Persistent, Protected and Cached: Building Blocks for Main Memory Data Stores

Iulian Moraru[1], David G. Andersen[1], Michael Kaminsky[2],
Nathan Binkert[3*], Niraj Tolia[3*], Reinhard Munz[1*], Parthasarathy Ranganathan[3]

[1] Carnegie Mellon University, [2] Intel Labs, [3] HP Labs

CMU-PDL-11-114 v2 (supperseds CMU-PDL-11-114)

November 2012

*Work done while the author was at the specified institution.

Parallel Data Laboratory
Carnegie Mellon University
Pittsburgh, PA 15213-3890

Acknowledgements: Hewlett-Packard Labs, Intel Science & Technology Center for Cloud Computing, Google, and the National Science Foundation under award CCF-0964474. We would like to thank the members and companies of the PDL Consortium (including Actifio, American Power Conversion, EMC Corporation, Emulex, Facebook, Fusion-io, Google, Hewlett-Packard Labs, Hitachi, Huawei Technologies Co., Intel Corporation, Microsoft Research, NEC Laboratories, NetApp, Inc., Oracle Corporation, Panasas, Riverbed, Samsung Information Systems America, Seagate Technology, STEC, Inc., Symantec Corporation, VMware, Inc., and Western Digital) for their interest, insights, feedback, and support.

Keywords: NVRAM, PCM, Main Memory

Abstract

This paper looks at systems design for consistent, durable, and safe memory management for future byte-addressable non-volatile (NV) memory. Specifically, we focus on how application-level interfaces need to change to accomodate this memory on the main memory bus and propose (1) a new NV-memory-aware memory allocator that incorporates wear leveling optimizations and stronger robustness to avoid data corruption—NVMalloc, (2) a new low-overhead mechanism for containing erroneous writes using an asynchronous implementation of mprotect, and (3) a new application interface to track and enforce update consistency at the processor cache level using cache line counters. Our proposed optimizations are designed holistically across the application, OS, and hardware layers.

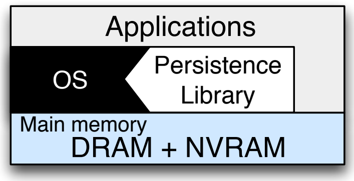
We implement our optimizations and evaluate their benefits for realistic scenarios (including a simulation study of the hardware changes). Across three different evaluations—a stress microbenchmark, an in-memory B+ tree, and Memcached—our proposed memory allocator achieves better write distribution and robustness than existing allocators, with low fragmentation and overhead. Similarly, our evaluation shows that our interfaces for asynchronous memory protection and cache-level update consistency can achieve significant improvements in throughput compared to conservative approaches to enforcing safety, and competitive to designs with no support for improved safety.

Compared to prior work, our paper is, to the best of our knowledge, the first to propose a general-purpose and flexible approach to rethinking the interfaces across the application, OS, and hardware layers for future high performance NV memory systems.

1 Introduction

For decades, with a few niche, expensive exceptions, fast memory has been volatile, and persistent storage has been slow. The advent of non-volatile, byte-addressable memories may force an overhaul of program and operating system structure comparable to that required to deal with multicore. There are multiple competing technologies (Phase Change Memory, Spin Torque Transfer, memristor) but the end result is the same: a non-volatile, byte-addressable memory (we refer to it as *NVRAM*) that, when placed directly on the memory bus, will be almost as fast to access as DRAM.

We expect that uses for fast persistent memory will range from easy to program but low-performance (e.g., using it as a replacement for Flash memory), to complex and challenging setups meant to benefit performance-critical applications. In this paper we focus on the latter category. For reasons outlined in Section 2.2, we argue that performance-critical applications will require placing fast non-volatile memory on the memory bus alongside DRAM, and accessing both through CPU loads and stores:



Furthermore, we argue that harnessing this performance can best be achieved through combined hardware-software designs—as opposed to exclusively hardware or exclusively software solutions.

The fundamental challenge when using non-volatile main memory is to *attain high throughput while providing strong durability guarantees*. Many more things can go wrong when using the main memory bus to access persistent storage. One challenge is *wear-out*: non-volatile memories degrade with writes. Although not nearly as fragile as NAND Flash, excessive writes to an NVRAM memory cell can still rapidly destroy it. A second challenge comes from the interface presented to the application writer with respect to durability and safety. Today's storage interfaces are *narrow* and *explicit*: Writes to storage can only affect data; only the kernel can change filesystem metadata. A `write()` makes data durable (in theory, depending on the implementation), and writes to hard disks typically succeed or fail atomically on a page or sector granularity, enforced by hardware sector checksums. With non-volatile memory, however, erroneous writes can affect persistent data or metadata. The order of writes is unknown unless special attention is taken to flush writes, due to interactions with processor caches.

Decades of experience with filesystems and databases have shown that developing safe and fast persistent data stores is challenging, and we do not believe that emerging memory technologies will change this. This paper describes three building blocks to ease the task of the persistent data structure designer (née filesystem or database designer), who develops libraries that would be linked against by application developers. These building blocks help address some of the primary potential sources of data loss and corruption when using non-volatile memory, while imposing minimal performance overhead. All three are designed across hardware-software boundaries, leveraging minimal hardware support: functionality already widely available but used sparingly, functionality previously proposed by other researchers, and new functionality that we propose in this paper:

Wear-aware memory allocation that is robust to erroneous writes. Hardware wear leveling [34, 35, 40] is necessary, but does not provide a complete solution. In particular, coping with frequent writes to the same or neighboring locations imposes overheads when writing (e.g., higher than 100% [35]), as the hardware constantly shifts the data to new locations. Maintaining maximal throughput therefore requires software to avoid these pathological patterns. Traditional memory allocators do not avoid these pitfalls. At the

Parameter	PCM	DRAM
Read Latency	50 ns	20-50 ns
Write Latency	150 ns	20-50 ns
Read Bandwidth	1 GB/s/die	2 GB/s/die
Write Bandwidth	200 MB/s/die	2 GB/s/die
Write cycles	10^8 cycles	∞

Table 1: Predicted PCM and DRAM characteristics [4, 6, 33, 32].

same time, corruption of memory allocator metadata can lead to permanent data loss and unrecoverable memory leaks. To address these problems, we introduce a new memory allocator that complements hardware wear leveling for NVRAM. Our allocator prevents the frequent reuse of memory locations, and stores its frequently-changing metadata in DRAM. It adds checksums to detect and recover from metadata corruption.

A low overhead mechanism for containing erroneous writes. With DRAM, the consequence of bugs or memory corruption is typically an application crash. With NVRAM, however, erroneous writes can cause permanent data loss. The larger the persistent memory area exposed to the application for direct access, the higher the risk of corruption through erroneous writes. Using virtual memory protection to contain this threat has been proposed previously in the context of databases [42] and reliable file system caches [11]. In this paper, we show how to implement a similar scheme in a way that avoids the overhead of system calls and mode switches. This scheme complements the protection provided by our robust memory allocator: VM protection prevents "long-range" erroneous writes to locations that clearly should not be written to; the robust allocator metadata protects against far-reaching consequences from "short-range" off-by-one errors or buffer overflows.

Cache-efficient, consistency-preserving updates. Making effective use of the CPU cache is critical for overall program speed. However, ensuring the consistency of persistent data in the face of application crashes and power failures requires careful control over the order of writes to persistent memory. Today's CPUs provide only coarse ways to achieve such control, e.g., by marking memory non-cacheable or by flushing the cache, but these methods impose high costs. We propose a novel mechanism for implementing consistency-preserving updates without sacrificing performance: make applications aware of where their data is at any time—in cache or in persistent memory. Our solution requires application support, as well as lightweight hardware modifications.

2 Background and Assumptions

This section briefly overviews non-volatile, byte addressable memories and enumerates the assumptions we make about future system support for these memories.

2.1 Non-Volatile RAM

Three emerging technologies hope to provide fast, persistent, byte addressable memories: phase change, memristor, and spin torque transfer. We collectively refer to these memories as non-volatile RAM (NVRAM). We focus primarily on phase-change memory, because it is now becoming commercially available and is better characterized, to understand how these memories differ from conventional DRAM.

Phase Change Memory (PCM) is a non-volatile, random-access memory that stores bits by heating a nanoscale piece of chalcogenide glass and allowing it to cool into either a crystalline (1) or an amorphous state (0), each of which has a different electrical resistance. Heating is performed by injecting current through the memory cell. Several of PCM's most important characteristics can be derived from this method of operation: programmable at byte granularity, fast reads but limited write bandwidth, wear-out.

Byte programmable: Each cell can be read and written independently.

Fast reads: Cells can be read electrically by measuring their resistance. PCM read speed is comparable to that of DRAM, and orders of magnitude faster than Flash memory. Read latency may be as low as 48 ns [6].

Limited write bandwidth: Because PCM is programmed by heating, its write bandwidth is limited by the amount of power that can be delivered to the chip. Write bandwidth per die may be only 200 MB/s/die versus read bandwidth of 1 GB/s per die [31]. These numbers are optimistic until the technology matures further: A recent 1-gigabit prototype from Samsung achieves only 6.4MB/s program bandwidth. Write latency may be as low as 150 ns [4, 6].

Wear-out: Heating causes physical expansion and contraction of the cell, which eventually leads to wear-out. Cells may be limited to as few as 10^8 writes [33].

High storage density: PCM's storage density may scale better than that of DRAM or Flash memory: PCM has been demonstrated at a 20 nm process size and is expected to scale to 8 nm, while realizing Flash and DRAM at this scale is difficult [2]. Table 1 presents the PCM parameters that we assume.

Two alternative technologies, *memristor-based memory* and *spin-torque transfer memories* are also competing to be the next persistent storage technology. These technologies are further from production, but may offer improvements in writing: memristor could have an order of magnitude better endurance, and spin-torque, which is further yet from being realized, may have unlimited endurance. Both would benefit from the memory protection and consistency provided by our toolbox.

Memristor-based memory is a "resistive RAM" (RRAM) device that has garnered recent attention. In these devices, applying an electric field changes the resistance of the device, which can be later read to indicate a 0 or 1. As with PCM, cells can be read and written independently. Early lab samples of memristor memory suggest that its performance could be comparable to the DRAM performance shown in Table 1, and its write endurance could be 10^{10} cycles. However, memristor is still in early development, with no commercial availability.

Spin-Torque Transfer Memories use magnetic properties of certain materials and electric charges to enable states of different resistance in memory cells. Their biggest advantage over PCM is the potential for unlimited write cycles. Recent 32-megabit chip prototypes using tunneling magnetoresistance showed read latencies of 32 ns and write latencies of 40 ns with a power consumption of 300 μA per cell [22]. Unfortunately, the outlook for shrinking cell sizes is extremely uncertain, with no reliable estimates for when competitive sizes will be reached.

2.2 NVRAM on the Memory Bus

The latency of today's external busses—PCI at a few hundred nanoseconds [19] and SATA and SAS even slower—dominates that of NVRAM. This contrasts with Flash memory, whose approximately 40-microsecond access time dominates bus latency. Similarly, accessing NVRAM through a file system or system call interface incurs mode switch and data copy overhead, potentially doubling the access latency.

Systems are unlikely to abandon DRAM: Because of the wear-out and slower writes of NVRAM, we assume that at least for write-intensive workloads, systems will instead use a combination of DRAM and NVRAM, harnessing the best properties of each.

As prior work did [13], we assume that NVRAM devices will provide atomic writes at some granularity—here, a cache line (64 B). Because CPUs are optimized to access DRAM at the granularity of a cache line, we believe that performance considerations will lead to the same choice for NVRAM.

2.3 Limitations of Hardware Wear Leveling

There are at least two reasons why software will be required to complement hardware solutions for avoiding NVRAM wear out.

First, hardware wear leveling can reduce performance. These solutions transparently remap the physical NVRAM locations and the logical addresses exposed to the memory controller [34, 35, 40]. To do so, they must copy content between physical locations, reducing write bandwidth and increasing overall wear. This overhead is small for well behaved workloads [34], but wear leveling must also handle high write traffic to one or a small set of locations [40]. Doing so requires data to be moved as often as once per every application write [35]. The newest adaptive schemes penalize only those applications that concentrate many writes to few locations [35]. This solution is effective for applications using NVRAM as volatile memory, because most writes with high locality can be absorbed by a large DRAM cache, but not for applications using NVRAM persistently, where NVRAM sits only behind a small CPU cache. In this case, applications that desire high performance must avoid bad write patterns.

Second, software should avoid unnecessary writes even with wear-leveling: each write reduces the memory lifetime. As we show in Section 8, existing malloc implementations, designed for DRAM, generate excessive writes when used for NVRAM.

Finally, hardware wear leveling may not be available for low cost applications of NVRAM—e.g., in sensor nodes and mobile devices, which are likely to be early users of the technology [37].

The range of applications that could benefit from high-performance, persistent memory is large. Sensor nodes and embedded platforms could benefit from the ability to execute code directly from persistent memory, and the ability to suspend and resume execution nearly instantly by only flushing caches to NVRAM. High-performance computing could snapshot rapidly to NVRAM. And as exemplified by substantial recent interest in developing database techniques to capitalize on flash memory, data-intensive applications and transaction processing system designs might be completely overhauled for byte-addressable, high-throughput persistent memory.

Making effective use of NVRAM, however, will require software support: operating system support to provide lower-level access to the memory, and library support to hide the complexities of using persistent main memory from application developers.

2.4 Operating System Support

We expect that existing **virtual memory mechanisms** will be used to map both volatile and non-volatile memory into the processes address space. Processes will be able to request NVRAM pages from the operating system, using an extension to the `mmap` system call. We assume that the basic unit for NVRAM in operating systems will be the 4 KB page, with continuing CPU and operating system support for large pages for efficiency.

To restore persistent application state, a process must be able to map the same NV memory pages across machine reboots. Therefore, operating systems must provide a way to identify groups of pages under a **persistent namespace**. There are multiple satisfactory proposals [12, 39, 47]. We assume that (1) it will be possible to map the same NVRAM pages in different, not necessarily concurrent processes; and (2) the OS will provide access control to regions of NVRAM, perhaps akin to access control in file systems.

We do not address the concern of **persistent memory pointers**. There exist various solutions: making mappings fixed [47], or using pointer swizzling [12, 48]. Another simple solution could be using the CPU support for segmentation (still present on x86 processors): map each logical group of NVRAM pages in its own segment and make every pointer relative to that segment.

2.5 Application Library Support

Providing high-performance persistent storage on the memory bus introduces several challenges, alluded to in earlier sections. Sections 3–5 describe three lower-level techniques which, together with the OS support described above, enable applications to use NVRAM effectively. Using these techniques directly, however, has the potential to burden application developers with significant complexity. Rather, we propose "hiding" these mechanisms behind user-space library—software building blocks—which provide persistent main memory data structures and algorithms to higher-level applications.

3 Memory Allocation for NVRAM

The NVMalloc allocator helps address two of our three challenges for using NVRAM: preventing wear-out, by minimizing writes and helping with wear-leveling, and increasing the allocator's robustness to erroneous writes.

3.1 Wear-Aware Memory Allocation

Not only are today's memory allocators not optimized for NVRAM, they may actually contribute to wear-out or performance degradation when used for NVRAM. Consider as a representative example GNU malloc version 2.12.1. Malloc and several other allocators [49] cache and reuse small allocations ("blocks") in LIFO order after they are freed. This concentrates writes in a few locations, which we verify experimentally in Section 8.1.1. Malloc furthermore maintains metadata in headers and footers of each allocated (or freed) memory block, containing the size of that block. This metadata is updated for every merge and split of free blocks, which can happen for any allocation or deallocation that is not small enough to be cached. As a result, simple patterns such as a sequence of allocations followed by deallocations in the same order can cause multiple writes to the first header and the last footer of a series of contiguous regions. Finally, these in-place metadata updates, while memory efficient, cause additional writes that increase NVRAM wear-out; we show in Section 8.1.1 that malloc may cause 50% more writes than our allocator. Note that these malloc design decisions make sense for DRAM, providing memory efficiency and performance, but they have unintended results when applied to NVRAM.

NVMalloc is a robust, wear-aware memory allocator that avoids these problems. NVMalloc is based on two ideas: (1) limit the frequency with which any particular block of memory can be allocated, and (2) maintain frequently changing metadata in DRAM, separate from the managed blocks of non-volatile memory.

Allocator wear-leveling. NVMalloc will not reallocate a block as soon as it is freed. Instead, it timestamps the block and adds it to a FIFO queue of recently freed blocks—the *don't-allocate list*. On every allocation or release, the allocator examines the block at the head of the queue; if it has been in the queue for at least time T, it removes the block from the queue and marks it eligible for reallocation. Blocks will not be allocated with a frequency higher than $1/T$. Assuming that the sizes of allocation requests are smaller than the available amount of non-volatile memory in the system, we can improve wear leveling by increasing T. To avoid using extra space, the don't-allocate list is implemented as a simple linked list, with the pointers and the timestamps stored inside the free memory blocks themselves.

Reducing allocator metadata writes. The first and easiest step that NVMalloc takes to reduce writes is using a minimum allocation size of 64 bytes—the minimum NVRAM write size. Smaller blocks cause write amplification because writes would perturb the other bits in the write unit.

The second, more complex optimization reduces writes due to managing free space. Allocators reduce address space fragmentation by satisfying allocations from appropriately sized individual free blocks. For example, to allocate a 256 byte block, the allocator would only allocate from a new 4KB page if it could not find a 256 byte "hole" in otherwise allocated regions. To accomplish this, NVMalloc, like malloc and its predecessor dlmalloc [25], uses *segregated free lists*—a free list for different size free blocks, with a common list for large blocks.

NVMalloc reduces the number of writes to persistent memory by observing that tracking the information needed to minimize fragmentation is an optimization, not something required for correctness. NVMalloc persistently stores the correct allocated/free state of every block, but relegates fragmentation information to DRAM, and can rebuild this information if needed after a crash.

Traditional allocators store list pointers inside the headers and footers of freed blocks. When they merge adjacent free blocks into a larger free region, they update those in-place pointers. NVMalloc instead tracks the allocated/free state of each basic memory block using a DRAM bitmap. (Figure 1). To persist allocation state across reboots, each memory block has a header with its size and allocation state. When mapping a new NVRAM region, the allocator can rebuild the bitmap by scanning these headers plus the free lists described below. Unlike malloc, however, NVMalloc does not update the headers when merging free blocks, and no longer requires footers.

NVMalloc maintains its segregated free lists in DRAM. These lists may become inconsistent with the real allocation state, potentially requiring a few extra reads during allocation, but substantially reducing the number of writes. Upon freeing a block, the allocator examines the bitmap to find the maximal free region that includes that block, updates the bitmap, and adds an entry to the list corresponding to the region's size. A new allocation request is satisfied by searching through the free list with blocks of the corresponding size for the first entry that is still consistent with the information in the bitmap. If none is found, a larger free block will be split, or the allocator will request more NVRAM pages from the operating system.

NVMalloc only updates memory block headers when the block is the start of a memory region that is being allocated or when the block is freed by the application. The pointer and timestamp required for adding the block to the don't-allocate list are added in the same cache-line write as the header update (they are part of the header). As a result, NVMalloc guarantees that it will write each location at most twice per T seconds.

3.2 Robust Memory Allocation

Bugs such as off-by-one errors, uninitialized pointers, buffer overflows or buffer underflows can corrupt memory allocator metadata. With non-volatile memory, corrupted memory allocator metadata can cause permanent data loss or unrecoverable memory leaks. Consider for example a buffer underflow error that overwrites the header of an allocated memory block. When the buffer is freed, the allocator may reclaim more memory than it should, and a subsequent allocation will result in persistent application data and other block headers being overwritten, etc.

Previous approaches protect allocator metadata by storing it separately from the space allocated for use [26, 47]. While less likely, metadata corruptions could still occur (e.g., because of an uninitialized pointer that uses the data of an old stack frame corresponding to a memory allocator call). Because the consequences of corruption are larger for NVRAM, NVMalloc uses stronger techniques.

First, NVMalloc's design is robust to errors in its DRAM-based data structures: they only optimize allocation, their information is checked against the in-NVRAM metadata at allocation time, and they can be rebuilt if needed. Second, every NVMalloc block header contains a checksum over the size of the block,

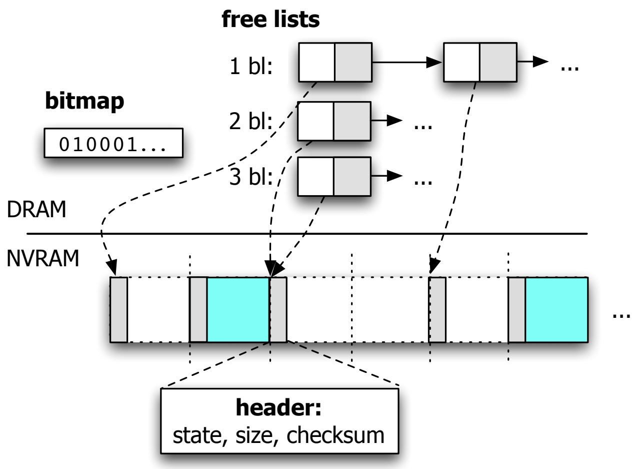


Figure 1: Memory allocator metadata example. Two of the total six basic memory blocks depicted in the diagram are allocated.

its state (allocated / free), and the position of the block relative to the beginning of the current NVRAM mapping. The latter helps detect accidental copying of whole blocks, headers included, over other blocks. The allocator can thus detect incorrect headers when allocating or freeing a block, or when scanning the block headers of a newly mapped NVRAM region. The allocator then isolates the corruption to the one or few affected blocks by scanning forward through every subsequent basic memory block (64 bytes) until it finds a new sequence of correct headers. It marks the corrupt header and notifies the application of the corruption. A side benefit of this scheme is that the allocator can also detect some erroneous writes to persistent *data*, something that cannot be achieved by simply separating the metadata from the data.

NVMalloc provides *containment*, not *prevention*: It limits the extent of data loss to the directly modified memory locations. The next section presents a safety mechanism to further reduce the possibility of data corruption. Our memory allocation optimizations do not prevent memory leaks; garbage collection to ameliorate this problem are complementary to NVMalloc's techniques.

4 Low-Overhead VM Protection

For speed, applications are likely to map the NVRAM storage (potentially tens or hundreds of gigabytes) into their process address space. Doing so, however, eliminates many traditional safety mechanisms that filesystems and databases offer. Erroneous writes by the process can corrupt not only data, but also metadata, where errors can potentially cause massive data loss or crashes. The interface through which errors can affect persistently stored data is now much wider: instead of writing to a buffer and invoking an explicit system call to persist that buffer, any stray memory store(s) could corrupt persistent data [1].

Concerns about the vulnerability of persistent data mapped into process address spaces date back decades, in the context of databases [42, 14] and reliable file system caches on battery-backed DRAM [11]. Today, OS-mediated, CPU-enforced memory protection is widely used to protect parts of the address space from corruption in JVMs [20], databases [38], and garbage collection [9]. Designers of persistent

[1]The decreasing cost of switching to kernel mode [44] makes it attractive to map persistent data in user space as read-only, and then make a syscall whenever the application needs to modify it (since the data would be mapped read-write in kernel space). However, this would make application data vulnerable to stray memory accesses by device driver code. Given the large variety of third-party device drivers, we believe this would be even more difficult to control than stray accesses by the application itself.

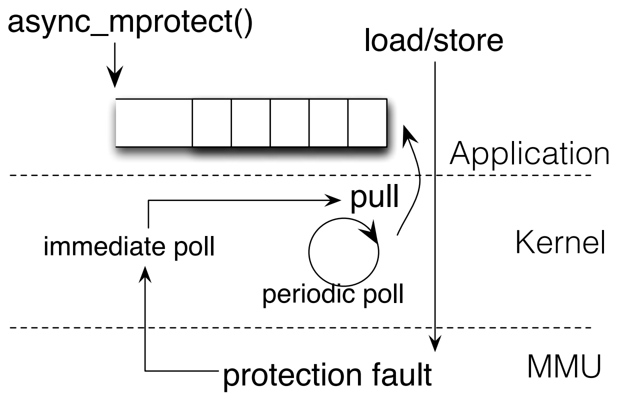


Figure 2: High-level operation of batched asynchronous memory protection.

data structures for NVRAM can follow this lead, but to enable them to do so, we must make this capability low-overhead enough to use in the context of hyper-fast persistent storage.

The simplest way to write-protect virtual memory on a POSIX-compliant system is to call `mprotect` after writing to a page. Subsequent (erroneous) writes to this page will then trigger an exception. Applications might then repair the error transparently, gracefully exit and restart, and/or simply generate a bug report noting the software bug or faulty hardware component. Unfortunately, frequent virtual memory protection changes have traditionally had a very high overhead: a page protection change involves a system call (so therefore a mode switch), acquiring locks on the page table of the process, and finally, a TLB invalidation which slows down all threads sharing the address space, not just the one issuing the system call. As a result, performance sensitive applications avoid using memory protection—e.g., database software vendors recommend turning virtual memory protection off when performance is important [46].

We improve upon the mprotect approach by (1) avoiding the system call mode switch; (2) by servicing page protection requests in batches to amortize some of their costs; and (3) coalescing redundant requests such as protection and unprotecting the same page.

Figure 2 depicts the high-level organization of our approach.

To protect pages, the kernel and process share a memory buffer used as a lockless producer-consumer queue [24]. The process inserts the addresses of the non-volatile memory pages that require protection against writes, and a kernel thread periodically removes these requests from the queue to protect them. The kernel thread sorts the requests and performs protection changes on ranges of pages. This approach confers two advantages:

- Changing the protection on ranges of pages and consolidating multiple requests to the same pages amortizes the cost of modifying the structures associated with the address space of the process (an expensive operation in Linux, as detailed in the next section);
- Batch processing requests means that the kernel only flushes the TLB once per batch, instead of once per protection operation, reducing slowdown imposed by frequent TLB invalidations.

This approach substantially reduces the overhead of protecting pages from writes [2]. Batched asynchronous protection does not hinder the normal operation of an application because write-protecting a page

[2]Some potential applications require a notification queue to tell the process when the protection request was complete. This is a simple extension, but we have not yet implemented it. Applications can, however, wait for all the pending requests to be processed.

does not need to happen immediately (at least not in the common case, where erroneous writes are the exception, not the rule).

Un-protecting pages that the application needs to write can follow one of two different paths, depending on the semantics that the application requires:

1) Applications, such as garbage collectors, that can predict beforehand where they will write can issue asynchronous unprotect requests early.[3] The same approach used for protecting is used to un-protect pages before the application writes: the application puts a request in the same queue used for protect requests. When the kernel decides (asynchronously) to process a batch of requests, it first sorts the requests using a stable sorting algorithm—so that relative ordering of protect and unprotect requests for the same page is preserved. Finally, the kernel coalesces different requests for the same page and performs the most recent protection change for each page in the queue.

Most applications, however, cannot predict far enough in advance which memory regions they will update. For those applications we use the second approach.

2) Applications that cannot foresee which page they will have to update next, use a *separate* unprotect queue where they write the page address/addresses of the memory object they are about to modify. They then access those pages, optimistically assuming that the page is already writable (or has become writable because of a previous asynchronous request). If the page is indeed writable, the access succeeds. If not, and the page is still protected (unwritable), the MMU generates a page fault exception.

The kernel exception handler checks the special unprotect queue, finds the unprotect request and changes the protection of the page, transparently to the application. Putting the exception handler in the kernel saves additional mode switches. This mechanism works well for applications with some access locality: a protect followed soon by an unprotect will result in no protection changes or page faults. For erroneous writes, the kernel exception handler will not find any unprotect requests, and will send the appropriate signal to the application generating that write.

5 Cache-Efficient Updates to Persistent Main Memory

Using CPU caches effectively has been the primary way of bridging the gap between the speed of the CPU and that of the memory. Making an application run faster often comes down to improving its use of the caches. We expect the CPU caches to continue to play a critical role for main memory, volatile and non-volatile alike[4]. Unfortunately, maintaining consistency in the face of crashes or power failures becomes more difficult when accessing persistent main memory through CPU caches that themselves control the order of writes to memory.

Continuously flushing the cache lines introduces high overhead (e.g., more than $4\times$ —see Section 8.3). On the other hand, bypassing the caches altogether is efficient only for those applications that write only once to a location and do not read after write (as counterexamples, consider frequent updates to a global counter—e.g., the global version of a multi-versioned data structure—or inserting ranges of keys into a B tree), and presents additional concerns with respect to the wear-out and write bandwidth limitations of NV memories. Furthermore, modern processors and memory controllers have been highly optimized for

[3]After garbage collection, memory does not have to be reallocated immediately. Therefore, if the recovered memory blocks are still write-protected, we can issue unprotect requests asynchronously and only consider those blocks for reallocation after the requests have been serviced.

[4]CPU caches will perhaps be even more important for non-volatile memory, since non-volatile memory is expected to be even slower than its volatile counterpart.

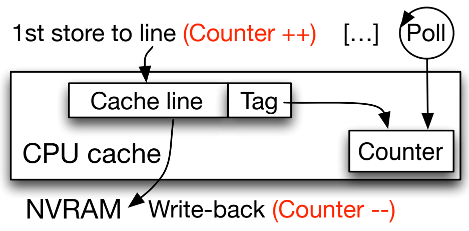


Figure 3: High-level functionality of cache line counters.

working with the CPU caches. We would like to *use the CPU caches as normal, without flushing, but still guarantee update consistency*.

The solution that we propose follows logically from our stated goal: make applications aware of the state of their writes in the cache so that they can make the appropriate adjustments in software. For this, we require hardware support. *Given a logical group of updates, an application can check whether all the updates belonging to that group have been written out to memory*—as a result of normal cache line replacement—or if some are still in the caches. Applications can use this capability in many implementations of failure atomicity: when performing shadowing, the software application will replace the original pointer with a pointer to the shadow copy after all the updates to the shadow have reached memory; when working with multi-versioned data structures, the application will increment the global version number only after the new version has been completely flushed out of the CPU caches; when updating a log, it will increment the log size only after the new record has entirely reached persistent memory, etc.

5.1 Cache Line Counters

To allow applications to check whether their writes have been flushed from cache to NVRAM, we propose light-weight hardware changes: The CPU caches are augmented with a set of counters, where each counter keeps track of how many cache lines dirtied during one atomic logical update have yet to be written back to NVRAM. An application marks the beginning of a logical update group by calling a special `sgroup` instruction, which tells the CPU to (1) choose a free counter, and (2) increment that counter whenever subsequent store instructions dirty new cache lines (an `sgroup` instruction must therefore also order stores to memory, just like the x86 `sfence` instruction). The high-level functionality of cache line counters is illustrated in Figure 3.

A counter is incremented automatically when a store dirties a cache line, and decremented when a cache line tagged with its ID is cleaned (either by normal write-back during cache line replacement, or as a result of a `clflush` call). A store to a cache line tagged with the ID of a counter other than the current counter will not modify any counter values—this behavior is consistent with the common situations where groups of updates must be ordered (i.e., a group is only committed after all previous groups have been committed). Minor operating system modifications are required to save and restore the register containing the current counter ID when a thread is preempted and re-scheduled for execution, respectively. A subsequent `sgroup` call ends the current update group.

Applications retrieve the counter ID from a CPU register and store it for later use with the `scheck` instruction, to poll the value of the counter. When `scheck` returns a counter value of zero, the corresponding logical update group can be considered *safe* in NVRAM, and the counter is internally marked as free.

In conclusion, the interface to the cache line counter functionality consists of two new instructions (`sgroup` and `scheck`), and a new CPU register to read/write the counter ID for the current logical update group.

Unlike previous proposals for hardware support for NVRAM (the BPFS epoch barriers mechanism

[13]), the modifications that we propose do not necessitate changes to the cache line replacement algorithm. This is important because forcing cache line flushes goes against the trend of enhancing the behavior of the CPU cache hierarchy with complex heuristics. Section 9 contains a more detailed comparison with BPFS.

One important implementation decision is how many counters to use. At the extreme, each cache line could be associated with a different counter, which would require as many counters as there are cache lines in the CPU caches. Die space limitations could make so many counters prohibitive. Provisioning fewer counters risks running out of counters if many applications perform many small atomic groups of updates. To solve this problem, we add a special virtual counter (we will call it `C0`) whose value is always zero. When there are no more free counters, the CPU will use `C0`. A store when `C0` is the current counter will behave like a non-temporal write that bypasses the CPU caches. Running out of counters may affect performance, but not correctness.[5]

5.2 Implementing Cache Line Counters

Augmenting CPU caches has previously been proposed for improving cache performance [23], implementing transactional memory [5, 29, 36], and even for ensuring update consistency in persistent main memory [13] (i.e., the same problem we are addressing). While we have not implemented in hardware the cache line counter mechanism, we describe here one possible implementation path. Because the counters keep track of dirty cache lines, the implementation consists mainly of a straightforward bookkeeping mechanism and a few additions to the cache coherence protocol.

Cache line tags are extended with space for a *counter ID*. Dirtying a cache line causes the CPU to increment the current counter and annotate the cache line with its ID. Cleaning a line decrements the counter corresponding to the ID stored in the cache line's tag (the decrement is done only after the write-back is acknowledged).

For processors with inclusive shared last level caches, like the modern Intel Core i7 CPU, it is sufficient for counters to track only the lines in the last level of the cache. This design has the advantages of avoiding the high churn in smaller caches, the counter namespace is global, and the counters are only stored in the larger cache. Reading counter values will have higher latency than if counters were stored in the smaller caches, but since applications using counters are likely to be data intensive, this latency would be dwarfed by the latencies of frequent memory accesses.

On a CPU where the last level caches are not inclusive, each core maintains a separate set of counters in its L1 cache, in a direct-mapped structure. Cache tags in shared caches store both a counter ID and a core ID. If the line cleaning/dirtying happens in a level of cache other than L1, the counter decrement/increment message must be sent up to the core that owns the counter associated with that line.[6] When an application reads the value of a counter maintained by a core other than the one on which it is running, that counter value is brought in through the cache subsystem, just like normal memory content—the counters are memory mapped read-only.

As a concrete example of a possible instantiation of cache line counters, consider a CPU like the Core

[5]This scheme has the drawback that one application might intentionally or unintentionally monopolize all the counters. Note, however, that the same is true for the CPU caches themselves.

[6]A special case for processors with non-inclusive caches occurs when a cache line is pulled into the private cache of a core (e.g., core B) other than the one that owns the counter associated with that line (e.g., core A). In this case, the line is cleaned from all the caches accessible to core A, and sent clean to core B. Thus, core A no longer keeps track of that line. Although flushing the cache line from the first core may cause overheads for applications with this access pattern, this is not a new problem, as applications already try to avoid expensive cache line "ping-pong-ing." If, however, this occurs as a result of a thread moving to a different core, the operating system notifies the application through a signal so that it can track a new counter on the new core in addition the old counter on the previous core until those lines are cleared. This mechanism introduces some awkwardness for programmers working with counters, but we don't expect this situation to be common since operating system schedulers try to maintain core affinity (and applications can even ask that this be enforced).

i7: a quad core CPU with 32KB private L1-D and L1-I caches, 256 KB private L2, and an 8 MB inclusive L3 cache. Such a CPU could provide 8192 one byte wide counters.[7] Each counter can count up to 255 cache lines, but special counters can combine the space of two or more normal counters to be able to count up to the total number of lines. When a counter reaches its limit, the CPU can either automatically upgrade it, or treat subsequent stores as non-cacheable. For 8192 counters, the cache line tags are extended by 13 bits. Overall, this amounts to a 216 KB space overhead (2.28%), incurred exclusively in the L3 cache.

6 A B+ Tree Example

This section describes how the proposed techniques can be used to implement a modified B+ tree data structure, including the algorithmic changes that are required. To hide their complexity, we expect that data structures such as this would be implemented in a "persistent storage" library for NVRAM, not necessarily by individual application developers. We have applied these changes to an existing main-memory B+ tree [8].

Using NVMalloc. NVMalloc can be used as any other memory allocator, requiring no program modifications to function correctly. However, applications can take advantage of NVMalloc's wear leveling properties by making small adjustments. In our B+ tree, for example, whenever a node requires splitting, we allocate two new nodes and free the original—otherwise the same memory location risks getting heavy write traffic (see the experimental results in section 8.1.1). This simple modification is also necessary for guaranteeing consistency with cache line counters, as explained below.

Using cache line counters. First, we define what consistency properties we want for our B+ tree: (1) after a crash or power failure, we want the B+ tree to be consistent (i.e., no dangling pointers, no partially written data), and (2) the B+ tree must be monotonic—that is, if a power failure occurred at time t_1, the recovered data would be a subset of the data that would be recovered if the failure occurred at time $t_2 > t_1$. This condition means the B tree behaves intuitively: once data becomes "persistent" (i.e., recoverable), it cannot be lost. To improve performance (achieve a higher insert throughput), we do not require data to be made persistent right away. These consistency guarantees are reminiscent of soft updates for file systems [17].

We start by focusing on inserts. The changes that we make to the B+ tree are to ensure consistent updates without having to flush CPU caches.

Change 1: Inserting at the end of nodes instead of keeping nodes internally sorted. In a typical B+ tree, new pointers to nodes or values are inserted into the node in sorted order, which may require shifting some of the existing entries. Because some entries will cross cache line boundaries, even if the algorithm is correct—it writes an entry in its new slot before overwriting the old slot—the unpredictability of cache line write-back can still cause data loss on a power failure if the update to the old slot of the entry reaches NVRAM before the update to the entry's new slot. Therefore, we append entries in a log fashion, at the end of each node's list of children pointers. A consequence of this change is that searching within a node becomes slower (linear instead of logarithmic time for a binary search), but this represents only a small penalty because nodes have few entries, and can be mitigated—see Section 8.3.

Change 2: Using cache line counters to distinguish between data that has been written completely to NVRAM and data that is not yet entirely durable. A counter tracks the cache lines dirtied by the data being written (i.e., the new inserted value). After the counter reaches zero, we mark the data as *valid* by writing the corresponding key in the header of the data portion (using a normal cached write)—this will make it easy to distinguish the valid data entries from the invalid ones after a power failure [8].

[7]For 1 KB average update group sizes, this many counters would ensure that running out of counters is rare.

[8]No change is needed for validating the B+ tree metadata after recovering from a crash or failure, as we identify valid node

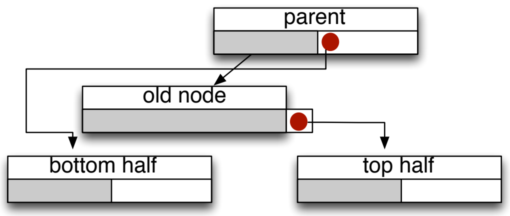


Figure 4: Transient B+-tree state after a node split. The dots mark the only pointers updated at this stage.

Change 3: Ensuring that the B+ tree is always consistent in NVRAM during node splits. Figure 4 depicts the modified node-splitting process. A node is not immediately deleted after being split. Its parent maintains its pointer to it, and the node itself is augmented with a pointer to the new node that contains the top half of its keys (the ones larger than the median key). The bottom half is inserted in the parent node as usual (at the end). All subsequent inserts and searches will be performed on the new nodes. After the old keys and pointers become persistent in the new nodes, and after the pointer to the bottom half becomes persistent in the parent node (we keep track of all this using one cache line counter), the pointer to the new top half node replaces the pointer to the old node in the parent [9]. To prevent chains of pointers, if the top half is split while in the transient state, we simply replace the pointer to it with a pointer to its top half (i.e., the top half of the top half) in the old node. In summary, by keeping pointers to old versions until the new versions become persistent, we ensure we do not lose data.

For deletes, we simply invalidate the pointer (we make it `NULL`) to a node that is to be removed, and allow subsequent inserts to overwrite it. Moving entries from one leaf node to a neighbor and merging nodes can be handled in one of two ways: (1) lazily, when the insert load is low, by forcing cache flushes, or (2) using cache line counters: copying entries to their new node and invalidating the old one but without removing it until the entries become persistent.

Using cache line counters in implementing our B+ tree algorithm required only small modifications to the program structure. We use a circular buffer to store "counter-action" pairs. Along with every B+ tree operation, we also check the value of the counter at the head of the buffer, and if 0, we perform the action, which consists in updating a pointer, or writing a key to the header of a data record. If the counter demand is high, we can check (and thus free) multiple counters in each round.

Using asynchronous memory protection required no modifications to the B+ tree algorithm. We simply write the page address (or addresses) of the node (or data region) that we are about to update into the unprotect buffer, and then into the protect buffer once the update is complete.

7 Implementation

7.1 NVMalloc

NVMalloc is implemented in 735 lines of C, and is drop-in compatible with GNUMalloc.

entries by checking that the key information in the parent matches that in the node.

[9] We do not need to force this pointer update to NVRAM. Because we assume cache line-sized atomic writes, a pointer will either point to the new or the old version of a node. Either way, the B+ tree will be consistent by our definition. The only provision that we have to make is to not free the old node until the pointer update has been persisted, and we can do that either with cache line counters or by relying on the delay with which our allocator reclaims space.

Allocation granularity. NVMalloc maintains a separate free list for each allocation size that is a multiple of 64 B (one cache line, the minimum allocation size), up to 4 KB; recall that these lists are hints pointing to free regions of NVRAM of that size. Hints for free regions larger than 4 KB are placed on a common list. NVMalloc does not handle large allocations as a special case, although extending it to do so is straightforward and would reduce metadata (bitmap) overhead.

Header checksum. NVMalloc uses the CRC instruction, if available, to checksum the header. Otherwise, it XORs the block size and position into a single byte.

Failures. To ensure that allocated memory is not lost because of a power failure, a program can either (a) request that the allocator write the header of the allocated region to NVRAM before it returns, or (b) with cache line counters, simply call `sgroup` before making the allocation request instead of after.

7.2 Asynchronous Memory Protection

We implemented the asynchronous memory protection mechanism described in the previous section as a 625 line kernel module for Linux kernel version 2.6.37. Additionally, we had to make small modifications to six kernel source files (mostly for accessing functionality already in the kernel). Applications do not access the request queues directly. Instead, they use a small user-space library as the interface with the in-kernel functionality.

Batching. To control the size of request batches, we set the maximum amount of time that the kernel thread that processes requests can sleep between two subsequent batches. The *actual* sleep time is dynamically set to be inversely proportional to the number of new requests enqueue while a batch was being processed.

Besides coalescing requests and amortizing the cost of TLB invalidations, another benefit derived from batching is the ability to perform protection changes on contiguous ranges of pages instead of on individual pages. In the Linux kernel, each contiguous range of pages that have the same access protection is associated with a `vm_area_struct` structure. `vm_area_struct`'s are chained in a doubly linked list, and they are also inserted into a red-black tree to ensure quick access to a structure given an address. As a consequence, changing the protection of a single page entails splitting one `vm_area_struct` into as many as three new structs, and then performing relatively expensive delete and insert operations in a red-black tree. Furthermore, this is needless effort in those situations when the protection of neighboring pages would subsequently be changed anyway.

Although our implementation targeted the Linux kernel, we believe that batching memory protection change requests would benefit any operating system: any synchronous implementation of mprotect will have to deal with the cost of TLB invalidations and will miss the chance to coalesce multiple requests for the same page.

Lock contention. In Linux, a single lock controls access to the `mm_struct` of a process (the root data structure with information about the virtual address space of a process). We observed large performance penalties caused by frequent contention for this lock: our kernel thread acquires the lock to perform page protection changes, and kernel exception handlers acquire the lock to fault-in new pages. To mitigate this slowdown, our module takes this lock only once for all the requests in a batch. Particularly performance-conscious applications can eliminate nearly all of this overhead by requesting memory from the operating system in large blocks and then touching most of the pages in advance to fault them in and prevent the cache line ping-ponging associated with different threads running on different cores contending for a lock [10].

7.3 Simulating Cache Line Counters

To understand the benefits and overheads of cache line counters, we simulate their effects in software:

Reading the cache-line counters: We simulate this in the B+-tree implementation by maintaining a 8,192 entry table of pointers to B+ tree nodes and values (the "virtual" counters) that the tree code reads and updates along with every insert operation. This corresponds to the application checking the value of the earliest-allocated still-active counter and performing the associated action (i.e., copying a pointer or a key) if the counter value is 0.

Counter allocation overhead (the `sgroup` instruction) is emulated using `sfence`, because the major overhead of `sgroup` is that it implies a barrier for stores.

We do not model the event of running out of counters because applications that access gigabytes of data per second cause cache line eviction to happen very often, and for large groups of updates (on the order of kilobytes) the provisioned 8 K counters (see Section 5.2) are sufficient, while for smaller inserts it is easy to save counters by associating multiple consecutive inserts to one counter. Even if the counters are exhausted, the fall-back behavior (non-cacheable writes) ensures correctness, as explained in Section 5.

We do not model the counter logic overhead; we believe that counter operations are simple enough that efficient implementations can hide the overhead.

8 Evaluation

We evaluate the techniques proposed in this paper on a desktop machine with an Intel Core i7 860 2.8 GHz CPU, 8 GB of DRAM memory, running Ubuntu Linux, kernel version 2.6.37, and glibc version 2.13. Since NVRAM devices for the memory bus are not yet available, we use DRAM as a proxy (Section 8.4 discusses how our results would change with NVRAM). For each performance test we report the mean (and standard error, if significant) of one hundred runs. All the applications tested run single threaded, except for the kernel thread that performs async memory protection.

8.1 NVMalloc

This section evaluates NVMalloc's wear leveling, fragmentation, and the overhead that it imposes on memory intensive applications.

8.1.1 Wear Leveling

First, we evaluate the ability of our allocator to avoid NVRAM wear-out using a simple test program that performs 100,000 random memory allocation and deallocation operations (50% allocations and 50% deallocations). The sizes are uniformly-distributed between 10 bytes and 4 kilobytes. For every allocation, the program writes the entire allocated block exactly once. We instrument the program using Pin [27] to record stores to the allocated memory, both when using glibc malloc and when using NVMalloc. We record the number of accesses to each 64 byte block of memory (the size of a cache line), coalescing consecutive stores to a single block (Figure 5). These stores approximate writes to NVRAM.

NVMalloc distributes writes much more evenly than malloc[11]. Increasing the time blocks spend on the don't-allocate list spreads the writes further. The current implementation sets this value to roughly 100 ms

[10]A more general long term solution would be to replace the single lock on the `mm_struct` with finer grained locks on `vm_area_struct`'s.

[11]We use unqualified "malloc" to refer to the glibc implementation.

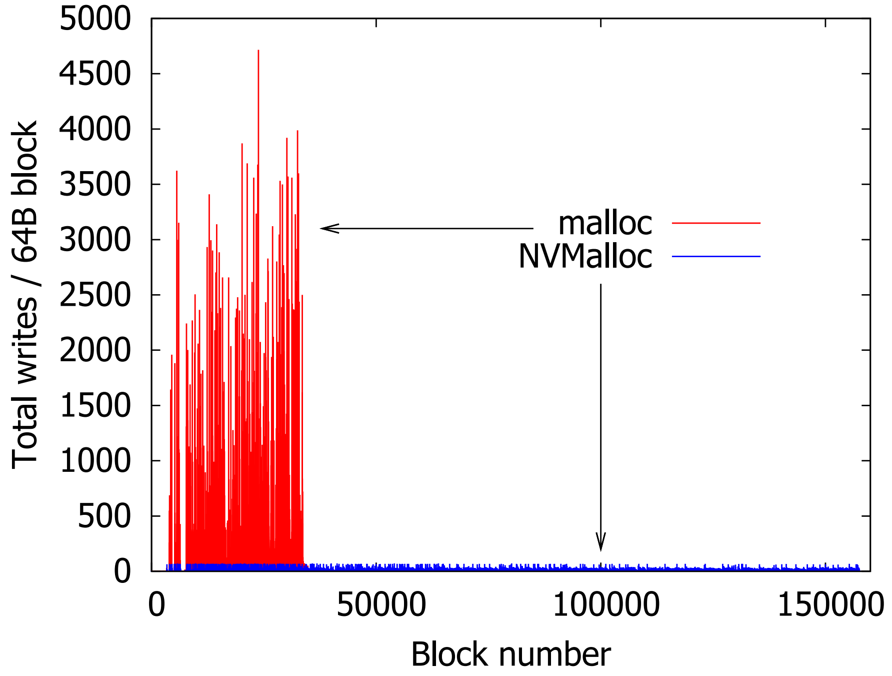


Figure 5: Writes per 64 B block for malloc and NVMalloc under the random alloc/free test (50K allocations and 50K random deallocations with 10 B to 4 KB uniformly distributed sizes).

of real time; when running under Pin, we approximate this target range by waiting 2 seconds of real time to compensate for the overhead of using binary instrumentation.

Without hardware wear leveling, NVMalloc's improved distribution is critical for avoiding wear-out. With hardware wear leveling, malloc's concentrated writes may require more internal remapping in the NVRAM device, degrading throughput and increasing overall wear as explained in Section 3.1.

Second, we evaluate the total number of writes. Malloc performs 1.3 times more writes to NVRAM than NVMalloc (2.39 million versus 1.83 million), because malloc maintains frequently changing metadata inside free blocks. With smaller allocation sizes (10 bytes to 256 bytes), malloc performs 1.5 times more writes. Even if perfect hardware wear leveling was possible, NVMalloc would increase NVRAM lifetime by 30% to 50% for this microbenchmark.

We repeat the wear leveling test for our modified B+ tree. Figure 6 presents the cumulative distribution of writes per cache line-sized block of memory in the metadata portion of the B+ tree (inner nodes and leaf nodes) for one million insert operations [12].

Finally, we run the same experiment for Memcached [28] version 1.4.7, for a workload composed of 60K inserts and 40K random deletes—10 B keys and 256 B values. We compare NVMalloc, glibc malloc, and two other popular allocators: jemalloc [16] and Hoard [7]. The results are presented in Figure 7. NVMalloc causes orders of magnitude fewer writes to the most access lines than the other allocators. Overall, glibc malloc causes 8% more writes than NVMalloc, jemalloc 5%, and Hoard 25%.

[12]This is a realistic scenario, since most memory allocators perform large allocations in a separate part of the address space than small allocations (for speed and fragmentation considerations).

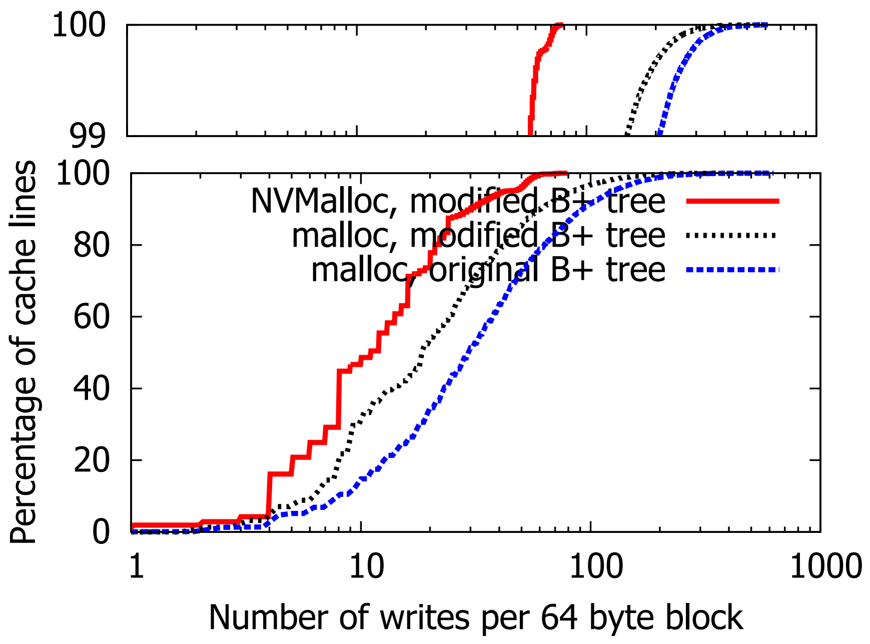


Figure 6: CDF of writes per block in B+ tree metadata, for 10^6 inserts (8 B keys). *Original* is the classic B+ tree implementation, *modified* uses NVMalloc and cache line counters. The top graph expands the 99%-100% interval. The X axis is log-scale.

Free:Alloc Ratio	malloc frag. total (external)	NVMalloc frag. total (external)	Slow-down
1/3	1.10% (0.33%)	2.14% (0.26%)	1.10
1/2	1.30% (0.52%)	2.29% (0.41%)	1.12
1/1	14.34% (13.61%)	12.76% (11.11%)	1.10

Table 2: Fragmentation and slowdown for 10^6 operations.

8.1.2 Fragmentation and Overhead

To compare the address space fragmentation of NVMalloc and malloc, we perform one million allocations and deallocations (in various ratios) of random, uniformly distributed sizes between 10 B and 4 KB, recording the total and external fragmentation (see Table 2)[13]. The results confirm that NVMalloc produces equivalent or better fragmentation compared with glibc malloc.

Table 2 also shows the slowdown that NVMalloc imposes relative to malloc. Malloc is faster for several reasons: most importantly, NVMalloc cannot cache allocations because this would contravene our wear leveling goals.

Our benchmarking tool, which essentially performs only memory allocations and deallocations, is not the best yardstick for measuring allocator overhead; therefore, we also perform tests on a real world data-intensive application: Memcached.

For a more realistic assessment of overhead, Table 3 compares Memcached using (a) its own slab allocator, (b) malloc, and (c) NVMalloc. The experiment sends put and delete requests in a loop embedded within the Memcached code to avoid measuring RPC overhead. There are two sets of experiments: (1)

[13]For a fair comparison, we set the time that free blocks spend in the don't-allocate list to zero—when set to non-null values, the space consumption grew by exactly the rate of deallocations multiplied by this time setting (we tested for 10 ms and 50 ms and observed the expected results).

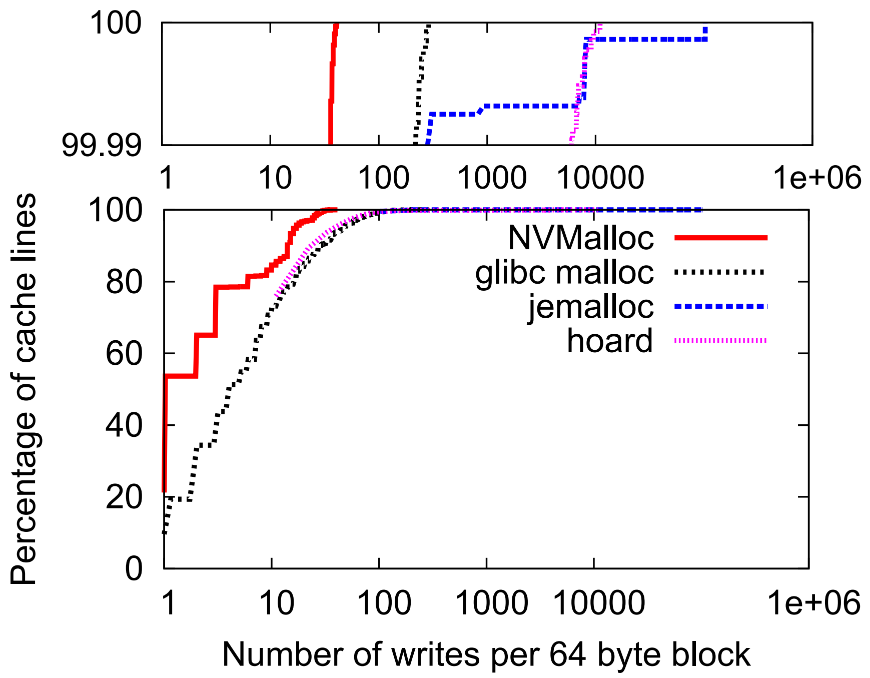


Figure 7: CDF of writes per block in Memcached (60K puts, 40K deletes, 10 B keys with 256 B values). The top graph expands the 99.99%-100% interval. The X axis is log-scale.

Allocator	Allocation size	Avg. time (std. err.) [ms]
Memcached slab	10 B - 4 KB	1768 (5)
malloc	10 B - 4 KB	1956 (9)
NVMalloc	10 B - 4 KB	1883 (5)
Memcached slab	1 KB	1149 (4)
malloc	1 KB	1278 (7)
NVMalloc	1 KB	1289 (5)

Table 3: Time for 10^6 Memcached operations (80% puts, 20% deletes).

requests of random, uniformly distributed value sizes between 10 B and 4 KB, and (2) fixed 1 KB values (10 B keys in both cases). Setting the don't-allocate time to one second (from zero) influences results by at most 1.5%. The testing overhead is 1.4%. NVMalloc causes little overhead for memory intensive applications.

8.2 Asynchronous Memory Protection

We measure the performance benefits of using asynchronous memory protection when compared to synchronous mprotect.

Figure 8 shows the time it took to perform one million random insert operations in the in-memory B+ tree and Memcached version 1.4.7 using asynchronous memory protection, no memory protection, and synchronous memory protection respectively. In the asynchronous case, we make an unprotect request right before we start an update to a memory object (B+ tree node or value slot, or Memcached value item), and make a protect request as soon as we are done updating it. For the synchronous mprotect, we also maintain a user-space bitmap that records the state of a page (protected/unprotected) to avoid superfluous system calls.

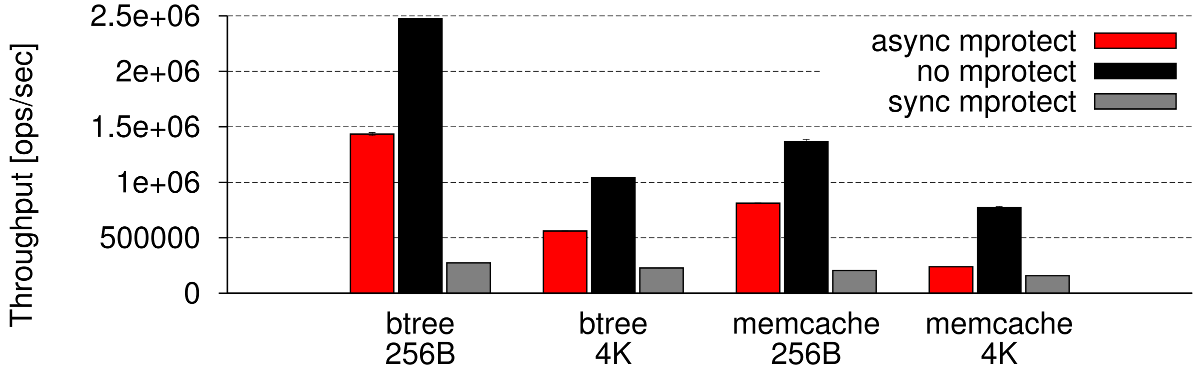


Figure 8: Operations throughput (with 95% c.i.) for (i) B+ tree inserts and (ii) Memcached operations (80% puts, 20% deletes), with 256 B and 4 KB values (8 B keys). For async mprotect, the maximum kernel thread sleep time is 150 ms.

We further modified Memcached to perform all operations in a tight loop, so that we do not take into account the RPC overhead. The overall application speedup when using asynchronous mprotect instead of regular mprotect varies between 34% and 500% depending on the memory object sizes—asynchronous mprotect performs better for small object sizes because more objects will occupy the same page/pages and therefore our batching approach is able to coalesce more requests into fewer page protection changes. The overhead of async mprotect over no mprotect varies between 67% and 323%.

These workloads are selected to stress the protection path in order to evaluate the speedup for protection requests conferred by the asynchronous mprotect mechanism. Real-world uses would likely incur a larger percentage of read operations, which do not require protection changes. The actual (workload-dependent) application throughput can therefore be determined from these results by weighting the overhead by the percentage of reads: For example, memcache-256B with 50% reads would observe a roughly 2.5x slowdown with normal mprotect, and 1.3x using async mprotect.

In reducing the overhead of synchronous mprotect, sorting batched requests by their page address plays an important role: without sorting, the B+ tree test with one million 256 B inserts is 66% slower than asynchronous mprotect with sorting.

We also measure the latency to complete a protection request from the time the application issues the request until the kernel thread protects the page. Figure 9 show a CDF of asynchronous mprotect performance with 1 M inserts of 8 B keys and 256 B values into the B+ tree. Using Linux high resolution timers we can achieve low latencies (50 μs on average) at the expense of throughput: the slowdown of low latency asynchronous mprotect is 7.9$\times$—up from 1.72$\times$ for the high latency async mprotect, but still smaller than that of synchronous mprotect (9$\times$).

In conclusion, our new asynchronous memory protection mechanism gives application programmers the ability to trade off the latency of protection changes (how long a memory object remains unprotected after an update operation) for application performance (the throughput for update operations).

It is important to note that the relative overheads of memory protection (both async and traditional) will be lower for NVRAM than presented here for DRAM, because of NVRAM's slower writes.

8.3 Cache Line Counters

This section compares the insert throughput achieved by our modified B+ tree and the original B+ tree [8] as follows: (a) no modifications and therefore no consistency guarantees, (b) cache line flushes to achieve consistency (as defined in Section 6), and (c) memory mapped so as to use the write-combining memory

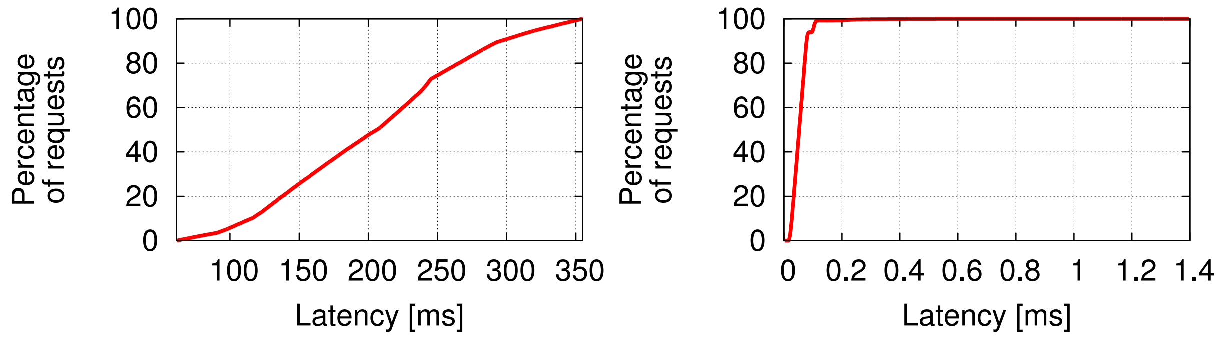


Figure 9: CDF of memory protection latency for the B+ tree using kernel thread wait times of 150 ms (left graph) and 0.01 ms.

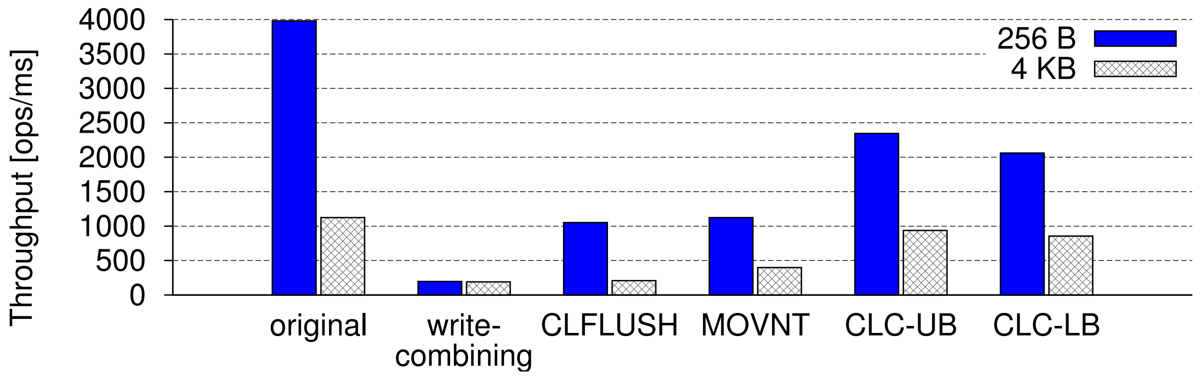


Figure 10: Insert throughput for 10^6 entries (8 B keys, 256 B and 4 KB values respectively) CLC corresponds to cache line counters (LB = lower bound, UB = upper bound).

access protocol, which guarantees that writes go (almost) immediately to memory [14]. Additionally, we also compare with the B+ tree modified to use the streaming writes with a non-temporal hint (MOVNTx Intel SSE instructions), even if they do not guarantee immediate write-back to memory, to give a higher bound on what can be achieved with cache-bypassing techniques [15]. These results use the simulated cache line counter overhead described in Section 7.3.

Figure 10 depicts the results. Because a node split has a transient state (see Section 6) whose duration depends on the cache line replacement rate, we present the results for the cache line counter enabled B+ tree as two data series: one corresponding to transient states ending immediately (an upper bound for performance), and one for transient states lasting indefinitely (a lower bound).

Cache line counters outperform the other consistency preserving mechanisms for inserts. However, this comes at the expense of the read throughput: unlike the B+ tree versions using cache flushes and cache bypassing writes, the nodes in our B+ tree do not maintain the keys in order. As a result, the read throughput decreases by 30% to 32% for 256 B values, and by 10% to 12% for 4 KB values. A possible solution is to lazily sort nodes during periods of low update load, but we leave this for future work.

[14]Linux lacks user-space Page Attributes Table support. We ran the experiments in this section on Windows Vista 64, using Visual C++ 2010, on the same Core i7 machine, except for the tests with SSE streaming writes, which we have only been able to implement with the GCC compiler. The results of those tests that ran on both Windows and Linux were very similar.

[15]Streaming writes may only update the cache if the targeted memory location is in the cache [21].

8.4 DRAM vs. NVRAM

Our evaluation used DRAM as a proxy for NVRAM. As discussed in Section 2, some of the characteristics of DRAM are overly optimistic predictions for NVRAM. We believe, nevertheless, that "high-order bits" of our results would remain when using NVRAM, were it available: VM page tables would still be maintained in DRAM, so the mprotect overhead would be the same (in absolute value; the relative overhead will be lower, due to NVRAM's slower writes). NVMalloc's write patterns would remain the same, but, if NVRAM writes were appreciably slower than DRAM writes, NVMalloc's decreased overall write traffic would give it a performance advantage over traditional allocators. Finally, the advantages of cache line counters would be more evident on NVRAM: by not forcing writes to go directly to memory, we decrease the number of memory bus transactions (which are likely to be slower with NVRAM), as well as the used write bandwidth.

9 Related Work

Related work falls into three main categories: systems that use NVRAM as persistent storage on the memory bus, memory management, and virtual memory protection.

NVRAM on the memory bus. Recent work has addressed some of the challenges of using NVRAM as persistent storage on the memory bus. BPFS [13] is a file system designed specifically for NVRAM. As we do, BPFS identifies cache-bypassing writes and cache flushes as an important source of inefficiency, and proposes CPU modifications—epoch barriers—to allow applications to use normal stores while getting strict guarantees for the order in which these stores will be persisted. Our cache line counter mechanism is a more general and flexible solution: applications themselves control the ordering of updates by delaying making those updates that depend on something still in the CPU caches. This allows for a less intrusive hardware implementation when compared to epoch barriers: the cache line replacement logic does not change (which is important, since otherwise this could negate the benefits of many years of CPU cache optimizations), and the CPU never has to perform cache walks when a cache line has to be written back to memory. Moreover, it is unclear if epoch barriers can work well in a general multi-process setting, outside of BPFS, because barriers would impose false dependencies between unrelated processes.

Since they do not rely on the correct functioning of external subsystems, cache line counters are more reliable than approaches that use capacitors or the residual energy in power supplies [30] to flush the CPU caches in case of power failure.

NV-heaps [12] and Mnemosyne [47] extend the software transactional memory semantics to include persistence. NV-heaps restrict the interface to NVRAM to an object-oriented programming model, and provide garbage collection for persistent memory objects—this is important, because memory leaks are especially problematic for non-volatile memory. We believe the techniques we presented in this paper are complementary to the ones introduced by NV-heaps. Mnemosyne, like Rio Vista [26] before it, also identifies the need for separating the memory allocator metadata from the allocated blocks of non-volatile memory, but it stores this metadata unguarded in NVRAM—this scheme is therefore more vulnerable to corruption and data loss than NVMalloc (see discussion in Section 3.2). NV-heaps uses the BPFS epoch barriers for guarantees about ordering, while Mnemosyne relies on non-cacheable writes.

A versioned B+ tree for NVRAM has been presented in [45]. It relies on cache line flushes for maintaining write ordering.

The aforementioned systems rely solely on external solutions for wear-out prevention.

Memory Management. Throughout this paper, we extensively compare NVMalloc with the GNU C Library's popular malloc implementation, which is based on dlmalloc [25]. Many other memory allocators exist [7, 16, 18], but they are generally focused on multithread performance (Section 8.1.1 also compares with jemalloc and Hoard). Some of their techniques work against our wear leveling goals (e.g., concentrating allocations into as few pages as possible), but most are optimizations orthogonal to ours.

Dhiman et al. [15] presented an OS-level page management technique for wear leveling page allocations. By contrast, NVMalloc is a general-purpose memory allocator.

Virtual Memory Protection Protecting data mapped in the address spaces of processes using virtual memory protection has been employed successfully for over two decades in the context of databases [42, 14] or reliable file system caches on battery-backed DRAM [11]. Mprotect continues to be used for the same purpose today in mainstream databases, but the vendors recommend turning it off when performance is important [46].

We improve the performance of this technique by making the protect operations asynchronous. Our implementation, by avoiding system calls, is reminiscent of FlexSC's exception-less system calls [41]. However, our approach gains more from being tailored to the particularities of mprotect than from avoiding mode switches.

Protecting the memory of modules running within the same address space, particularly device drivers in an OS kernel, has been the subject of much research. Nooks [43] statically provides each kernel extension with its separate page table (with write rights only for its own memory), and uses the regular hardware-enforced virtual memory protection mechanisms. Nooks does not permit direct cross-domain memory access, instead using cross-domain procedure calls. This provides stronger protection than our approach, but is less backwards compatibile and has lower performance when accessing memory directly. For reasons similar to ours, Nooks uses batching to mitigate the cost of changing protection domains (it batches cross-domain procedure calls). BGI [10] uses a compiler plug-in to generate instrumented code that checks permissions for every write to memory, at byte granularity. Unlike BGI, mprotect-based approaches work with existing, un-instrumented library code.

10 Conclusions

Upcoming memory technologies will soon provide a new way for applications to store persistent data at near-DRAM speeds. Harnessing this speed, however, will require placing NVRAM directly on the memory bus. Applications, supported by user-space libraries and the OS, must ensure that this persistent data remains safe from wear-out and corruption. We believe that the techniques described in this paper—a new memory allocator for NVRAM, a virtual memory protection scheme, and cache line counters—can substantially ease the task of creating safe, high-performance persistent data structures for emerging non-volatile memories.

References

[1] Big Sky, MT, October 2009.

[2] ITRS international technology roadmap for semiconductors, 2009.

[3] Newport Beach, CA, March 2011.

[4] S.J. Ahn, Y.J. Song, C.W. Jeong, J.M. Shin, Y. Fai, Y.N. Hwang, S.H. Lee, K.C. Ryoo, S.Y. Lee, J.H. Park, H. Horii, Y.H. Ha, J.H. Yi, B.J. Kuh, G.H. Koh, G.T. Jeong, H.S. Jeong, Kinam Kim, and B.I. Ryu. Highly manufacturable high density phase change memory of 64Mb and beyond. In *Electron Devices Meeting, 2004. IEDM Technical Digest. IEEE International*, pages 907–910, December 2004.

[5] C. Scott Ananian, Krste Asanovic, Bradley C. Kuszmaul, Charles E. Leiserson, and Sean Lie. Unbounded Transactional Memory. *IEEE Micro*, 26:59–69, January 2006.

[6] F. Bedeschi, C. Resta, O. Khouri, E. Buda, L. Costa, M. Ferraro, F. Pellizzer, F. Ottogalli, A. Pirovano, M. Tosi, R. Bez, R. Gastaldi, and G. Casagrande. An 8Mb demonstrator for high-density 1.8V phase change memories. In *VLSI Circuits, 2004. Digest of Technical Papers. 2004 Symposium on*, pages 442–445, June 2004.

[7] Emery D. Berger, Kathryn S. McKinley, Robert D. Blumofe, and Paul R. Wilson. Hoard: A scalable memory allocator for multithreaded applications. In *ASPLOS*, pages 117–128, 2000.

[8] Timo Bingmann. STX B+ Tree C++ Template Classes. http://idlebox.net/2007/stx-btree/, 2008.

[9] Hans-J. Boehm. The Boehm-Demers-Weiser Conservative Garbage Collector. http://www.research.ibm.com/ismm04/slides/boehm-tutorial.ppt, 2004.

[10] Miguel Castro, Manuel Costa, Jean-Philippe Martin, Marcus Peinado, Periklis Akritidis, Austin Donnelly, Paul Barham, and Richard Black. Fast byte-granularity software fault isolation. In *Proc. 22nd ACM Symposium on Operating Systems Principles (SOSP)* [1], pages 45–58.

[11] Peter M. Chen, Wee Teck Ng, Subhachandra Chandra, Christopher Aycock, Gurushankar Rajamani, and David Lowell. The Rio file cache: Surviving operating system crashes. In *Proc. 7th International Conf. on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, pages 74–83, Cambridge, MA, October 1996.

[12] Joel Coburn, Adrian M. Caulfield, Ameen Akel, Laura M. Grupp, Rajesh K. Gupta, Ranjit Jhala, and Steven Swanson. NV-Heaps: Making persistent objects fast and safe with next-generation, non-volatile memories. In *Proc. 16th International Conf. on Architectural Support for Programming Languages and Operating Systems (ASPLOS)* [3].

[13] Jeremy Condit, Edmund B. Nightingale, Christopher Frost, Engin Ipek, Benjamin Lee, Doug Burger, and Derrick Coetzee. Better I/O through byte-addressable, persistent memory. In *Proc. 22nd ACM Symposium on Operating Systems Principles (SOSP)* [1], pages 133–146.

[14] G. Copeland, T. Keller, R. Krishnamurthy, and M. Smith. The case for Safe RAM. In *Proceedings of the 15th International Conference on Very Large Data Bases*, VLDB '89, pages 327–335, San Francisco, CA, USA, 1989. Morgan Kaufmann Publishers Inc.

[15] G. Dhiman, R. Ayoub, and T. Rosing. PDRAM: A Hybrid PRAM and DRAM Main Memory System. In *Design Automation Conference, 2009. DAC '09. 46th ACM/IEEE*, pages 664 –669, July 2009.

[16] Jason Evans. A Scalable Concurrent malloc(3) Implementation for FreeBSD. BSDCan - The BSD Conference, 2006.

[17] Gregory R. Ganger and Yale N. Patt. Soft updates: a solution to the metadata update problem in file systems. *ACM Transactions on Computer Systems*, 18:127–153, 2000.

[18] Sanjay Ghemawat and Paul Menage. TCMalloc : Thread-Caching Malloc. http://goog-perftools.sourceforge.net/doc/tcmalloc.html.

[19] Brian Holden. Latency comparison between HyperTransport$^{\mathrm{TM}}$ and PCI-Express$^{\mathrm{TM}}$ in communications systems. http://www.hypertransport.org/docs/wp/Low_Latency_Final.pdf, 2006.

[20] IBM WebSphere Real Time for Real Time Linux, version 2 Information Center. http://publib.boulder.ibm.com/infocenter/realtime/v2r0/index.jsp, 2006.

[21] Intel 64 and IA-32 Architectures Developer's Manual: Vol. 1. http://www.intel.com/content/www/us/en/architecture-and-technology/, 2011.

[22] Takayuki Kawahara. Scalable spin-transfer torque ram technology for normally-off computing. *IEEE Design & Test of Computers*, 28(1):52–63, 2011.

[23] M. Kharbutli and Yan Solihin. Counter-based cache replacement and bypassing algorithms. *Computers, IEEE Transactions on*, 57(4):433 –447, april 2008.

[24] Leslie Lamport. Proving the Correctness of Multiprocess Programs. *Software Engineering, IEEE Transactions on*, SE-3(2):125 – 143, March 1977.

[25] Doug Lea. A Memory Allocator. http://g.oswego.edu/dl/html/malloc.html, 2000.

[26] David E. Lowell and Peter M. Chen. Free Transactions with Rio Vista. In *Proceedings of the sixteenth ACM Symposium on Operating Systems Principles*, SOSP '97, pages 92–101, New York, NY, USA, 1997. ACM.

[27] Chi-Keung Luk, Robert Cohn, Robert Muth, Harish Patil, Artur Klauser, Geoff Lowney, Steven Wallace, Vijay Janapa, and Reddi Kim Hazelwood. Pin: Building Customized Program Analysis Tools with Dynamic Instrumentation. In *In Programming Language Design and Implementation*, pages 190–200. ACM Press, 2005.

[28] A distributed memory object caching system. http://memcached.org/, 2011.

[29] K. E. Moore, J. Bobba, M. J. Moravan, M. D. Hill, and D. A. Wood. LogTM: log-based transactional memory. In *High-Performance Computer Architecture, 2006. The Twelfth International Symposium on*, pages 254–265, February 2006.

[30] Dushyanth Narayanan and Orion Hodson. Whole-system persistence. In *Proceedings of the seventeenth international conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '12, pages 401–410, New York, NY, USA, 2012. ACM.

[31] Numonyx. The basics of phase change memory (PCM) technology. http://www.numonyx.com/Documents/WhitePapers/PCM_Basics_WP.pdf, 2008.

[32] Numonyx. Phase change memory. http://www.pdl.cmu.edu/SDI/2009/slides/Numonyx.pdf, 2009.

[33] Numonyx. Phase change memory (PCM): A new memory technology to enable new memory usage models. http://www.numonyx.com/Documents/WhitePapers/Numonyx_PhaseChangeMemory_WhitePaper.pdf, 2009.

[34] Moinuddin K. Qureshi, John Karidis, Michele Franceschini, Vijayalakshmi Srinivasan, Luis Lastras, and Bulent Abali. Enhancing Lifetime and Security of PCM-based Main Memory with Start-Gap Wear Leveling. In *Proc. ACM MICRO*, 2009.

[35] Moinuddin K. Qureshi, Andre Seznec, Luis Lastras, and Michele Franceschini. Practical and Secure PCM Systems by Online Detection of Malicious Write Streams. In *Proc. HPCA*, 2011.

[36] Bratin Saha, Ali-Reza Adl-Tabatabai, and Quinn Jacobson. Architectural Support for Software Transactional Memory. In *Proceedings of the 39th Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO 39, pages 185–196, Washington, DC, USA, 2006. IEEE Computer Society.

[37] Samsung. Samsung Ships Industry's First Multi-chip Package with a PRAM Chip for Handsets. http://www.samsung.com/us/aboutsamsung/news/newsIrRead.do?news_ctgry=irnewsrelease&page=1&news_seq=18828&rdoPeriod=ALL&from_dt=&to_dt=&search_keyword=, 2010.

[38] Virtualized SAP Performance with VMware vSphere 4. http://www.vmware.com/files/pdf/perf_vsphere_sap.pdf, 2009.

[39] M. Satyanarayanan, Henry H. Mashburn, Puneet Kumar, David C. Steere, and James J. Kistler. Lightweight Recoverable Virtual Memory. *ACM Transactions on Computer Systems*, 12:33–57, February 1994.

[40] Andre Seznec. A Phase Change Memory as a Secure Main Memory. *IEEE Comp. Arch. Letters*, 9:5–8, 2010.

[41] Livio Soares and Michael Stumm. FlexSC: Flexible system call scheduling with exception-less system calls. In *Proc. 9th USENIX OSDI*, Vancouver, Canada, October 2010.

[42] Mark Sullivan and Michael Stonebraker. Using write protected data structures to improve software fault tolerance in highly available database management systems. In *Proc. VLDB*, 1991.

[43] Michael M. Swift, Brian N. Bershad, and Henry M. Levy. Improving the reliability of commodity operating systems. *ACM Transactions on Computer Systems*, pages 77–100, 2005.

[44] How long does it take to make a context switch? http://blog.tsunanet.net/2010/11/how-long-does-it-take-to-make-context.html.

[45] Shivaram Venkataraman, Niraj Tolia, Parthasarathy Ranganathan, and Roy Campbell. Consistent and Durable Data Structures for Non-Volatile Byte-Addressable Memory. In *Proceedings of the 9th USENIX Conference on File and Storage Technologies (FAST '11)*, San Jose, CA, February 2011.

[46] VMware Best Practices for SAP installations. http://communities.vmware.com/blogs/SAPsolutions/2008/01/18/vmware-best-practices-for-sap-installations, 2008.

[47] Haris Volos, Andres Jaan Tack, and Michael M. Swift. Mnemosyne: Lightweight persistent memory. In *Proc. 16th International Conf. on Architectural Support for Programming Languages and Operating Systems (ASPLOS)* [3].

[48] Paul R. Wilson. Pointer Swizzling at Page Fault Time: Efficiently Supporting Huge Address Spaces on Standard Hardware. *SIGARCH Computer Architecture News*, 19:6–13, July 1991.

[49] Paul R. Wilson, Mark S. Johnstone, Michael Neely, and David Boles. Dynamic storage allocation: A survey and critical review. In Henry G. Baker, editor, *IWMM*, volume 986 of *Lecture Notes in Computer Science*, pages 1–116. Springer, 1995.