2017 IEEE International Parallel and Distributed Processing Symposium

Adaptive Software Caching for Efficient NVRAM Data Persistence

Pengcheng Li
University of Rochester
Rochester, USA
pli@cs.rochester.com

Dhruva R. Chakrabarti
Hewlett Packard Labs
Palo Alto, USA
dhruvac@gmail.com

Chen Ding
University of Rochester
Rochester, USA
cding@cs.rochester.edu

Liang Yuan
Inst. of Computing Technology, CAS
Beijing, China
yuanliang@ict.ac.cn

***Abstract*—Non-volatile main memory (NVRAM) enables data persistence in memory. However, the existence of transient CPU caches in modern computer architectures brings a serious performance issue. In particular, cache lines have to be flushed frequently to guarantee consistent persistent program states. Hence, persistence and performance cannot be easily obtained simultaneously.**

In this paper, we optimize data persistence by proposing a software cache. The software cache first buffers lines that need to be flushed, and then flushes them out at an appropriate later time. The software cache aims to maximize the combination of cache line flushes. We designed a new linear-time algorithm to calculate cache miss ratio curve (MRC) so as to adaptively select the best cache capacity at run-time based on program behavior. We evaluated the software cache on a real-world memory-based database benchmark, the SPLASH2 benchmark suite and four micro-benchmarks. Results indicate that the software cache solution reduces cache write backs to persistent memory by $12\times$ and improves performance over the state-of-the-art methods by $2.1\times$ on average, measured on a real system emulator.

***Keywords*-NVM persistence; writing caching; timescale reuse; linear-time MRC**

I. Introduction

Persistent memory or non-volatile memory (NVRAM) technologies, such as memristors and phase change memory, are increasingly popular. Micron and Intel have announced that their non-volatile memory technology, 3D XPoint, would be productized in the near future.

Persistent memory offers higher density and lower power than DRAM and much faster access than SSD, and it is byte-addressable [36]. This in-memory durability model [9] can greatly change the programming paradigm for many applications. While an application today typically maintains an in-memory object format and a separate durable format for a durable block device, only one format of data will suffice in this new world.

A problem of NVRAM data persistence is the transient memories in current computer architectures, such as CPU caches. As described in [9], at any point of program execution, some of the updates to persistent memory may only reside in CPU caches and have not yet propagated to NVRAM. If there is a failure at this point of execution, the program state in NVRAM may not be consistent thus preventing full recovery. The solution is to flush out dirty cache lines at appropriate program points so that the above situation cannot occur.

The entire cache can be flushed to memory. For example, ARM provides "flush entire cache" operations [2]. However, the cost is often too high and largely unnecessary depending on the consistency model. In this paper, we consider a software system, namely `Atlas` [9], which uses the consistency model called the failure-atomic section (FASE). To ensure consistent persistency, `Atlas` writes back the cache lines written in a FASE before the end of the FASE. Write-back is implemented in software using the cache line flush operation. We use the two terms, write-back and flush, interchangeably.

Two basic solutions are eager and lazy write backs. The eager solution flushes every store out of cache immediately after the store finishes, and the lazy solution records the addresses of modified data and flushes them at the end of the FASE.

The eager solution has the benefit of hiding memory transfer cost via asynchronous cache line flushes but incurs too many flushes. We measured this approach for SPLASH2 [45] benchmark, with the test setup given in Section IV. The performance drops by $22\times$ on average, as shown in Table I. On the other hand, the lazy solution flushes each data just once, but its cost does not overlap with computation. We found in our tests that the CPU stall at the end of a FASE severely hurts performance.

Program	Slowdown	Program	Slowdown
barnes	22×	fmm	24×
ocean	17×	raytrace	6×
volrend	26×	water-nsquared	24×
water-spatial	33×	**average**	22×

Table I: The cost to eager data persistence in SPLASH2 benchmarks.

This paper describes a new software cache solution. It buffers the writes to persistent memory and flushes a cache line either at each cache eviction or at the end of a FASE. It benefits from asynchronous cache line flushes and bounds the stall time at the end of a FASE. In addition, the software cache can adaptively control its size at run time. This paper describes a new theory to select the best cache size.

1530-2075/17 $31.00 © 2017 IEEE
DOI 10.1109/IPDPS.2017.83

The paper makes the following contributions:

- We describe a new, software cache solution to reduce the overhead of NVRAM data persistence.
- We develop a reuse-based locality theory to analyze the locality of persistent memory writes and choose the best size for write caching. It includes a linear time algorithm for efficient online analysis.
- We give a complete system design, including an efficient software cache that can be adapted during execution, as well as the compiler and run-time support to ensure consistency semantics.
- We evaluate the software cache on 12 applications and compare it with four other approaches. Results show that the software cache approach reduces cache line flushes by 12× and improves performance by 2.1× over the previous state-of-the-art solution.

II. Atlas Software Cache

Our system is built on `Atlas`, which is the main implementation of the FASE persistency model. This section describes first the `Atlas` design and then our new solution.

A. FASE in `Atlas`

In `Atlas` programming model, a data structure has a set of invariants, which must be satisfied for it to be in a consistent state. A program mutates one or more data structures and may temporarily violate an invariant. The programming model requires that all the codes that violate a program invariant be grouped into a *failure-atomic section (FASE)* [9]. Upon a system failure, either all or none of the updates in a FASE are visible in NVRAM. Therefore, persistent data is guaranteed to be consistent at the end of a FASE.

`Atlas` monitors data writes at cache-line granularity. It uses a table to record the address of all modified cache blocks. Upon a write, if its cache-line address is in the table, `Atlas` does nothing. Otherwise, the address is inserted. If the table is full, a previously stored cache-line address is read and then flushed before the new insertion. The whole table is flushed at the end of a FASE.

Modern computers provide support for flushing out a given cache line. For example, on x86 processors, `clflush` and `clflushopt` flush and invalidate a cache line. A new instruction `clwb` flushes without invalidating a cache line. Hence, `clwb` may cause other threads to access a stale value. `Atlas` uses `clflush`.

A flush operation has a non-trivial cost. In addition, since `Atlas` uses `clflush` and invalidates the cache line, the next access will be a cache miss. For both its direct and indirect costs, the number of cache line flushes is highly correlated with running time.

B. Software Cache

To avoid conflating hardware and software cache, we use the word "cache" and "software cache" interchangeably to mean software cache and "hardware cache" otherwise.

The `Atlas` table is equivalent to a direct-mapped, fixed size cache. In the new solution, we use a fully associative LRU-based resizable cache implemented in software. The goal of software cache is to minimize the number of flushes by buffering the writes to the same cache line and by adapting its size, i.e. being workload aware. It would minimize not only the direct cost of flush operations but also the indirect cost due to the subsequent misses on the flushed data.

The software cache is per thread. Figure 1 shows its basic execution model. For a program with two threads, it has two software caches. Each time a thread running in a FASE writes to persistent memory, the thread stores the cache line address to its software cache. When two threads write to the same cache line, they each stores its address in its own cache. There is no data sharing between software caches.

Because there is no data sharing, software caches are isolated from each other. Each thread independently manipulates its own cache. Because of the isolation, the cost of software caches is fully distributed and does not affect program scalability. In addition, the implementation of cache is efficient because it does not use locking.

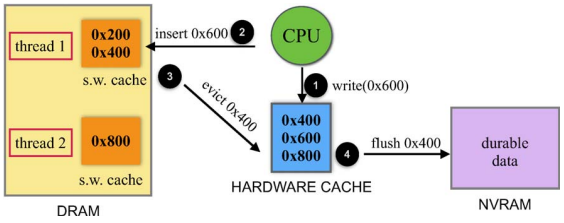

Figure 1: Illustration of the software cache. The software cache of Thread 1 has two blocks and is full. When Thread 1 accesses a new cache line 0x600, the address is inserted into the cache, 0x400 is evicted from the software cache, and the cache line flushed out of the hardware cache.

Software cache does not affect original program states. It is placed in the faster DRAM, rather than NVRAM.

When a software cache is full, it chooses the least recently accessed cache-line address. Its thread then issues the flush command to write back the cache line from hardware cache to NVRAM. Figure 1 shows that the local store of thread 1 is full, and after inserting the new cache line address 0x600, thread 1 uses the flush operation to force the hardware cache to evict 0x400 and write it back to NVRAM.

It is well known that cache often does not yield benefits proportional with capacity. This is often shown using the miss ratio curve (MRC). Figure 2 shows the MRC of the software cache of the *water-spatial* program in the SPLASH2 benchmark. Software cache would choose 23

since it is the most cost effective. Although a larger size means fewer cache misses, it also increases the stall time at the end of a FASE. Section III-C describes how to choose the best cache size at run time.

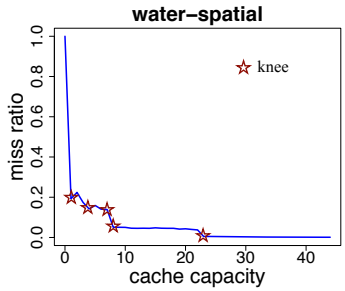

Figure 2: The MRC of *water-spatial* program. It has a few "knees," where cache miss ratio dramatically drops. In this case, we choose 23 as the software cache size, since it corresponds to the knee that has the smallest cache miss ratio and is not overly large.

III. Optimized Adaptive Write Caching

Locality theory is useful in studying memory systems because it gives a precise reading of the locality of a program or a system and provides the basis for optimization. This section formulates the theory of write caching, which we use to determine the best cache size.

A. Write Caching

Traditionally, cache has been optimized for fast reads. For persistent memory, the software cache we use stores only modified data. Furthermore, its performance is defined entirely by data writes, more precisely, how many writes are combined while the data resides in cache. Indeed, previous Atlas table does not consider data reads at all. As an extension, write caching considers only data writes. We call the cache *write-combining*.

In write-combining cache, a reuse happens when there is a write to an existing data block in cache. Since the existence implies an earlier write, the reuse enables the cache to combine this write. Therefore, maximizing the performance of write caching is equivalent to maximizing the number of data reuses in the write-combining cache.

For optimization, we may build on two types of locality theories. The first is reuse distance, which measures the locality of a memory access [34]. The second is the working-set size, which measures data usage in a time window [14]. We call the first type *access locality* and the second *timescale locality*. Both locality types have been used to optimize the cache size. However, reuse distance is costly to measure, especially online for CPU caches. Recent solutions use timescale locality [17, 41, 46]. A key advantages is the asymptotic time complexity, which is linear for timescale locality but more than linear for reuse distance.

Next we extend timescale locality to model reuses in write caching and show that the new model has the same asymptotic efficiency.

B. Reuse-based Timescale Locality

We consider an execution as a sequence of data accesses (writes). A logical time is assigned to each data access. A time window is designated by two data accesses and includes all intervening accesses. The length of a window is the number of accesses it contains.

Reuse locality is measured by the number of data reuses in a time window. Counting the number of intra-window reuses is the same as counting number of reuse intervals that fall within the window. We define the following:

Definition 1. Reuse interval and Intra-window reuse *The time interval between a data access and its next access to the same datum is defined as a reuse interval. If a reuse interval is enclosed within a window, we say that the window has an intra-window reuse.*

Different windows may contain different numbers of reuses. We define the timescale reuse *reuse(k)* as the average number of intra-window reuses of all windows of length k. We call the length k the timescale parameter. Given any trace, *reuse(k)* is uniquely defined.

For example, the trace "abb" has two windows of length two. They have 0 and 1 intra-window reuses, respectively. Thus, *reuse(2)* = $\frac{1}{2}$. It is straightforward to measure *reuse(k)* for any single k. As a locality measure, however, we need the value for all $k \geq 0$. Assume a trace of n data accesses. Consider all windows of all lengths ($1 \leq k \leq n$). The number of these windows is n-choose-2 or $\frac{n(n+1)}{2}$. Instead of counting reuses in all windows, we transform the problem to make it solvable in linear time $O(n)$.

Instead of counting reuse intervals in each window w, we count the number of windows enclosing each reuse interval, $[s_i, e_i]$, whereby s_i is the start time of the interval and e_i the end time. The total of the two counts are the same, as shown in Eq. 1.

$$\begin{aligned} \textit{reuse}(k) &= \frac{\sum_{\text{all window } w} (\text{number of reuse intervals in } w)}{n-k+1} \\ &= \frac{\sum_{\text{all interval } i, e_i - s_i \leq k} \text{windows enclosing } (i)}{n-k+1} \end{aligned} \quad (1)$$

Let's consider how to count the number of k-length windows enclosing a reuse interval, $[s_i, e_i]$. Obviously we must have $e_i - s_i \leq k$; otherwise the count is 0. Figure 3 shows four cases for this counting. Case 1, $s_i \geq k$ and $e_i \leq n - k + 1$, is an "internal" case, while the other three are boundary cases. For Case 1, the windows starting from $e_i - k - 1$ to s_i are all counted, therefore the number is $k - (e_i - s_i) + 1$. The counts of the other three cases are given by the formulas in Figure 3.

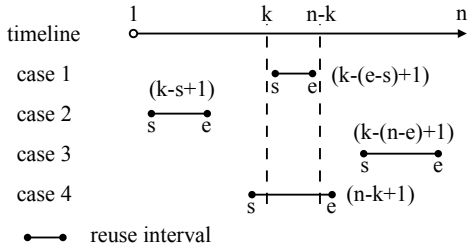

Figure 3: Four cases of window counting for a reuse interval, that is, the number of k-length windows that enclose an interval $[s, e]$.

Given a trace of length of n accesses and r distinct reuse intervals, $[s_i, e_i]$ $(i = 1 \ldots r)$, Eq. 2 shows the complete formal solution. The function I is a predicate, which equals to 1 if the condition is true and otherwise 0. In the equation, $I(e_i - s_i \leq k)$ counts only reuse intervals not longer than k.

$$\textit{reuse}(k) = \frac{\sum_{i=1}^{r} I(e_i\text{-}s_i \leq k)(\textit{min}(n\text{-}k, s_i)\text{-}\textit{max}(k, e_i)\text{+}k\text{+}1)}{n-k+1} \tag{2}$$

The solution of interval counting is based on our prior work of all-window liveness [27]. In liveness analysis, each interval is the life of a memory object from allocation to free. Hence, Eq. 2 is mathematically equivalent to the liveness count in [27] Eq. 2 is linear time $O(r)$ to compute for each timescale k. We need to compute *reuse*(k) in linear time for all k. For this, we refer to the solution in [27].

From Reuse to Cache Hit Ratio: At any moment t in fully associative LRU cache, the content consists of data referenced by previous k accesses for some k. *reuse(k)* is the average number of reuses in each k consecutive data accesses. It follows that on average, there are *k-reuse(k)* distinct data in these accesses. The next access is a hit if it is a reuse; otherwise, it is a miss. The difference, *reuse*$'(k)$*=reuse(k+1) - reuse(k)*, shows the average portion of times that the next access is a reuse. Hence, the hit ratio of cache of size (k - *reuse(k)*) is the derivative of *reuse(k)* at k, as shown in Eq. 3.

$$\textit{hr}(c) = \textit{reuse}'(k) = \textit{reuse(k+1)} - \textit{reuse(k)} \tag{3}$$

where $c = k - \textit{reuse}(k)$. To illustrate, consider an example pattern "$abab$..." that is is infinitely repeating. The following table shows discrete values of *reuse(k)* and hit ratio, where c denotes cache size, i.e. *k-reuse(k)*.

k	$reuse(k)$	c	hr
1	0	1	0
2	0	2	1
3	1	2	1
4	2	2	1

Reuse vs. Footprint Locality: The working-set size (WSS) of a time window is the number of distinct data accessed in the window. Footprint *fp*(k) is the average WSS in all windows of length k [46]. Xiang et al. give the following (linear-time) formula to compute *fp*(k) for all k.

$$\begin{aligned} \textit{fp}(k) \quad &= m - \frac{1}{n-k+1}\Big(\sum_{i=1}^{m}(f_i - k)I(f_i > k) \\ + \quad &\sum_{i=1}^{m}(l_i - k)I(l_i > k) + n\sum_{t=k+1}^{n-1}(t-k)P(rt = t)\Big) \end{aligned} \tag{4}$$

where m is the number of distinct data, f_i the first access time, l_i the last access time, $P(rt = t)$ the fraction of accesses with a reuse time t, and I the predicate function.

For each window, it is obvious that the working-set size plus the number of reuses is total number of accesses. It follows that the average WSS and average reuse have the same relation. Thus we have,

$$\textit{reuse}(k) + \textit{fp}(k) = k \tag{5}$$

The relation in Eq. 5 shows that reuse and footprint are dual metrics of cache locality. Mathematically, the simple form is highly interesting, because the equations to compute *reuse*(k),*fp*(k) are complex and completely different. The reuse calculation, Eq. 2, uses the start and ending time of reuse intervals, while the footprint calculation, Eq. 4, uses the length of reuse intervals. The simple relation, although obvious from the definitions, would have been completely unexpected to someone who looked only at the equations.

Xiang et al. showed that the miss ratio is the derivative of footprint and called this and other conversions the higher-order theory of locality (HOTL) [46]. Using Eq. 5, we can easily derive their result as follows:

$$\textit{reuse}'\textit{(k)} = 1 - \textit{fp}'\textit{(k)} = 1 - \textit{mr(c)} = \textit{hr(c)} \tag{6}$$

where c is k - *reuse*(k), and $fp'(k)$ denotes the derivation of $fp(k)$.

Correctness: Xiang et al. gave a correctness condition for the miss-ratio conversion. It is called reuse-window hypothesis, which states that the WSS distribution in all reuse windows is the same as the WSS distribution in all windows [46]. From Eq. 6, we conclude that the hit-ratio conversion in Eq. 3 has the same correctness condition.

In this section, we have presented a new formulation of the timescale locality based on data reuses. The result is mathematically derivable from footprint, so it is not new. However, the formulation is new and has not been considered in past work. The new derivation gives a new linear-time algorithm to calculate cache performance. It is more intuitive for understanding write caching. In addition, it is the first mathematical connection between the theory of locality [46](data caching) and the theory of liveness [27] (memory allocation).

Adaptation to FASE Semantics: To use the reuse locality for FASEs, we make two modifications. The first is obvious — we analyze only the sequence of persistent writes. The next is more subtle. Consider a trace under the FASE semantics $ab|ab|ab...$, where a $|$ denotes the end of a FASE. If we ignore the FASEs, the trace is $ababab...$. *reuse(2)* = 0 and *reuse(3)* = 1, thus cache hit ratio of size 2 cache is 100%. However, since we flush a, b at every $|$, every write is actually a miss, whatever the cache size is. In essence, the FASE semantics invalidates all data reuses across a FASE boundary. The problem cannot be solved online by flushing the content of cache at the end of FASE, because we need the locality for all cache sizes. We modify a write trace so the writes from different FASEs use completely different addresses. In other words, the same address cannot be used in more than one FASE. In this example, the trace will be converted into $abcdef...$ before locality analysis.

C. Adaptive Caching Implementation

This section gives the implementation of the adaptive software write-combining cache, including MRC analysis, online cache size selection, and compiler support to for FASE code.

The Cache: A software cache is created for each thread. Each cache includes a hash map and a doubly linked list. Each node of the list stores a cache block. The hash table maps memory address to the cache block. All cache operations have $O(1)$ time complexity: including search using the hash map; insertion, update and deletion using the linked list. A similar data structure is used in Linux to manage pages [20], whereby a red-black tree for insertion, update and deletion and a doubly linked list for traversal. Our cache is faster by using a hash map rather than a red-black tree.

MRC Analysis: Online MRC analysis is based on bursty sampling [3, 11]. It partitions a program execution into bursts and hibernation periods. At a burst, we monitor the sequence of persistent writes. At the end of a burst period, we calculate MRC and then adjust the cache capacity, In the evaluation, a burst has 64M writes. We found it is sufficient to analyze MRC just once, so we set the hibernation to an infinite length. Offline analysis uses the whole-trace of persistent writes and chooses the best cache size after profiling and before the testing run. The offline choice is the best single cache size for the whole execution. Both online and offline calculations use the formula discussed in Section III-B, whose time complexity is linear (actually constant in online analysis).

We assume that threads have different cache behavior and analyze MRC for each thread. To reduce the overhead, we could group threads with similar write locality and calculate one MRC for each group. We will consider it in our future work.

Cache Size Optimization: We set 8 as the default cache size and choose the best cache size once we have the MRC. To avoid excessive stalls due to cache flushes at the end of a FASE, we bound the maximal cache size. Based on testing, we set the maximal cache size to 50. From the MRC, we find inflection points or "knees". First, we calculate the decrease in miss ratio for every cache size increase (i.e. the gradient), rank the decreases, and pick the top few as candidate knees. We then choose the knee that has the largest cache size. For example, Figure 2 has five inflection points. We choose 23. If a MRC does not have obvious inflection points, we choose the maximal cache size.

Compiler Support: We implemented a pass in LLVM [22] to instrument all memory stores and FASE locking and unlocking operations. At run time, these operations are monitored by the software cache. The same compiler support is used by Atlas [9].

IV. Evaluation

A. Experimental Methodology

Emulation System: Intel and Micron announced that their NVRAM technology 3D XPoint [23] will be productized in the near future. Because real NVRAM is not yet available, we used an emulator that uses DRAM to simulate NVRAM. The test platform is a machine shipped with 60 Intel Xeon E7-4890 cores at 2.8GHz, running Linux OS 3.10. We use `tmpfs` [42] to allocate persistent data. At the start of a process, it is directly memory mapped into the address space. Data in tmpfs survives process termination and hence can be used or shared by another process. Hence, the emulation system provides a directly mapped, byte-addressable persistent memory across process terminations, which mimics the functionality of NVRAM for persistence across failures.

Persistent Data Caching: We have implemented and tested six techniques of persistent data caching. The first, **SC**, is the online solution that measures the MRC and adjusts the cache size as a program executes. For comparison, the second uses the MRC measured offline. We call it **SC-offline**.

We compare SC and SC-offline with four alternatives. The first approach is the state-of-the-art `Atlas` approach [9] (**AT**),as described in Section II. The second is the eager approach (**ER**), which flushes cache lines instantly every time a persistent store happens. The third is the lazy approach (**LA**), which flushes all cache lines at the end of a FASE. LA gives the lowest possible. The last is the approach that does not flush any cache lines at all (**BEST**). BEST is not a valid solution but approximates the effect of optimal caching, i.e. minimal number of flushes and perfect flush scheduling to overlap computation and memory transfer. Obviously, BEST is the upper bound of the optimal caching. To compare these six solutions, we time their executions and take the averages of five runs.

B. Applications

Memory-mapped database (MDB): MDB [12] is a read-optimized key-value store based on B+-tree to service the OpenLDAP [38] backend. Compared with the Berkeley-DB, MDB is more excellent in many aspects. MDB supports transactions and uses Multi-Version Concurrency Control (MVCC). A memory-mapped file of in-disk key-value pairs is used for gets and puts. Readers start with the snapshot at the beginning of a transaction and run in parallel with writers. Writers use copy-on-write policy. A reader always sees a valid B+-tree without having to acquire locks. A write transaction is required to acquire an exclusive lock to update an old version to a new version. MDB gives us a sense of tradeoff between persistence and performance in the cloud-level applications.

SPLASH2: All applications from the SPLASH2 [45] benchmark suite are evaluated except for *radiosity*, which we couldn't compile on our system. The basic statistics of SPLASH2 benchmark is shown in Table III. Variations shown by problem sizes and the number of FASEs demonstrate that they are good representatives. We persist all data structures in SPLASH2 that are not stack-allocated to create the most demand for data durability, since the cost of durability is often proportional to the amount of persisted data [9], and then to give us a fair idea of the performance we are able to obtain while maintaining the worst-case consistent durability. The benchmarks can be run with up to 32 threads.

Micro benchmarks: To understand the cost of cache line flushes and the effect of cache size on that cost, we start with a simple sequential program *persistent-array*. It has only one FASE, which consists of a two-level nested loop. The inner loop iterates 400 times and writes in iteration i to the i-th element of an array of integers. The outer loop repeats the inner loop 2500 times. On the tested machine, a cache block has 64 bytes, i.e. 16 (4-byte) integers. The inner loop accesses 25 (cache line aligned) or 26 (not cache line aligned) cache blocks, which is the working set. `Atlas` uses table size of 8, and removes around $15/16$ data flushes, due to cache spatial locality. Hence, the data flush ratio is 0.0625, However, the software cache captures the best cache size, 26, which further optimizes the data flush ratio to only 0.00003. These numbers are shown in Table III.

The *queue* is a multithreaded benchmark we wrote based on the blocking algorithm of Michael and Scott [35]. The *hash* is a single-threaded open-source hash table [13]. The singly *linked-list* is a multi-threaded benchmark, whereby a total of N elements are inserted in a perfect shuffle pattern for a given number of elements added atomically at each step. A concurrent queue is perhaps the most common parallel data structure used by a concurrent application. We measure it as a micro-benchmark to show the performance of making a concurrent queue persistent. All micro-benchmarks can be found in the `Atlas` Github repository [21].

C. Case study: MDB

We use the `Mtest` workload, from the MDB test suite [12], to evaluate performance. The workload inserts 1 million key/value pairs along with many traversals and deletions. In the entire execution, there are 65558123 persistent memory stores. The number of durable FASEs is 100516. Each has 652 persistent memory stores on average.

Method	ER	AT	SC	SC-o	BEST
Time(sec)	24.58	8.36	4.84	4.39	3.54
Speedup	1	2.94x	5.07x	5.60x	6.94x

Table II: Execution times of `Mtest` on MDB. SC-o denotes SC-offline.

Table II shows the execution times of the five techniques and the speedups normalized to ER. The speedup of the state-of-the-art AT is 2.9×. SC is $5.1\times$ faster, further improving performance by $1.7\times$.

The performance improvement primarily results from the reduction in cache line flushes, as shown in Table III. *mdb* has 66 millions total writes. AT's write-back frequency is around 30% (20 millions), while SC reduces the number of write backs to 7.4 millions, 11% of total writes and 37% of AT's. The lower bound is LA, which has 3.3 millions write backs. However, due to LA flushes all cache lines at the end of a FASE, its performance is extremely bad. We will show an example later. That's why we didn't measure its execution time here.

Fewer write backs lead to smaller hardware cache miss ratios, as explained in Section II. For example, the L1 hardware cache miss ratio is 83.21% for AT and 68.50% for SC . The time difference between SC and SC-offline shows that the overhead of online cache size selection is 0.45 seconds, near 10% of execution time of SC. It is cost effective, since SC leads to almost 4 seconds reduction in running time, compared with AT.

D. The Write-Back Reduction

Table III shows statistics of all benchmarks under certain problem sizes, and compares the write-back ratios of the six techniques. The number of FASEs ranges from 1 to 300 thousands and the number of cache write backs ranges from 50 thousands to 391 millions.

The write-back frequency of ER is 1, since it flushes the cache line at every store. LA reaches the lowest possible, 16%, since it maximally combines write backs. However, since all cache lines are written back at the end of a FASE, CPU resources are wasted and hence performance is extremely bad. For example, for *volrend* program, LA is slower than AT by 17.8×. Hence, we omit the performance report of LA, but instead only show its data flush ratio to give the lower bound for comparison.

Benchmarks	Problem Size	Total FASEs	Total Flushes	ER	LA	AT	SC	AT/SC	SC/LA
linked-list	10000	10K	49999	1.00000	0.60001	0.60001	0.60001	1×	1×
persistent-array	100000	1	1000001	1.00000	0.00003	0.06250	0.00003	2083.333×	1×
queue	400000	300K	400006	1.00000	0.62500	0.62500	0.62500	1×	1×
hash	4000	7K	83061	1.00000	0.50092	0.62128	0.59531	1.044×	1.188×
barnes	16384	69K	270762562	1.00000	0.00295	0.08206	0.00391	20.987×	1.325×
fmm	16384	43K	87711754	1.00000	0.00246	0.01683	0.00328	5.131×	1.333×
ocean	1026	648	25242763	1.00000	0.09203	0.40290	0.16467	2.447×	1.789×
raytrace	car	346K	65509589	1.00000	0.07140	0.13952	0.07918	1.762×	1.108×
volrend	head	45	391692398	1.00000	0.00219	0.03189	0.00219	14.561×	1×
water-nsquared	512	2.1K	45338822	1.00000	0.00107	0.05334	0.00411	12.978×	3.748×
water-spatial	512	77	40981496	1.00000	0.00103	0.07122	0.00157	45.363×	1.524×
mdb	1000000	100K	65558123	1.00000	0.05163	0.30140	0.11289	2.669×	2.223×
average	-	65.1K	77420588	1.00000	0.16256	0.25066	0.18268	11.882×	1.427×

Table III: The benchmark statistics and data flush ratios of different techniques. The number of flushes is almost identical for SC and SC-offline, which is shown by SC. The average is arithmetic for all tests except for *persistent-array*, which is artificial, and *linked-list* and *queue*, which are already optimal.

AT vastly decreases data flushes, but there is still big room to improve. Excluding *persistent-array*, SC outperforms AT by $10\times$ significantly, as a result of selection of the best cache size. We will show that different programs choose different cache sizes in Section IV-G. SC is as good as AT on *linked-list* and *queue*, since they both achieve the lowest possible write-back frequency for the two programs. However, SC can choose the smallest cache size among all sizes that have the lowest possible. What's more, SC achieves the best for *persistent-array* and *volrend*. *volrend* especially shows the benefit. AT reduces the write-back frequency to 3%. Still, SC goes one step further and is able to remove all unnecessary write backs to the lowest possible (LA).

E. The Running Time

The performance in execution time is compared in Figure 4, which shows AT, SC, SC-offline and BEST by their speedups over ER. In this comparison, all programs are measured for single-thread runs except for *mdb*, which uses eight threads.

Over ER, the speedup of SC ranges from $1.4\times$ to $34.2\times$, with an average of $9.6\times$, Over half of them are over $5\times$, 4 programs are more than $15\times$, and 2 programs are over $20\times$. The greatest speedup is *volrend*, $34.2\times$. For comparison, AT outperforms ER by $4.5\times$ on average.

SC is uniformly better than AT. The average speedup over AT is $2.1\times$ and the greatest speedup reaches $4.3\times$ on *water-nsquared* program. The data flush ratio reduction shown in Table III explains the reason that SC significantly outperforms AT. For *persist-array*, though AT has $2082\times$ more flushes than SC, most of the flushes overlap with computation. Hence, the execution time of SC is just slightly less than AT.

SC-offline is faster than SC. The speedup of SC-offline over ER ranges from $1.9\times$ to $36.2\times$, with an average of $10.3\times$. SC is slightly slower than SC-offline, by 7% on average. The slowdown results from two parts. One is the overhead of online sampling of MRC. The other is that SC starts program execution with cache size of 8 by default, which results in worse performance than after selecting the best cache size. We will clearly show more data regarding the overhead of online cache size selection in Section IV-G.

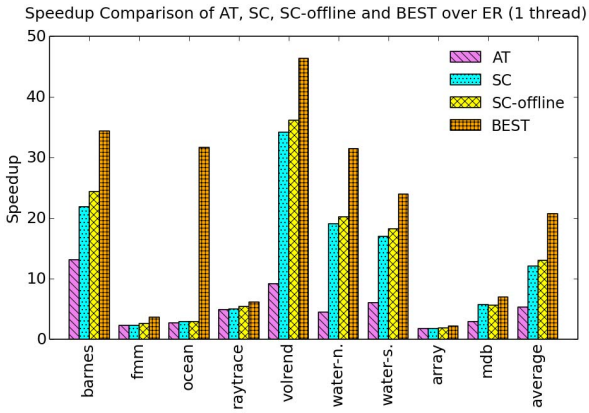

Figure 4: Comparison of ER, AT, SC, SC-offline and BEST performance, measured by speedups over ER. *water-n.* denotes *water-nsquared*. *water-s.* denotes *water-spatial*.

BEST shows the upper bound performance, which is $16.1\times$ speedup over ER. It is even faster than the optimal persistent caching, as explained in Section IV-A. SC is slower than BEST. The *ocean* is the worst, near 91% slowdown. In *mdb*, SC is very close to BEST, just 6% slower. On average, SC is 41% slower than BEST. The narrow difference sets a discouraging bound on any further improvement over SC but suggests SC is effectively near optimal in practice.

F. Parallel Performance

Figure 5 compares performance of SC and SC-offline with the state-of-art AT for different thread counts from 1 to 32. In 85% (36 out of 42) of tests, SC is better than AT.

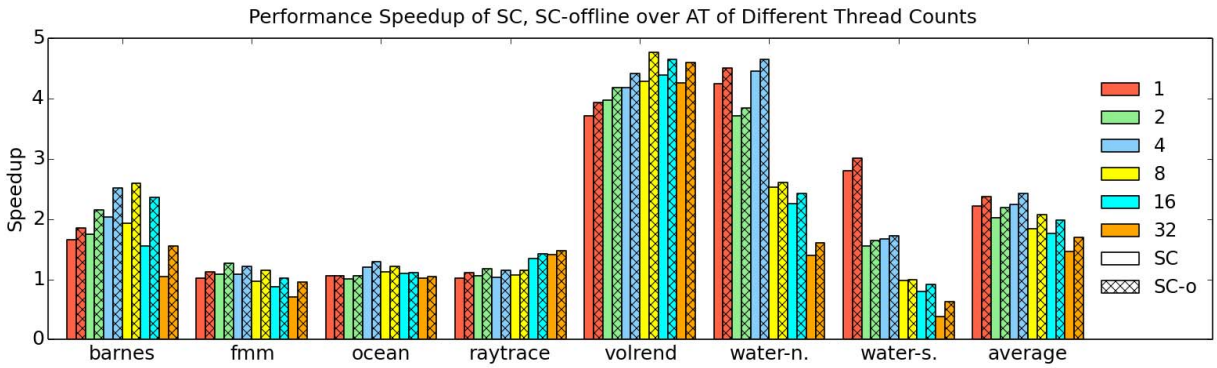

Figure 5: Parallel performance SC and SC-offline for different numbers of threads, shown as speedups over the parallel performance of AT. *water-n.* denotes *water-nsquared*. *water-s.* denotes *water-spatial*.

The greatest speedup is $4.13\times$ on *water-nsquared* with 4 threads. The speedup of SC over AT increases steadily with the number of threads for *raytrace* and *volrend* but decreases in the other programs. SC uniformly outperforms AT in all programs at small thread counts from 1 to 8. SC is better than AT for 16 and 32 threads counts for all programs but *fmm* and *water-spatial*. We will analyze the reason in details.

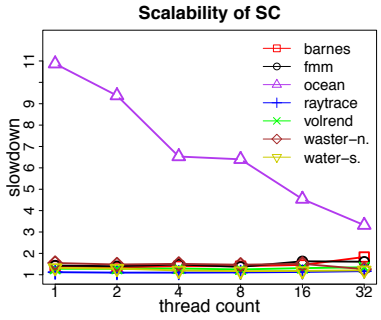

Figure 6: Slowdown of SC over BEST. *water-n.* denotes *water-nsquared*. *water-s.* denotes *water-spatial*.

We divide the overhead of adaptive caching in two parts: the instruction overhead and cache contention. The first part is measured by the performance slowdown of SC over BEST, which is shown in Figure 6 for 7 programs. *Ocean* starts with a large slowdown around 11 which drops linearly to 3. The slowdown for other programs are between 1 and 2 and is almost constant across the thread counts. The slowdown of *barnes* increases slightly at thread count 32. The measurement shows that the overhead of adaptive caching does not increase with the thread count.

Part of the instruction overhead is MRC analysis. As Figure 5 shows, SC-offline is better than AT in 90% (38 out of 42) of tests, 2 more than SC (36 out of 42). In these two tests, MRC analysis is counter productive, compared to AT. The loss, however, is very small.

The second overhead is cache contention at L1 data cache, as more threads are run sharing the same cache space. This is why SC performs worse than AT in *fmm* and *water-spatial*. We analyze *water-spatial* in detail in Table IV.

Threads		1	2	4	8	16	32
inst. (billions)	AT	3.34	3.39	3.47	3.55	3.67	3.96
	SC	3.55	3.66	3.71	3.79	3.96	4.32
	BE	2.56	2.60	2.63	2.66	2.79	2.96
flush ratio	AT	2.61%	4.03%	4.77%	5.53%	5.48%	5.91%
	SC	0.43%	0.44%	0.48%	0.55%	0.71%	1.00%
	BE	0%	0%	0%	0%	0%	0%
hw L1 cache mr	AT	58.16%	71.14%	73.95%	72.72%	75.97%	76.44%
	SC	30.78%	43.47%	58.99%	65.73%	68.48%	72.24%
	BE	20.25%	25.21%	33.79%	55.48%	63.31%	70.61%

Table IV: Performance of *water-spatial*: instruction count in billions, cache line flushes, measured in software, and L1 cache miss ratios, measured using Linux Perf tool [39]. BE denotes BEST.

Table IV shows performance analysis data of *water-spatial* program for different thread counts, either by software accounting or by hardware performance counters [39].

The SPLASH2 benchmark is strong scaling. A fixed amount of work partitioned among all threads. Hence, the total persistent memory stores stay almost the same regardless of the thread count. However, the amount of FASEs is proportional to the thread count. Since the software cache is cleared at the end of a FASE, more FASEs incur more (compulsory) software cache misses and cache line flushes. Therefore, the data flush ratio slightly increases with the number of threads. This is confirmed in Table IV by the flush ratios of SC.

We assumed that performance is proportional to the amount of cache line flushes in normal cases. The number of cache line flushes affects the hardware cache miss ratio, because the flushed data block is evicted from the hardware cache, as explained in Section II. By comparing flush ratios of SC and AT in Table IV, SC consistently outperforms AT by $6\times$ to $10\times$. Hence, SC should have had similar speedups over AT for different thread counts.

However, for *water-spatial*, hardware cache contention becomes severe as more threads are run. The contention

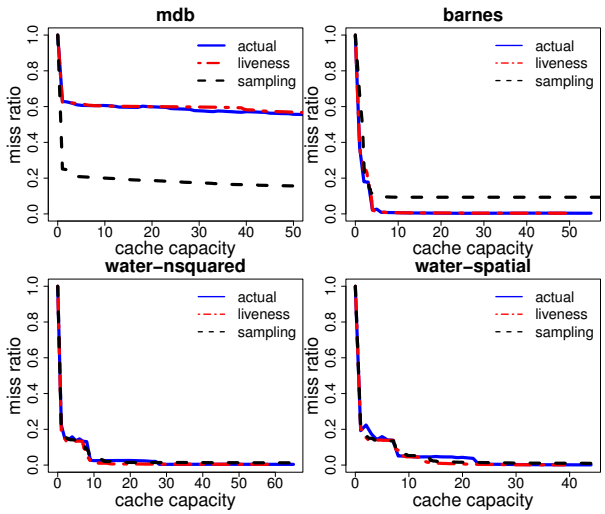

Figure 7: The comparison between actual MRC, full-trace (offline) MRC and sampled (online) MRC of four programs.

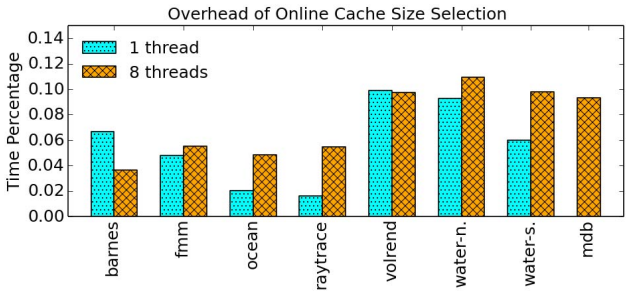

Figure 8: The time percentage of online cache selection overhead. *water-n.* denotes *water-nsquared*. *water-s.* denotes *water-spatial*.

is shown by L1 cache miss ratios of BEST in Table IV. BEST has no cache line flush, thus its L1 cache miss ratio only results from the contention caused by the program, not the software cache. Such contention might be caused by OS task scheduling. As a result of the contention, SC does not improve the L1 cache performance by reducing the data flush ratio. Hence, the performance is improved a little by SC flush reduction at large thread counts. In contrast, for this program, AT has similarly high hardware cache miss ratios for all thread counts, thus its performance relative to ER remains steady. Therefore, the speedup of SC over AT varies. The speedup variations of SC over AT in *barnes*, *fmm*, *ocean* and *water-nsquared* are due to similar reasons.

When hardware cache misses are similar, the number of instructions executed dominates performance. Because of the software cache, SC obviously runs more instructions than AT, which according to Table IV is 8% more on average. Therefore, AT is faster than SC at large thread counts, e.g. for *water-spatial*.

G. MRC Analysis Precision and Overhead

Figure 7 shows the accuracy of MRC prediction in four programs. Sampled MRC is not as precise as the accurate MRC. But in terms of cache size selection, it is sufficiently good, since the sampled MRC has the same inflection points as accurate MRC. Sampling overhead is negligible, which makes it fast and effective for online analysis.

Based on MRCs, the selected cache sizes of *barnes*, *fmm*, *ocean*, *raytrace*, *volrend*, *water-nsquared*, *water-spatial*, and *mdb* are 15, 10, 2, 8, 3, 28, 23 and 20, respectively. Surprisingly, these small cache sizes are sufficient to achieve very small data flush ratios, as shown in Table III. In addition, we can see that there is no one-fits-for-all solution for cache size selection. Cache size needs to be workload-aware. This is a major reason that SC outperforms AT.

Figure 8 shows the online overhead for 1 thread and 8 threads. To compute the overhead, we run SC with the best cache size and compute the difference of the running time between using the preset size and finding the size online. Sampling analyzes 64 millions memory stores and then compute the MRC and choose the best cache size. The cost of online sampling and adaptation is almost a fixed amount. The relative overhead ranges from 1% to 10%, due to different execution times in different programs. The average overhead is 0.52 seconds, and hence 6.78% of the program execution time for all thread counts.

V. Related Work

Systems such as Mnemosyne [43] and NV-heaps [43] built durable transactions on top of persistent regions. `Atlas` [9] uses a lock-based model, i.e. failure-atomic sections (FASEs). A FASE is more general than transactions because of nesting, which permits more parallelism as well as updates to persistent memory outside an atomic section. The design space of FASE-based persistence is explored later especially the trade-off between semantics and implementation efficiency [6]. In addition, Echo added durability to key-value stores [4], and Makalu supported recoverable memory management [5]. The notion of failure atomic updates has recently been extended by the formalism called durable linearizability, which guarantees safe composition of atomic operations and unifies persistent atomicity and recoverable linearizability [19].

A major obstacle to data persistence is the volatile cache. Some systems assume hardware whole-cache flushing at a power failure or machine crash. These include Echo [4] and JUSTDO logging, which resumes and finishes (lock-based) FASEs following a failure [18]. In the absence of whole-cache flushing, incremental cache write backs are necessary. Mnemosyne relied on hardware support, in particular the *x86 write-combining buffers* [1]. NV-heaps had short transactions and hence no need for write combining [43]. The software solution is pioneered in `Atlas` as a 8-entry table to store the address of modified cache lines. It combines data writes with

a minimal performance cost because of its simple design. In this work, we present the general solution of software write caching and uses the flexibility of software control to adapt the cache size based on how a program reuses its data.

We first discussed the basic idea in this work at a workshop [25]. To distinguish, this paper exclusively covers `Atlas` semantics, a complete description of reuse-based locality theory and its linear-time calculation and correctness proofs, system implementation, and rigorous evaluation. This paper studies writing caching. Chen et al. defined write locality by write reuse distance and write reuse time [10]. Brock et al. considered the asymmetric cost of read- and write-caching on persistent memory [7].

Mattson et al. [34] were first to define LRU stack reuse distance, which is commonly called reuse distance in short. The efficiency has steadily improved. Recent techniques include sampling [8, 17, 40, 44] and parallelization [37]. Xiang et al. [46] defined footprint as the average working-set size, and developed a higher order theory (*HOTL*) to formalize the relation between footprint, miss ratio and reuse distance. This work complements the footprint theory by exploring the relation between timescale reuse and miss ratio. Footprint and timescale reuse are both timescale metrics, which are a new type of statistics broadly studied recently [15, 16, 24, 26, 27, 28, 29, 30, 31, 32, 33].

VI. Summary

We have developed a software cache solution minimizing the cache flush overhead of NVRAM data persistence, using a reuse-based locality theory and a linear time algorithm for efficient online analysis to select the best cache size. We extensively evaluated the effect of write caching and found it efficient and effective, reducing the number of cache line flushes by 12× and improves the running time by 2.1× for both sequential and parallel applications.

Acknowledgment

This work started when the first author was a summer intern at Hewlett Packard Labs. The authors wish to thank Yu David Liu, Joseph Izraelevitz, John Criswell, and IPDPS reviewers for their comments. The research is supported in part by the US Department of Energy under Cooperative Agreement no. DESC0012199, the National Science Foundation (Contract No. CCF-1629376, CNS-1319617, CCF-1116104), IBM CAS Faculty Fellowship, a grant from Huawei, NFSC (No.61432018, No. 61402441, No. 61521092) and 863 Project (2015AA011505). Any opinions, findings, and conclusions or recommendations expressed in this material are those of the author(s) and do not necessarily reflect the views of the funding organizations.

References

[1] Inc. AMD. Software optimization guide for amd64 processors, 2005. *http://support.amd.com/TechDocs/25112.PDF*.

[2] Inc. ARM. Arm technical reference manual, 2015.

[3] Matthew Arnold and Barbara G. Ryder. A framework for reducing the cost of instrumented code. In *Proceedings of PLDI*, pages 168–179, Snowbird, Utah, June 2001.

[4] Katelin A. Bailey, Peter Hornyack, Luis Ceze, Steven D. Gribble, and Henry M. Levy. Exploring storage class memory with key value stores. In *Proceedings of the 1st Workshop on Interactions of NVM/FLASH with Operating Systems and Workloads*, 2013.

[5] Kumud Bhandari, Dhruva R. Chakrabarti, and Hans-Juergen Boehm. Makalu: fast recoverable allocation of non-volatile memory. In *Proceedings of OOPSLA*, pages 677–694, 2016.

[6] Hans-Juergen Boehm and Dhruva R. Chakrabarti. Persistence programming models for non-volatile memory. In *Proceedings of ISMM*, pages 55–67, 2016.

[7] Jacob Brock, Chencheng Ye, and Chen Ding. Replacement policies for heterogeneous memories. In *Proceedings of the International Symposium on Memory Systems*, pages 232–237, 2016.

[8] Calin Cascaval, Evelyn Duesterwald, Peter F. Sweeney, and Robert W. Wisniewski. Multiple page size modeling and optimization. In *Proceedings of PACT*, pages 339–349, 2005.

[9] Dhruva R. Chakrabarti, Hans-Juergen Boehm, and Kumud Bhandari. Atlas: leveraging locks for non-volatile memory consistency. In *Proceedings of OOPSLA*, pages 433–452, 2014.

[10] Dong Chen, Chencheng Ye, and Chen Ding. Write locality and optimization for persistent memory. In *Proceedings of the International Symposium on Memory Systems*, pages 77–87, 2016.

[11] T. M. Chilimbi and M. Hirzel. Dynamic hot data stream prefetching for general-purpose programs. In *Proceedings of PLDI*, pages 199–209, 2002.

[12] Howard Chu. MDB: A memory-mapped database and backend for openldap, 2014. http://www.openldap.org/pub/hyc/mdm-paper.pdf.

[13] C. Clark. A hash table data structure in c., 2005. https://github.com/davidar/c-hashtable.

[14] Peter J. Denning and Stuart C. Schwartz. Properties of the working set model. *Communications of the ACM*, 15(3):191–198, 1972.

[15] Chen Ding and Pengcheng Li. Cache-conscious memory management. In *Proceedings of the ACM SIGPLAN Workshop on Memory System Performance and Correctness*, 2014.

[16] Chen Ding and Pengcheng Li. Timescale stream statisitcs for hierarchical management. In *STREAM workshop*, 2016.

[17] David Eklov and Erik Hagersten. StatStack: Efficient modeling of LRU caches. In *Proceedings of ISPASS*, pages 55–65, 2010.

[18] Joseph Izraelevitz, Terence Kelly, and Aasheesh Kolli. Failure-atomic persistent memory updates via JUSTDO logging. In *Proceedings of ASPLOS*, pages 427–442, 2016.

[19] Joseph Izraelevitz, Hammurabi Mendes, and Michael L. Scott. Linearizability of persistent memory objects under a full-system-crash failure model. In *Proceedings of the International Conference on Distributed Computing*, pages 313–327, 2016.

[20] Linux OS kernel. Understanding the linux virtual memory manager, 2007. *https://www.kernel.org/doc/gorman/pdf/understand.pdf*.

[21] Hewlett Packard Labs. Atlas: Programming for persistent memory, 2016. *https://github.com/HewlettPackard/Atlas*.

[22] Chris Lattner and Vikram S. Adve. Automatic pool allocation: improving performance by controlling data structure layout in the heap. In *Proceedings of PLDI*, pages 129–142, 2005.

[23] Stephen Lawson. Intel and micron unveil 3d xpoint – a new class of memory, 2015. http://www.computerworld.com/article/2951869/computer-hardware/intel-and-micron-unveil-3d-xpoint-a-new-class-of-memory.html.

[24] P. Li, H. Luo, C. Ding, Z. Hu, and H. Ye. Code layout optimization for defensiveness and politeness in shared cache. In *Proceedings of ICPP*, 2014.

[25] Pengcheng Li and Dhruva R. Chakrabarti. Adaptive software caching for efficient NVRAM data persistence. In *Proceedings of the LCPC Workshop*, pages 93–97, 2016.

[26] Pengcheng Li and Chen Ding. All-window data liveness. In *Proceedings of the ACM SIGPLAN Workshop on Memory Systems Performance and Correctness*, 2013.

[27] Pengcheng Li, Chen Ding, and Hao Luo. Modeling heap data growth using average liveness. In *Proceedings of the 2014 International Symposium on Memory Management*, 2014.

[28] Pengcheng Li, Hao Luo, and Chen Ding. Rethinking a heap hierarchy as a cache hierarchy: a higher-order theory of memory demand (hotm). In *Proceedings of the 2016 ACM SIGPLAN International Symposium on Memory Management*, 2016.

[29] Hao Luo, Guoyang Chen, Pengcheng Li, Chen Ding, and Xipeng Shen. Data-centric combinatorial optimization of parallel code. In *Proceedings of PPoPP*, 2016. *poster paper*.

[30] Hao Luo, Chen Ding, and Pengcheng Li. Optimal thread-to-core mapping for pipeline programs. In *Proceedings of the ACM SIGPLAN Workshop on Memory System Performance and Correctness*, 2014.

[31] Hao Luo, Pengcheng Li, and Chen Ding. Parallel data sharing in cache: Theory, measurement and analysis. Technical Report URCS #994, Department of Computer Science, University of Rochester, December 2014.

[32] Hao Luo, Pengcheng Li, and Chen Ding. Thread data sharing in cache: Theory and measurement. In *Proceedings of PPoPP*, 2017.

[33] Hao Luo, Chencheng Ye, Pengcheng Li, and Chen Ding. Composable modeling of coherence and numa effects for optimizing thread and data placement. In *Proceedings of ISPASS*, 2016. *poster paper*.

[34] R. L. Mattson, J. Gecsei, D. Slutz, and I. L. Traiger. Evaluation techniques for storage hierarchies. *IBM System Journal*, 9(2):78–117, 1970.

[35] Maged M. Michael and Michael L. Scott. Simple, fast, and practical non-blocking and blocking concurrent queue algorithms. In *Proceedings of the ACM Symposium on Principles of Distributed Computing*, pages 267–275, 1996.

[36] Sparsh Mittal. A survey of power management techniques for phase change memory. *Int. J. of Computer Aided Engineering and Technology*, 2014.

[37] Qingpeng Niu, James Dinan, Qingda Lu, and P. Sadayappan. PARDA: A fast parallel reuse distance analysis algorithm. In *Proceedings of IPDPS*, 2012.

[38] OpenLDAP. http://www.openldap.org/.

[39] perf: Linux profiling with performance counters. https://perf.wiki.kernel.org/index.php/Main_Page.

[40] Derek L. Schuff, Milind Kulkarni, and Vijay S. Pai. Accelerating multicore reuse distance analysis with sampling and parallelization. In *Proceedings of PACT*, pages 53–64, 2010.

[41] Xipeng Shen, Jonathan Shaw, Brian Meeker, and Chen Ding. Locality approximation using time. In *Proceedings of POPL*, pages 55–61, 2007.

[42] Peter Snyder. tmpfs: A virtual memory file system. In *In Proceedings of the Autumn 1990 European UNIX Users Group Conference*, 1990.

[43] Haris Volos, Andres Jaan Tack, and Michael M. Swift. Mnemosyne: Lightweight persistent memory. In *Proceedings of ASPLOS*, 2011.

[44] Carl A. Waldspurger, Nohhyun Park, Alexander Garthwaite, and Irfan Ahmad. Efficient mrc construction with shards. In *Proceedings of the 13th USENIX Conference on File and Storage Technologies*, 2015.

[45] Steven Cameron Woo, Moriyoshi Ohara, Evan Torrie, Jaswinder Pal Singh, and Anoop Gupta. The splash-2 programs: Characterization and methodological considerations. In *Proceedings of ISCA*, 1995.

[46] Xiaoya Xiang, Chen Ding, Hao Luo, and Bin Bao. HOTL: a higher order theory of locality. In *Proceedings of ASPLOS*, pages 343–356, 2013.