The Flipside

A Bit Flip Saved Is Power and Lifetime Earned

DANIEL BITTMAN, PETER ALVARO, DARRELL D. E. LONG, AND ETHAN L. MILLER

Daniel Bittman is a PhD candidate at the University of California, Santa Cruz, studying under Ethan Miller, Darrell Long, and Peter Alvaro. His research interests include operating systems, non-volatile memory, concurrency, and systems security. He is currently working on developing operating system techniques for improving the use of persistent memory, reducing power and wear for persistent memory, and studying non determinism in distributed systems. dbittman@ucsc.edu

Peter Alvaro is an Assistant Professor of Computer Science at the University of California, Santa Cruz, where he leads the Disorderly Labs research group (disorderlylabs.github.io). His research focuses on using datacentric languages and analysis techniques to build and reason about data-intensive distributed systems in order to make them scalable, predictable, and robust to the failures and nondeterminism endemic to large-scale distribution. Peter earned his PhD at the University of California, Berkeley, where he studied with Joseph M. Hellerstein. He is a recipient of the NSF CAREER Award and the Facebook Research Award. palvaro@ucsc.edu

We have an opportunity to rethink, from scratch, the design of our data structures. New byte-addressable non-volatile memory (BNVM) technologies promise the construction of systems with large persistent memories, potentially improving reliability and performance. With these technologies come new characteristics that deviate from those of flash and spinning disk—and with new characteristics come new optimization goals. In particular, the read/write cost disparity and fine granularity of updates allows us to save power and wear by reducing the bits flipped during writes to memory. Targeting *these* optimizations by formulating new data structure design and implementation strategies instead of relying on existing ideas will be vital for BNVM technology to reach its full potential. We modified a full-system simulator to count bit flips during program operation, opening the door for future research to design, construct, and evaluate data structures for these new goals.

New Optimization Targets

As byte-addressable non-volatile memories (BNVMs) become common, 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. While still worthwhile, write-reduction fails to take advantage of a key optimization made by the memory controller in those non-volatile memories.

Some technologies, including phase-change memory (PCM), have a significant disparity between the cost—be it power, time, or wear—of reading a cell and writing a cell. When these technologies also support fine granularity updates, they can make use of a clever optimization: checking if a cell already contains the new, target value [10] instead of blindly overwriting it. Such an optimization yields a change in perspective on what is costly when operating on BNVM; it is not the writes themselves so much as *bits flipped* during the writes. In PCM, for example, changing a cell consumes 15.7–22.5x more power than reading a cell [5, 6] in addition to causing wear-out (a significant problem for PCM as it has limited endurance).

Therefore, system designers ought to consider the effects of bit flips when building systems for BNVM, both when considering the target use-case for the hardware and picking an appropriate combination of BNVM and DRAM, but also when considering the design of the *software* that issues the writes in the first place. To get a sense of how write patterns might affect power consumption, Figure 1 shows a model of power consumption of DRAM and PCM under a varying number of bit flips per second. The power consumption of PCM depends heavily on the bit flips per second, while DRAM's power consumption is relatively independent. We also see that DRAM requires a high "maintenance" power (due to the need to refresh), whereas PCM does not. The choice to use a particular technology could depend,

The Flipside: A Bit Flip Saved Is Power and Lifetime Earned

Dr. Darrell D. E. Long is Distinguished Professor of Engineering at the University of California, Santa Cruz. He holds the Kumar Malavalli Endowed Chair of Storage Systems Research and is Director Emeritus of the Storage Systems Research Center. His broad research interests include many areas of mathematics and science, and in the area of computer science include data storage systems, operating systems, distributed computing, reliability and fault tolerance, and computer security. He is currently Editor-in-Chief of the *Letters of the Computer Society*, and Editor-in-Chief Emeritus of the *ACM Transactions on Storage*.
darrell@ucsc.edu

Ethan L. Miller is a Professor of Computer Science in the Jack Baskin School of Engineering, where he holds the Veritas Presidential Chair in Storage. He is the Director of the NSF I/UCRC Center for Research in Storage Systems and the Director of the Storage Systems Research Center. He was a member of the RAID project at UC Berkeley, where he did his PhD on a decentralized parallel file system for high-end scientific computing. His current research interests include archival storage systems, file systems for storage-class memories, scalable view-based metadata management, and issues in reliability, scalability, and security, both for short-term and archival storage.
elm@ucsc.edu

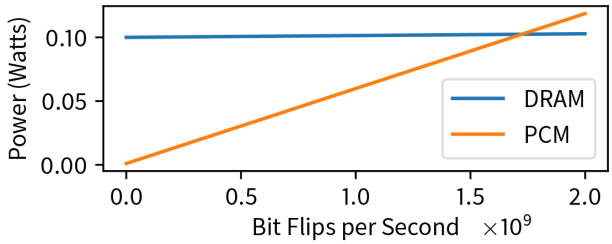

Figure 1: Power use of 1 GB devices as a function of flips per second [2]. DRAM's power consumption is largely proportional to memory size whereas PCM's is largely proportional to bit flip rate.

therefore, on the expected write patterns to memory, since there is a crossover point on the graph. This is particularly important for Internet of Things (IoT) devices, where power consumption and conservation is critical.

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, we can make use of hardware techniques such as row shifting [11] to *spread out* the "hot spots," thus translating a reduction of bit flips in part of a word to an average reduction across the entire word.

Optimizing software for a novel optimization goal such as bit flipping requires rethinking some core design ideas. The need to incorporate an underlying technology's characteristics into software is not new; indeed, it has been seen with block-oriented sequential access data structures for disk and trading writes for random reads in flash. For BNVM, research has focused on reducing writes while often ignoring the importance of the bits flipped by the writes. Prior work that looks at the bit flips directly either merely considers hardware solutions [4, 7, 8] or suggests that write reduction is a good analog for bit flip reduction [3]. While hardware techniques are certainly a more *general* solution to the problem, they lack the semantic knowledge available to software to improve bit flip reduction. Similarly, write reduction by itself *may* reduce bit flips, but we have found that this is not always the case [1, 2].

Once we accept that bit flips play a significant role in the power consumption and wear of BNVM technologies, we must ask the questions, what changes can we make to software to improve bit flip reduction, and how do we measure our work? We approached this problem by focusing on optimizing *data structures* for bit flip reduction, since data organization plays a large role in the writes that make it to memory. Although 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.

To show that bit flips can be optimized for, and to explore several techniques we thought of to do so, we designed and built several data structures and evaluated them by counting their bit flips and writes at the memory controller, as well as measuring the performance of each. While our earlier work [2] focused on manual instrumentation of code to count bit flips, we decided to use a full-system simulator (Gem5) to count bit flips so we could take into account caching layers and compiler optimizations. More details for our current work, including more experiments, data structures, and bit flip reduction techniques, are available [1].

Pointer Distance in Data Structures

Data structures are often made up of a significant number of pointers. Take the doubly linked list, for instance: each node contains two pointers, one forward and one back. A clever technique to reduce the memory footprint is to XOR the pointers together, storing pointer *distance* instead of absolute addresses. This is known as an XOR linked list [9]. The program can still traverse the list in either direction with two adjacent pointers, but the overhead of the node is halved. When XOR linked lists were originally proposed, there wasn't much of an advantage to using them beyond a modest memory saving. However, they reduce bit flips by not only cutting the number of writes in half but also zeroing-out many of the bits contained within a standard pointer value.

We can extend XOR linked lists into the domain of indexing structures by reapplying the pointer distance technique to binary search trees. Binary search trees are commonly used for data indexing and support range queries, and they allow efficient lookup and modification, as long as they are balanced. In a standard red-black tree (RBT), for example, a node stores a left child pointer, a right child pointer, and a parent pointer. We can instead store "xleft" and "xright" by XORing the left child pointer with the parent pointer and the right child pointer with the parent pointer, respectively. This reduces the size of the node from three pointers to two pointers while still allowing easy up and down traversal (and thus keeping the benefits of the three-pointer approach), and saves bit flips for the same reason as the XOR linked list.

Tree traversal and update operations in the XOR red-black tree are largely the same as in a standard red-black tree implementation. However, since we are storing XORs of pointers and not the pointers themselves, some additional effort from the programmer is required to "decode" the stored values into a "true" address. Additionally, while traversal *down* the tree is straightforward (given a parent node pointer and a current node's xleft value, we can traverse to the left child by XORing together the parent pointer and the xleft value), traversing up the tree is more difficult. Given a current node and one of its children, the traversal algorithm needs to know *which* child it is. Fortunately, we can make use of the node ordering of a binary search tree to determine which child we have, thus enabling upward traversal.

Results and Discussion

We implemented our XOR red-black tree design alongside a traditional red-black tree and evaluated both under a full-system simulator—Gem5—which simulates the cache hierarchy and allowed us to collect bit flip numbers on unmodified code, thus more faithfully representing the behavior of a system. We found that the programmer overhead required for dealing with pointer distance was not high, especially when considering the abundance of tooling that could be used and harnessed to make

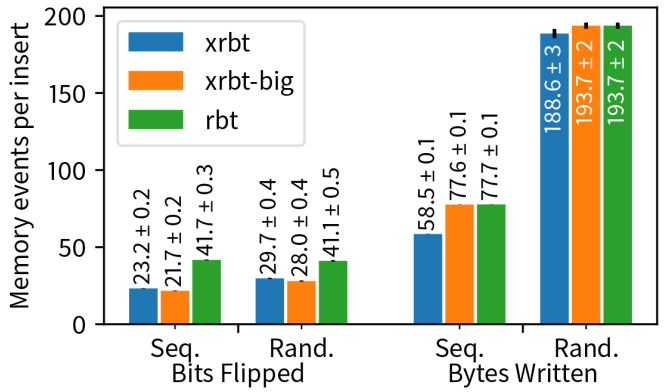

Figure 2: Memory characteristics of XOR red-black trees compared to normal red-black trees (lower is better). The XOR technique significantly reduces bit flips.

debugging easier. The patch to Gem5 to enable bit flip counting at the memory controller was similarly straightforward, but opens up a significant amount of evaluation and research that can be done to evaluate the bit flipping characteristics of existing systems and data structure design (https://gitlab.soe.ucsc.edu/gitlab/crss/opensource-bitflipping-fast19).

Figure 2 shows the bit flips and bytes written of `xrbt` (our XOR RBT implementation) and `rbt` (our standard RBT) under sequential and random inserts of one million unique items. We also evaluated `xrbt-big`, which was the same implementation as `xrbt` but with the same node size as `rbt` (to control for node-size in our results). Both `xrbt` and `xrbt-big` cut bit flips by 1.92x (nearly in half) in the case of sequential inserts and by 1.47x in the case of random inserts, a dramatic improvement for a simple implementation change. We can also compare the bytes written, noting that 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.

Because this new optimization target adds additional overhead, we wanted to get an idea of the performance impact of our changes. Figure 3 shows the latency per insert operation for all three variants for both sequential and random insert. Somewhat surprisingly (at first), the `xrbt` is *faster* than `rbt`! But, when looking at `xrbt-big`, this makes some sense. There are two conflicting effects in play: the performance *cost* of doing the extra XOR operations, and the performance *gain* from reducing the size of the node. The interval labeled "a" in Figure 3 is the former, while the interval labeled "b" is the latter. The two nearly cancel out, and we see a similar result for lookup latency.

These results indicate that 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, and they come at little cost to performance and low program-

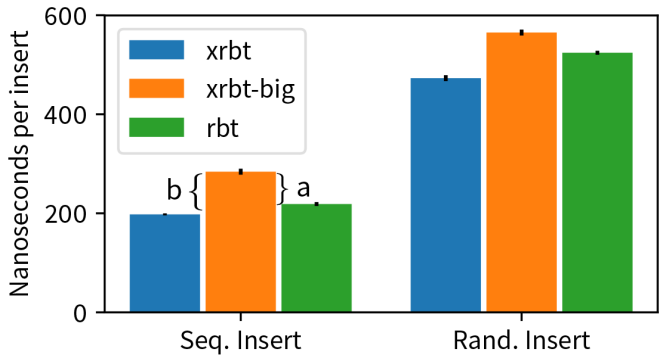

Figure 3: Insert latency for XOR red-black trees compared to normal red-black trees (lower is better). The label "a" shows the cost of the XORs (small), while "b" shows the cost of the larger node.

ming overhead. Furthermore, while reducing writes *can* reduce bit flips, we have confirmed that this is not *always* true—`xrbt` reduced writes over `xrbt-big` at the cost of increasing bit flips.

We can use the results of prior research reporting on power consumption and wear-out of PCM to estimate the effects of our XOR red-black tree. Since PCM power consumption is largely dependent on bit flip rate, we estimate that the power consumption per second of `rbt` and `xrbt` running at full speed are 13mW and 6.6mW, respectively—a ratio of nearly two.

Lifetime is more complex, but a quick calculation taking into account row-shifting and the differences in bytes written by the two variants shows a savings of 1.83x, assuming that the memory controller spreads out writes in larger regions [11]. These savings are estimates, and we may see more savings since potential nonlinearity in power consumption due to heat could improve the power savings from bit flip reduction, and the overall operational power use of controllers may reduce slightly along with the number of writes.

Discussion and Future Research

The data structures presented here emerge from 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. They do little to impact performance, but performance increases are not the direct goal of this work. Instead, these modest changes can gain us a significant reduction in bit flips that corresponds directly to power and wear reductions, a worthwhile effort even if the saving is small (which, in our work, it is not).

The implications are far-reaching when considering the promise of BNVM and the potential for disruption throughout the system stack. This work is merely the beginning, and we hope that there are future bit flip reduction techniques discovered that we have not considered here. By providing a framework that counts bit flips on data structures, we hope to open an avenue into developing more sophisticated profiling tools that help navigate the tradeoffs between performance, consistency, power consumption, and wear-out.

Considering these results in the context of larger systems is important to understanding the overall effect of bit flip reduction. For example, it would be useful to compare existing key-value stores and observe their memory behavior. However, applying the data structures discussed here as a drop-in replacement for data structures in an existing system would sell them short. Since current systems are designed for non-BNVM technologies, they would fail to make basic optimizations and structural changes that one would expect in a BNVM-optimized system *even without* taking bit flips into consideration. A more effective evaluation would be to construct a BNVM-optimized system from scratch, taking into account write reduction, consistency, *and* bit flips, and then compare it to an existing, unmodified system.

There are a number of implementation details in real hardware that might affect bit flip optimizations. While the basic optimization of avoiding unnecessary overwrites would remain, there are several questions that we do not know the answers to when it comes to bit flip reduction on real hardware. First, what is the *actual* power cost? We will need to wait for real hardware to become available to test this. Second, is there a difference between flipping from a 0 to a 1 compared to flipping from a 1 to a 0? If there is, a new contract between hardware and software would need to contain information that ensures software can predict which is cheaper. Third, is there a performance difference between a write that flips few bits compared to many bits? This depends on hardware implementation details, but if there is, it might make the benefits from bit flip reduction even more significant.

Data structures are not the only causes of memory writes, of course. The obvious candidate for targeted bit flip reduction is the data itself, for which we could rely on existing hardware reduction techniques to work in tandem with software techniques. Another significant source of writes is from the program stack, especially when considering the desire for efficient restart that BNVM offers. We evaluated potential backward-compatible ABI modifications [1], but plenty more work can be done to study these modifications in a real compiler or take them further.

Finally, there are many existing data organization techniques that can be evaluated and tweaked for bit flips. Not only data structures, but algorithms too can be evaluated. For example, if one were to sort a collection of items in BNVM, what would be the most efficient sorting algorithm in terms of bit flips? While it is likely one that minimizes the number of moves, this might not always be the case; we saw above that write reduction does not always correlate with bit flip reduction.

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. Reasoning about bit flips should be done at the application level instead of just in hardware to take into account the semantic knowledge of data structure operations, and we cannot get away with simply reducing writes if we strive to reduce power consumption and wear. While hardware techniques apply more broadly, software techniques open the door for significant future research at a variety of levels of the stack. Our work translates 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.

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.

References

[1] D. Bittman, P. Alvaro, D. D. E. Long, and E. L. Miller, "Optimizing Systems for Byte-Addressable NVM by Reducing Bit Flipping," in *Proceedings of the 17th USENIX Conference on File and Storage Technologies (FAST '19)*, February 2019: https://www.usenix.org/system/files/fast19-bittman.pdf.

[2] 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), August 2018.

[3] 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*, January 2011, pp. 21–31.

[4] 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)*, 2009, pp. 347–357.

[5] 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)*, 2009, pp. 664–669.

[6] 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*, vol. 31, no. 7, July 2012.

[7] 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)*, 2013, pp. 222–233.

[8] 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*, vol. 65, no. 11, 2016, pp. 3345–3356.

[9] P. Sinha, "A Memory-Efficient Doubly Linked List," *Linux Journal*, vol. 129, 2004: http://www.linuxjournal.com/article/6828.

[10] 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.

[11] 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*, 2009, pp. 14–23.