Reducing the Cost of Persistence for Nonvolatile Heaps in End User Devices

Sudarsun Kannan, Ada Gavrilovska, Karsten Schwan
Georgia Institute of Technology
College of Computing, Atlanta
{sudarsun.kannan, ada, schwan}@cc.gatech.edu

Abstract

This paper explores the performance implications of using future byte addressable non-volatile memory (NVM) like PCM in end client devices. We explore how to obtain dual benefits – increased capacity and faster persistence – with low overhead and cost. Specifically, while increasing memory capacity can be gained by treating NVM as virtual memory, its use of persistent data storage incurs high consistency (frequent cache flushes) and durability (logging for failure) overheads, referred to as 'persistence cost'. These not only affect the applications causing them, but also other applications relying on the same cache and/or memory hierarchy. This paper analyzes and quantifies in detail the performance overheads of persistence, which include (1) the aforementioned cache interference as well as (2) memory allocator overheads, and finally, (3) durability costs due to logging. Novel solutions to overcome such overheads include (1) a page contiguity algorithm that reduces interference-related cache misses, (2) a cache efficient NVM write aware memory allocator that reduces cache line flushes of allocator state by 8X, and (3) hybrid logging that reduces durability overheads substantially. With these solutions, experimental evaluations with different end user applications and SPEC2006 benchmarks show up to 12% reductions in cache misses, thereby reducing the total number of NVM writes.

1 Introduction

Future byte addressable non-volatile memory technologies (NVM) like phase change memory promise the benefits of faster persistent storage than SSDs coupled with larger capacity with less power, compared to DRAM. Prior research has typically either used NVM as an additional virtual memory to increase total memory capacity, or placed it behind the I/O stack for fast access to persistent storage. Those methods are beneficial for high end servers, with NVM placed into select memory sockets along with its additional use as a block cache, but for resource- and cost-constrained end devices like smartphones and tablets, it is preferable to use the device's NVM both for increased memory capacity and for fast access to persistent storage.

This paper addresses the challenge of how to efficiently use NVM's byte addressability, in terms of bypassing software stack overheads, while at the same time, enabling persistence for such memory when and if desired. Referring to the use of NVM for additional heap capacity without persistence as *NVMCap* vs. its use of persistence as *NVMPersist*, we contribute (1) detailed studies of the performance overheads of simultaneously exploiting these two capabilities of NVM, followed by (2) the creation and evaluation of techniques that mitigate these performance overheads. Specifically, concerning (1), NVM's high write latency compared to DRAM (5x-10x) [3] makes it difficult to use it for extended capacity – NVMCap. Obtaining comparably high performance requires the efficient use of system caches by the end client applications being run. Yet these same caches are also in the path of accesses to NVMPersist, where to guarantee consistency, durability, and failure recovery, the application data as well as its metadata must be frequently serialized and flushed from cache. Cache line flushes involve writebacks of dirty data (if any) and cache lines invalidation broadcasts across all cores. Further, since evictions from the cache can be in any order, the updates from cache must be serialized, by fencing memory write operations [26, 10]. An expected outcome of such actions is increased cache misses and higher NVM access latency for NVMPersist-based applications. A perhaps less obvious, yet quite undesirable outcome is that such cache flushes can also substantially impact the NVMCap applications using the same last level cache.

We present experimental evidence documenting these facts. Concerning (2) above – the mitigation of performance overheads – key to attaining high performance when using NVM is to reduce the number of direct NVM writes and to reduce the last level cache misses suffered by NVMpersist and by NVMCap. We find that to deal with NVMPersist's cache usage (i.e., frequent cache flushes) requires a multistep, end-to-end solution that includes (i) OS/hardware techniques that provide efficient cache sharing between NVMPersist and NVMCap applications, coupled with (ii) user level techniques that reduce cache misses due to NVM writes for maintaining the ACID (atomicity, consistency, integrity, and durability) requirements of NVMPersist applications. For (i), we identify and evaluate appropriate cache sharing mechanisms. For (ii), we redesign traditional memory allocation methods and develop a cache efficient data versioning/logging method.

This paper makes the following specific technical contributions toward efficiently using NVM for both NVMCap and NVMPersist in end client devices with focus on reducing the cache misses:

1. Persistence Impact: we analyze end client device workloads to better understand the impact of NVMPersist applications on NVMCap applications sharing the same cache. used to quantitatively capture NVM writes.
2. OS-level Cache Sharing: to reduce cache misses due to sharing, we propose a novel but simple page coloring mechanism that exploits as a metric 'physical page contiguity misses'. The approach is implemented in the Linux kernel memory management layer and reduces cache misses by 4% on a average, validated through hardware performance counters.
3. Library-level Optimizations: the metadata structures needed for persistence cause overheads in terms of increased cache misses. We analyze persistent memory allocators and their durability-related data structures across the system stack, and propose a novel cache-efficient allocator and an efficient hybrid (word-object) logging approach that significantly reduces the number of writes to NVM.

All solutions are evaluated with standard benchmarks and with the realistic end user device workloads.

2 Background and Related Work

NVM as virtual memory. NVMs like PCM are byte addressable persistent devices and are expected to be 100x faster (read-write performance) compared to current SSDs. Compared to DRAM, these devices have higher density scaling as they can store multiple bits per cell with no refresh power, with known limitations imposed by an endurance of a few million writes per cell. These attributes make NVM a suitable candidate for replacing SSDs, but in addition, NVM can also be used as memory, placed in parallel with DRAM, connected via the memory bus. In contrast to SSDs, NVM can be accessed via processor load/stores (read/write), with read speed comparable to that of DRAM, but writes around 10x slower due [3] to high SET and PRESET times. NVM therefore, presents one way forward to solving memory capacity problems as well as fast persistent data access for end clients.

Role of cache. For NVMPersist, NVM's high write latency can be mitigated by using a fast intermediate cache and DRAM, as shown in [23, 15, 11]. For a write-back cache, writes that are evicted from the cache are then moved to the DRAM cache that behaves like a disk buffer cache. This model works well when there is sufficient buffer space (e.g., on high end servers), but when DRAM is scarce (e.g., on mobile devices), forcefully reserving pages for buffering can reduce overall system throughput. The Android OS, for instance, avoids DRAM use for page buffering by disabling swapping. A better alternative is to simply use the processor cache [26, 9]. The unfortunate consequence, however, is that, consistency and durability guarantees require it to be frequently flushed from the cache. We adopt this approach, contributing a thorough study of its performance implications.

Software support for NVM. The usage model for NVM – capacity extension, persistent storage, or dual-use – determines the systems software support needed for NVM management. (1) *Application involvement.* For NVMCap, the system can treat NVMs as a swap device [11, 18, 15], not involving applications. This is not the case for NVMPersist, which requires applications to explicitly identify specific data structures to be saved. A recent work [14] proposes using PCM by intercepting the memory access to specific ranges, and a modified memory(SCM) controller to redirect access to PCM. The key contribution of this work, is to provide atomicity and durability guarantees with efficient use of cache.

(2) *Implementation approaches.* (i) Using existing block I/O interfaces for NVMPersist provides backwards compatibility to legacy applications. (ii) Providing a memory mapped (mmap) interface loses backward compatibility but offers byte-addressability via a system's I/O stack, and requires application level changes for applications using POSIX I/O. (iii) Treating it as managed nonvolatile heap (i.e., via NVM allocation calls) results in byte-addressable NVMPersist, but entirely avoids using the I/O software stack. For (i) and (ii), previous work has shown the importance to redesign the system's I/O stack, as unlike for flash or disk, current I/O software stack is the bottleneck dominating total memory access cost [10, 6]. J.Condit [10] et al. propose shadow buffering for NVM-based files, with epoch-based cache flush methods, but shadow buffering performs poorly for large I/O volumes [10]. Moneta [6] proposes a user level file system to reduce frequent I/O call overheads and consequent large aggregate kernel switch times. File-based I/O accesses are software controlled, however, meaning that read and write will be controlled by the OS, thus not exploiting NVM's byte addressability. Instead, using a mmap() interface leverages NVM's byte addressability, but frequent mmap() system calls (and consequent overheads) force applications to statically reserve large regions of memory and then self-manage their mapped memories. The resulting inefficiencies in memory use are not desirable for memory-limited end user devices.

The issues raised for (1) and (2) above prompt us to treat NVM as nonvolatile heap managed by user level and kernel allocators. Further, and in contrast to prior work [9, 26] using NVM only for persistence, we design interfaces for

```
hash *table = PersistAlloc(entries, "tableroot");
  for each new entry:
    entry_s *entry = PersistAlloc (size, NULL);
    table =  entry;
    count++;
    temp_buff = CapAlloc(size);
```

Figure 1. Using NVM for Capacity and Persistence: An Application's View.

using NVM both for persistence – NVMPersist – and as slow memory –NVMCap, the latter addressing memory capacity issues. Figure 1 shows an example of a persistent hashtable using the NVMPersist persistent allocation interface, as well as creating a temporary volatile buffer using the NVMCap capacity interface. The example presents a persistent allocation interface similar to prior work, with the additional offering of the NVMCap option, to use NVM as volatile memory. The allocations made via these APIs are internally managed by user level and kernel level memory managers supporting them. This is similar to a recent work [17] for OS support for treating NVM as virtual memory, using efficient persistent and non persistent memory managers. In comparison, complementing such work, this paper evaluates and then addresses the overheads of NVM's dual-use as both NVMCap and NVMPersist. Detailed studies of the overheads of managing nonvolatile heap and methods for mitigating them include issues with user level allocators, and overheads related to simultaneously providing strong consistency and durability guarantees.

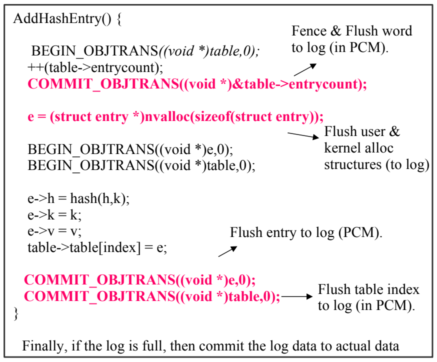
Applications use of NVMcap and NVMpersist. The number of cores and application threads in end clients is increasing along with the increasing DRAM capacity and storage requirement. For instance, take the case of a multi-threaded memory hungry web browser, where the front end browser tab uses NVMCap for additional memory buffer, and the backend browser thread caches user data to transactional database. Similarly, in a multi-threaded game engine, the GUI thread can use NVMCap as graphics buffer, and the game I/O thread can access NVMPersist for storage (load/store user state to the database). For NVMCap-based allocation, the OS NVM manager does not track application (user level allocator) and kernel data structures, but simply allocates pages (like DRAM). But for NVMPersist, both user and kernel data structures are tracked. Hence, for NVMPersist, applications use explicit NVM allocation interface, whereas for NVMCap use of NVM is transparent (by linking to NVM library) and no application level changes is required.

Durability and consistency. NVM hardware-software must support required consistency and durability across application sessions. When using the processor cache to hide write latencies, since cache data can be evicted in any order, to maintain ACID properties, prior efforts have used write-through caches [26], or epoch-based cache eviction methods [10] using memory barriers to order NVM writes. Further, durability can be affected during power failures or device crashes, leaving the application in a non-deterministic state, e.g., due to partial updates (note that both data and metadata must be saved consistently). A common approach to deal with this relies on application commits, which in turn trigger cache flushes. Although sufficiently frequent flushes can reduce the possibility of non-recoverable failures, additional transactional mechanisms are needed for atomicity, accompanied with logging (e.g., undo/redo) support for durability. A recent work [29] proposed hardware-based nonvolatile cache and nonvolatile memory to enable multi-version support with in-place updates (avoids the logging cost). The cache contains the dirty version and the memory contains the cleaner version. While such micro-architectural changes can reduce the cost of logging, we focus on the software optimizations for existing hardware (volatile cache).

Other approaches. Prior work like whole system persistence (WSP) [21] proposed a hardware power monitor to detect power failures and flush cache, processor registers, interrupt signals etc., to the persistent storage. During the restart, the OS and application states are restored like a transparent checkpoint. Two issues with such proposals are (i) they require additional hardware, like a microcontroller-based power monitor dedicated to detecting power failures, and (ii) WSP works for whole system persistence, but when both DRAM and NVMs are used, it is unclear how to distinguish volatile from nonvolatile cache lines, and support for flushing only nonvolatile cache lines may require modifications to the hardware cache. Furthermore, they do not help harden durability guarantees in the event of system crashes. Other nonvolatile heap-based research [9, 26] used strong transactional semantics with word [26] or object-based [9] logging. We discuss these logging methods, and demonstrate the benefits of our proposed novel hybrid (word + object) approach developed in our research.

3 The Costs of Persistence

This section motivates the need for cache efficient, end-to-end solutions when using NVM both for extended capacity and for persistent storage. We analyze the impact of cache sharing between persistent and non persistent applications, and establish that a key factor contributing to increased cache misses is the use of cache inefficient data structures to obtain ACID requirements for data persistence. These inefficiencies arise in both the memory allocator and the logging mechanisms. To analyze the overheads, we implemented a complete NVM software stack (OS/Application library). In keeping with the end client focus of this research, all experimental evaluations use a dual core 1.66 GHz 64 bit D510 Atom-based development kit running a 2.6.39 Linux kernel with our OS-based



```
AddHashEntry() {

  BEGIN_OBJTRANS((void *)table,0);
  ++(table->entrycount);
  COMMIT_OBJTRANS((void *)&table->entrycount);

  e = (struct entry *)nvalloc(sizeof(struct entry));

  BEGIN_OBJTRANS((void *)e,0);
  BEGIN_OBJTRANS((void *)table,0);

  e->h = hash(h,k);
  e->k = k;
  e->v = v;
  table->table[index] = e;

 COMMIT_OBJTRANS((void *)e,0);
 COMMIT_OBJTRANS((void *)table,0);
}

 Finally, if the log is full, then commit the log data to actual data
```

Fence & Flush word to log (in PCM).

Flush user & kernel alloc structures (to log)

Flush entry to log (PCM).

Flush table index to log (in PCM).

Figure 5. Transactional Persistent Hash Table

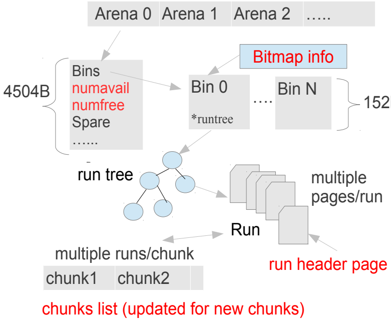


Figure 6. Jemalloc- Rectangular blocks represent C structures. Fields in red are frequently flushed when metadata is in NVM.

NVM support, with 2GB DDR2 DRAM, Intel 520 120GB SSD, 32KB L1 and 8 ways 1MB L2 writeback cache [1]. Out of the total 2GB system RAM, 1GB is used towards NVM. MSR-based performance counters are used to measure cache misses, and the VTune analyzer is employed for function level miss estimation, for both user and kernel code. Applications are run in ways that maintain similar execution times, to better capture the effects of resource sharing.

3.1 Impact of Unmanaged Cache Sharing

Representative end client workloads are used to verify the performance penalties of sharing last level cache across NVMCap and NVMPersist applications, the former using NVM for additional capacity but co-running with a single additional NVMPersist workload – a persistent hashtable (e.g., like those used in key-value stores), labeled as PHT (Persistent Hash Table). Again, the NVMCap applications do not require data persistence and thus, do not flush state of the cache, whereas the PHT NVMPersist-type application frequently flushes cache to obtain consistency and durability guarantees. Figure 5 shows the pseudocode of the PHT with strict transactional guarantees, which we implement using a transactional heap library that guarantees consistency and durability for applications. The pseudocode lines marked in red indicate needed cache flush actions. The X axis in Figure 3 shows representative end client NVMCap applications. X264 is a video conversion application converting a 50MB '. avi' to a mobile compatible '. mp4' file. *B-tree* is a cache efficient data structure commonly used for clients' in-memory databases. Animate provides animation for image files. Each such NVMCap application is co-run with the PHT with random puts and gets for 500K keys (this size is based on our experimental device's available memory resources). The Y axis shows the cache miss (%) increase of co-running these NVMCap applications with the NVMPersist hash table relative to running NVMCap application with a version of the hashtable that is not persistent – NVMCap. Results obtained from reading MSR performance counters demonstrate that while cache efficient NVMCap workloads like B-tree are not heavily impacted by the presence of the PHT NVMPersist application, codes like X264 and animate suffer substantial increase in cache miss rates. Also of interest is the high variability of cache miss rates for NVMCap, an unintended side effect of co-running NVMCap with NVMPersist applications.

We also validate our previous analysis using a cycle accurate instruction level architectural MACSim simulator [2]. We use CPU intensive workloads for this study and replace the I/O intensive animate use case with the end user benchmark WebShootbench [4], a popular workload now used by Google for Chrome OS tablet benchmarking. Also used are some memory intensive and CPU intensive SPEC workloads, since our goal is to investigate the impact of dual-use NVM. To model cache impact, the simulator is modified to identify all cache flush instructions in the trace generated by the PIN tool, invalidate those cache lines and writeback the cache lines if they are dirty. We use writeback cache as prior work [19, 26] have evaluated the performance impact due to write-through cache. As seen in Figure 4, most of the memory intensive benchmarks show substantially increased cache misses and writebacks when co-running with the PHT. Simulation results report only the cache misses incurred by applications, whereas the hardware counter-based measurements using the Intel VTune analyzer in Figure 3 also report cache misses due to OS functions (kernel mode execution of the application), constituting about 11-16% of the overall cache misses observed. The clear conclusion from these experimental evaluations is the need for effective ways to reduce the impact of NVMPersist applications on co-running NVMCap applications, particularly given the ever increasing number of concurrent applications being run on today's end user devices. One way forward is described in Section 4.

3.2 Library Overheads

Preventing NVMPersist applications from impacting the performance of NVMCap applications requires end-to-end solution that begin at user level, for two important components: (1) the memory allocator used by all NVM applications and (2) the logging manager guaranteeing durability for NVMPersist codes. They are each discussed below.

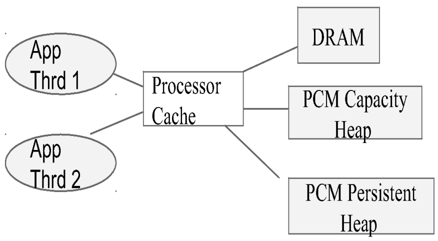


Figure 2. Dual-use NVM: High Level Model

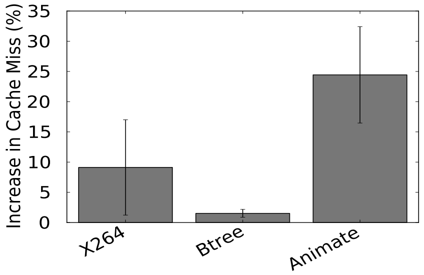


Figure 3. Impact of co-running NVMCap apps with NVMPersist (hashtable)

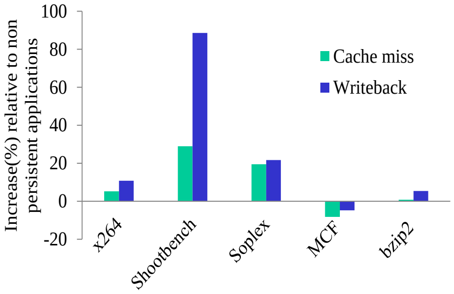


Figure 4. Simulator Analysis.

3.2.1 Cache Inefficient Persistent Memory Allocators

The allocator strongly influences application performance, particularly for data structures requiring frequent allocations (e.g., tree structures, linked lists, key-value stores using hash tables, etc.). Modern allocators, however, maintain complex hierarchical metadata structures for fast free space lookup, object (malloc'd memory) deletion, and more importantly, for reducing fragmentation. Jemalloc (see Figure. 6), for instance, is a multithreaded cache efficient allocator that allocates large regions of memory, called chunks (1024 pages per chunk), where each chunk is further divided into page runs. Each page run maintains a class of uniformly sized objects that vary from 8 bytes to 512 KB. Every pagerun has a fixed number of equally sized objects. The page run has one header with a bitmap to indicate used and freed objects. When an application allocates memory, based on the requested size, a corresponding page run is selected and checks for free objects, and the corresponding bitmap and page run header are updated. For a group of objects in a page run, one header and a bitmap are sufficient. The allocator data structure and application data are placed separately, to keep the application data contiguous and reduce cache misses on application data. For efficient memory usage and to reduce fragmentation, the allocator's metadata is frequently updated.

Most prior proposals, to the best of our knowledge, maintain all allocator metadata in NVM [19, 9, 26]. Yet keeping such frequently updated data on NVM results in a large number of writes to NVM, with consequent numbers of cache flushes [19], thus impacting performance. Further, compared to volatile object allocations, additional metadata is required for each persistent object. This is because for volatile objects, the current virtual address is sufficient to locate an object in a pagerun and update its metadata and per object additional properties are required, but for persistent objects, the virtual address is invalid across restarts. Hence, objects contain additional information to locate them and identify their commit status (some prior work [19] even maintains CRC with each object). Furthermore, every update to the allocator data must be logged and flushed from cache. Such cache flushing writes dirty lines if inconsistent with memory, followed by an invalidation broadcast across cores. This can result in a large number of cache misses experienced by applications, resulting in direct writes to the NVM and consequent application slowdown. In summary, frequent allocator metadata updates will result in substantial 'persistence cost' (e.g., consider a PHT with millions of new entry addition and deletion). Section 4.2 shows solutions that improve upon metadata structures and updates to mitigate these problems.

3.2.2 Durability-based Write Latencies

To provide ACID properties to applications, the NVM stack must support transactional semantics, coupled with a failsafe mechanism where every change to application memory is also logged. Logging mechanisms are used for recovering from failure to a consistent state, and can be broadly classified into 1.) UNDO and REDO methods, 2.) and based on the logging granularity, as word- vs. object-based. For UNDO logging, before every write to a log, the stable version is first copied to a log, whereupon the application can continue writing to the original data location. If a transaction fails, recovery actions copy the stable data from the log back to the original memory location. For REDO logging, all writes are appended to a log, and when a log fills up, the log entries are copied to original memory locations. With respect to logging granularity, prior NVM works uses either (i) word-based logging [8, 24, 26], where every word is logged along with log metadata (described shortly) or (ii) an object-based log for NVMs [9], where the entire object is copied to log. We next discuss problems with this current state of the art.

Each log entry is a record consisting of a metadata and the actual data stored in different locations. The record contains the actual word address, a pointer to the data in a log, and a pointer to next log record. To log a word of data (8 bytes), 24 bytes of data must be written to NVM, thus drastically increasing the overall writes to NVM. Further, word-based logging requires substantial rollback time (scan word by word and apply updates). Recent work avoids repeated updates by logging at object granularity [5, 9]. While this scales well for large objects, updates involving smaller member variables or counters of an object (e.g., updating a counter in a hash table structure when new entries are added), the object copy cost from its actual address to the log (in case of UNDO) or back from a modified object to an actual data address (in case of REDO) can be substantial. Further, cache misses increase with increasing object copy sizes, resulting in slower NVM access. Section 4.3 describes a novel hybrid logging approach that combines word-based and object-based logging to provide an adaptive approach to reduce NVM write latency issues. The approach does not require substantial developer effort to classify word- and object-based logging.

4 NVM-Efficient End-to-End Software

This section describes solutions to the cache inefficiencies identified in the previous section. It first describes a physical page contiguity-based page allocation mechanism that seeks to partition the cache entries used by NVMPersist vs. NVMCap applications. This simple but effective solution avoids the complexity of using traditional page coloring methods for this purpose. Second, allocator metadata management is improved to reduce allocator overheads, the key idea being to maintain complex allocator metadata structures in DRAM and logging their updates in NVM. Third, we reduce the cost of transaction logging via a hybrid logging mechanism that automatically adapts to the appropriate logging granularity (word vs. object). While our cache partitioning mechanism reduces the cache impact of NVMPersist apps on NVMCap apps, the allocator and logging optimizations reduce the cache misses suffered by NVMPersist applications and hence improves the overall cache misses. Experimental evaluations for the effects of each such NVM write-aware optimization are run on the same Atom-based development platform described in Section 3, using the same methods to gather experimental results via the MSR performance counters. All evaluations use realistic end user NVMCap applications that include ’animate’ used for image animation, ‘x264’ used for 64 MB file mp4 conversion, ‘convert’ used for converting images from .jpeg to .png. We also use SPEC 2006 benchmark applications that are more representative client-side applications, like ‘astar’ which is a portable 2D path-finding library, ‘povray’, a ray tracing application, and other memory intensive benchmarks like ‘omnetpp’, ‘mcf’, ‘soplex’ and computation intensive benchmarks like ‘sjeng’, and ‘libquantum’. Details about these benchmarks concerning their cache and memory intensity can be found in [16].

4.1 Reducing the Cache Sharing Impact

Cache sharing between NVMCap and NVMPersist applications can result in increased conflicts, false sharing, higher NVM writes, and reduction in memory bandwidth due to increased NVM traffic. We hypothesize that increased cache misses experienced by NVMCap applications, caused by co-running NVMPersist application, can be avoided by partitioning the shared cache across these applications. Hardware and software cache partitioning strategies for multiprocessor systems have been extensively studied in the past. Hardware mechanisms [20, 7, 22, 25] include simple static as well as dynamic partitioning methods, the latter monitoring the cache miss suffered by applications, and then adjusting the number of cache ways between NVMCap and NVMPersist applications. Software-based partitioning approaches [28] typically use page coloring mechanisms, which we describe shortly. Generally, hardware approaches have shown higher benefits [27] compared to software partitioning, but software partitioning provides the flexibility to easily enable/disable page coloring, and offers scope for various application-specific optimizations.

We study the effectiveness of cache partitioning using the cycle accurate MACSim simulator, by statically partitioning one 1MB LLC cache to use 3/4 of the cache sets dedicated to NVMCap application, and 1/4 for the NVMPersist applications. The analysis uses 500 million instructions of the same applications as those used in Figure 4. Note that most memory-intensive applications show up to 12% improvement in performance, gained from cache partitioning, while there is no or little impact on other benchmarks like bzip, and MCF.

Page Coloring. OS-based cache partitioning between applications using software page coloring has been studied extensively. Implementing page coloring perfectly requires substantial changes to the memory management layer of the OS [12]. Further, even when such changes are present, due to increasing cache associativity, operating systems often disable the page coloring feature. For end client devices with their few way caches (4-8 ways) and given our diagnosis of high conflict misses with NVMCap and NVMPersist applications, however, we posit the need to revisit page coloring. We therefore, prioritize two goals for suitable OS-based cache partitioning methods: (1) to reduce the complexity of page allocation (i.e., to avoid looking for specific pages at the time of page allocation) of existing, low overhead page coloring mechanisms [12, 27], and (2) to make it easy to disable OS-based partitioning when only NVMCap applications are currently running (i.e., no co-running persistent applications), or when there is little or no impact of persistent on non-persistent applications.

Page Contiguity Based Partitioning. We propose a novel adaptive method for cache partitioning that leverages the high probability with which current caches map contiguous physical pages to contiguous cache lines. The key idea is to increase the physical contiguity of pages allocated to an application. Intuitively, the more contiguous an application page, the more contiguous its cache lines, and the less likely the cache interference with other applications sharing the same cache. This is in contrast to non-contiguous allocations in which different applications’ random pages are mapped to various cache lines, thus increasing the chances of cache conflicts.

Allocating a single page during first touch and page fault

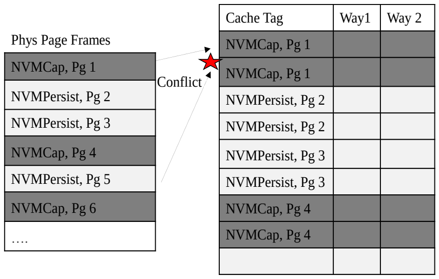


Figure 7. Cache conflicts due to JIT allocation. NVM-Cap & NVMPersist page maps to same set.

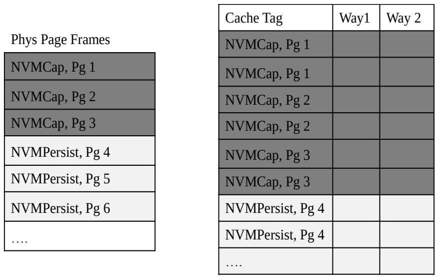


Figure 8. Reducing conflicts with CAA.

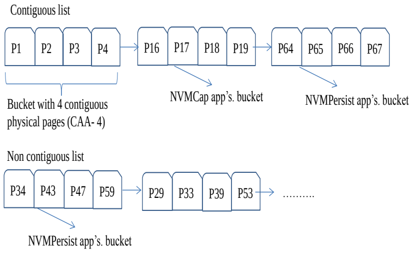


Figure 9. CAA bucket-based design. PX - physical page no. X.

can result in high page contiguity misses. We call this policy JIT (just in time) allocation. For instance, let N be the number of applications simultaneously accessing/allocating NVM pages. With JIT allocation, the probability of an application receiving a physically contiguous page reduces to (1/N). Hence, with an increasing number of co-running applications, cache conflicts increase. Reducing the number of physical page (physical frame) contiguity miss can substantially reduce cache misses. The key idea to increase the contiguous pages allocated to application is that, for page faults (minor faults on first touch), instead of allocating one page, a batch of physically contiguous pages is reserved for the application, and on subsequent page faults, specific physical contiguous pages are added to the page table. We refer this approach as contiguity aware allocation (CAA).

Implementation of CAA. As a first step, to avoid contiguity misses, we create two types of lists in the kernel: 1) a contiguous list, and 2) a non-contiguous list. Each list contains one or more buckets. Each bucket in the contiguous list contains an array of physically contiguous pages. For instance, Figure 9 shows a contiguous list with three buckets and each bucket contains 4 physically contiguous pages. A list here refers to linked list of buckets. Buckets in the non-contiguous list (the list at the bottom of the figure), contain pages that are ordered but not contiguous.

For adding contiguous pages to the bucket, when there is a page fault, a batch of contiguous physical pages is allocated and added to an application specific bucket in the contiguous bucket list. While only the page corresponding to faulting address is added to page table, the other contiguous pages are used during subsequent page faults thereby using physically contiguous pages as shown in the Figure 8. Our design creates a separate bucket for each application, and as the pages of the buckets are exhausted, new batch allocations refill the bucket. It is not always possible, that contiguous batch allocations succeed (and depends on memory availability). In the event that batch allocations return non-contiguous pages, such pages are moved to a bucket in the non-contiguous list. We avoid creating multiple buckets so as to increase the locality of the bucket data structure in the cache, but can easily support multiple buckets per application thread.

Our approach divides the applications into cache friendly and non cache friendly applications. All applications are allocated from contiguous buckets initially. We maintain two memory watermarks (higher: less critical, lower: highly critical). As the free available memory reaches less than the 'higher water mark', we start using the non-contiguous pages for NVMPersist applications, and when the memory limit reaches less than the low water mark, we disable page contiguity aware allocation. Our approach substantially reduces the page contiguity misses and reduce the cache misses due to conflicts as shown in our evaluation.

Evaluation Baseline. In all evaluations, as a baseline, We use the PHT (persistent hash table) as the NVMPersist application. The PHT uses the current JIT-based OS page allocation combined with a naive allocator that stores/access all its complex allocator metadata in NVM, and uses a word-based logging as proposed by prior 'NVM as a heap' research [26]. We also report average (across all workloads) cache miss reduction after applying each optimization.

Page Contiguity Miss Analysis. We next evaluate the effect of bucket size (the number of contiguous pages) on the page contiguity misses. In Figure 10, the X-axis shows several client and SPEC benchmarks, and the Y-axis show the percentage of reduction in physical page contiguity miss compared to JIT allocation. We evaluate our experiments for two different bucket sizes – CAA-4, CAA-16, where 4 and 16 indicates the number of contiguous pages allocated in a batch and added to a bucket in the contiguous list when handling a page fault. As seen from the results, we are able to reduce the physical contiguous page misses for applications by up to 75% for CAA-4, and 91% for CAA-16, for both client and SPEC workloads, compared to JIT-based method. Table 1 shows the actual number of page contiguity misses for some of the applications using all the three (JIT, CAA-4, CAA-16) methods. Applications like x264, animate, sjeng, show a higher reduction in miss counts, however, their expected benefits due to physical contiguity can be small since the total contiguity miss for these applications are substantially low even when using JIT-based allocation. In comparison, for memory-hungry applications like to convert, soplex, astar, the benefits due to page contiguity is higher. We next evaluate whether page contiguity miss reductions results in reduced cache misses of NVM-Cap applications.

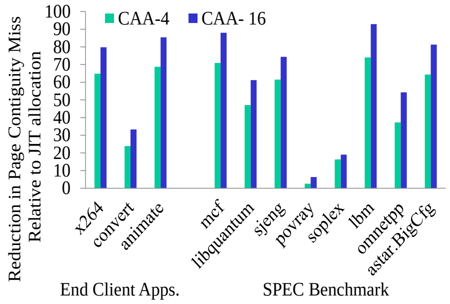


Figure 10. Page Contiguity Miss Analysis

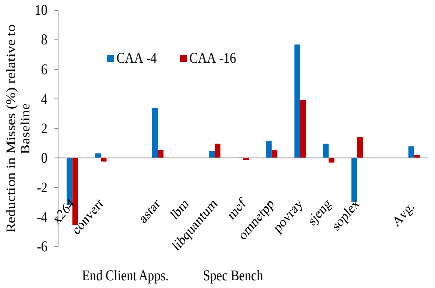


Figure 11. Overall Cache Miss Analysis.

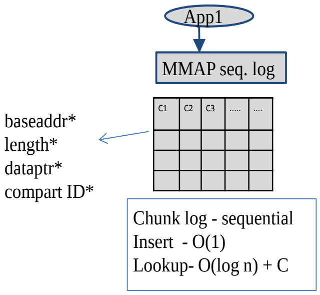


Figure 12. Two level log-based PCM allocator

App	JIT	CAA-4	CAA- 16	Expec. Benefits
x264	4890	1720	990	Low
convert	177340	135080	118360	High
animate	4860	1520	710	Low
povray	8160	7950	7640	Low
soplex	989560	828080	801090	High
astar.BigCfg	107360	38310	20120	High

Table 1. Page contiguity miss count

Reduction in overall cache misses. Figure 11 shows the relative reduction (in percent) in cache misses for the same set of benchmarks compared to the baseline. We use two different contiguity-aware allocation bucket sizes, CAA-4, and CAA-16. We observe that, with CAA-4, the benefits in cache miss reduction vary from 1% to 8%. While improvements are observed in most applications, for some applications like x264 and soplex, the misses increase compared to the baseline. Maximum gains are seen for memory intensive astar, and for the moderately intensive povray. Also, when varying the bucket size from CAA-4 to CAA-16, for applications with smaller memory footprint, like animate, sjeng, cache misses increase. Page contiguity for these applications does not provide much benefit, as the number of pages allocated by these applications is relatively small, the impact due to cache sharing is relatively less and the working set of these applications fits well into cache. When using CAA-16 during page fault for these applications, the kernel tries to allocate contiguous pages in a batch and then add them to the bucket linked list. The traversal across linked lists to add new pages or remove free pages adds cache overhead, but with no benefits due to physical page contiguity. Such effects would be higher when the bucket size is further increased, outweighing the benefits of page contiguity. For applications with larger memory footprint (soplex, libquantum, omnetpp), CAA-16 based allocation provides more reduction in cache misses compared to CAA-4. These results emphasize the fact that *physical page contiguity-aware allocations reduce the cache sharing impact of NVMCap and NVMPersist applications and that using the right granularity of bucket sizes (i.e., CAA-4 vs. CAA-16), based on applications' memory usage, can lead to higher cache miss reduction*. The average reduction in cache misses due to CAA is around 1%. We do not run the animate benchmark due to its high memory requirements of the Atom platform. For further improvements, we next focus on NVM write-aware allocators.

4.2 Addressing Allocator Overheads

Cache misses and writebacks due to application allocators can be reduced with an NVM write-aware allocator (NVWA). The key idea is to keep the complex hierarchical allocator metadata in DRAM, and maintain only an allocator data structure-independent log in NVM. The log of all NVM allocations, deletions, or reallocations by application threads is sufficient for restarting and rebuilding the allocator state. The allocator log is always kept strongly consistent with the actual DRAM metadata state by flushing to NVM. The log data structure in NVM is written sequentially and writes are aligned with cache boundaries.

Figure 12 shows a two-level allocator log. For every memory allocation by an application, the first log level contains information about allocated chunks, and the second level contain information about the physical page to locate the corresponding chunk. For deletion of a memory chunk, there are two possibilities based on available storage space: (1) the log can be parsed sequentially, and the corresponding entry is invalidated (by marking a exclude bit for garbage collection), or (2) one can just 'append' chunk deletion information to the log. For (1), when there is insufficient NVM space, to avoid sequential log parsing, we maintain an in-memory (DRAM) red-black tree with chunk pointers, which makes it possible to locate and update chunk status in O(log n) time. For example, just appending a log entry for a new chunk and for deleting a new chunk (say a million entries hashtable) can consume twice the allocated data size. Further, these updates can cause an additional cache miss overhead. When NVM space is not a constraint, method (2) above can be used. For deletion, only a single bit is modified to indicate whether the chunk is still useful, and for reallocation, only the length field is updated. The benefits of such a log-based approach are:

- During most metadata updates, no more than two cache line flushes are required, unlike with prior research [19] in which a hierarchy of allocator data is maintained and flushed to NVM.
- Updates to NVM logs are mostly sequential, cache aligned, and hence, no more than two cache line flushes are required.
- Maintaining simple persistent restart metadata independent of allocator metadata provides the flexibility

of using different allocators.

- A final benefit is a reduction in the total amount of persistent data kept in NVM, because only the log is persistent.

Allocator Overheads. We next analyze the implications of the allocator optimizations using microbenchmarks and representative client applications.

Microbenchmark analysis. We run only the persistent hash table (PHT) benchmark using the default persistent Jemalloc allocator referred as a 'naive allocator' that keeps all of its complex allocator structures in NVM. This is compared to the cache misses seen with the PHT using the NVM write-aware allocator (NVWA). We vary the number of PHT operations that include random hash table puts and gets. Along the x-axis in Figure 13, we vary the number of elements in a hash from 600K to 1.5Million, each with key and value of 64 bytes. The Y axis shows the increase in cache miss percentage compared to a baseline ideal PHT that does not flush the allocator metadata. As can be observed, in all cases, the NVWA approach outperforms the naive allocator by around 3%. This is mainly because of the reduced number of cache line flushes by NVWA compared to the naive allocator, with only 2 flushes per every newly allocated memory chunk and a single cache line flush for deletion or resize operations. Table 2 compares the total number of allocator-specific cache line flushes of both approaches. Observe that the NVWA allocator reduces flushes as high as 8x compared to the naive allocator, thus substantially reducing allocator-specific misses.

Application benchmarks. Next, we compare the impact of the NVWA allocator on all of the application benchmarks used in Section 4.1. Figure 14 compares the cache misses under different allocator designs. For memory intensive client benchmarks like 'animate', we observe close to a 2% improvement when using NVWA, whereas 'convert' shows less than a 1% improvement. Surprisingly, the lesser memory intensive x264 also experience substantial benefits from using NVWA. This is because x264 processes target files in frame size (192 bytes) granularity. Hence, for a 62MB file, the number of allocation-related flushes are substantial for the naive vs. the NVWA allocator. The 'convert' application, in comparison, experiences a relatively smaller number of allocations. Similar trends are observed in memory intensive benchmarks, where the benefits are higher for memory capacity intensive applications like soplex (4%) and libquantum (2%), but other benchmarks show less than a 1% improvement.

Hash Elements(millions)	Naive	NVWA
.5	2500036	500032
1	5000044	1000043
1.5	7500052	1500054

Table 2. Cache Line Flush Comparison.

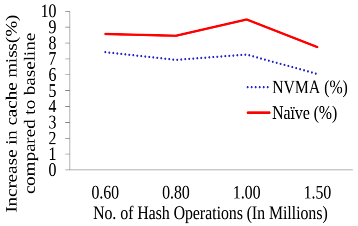


Figure 13. Allocator Metadata Persistence Cost Analysis.

An interesting difference between our representative client applications and the SPEC benchmarks is that with realistic client applications, the total number of allocations is larger, and are done throughout the applications' lifetimes, whereas the SPEC benchmarks (including memory intensive benchmarks), have fewer allocations, mostly grouped in their initial execution stages. This abnormal behavior of the SPEC benchmark means that the impact of interference on NVMCap applications due to persistent allocations is relatively smaller. We conclude that gains from Using an NVWA depends both on the total number of allocations/deletions made by the applications and on when such operations are performed. Figure 15 quantifies the total reduction in cache misses from using the NVWA allocator compared to the ideal baseline.

4.3 Hybrid Logging

The purpose of the hybrid logging method introduced in this paper is to reduce NVM writes due to application logging, while still providing with failsafe durability guarantees. Specifically, hybrid logging (i) reduces the metadata writes for the word-based log (every word data logged requires three words of metadata), and (ii) reduces the data writes of object-based logs in which entire object is copied even when only a single word in an object has been changed. Hybrid logging provides developers with a flexible object- and word-based logging interface that permits them to pass hints to the write aware NVM log manager concerning the granularity of changes. In Figure 17, for hash table data structure changes, incrementing the entry count in the hash table or dereferencing to key/value pairs, are word size updates, and the developer can commit these by passing the word logging hint. For changes larger than the word size, e.g., when modifying the key/value object, object-based logging makes it unnecessary to maintain a log record for every word of an object.

Implementation details in Figure 16 show how hybrid logging is used with redo logs (undo logging is supported, as well). The object and word logs for an application are maintained independently. Further, for each log (word and object), the log metadata and actual log data are maintained in separate locations, by mapping a fixed contiguous size of NVM. Further, the log truncation frequency is set to LLC size (1MB in our case). Separating the log metadata from the log data enable easier traversal of log metadata for retrieval or clean up. When an application developer de-

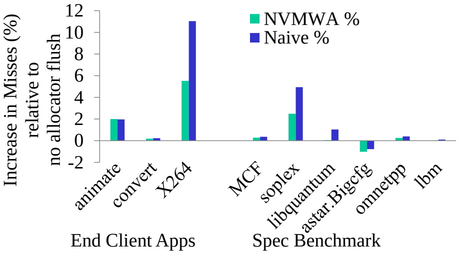


Figure 14. Allocator Cache Miss Reduction(%) compared to Naive approach.

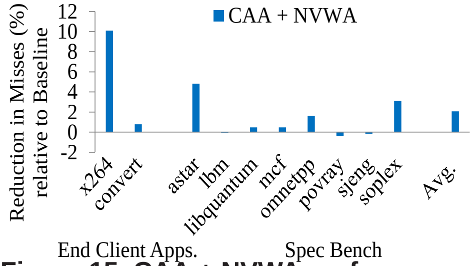


Figure 15. CAA + NVWA performance

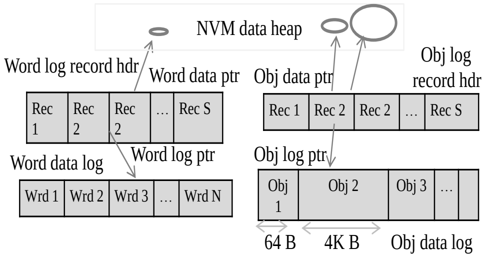
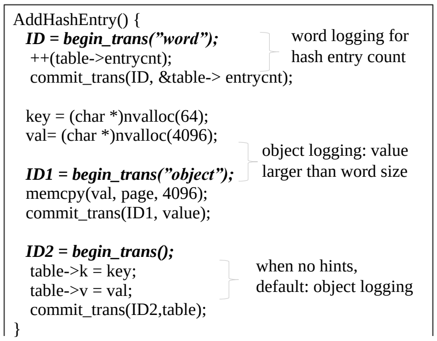


Figure 16. Hybrid logging design



```
AddHashEntry() {
  ID = begin_trans(”word”);                    word logging for
   ++(table->entrycnt);                        hash entry count
   commit_trans(ID, &table-> entrycnt);

  key = (char *)nvalloc(64);
  val= (char *)nvalloc(4096);
                                               object logging: value
  ID1 = begin_trans(”object”);                 larger than word size
  memcpy(val, page, 4096);
  commit_trans(ID1, value);

  ID2 = begin_trans();
   table->k = key;                             when no hints,
   table->v = val;                             default: object logging
   commit_trans(ID2,table);
}
```

Figure 17. Hybrid logging interface

cides to commit persistent data to NVM, a virtual address is passed, along with a hint of the log granularity. The log record has a transaction ID (TID), the address/offset that points to log data, a pointer to the original data address (not the log data address). The monotonically increasing TID helps with data version conflicts, i.e., conflicts of the log records with the same virtual address are solved using the TID (higher ID number indicates newer log data). For word-based logging, log data and log metadata are similar in size. For object-based logging, the log metadata contains additional fields indicating the size of of the objects logged. Again, key benefits of this hybrid logging approach are that by providing a flexible hybrid interface, (i) the high ratio of log metadata/data is reduced, by avoiding logging every word change, and (ii) high log data costs are reduced, by avoiding logging the entire object even when the data change is less than a word.

Hybrid Logging – Experimental Evaluation. Similar to the allocator evaluation, we run the same PHT benchmark enabled with two different logg ing methods: 1.) word-based logging and 2.) the proposed hybrid logging. The X axis in Figure 18 shows increasing numbers of hash operations that would consequently lead to increasing numbers of logging operations. The Y axis, shows the percentage reduction in relative cache misses using the hybrid logging approach. We use the same hash table implementation code as in prior work [26] with a similar transaction interface. For every hashtable key insert, in the case of hybrid logging, 5 word level transactions are replaced by one object transaction, and for a delete operation, 3 word transactions are replaced by one object transaction. This reduces the total NVM writes of 120 bytes (5 log entries with 24 bytes each entry) to 28 bytes (three 8 byte record pointers (see Figure 16) plus a 4 byte object size field) when adding a new key and value to a hash. The outcome is a reduction of an average of two cache line flushes to one cache line flush. Additional optimizations like flush batching can combine multiple object record updates (approx. 3 log records) to one flush. The data written to log would be the same for both word-based or hybrid logging, as it is application dependent. We observe that irrespective of the number of hash operations, the relative reduction in cache misses with hybrid log is almost constant, as expected. Additional benefits of hybrid logging can be expected when batching the log metadata flushed. Further, the impact of our optimizations on NVMPersist application was less than 0.7%.

To understand the effectiveness of our mechanisms on other cache efficient applications, we used the persistent B-tree described earlier in section 4 with 1.5M operations (same as PHT). Figure 21 shows the performance gains of B-tree using all our optimizations. As expected, gains for cache efficient B-tree was restricted to less than 3.5%, and the average gain was around 1.2%. Most benefits (73%) for B-tree was due to the allocator optimizations and the rest from the page contiguity method with no logging related gains. This is because, each B-tree node was aligned to a word size (we used 8 byte ints as node values), and hence, use of a hybrid approach instead of default word logging was not required. While our current optimizations show less benefits for cache efficient B-tree, our future work would explore more such optimizations.

4.4 Discussion

With the goal of enabling dual use of NVMs in end clients, we discussed three optimizations that include page contiguity-based cache partitioning implemented at the OS level, an NVM write aware allocator for reducing the allocator NVM updates, thereby reducing the cache impact on NVMCap applications, and finally, we show the benefits of the hybrid logging mechanism. Each mechanism incremen-

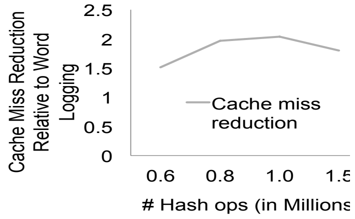


Figure 18. Logging microbench- PHT

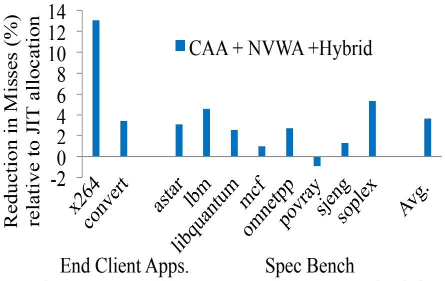


Figure 19. CAA+NWAA+Hybrid log gains

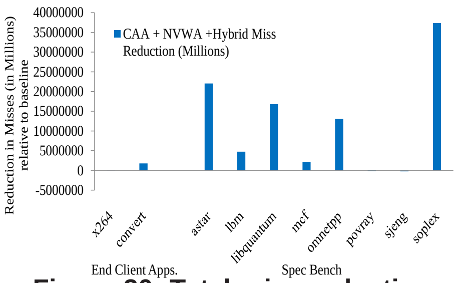


Figure 20. Total miss reduction CAA+NWAA+Hybrid

tally improves the average reduction in cache misses (up to 3.6-4%) and up to 12% in some end client applications.

Figure 19 shows the overall effectiveness of combining all three optimizations: page contiguity, write aware allocator, and finally hybrid logging. In most application benchmarks our proposed mechanisms provide 1-6% benefits and in some benchmarks like x264, the benefits are more than 12%. While in the case of x264, the benefits primarily are from allocator optimizations, in case of other workloads, the benefits add up from all the three optimizations discussed. In some workloads like ‘mcf’ we noticed less than 1% improvement. In case of ‘povray’, we found the working set of application to be comparatively less, and also most memory allocations are by these applications are done towards the initial stages and our optimizations are not effective. Figure 20 shows the total number of misses reduced with CAA+NWAA+Hybrid methods compared to baseline.

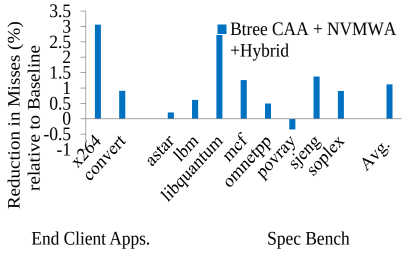


Figure 21. B-tree gains

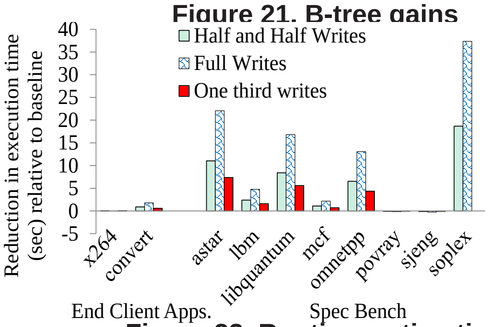


Figure 22. Runtime estimation

Simple Execution Time Estimation. Our design optimizations and evaluations specifically focus on methods to reduce the cache misses of NVMCap and NVMPersist applications. Reducing cache misses reduces the need for direct access to NVM, thus avoiding execution time overheads due to poor NVM write latencies compared to DRAM. Our evaluation shows that for some applications our approach leads to up to 12% reduction is cache misses, where for others the reductions are more modest. Based on prior studies, however, we believe that even 1% decrease in the total cache misses suffered by an application can have a substantial performance impact on the end client applications [13].

Performing an accurate and direct assessment of the impact of our methods on execution time is challenging, however. Currently, PCM devices are not commercially available, and emulating varying NVM read/write latencies using DRAM is not possible or accurate. Further, hardware performance counters for end client devices (Atom) do not classify cache misses due reads vs. writes. Hence, to provide a simple back of the envelope estimation, we use PIN-based instrumentation similar to several other prior efforts [9, 26, 19] to capture total the NVM read/write and estimate the runtime impact of our optimizations. We use three models ‘Half-Half’ - half of the misses reduced by our methods is NVM writes, ‘Full writes’- all the reduction is for NVM writes, and finally, one-third of misses are reduced for NVM writes. Using the write latency projections of 1 microsecond from [3] and assuming DRAM read latency for NVM reads, Figure 20 shows a simple projection of the execution time reduction. We stress the fact that, our current scope of work is limited to reducing NVM writes (i.e., cache misses) by software optimization. Also, our estimation does not consider features like out of order execution, parallel issue of read/writes to memory in modern processors and should be viewed as a worst case analysis. As expected, when the number of cache misses due to NVM writes increases (Full writes), our optimizations can provide substantial performance gains for memory intensive benchmarks. Even when the write misses are substantially less (one third due to writes), our optimizations can improve the application execution time by around 6%-8%. Our future work will focus on a more detailed investigation of the execution time impact of our techniques.

5 Conclusions

This paper analyzes the dual-use of NVM, for memory capacity and for persistent storage, in end client devices. Analyzing NVM writes, we find that effective dual-use NVM requires new methods that address cache sharing between persistent and non-persistent applications, in addition to optimizations to the memory subsystem's software stack, including allocators and logging. Cache sharing solutions use a novel contiguity-based approach to memory allocation, which reduces by up to 10% the overall cache sharing impact experienced by non-persistent applications and caused by persistent co-runners. Further improvements are obtained with an efficient write aware persistent allocator, leading to reductions in the overall cache misses of up to 12% for client applications and around 4% for SPEC benchmark. Finally, an NVM write-aware hybrid logging approach substantially reduces the NVM writes. An interesting outcome of this research is that for efficient use of NVM, there is an experimentally demonstrated need for end-end solutions that include optimizing the ways caches are handled, changing the memory allocators used by the operating system and by application libraries, and considering carefully the manner in which persistence guarantees are made, e.g., through logging.

While this paper's contribution specifically focuses on reducing NVM writes, by avoiding cache misses, an interesting and important future work is to accurately assess the performance and power benefits from such reductions in NVM access. Also of interest is an analysis of additional persistent and non-persistent applications, coupled with a careful look at how such applications and the library and OS functionalities described in this paper would be implemented in the popular Android OS for end client devices.

6 Acknowledgments

This research is supported in part by the Intel URO program on software for persistent memories and by NSF award CCF-1161969.

References

[1] Intel Atom - Xolo. `http://www.xolo.in/`.

[2] MacSim: A CPU-GPU Heterogeneous Simulation Framework. `https://code.google.com/p/macsim/`.

[3] Numonyx pcm characteristics. `http://bit.ly/e48Gdh`.

[4] WebShootOut browser benchmark. `bit.ly/19qkSnr`.

[5] C. S. Ananian and M. Rinard. Efficient object-based software transactions. In *SCOOL '05*.

[6] A. M. Caulfield, A. De, J. Coburn, T. I. Mollow, R. K. Gupta, and S. Swanson. Moneta: A high-performance storage array architecture for next-generation, non-volatile memories. MICRO '43, 2010.

[7] J. Chang and G. S. Sohi. Cooperative cache partitioning for chip multiprocessors. SC 07.

[8] J. Coburn, T. Bunker, R. K. Gupta, and S. Swanson. From aries to mars: Reengineering transaction management for next-generation, solid-state drives. In *UCSD CSE Technical Report*, 2012.

[9] J. Coburn, A. M. Caulfield, A. Akel, L. M. Grupp, R. K. Gupta, R. Jhala, and S. Swanson. Nv-heaps: making persistent objects fast and safe with next-generation, non-volatile memories. ASPLOS XVI.

[10] J. Condit, E. B. Nightingale, C. Frost, E. Ipek, B. Lee, D. Burger, and D. Coetzee. Better i/o through byte-addressable, persistent memory. SOSP '09.

[11] G. Dhiman, R. Ayoub, and T. Rosing. Pdram: a hybrid pram and dram main memory system. DAC '09.

[12] M. Dillon. Design elements of the freebsd vm system. `http://tinyurl.com/kftjzqy`.

[13] R. Duan, M. Bi, and C. Gniady. Exploring memory energy optimizations in smartphones. IGCC '11.

[14] E. Giles, K. Doshi, and P. Varman. Bridging the programming gap between persistent and volatile memory using wrap. CF '13.

[15] E. Ipek, J. Condit, E. B. Nightingale, D. Burger, and T. Moscibroda. Dynamically replicated memory: building reliable systems from nanoscale resistive memories. *SIGARCH Comput. Archit. News*, 38(1), Mar. 2010.

[16] A. Jaleel. Memory Characterization of Workloads Using Instrumentation-Driven Simulation. `http://bit.ly/15zntbv`.

[17] S. Kannan, A. Gavrilovska, K. Schwan, and D. Milojicic. Optimizing checkpoints using nvm as virtual memory. IPDPS 12.

[18] B. C. Lee, E. Ipek, O. Mutlu, and D. Burger. Architecting phase change memory as a scalable dram alternative. In *ISCA09*.

[19] I. Moraru, D. G. Andersen, M. Kaminsky, N. Binkert, N. Tolia, R. Munz, and P. Ranganathan. Persistent, protected and cached: Building blocks for main memory data store. In *CMU tech report*, 2011.

[20] F. Mueller. Compiler support for software-based cache partitioning. LCT-RTS 95.

[21] D. Narayanan and O. Hodson. Whole-system persistence. ASPLOS XVII, 2012.

[22] M. K. Qureshi and Y. N. Patt. Utility-based cache partitioning: A low-overhead, high-performance, runtime mechanism to partition shared caches. MICRO 39, 2006.

[23] M. K. Qureshi, V. Srinivasan, and J. A. Rivers. Scalable high performance main memory system using phase-change memory technology. In *ISCA '09*.

[24] T. Riegel, C. Fetzer, and P. Felber. Time-based transactional memory with scalable time bases. SPAA '07.

[25] G. Suh, L. Rudolph, and S. Devadas. Dynamic partitioning of shared cache memory.

[26] H. Volos, A. J. Tack, and M. M. Swift. Mnemosyne: lightweight persistent memory. In *ASPLOS '11*.

[27] X. Zhang, S. Dwarkadas, and K. Shen. Hardware execution throttling for multi-core resource management. USENIX ATC '09.

[28] X. Zhang, S. Dwarkadas, and K. Shen. Towards practical page coloring-based multicore cache management. EuroSys '09.

[29] J. Zhao, S. Li, D. H. Yoon, Y. Xie, and N. P. Jouppi. Kiln: Closing the performance gap between systems with and without persistence support. MICRO '46, 2013.