NVM Cache with Predictive Allocation

Alexander Rucker, Andrew Bartolo, Christopher Chute
acrucker@stanford.edu, bartolo@stanford.edu, chute@stanford.edu

Abstract

Modern cloud applications, including databases such as MySQL and analytics libraries such as Apache Spark, keep as much of their working sets as possible in DRAM to avoid the penalty of paging to disk. However, when serving potentially millions of queries per second, we need to keep hot cache lines as close to the CPU as possible—preferably in an on-die cache. SRAM caches, though commonplace, are costly in terms of area and power. Emerging nonvolatile memory (NVM) technologies offer an alternative to SRAM with lower area and power requirements, albeit with poor endurance. This paper explores the application benefits of adding either an L2 or L4 CPU cache constructed out of NVM technologies, and shows that an L2 cache can improve performance only if it is relatively fast. Because NVM presents significant endurance concerns, we separately evaluate the effectiveness of several cache management policies in decreasing cache wearout for a simulated cache, including selecting which lines to insert and profiling eviction policies.

I. Introduction

FOR decades, CPUs have exploited the the phenomena of temporal and spatial locality by using on-die caches to improve performance. A hierarchy of caches can typically decrease the Average Memory Access Time (AMAT) by responding to most memory accesses without needing to send the data to main memory. However, a 2015 paper by Google [12] found that, for common cloud workloads like search indexing and database queries, 50% to 60% of a CPU's cycles are spent stalled on caches. The authors speculate that this is due to complicated access patterns within the applications, which are hard for a prefetcher to predict and do not leave much room to exploit instruction-level parallelism. While the size of cloud datasets is growing to match the increasing density of storage [13], the density of CMOS is increasing far more slowly, and with extreme expense and poor power efficiency [21].

Traditionally, CPUs have included a hierarchy of SRAM caches in an attempt to keep recently referenced data close at hand. A recent Intel Xeon Broadwell system which we profiled contained 16 cores on-die, each of which maintained its own 32KB L1 cache and 256KB L2 cache. All 16 cores shared a 40MB L3 cache [3]. For comparison, a single 24-bit 4K image is 24MiB in size, and machine learning workloads must frequently reference weight and embedding matrices in the tens or hundreds of megabytes. Furthermore, Intel's recent decision to support per-process last level cache (LLC) partitioning [18] leaves applications with even less aggregate LLC. For this paper, we therefore posit that computation on modern datasets would benefit from much larger last-level CPU caches.

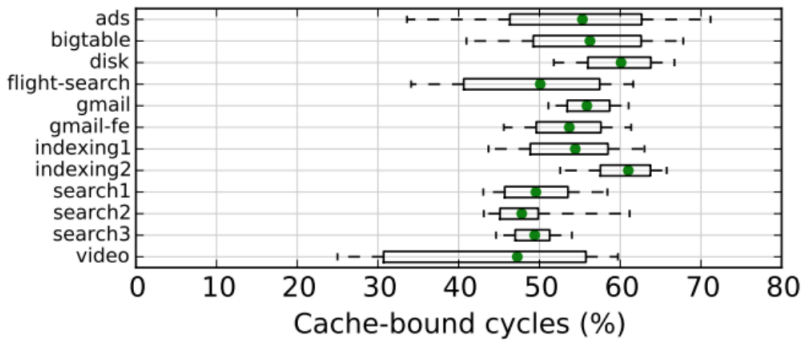

Fig. 1: Cycles spent waiting on cache results [12].

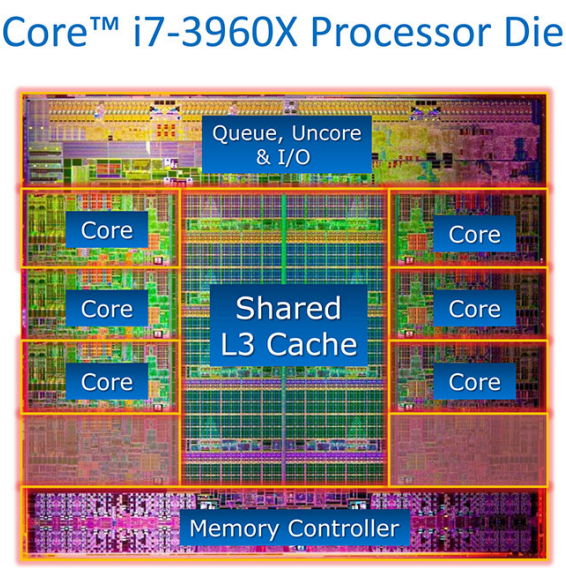

Fig. 2: Modern CPU die shot [2].

We start by evaluating the potential for impact of faster caches. We compute potential configurations for two scenarios: replacing the L2 with a nonvolatile memory (NVM) cache, and adding an NVM cache between the L3 and main memory. We elected not to evaluate replacing the L3 entirely with NVM because effective multi-core programming relies on using the L3 as a high-bandwidth communication channel between cores, and a NVM L3 would either sustain overly high latency and wearout penalties from this programming paradigm or force all inter-process communication to happen through main memory.

We find that replacing the L2 with a much larger, but slightly slower, cache yields significant improvements for several benchmarks, and a slight slowdown for other benchmarks that fit entirely in the L2 cache, but the L2 cache latency cannot increase too much or the L2 hits become too slow. We were unable to elucidate a difference between including an RRAM L4 and not including one; we believe this is due to having a large, high-performance L3, not accurately simulating the load to memory, and having the L4 cache not taking much area away from the L3.

After demonstrating that there is potentially a speedup to be gained from the use of NVM caches, we then investigate strategies for reducing the number of updates to these caches. We evaluate several strategies against a normally sized L2 cache, which allows a comparison between the different update strategies. We find that randomly inserting cache lines actually performs better than any other update strategy, and allows the user to trade off the amount of wear that the cache sustains while not changing MPKI too significantly.

II. Constructing a Much Larger LLC

SRAM is a traditionally attractive cache technology for for several reasons. Standard 6T SRAM cells are built from cross-coupled inverters, and are thus a natural fit for CMOS integration. SRAM read and write latencies are on the order of one nanosecond for small arrays [5]. However, for constructing LLCs in the hundreds of megabytes to gigabytes, SRAM is prohibitively area-intensive [5], [16]. Die shots of any modern high-performance CPU reveal the vast amount of real estate that must be dedicated solely to L3 SRAM (see Fig. 2).

A. RRAM

In the past several years, a handful of new memory technologies have emerged to fill the gap between SRAM and DRAM. These memories are generally nonvolatile, and with various latency, area, and

Technology	Read latency (ns)	Write latency (ns)	Endurance (num. writes)	Area (F^2)
DRAM	50	50	$> 10^{15}$	6
RRAM	10	50	10^{11}	4
SRAM	1	1	$> 10^{15}$	125

TABLE I: A summary of LLC technology tradeoffs

L2 Size (B)	256k	2M	256k
L2 Associativity	8	8	8
L2 Hit Latency (ns)	3	12	3
L3 Size (B)	40M	40M	38M
L3 Associativity	20	20	19
L3 Hit Latency (ns)	21	21	21
L4 Size (B)	-	-	1G
L4 Associativity	-	-	16
L4 Hit Latency (ns)	-	-	40
Memory Latency (ns)	61	61	61

TABLE II: Simulation parameters for the three configurations. The NVM L2 has on-chip tags using space from the L2, and the NVM L4 has off-chip tags and on-chip status bits using space from the L3.

endurance characteristics. For this paper, we focus on resistive RAM (RRAM, also known as ReRAM), a moderately mature technology with increasing industry interest. RRAM stores bits by varying the resistance across a dielectric using ion movement [8].

Compared to SRAM, which has a typical cell size of 125 F^2 (where F is the minimum feature size for some process), RRAM cell sizes are on the order of 4 F^2 [5], [16]. Thus, at first pass, RRAM arrays can be made approximately 31 times denser than SRAM. However, owing to its lower required fabrication temperature compared to CMOS SRAM, RRAM may also be arranged in a 3D crosspoint configuration [8]. This is similar to Intel's 3D XPoint technology, but using memristors (RRAM) instead of phase-change memory (PCM). In the future, this arrangement may significantly increase RRAM density, by allowing RRAM to be fabricated over the top of silicon in the back-end of semiconductor fabrication.

B. Sizing the cache

We used the aforementioned Broadwell CPU from [3] as a base to explore adding our RRAM LLC. The system has 16 cores, 32KB of L1 and 256K of L2 per core, and a shared 40MiB L3 [3]. Our initial benchmark, a cloud TPC-H workload for MySQL, featured working sets around 7-8GiB in size. For economic reasons, fabricating this much RRAM on-die is infeasible.

However, we believe an amount of on-die RRAM in the range of 256MiB to 1GiB is feasible. As RRAM is roughly 31 times denser than SRAM, a 1GiB array would occupy an effective $\frac{1024}{31} = 33$MiB of SRAM-area. Our Xeon has 40MiB of existing L3, which could potentially be replaced entirely by 1GiB of RRAM, or augmented in an L4 cache "on-top" of the L3 via 2.5D integration or an interposer.

We considered 16- and 32-way set associativity for the LLC. Assuming 8-byte words and 64-byte lines, each address for a 1GiB direct-mapped cache would correspond to a 34-bit tag, 24-bit index, 3-bit word offset into line, and 3-bit byte offset into word. Increasing cache size by a power of two adds one to the number of index bits and subtracts one from the number of tag bits. Increasing set associativity does the opposite. In our cache index table, in addition to tags, we must also maintain one valid and one dirty bit. Note that x86-64 only really implements a 48-bit virtual address space, so we may be able to reduce the size of each tag by 16 bits.

After the question of cache size is answered, the question of cache latency remains. The latency of a cache at a given level is the result of three factors: it takes time to send the request to the cache, it takes time for the cache to check the tags, and it takes time for the cache to read the data. Table II shows the measured dual-random-read latency of the Broadwell reference system, measured using TinyMemBench[6].

	5×10^7 Writes	10^{11} Writes
256 kiB Private L2 (10000 UPMI)	1.9 hours	5.3 months
256 kiB Private L2 (1000 UPMI)	19 hours	4.3 years
1 GiB Shared L4 (10 UPMI)	79 days	433 years

TABLE III: Expected cache lifetimes assuming 1 IPC, varying **U**pdates **P**er **M**illion **I**nstructions, 3 GHz, 64 byte cache lines, and 16 cores.

Benchmark	Workload	Reason for Inclusion
MySQL	TPC-H Queries 1, 5, 10, 16, 20	Transaction processing is an important cloud workload.
glucose	newton.5.1.i.smt2-cvc4.cnf	Mixture of read-write and read-mostly data structures.
firefox	Acid Test 3, Acid Test 2, reddit.com	Complicated application and an unstructured workload.
libreoffice	TPC-H spec docx to pdf conversion	Java application, moderately complicated.
graph500 BFS	2^{24} node graph search	Poor cache performance, graph analytics.
gcc	Compile C++ cache simulator with -O3	Random cache accesses, many different analysis steps

TABLE IV: Summary of included benchmarks

C. Endurance: A critical question

One of the key problems for RRAM caches is that of endurance. Because RRAM is still a new technology, there are a wide variety of endurance specifications reported in the literature, ranging from 5×10^7 to 10^{11} [24]. In our paper and in general, "endurance" refers to the number of *writes* a device block can sustain before becoming unreliable. Our methods seek to reduce wear on the NVM cache by only including lines which will be read frequently, but updated (written to) infrequently. The expected lifetimes for several cache configurations are shown in Table III.

III. Methodology

A. Benchmark Selection

For several reasons, we were unable to benchmark our proposed schemes against cloud applications. One of the key problems that we encountered is that cloud applications, by their nature, are extremely I/O oriented. This causes them to have a heavy dependence on the operating system, network architecture, and I/O devices such as NICs. We examined `memcached` and `redis`, and found that user-level cycles only made up about 20% of the total cycles. We also saw that even if we disregarded the cache impact of the system code itself, and only looked at the user code, we could not begin to approximate an accurate trace of the entire workload. This is because applications pass pointers directly to the kernel to eliminate copying overhead.

Instead of using entirely cloud-based benchmarks, we selected a variety of applications that have interesting instruction and data access patterns. These are shown in Table IV. We allowed the applications to run for 20 billion instructions before tracing them, except libreoffice and gcc. These were fast-forwarded up for 1 billion instructions due to their shorter length. We then modified DynamoRIO to dump the L1 instruction and data cache misses and evicts to a file, instead of logging every instruction and memory access. This provided several orders of magnitude reduction in the amount of data requiring processing, and allowed us to ensure that all the cache insertion algorithms were run against the same data.

We then used the first 1 million L1 misses to warm up the L2 caches, and simulated the L2 caches for 50 million L1 misses. We attempted to simulate the L4 cache endurance limitations, but we were unable to gather a sufficient quantity of L3 cache misses to enable accurate measurements. Therefore, we present endurance statistics for the L2 caches instead, which have more misses.

All evaluation was performed on a cache simulator only, because we did not have enough time during the project to setup and run a core timing simulator. Therefore, all performance evaluation is done assuming an in-order core model, and shown as average memory access time (AMAT). This is not an ideal metric for evaluating performance, but, because cloud applications spend significant time stalled on memory accesses and do not make use of cache bandwidth, they most likely do not have the ability to keep a

Policy	Parameters	Description
all		Include all cache lines
none		Don't include any lines
inst		Include lines that were evicted from the instruction cache
read	Threshold	Include lines that were read at least *threshold times*
write	Threshold	Include lines that were written less than *threshold times*
rand	Rand%	Only include a random percentage of candidate lines
bloom	Size, Rand%	Random include, with a Bloom filter to prevent re-insertion
cbloom	Size, Rand %	Same as bloom, but only include clean evictions

TABLE V: A summary of the benchmarked cache inclusion policies.

large number of requests in flight and an improvement in AMAT should directly translate into improved system performance.

We took several steps to ensure that our simulator was accurately modeling reality. The first test we ran was using valgrind's `cachegrind` cache simulation tool [1]. Although `cachegrind` does not model as complicated a cache hierarchy as we do, we were able to use it to ensure that both L1 and L3 misses matched reality for several small programs. This validated the trace generation and unmodified cache simulator. Then, we validated our modifications to the cache simulator by comparing our insertion on evict approach to the default insertion on miss approach, ensuring that the miss rates and wearout rates were nearly identical. Finally, we implemented a cache that was configured to not insert any data, and used it to verify that data is not incorrectly written to the cache when an insertion policy says that it should be dropped. We also ensure that accesses that bypass the cache are correctly reported as cache misses.

B. Inclusion Algorithms

We investigated the impact of two separate policies on the cache wearout. Several of these were selected to validate our simulation flow, including the policy of not including any misses and the policy of including all misses. A description of all the policies is shown in Table V. After a line is inserted into the cache, it is necessary to determine what happens when the line is written to. A naive cache might allow a line to be updated arbitrarily in the cache, which could lead to pathological access patterns wearing only one cache line to the detriment of others. Therefore, we evaluated the impact of varying the number of updates necessary to evict a cache line. We found empirically that the optimal threshold is very low, and is around 1 or 2 updates. These updates are only updates at the same level, not counting updates at previous, nonvolatile levels. For example, consider a line that misses in the L1 data cache with a nonvolatile L2. It is allocated into the L1, and then evicted and stored into the L2. This counts as zero updates. If the line is later read into the L1, modified, and evicted, this will count as 1 update and may be reason to evict the line from the L2.

The Bloom filter-based policy was the policy that we expected to perform best, but it lags behind several simpler policies. The Bloom filter is fixed to use only one hash, and uses a very large table to to try and get the best performance; the table is typically 5% full at the end of program execution. Lines are inserted into the Bloom filter only when they are forced out of the cache for having been evicted too much, and are allocated with a random percentage that is customizable as a parameter. We also evaluated a clean Bloom filter which only inserts lines that were evicted from the previous level of the cache hierarchy without being written to.

IV. Experimental Results

A. NVM Cache Performance

We started by evaluating the average memory access time for an L4 cache composed of RRAM, with RRAM device latencies of 3ns and 10ns over the existing L3 cache latency. To achieve the maximum size possible for our L4 cache, we model it as containing both the tags and data off-chip, accessed sequentially,

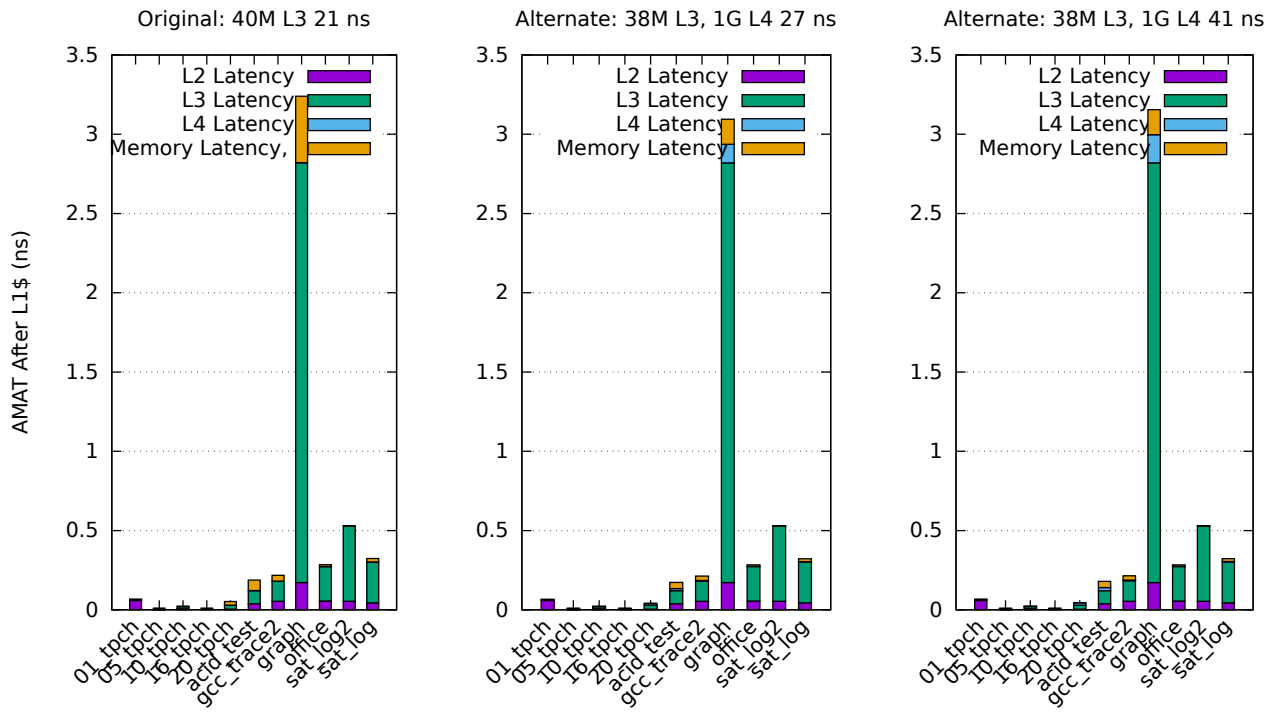

Fig. 3: AMAT difference when adding in a 1 GiB L4 with different overall hit latencies. The majority of time, with and without the L4, is taken by the L3.

and use the reference memory latencies measured from the Xeon E5-2683v4. This makes the overall L4 hit latencies 27 and 41 nanoseconds. Although these should provide some benefit, the benchmarks and traces that we gathered do not show a significant difference for the L4, even though they are being run for 200M L1 misses following a 10M L1 warmup. We believe that this may be the result of the large L3 (40M) and low memory latency (61ns) that our configuration has, which limit both the number of accesses to the L4 cache and the L3 miss penalty that the L4 is supposed to ameliorate. We also do not use a DRAM simulator in our model, which limits the impact of L3 misses on performance. In a real system, LLC misses contend with each other for a limited number of scheduler slots, while in our system they are allowed to complete as if there were no other misses. The AMAT breakdown for the NVM L4 is shown in Figure 3. The majority of cycles are spent accessing the L3 for all of our applications, which severely limits the potential benefit of the L4. Because the L4 uses on-chip space very efficiently as well, the potential slowdown of the L3 is limited and the results are effectively the same for both cases.

We then evaluated the potential impact of using NVM for the L2 cache, shown in Figure 4. We expected that this would have a significant impact, because a larger L2 cache would allow reads to be served closer to the processor, and our benchmarks make heavy use of the L2. We found that the average-case latency is extremely critical to the L2 cache performance, because so many accesses hit in the L2. If the RRAM L2 array takes 3ns longer than the SRAM L2 array, the decrease in L2 misses makes up for the increased L2 access time. However, if the RRAM L2 array slows to 10ns, then all of the L2 hits start to take more time to serve than the L2 misses, and there is a slowdown. RRAM read latencies do not appear to be commonly reported, but depend on the ratio between the high-resistance and low-resistance states of the RRAM cell. Because we only simulate MPKI, and then extrapolate to AMAT, we are not able to accurately simulate the impact that store latency and load latency has. We expect that, if we had an OoO core simulator, we would see that slowing down writes by forcing them to access the L3 would not be as limiting as shown here.

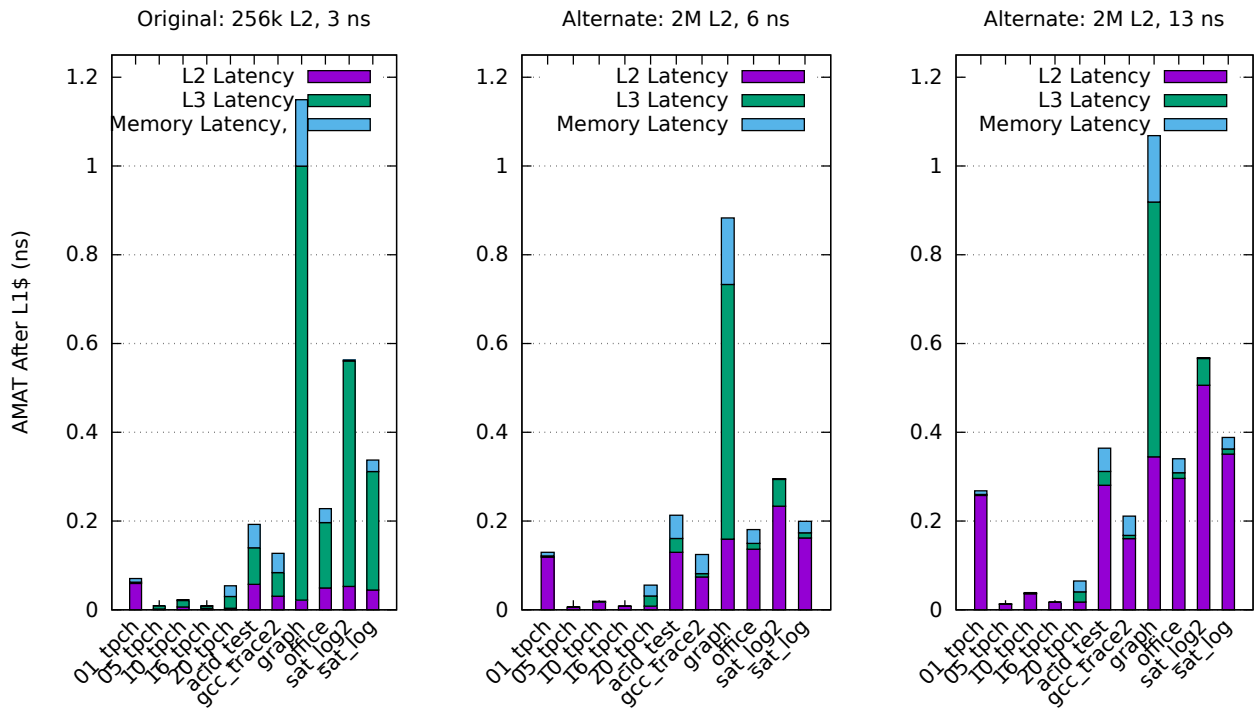

Fig. 4: AMAT difference when changing the L2 to using an RRAM data array and SRAM tag arrays. The values are computed for a 3ns and 10s latency increase.

B. Write Endurance Limitation

Our main contribution is a study of the impact of different cache insertion policies and update policies on cache endurance, which is shown in Figure 6. Typically, we see that the insert all policy produces both the lowest MPKI and the highest block updates. The insert all policies are run with no evict on update and evict on 1, 2, and 4 updates. Except for one benchmark, however, there is no difference in MPKI or updates for these cases, because once a line is evicted there is nothing to stop it from being reinserted. The opposite extreme of this spectrum is observed in the bloom filter. These policies insert a random percentage of values, track which are force-evicted for updates, and prevent them from being reinserted. The bloom filter reaches a vertical asymptote, which means that it allows lines to be inserted, but does not decrease the MPKI past some point. We believe that this is the result of the bloom filter being too sensitive: just because a line is written once, does not mean that it won't be repeatedly read again without being written. In this case, the bloom filter will prevent it from ever being a cache hit again, and can cause an unbounded number of misses with a few cache lines.

The other interesting results are the read and write threshold policies. With the exception of Graph500, these policies do not make a major difference to updates. We believe that this is due to the lines not living in the L1 cache long enough to allow a meaningful distinction to be made based on update count.

The most surprising result is the superior performance of the random algorithm. We found that the random algorithm almost always outperforms all other algorithms in terms of MPKI for a given update count. Although counterintuitive, randomly inserting a small fraction of lines actually results in a lower MPKI for some applications. We believe that this is because these applications have their accesses to reused lines interspersed with accesses to many non-reused lines. If there are enough non-reused lines between accesses to a reused line, the reused line will be pushed out of the cache. By randomly inserting only a few lines, the reused lines will be accessed enough times that they are never the LRU line.

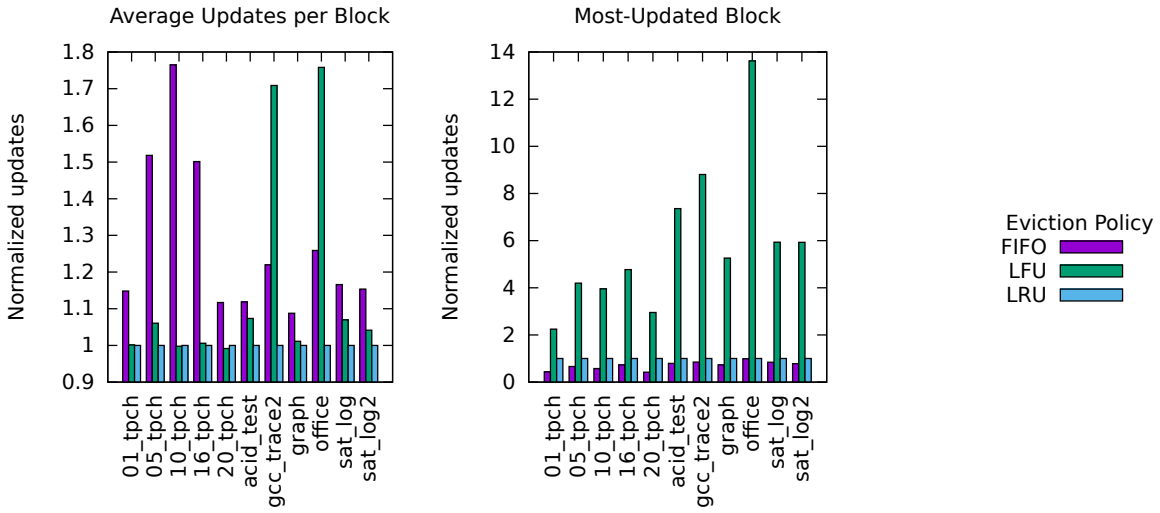

Fig. 5: The impact of cache replacement policies on average and worst-case wear.

C. Wear Evaluation

Although we used LRU cache replacement for the majority of our evaluation, we also ran tests with LFU and FIFO cache replacement to quantify the impact that these would have on wear. The results of this experiment, running on a 3% random cache insertion policy, are shown in Figure 5. We found that LRU is the best policy for minimizing the average updates per cache block. However, FIFO is better than LRU for minimizing the number of updates to the most-updated cache block. This is because if a few lines are frequently used, they won't ever be replaced, and the other lines in the set will take all of the updates. LFU is the worst for the most-updated block because it has a bias towards low-numbered ways (it's implemented with a for loop), and because it will tend to replace the most recently inserted cache line.

V. Related Work

A. Nonvolatile Memory

In contrast with our work, the current literature on NVM caches mostly sidesteps the concern of NVM write endurance. In [16], Mittal et al. provide a survey of nonvolatile memory management techniques, including strategies for lifetime enhancement. Their survey of roughly 140 works shows a shortage of literature on minimizing writes to cache-level nonvolatile memory. In particular, the vast majority of write minimization strategies are specific to block storage (*e.g.,* flash), or focus primarily on wear leveling as opposed to write minimization.

In the works that do focus on cache-level use of NVM technology, the write endurance problem is rarely addressed. For instance, UniFI [17] is a system that employs NVM for its last level cache and main memory, but with the goals of fault tolerance and low power consumption. Kiln [25] is another NVM cache and memory design whose goal is to decrease write latency by avoiding logging or copy-on-write schemes for fault tolerance. In both of these systems, write endurance is mentioned only briefly, and the authors choose to leave write endurance for other work.

Perhaps the most similar work to ours is done by Wang et al. [22], who employ cache inclusion policies to address the poor write endurance of NVM caches. They show that by modifying existing SSD wear leveling mechanisms to account for intra- and inter-set write variation, they can increase expected NVM cache lifetimes by 75%. However, the authors focus solely on wear leveling, rather than on reducing the absolute write count to the nonvolatile cache. Ideally, writes to NVM are both uniform in physical

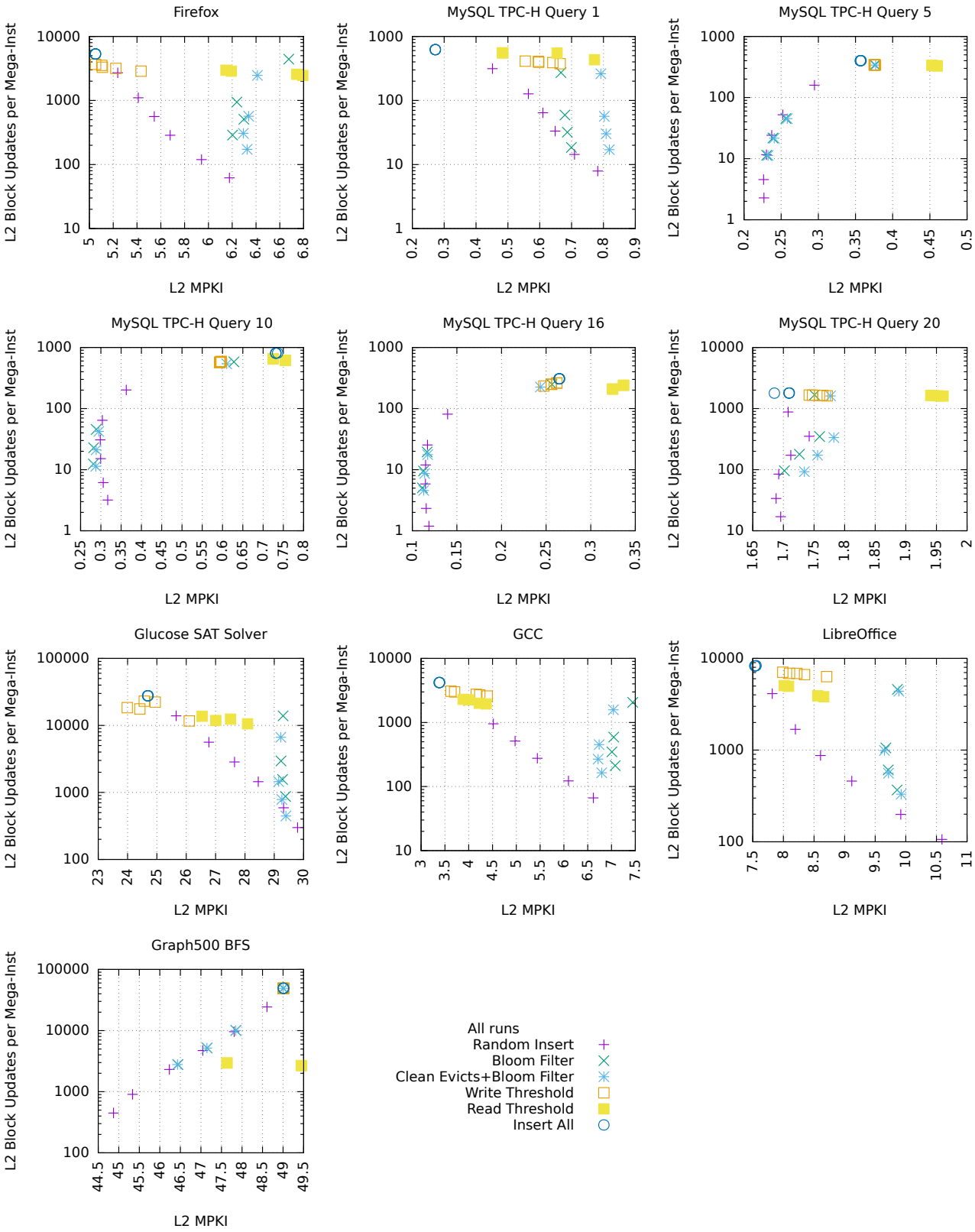

Fig. 6: Updates per Mega-Inst for all benchmarks and several insertion policies. Each insertion policy is simulated with a range of parameters to establish the range of results it can produce.

location *and* minimal in number, so our techniques could be used in conjunction with the cache-specific wear leveling schemes introduced in [22].

Sequoia [11] is a more recent (2016) proposal for improving NVM cache lifetime. Sequoia builds on the work of Wang et al. by reducing the overhead of their intra-set wear leveling mechanism. Still, Sequoia makes no attempt to reduce the absolute number of writes to the NVM cache, instead focusing entirely on achieving uniform wear leveling. In sum, our work differs from current literature in that we reduce absolute write count to the NVM cache rather than focusing on wear leveling schemes.

B. Level 4 Caches

There is also a significant body of work discussing the performance impacts of adding an L4 cache, most of which focuses on adding DRAM caches. Several industrial x86 processors also include an additional L4 cache, including Intel's Knights Landing and Crystalwell [20], [19]. Both of these are throughput-oriented devices, with Knights Landing being an accelerator for traditional HPC applications and Crystalwell having a large, bandwidth-hungry integrated GPU. This is likely due to the constraints of DRAM, which has relatively worse random access performance due to the need to open a row before reading or writing.

Several research projects have attempted to decrease the overhead of implementing an off-chip L4 cache using DRAM. Loh et al. implemented a scheme that stores tags and 64-byte data blocks together in DRAM, and uses an on-chip data structure to determine when an access to DRAM will be a miss, and thus bypass it [14]. Other related works assume that the cache lines used by an off-chip L4 cache will be larger, and optimize mapping entire pages at a time, sparsely, into the cache. [15] evaluates replacement policies for an L4 DRAM cache, focusing on avoiding interference between different processor cores.

CHOP is also similar to our proposed solution, but again focuses on unlimited-endurance caches [10]. CHOP uses an on-chip filter cache to track hot pages. Pages that hit frequently in the filter cache are then allocated in the off-chip DRAM cache. The primary goal of CHOP is to make the best use of the limited bandwidth to the off-chip DRAM cache, while not exploding the size of the tag arrays. If all L3 cache evictions were allocated into the page-based L4 cache, in the worst case, the memory controller would have to fetch 15 lines from DRAM for every L3 eviction to complete the page insertion. It is also possible to sparsely insert lines within a page, which is the approach taken by the footprint cache [9]. The footprint cache only inserts lines that are accessed by the processor into the L4 cache, eliminating the majority of the traffic.

The focus that the prior work has on DRAM last level caches is based around compensating for the limitations of DRAM: it is challenging to fabricate DRAM in the same process as logic, and there is no good way to do tight integration of DRAM into a logic process. Additionally, DRAM cells are not very readable, and the charge on the bitlines needs to be amplified aggressively to be read; this imposes a high penalty on opening rows in the array and random accesses, which then requires the use of a DRAM scheduler and controller. Taken together, these factors conspire to produce a caching layer that has a high latency per access. RRAM may be able to solve these problems, by being fabricated directly on top of logic without the need for an access transistor [7].

VI. Future Directions

Although our work shows that RRAM caches could provide a significant speedup at several levels of the cache hierarchy, several questions still need to be answered before a complete system could be designed. One problem with inserting cache lines upon evicts from lower-level caches is cache coherence. This is a problem for the proposed L2 RRAM cache, and might be a problem for the RRAM L4 cache. We also identified a major simulation discrepancy regarding the simulation of kernel code, which could be fixed through more thorough simulation. Another major problem is dealing with cache wearout when it actually occurs through the addition of a translation layer.

A. Cache Coherency

Snooping based cache coherence is made significantly simpler by the use of an inclusive cache hierarchy. If the L2 tags are guaranteed to contain all the data that is stored in the L1 tags, then a coherence controller can respond to coherence requests using only the data stored in the L2. There are several ways that an RRAM cache could be integrated in a coherent system. The simplest is adding an array to store tags from previous levels of the cache hierarchy alongside the cache, which would increase the area required by our design by approximately 25%. Another possibility is to query both the L1 and L2 on a coherence request, which would impact coherence latency. Finally, it may be possible to store some data about the L1's contents, such as the list of mapped pages, in the L2. This would decrease the area overhead while minimizing the number of requests forwarded to the L1.

B. Full-System Simulation

Because our simulator is based on translating a user-level binary, it does not support tracing system level code. We considered several options to get more accurate system traces but were not able to implement them during the quarter. The simplest solution would be to use libraries that perform computationally intensive system level tasks, such as networking, in user space [4]. Although not an exact model of the kernel code, the computation performed and overall access pattern should be similar. Another possibility would be to use a system level simulator, such as gem5. This would allow an exact trace of Linux's execution to be gathered, at the cost of slower trace generation time.

C. Translation Layer

Another problem results from the impact of cache wearout. Current processors allow for remapping lines or columns of memory arrays to compensate for manufacturing defects or wearout [23]. However, this requires the cache to be tested either during manufacturing or at boot, eliminates many good bits for one bad bit, and therefore does not handle partially failed cache lines well. Because RRAM can be far denser than the underlying tag arrays, it should be possible to store a significant amount of error correction data with the data in the cache array to tolerate sporadically failed bits.

An ideal RRAM cache would also enforce even wear. Our evict on write approach decreases the opportunity for a single line to burn a hole in the cache, but there are still potential pitfalls with using an LRU or pseudo-LRU cache replacement strategy. If all but one line in a cache set are frequently used, the cache will keep those and repeatedly update the one unused line, leading to uneven wear. This could be solved by wear leveling algorithm to move lines around, or by using a different replacement policy.

D. Adaptive Allocation Policies

When evaluating insertion policies, we found that different applications have extremely different baseline update rates to the cache. Although we are able to restrict updates to be some fraction of the baseline update rate using a random insertion policy, it is necessary to manually set the random insertion rate to guarantee a maximum number of cache updates. A real-world system would need to use a control loop of some kind to handle setting this insertion rate, and may need to add logic to guard against phase behavior to ensure that the total number of updates remains bounded.

VII. Conclusion

The increasing availability of NVM technologies such as RRAM make them interesting candidates for cache construction. However, write endurance concerns led us to seek a way to reduce updates to the cache when possible, while still maintaining adequate MPKI and AMAT as measures of application performance. Our work finds that replacing a conventional SRAM L2 with a much larger RRAM L2 yields performance benefits for some applications (the Graph500 breadth-first search and Firefox browser), and

slight slowdowns for others (the SAT solver and document conversion). Our evaluation is predicated on the existence of fast, tightly integrated RRAM, which can be fabricated directly on top of the CPU logic instead of on a separate die.

We were unable to run the cloud applications we wanted to, because they had a significant system component we could not trace. Instead, we selected benchmarks that we expected would have complicated code and data access patterns. Trace-based simulation allowed us to run each application once, and then quickly assess its behavior on a wide range of cache configurations and inclusion policies. Somewhat surprisingly, a random inclusion policy performed best for deciding which lines were kept around in the cache. In the scatterplots in Figure 6, note that random inclusion forms the Pareto curve for every application we profiled, with the exception of write threshold based insertion dominating a small portion of the space. We discovered that random inclusion is able to make the update rate arbitrarily small, and despite doing so it does not not significantly increase MPKI (and may even decrease MPKI).

Finally, we discovered that our initial goal of an RRAM L4 cache may not be the best option. This may be the result of simulation methodology limitations, or it may be a function of our baseline's large L3 cache and low main memory access time, both of which were not represented in the L4 cache literature we surveyed. It may be feasible to implement an even larger L4 cache, on the order of several gigabytes, or to implement an L4 cache for NUMA nodes that have significant memory access times.

REFERENCES

[1] Cachegrind: a cache and branch-prediction profiler. http://valgrind.org/docs/manual/cg-manual.html.

[2] Intel core i7-3960x processor die detail. https://www.pcworld.com/article/243749/lab_tested_intel_core_i7_3960x_extreme_edition.html.

[3] Intel xeon e5-2683-v4. http://www.cpu-world.com/CPUs/Xeon/Intel-Xeon

[4] Seastar. seastar-project.org.

[5] Stt-ram as a sub for sram and dram. http://arch.cs.utah.edu/arch-rd-club/STT-RAM.pptx.

[6] Tinymembench: Simple benchmark for memory throughput and latency. https://github.com/ssvb/tinymembench.

[7] Christophe Chevallier. Resistive ram, present status and future applications. https://n3xt.stanford.edu/system/files/c_chevallier_resistive_-ram_present_status_and_future_applications.pdf, 2014.

[8] PAN Feng, CHEN Chao, Zhi-shun Wang, Yu-chao Yang, YANG Jing, and ZENG Fei. Nonvolatile resistive switching memories-characteristics, mechanisms and challenges. *Progress in Natural Science: Materials International*, 20(Supplement C):1 – 15, 2010.

[9] Djordje Jevdjic, Stavros Volos, and Babak Falsafi. Die-stacked dram caches for servers: hit ratio, latency, or bandwidth? have it all with footprint cache. In *ACM SIGARCH Computer Architecture News*, volume 41, pages 404–415. ACM, 2013.

[10] Xiaowei Jiang, Niti Madan, Li Zhao, Mike Upton, Ravishankar Iyer, Srihari Makineni, Donald Newell, Yan Solihin, and Rajeev Balasubramonian. Chop: Adaptive filter-based dram caching for cmp server platforms. In *High Performance Computer Architecture (HPCA), 2010 IEEE 16th International Symposium on*, pages 1–12. IEEE, 2010.

[11] Mohammad Reza Jokar, Mohammad Arjomand, and Hamid Sarbazi-Azad. Sequoia: A high-endurance nvm-based cache architecture. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, 24(3):954–967, 2016.

[12] Svilen Kanev, Juan Pablo Darago, Kim Hazelwood, Parthasarathy Ranganathan, Tipp Moseley, Gu-Yeon Wei, and David Brooks. Profiling a warehouse-scale computer. In *Proceedings of the 42Nd Annual International Symposium on Computer Architecture*, ISCA '15, pages 158–169, New York, NY, USA, 2015. ACM.

[13] Mark Kryder. Kryders law. *Scientific American*, pages 32–33, 2005.

[14] Gabriel Loh and Mark D Hill. Supporting very large dram caches with compound-access scheduling and missmap. *IEEE Micro*, 32(3):70–78, 2012.

[15] Gabriel H Loh. Extending the effectiveness of 3d-stacked dram caches with an adaptive multi-queue policy. In *Proceedings of the 42nd Annual IEEE/ACM International Symposium on Microarchitecture*, pages 201–212. ACM, 2009.

[16] S. Mittal and J. S. Vetter. A survey of software techniques for using non-volatile memories for storage and main memory systems. *IEEE Transactions on Parallel and Distributed Systems*, 27(5):1537–1550, May 2016.

[17] Somayeh Sardashti and David A Wood. Unifi: leveraging non-volatile memories for a unified fault tolerance and idle power management technique. In *Proceedings of the 26th ACM international conference on Supercomputing*, pages 59–68. ACM, 2012.

[18] V. Selfa, J. Sahuquillo, L. Eeckhout, S. Petit, and M. E. Gmez. Application clustering policies to address system fairness with intel cache allocation technology. In *2017 26th International Conference on Parallel Architectures and Compilation Techniques (PACT)*, pages 194–205, Sept 2017.

[19] Eric Shiu and Simon Prakash. System challenges and hardware requirements for future consumer devices: From wearable to chromebooks and devices in-between. In *VLSI Circuits (VLSI Circuits), 2015 Symposium on*, pages 1–5. IEEE, 2015.

[20] Avinash Sodani. Knights landing (knl): 2nd generation intel® xeon phi processor. In *Hot Chips 27 Symposium (HCS), 2015 IEEE*, pages 1–24. IEEE, 2015.

[21] M Mitchell Waldrop. The chips are down for moore's law. *Nature*, 530(7589):144–147, 2016.

[22] Jue Wang, Xiangyu Dong, Yuan Xie, and Norman P Jouppi. i 2 wap: Improving non-volatile cache lifetime by reducing inter- and intra-set write variations. In *High Performance Computer Architecture (HPCA2013), 2013 IEEE 19th International Symposium on*, pages 234–245. IEEE, 2013.

[23] Neil Weste, David Harris, and Ayan Banerjee. volume 11. 2005.

[24] Yi Wu, Jiale Liang, Shimeng Yu, Ximeng Guan, and H.-S. Philip Wong. Resistive switching random access memory - materials, device, interconnects, and scaling considerations. pages 16–21. IEEE, 2012.

[25] Jishen Zhao, Sheng Li, Doe Hyun Yoon, Yuan Xie, and Norman P Jouppi. Kiln: Closing the performance gap between systems with and without persistence support. In *Microarchitecture (MICRO), 2013 46th Annual IEEE/ACM International Symposium on*, pages 421–432. IEEE, 2013.