Optimizing Systems for Byte-Addressable NVM by Reducing Bit Flipping

Daniel Bittman, Darrell D. E. Long, Peter Alvaro, and Ethan L. Miller, *UC Santa Cruz*

https://www.usenix.org/conference/fast19/presentation/bittman

This paper is included in the Proceedings of the
17th USENIX Conference on File and Storage Technologies (FAST '19).
February 25–28, 2019 • Boston, MA, USA
978-1-939133-09-0

Open access to the Proceedings of the
17th USENIX Conference on File and
Storage Technologies (FAST '19)
is sponsored by

Optimizing Systems for Byte-Addressable NVM by Reducing Bit Flipping

Daniel Bittman
UC Santa Cruz

Peter Alvaro
UC Santa Cruz

Darrell D. E. Long
UC Santa Cruz

Ethan L. Miller
UC Santa Cruz
Pure Storage

Abstract

New byte-addressable non-volatile memory (BNVM) technologies such as phase change memory (PCM) enable the construction of systems with large persistent memories, improving reliability and potentially reducing power consumption. However, BNVM technologies only support a limited number of lifetime writes per cell and consume most of their power when flipping a bit's state during a write; thus, PCM controllers only rewrite a cell's contents when the cell's value has changed. Prior research has assumed that reducing the number of *words* written is a good proxy for reducing the number of *bits* modified, but a recent study has suggested that this assumption may not be valid. Our research confirms that approaches with the fewest writes often have *more* bit flips than those optimized to reduce bit flipping.

To test the effectiveness of bit flip reduction, we built a framework that uses the number of bits flipped over time as the measure of "goodness" and modified a cycle-accurate simulator to count bits flipped during program execution. We implemented several modifications to common data structures designed to reduce power consumption and increase memory lifetime by reducing the number of bits modified by operations on several data structures: linked lists, hash tables, and red-black trees. We were able to reduce the number of bits flipped by up to $3.56\times$ over standard implementations of the same data structures with negligible overhead. We measured the number of bits flipped by memory allocation and stack frame saves and found that careful data placement in the stack can reduce bit flips significantly. These changes require no hardware modifications and neither significantly reduce performance nor increase code complexity, making them attractive for designing systems optimized for BNVM.

1 Introduction

As byte-addressable non-volatile memories (BNVMs) become common [15, 18, 24], it is increasingly important that systems are optimized to leverage their strengths and avoid stressing their weaknesses. Historically, such optimizations have included reducing the number of writes performed, either by designing data structures that require fewer writes or by using hardware techniques such as caching to reduce writes. However, it is the number of *bits flipped* that matter most for BNVMs such as phase-change memory (PCM), *not* the number of words written.

BNVMs such as PCM suffer from two problems caused by flipping bits: energy usage and cell wear-out. As these memory technologies are adopted into longer-term storage solutions and battery powered mobile and IoT devices, their costs become dominated by physical replacement from wear-out and energy use respectively, so increasing lifetime and dropping power consumption are vital optimizations for BNVM. Flipping a bit in a PCM consumes $15.7-22.5\times$ more power than reading a bit or "writing" a bit that does not actually change [13, 14, 24, 29]. Thus, many controllers optimize by only flipping bits when the value being written to a cell differs from the old value [39]. While this approach saves some energy, it cannot eliminate flips required by software to update modified data structures. An equally important concern is that PCM has limited endurance: cells can only be written a limited number of times before they "wear out". Unlike flash, however, PCM cells are written individually, so it is possible (and even likely) that some cells will be written more than others during a given period because of imbalances in values written by software. Reducing bit flips, an optimization goal that has yet to be sufficiently explored, can thus both save energy and extend the life of BNVM.

Previously, we showed that small changes in data structures can have large impacts in the bit flips required to complete a given set of data structure modifications [4]. While it is possible to reduce bits flipped with changes to hardware, we can gain more by optimizing compiler constructs and choosing data structures to take advantage of semantic information that is not available at other layers of the stack; it is critical we design our data structures with this in mind. Successful BNVM-optimized systems will need to target new optimizations for BNVM, including bit flip reduction.

We implemented three such data structures and evaluated the impact on the number of writes and bit flips, demonstrating the effectiveness of designing data structures to minimize bit flips. These simple changes reduce bit flips by as much as 3.56×, and therefore will reduce power consumption and extend lifetime by a proportional amount, with no need to modify the hardware in any way. Our contributions are:

- Implementation of bit flip counting in a full cycle-accurate simulation environment to study bit flip behavior.
- Empirical evidence that reducing memory writes may not reduce bit flips proportionally.
- Measurements of the number of bit flips required by operations such as memory allocation and stack frame use, and suggestions for reducing the bit flips they require.
- Modification of three data structures (linked lists, hash tables, red-black trees) to reduce bit flips and evaluation of the effectiveness of the techniques.

The paper is organized as follows. Section 2 gives background demonstrating how bit flips impact power consumption and BNVM lifetime. Section 3 discusses some techniques for reducing bit flips in software, which are evaluated for bit flips (Section 4) and performance (Section 5). Section 6 discusses the results, followed by comments on future work (Section 7) and a conclusion (Section 8).

2 BNVM and Bit Flips

Non-volatile memory technologies [6] such as phase-change memory (PCM) [24], resistive RAM (RRAM, or memristors) [33, 35], Ferroelectric RAM (FeRAM) [15], and spin-torque transfer RAM (STT-RAM) [22], among others, have the potential to fundamentally change the design of devices, operating systems, and applications. Although these technologies are starting to make their way into consumer devices [18] and embedded systems [33], their full potential will be seen when they replace or coexist with DRAM as byte-addressable non-volatile memory (BNVM). Such a memory hierarchy will allow the processor, and thus applications, to use load and store instructions to update persistent state, bypassing the high-latency I/O operations of the OS. However, power consumption, especially for write operations, and device lifetime are more serious concerns for these technologies than for existing memory technologies.

2.1 Optimizing for Memory Technologies

Data structures should be designed to exploit the advantages and mitigate the disadvantages of the technologies on which they are deployed. For example, data structures for disks are block-oriented and favor sequential access, while those designed for flash reduce writes, especially random writes, often by trading them for an increase in random reads [10]. Prior data structures and programming models for NVM [9, 11, 16, 25, 36, 38] have typically exploited its byte-addressability while mitigating the relatively slow access times of most BNVM technologies. However, in the case of technologies such as PCM or RRAM, existing research ignores two critical characteristics: asymmetric read/write power usage and the ability to avoid rewriting individual bits that are unchanged by a write [6, 39].

For example, writes to PCM are done by melting a cell's worth of material with a relatively high current and cooling it at two different rates, leaving the material in either an amorphous or crystalline phase [30]. These two phases have different electrical resistance, each corresponding to a bit value of zero or one. The writing process takes much more energy than reading the phase of the cell, which is done by sensing the cell's resistance with a relatively low current. To save energy, the PCM controller can avoid writing to a cell during a write if it already contains the desired value [39], meaning that the major component of the power required by a write is proportional not to the number of bits (or words) *written*, but rather to the number of bits *actually flipped* by the write. Based on this observation, we should design data structures for BNVM to minimize the number of bits flipped as the structures are modified and accessed rather than simply reducing the number of writes, as is more commonly done.

2.2 Power Consumption of PCM and DRAM

While our research applies to any BNVM technology in which writes are expensive, we focus on PCM because its power consumption figures are more readily available. Figure 1 shows the estimated power consumption of 1 GB of DRAM and PCM as a function of bits flipped per second, using power measurements from prior studies of memory systems [4, 7, 13, 14, 24, 29]. The number of writes to DRAM has little effect on overall power consumption since the *entire* DRAM must be periodically refreshed (read and rewritten); refresh dominates, resulting in a high power requirement regardless of the number of writes. In contrast, PCM requires no "maintenance" power, but needs a great deal more energy to write an individual bit (~50 pJ/b [2]) compared to the low overhead for writing a DRAM page (~1 pJ/b [24]). The result is that power use for DRAM is largely proportional to memory *size*, while power consumption for PCM is largely proportional to *cell change rate*. The exact position of the cross-over point in Figure 1 will be narrowed down as these devices become more common; many features of these devices, including asymmetric write-zero and write-one costs, increased density of PCM over DRAM, and decreasing feature sizes, will affect the trade-off point over time.

Figure 1 demonstrates the need for data structures for PCM to minimize cell writes. Because the memory controller can minimize the cost of "writing" a memory cell with the same value it already contains, the primary concern for

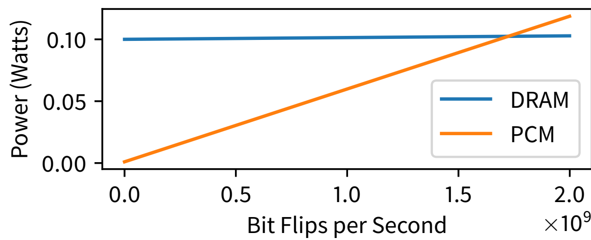

Figure 1: Power use as a function of flips per second [4].

data structures in PCM is reducing the number of bit flips, which the memory controller cannot easily eliminate.

Power consumption is particularly concerning for battery-operated Internet of Things (IoT) devices, which may become a significant consumer of BNVM technologies to facilitate fast power-up and reduce idle power consumption [20, 21]. Devices that collect large amounts of data and write frequently to BNVM may find power usage increasing depending on access patterns. Thus, IoT devices may benefit significantly from bit-flip-aware systems and data structures.

2.3 Wear-out

Another significant advantage to avoiding bit flips is reducing memory cell wear-out. BNVM technologies typically have a maximum number of lifetime writes, and fewer writes means a longer lifetime. However, by avoiding unnecessary overwrites, the controller would introduce uneven wear *within* BNVM words where some of the bits flip more frequently than others due to biases of certain writes. For example, pointer overwrites may only alter the low-order bits, except for the few that are zero because of structure alignment in memory, if the pointers are to nearby regions. Thus, the middle bits in a 64-bit word may wear out faster than the lowest and highest bits. While reducing bit-flips increases the average lifetime of the cells in a word, it has the potential to exacerbate the uneven wear problem since such techniques might increase the biases of certain writes.

Fortunately, we can take advantage of existing research in wear-leveling for BNVM that allows the controller to spread out the cell updates within a given word. While a full remapping layer similar to a flash translation layer is infeasible for BNVM—the overhead would be too high—hardware techniques such as row shifting [40], content-aware bit shuffling [17], and start-gap wear leveling [28] may be able to mitigate biased write patterns with low overhead. This would allow BNVM to leverage bit flip reduction to reduce wear even if the result is that some bits are flipped more frequently than others. These techniques, implemented at the memory controller level, can work in tandem with the techniques described in this paper since they benefit bit flip reduction and can distribute "hot" bits across a word, mitigating the biased write patterns bit flip reduction techniques may introduce.

2.4 Reducing Impact of Bit Flips in BNVM

Although bit flips in BNVM have been studied previously, much of that work has focused on hardware encoding, which re-encodes cache lines to reduce bit flips, but re-encoding has limited efficacy [8, 19, 32] because it must also store information on *which* encoding was used. While hardware techniques are worth exploring, software techniques to reduce bit flips can be more effective because they can leverage semantic knowledge available in the software but not visible in the memory controller's limited view of single cache lines.

Chen *et al.* [7] evaluate data structures on BNVM and argue that reducing bit flips is workload dependent and difficult to reason about, so we should strive to reduce writes because writes are approximately proportional to bit flips. We found that this is often *not* the case—our prior experiments revealed that bit flips were often *not* proportional to writes, and we were able to examine bit flips and optimize for them in an example data structure [4]. These findings are further corroborated by our experiments in Section 4.

Since bit flips directly affect power consumption and wear, we can study three separate aspects for bit flip reduction:

- **Data structure design**: Since data organization plays a large role in the writes that make it to memory, we designed new data structures built around the idea of *pointer distance* [34] instead of storing pointers directly. While *data* writes themselves significantly affect bit flips, these writes are often unavoidable (since the data must be written), while data *structure* writes are more easily optimized (as we see in existing BNVM data structure research). Furthermore, data structures often require a significant number of updates over time, while data is often written once (since we can reduce writes by updating pointers instead of moving data). Thus the overall proportion of bit flips caused by data writes may drop over time as data structures are updated.
- **Effects of program operation**: A common source of writes is the stack, where return addresses, saved registers, and register spills are written. Understanding how these writes affect bit flips plays a critical role in recommendations for bit flip reduction for system designers.
- **Effects of caching layers**: Since writes must first go through the cache, it is vital to understand how different caching layers and cache sizes affect bit flips in memory. Complicating matters is the unique consistency challenges of BNVM [9, 11, 36], wherein programs often flush cache-lines to main memory more frequently than they otherwise would, use write-through caching, or more complex, hardware-supported cache flushing protocols. These questions are evaluated in Section 4.6.

3 Reducing Bit Flips in Software

By reducing bit flips in software, we can effect improvements in BNVM lifetime and power use without the need for hardware changes. To build data structures to reduce bit flips (Sections 3.1–3.3), we propose several optimizations to pointer storage along with additional optimizations for indicating occupancy. For stack writes, we propose changes to compilers to spill registers such that they avoid writing different registers to the same place in the stack (Section 3.4).

3.1 XOR Linked Lists

XOR linked lists [34] are a memory-efficient doubly-linked list design where, instead of storing a previous and next node pointer, each node stores only a `siblings` value that is the XOR between the previous and next node. If the previous node is at address p and the next node is at address n, the node stores $siblings = p \oplus n$. This scheme cuts the number of stored pointers per node in half while still allowing bi-directional traversal of the list—having pointers to two adjacent nodes is sufficient to traverse both directions. However, an XOR linked list has disadvantages; it does not allow $O(1)$ removal of a node with just a single pointer to that node, as a node's siblings cannot be determined from the node alone, and it increases code complexity by requiring XOR operations before pointers are dereferenced.

When they were proposed, XOR linked lists had little advantage over doubly linked lists beyond a modest memory saving. However, with the need for fewer bit flips on BNVM, they gain a critical advantage: they cut the number of stored pointers in half, reducing writes, but they also store the XOR of two pointers, which are likely to contain similar higher-order bits, making the `siblings` pointer mostly zeros.

One problem with the original design for XOR linked lists is that each node stores $siblings = p \oplus n$, but for the first and last node, p or n are `NULL`, so the full pointer value for its adjacent node is stored in the head and tail. To further cut down on bit flips, we changed this design so that the head and tail XOR their adjacent nodes with themselves (if the node at address h is the head, then it stores $siblings = h \oplus n$ instead of $siblings = 0 \oplus n$). The optimization here is *not* a performance optimization—in fact, it's likely to reduce performance—and only makes sense in the context of bit flips, an optimization goal that would not be targeted before the introduction of BNVM. However, with bit flips in-mind, it becomes critical. Other data structures may have similar optimizations that we can easily make to reduce bit flips [1].

[1] Circular linked lists solve the head and tail siblings pointer problem automatically, since no pointers are stored as `NULL`; however, in XOR linked lists this increases the number of pointer updates during an insert operation and requires storing two adjacent head nodes to traverse.

3.2 XOR Hash Tables

A direct application of XOR linked lists is chained hashing, a common technique for dealing with hash table collisions [12]. An array of linked list heads is maintained as the hash table, and when an item is inserted, it is appended to the list at the bucket that the item hashes to. To optimize for bit flips, we can store an XOR list instead of a normal linked list, but since bidirectional traversal is not needed in a hash table bucket, we need not complicate the implementation with a full XOR linked list. Instead, we apply the property of XOR linked lists that we find useful—XORing pointers.

Each pointer in each list node is XORed with the address of the node that contains that pointer. For example, a list node n whose next node is p will store $n \oplus p$ instead of p. In effect, this stores the distance between the nodes rather than the absolute address of the next node and exploits locality in memory allocators. The end of the list is marked with a `NULL` pointer. In addition to a distance pointer, each node contains a key and a pointer to a value. The list head stored in the hash table is a full node, allowing access to the first entry in the list without needing to follow a pointer.

A second optimization we make is that an empty list can be marked in one of two ways: the least-significant bit (LSB) of the `next` pointer set to one, or the data pointer set to `NULL`. When we initialize the table, it is set to zero everywhere, so the data pointers are `NULL`. During delete, if the list becomes empty, the LSB of the next pointer in the list head is set to 1, a value it would never have when part of a list. This allows the data pointer to remain set to a value such that when it is later overwritten, fewer bits need to change. This is an example of an optimization that only makes sense in the context of bit flips, as it increases code complexity for no other gain.

3.3 XOR Red-Black Trees

Binary search trees are commonly used for data indexing, support range queries, and allow efficient lookup and modification, as long as they are balanced. Red-black trees [12, 31] are a common balanced binary tree data structure with strictly-bounded rebalancing operations during modification. A typical red-black tree (RBT) node contains pointers to its left child and right child, along with meta-data. They often also contain a pointer to the parent node, since this enables easier balancing implementation and more efficient range-query support without significantly affecting performance due to the increased memory usage [23].

We can generalize XOR linked lists to *XOR trees*. Instead of storing `left`, `right`, and `parent` pointers, each node stores `xleft` and `xright`, which are the XOR between each child and the parent addresses. This reduces the memory usage to the two-pointer case while maintaining the benefits of having a parent pointer, since given a node and one of its children (or its parent), we can traverse the entire tree.

Like XOR linked lists, the root node stores `xleft = root ⊕ left`, where `root` is the address of the root node and `left` is the address of its left child, saving bit flips. To indicate that a node has no left or right child, it stores `NULL`.

Determining the child of a node requires both the node and its parent:

```
get_left_child(Node *node, Node *parent) {
  return (parent ⊕ node->xleft);
}
```

Getting a node's parent, however, requires additional work. Given a child c and a node n, getting n's parent requires we know *which* child (left or right) c is. Fortunately, in a binary search tree we store the key k of a node in each node, and the nodes are well-ordered by their k. Thus, getting the parent works as follows:

```
get_parent(Node *n, Node *c) {
  if(c->k < n->k) return (n->xleft ⊕ c);
  else return (n->xright ⊕ c);
}
```

Note that this is not the only way to disambiguate between pointers. In fact, it's not strictly necessary to do so because the algorithms can be implemented recursively without ever needing to traverse up the tree explicitly. However, providing upwards traversal can reduce the complexity of implementation and improve the performance of iteration over ranges. Another solution to getting the parent node would be to record whether a node is a left or right child by storing an extra bit along with the color. We did not evaluate this method, as it would increase both writes and bit flips over our method.

With these helper functions, we implemented both an XOR red black tree (`xrbt`) and a normal red-black tree (`rbt`) using similar algorithms. The code for `xrbt` was just 20 lines longer, with only a minor increase in code complexity. Node size was smaller in `xrbt`, with a node being 40 bytes instead of 48 bytes as in `rbt`. To control for the effects of node size on performance and bit flips, we built a variant of `xrbt` with the same code but with a node size of 48 bytes (`xrbt-big`).

Generalization These techniques can generalize beyond a red-black tree. Any ordered k-ary tree can use XOR pointers in the same way. As discussed above, disambiguating between pointers during traversal depends on either additional bits being stored or using an ordering property. Either technique can work with arbitrary graph nodes.

3.4 Stack Frames

Data structure layout and data writes are only some of the writes made by a program. Register spills, callee-saved register saving, and return addresses pushed during function calls are all writes to memory, and if these writes make it to BNVM, they will cause bit flips as well. These writes may make it to main memory if the cache is saturated or if the program is designed to keep program state in BNVM to enable instantaneous restart after power cycles [26]. Additionally, systems designed for BNVM may run with write-through caches to reduce consistency complexity, resulting in execution state reaching BNVM.

The exact pattern of stack writes depends on the ABI and the calling convention of a system and processor, though we focus on x86-64 Linux systems. When a program calls a function, it (potentially) pushes a number of arguments to the stack, followed by a return address. In the called function, callee-saved registers are pushed to the stack, but *only* if they are modified during that function's execution. When finished, the callee pops all the saved registers and returns.

Our observation is that the order that callee-saved registers are pushed to the stack is *not specified*, meaning that two different functions could push the same registers in a different order. Secondly, the same callee-saved register is less-likely to change drastically in a small amount of code in a tight loop, since these registers are typically used for loop counters or bases for addressing. Thus, a loop that calls two functions alternately will likely have similar or the same values in the callee-saved registers during the invocation of both functions. If these two functions push the (often unchanged) callee-saved registers to the same place both times, fewer bit flips will occur than if the functions pushed them in different orders. While this is just a simple example, such loops that call out to alternating functions with different characteristics can occur, for example, when rehashing a table, rebuilding a tree, or reading task items from a linked list.

We propose specifying a callee-saved register frame layout that functions adhere to, so that the registers are always pushed in the same order. To handle variable numbers of arguments, we make use of passing arguments in registers, common in many modern ABIs. If a function need not push any callee-saved registers, it can still reserve the stack space for that frame and then not push anything to save writes. Functions which only save a small number of registers can still push them to the correct locations within the frame. Finally, if this is standardized, programs need not worry about library calls increasing bit flips.

For example, if we have two functions `A` and `B` in an ABI where registers e, f, g, h are callee-saved, and `A` uses e while `B` uses g, then traditionally each function would simply push the frame pointer followed by the register they wish to save:

```
A:                        B:
push fp                   push fp
mov fp ← sp               mov fp ← sp
push e                    push g
...                       ...
pop e                     pop g
pop fp; ret               pop fp; ret
```

If e and g are significantly different, then a significant amount of needless bit flips could occur if these functions

are called often. Instead, if we define a layout that functions adhere to for register saving, the code would look like:

```
A:
push fp
mov fp ← sp
push e
sub sp, 24
...
add sp, 24
pop e
pop fp; ret
```

```
B:
push fp
mov fp ← sp
sub sp, 16
push g
sub sp, 8
...
add sp, 8
pop g
add sp, 16
pop fp; ret
```

Here the code always pushes the same register to the same place, regardless of the registers it needs to save, thereby allowing overwrites by likely similar values. While it does add some additional instructions, code could instead write registers directly to the stack locations using offset style addressing, reducing code size.

4 Memory Characteristics Results

We evaluated XOR linked lists, XOR hash tables, and XOR red-black trees, tracking bits flipped in memory, bytes written to memory, and bytes read from memory during program execution. Our goal was not only to demonstrate that our bit flip optimizations were effective, but to also understand how different system and program components affected bit flips. In addition to tracking bit flips caused by our data structures, we also studied bit flips caused by varying levels and sizes of caching, calls to `malloc`, and writes to the stack. Finally, we evaluated the accuracy of in-code instrumentation for bit flips, which would allow programmers to more easily optimize for bit flips at lower cost than full-system simulation. All of these experiments were designed to demonstrate how effective certain bit flipping reduction techniques are. Existing systems are poorly equipped to handle evaluation of these techniques, since existing systems are poorly optimized for BNVM. The techniques we present here are designed to be used by system designers when building *new*, BNVM-optimized systems.

4.1 Experimental Methods

Evaluating bit flips during data structure operations requires more than simply counting the bits flipped in each write in the code. Compiler optimizations, store-ordering, and the cache hierarchy can all conspire to change the order and frequency of writes to main memory, potentially causing a manual count of bit flips in the code to deviate from the bits flipped by writes that *actually* make it to memory. To record better metrics than in-code instrumentation, we ran our test programs on a modified version of Gem5 [3], a full-system simulator that accurately tracks writes through the cache hierarchy and memory. We modified the simulator's memory system so that, for each cache-line written, it could compute the Hamming distance between the existing data and the incoming write, thereby counting the bit flips caused by each write to memory. The bit flips for each write were added to a global count, which was reported after the program terminated, along with the number of bytes written to and read from memory. This gave us a more accurate picture of the bit flips caused by our programs, since writes that stay within the simulated cache hierarchy do not contribute to the global count. We ran the simulator in *system-call emulation* mode, which runs a cycle-accurate simulation, emulating system calls to provide a Linux-like environment, while tracking statistics about the program, including the memory events we recorded.

Table 1: Cache parameters used in Gem5.

Cache	Count	Size	Associativity
L1d	1	64KB	2-way
L1i	1	32KB	2-way
L2	1	2MB	8-way

We used the default cache hierarchy (shown in Table 1) provided by Gem5, using the command-line options "`--caches --l2cache`". For the XOR linked list and stack writes experiments, we used `clwb` instructions to simulate consistency points (in the linked list, `clwb` was issued to persist the contents of a node before persisting the pointers to the node, and for stack writes, `clwb` was issued after each write). This was not done for the `malloc` experiment (we used an unmodified system `malloc` for testing), the XOR hash table (the randomness of access to the table quickly saturated the caches anyway), or manual instrumentation (caches were irrelevant). For the XOR red-black tree, in addition to the bit flip characteristics, we focused on observing how cache behavior affected more complex data structures; these results, along with the results of varying L2 size, are discussed in Section 4.6.

Most of the programs we ran accept as their first argument an `iteration_count`, which specifies how many iterations the program should run. For example, the red-black tree would do `iteration_count` number of insertions. We ran the simulator on a range of `iteration_counts`, recording the bits flipped, bytes written, and bytes read (collectively referred to as *memory events*) for each value of `iteration_count`. An example of a typical result is shown in Figure 2. The result was often linear, allowing us to calculate a linear regression using gnuplot, giving us both a slope and confidence intervals. The slope of the line is "bit flips per operation"—for example, a slope of 10 for linked list insert means that it flipped 10 bits on average during insert operations. Throughout our results, only the slope is presented unless the raw data is non-linear. Since the slope encodes

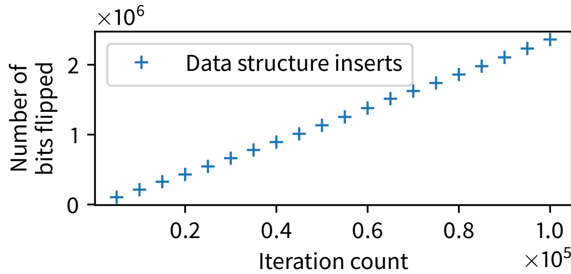

Figure 2: A typical result of running a test program with increasing values of `iteration_count`.

the bit flips per operation, we can directly compare variants of a data structure by comparing their slopes. Error bars are 95% confidence intervals.

4.2 Calls to `malloc`

Many data structures allocate data during their operation. For example, a binary tree may allocate space for a node during insert or a hash table might decide to resize its table. An allocator allocating data from BNVM must store the allocation metadata within BNVM as well, so the internal allocator structures affect bit flips for data structures which allocate memory. Additionally, the pointers returned *themselves* contribute to the bits flipped as they are written.

We called `malloc` 100,000 times with allocation sizes of 16, 24, 40, and 48 bytes. We chose these sizes because our data structure nodes were all one of these sizes. The number of bits flipped per `malloc` call is shown in Figure 3. As expected, larger allocation sizes flip more bits, since the allocator meta-data and the allocated regions span additional cache lines. Interestingly, 40 byte allocations and 48 byte allocations switch places partway through, with 40 byte allocations initially causing fewer bit flips and later causing more after a cross-over point. We believe this is due to 40 byte allocations using fewer cache lines, but 48 byte allocations having better alignment.

After a warm-up period where the cache hierarchy has a greater effect, the trends become linear, allowing us to calculate the bit flips per `malloc` call. Allocating 40 bytes costs 1.5× more bit flips on average than allocating 48 bytes. Allocating 24 or 16 bytes has the same flips per `malloc` as 48 bytes but has a longer warm-up period, such that programs would need to call `malloc` (24) 1.56× as often to flip the same number of bits as `malloc` (48).

While the relative savings for bit flips between `malloc` sizes are significant, their absolute values must be taken into consideration. Calls to `malloc` for 16 and 48 bytes cost 2 ± 0.1 flips per `malloc` (after the warm-up period) while calls to `malloc` for 40 bytes cost 3 ± 0.1 flips per `malloc`. As we will see shortly, the data structures we are evaluating flip tens of bits per operation, indicating that savings from `malloc` sizes are less significant than the specific optimizations they employ.

4.3 XOR Linked Lists

We evaluated the bit flip characteristics of an XOR linked list compared to a doubly-linked list, where we randomly inserted (at the head) and popped nodes from the tail at a ratio of 5:1 inserts to pops. The results are shown in Figure 4. As expected, bit flips are significantly reduced when using XOR linked lists, by a factor of 3.56×. However, both the number of bytes written to and read from memory were the same between both lists. The reason is that, although an XOR list node is smaller, `malloc` actually allocates the same amount of memory for both.

We counted the number of pointer read and write operations in the code, and discovered that, although the XOR linked list performs fewer write operations during updates, it performs *more* read operations than the doubly-linked list. This is because updating the data structure requires more information than in a doubly-linked list. However, Figure 4 shows that the number of reads *from memory* are the same, indicating that the additional reads are always in-cache.

4.4 XOR Hash Tables

We implemented two variants of our hash table: "single-linked", which implemented chaining using a standard linked list, and "XOR Node", which XORs each pointer in the chain with the address of the node containing the pointer. We ran a Zipfian workload on them [5], where 80% of updates happen to 20% of keys[2], where keys and values were themselves Zipfian. During each iteration, if a key was present, it was deleted, while if it was not present, it was

[2]Skew of 1, with a population of 100,000.

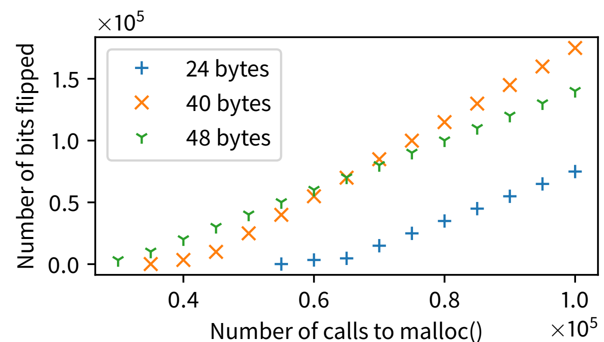

Figure 3: Bit flips due to calls to `malloc`. Allocation size of 16 bytes is not shown because it matches with 24 bytes.

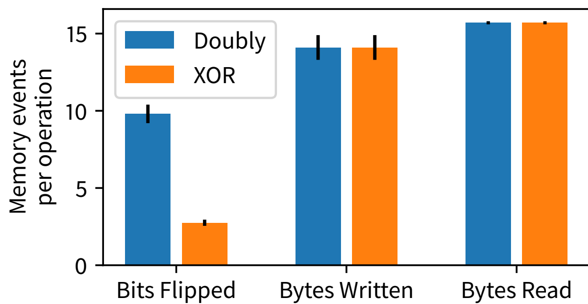

Figure 4: Memory characteristics of of XOR linked lists compared to Doubly-Linked Lists.

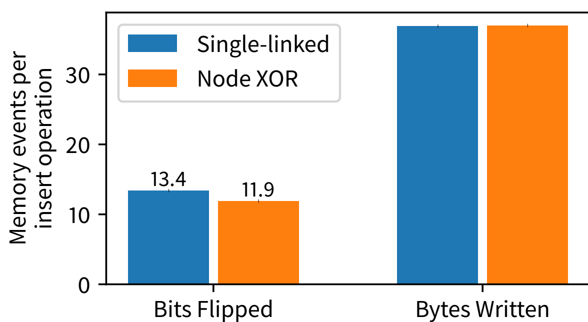

Figure 5: Memory characteristics of XOR hash table variants under Zipfian workload.

inserted. This resulted in a workload where a large number of keys were rarely modified, but a smaller percentage were repeatedly inserted or removed from the hash table.

Figure 5 shows the bits flipped and bytes written by the hash table after 100,000 updates. As expected, the XOR lists saw a reduction in bit flips over the standard, singly-linked list implementation while the number of bytes written were unchanged. We were initially surprised by the relatively low reduction in bit flips (1.13×) considering the relative success of XOR linked lists; however, the common case for hash tables is short chains. We observed that longer chains improve the bit flips savings, but forcing long hash chains is an unrealistic evaluation. Since buckets typically have one element in them, and that element is stored in the table itself, there are few pointers to XOR, meaning the reduction is primarily from indicating a list is valid via the least-significant bit of the next pointer. The bit flips in all variants come primarily from writing the key and value, which comprise 9.3 bit flips per iteration on average. Thus, this data structure had little room for optimization, and the improvements we made were relatively minor—although they still translate directly to power saving and less wear, and are easy to achieve while not affecting code complexity significantly.

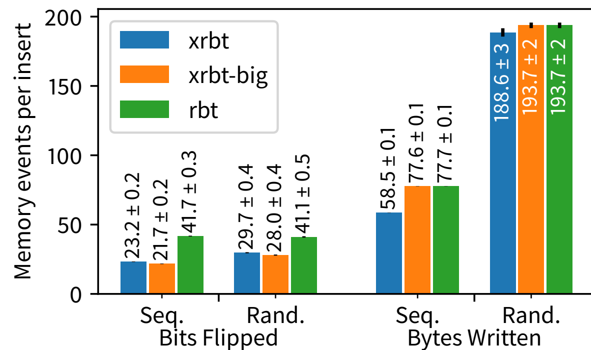

Figure 6: Memory characteristics of XOR red-black trees compared to normal red-black trees.

4.5 XOR Red-Black Trees

Figure 6 shows the memory event characteristics of `xrbt` (our XOR RBT with two pointers, `xleft` and `xright`), `xrbt-big` (our XOR RBT with each node inflated to the size of our normal RBT nodes), and `rbt` (our standard RBT) under sequential and random inserts of one million unique items. Each item comprises an integer key from 0 to one million and a random value. Both `xrbt` and `xrbt-big` cut bit flips by 1.92× (nearly in half) in the case of sequential inserts and by 1.47× in the case of random inserts, a dramatic improvement for a simple implementation change. The small saving in bit flips in `xrbt-big` over `xrbt` is likely due to the allocation size difference as discussed in Section 4.2.

The number of bytes written is also shown in Figure 6. Due to the cache absorbing writes, `xrbt-big` and `rbt` write the same number of bytes to memory in all cases, even though `rbt` writes more pointers during its operation. We can also see a case where the number of writes was not correlated with the number of bits flipped, since `xrbt` writes fewer bytes but flips more bits than `xrbt-big`.

We did not implement and test delete operation in our red-black trees because the algorithm is similar to insert in that its balancing algorithm is tail-recursive and merely recolors or rotates the tree a bounded number of times. Since the necessary functions to implement this algorithm are present in all variations, it is certainly possible to implement, and we expect the results to be similar between them.

4.6 Cache Effects

Although it is easy to exceed the size of the L1 cache during normal operation of large data structures at scale, larger caches may have more of an effect on the frequency of writes to memory. Of course, a persistent data structure which issues cache-line writebacks or uses write-through caching by-

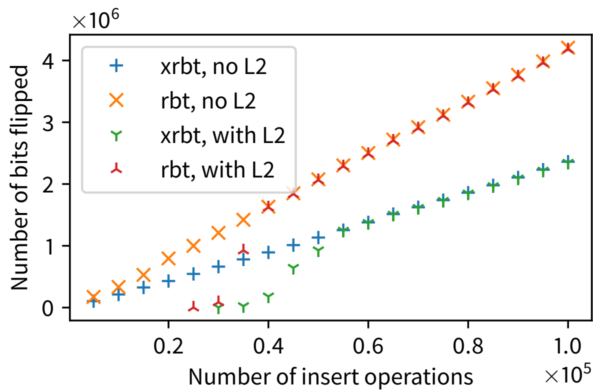

Figure 7: Bits flipped by xrbt and rbt over a varying number of sequential inserts, with and without the L2 cache present.

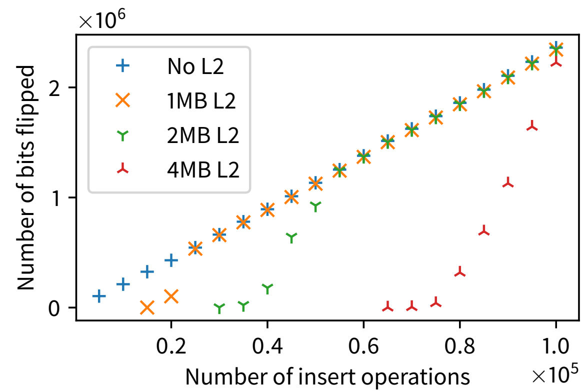

Figure 8: Bits flipped by xrbt over a varying number of sequential inserts, with different sizes for L2.

passes this by causing *all* writes to go to memory[3], but it is still worth studying the effects of larger write-back caches on bit flips. They may absorb specific writes that have higher than average flips, or they may cause coalescing even for persistent data structures worrying about consistency.

We studied cache effects in two ways—how the mere presence of a layer-2 cache affects the data structures we studied and how varying the size of that cache affects them. Figure 7 shows `xrbt` compared with `rbt`, with and without L2. The effect of L2 is limited as the operations scale, with the bit flips for both data structures reaching a steady, linear increase once L2 is saturated. The bit flips per operation for both data structures with L2 is the same as without L2 once the saturation point is reached, indicating that while the presence of the cache *delays* bit flips from reaching memory, it does little to reduce them in the long term. Finally, since `xrbt` has fewer bit flips overall and fewer memory writes, it took longer to saturate L2, delaying the effect.

Next, we looked at different L2 sizes, running `xrbt` with no L2, 1MB L2, 2MB L2 (the default), and 4MB L2, as shown in Figure 8. The exact same pattern emerges for each size, delayed by an amount proportional to the cache size. This is to be expected, and it further corroborates our claim that cache size has only short-term effects.

4.7 Manual Instrumentation

While testing data structures on Gem5 was straightforward, if time consuming, more complex structures and programs may be difficult to evaluate, either due to Gem5's relatively limited system call support or due to the extreme slowdown caused by the simulation. Since real hardware does not provide bit flip counting methods, we are left with using in-

[3]Even if this is the case, a full system simulator will give a more accurate picture than manually counting writes, since store ordering and compiler optimizations still affect memory behavior.

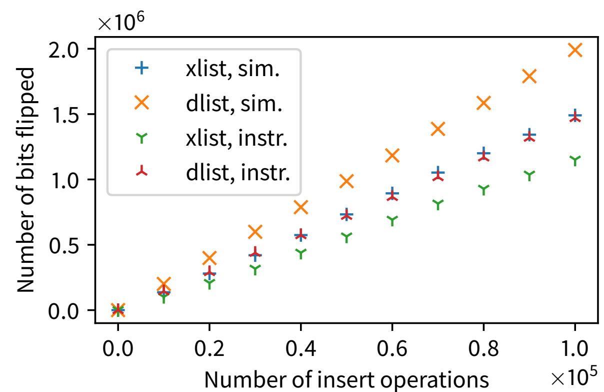

Figure 9: Manual instrumentation for counting bit flips (instr) compared to full-system simulation (sim).

program instrumentation if we want to avoid the Gem5 overhead. However, these results may be less accurate.

To study the accuracy of in-code instrumentation, we manually counted bit flips in the XOR and doubly-linked lists. We did this by replacing all direct data structure writes (*e.g.*, `node->prev = pnode`) with a macro that both did that write and also counted the number of bytes (by looking at the types), and computing the Hamming distance between the original and new values. Totals of each were kept track of and reported at the end of program execution. While not difficult to implement, manual instrumentation adds the possibility of error and increases implementation complexity.

Figure 9 shows the results of manual instrumentation compared to results from Gem5. While accuracy suffered, manual counting was not off by orders of magnitude. It properly represented the relationship between XOR linked lists and doubly-linked lists in terms of bit flips, and it was off by a constant factor across the test. We hypothesize that the dis-

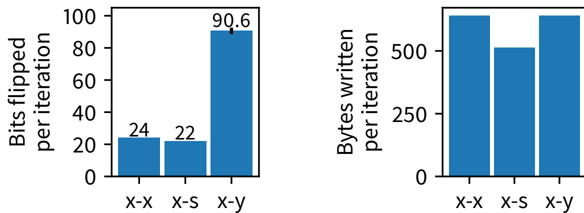

Figure 10: Evaluating memory events for different stack frame layouts.

crepancy arose from the fact that our additional flip counting code affected the write combining and (possibly) the cache utilization. We expect that future system designs could "calibrate" manual instrumentation by running a smaller version of their system on Gem5 to calculate the discrepancy between its counts and theirs, allowing them to more accurately extrapolate the bits flipped in their system using instrumentation. Additionally, one could modify toolchains and debugging tools to automatically emit such instrumentation code during code generation. Manual instrumentation may find its use here for large systems that are too complex or unwieldy to run on Gem5, or as a way to quickly prototype bit flipping optimizations.

4.8 Stack Frames

To study bit flips caused by stack writes, we wrote an assembly program that alternates between two function calls in a tight loop while incrementing several callee-saved registers on x86-64. The loop could call two of three functions—function *x*, which pushed six registers (the callee-save registers on x86_64, including the base pointer) in a given order, *y*, which pushed the registers in a different, given order, and *s*, which pushed only two of the registers, but pushed them to the same locations as function *x*. Our program had three variations: **x-x**, which called function *x* twice, **x-s**, which alternated between functions *x* and *s*, and **x-y**, which alternated between functions *x* and *y*. The x-y variant represents the worst-case scenario of today's methods for register spilling, while x-s demonstrates our suggestion for reducing bit flips. To force the writes to memory, we used `clwb` after the writes to simulate write-through caching or resumable programs.

Figure 10 shows both bit flips (left) and bytes written (right) by all three variants. The x-s and x-x variants have similar behavior in terms of bit flips, which is understandable because they are pushing registers to the same locations within the frame. The x-y variant, however, had 3.8× the number of bit flips compared to x-x and 4.1× the number of bit flips compared to x-s, showing that consistency of frame layout has dramatic impact. Unsurprisingly, x-x and x-y had the same number of bytes written, since they write the same number of registers, while x-s wrote fewer registers. By keeping frame layout consistent, we can reduce bit flips, and the optimization of only pushing the registers needed but to the same locations can further reduce writes as well.

Table 2: Performance of XOR linked lists compared with doubly-linked lists.

Operation	XOR Linked	Doubly-Linked
Insert (ns)	45 ± 1	45 ± 1
Pop (ns)	27 ± 1	28 ± 1
Traverse (ns/node)	2.6 ± 0.1	2.2 ± 0.1

5 Performance Analysis

While bit flip optimization is important, it is less attractive if it produces a large performance cost. We compared our data structures' performance to equivalent "normal" versions not designed to reduce bit flips. Benchmarks were run on an i7-6700K Intel processor at 4GHz, running Linux 4.18, `glibc` 2.28. They were compiled using `gcc` 8.2.1 and linked with GNU `ld` 2.31.1. Unless otherwise stated, programs were linked dynamically and compiled with `O3` optimizations.

XOR Linked Lists The original publication of XOR linked lists found little performance difference between them and normal linked lists [34]; we see the same relationship in our implementation (see Table 2). The only statistically significant difference was seen in traversal, where XOR linked lists have a 1.18× increase in latency; however, both lists average less than three nanosecond-per-node during traversal.

XOR Hash Tables Figure 11 shows the performance of the two hash table variants we developed. We inserted 100,000 keys, followed by lookup and delete. As expected, both variants have nearly identical latencies, with a slowdown of only 1.06× for using XOR lists during lookup.

XOR Red-Black Trees We measured `xrbt`, `xrbt-big`, and `rbt` during 100,000 inserts and lookups, the results of

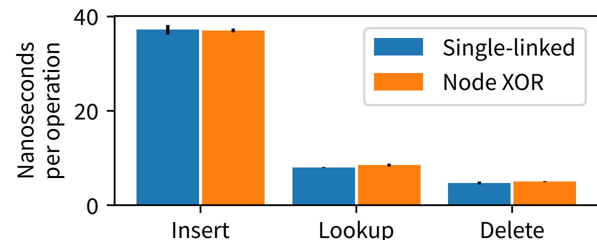

Figure 11: Performance of XOR hash table variants.

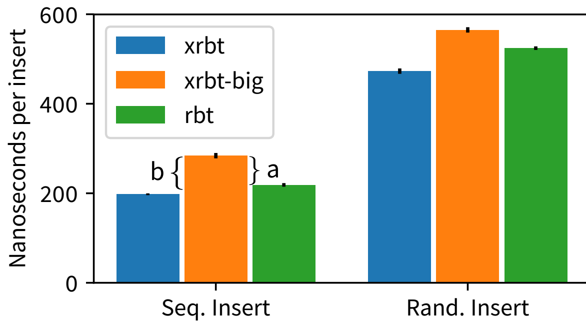

Figure 12: Insert latency for XOR red-black trees compared to normal red-black trees. The label "a" shows the cost of the XORs, while "b" shows the cost of the larger node.

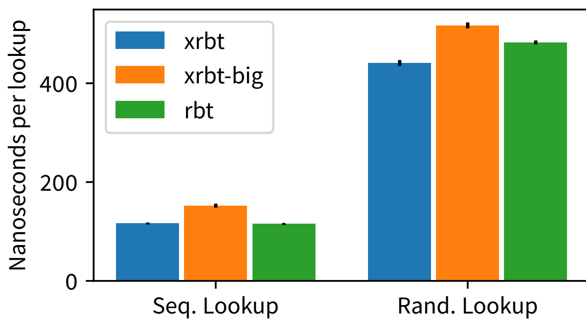

Figure 13: Lookup latency for XOR red-black trees compared to normal red-black trees.

which are shown in Figure 12 and Figure 13. During insert, `xrbt` is actually slightly faster than `rbt`, with `xrbt-big` being slower, indicating that although there is a non-zero cost for the additional XOR operations, it is outweighed by the performance improvement from smaller node size and better cache utilization. The lookup performance shown in Figure 13 demonstrates a similar pattern, although for sequential lookup the overheads are similar enough that there is no significant performance difference between `xrbt` and `rbt`.

6 Discussion

Software Bit Flip Reduction The data structures presented here are both old and new ideas. While not algorithmically different from existing implementations (both `xrbt` and `rbt` use the same, standard red-black tree algorithms), they present a new approach to implementation with optimizations for bit flipping. This has not been sufficiently studied before in the context of software optimization; after all, there is no theoretical advance nor is there an overwhelming practical advantage to these data structures outside of the bit flip reduction, an optimization goal that is new with BNVM. However, keeping this in mind has huge ramifications for data structures in persistent memory and applications for new storage technologies, as it presents a whole new field of study in optimization and practical data structure design. The goal is not performance improvements; instead we strive to prolong the lifetime of expensive memory devices while reducing power use, with at most a minor performance cost. These improvements can be achieved without hardware changes, meaning even savings of 10% ($1.1\times$) or less are worthwhile to implement because savings are cumulative.

These optimizations are not specific to PCM; any memory with a significant read/write disparity and bit-level updates could benefit from this. The energy savings from bit flip optimization will, of course, be technology-dependent, numbers for which will solidify as the technologies are adopted. Our estimates of the linear relationship between flips and power use (Figure 1) indicate that, on PCM, the energy savings will be roughly proportional to the bit flip savings since the difference between read and write energy is so high.

Bit flips can and should be reasoned about directly. Not only is it possible to do so, but the methods presented here are straightforward once this goal is in mind. Furthermore, while reducing writes *can* reduce bit flips, we have confirmed that this is not *always* true. XOR linked lists reduced bit flips without affecting writes, while `xrbt` reduced writes over `xrbt-big` at the cost of increasing bit flips. With stack frames, the biggest reduction in bit flips corresponded with no change in writes, while the reduction of writes was correlated with only a modest bit flip reduction.

The implications are far-reaching, especially when considering novel computation models that include storing program state in BNVM. Writes to the stack also affect bit flips, but these can be dramatically optimized. Compilers can implement standardized stack frame layouts for register spills that save many bit flips while remaining backwards compatible since nothing in these optimizations breaks existing ABIs. Further research is required to better study the effects of stack frame layout in larger programs, and engineering work is needed to build these features into existing toolchains.

Of course, we must be cautious to optimize where it matters. While different allocation sizes reduced bit flips relative to each other, the overall effect was minor compared to the savings gained in other data structures. In fact, the reduction in allocation size from 48 to 40 bytes in `xrbt` actually *increased* bit flips in calls to `malloc`, but this increase is dwarfed by the savings from the XOR pointers. Additionally, the hash table saw a relatively small saving compared to other data structures since it already flipped a minimal number of bits in the average case; red-black trees often do more work during *each* update operation, resulting in a number of pointer updates. Hash tables often do their "rebalancing" during a single rehash operation; perhaps bit flip optimization for hash tables should focus on these operations, something we plan to investigate in the future.

Cache Effects The data structures we tested all had the same behavior—a warm-up period where the cache system absorbed many of the writes followed by a period of proportional increasing of bit flips as the number of update operations increased. We must keep this in-mind when evaluating data structures for bit flips, since we must ensure that the ranges of inputs we test reach the expected scale for our data structures, or we may be blind to its true behavior. The cache size affects this, of course, but it does so in a predictable way in the case of `xrbt`, with only the warm-up period being extended by an amount proportional to cache size. Of course, the behaviour may be heavily dependent on write patterns. Thus, we recommend further experiments and that system designers take caches into account when evaluating bit flip behaviour of their systems.

The cache additionally affects the read amplification seen in XOR linked lists, wherein the XOR linked list implementation issues more reads than a doubly-linked list implementation. However, the reads that make it to memory are the same between the two, indicating that those extra reads are always in-cache. The resultant write reduction and bit flip reduction is well worth the cost since a read from cache is significantly cheaper than a write to memory.

7 Future Work

Although we covered a range of different data structures, there are many more used in storage systems that should be examined, such as B-trees [1] and LSM-trees [27], both to understand their bit flipping behavior as compared to other data structures and to examine for potential optimizations. In addition to data structures, different algorithms such as sorting can be evaluated for bit flips. Though this may come down to data movement minimization, there may be optimizations in locality that could affect bit flips.

While data structure and algorithm evaluation can provide system designers with insights for how to reduce bit flips, examining bit flips in a large system, including one that properly implements consistency and our suggested stack frame modifications (perhaps through compiler modification), would be worthwhile. There are a number of BNVM-based key-value stores [37]; comparing them for bit flips could demonstrate the benefits of some designs over others.

Studying bit flips directly is a good metric for understanding power consumption and wear, but a better understanding through the evaluation of real BNVM would be illuminating. The power study discussed earlier was derived from a number of research papers that give approximate numbers or estimates. On a real system, we could *measure* power consumption, and cooperation with vendors may enable accurate studies of wear caused by bit flips. Additionally, some technologies (*e.g.*, PCM) have a disparity between writing a 1 or a 0, something that could be leveraged by software (in cooperation with hardware) to further optimize power use.

8 Conclusion

The pressures from new storage hardware trends compel us to explore new optimization goals as BNVM becomes more common as a persistent store; the read/write asymmetry of BNVM must be addressed by reducing bit flips. As we showed, the number of raw writes is not always a faithful proxy for the number of bit flips, so simple techniques that minimize writes overall may be ineffective. At the OS level, we can reconsider memory allocator design to minimize bit flips as pointers are written. Different data structures use and write pointers in different ways, leading to different tradeoffs for data structures when considering BNVM applications. At the compiler level, we show that careful layout of stack frames can have a significant impact on bit flips during program operation. Since it can be challenging to reason directly about how application-level writes translate to raw writes due to the compiler and caches, more sophisticated profiling tools are needed to help navigate the tradeoffs between performance, consistency, power use, and wear-out.

Most importantly, we demonstrated the value of reasoning at the *application level* about bit flips, reducing bit flips by $1.13-3.56\times$ with minor code changes, no significant increase in complexity, and little performance loss. The data structures we studied had novel implementations, but were *algorithmically* the same as their standard implementations; yet we still saw dramatic improvements with little effort. This indicates that reasoning about bit flips in software can yield significant improvements over in-hardware solutions and opens the door for additional research at a variety of levels of the stack for bit flip reduction. These techniques translate directly to power saving and lifetime improvements, both important optimizations for early adoption of new storage trends that will have lasting impact on systems, applications, and hardware.

Availability

Source code, scripts, Gem5 bit flip patch, and raw results are available at `https://gitlab.soe.ucsc.edu/gitlab/crss/opensource-bitflipping-fast19`.

Acknowledgments

This research was supported in part by the National Science Foundation grant number IIP-1266400 and by the industrial partners of the Center for Research in Storage Systems. The authors additionally thank the members of the Storage Systems Research Center for their support and feedback. We would like to extend our gratitude to our paper shepherd, Sam H. Noh, and the anonymous reviewers for their feedback and assistance.

References

[1] R. Bayer and E. McCreight. Organization and maintenance of large ordered indices. In *Proceedings of the 1970 ACM SIGFIDET (Now SIGMOD) Workshop on Data Description, Access and Control*, SIGFIDET '70, pages 107–141, New York, NY, USA, 1970. ACM.

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

[3] N. Binkert, B. Beckmann, G. Black, S. K. Reinhardt, A. Saidi, A. Basu, J. Hestness, D. R. Hower, T. Krishna, S. Sardashti, R. Sen, K. Sewell, M. Shoaib, N. Vaish, M. D. Hill, and D. A. Wood. The Gem5 simulator. *SIGARCH Computer Architecture News*, 39(2):1–7, Aug. 2011.

[4] D. Bittman, M. Gray, J. Raizes, S. Mukhopadhyay, M. Bryson, P. Alvaro, D. D. E. Long, and E. L. Miller. Designing data structures to minimize bit flips on NVM. In *Proceedings of the 7th IEEE Non-Volatile Memory Systems and Applications Symposium (NVMSA 2018)*, Aug. 2018.

[5] L. Breslau, P. Cao, L. Fan, G. Phillips, and S. Shenker. Web caching and Zipf-like distributions: evidence and implications. In *In Proceedings of Conference on Computer Communications, IEEE INFOCOM '99*, volume 1, pages 126–134, March 1999.

[6] G. W. Burr, B. N. Kurdi, J. C. Scott, C. H. Lam, K. Gopalakrishnan, and R. S. Shenoy. Overview of candidate device technologies for storage-class memory. *IBM Journal of Research and Development*, 52(4/5):449–464, July 2008.

[7] S. Chen, P. B. Gibbons, and S. Nath. Rethinking database algorithms for phase change memory. In *Proceedings of the 5th Biennial Conference on Innovative Data Systems Research*, pages 21–31, January 2011.

[8] S. Cho and H. Lee. Flip-N-Write: a simple deterministic technique to improve PRAM write performance, energy and endurance. In *Proceedings of the 42nd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pages 347–357. ACM, 2009.

[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. In *Proceedings of the 16th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS '11)*, pages 105–118, Mar. 2011.

[10] J. Colgrove, J. D. Davis, J. Hayes, E. L. Miller, C. Sandvig, R. Sears, A. Tamches, N. Vachharajani, and F. Wang. Purity: Building fast, highly-available enterprise flash storage from commodity components. In *Proceedings of SIGMOD 2015*, June 2015.

[11] J. Condit, E. B. Nightingale, C. Frost, E. Ipek, B. Lee, D. Burger, and D. Coetzee. Better I/O through byte-addressable, persistent memory. In *Proceedings of the 22nd ACM Symposium on Operating Systems Principles (SOSP '09)*, pages 133–146, Oct. 2009.

[12] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein. *Introduction to Algorithms, Third Edition*. The MIT Press, 3rd edition, 2009.

[13] G. Dhiman, R. Ayoub, and T. Rosing. PDRAM: A hybrid PRAM and DRAM main memory system. In *Proceedings of the 46th IEEE Design Automation Conference (DAC '09)*, pages 664–669. IEEE, 2009.

[14] X. Dong, C. Xu, Y. Xie, and N. P. Jouppi. NVSim: a circuit-level performance, energy, and area model for emerging nonvolatile memory. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 31(7), July 2012.

[15] G. Fox, F. Chu, and T. Davenport. Current and future ferroelectric nonvolatile memory technology. *Journal of Vacuum Science & Technology B: Microelectronics and Nanometer Structures Processing, Measurement, and Phenomena*, 19(5):1967–1971, 2001.

[16] K. M. Greenan and E. L. Miller. PRIMS: Making NVRAM suitable for extremely reliable storage. In *Proceedings of the Third Workshop on Hot Topics in System Dependability (HotDep '07)*, June 2007.

[17] M. Han, Y. Han, S. W. Kim, H. Lee, and I. Park. Content-aware bit shuffling for maximizing PCM endurance. *ACM Transactions on Design Automation of Electronic Systems*, 22(3):48:1–48:26, May 2017.

[18] Intel Newsroom. Intel and Micron produce breakthrough memory technology, 2015. `http://newsroom.intel.com/news-releases/intel-and-micron-produce-breakthrough-memory-technology/`; Accessed 2019-01-10.

[19] A. N. Jacobvitz, R. Calderbank, and D. J. Sorin. Coset coding to extend the lifetime of memory. In *Proceedings of High Performance Computer Architecture (HPCA '13)*, pages 222–233. IEEE, 2013.

[20] H. Jayakumar, K. Lee, W. S. Lee, A. Raha, Y. Kim, and V. Raghunathan. Powering the internet of things. In *Proceedings of the International Symposium on Low Power Electronics and Design (ISLPED '14)*, pages 375–380, New York, NY, USA, 2014. ACM.

[21] H. Jayakumar, A. Raha, and V. Raghunathan. Quickrecall: A low overhead HW/SW approach for enabling computations across power cycles in transiently powered computers. In *Proceedings of the 27th International Conference on VLSI Design and 13th International Conference on Embedded Systems*, pages 330–335. IEEE, 2014.

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

[23] E. Kohler. Left-leaning red-black trees considered harmful. `http://read.seas.harvard.edu/~kohler/notes/llrb.html`. Accessed 2018-09-22.

[24] B. C. Lee, E. Ipek, O. Mutlu, and D. Burger. Architecting phase change memory as a scalable dram alternative. In *ACM SIGARCH Computer Architecture News*, volume 37, pages 2–13. ACM, 2009.

[25] J. Meza, Y. Luo, S. Khan, J. Zhao, Y. Xie, and O. Mutlu. A case for efficient hardware/software cooperative management of storage and memory. In *5th Workshop on Energy-Efficient Design (WEED '13)*, June 2013.

[26] D. Narayanan and O. Hodson. Whole-system persistence. In *Proceedings of the 17th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS '12)*, pages 401–500, Mar. 2012.

[27] P. O'Neil, E. Cheng, D. Gawlick, and E. O'Neil. The log-structured merge-tree (LSM-tree). *Acta Informatica*, 33(4):351–385, June 1996.

[28] M. K. Qureshi, J. Karidis, M. Franceschini, V. Srinivasan, L. Lastras, and B. Abali. Enhancing lifetime and security of PCM-based main memory with start-gap wear leveling. In *Proceedings of the 42nd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, 2009.

[29] M. K. Qureshi, V. Srinivasan, and J. A. Rivers. Scalable high performance main memory system using phase-change memory technology. In *Proceedings of the 36th annual international symposium on Computer architecture (ICSA '09)*, pages 24–33, 2009.

[30] S. Raoux, G. W. Burr, M. J. Breitwisch, C. T. Rettner, Y.-C. Chen, R. M. Shelby, M. Salinga, D. Krebs, S.-H. Chen, H.-L. Lung, and C. H. Lam. Phase-change random access memory: A scalable technology. *IBM Journal of Research and Development*, 52(4/5):465–480, July 2008.

[31] R. Sedgewick and L. J. Guibas. A dichromatic framework for balanced trees. In *Proceedings of the 19th Annual Symposium on Foundations of Computer Science (SFCS '78)*, volume 00, pages 8–21, 10 1978.

[32] S. M. Seyedzadeh, R. Maddah, D. Kline, A. K. Jones, and R. Melhem. Improving bit flip reduction for biased and random data. *IEEE Transactions on Computers*, 65(11):3345–3356, 2016.

[33] S.-S. Sheu et al. Fast-write resistive RAM (RRAM) for embedded applications. *IEEE Design & Test of Computers*, pages 64–71, Jan. 2011.

[34] P. Sinha. A memory-efficient doubly linked list. *Linux Journal*, 129, 2004. `http://www.linuxjournal.com/article/6828`.

[35] D. B. Strukov, G. S. Snider, D. R. Stewart, and R. S. Williams. The missing memristor found. *Nature*, 453:80–83, May 2008.

[36] H. Volos, A. Jaan Tack, and M. M. Swift. Mnemosyne: Lightweight persistent memory. In *Proceedings of the 16th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS '11)*, Mar. 2011.

[37] F. Xia, D. Jiang, J. Xiong, and N. Sun. HiKV: A hybrid index key-value store for DRAM-NVM memory systems. In *2017 USENIX Annual Technical Conference (USENIX ATC 17)*, pages 349–362, Santa Clara, CA, 2017. USENIX Association.

[38] J. Xu and S. Swanson. NOVA: a log-structured file system for hybrid volatile/non-volatile main memories. In *Proceedings of the 14th USENIX Conference on File and Storage Technologies (FAST '16)*, Feb. 2016.

[39] B. D. Yang, J. E. Lee, J. S. Kim, J. Cho, S. Y. Lee, and B. G. Yu. A low power phase-change random access memory using a data-comparison write scheme. In *Proceedings of IEEE International Symposium on Circuits and Systems*, May 2007.

[40] P. Zhou, B. Zhao, J. Yang, and Y. Zhang. A durable and energy efficient main memory using phase change memory technology. In *Proceedings of the 36th International Symposium on Computer Architecture*, pages 14–23, 2009.