW_b` under `WC` mode and `RW_nt`. We note here that results not shown in this paper indicate that the performance characteristics of the synthetic benchmark remain unchanged regardless of whether the array fits in `L1` cache.

Figure 4 shows the performance comparison for the persistent data structures. `queue` performs the best in `WB` mode – it is 43%, 80%, and 69% slower in `WT`, `WC`, and `UC` modes respectively. `queue` is write-intensive with some interspersed reads and most of the updates in this benchmark occur within transactions. Note that transactional implementations require frequent cache line flushes in `WB` and frequent memory fences in `WB` and `WC` modes.

[5] We believe that this is semantics-preserving but the `Intel x86-64` memory model may need further clarification for non-`WB` caching modes.

[6] This eviction cost may explain the slight difference between the runtimes of `RW_b` in `WC` mode and `RW_nt`.

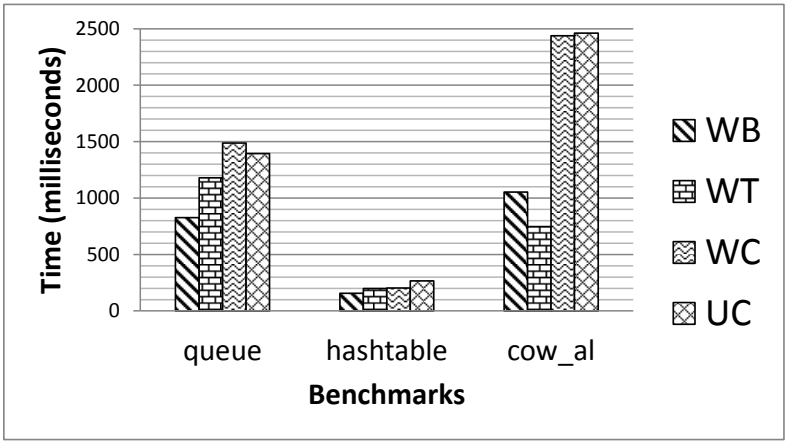

Figure 4. Runtimes for persistent data structures

`hashtable` also performs the best in `WB` mode – it is 26%, 30%, and 70% slower in `WT`, `WC`, and `UC` modes respectively. Note that `hashtable` has a sizable number of both reads and writes, so a mode that offers good performance for both access patterns will provide good performance. Additionally, all updates to persistent data structures occur within transactions in `hashtable`.

On the other hand, `cow_al` performs the best in `WT` mode – it is 41%, 226%, and 229% slower in `WB`, `WC`, and `UC` modes respectively. `cow_al` has a balanced number of reads and writes but the distinguishing characteristic is that almost all of them are non-transactional. It appears that in a workload with such a pattern (as exemplified by a copy-on-write style implementation), `WT` mode may perform the best because most of the memory accesses are unconstrained and the long latency of persistent writes in this mode can be hidden by the micro-architecture. `WB` mode suffers because the non-transactional writes still need to be flushed out of the caches. Reads are still important which explains the poor performance of `WC` and `UC`. We believe that the performance characteristics of `cow_al` (shown in figure 4) mostly track that of `RW_b` (shown in figure 3). In general, the average performance of the 3 benchmarks in `WT` mode is competitive, and it provides the additional programmability benefits discussed in Section 2.

4. Related Work

Volos et al. [4] use `movntq` along with `mfence` to make writes to persistent data instantly visible in persistent memory. They use `FLUSH` along with regular `WB` stores if reads are involved. To the best of our knowledge, we are the first to present the implications of using `WT` caching mode for persistent data. Additionally, we present a comprehensive comparative study and analysis of all the caching modes typically found in modern architectures. Coburn et al. [3] and Condit et al. [8] both advocate changes at the hardware level to enforce consistency and visibility of persistent data. In contrast, we focus on techniques that use existing architectural support.

5. Conclusions

While persisting data in `NVRAM`, certain consistency semantics have to be maintained in order for such data to be reusable across machine crashes and restarts. We presented performance tradeoffs of various caching modes on modern architectures that can be used in such a context. There is no universally dominant strategy, but overall `WT` appears competitive. Its ability to accommodate unmodified legacy code outside transactions may give it an edge among caching strategies supported by current processors. Clearly these results may change in the presence of new cache flushing primitives, such as a mechanism for flushing a cache line to memory without invalidating the cached copy.

References

[1] Intel64 and IA-32 Architectures Software Developer's Manuals Combined. http://www.intel.com.

[2] Menon et al., Single global lock semantics in a weakly atomic STM, TRANSACT 2008.

[3] Coburn et al., NV-Heaps: Making persistent objects fast and safe with next-generation, non-volatile memories, ASPLOS 2011.

[4] Volos et al., Mnemosyne: Lightweight persistent memory, ASPLOS 2011.

[5] Bovet et al., Understanding the Linux Kernel, O'Reilly, 2005.

[6] Christopher Clark, A hash table data structure in C, https://github.com/davidar/c-hashtable.

[7] Venkataraman et al., Consistent and durable data structures for non-volatile byte-addressable memory, FAST 2011.

[8] Condit et al., Better I/O through byte-addressable, persistent memory, SOSP 2009.

A. Appendix: Kernel implementation Approach

In `Intel x86-64` with kernel-level privilege, one can specify caching policy at a page-level granularity. Using bits `PAT` (7th bit), `PCD` (4th bit), and `PWT` (3rd bit) in the page table entry, one can point to the suitable entry in the Page Attribute Table (`PAT`) [5]. `PAT` is a 64 bit register where each of eight entries is represented by 8 bits. Table 1 shows the default `PAT` values. Notice that entries 4-7 mirror entries 0-3. For instance, setting `PAT` to 1, `PCD` to 0 and `PWT` to 1 would result in pointing to the 5th entry in `PAT` which is write through (`WT`) by default in `Intel x86-64` following a restart or power-up. Current Linux (2.6.x or higher) does not support `WT` policy. Hence, as a part of initialization, `Linux` changes the 1st and the 5th entry in `PAT` register to Write Combining (`WC`). We change the 5th entry to Write Through (`WT`) and leave other entries unchanged. Next, we add system calls to Linux kernel to change cache policy of page(s) in a given address range to write back, write through, write-combining, and uncacheable.

To modify bits in page table entry, we utilize Linux kernel API. When a system call is made by a user application process, the system call code obtains the pointer to `task_struct` for the current process. Next, it obtains the `mm_struct` inside the `task_struct`, which contains information related to memory management. Linux employs four levels of hierarchy for page table management [5]. The starting address of the top level page table Page Global Directory (`PGD`) is stored in `mm_struct`. Using the `mm_struct` and the given address, the provided macro computes the corresponding `PGD` entry for the given address. Next, using the `PGD` entry and the current address, we compute the corresponding entry in second level table called the Page Upper Directory (`PUD`). Likewise, using the `PUD` entry and the current address, we compute the corresponding entry in third level table called the Page Middle Directory (`PMD`). Finally, we obtain the Page Table Entry (`PTE`) using the `PMD` entry and the current address. We modify the `PTE`, write the new value using the macro `set_pte`, and flush the Translation Lookaside Buffer (TLB) entry for the modified page.

PAT Entry	Cache Policy
PAT0	Write Back
PAT1	Write Through
PAT2	Uncacheable
PAT3	Strictly Uncacheable
PAT4	Write Back
PAT5	Write Through
PAT6	Uncacheable
PAT7	Strictly Uncacheable

Table 1. Page attribute table after Linux boots up in `Intel x86-64` [1]