Modernizing Models and Management of the Memory Hierarchy for Non-Volatile Memory

(Thesis Proposal)

Charles McGuffey
Computer Science Department
Carnegie Mellon University

Thesis Committee
Phillip Gibbons (Chair)
Guy Blelloch
Nathan Beckmann
Julian Shun (MIT)
Michael Bender (SUNY Stony Brook)

ABSTRACT

Non-volatile memory technologies (NVMs) are a new family of technologies that combine near memory level performance with near storage level cost density. The result is a new type of memory hierarchy layer that exists and performs somewhere between the two. These new technologies offer many opportunities for performance improvements, but bring many of their own unique challenges.

In this thesis, we focus on how the introduction of NVMs affects memory management and caching. Our work is broken into four primary categories. 1. We study how fault tolerance can be achieved at a program level through the use of NVM memory technologies that survive faults. 2. We extend the traditional model of caching to account for the data writes that become more significant in NVM memories. 3. We consider a variant caching model with a cache that works at the granularity of lines above a storage layer that works at block granularity. 4. We approach the increasing difficulty of caching problems in the modern hierarchy using the tools available in information theory to close the gap between theoretical and practical results.

Throughout our work we rely on a blend of theoretical and practical approaches. We provide models for processor faults, cache writebacks, cache-storage communication, and trace complexity that isolate the targeted effects from orthogonal complications. For each model, we show worst case theoretical bounds for our algorithms along with proofs that explain how the benefits are derived. We then take our results and provide empirical evaluations to show their effectiveness in practice. We believe that our ideas and approach provide a solid foundational study on memory hierarchy design in the era of non-volatile memories.

1. INTRODUCTION

In the modern era, computing continues to grow at an astounding rate. This growth requires continuous improvements in both processing and data management. Unlike processing, storage and memory technologies have shown slow growth rates in performance. This performance gap has been bridged through the use of the memory hierarchy: a collection of different data management layers that progress from large, cheap, and slow to small, expensive, and fast.

Traditionally, the layers of the memory hierarchy were divided between storage and memory. Storage is where data resides permanently; it is slower and larger, using an interface where data are accessed in blocks. Memory is used to provide data access to processing; it is smaller and faster, and can be accessed on a cache line or byte granularity. These layers are traditionally separated by a file system or other data management interface that translates between their separate access styles.

This traditional partitioning of the memory hierarchy is being upended by the arrival of non-volatile memories (NVMs). NVMs have performance within an order of magnitude of existing random access memory (DRAM), have low idle power consumption, have large capacity (more bits per unit area than existing random access memory), and have the capability of surviving power outages and other failures without losing data (the memory is persistent).

NVM technologies that use several different versions of storing data in the physical state of material are being developed [89] or are already available [40, 63] These technologies are currently available in both solid state drive (SSD) and memory module (DIMM) form factors. Although these forms ostensibly fit into the categories of storage and memory, respectively, they have many characteristics that differ from their traditional counterparts.

In this thesis, we investigate how the introduction of NVMs to the memory hierarchy affects caching and data management. In addition to changes specific to NVMs, we extend our analysis to previously existing aspects of the memory hierarchy whose importance has increased due these new devices and the demands that necessitated their conception. Our work is broken into the following sections:

Program Persistence. Memory level NVM devices provide memory tier performance while surviving power failures. This provides an opportunity to achieve program-level fault tolerance without the expense of saving state into storage. We study this opportunity and provide an approach that provides theoretical guarantees on both consistency and progress.

Writeback-Aware Caching. For traditional memory technologies, reads and writes are usually treated as equivalent operations. However, this is not true for most NVM technologies. Writes are more expensive than reads in terms of latency, bandwidth, energy, and device lifetime. This is further exacerbated by the increasing importance of energy and bandwidth consumption in computers. Motivated by these changes, we extend the traditional caching model to account for the cost of writes, and provide an investigation of the resulting model.

Block-Aware Caching. Storage systems and memory system rely on a fundamentally different interface: storage manages data in units of blocks while memory works at cache line or byte granularity. This distinction is blurred by NVM technologies, which often support cache line granularity but perform best with accesses in the size of KBs. We study how caching can help support this interface tradition, with a model and associated results for line granularity caches that work above block granularity storage levels.

Applying Information Theory to Caching. As the storage systems that we develop become more complex, so too does their analysis and the difficulty of coming up with cache management strategies that can handle the variety of different demands set upon them. Furthermore, as our computing demands become more stringent, it becomes more important to close the gap between practical and theoretical performance. To improve our understanding of these difficult problems and what gains remain available to collect, we apply the techniques of information theory to real-world cache traces and replacement policies.

When combined, these related research areas support the following thesis: **With the advent of non-volatile memory technologies, many new and understudied aspects of memory management become crucial for achieving fast and efficient performance**.

2. PROGRAM PERSISTENCE

An important consequence of the prevalence and expansion of computing in the modern era is that systems are increasing in size and parallelism. In such systems, the probability that individual components fault is not negligible [30].

Traditionally, checkpointing and other fault handling techniques managed this by periodically storing data to storage or other redundant systems. However, non-volatile main memories provide an opportunity to minimize the cost of fault tolerance by persisting data in memory.

To investigate this opportunity, we define a parallel computational model, the *Parallel Persistent Memory (Parallel-PM) model*, that consists of several processors with a fast local ephemeral memory of limited size, and sharing a large slower persistent memory. As in the external memory model [7, 6], each processor runs a standard instruction set from its ephemeral memory and has instructions for transferring blocks to and from the persistent memory. The cost of an algorithm is calculated based on the number of such transfers. A key difference, however, is that the model allows for individual processors to fault at any time. If a processor faults, all of its processor state and local ephemeral memory is lost, but the persistent memory remains. We consider both the case where the processor restarts (soft faults) and the case where it never restarts (hard faults). Our model captures a useful view of the interaction of NVM main memory with traditional memory technologies and allows us to provide insight into interesting and theoretically guaranteed fault tolerance.

Our Contributions (Completed). In this work, we study how to handle processor faults in systems with NVRAM. We consider both how an individual processor should restart, and how to run a computation that can continue to make progress despite faults. We make the following contributions:

1. We define the *Persistent Memory Model*, a single processor model for processor faults and their effect on memory. That model allows us to devise a technique to limit the progress lost due to faults. We then define the *Parallel Persistent Memory Model*, which extends the Persistent Memory Model to support multiple processors.

2. We identify a technique where breaking a computation into "capsules" that have no write-after-read conflicts (writing a location that was read earlier within the same capsule) provides idempotent behavior in our single processor model. We then use this technique to implement RAM, external memory, and cache-oblivious algorithms [51] asymptotically efficiently in the model.

3. The implementations of algorithms from other models is asymptotically efficient, but unlikely to be practical. We therefore consider a programming methodology in which the algorithm designer can identify capsule boundaries to ensure that the capsules are free of write-after-read conflicts.

4. We extend our ideas to the Parallel Persistent Memory Model, and consider conditions under which programs are correct when the processors are interacting through the shared memory. We identify that if capsules are free of write-after-read conflicts and atomic, in a way that we define, then each capsule acts as if it ran once despite many possible restarts.

5. Our most significant result is a work-stealing scheduler, based on that of Arora, Blumofe, and Plaxton (ABP) [7], that can be used on the Parallel Persistent Memory Model. This scheduler (i) ensures that each stolen task gets executed despite faults, (ii) properly handles hard faults, and (iii) remains efficient in the presence of soft or hard faults. We use our scheduler to show that any race-free, write-after-read conflict free, multithreaded fork-join program can be scheduled in bounded runtime. This bound differs from the ABP result only by a logarithmic factor on the depth term, due to faults along the critical path.

6. We apply our prior techniques to achieve algorithms for prefix-sum, merging, sorting, and matrix multiply that are idempotent in the Parallel Persistent Memory Model. The results for prefix-sums, merging, and sorting are work-optimal, matching lower bounds for the external memory model.

Related Work. When a processor crashes, writes that are still in the cache (have not been recorded in persistent memory) are lost while other writes are not. Prior work includes schemes for encapsulating updates to persistent memory in either *transactions* or *lock-protected failure atomic sections* and using various forms of (undo, redo, resume) logging to ensure correct recovery [22, 42, 62, 65, 59, 31, 78, 83, 61, 37, 110, 95, 32, 77, 90, 53, 55, 82, 33, 20, 50].

The intermittent computing community works on the related problem of small systems that will crash due to power loss [84, 39, 10, 38, 57, 108, 28, 87]. Lucia and Ransford [84] describe how faults and restarting lead to errors that will not occur in a faultless setting. Several of these works [84, 39, 38, 108, 87] break code into small chunks, referred to as *tasks*, and work to ensure progress at that granularity. Avoiding write-after-read conflicts is often the key step towards ensuring that tasks are idempotent. Because these works target intermittent computing systems, which are designed to be small and energy efficient, they do not consider multithreaded programs, concurrency, or synchronization.

In contrast to this flurry of systems research, there is relatively little work from the theory/algorithms community aimed at this setting [41, 67, 66, 92]. David et al. [41] presents concurrent data structures (e.g., for skip-lists) that avoid the overheads of logging. Izraelevitz et al. [67, 66] presents efficient techniques for ensuring that the data in persistent memory captures a consistent cut in the happens-before graph of the program's execution. Nawab et al. [92] defines periodically persistent data structures, which combine mechanisms for tracking proper write ordering with a periodic flush of all cache lines to persistent memory. None of this work defines an algorithmic cost model, presents a work-stealing scheduler, or provides the provable bounds in this paper.

There is a very large body of research on models and algorithms where processors and/or memory can fault, but to our knowledge, none of it (other than the works mentioned above) fits the setting we study with its two classes of memory (local volatile and shared nonvolatile). Papers focusing on memory faults (e.g., [49, 1, 34] among a long list of such papers) consider models in which individual memory locations can fault. Papers focusing on processor faults (e.g., [9] among an even longer list of such papers) either do not consider memory faults or assume that all memory is volatile.

2.1 Model Definition

Single Processor. Our memory model has two layers: a small fast *ephemeral memory* of size M (in words) and a large slower *persistent memory* of size $M_p \gg M$, which are both partitioned into blocks of B words. We assume standard RAM instructions, as well as an *external read* that transfers a block from persistent memory into ephemeral memory, and an *external write* that transfers a block from ephemeral memory to persistent memory. These assumptions mirror those in the (M, B) external memory model [2].

Our fault model assumes that the processor can *fault* between any two external memory instructions with constant and independent probability. After faulting, the processor *restarts*, with the ephemeral memory and processor registers in an arbitrary state, but the persistent memory in the same state as immediately before the fault. To enable forward progress, we assume there is a fixed persistent memory location referred to as the *restart pointer location*, containing a *restart pointer*, which is used to set the program counter on restart. The processor can update this pointer during execution. We view the computation as being partitioned into *capsules* that correspond to maximally contiguous sequences of instructions with the same restart pointer. We refer to writing a new restart pointer as *installing* a capsule, and assume that each restart pointer points to the beginning of its capsule. We define the *capsule work* to be the number of external reads and writes in the capsule, and the capsule whose restart pointer is installed as *active*.

Our cost model can be adapted to various instruction costs, but for this work we follow the external memory [2] and ideal cache [51] models in assuming that external reads and writes take unit cost and all other instructions have no cost. We assume that processor faults and restarts have constant cost, since the machine downtime is outside of software control and the restarting process can be made to take a constant number of external memory transfers.

In our analysis, we consider two ways to count the total cost. We say that the *faultless work* (or *work*), W, is the number of external memory transfers assuming no faults. We say that the *total work* (or *fault-tolerant work*), W_f, is the number of external transfers for an actual run including all transfers due to having to restart.

We refer to the result of aggregating these memory, fault, and cost models as the (single processor) (M, B) Persistent Memory Model (PM model).

Multiple Processors. The Parallel Persistent Memory Model (Parallel-PM model) consists of P processors each with its own fast local ephemeral memory of size M, but sharing a single slower persistent memory of size M_p (see Figure 2.1). Each processor works as in the single processor model, and the processors run asynchronously. Any processor can fault between instruction. If the processor restarts, using its own restart pointer location as in the single processor model, it is called a *soft fault*. We also allow for a *hard fault*, in which the processor never restarts—

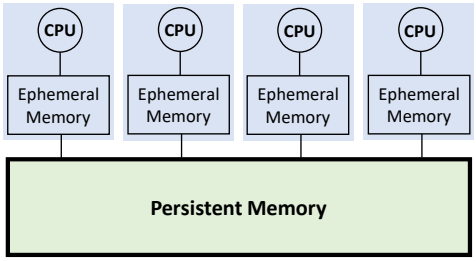


Fig. 2.1: The Parallel Persistent Memory Model

we say that such a processor is *dead*. We allow for concurrent reads and writes to the shared persistent memory, as well as a compare-and-swap (CAS) instruction. All of these operations are assumed to be sequentially consistent.

We consider the same form of multithreaded computations as considered by Arora, Blumofe, and Plaxton (ABP) [7]. In the model, a computation starts as a single thread. On each step, a thread can run an instruction, fork a new thread, or join with another thread. The (faultless) work W is the work summed across all threads in the absence of faults, and the total work W_f is the summed work including faults. In addition, we define the (faultless) *depth* D (and the *fault-tolerant* or *total depth* D_f) to be the maximum work (total work, respectively) along any path in the DAG.

Write-back Caches. Note that while the PM models are defined using explicit external read and external write instructions, they are also appropriate for modeling the (write-back) cache setting. Explicit instructions are used to ensure that external writes get to the persistent memory. Writes to local memory could end up being evicted from the cache and written back to persistent memory, but this does not affect correctness for programs that are race-free and well-formed, as defined in Section 2.2.

2.2 Robustness on a Single Processor

Our goal is to partition the computation into capsules in a way that ensures correctness regardless of faults. Specifically, we want each capsule to look like it has been run exactly once after its completion from an external view, regardless of the number of times it was restarted. We say that a capsule is *idempotent* if, when it completes, all modifications to the persistent memory are consistent with running once from the initial state.

We say that a capsule has a *write-after-read conflict* if the first transfer from a block in persistent memory is a read (called an "exposed" read), and later there is a write to the same block. Such conflicts cause problems on restarts because the original input data may have been overwritten. We say a capsule is *well-formed* if the first access to each ephemeral word in the registers is a write. Being well-formed means that a capsule will not read the undefined values from registers and ephemeral memory after a fault. We say that a capsule is *write-after-read conflict free* if it is well-formed and had no write-after-read conflicts. Our paper [24] provides a proof that all such capsules are idempotent in the single processor model.

Theorem 1. *With a single processor, all write-after-read conflict free capsules are idempotent.*

We also show that it is possible to simulate models such as the RAM model, the external memory model, and the ideal cache model efficiently in the persistent memory model. Proofs for these theorems can be found in the full version of our paper [24].

Theorem 2. *Any RAM computation taking t time can be simulated on the $(O(1), B)$ PM model with $f \leq 1/c$ for some constant $c \geq 2$, using $O(t)$ expected total work, for any B ($B = 1$ is sufficient).*

Theorem 3. *Any (M, B) external memory computation with t external accesses can be simulated on the $(O(M), B)$ PM model with $f \leq B/(cM)$ for some constant $c \geq 2$, using $O(t)$ expected total work.*

Theorem 4. *Any (M, B) ideal cache computation with t cache misses can be simulated on the $(O(M), B)$ PM model with $f \leq B/(cM)$ for a constant $c \geq 2$, using $O(t)$ expected total work.*

2.3 Programming for Robustness

The simulation of external memory is not completely satisfactory because its overhead, although constant, could be significant. Here we describe one protocol to program directly for the model, which can greatly reduce the overhead.

Our protocol is designed so capsules begin and end at the boundaries of certain function calls. We call functions with capsule boundaries *persistent function calls* and those without ephemeral functions. We assume that all user code between persistent boundaries is write-after-read conflict free, or otherwise idempotent. In addition to the persistent function call we assume a `commit` command that forces a capsule boundary at that point. As with a persistent call, this command requires a constant number of external reads and writes.

Implementing persistent function calls requires some care with a standard stack protocol. Our full paper [24] shows two different implementations, one that modfies a standard stack discipline, and one that uses closures [5].

2.4 Robustness on Multiple Processors

Since our previous definition of idempotent is inadequate, we consider a stronger variant of idempotency that requires that the capsule acts as if it ran atomically. Atomicity is not necessary for correctness, but provides a simple definition that is sufficient for our purposes.

We consider the history of a computation, which is an interleaving of the persistent memory instructions from each of the processors. The history includes the additional instructions due to faults (i.e., it is a complete trace of instructions that actually happened). A capsule within a history is *invoked* at the instruction it is installed and *responds* at the instruction that installs the next capsule on the processor.

We say that a capsule in a history is *atomically idempotent* if all its instructions can be moved in the history to be adjacent somewhere between its invocation and response without violating the memory semantics (atomicity), and the instructions are idempotent at the spot they are moved to (idempotency).

We say that two persistent memory instructions on separate processors *conflict* if they operate on the same block and one is a write. For a capsule within a history we say that one of its instructions has a *race* if it conflicts with another instruction that is between the invocation and response of that capsule.

Theorem 5. *Any capsule that is write-after-read conflict free and race free in a history is atomically idempotent.*

Proof. Because the capsule is race free we can move its instructions to be adjacent at any point between the invocation and response without affecting the memory semantics. Once moved to that point, the idempotence follows from Theorem 1 because the capsule is write-after-read conflict free. □

This property is useful for user code if one can ensure that the capsules are race free via synchronization. We use this extensively in our algorithms. However, races are required for program synchronizations. We therefore consider other conditions that are sufficient for atomic idempotence.

1. **Racy Read Capsule.** This capsule contains one read from persistent memory, which is allowed to race, and writes its value to another location in persistent memory. The capsule can have other instructions, but none of them can depend on the value that is read.

2. **Racy Write Capsule.** In this capsule, the only instruction with a race is a write instruction to persistent memory, and the instruction can only race with either read instructions or other write instructions, but not both kinds.

3. **Compare-and-Modify (CAM) Capsule.** In our model, it is difficult to perform an idempotent CAS. Instead, we use a *compare-and modify (CAM)*, which is simply a CAS for which no subsequent instruction in the capsule reads the local result. A *CAM capsule* is a capsule that contains one CAM (with protection against the ABA problem) and may contain other write-after-read conflict free and race free instructions. Ben-David et al. [18] examine the performance of CAS in more detail and provide a way to construct an idempotent version. Such an operation could be used in this capsule in place of a CAM.

4. **Racy Multiread Capsule.** This capsule consists of multiple racy read capsules that have been combined together into a single capsule. It is an example of how to design capsules that are idempotent without the requirement of atomicity.

These capsules provide a strong foundation for building program synchronizations. In fact, our work-stealing scheduler is constructed using these capsules. Our paper [24] shows how each capsule is idempotent.

2.5 Work Stealing

We provide an efficient work stealing scheduler (WS) for the Parallel-PM model based on the work-stealing scheduler of Arora, Blumofe, and Plaxton (ABP) [7]. Our scheduler, like theirs, uses a work-stealing double ended work queue and works in a multiprogrammed environment where the number of active processors can change. However, care is required to ensure that both hard and soft faults can be handled correctly and efficiently, regardless of whether they occur in the computation being scheduled or the scheduler itself.

Scheduling Algorithm Overview. Our scheduler relies on a work-stealing deque (WS-deque). The deque interface supports the `popTop`, `pushBottom`, and `popBottom` functions. Any number of concurrent processors can execute `popTop`, but only the deque owner can execute either `pushBottom` or `popBottom`. The deque is linearizable except that `popTop` can return empty on a non-empty deque if another concurrent `popTop` succeeds while the failed one is in process.

Each deque contains an array of entries that refer to threads that the processor has either enabled or stolen while working on the computation, as well as pointers to the top and bottom of the deque. An *entry* consists of a tag used to avoid the ABA problem [56] and one of the following states:

- *empty*: An entry not yet associated with a thread. Newly created elements in the array are initialized to empty.
- *local*: Refers to a thread currently being run by the owner of this WS-Deque. Used to deal with hard faults.
- *job*: Contains a thread that is ready to be computed or stolen.
- *taken*: Refers to a thread that is being or has been stolen. Used to ensure steal finish.

The way that we implement our functions and entry array ensures that there is a consistent structure that contains both thread and processor state. The maintenance of this consistent structure is useful for proving the correctness and efficiency of the WS-Deque.

Each process is initialized with an empty WS-Deque containing `empty` entries. The process that is assigned the root thread installs the first capsule of this thread, and sets its first entry to `local`.

During computation, processes that have work to do will perform that work while marking the bottom of their deque as `local`. Processes that do not have work will first try to take a thread from their own deque using `popBottom`. If `popBottom` fails, the process will pick a random victim and try to steal work from the victim using `popTop`. Successful steals install a `taken` entry in the victim's deque, which allows other would-be thieves to help the steal. To handle hard faults, we allow `local` entries to be stolen from dead victims.

A full proof of correctness, dealing with the many possible code interleavings that arise when considering combinations of faulting and concurrency, can be found in the full version of the paper [24].

Time Bounds. We consider the total amount of work done by a computation, and the depth of the computation, also called the critical path length. In our case we have W, the work assuming no faults, and W_f, the work including faults. In algorithm analysis the user analyzes the first, but in determining the runtime we care about the second. Similarly we have both D, a depth assuming no faults, and D_f, a depth with faults.

Our proof of the scheduler's time bounds follows that of ABP. However, we need additional analysis to prove that every P steal attempts will result in a constant fraction of non-empty deques being stolen from with constant probability. We also need to do account for the additional work due to faults, which we do using a high-probability bound. A more in-depth discussion can be found in our paper [24].

Theorem 6. *Consider any multithreaded computation with W work, D depth, and C maximum capsule work (all assuming no faults) for which all capsules are atomically idempotent. On the Parallel-PM with P processors, P_A average number of active processors, and fault probability bounded by $f \leq 1/(2C)$, the expected total time T_f for the computation is*

$$O\left(\frac{W}{P_A} + D\left(\frac{P}{P_A}\right)\left\lceil \log_{1/(Cf)} W \right\rceil\right).$$

This time bound differs from the ABP bound only in the extra $\log_{1/(Cf)} W$ factor. If we assume P_A is a constant fraction of P then the expected time simplifies to $O(W/P + D\lceil \log_{1/(Cf)} W \rceil)$.

2.6 Fault-Tolerant Algorithms

We have implemented several nested parallel algorithms for the Parallel-PM model. The algorithms that we use are already race-free. Making them write-after-read conflict free simply involves ensuring that reads and writes are to different locations. In-depth descriptions of the algorithms and their bounds can be found in our paper [24].

Prefix Sum. Given n elements $\{a_1, \cdots, a_n\}$ and an associative operator "+", the prefix sum algorithm computes a list of prefix sums $\{p_1, \cdots, p_n\}$ such that $p_i = \sum_{j=1}^{i} a_j$. Prefix sum is one of the most commonly-used building blocks in parallel algorithm design [71].

We modify the standard prefix sum algorithm [71] by placing the body of each function call (without the recursive calls) in an individual capsule. This provides write-after-read conflict-freedom and limits the maximum capsule work to a constant.

Theorem 7. *The prefix sum of an array of size n can be computed in $O(n/B)$ work, $O(\log n)$ depth, and $O(1)$ maximum capsule work, using only atomically-idempotent capsules.*

Merging. A merging algorithm takes two sorted arrays A and B of size l_A and l_B ($l_A + l_B = n$), and returns a sorted array containing the elements in both input lists. Our algorithm is based on the classic divide-and-conquer approach [25].

The first step is to allocate the output array of size n. Then parallel dual binary searches are conducted to find elements with rank that is a multiple of $n^{2/3}$. These are used to partition the arrays into subarrays, which can be recursed on. Each binary search and each base case are in their own capsule.

Theorem 8. *Merging two sorted arrays of overall size n can be done in $O(n/B)$ work, $O(\log n)$ depth, and $O(\log n)$ maximum capsule work, using only atomically-idempotent capsules.*

Sorting. We outline a samplesort algorithm based on Blelloch et al. [25]. The algorithm first splits the elements into subarrays of size $\sqrt{n}$ and recursively sorts each subarray. Recursion terminates when the subarray size is less than M, and the algorithm then sequentially sorts within a single capsule. Then the algorithm samples and sorts every $\log n$'th element from each subarray. $\sqrt{n}$ pivots with fixed stride are picked from the result and used to determine bucket boundaries within each subarray. We use our prefix sum algorithm and the divide-and-conquer bucket transpose algorithm from [25] to move the keys to buckets. The elements are then sorted within each bucket. All steps can be made write-after-read conflict free by writing to locations separate than those being read.

Theorem 9. *Sorting n elements can be done in $O(n/B \cdot \log_M n)$ work, $O((M/B + \log n) \log_M n)$ depth, and $O(M/B)$ maximum capsule work, using only atomically-idempotent capsules.*

Matrix Multiplication. Our algorithm is a slight modification of the classic 8-way divide-and conquer approach [51]. The computation is made race-free by setting correct capsule boundaries.

Theorem 10. *Multiplying two square matrices of size n can be done in $O(n^3/(B\sqrt{M}))$ work, $O(M/B + \log^2 n)$ depth, and $O(M/B)$ maximum capsule work, using only atomically-idempotent capsules.*

3. WRITEBACK-AWARE CACHING

The long history of papers on caching problems [4, 11, 15, 16, 17, 35, 36, 44, 46, 48, 69, 72, 75, 88, 99, 102, 114, 115] has largely overlooked an increasingly important cost in real caches: the cost of *writebacks*. Any data item that has been modified since being fetched into the cache (i.e., a *dirty* item) must be written back to memory on eviction. In contrast, a data item that has *not* been updated since being fetched (a *clean* item) can simply be discarded from the cache on eviction. Although largely ignored by real-world cache replacement policies in the past, writebacks are becoming increasingly important in real memory systems:

Most NVM technologies store data in the physical state of material [89], including the Intel Optane (3D-Xpoint) technology that is available today. This means that writing data into these memories requires more time and energy than reading data, sometimes by an order of magnitude or more [60, 68, 76, 98, 109]. Furthermore, these technologies have limited write endurance, so reduced writes mean increased device longevity. Caches in front of such devices should therefore consider the costs of such writes.

Traditional memory systems were designed to minimize response time, with replacement policies designed to maximize the number of cache hits. Modern processors, however, have greatly increased their instruction throughput by increasing parallelism (number of cores) rather than increasing clock frequency. For memory-intensive programs, the number of concurrently in-flight memory requests grows linearly with the number of cores, such that the available memory bandwidth is often the primary performance bottleneck. Moreover, these additional requests combined with the end of Dennard scaling [26, 43] has caused power consumption to become critical for computing systems ranging from exascale computing [107] to microcomputing [39]. The practical importance of these metrics has been underscored by a significant amount of systems research [81, 106, 111]. Reducing writebacks reduces the strain on memory system bandwidth and significantly reduces power consumption [58, 81].

Motivated by the lack of research in this blossoming performance consideration, we study the effects of writebacks in caching. We extend traditional caching models to account for this new variable and show a variety of results.

Our Contributions (Completed). In this work, we initiate a general exploration of writeback-aware caching, seeking to bridge the gap between real caching systems and the theoretical understanding of caches. We make the following contributions:

1. We define and study the *Writeback-Aware Caching Problem*, which generalizes traditional caching problems by adding writeback costs: Given a sequence of reads and writes to data items and a specified cache size, the goal is to minimize the sum of the miss and writeback costs when servicing the sequence in order. For our algorithms, we allow data items to have variable sizes, variable miss costs, and variable writeback costs. For our hardness results, we assume data items have unit size, unit miss cost, and any fixed positive writeback cost.

2. Our main result is an online algorithm, called Writeback-Aware Landlord, and an analysis showing that it achieves the following (optimal) bound:

   Theorem 11. *For the Writeback-Aware Caching Problem, Writeback-Aware Landlord with cache size k has a competitive ratio of $k/(k-h+1)$ to the optimal (offline) algorithm with cache size h.*

   Our algorithm and analysis is a careful generalization of the well-studied Landlord algorithm [115] to properly account for the distinction between clean and dirty items. Comparing to Blelloch et al.'s [23] partitioning-based approach, our new algorithm uses a completely different approach/analysis (no cache partitioning), handles general sizes and costs, and improves the bound from 3-competitive with $3\times$ more cache to 2-competitive with $2\times$ more cache.

3. Although we prove a competitive ratio between our algorithm and the offline optimal, computing that optimal is hard. We extend Farach-Colton and Liberatore's NP-completeness proof to show NP-completeness regardless of the items' writeback cost(s) and miss cost(s). We further show the Writeback-Aware Caching Problem is Max-SNP hard, using a reduction from 3D-matching.

4. Because finding an exact solution is difficult, we turn to approximations. We show that Furthest-in-the-Future, the optimal deterministic policy when ignoring writebacks, is only a $(\omega + 1)$-approximation to optimal in our setting, and this is tight. We also provide an algorithm that is a 2-approximation of the cost difference between a cache of size zero and the optimal cache.

5. Furthermore, we provide practical algorithms for bounding the offline optimal cost from above and below. Although there are no formal guarantees of their accuracy, we show they work reasonably well for large real-world traces that would otherwise be difficult to analyze.

6. Finally, we perform a detailed experimental study using real-world storage traces. Our main finding is that Writeback-Aware Landlord outperforms state-of-the-art online replacement policies when writebacks are expensive, reducing the total cost by 14% on average across these traces. This illustrates the practical gains of explicitly accounting for writebacks.

Related Work. Theoretical work on the caching problem traditionally begins with the offline version of the problem. Different variants of the problem have been introduced and widely studied, including optimal solutions, complexity analyses, and approximation schemes [16, 88, 35, 4, 11, 36, 27].

Initial work comparing the offline and online versions of caching problems was done by Sleator and Tarjan [102]. They provided a lower bound for the competitive ratio and showed that several deterministic algorithms had matching upper bounds in that model. Fiat et al. [48] provided a similar bound for randomized policies and showed ways of approximating online policies using other online policies. Young [114] found that the 'greedy-dual' algorithm for the variable cost model matched Sleator and Tarjan's bound. He later generalized this algorithm to the variable sizes and obtained a matching upper bound [115] using the Landlord algorithm. Even et al. [46] considered a model where the cost and size of an item can change when it is accessed. Although this has some similarities to the model we introduce, neither the model nor their online algorithm can accurately model writebacks.[1]

The effects of writebacks have been well studied at the storage layer. Some of this work [52, 101] studies how to schedule writebacks to disk in order to minimize cost. Other work studies using write caches in front of storage to achieve sequential rather than random performance [21, 103]. These works provide many useful ideas that could be used to extend this work, but ignore the issues that arise with cache workloads containing mixed reads and writes.

With the emergence of highly asymmetric memory technologies, the systems community has begun to investigate the effects of writebacks on cache performance. Zhou et al. [116], motivated by phase-change memory technology, explicitly considered writebacks and proposed a partitioning scheme to reduce the effect of writes to main memory. Wang et al. [112] and Qin and Jin [97] provided similar techniques for reducing writebacks to memory by keeping track of frequently written items. These replacement policies lack worst-case bounds, and in fact it is not hard to construct request traces that yield arbitrarily bad performance.

On the theory side, we are aware of only two prior works. Back in 2000, Farach-Colton and Liberatore [47] studied a local register allocation problem that is a special case of writeback-aware caching with unit size data items, unit miss cost and unit writeback cost. They showed the offline decision problem is NP-complete using a reduction from set cover. Second, Blelloch et al. [23] provided a writeback-aware online algorithm that is 3-competitive to offline optimal when given $3\times$ the cache size, for the setting with unit size, fixed miss cost and fixed writeback costs. Their algorithm partitions the cache into a dirty half and a clean half, and applies Sleator and Tarjan's analysis [102] to each half.

3.1 Problem Formulation

Traditional Caching. The widely studied *caching problem* focuses on a single level of the memory hierarchy (cache), with capacity k, that must serve a *trace*, which is a sequence of requests for data. A request is considered to have been served when the cache contains or loads the data *item* associated with that request. Associated with each item e is a size $S(e)$ and a load cost $L(e)$. In order to load e, the cache first evicts items from the cache as needed in order to have $S(e)$ available space, and then pays $L(e)$ to load the item. Solutions to the caching problem, known as *replacement policies*, are strategies for selecting items to evict in order to minimize the total cost of the loads. *Offline* policies are given the entire trace in advance, whereas *online* policies observe the next request in the trace only after serving the previous request.

Variants. For the *generalized* caching problem (*generalized model*), the cost and size of an item may be arbitrary positive functions. Simpler versions include, for all items e: (i) the *basic model* in which items have unit size and

[1] Personal communication with Guy Even at SPAA'18.

cost: $S(e) = L(e) = 1$, (ii) the *bit model* in which cost equals size: $S(e) = L(e)$, (iii) the *cost model* in which items have unit size: $S(e) = 1$, and (iv) the *fault model* in which items have unit cost: $L(e) = 1$ [4].

Writeback-Aware Caching. We modify the caching problem to account for writebacks by identifying each request in the trace as either a read or a write. An item in the cache is *dirty* if either (i) it was loaded as a result of a write request or (ii) there has been a write request for the item since it was loaded. All other items in the cache are *clean*. Because clean items have no changes that need to be propagated to memory, evicting them has no cost. However, dirty items need to be written back to memory upon eviction. The *Writeback-Aware Caching Problem* (WA Caching Problem for short) adds a writeback cost $V(e)$ for evicting an item e that is dirty, and modifies the goal to be minimizing the sum of the miss and writeback costs.

Definition 1. In the (generalized) *Writeback-Aware Caching Problem*, we are given *(i)* a cache size k, *(ii)* an (online or offline) trace σ of requests, where each request is an item e and a flag indicating whether it is a read or write, and *(iii)* each item e has an associated size $S(e) > 0$, miss cost $L(e) > 0$ and writeback cost $V(e) > 0$. Starting and ending with an empty cache, the goal is to minimize the sum of the miss and writeback costs while serving all the requests in σ.

Since none of the original parameters of the caching problem are changed, any variant of the original problem can be made writeback-aware. In fact, the original problem is equivalent to setting the writeback costs to zero, i.e. $V(e) = 0 \; \forall \; e$. Unless stated otherwise, when we refer to the WA Caching Problem, we mean the generalized variant defined above.

3.2 A Competitive Writeback-Aware Policy

We present a deterministic online algorithm called Writeback-Aware Landlord, and show that it achieves the optimal competitive ratio for deterministic algorithms.

Our algorithm is based on the classic Landlord algorithm [115]. In Landlord, there is a credit assigned to each item that is used to determine how long the item will remain in the cache. When an item e is accessed, its credit is set to its load cost $L(e)$. Whenever items must be evicted to make space in the cache, Landlord decreases the credit of each item in proportion to the item's size until an item reaches zero credit. This item (or items) may then be evicted.

To adapt Landlord to the writeback-aware setting, we must account for writeback costs. In particular, we must determine how to balance loads and writebacks in a way that leads to an optimal competitive ratio. Our algorithm, called *Writeback-Aware Landlord* and shown in Figure 3.1, maintains two separate credits that are increased independently. In particular, accessing an item e sets (increases) its load credit to $L(e)$ and writing e sets its writeback credit to $V(e)$.

We show that Writeback-Aware Landlord has a competitive ratio matching the lower bound proven by Sleator and Tarjan [102] for the traditional caching problem. This means that Writeback-Aware Landlord is an optimal online writeback-aware policy, and that adding consideration of writebacks does not reduce the competitiveness of online policies compared to offline policies. A proof can be found in our paper [13].

Theorem 12. *Writeback-Aware Landlord with size k has a competitive ratio of $k/(k-h+1)$ to the optimal (offline) algorithm with size $h \leq k$.*

3.3 Offline Complexity Results

In 2000, Farach-Colton and Liberatore (FL) showed that the offline writeback-aware paging decision problem is NP-complete using a reduction from set cover [47]. We extend their results to cover the more general Offline WA Caching Problem and further show that problem is MaxSNP-Hard. We give only proof sketches here, full proofs can be found in our paper [13].

The high-level idea of the FL reduction is that writing back an item corresponding to a subset means choosing that subset for the cover. We the trace generation process in the FL reduction to fit our model and add support for general writeback costs, which the FL reduction did not handle. The result is a reduction from the set cover problem to the Offline WA Caching Problem. Since set cover is NP-Complete, this shows that the Offline WA Caching Problem is also.

We prove that the Offline WA Caching Problem is max SNP-hard using a reduction from bounded three-dimensional (3D) matching [74]. Our reduction uses items to represent the edges in the matching problem, with some additional items used to force evictions and occupy space. For each vertex in the matching problem, we design

```
def WritebackAwareLandlord(item e, bool write):
    if e is not in cache:
        # make space for the item
        while freeSpace < e.size:
            # find victim
            minRank, victim = infinity, none
            for f in cache:
                credit = f.wbCredit + f.loadCredit
                if credit / f.size < minRank:
                    minRank = credit / f.size
                    victim = f
            evict(victim)
            # decrease other items' credit
            for f in cache:
                delta = f.size * minRank
                # decrease wb credit first
                if delta > f.wbCredit:
                    f.loadCredit -= (delta - f.wbCredit)
                    f.wbCredit = 0
                else:
                    f.wbCredit -= delta
        # add the item to the cache
        insert(e)
    # update requested item's credit
    e.loadCredit = e.loadCost
    if write:
        e.wbCredit = e.wbCost
```

Fig. 3.1: Writeback-Aware Landlord assigns each item two *credit* values: one for loads and one for writebacks. On access, an item's credits are updated to the cost of the request (i.e., writeback cost for writes). When needed, the item with the least credit per size is evicted, and all other items' credits are reduced in proportion.

a subsequence of the trace with requests such that the optimal caching decisions will be forced to evict exactly one item corresponding to an edge that is incident upon the vertex associated with the subtrace. We are able to devise the trace such that the number of loads required for service is independent of which edge item is evicted during each vertex subtrace. Furthermore, each edge item is written to exactly twice, at the beginning and end of the trace. Thus each distinct edge item that is evicted will result in one writeback regardless of the number of evictions it suffers. Since our generated trace and resulting costs are all polynomial functions of the size of the matching instance, the number of writebacks suffered by an optimal cache directly maps to the size of the maximum matching. This suffices to prove the reduction.

Theorem 13. *The Offline WA Caching Problem is MaxSNP-Hard.*

3.4 Approximations with Theoretical Guarantees

Since we have shown that solving the Offline WA Caching Problem in polynomial time is not possible, we turn to approximation schemes. We begin with an analysis of the performance of the optimal policy for the basic writeback-oblivious caching problem in the presence of writeback costs. This policy is guaranteed to have an approximation ratio no greater than the ratio of load plus writeback cost to load cost. It is unfortunate, if unsurprising, that this ratio scales linearly with write costs.

The Furthest-in-the-Future (FitF) policy [16, 88], which evicts the item accessed furthest in the future, optimally solves the basic version of the traditional offline caching problem. However, it cannot distinguish between reads and writes. This means that there are traces where FitF will require the minimum number of loads, but each load will also cost a writeback, whereas the optimal policy can achieve the same number of loads while avoiding the writebacks. A full proof can be found in our paper [13].

Theorem 14. *FitF is an $\omega + 1$ approximation to the basic Offline WA Caching with maximum writeback cost ω. This bound is tight.*

One reason that FitF is not optimal is that it is a so-called *stack algorithm* [88]. Stack algorithms are replacement policies where the content of a larger cache is always a superset of the content of a smaller cache serving the same trace. We show in our paper [13] that stack algorithms, despite being intuitive and useful, cannot optimally solve caching problems with multiple costs, such as the WA Caching Problem.

A 2-Approximation for Savings. We give an algorithm that provides a 2-approximation of the *savings* of the optimal solution. We define the savings of a solution as the difference between the cost of the solution and the cost of loading and then immediately evicting each item accessed by the trace.

The scheme considers loads and writebacks separately. Although running any writeback-oblivious optimal algorithm on the trace is an $\omega + 1$ approximation of the cost (see above), it will provide an upper bound for the savings that can be obtained due to loads. A similar bound for the savings due to writebacks can be found by running the same algorithm on a modification of the original trace that treats reads as having load cost zero and writes as having load cost equal to their writeback cost. As the eviction decisions of both of these algorithms are valid solutions to the original problem, we choose the one with greater savings as the approximate solution. Because the optimal savings must lie between the larger of the savings and the sum, we can be off by at most a factor of two.

This technique will likely perform well when the savings available in the trace are dominated by either loads or writebacks, but will perform poorly when both metrics contribute evenly to the total savings.

3.5 Efficient Approximations for Practical Use

Lower Bound. We compute a practical lower bound for the cost of the optimal solution by considering the relaxed view of time introduced in Berger et al. [19]. In this view, the solution has capacity equal to the size of the cache multiplied by the length of the trace. Intervals between consecutive accesses to an item take up space equal to the product of the item size and interval length, and have cost equal to the savings obtained by holding the item in cache for the entire interval. By packing the cache with intervals of highest density, the ratio of interval cost to space, a solution is generated that reflects a cache with the same average size, but that can change size over the course of the trace.

To make this scheme writeback-aware, we add into consideration intervals between consecutive writes to the same item. These intervals are assigned cost equal to the sum of the costs of the load intervals to the item during its time period and the item's writeback cost. The addition of the writeback intervals also affects the packing scheme. While the writeback-oblivious version could simply choose intervals while it had space, the aware version must update the dependent intervals of each interval it selects. Chosen writeback intervals invalidate load intervals to the same item that occur during their time period. Chosen load intervals cause the writeback interval (if any) that share an item and time period with them to decrease in cost and capacity by the values of the load interval. Despite these complications, the result is a lower bound for the optimal offline solution that is accurate and efficient for many real-world traces. Following the naming convention of Berger et al., we call this algorithm writeback-aware practical flow-based offline optimal - lower (WAPFOO-L).

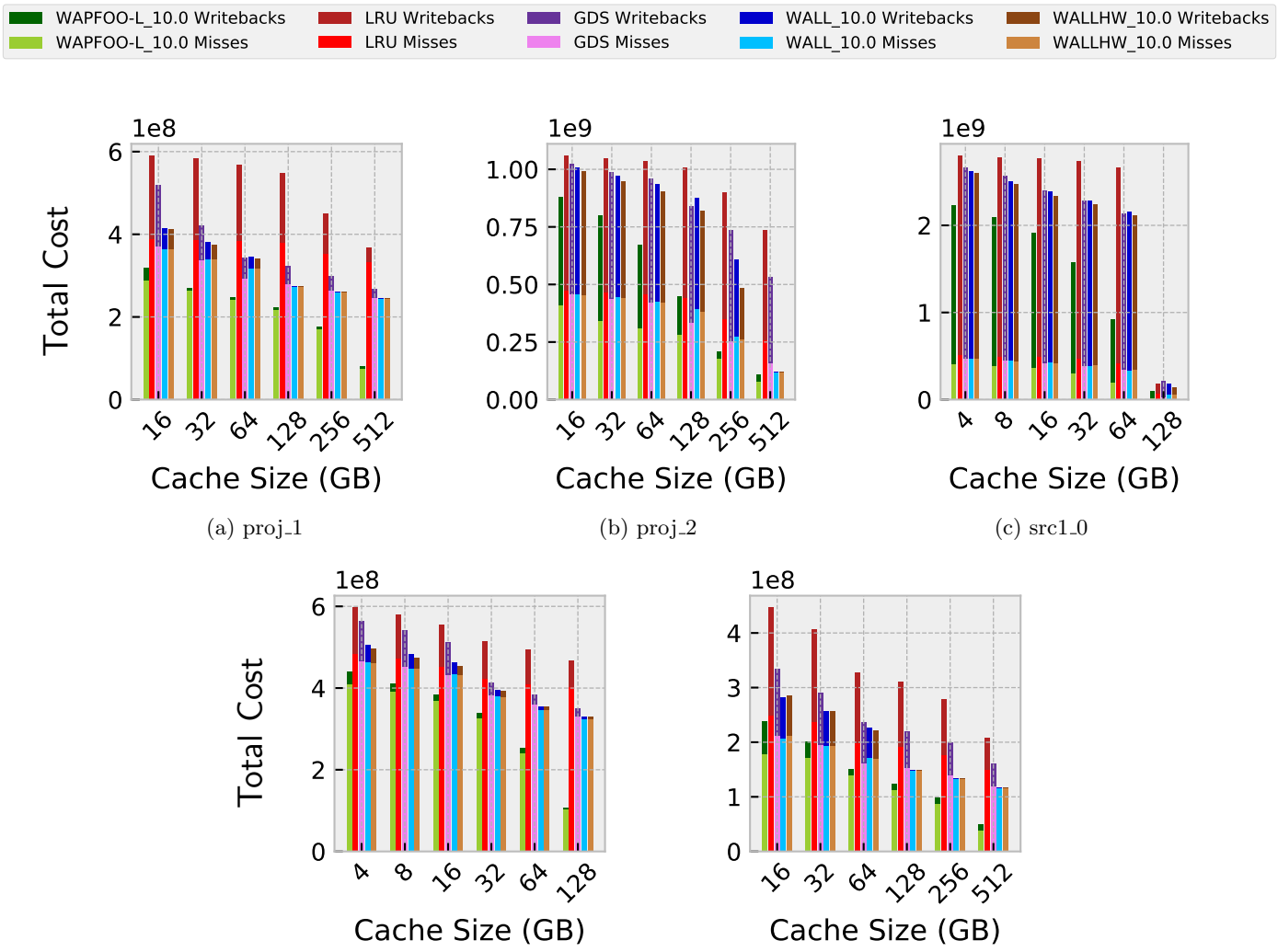
Upper Bound. We similarly adapt the ideas of Berger et al. [19] to create a practical upper bound. Their bound relies on converting the instance of the caching problem to an instance of the minimum-cost flow (MCF) problem. In the writeback-oblivious setting, this transformation completely captures the caching problem instance. However, computing the MCF for instances generated from large traces is prohibitively expensive. To make this more practical, Berger et al. consider subsets of edges at a time, breaking the graph into bite-size chunks and reducing the processing complexity.

By applying the same principles used to make the lower bound writeback-aware, we can achieve the same result for the upper bound. The difference here is that the changes are being made to edges rather than intervals, involving increased data management and requiring careful ordering or edge processing. Although these changes are expensive, we believe that the resulting algorithm remains reasonably practical.

3.6 Experimental Evaluation

To demonstrate that the theory behind Writeback-Aware Landlord holds up well in practice, we evaluate it against several state-of-the-art replacement policies on real-world storage traces [91]. This study shows that Writeback-Aware Landlord is effective in the presence of significant read-write asymmetry, reducing total cost to cache the trace by 41% over LRU and by 24% over GDS [29]. We further study how Writeback-Aware Landlord's performance varies for different writeback costs, performance metrics, and additional heuristics, and analyze from where its benefits come.

Methodology. Our simulations make use of block traces from Microsoft Research (MSR) [91], which represent access patterns experienced by MSR servers, and represent many commonly seen behaviors. They are distributed in



(a) proj_1 (b) proj_2 (c) src1_0

(d) src1_1 (e) usr_1

Fig. 3.2: Total cost (misses + weighted writebacks) for different replacement policies on the five storage traces at cache sizes 4–512 GB.

a format that specifies the time, type, offset, and size of the request. We use the size as specified and treat the offset as a request ID. We evaluate 512 M requests for each trace, replaying the trace if necessary.

We compare Writeback-Aware Landlord against LRU, GDS, and WAPFOO-L. LRU is the simplest policy commonly used in practice, and works well on traces with high temporal locality. GDS is an efficient implementation of (non-writeback-aware) Landlord that considers item cost and size when making decisions. Both LRU and GDS have theoretical worst-case bounds on their performance similar to Writeback-Aware Landlord in the basic and generalized model, respectively (Section 3.1 describes these models). WAPFOO-L is the lower bound described in Section 3.5. Comparing against these policies allows us to isolate the importance of accounting for writebacks as well as see how much potential for improvement exists.

We compare policies on their total cost over the trace, as defined in Section 3.1. Because the traces do not specify cost, we consider each item to have unit load cost and writeback cost 10. We choose this value to be between the read-write asymmetries of emerging technologies like Intel Optane [40] and flash memory [54].

For the simulations, we implement a version of WALL that requires logarithmic work per eviction, rather than the linear work required by the algorithm in Figure 3.1. Our implementation, based on Greedy Dual Size (GDS) [29], uses a global "inflation value" L rather than reducing credit and a min-heap for victim selection. We also test a version of WALL, called WALLHW, that reduces load credit before writeback credit. Our proof of optimality also holds for this policy.

Results. Figure 3.2 shows the total cost for the chosen caching algorithms across five different MSR traces for cache

sizes from 4 to 512 GB. Each cost bar is split between cost due to misses, and costs due to writebacks.

The arithmetic mean across traces and cache sizes of the miss cost of WALL is only 3.2% greater than that of GDS. However, WALL reduces writeback cost relative to GDS by 47%, significantly saving on writebacks without significantly harming hit rate. The result is that WALL's total cost is, on average, 88% of GDS, 72% of LRU, and 156% of WAPFOO-L. WALLHW performs even better, increasing miss rate by 2.6% for a 51% writeback cost reduction. This results in average total cost that is 86% of GDS, 70% of LRU, and 151% of WAPFOO-L.

Our paper [13] also contain broader experiments. We show that applying the frequency heuristic (a commonly used caching heuristic) to Writeback-Aware Landlord results in fewer misses and writebacks, similar to writeback-oblivious policies, although with somewhat reduced effect. In addition, we provide a sensitivity analysis of the effect on writeback cost on Writeback-Aware Landlord's performance. Our results show that Writeback-Aware Landlord performs well across a range of writeback costs, with only moderate degradation as writebacks become less significant.

4. BLOCK-AWARE CACHING

As the systems and data of the world continue to grow larger and larger, more dense storage media is required to manage that data. These larger storage devices, like NVMe and Flash, commonly operate using a block-based interface. This means those data are read or written in the granularity of large (commonly 4 KB) blocks rather than cache lines. Caches in front of these systems typically ignore the interface below them and any data that exists beyond what is specifically being requested.

We believe that there is an opportunity to improve cache performance in such systems by making informed decisions about the entire data block rather than simply the requested data. By allowing additional lines to be brought into the cache, we can potentially avoid future requests for those same lines from the storage.

The Block-Aware Caching Problem can be broken up into two major questions: how to assign items to blocks, and how to operate the cache effectively given such an assignment. We begin by considering the latter question given a static assignment. Using the insights we gain from this question, we begin to consider how to do assignment. Finally, we intend to apply our results to study the combined problem.

Our Contributions (In Progress). In this work, we initiate a general exploration of block-aware caching, seeking to make use of data that is already in-flight to improve cache performance. We propose to make the following contributions:

1. We define the *Static Block-Aware Caching Problem* (SBA Caching Problem) and the *Dynamic Block-Aware Caching Problem* (DBA Caching Problem). In the SBA Caching Problem, the goal is to minimize the number of block reads from storage given a sequence of accesses to data items, a specified cache size, and an assignment of data items to blocks that is unchangeable. The DBA Caching Problem modifies the SBA Caching Problem by allowing the cache to control the assignment of data items to blocks. (Completed)

2. Our analysis of the SBA Caching Problem begins with the complexity of the offline version, and we prove that the problem is NP-complete using a reduction from the variable size caching problem. (Completed)

3. Having shown that the Static Block-Aware Caching Problem is difficult to solve, we intend to find an algorithm that can be used to find approximate solutions in polynomial time. (Future)

4. We then move to competitive analysis for the Static Block-Aware Caching Problem. Here we show that the gap between optimal and online policies has increased through a new competitiveness lower bound. (In Progress)

5. To match the lower bound of competitiveness, we are in the process of developing a block-aware replacement policy that has strong competitiveness guarantees. (In Progress)

6. To address the question of how to assign items to blocks, we intend to provide an analysis of the impact of different static assignments on the cost of a trace, as well as an assignment scheme that can perform caching assignments with bounded competitive ratio. For this analysis, we intend to rely on our approximations of the offline optimal policy as well as our online competitive policy. (Future)

7. After investigating how to choose static assignments, we will turn to dyanmic assignments. By applying the principles we learn from the static setting, we intend to provide a replacement policy for the DBA Caching Problem that will dynamically rewrite the blocking scheme while servicing the trace. (Future)

8. To ensure that the policies we develop are useful in practice rather than being a theoretical construct, we intend to run a performance comparison between our policies and several currently used policies on real-world traces. For each of our approaches, we will show how it impacts performance both as an individual change, and in combination with the other approaches we discuss. (Future)

Related Work. As discussed in previous chapters, cache replacement has been the subject of a vast body of research work. However, the subset that pertains to interacting with block interfaces is much smaller.

Some practical caching work has been done that focuses on the storage layer below. One such area focuses on scheduling writebacks to minimize cost based on disk performance [52, 101, 103]. Similar work has been done focusing on using NVM in the SSD form factor as a cache [113].

Bandana [45] targets a Facebook system where similar vector embeddings are often requested together. Their system tries to locate items with good temporal locality together and provides the option for the cache to load either the particular item requested or the entire 4KB block. They assign items to blocks using two different methods: a k-means clustering [8] based on the position of vectors in Euclidean space, and Social Hash Partitioning [73], a hypergraph partitioning scheme that they base on representative trace data. Bandana is a promising starting point for assigning items to blocks, but considers only limited caching options and does not consider dynamic assignment.

BORG [21] applies the ideas of item to block remapping in order to improve storage system performance. They make use of a disk partition as a cache, and manage it at the item level rather than the block level. Their result makes use of an interesting greedy heuristic to perform remapping. However, their assignment is based on overall frequency rather than inter-object correlation, and they use I/O traces with logical block address ranges rather than objects.

4.1 Problem Formulation

We begin with the traditional caching model described in Section 3.1, and modify it to account for the storage level below using a block-based interface. This modification manifests itself in loading data: each item is located in a particular block on the storage level below, and when the cache loads that item it can also choose to load any subset (including the empty subset) of other items located in that block. We refer to the resulting family of problems as the Block-Aware Caching Problem.

In order to examine the dimensions of this problem individually rather than being forced to consider them in aggregate, we define the Static Block-Aware Caching Problem (SBA Caching Problem for short). This problem forces the assignment of data items to blocks to remain unchanged throughout the trace.

Definition 2. In the Static Block-Aware Caching Problem, we are given *(i)* a cache size k, *(ii)* an (online or offline) trace σ of requests, where each request is for an item e, and *(iii)* a mapping of items to blocks, such that loading any subset of a block into cache has the same cost. Starting with an empty cache, the goal is to minimize the cost of loads while serving all the requests in σ.

After defining and analyzing the individual dimensions of the problem, we plan to investigate a more general version of the problem to see how our approaches interact. We define the Dynamic Block-Aware Caching Problem (DBA Caching Problem for short), which allows items to be dynamically remapped between blocks.

Definition 3. In the Dynamic Block-Aware Caching Problem, we are given *(i)* a cache size k, *(ii)* an (online or offline) trace σ of requests, where each request is for an item e, and *(iii)* an initial mapping of items to blocks. , Again, loading any subset of a block into cache has the same cost. However, the cache gains the capability to move items from one block to another, provided that no block ever exceeds the maximum block size B. Each remapping operation has cost R. Starting with an empty cache, the goal is to minimize the sum of the cost due to loads and the cost due to remappings while serving all the requests in σ.

For the initial exploration of this problem space, we limit ourselves to the basic variant, where each item has unit size and each block has unit cost. We believe that this approach will limit the complexity of the problem, while maintaining its value as a practical model. Since the data is being transferred in units of blocks rather than items, modelling the cost of items is not applicable. On the other hand, while unit size items do a good job of modelling CPU caches that store cache lines, the problem can easily be extended to model a system with variable size objects.

4.2 Static Block-Aware Caching Problem Complexity

Our analysis of the SBA Caching Problem begins with a complexity analysis of the offline problem, which we prove is NP-complete. Our proof reduces the offline *fault model* caching problem (See Section 3.1 for a description), which is NP-complete [36], to the SBA Caching Problem.

We assume that each item in the fault model is integral size. We generate one block for each such item, and use a number of items from that block equal to the size. The SBA Caching trace is generated by replacing each fault model access with accesses to the corresponding block-aware items. The cache size remains unchanged.

The resulting SBA Caching instance will have the same optimal cache cost as the fault model instance. Since the items in a block are all accessed consecutively, they can all be loaded into cache upon the first miss. This means that evicting any such item is the same as evicting all of them from a cost perspective.

Theorem 15. *The Offline Static Block-Aware Caching Problem is NP-Complete.*

4.3 Static Block-Aware Caching Problem Competitiveness

The SBA Caching Problem offers increased opportunity for good caching decisions to reduce performance cost. However, the gap between optimal caching and what can be done with an online policy has increased. We show this by providing a general lower bound for the competitive ratio of any deterministic online policy with size k compared to the optimal policy (OPT) with size $h \leq k$ in the SBA Caching Problem.

Our lower bound assumes that both caches are full at the start of the trace. We then request $\lceil (k-h+1)/B \rceil$ blocks, where B is the blocks size. For each block, we request items that have not been brought into the online cache until no such item exists. We use a to represent the number of requests that occur per block. Each such request will cause a miss for the online cache, while OPT can load each requested item upon the first request. Between the h items that were originally in OPT's cache and the $k-h+1$ that have just been accessed, the online cache must have evicted at least one. The next $h-a$ requests are to whichever item from this set is not in the online cache. These will all be misses for the online cache, and hits for OPT.

Theorem 16. *No deterministic online cache replacement policy can have a competitive ratio lower than:*

$$\frac{a(k-h+1)+B(h-a)}{(k-h+1)}$$

Here k is the size of the online cache, h is the size of the optimal cache, B is the block size, and a is how many distinct requests to a block are required for the online policy to load each item in the block.

4.4 A Competitive Policy for Static Block-Aware Caching Problem

As with other caching problems, we would like to have a policy that provides a matching upper bound to the lower bound on competitiveness. We are currently in the process of developing such a policy.

Our policy, which we call Prefetch-Access Partitioning, splits the cache into two partitions. Each partition targets one methods of obtaining hits in the SBA Caching Problem. The access partition targets temporal locality by acting like a traditional cache; it ignores the possibility of admitting items beyond what is specifically requested in order to reserve space for things that have. The prefetch partition targets spatial locality; it treats every request as targeting the entire block. For both partitions, we run the Least-Recently Used (LRU) replacement policy.

We choose this strategy in order to provide worst-case guarantees on both types of cache hits that may occur. If the access partition is too small, worst-case traces from traditional caching will experience poor performance. However, as we see in the lower bound from Section 4.3, policies that are slow to prefetch the rest of the block can suffer from traces that access many different items from a small set of blocks. Since the prefetch partition loads the entire block, meaning that $a = 1$, it can reduce the degradation suffered on such traces.

We are currently in the process of determining the relative sizes of each partition and ensuring that worst-case mixtures of spatial and temporal locality do not hinder the bounds. Furthermore, the fact that the partitions duplicate data (requested items will be loaded into both partitions) will ensure that the bounds will not match exactly. However, we believe that this policy has promising chances of asymptotically matching the lower bound.

4.5 Assigning Items to Blocks

A question that goes hand-in-hand with how to cache blocked items is how the assignment of items to blocks affects cacheability. We believe that assignment of items to blocks is likely to be quite significant, especially for traces where the access pattern remains primarily static.

Our goal is to be able to determine for a given trace and cache size, how much the cost of caching that trace can be reduced with an intelligent assignment of items to blocks. For the baseline we intend to use the block-oblivious instance, which simulates an assignment of each item to is own block, and provides no spatial locality benefits. Since we have already proven that this is hard to calculate (Section 4.2), we will make use of approximation schemes and the competitive online policy we are developing (Section 4.4).

Our intended approach is to identify pairs of items that have accesses that are close together temporally as candidates to shares blocks. We can represent relationships between items as a graph where vertices represent items and weighted edges represent the number of candidate accesses shared. The graph can be partitioned into clusters with maximum intra-cluster edge weights such that each partition has at most B vertices. All items whose vertices share a cluster are assigned to the same block.

We intend to identify candidate access pairs based on the trace's cacheability. Loading the second item at the earlier access time only makes sense if it is more valuable to the cache than alternative items being reused. Accordingly, we intend to compare the distance between the two accesses to the distance between accesses that are being cached in the trace.

We also intend to explore an additional dimension of this space: duplicating items in multiple blocks. This requires extra storage space, forces the cache to specify which block is being read on access, and introduces issues for data that is not read-only. However, it can provide cache performance benefits that may justify the tradeoff.

We do not expect our results in this area to be exact solutions, as the caching schemes used for evaluation are not exact, but we hope to gain useful insights for improving cache performance through item to block assignment.

4.6 The Dynamic Block-Aware Caching Problem

Once we have determined how to find the best static assignments of items to blocks, we would like to apply that knowledge to dynamic assignments. Static assignments work well if access patterns remain steady throughout a trace, but do not adapt well to variation. When the assignment is dynamic, objects can be moved between blocks to best serve the current portion of the trace.

The drawback of dynamic item to block assignment moving the items to their new blocks comes at a cost. We consider two different models for performing reassignment.

The first model assumes a memory manager separate from the cache. In this model, the manage performs behind-the-scenes reassignment of items. These changes appear atomic and instantaneous to the cache: at any given request their appears to be a consistent assignment, but the assignment can change from request to request. The cost of these reassignments occurs on a per-operation basis: for each request, the cost due to reassignments is proportional to the number of blocks that have changed from the assignment of the previous request.

In the second model the cache manages data movement by sending messages to the storage layer below. Each message contains the items to be moved and their new location. Since the cache needs to send the item explicitly, items must be loaded into cache to be moved. This means that moving items while they are not in the current access pattern costs a load(s) in addition to the cost of the reassignment message.

We can use the techniques that we develop for static assignment across varying windows of the trace in order to approximate dynamic assignments. By picking reasonable windows, we can find a practical balance between assignment computation cost and adaptivity. We believe that this problem will be highly applicable to real caching systems, where traces often change behaviors, and the system will be expected to adapt in real-time.

4.7 Experimental Evaluation

In addition to our theoretical work on the BA Caching Problem, we intend to show how our work performs in practice. For each policy we create, we will test its performance both individually and in combination with our other ideas.

We intend to use real industry traces, like those available from the SNIA Iotta repository [64], for our simulations. These traces are similar to the MSR traces used in Section 3.6, but the repository continues to bring in new, modern traces. For our static policies, we intend to consider three different methods of assigning objects to blocks: using the indexing scheme of the specified trace, randomly, and using the assignment scheme that we devise.

Our comparisons will be made against the policies commonly used today in the caching systems we model. The results will show how the number of block accesses, and thus performance, changes with each policy and assignment combination.

5. APPLYING INFORMATION THEORY TO CACHING

As the memory hierarchy becomes more complicated in order to support new memory technologies and tighter performance demands, the demands on caching become greater and more challenging. This makes further improvements to cache performance more complex and piecemeal.

To ensure that such improvements are as efficient and well-targeted as possible, understanding the cache system is crucial. This is made difficult by the breadth of existing caching problems and the theoretical targeting of worst-case analysis. We attempt to provide a framework for analyzing caching problems on a case-by-case basis where the specific problem and trace style can be the focus.

Our approach is to apply techniques developed in information theory to analyze sample traces for cacheability and predictability. We believe that this approach will provide several benefits. Analyzing traces specific to an individual problem can provide more relevant results than worst-case analysis. Information theory metrics can be computed for larger input sizes than optimal caching solutions. These metrics can be used to build a predictor of future access patterns. This predictor will provide a tradeoff between resource consumption and accuracy. The accuracy of the predictor shows roughly the maximum performance that can be reasonably achieved.

Our Contributions (Future). In this work, we study the application of information theory to caching in order to better understand how different traces are affected by different replacement policies and how to predict future requests. We propose to make the following contributions:

1. We intend to investigate several different methods of computing the predictability of a variety of real traces. Using publicly available datasets, we intend to characterize traces using entropy, compressability, and stateful probability models.

2. After analyzing the compressability of traces, we intend to compare our results to the cacheability of the traces. By analyzing the correlation between various information theoretical metrics and the various cache policy performances, we hope to be able to identify metrics that are useful in determining how to approach caching a trace.

3. For our identified metrics, we intend to investigate the tradeoff between the amount of data used to model the trace, and the accuracy of future item prediction.

4. Based on the data we gather about modeling and predicting traces, we intend to introduce a cache replacement policy utilizing our finding. The policy will be parameterized to support varying levels of complexity based on the expected improvement in caching performance. We also intend to test our policy on real world traces to evaluate its performance.

Related Work. As discussed in previous chapters, cache replacement has been the subject of a vast body of research work. However, there is still a gap between theoretical worst-case and practical cache system performance.

On the practical side, many different works have been performed that attempt to handle the challenges that being an online policy poses. Many policies [14, 12, 72] use item evaluation metrics that serve as a proxy for future knowledge. Others directly attempt to predict future requests, and make decisions based on these predictions [69, 70].

From the theoretical direction, attempts have been made to improve upon worst-case analysis, and more accurately capture the characteristics of real-world traces. Some of the more prominent approaches include average-case analyses [3, 79] and smoothed analyses [104, 105]. Some theoretical works have also modeled systems that provide limited knowledge of future requests. Kumar et al. [80] model future requests using access graphs, and show results that depend on the number of such graphs running concurrently. Lykouris and Vassilvitskii [85] introduced a model where an oracle provides estimates for item reuse and provide competitive bounds in that model. These bounds are made nearly tight in a followup work in the same model by Rohatgi [100]. For both papers, the bound proved are a function of the l_1 error of the reuse estimates. Our model is very similar to theirs, but we have white-box access to our oracle, which we believe will allow us to obtain additional performance benefits.

There have also been some prior works seeking to apply information theory to caching. Pandurangan and Upfal [93, 94] study how the entropy of a caching system can be used to lower bound cache performance. However,

their model assumes a stationary ergodic process, which is a significant assumption for caching systems. Maddah-Ali and Niesen [86] study a multi-level network cache, where they apply information theory to formulate the problem and propose a coded caching scheme to achieve global performance despite different access patterns in different network locations. Our work ignores the dimensions of networking and multi-level architectures, focusing on applying information theory to trace analysis. Phalke and Gopinath [96] use compression techniques to predict future cache accesses. We intend to build on their results, and cover a broader array of problems and approaches.

5.1 Trace Analysis

Our application of information theory to caching begins with an analysis of real world traces. We consider several different metrics used in information theory, and for each, we will show how accurately it characterizes traces.

For our analysis, we intend to consider a variety of real-world storage traces consisting of different times, item size ranges, cost models, and applications. We plan to compare and contrast these different types of traces to show what elements of locality persist across applications and which are domain-specific. Furthermore, we believe that this variety of applications will be useful in separating useful information theoretical metrics from those that capture characteristics of the trace less useful to caching.

The metrics we intend to use will capture a breadth of information theoretical approaches. We intend to consider methods entropy, conditional entropy, both lossless and lossy data compression, and both exact and approximate markov chains. This wide exploration should maximize the chances of finding a metric(s) that can be used to predict cacheability, and additionally provides an opportunity to enhance our understanding of locality.

5.2 Converting Information Theory to Cacheability

Although analyzing traces using information theory metrics has uses in its own right, our primary goal is to better understand and predict how to cache traces. To that end, we intend to compare the analysis performed in the previous section with the performance of a variety of cache replacement policies on the same traces.

For each trace, we will analyze policies that cover a variety of circumstances. We intend to analyze the optimal policy (or the closest available approximation for problems where this is not feasible), a few simple online policies, and a selection of more complicated online policies. For each policy, we intend to consider cache sizes ranging from a small fraction of the frequently reused items to a large portion of the entire dataset.

This analysis will result in a large amount of data, which we hope to use to determine the correlation between the information theory metrics and cache performance. It is our hope that some metrics will provide a high correlation with cache performance. These metrics will be further investigated for direct applications to item prediction and replacement policies.

5.3 Evaluating Accuracy Tradeoffs

In order for cache replacement policies to be practical, they need to make decisions in real-time with limited resources. Accordingly, we intend to investigate how the metrics we identify as useful perform under resource restriction.

Many concepts in information theory already explicitly model the tradeoff between resource consumption and accuracy. We intend to exploit this existing work for metrics where it exists, and extend it to any metrics we target where it does not already. We anticipate results that look similar to that of streaming models or sketching, where the size of the model maps to the accuracy of results that can be obtained.

Using our analysis, we can distinguish between metrics that can efficiently be used to predict future cache accesses and those that are limited to more theoretical uses. In addition, we can estimate how detailed a model is needed for these metrics to be effective, and apply that understanding directly to replacement policies.

5.4 Model-Based Policies

After carefully analyzing our information theory metrics and pruning those that fall short, we intend to apply our results directly to cache prediction and replacement.

The first step in this process is to generate a model that can predict future cache requests. In addition to solving the theoretical prefetching problem effectively, such a model can be used as an oracle that existing replacement policies can use to improve their performance.

Such an oracle can also be used to estimate the locality of items in the trace. We intend to generate replacement policies through the use of such estimations in combination with optimal policies and prior work such as that of

Rohatgi [100] that is explicitly designed for such estimations. In addition, we intend to investigate whether our white-box access to the metric can be used to inform more intelligent use of the estimations.

For each of the policies we generate, we intend to perform a thorough empirical analysis on the traces that we collect. We will show several points on the resource-accuracy tradeoff curve for each policy, and discuss how each performs in comparison with current state-of-the-art competitors.

6. PROPOSED THESIS TIMELINE

- April 2020 - May 2020:
  Finish design of SBA Caching policy and analysis of upper bound (Section 4.4).
  Finalize list of metrics for trace analysis (Section 5.1).
- May 2020 - June 2020:
  Analyze the value of different item to block assignments (Section 4.5).
  Select traces for use in BA Caching Problem evaluation (Section 4.7).
- June 2020 - August 2020:
  Perform empirical analysis of SBA Caching Problem on selected traces (Section 4.7).
  Analyze traces using selected information theory metrics (Section 5.1).
  Submit paper on Static Block-Aware Caching Problem to APoCS (Chapter 4).
- August 2020 - September 2020:
  Design and analyze a policy for assigning items to blocks (Section 4.5).
  Simulate cache performance on traces selected for difficulty analysis (Section 5.2).
- September 2020 - November 2020:
  Design and analyze policies for the DBA Caching Problem (Section 4.6).
  Analyze the correlation between information theory metrics and cache performance (Section 5.2).
- November 2020 - January 2021:
  Perform empirical analysis of item to block assignment (Section 4.5) and DBA Caching Problem (Section 4.6).
  For metrics that appear to be reliable indicators of cacheability, analyze the tradeoff between resource consumption and accuracy (Section 5.3).
- January 2021 - March 2021:
  Design and analyze policies relying on selected information theory metrics (Section 5.4).
  Submit paper on Dynamic Block-Aware Caching Problem to SPAA (Chapter 4).
- March 2021 - May 2021:
  Perform empirical evaluation of information theory based policies (Section 5.4).
  Write paper on applying information theory to caching (Chapter 5).
  Write and defend thesis.
- August 2021:
  Submit paper on applying information theory to caching to APoCS (Chapter 5).

BIBLIOGRAPHY

[1] Afek, Y., Greenberg, D. S., Merritt, M., and Taubenfeld, G. Computing with faulty shared memory. In *PODC* (1992).

[2] Aggarwal, A., and Vitter, J. S. The Input/Output complexity of sorting and related problems. *Communications of the ACM 31*, 9 (1988).

[3] Ajwani, D., and Friedrich, T. Average-case analysis of online topological ordering. In *International Symposium on Algorithms and Computation* (2007), Springer, pp. 464–475.

[4] Albers, S., Arora, S., and Khanna, S. Page replacement for general caching problems. In *SODA* (1999), vol. 99, Citeseer, pp. 31–40.

[5] Appel, A. W., and Jim, T. Continuation-passing, closure-passing style. In *POPL* (1989).

[6] Arge, L., Goodrich, M. T., Nelson, M., and Sitchinava, N. Fundamental parallel algorithms for private-cache chip multiprocessors. In *ACM symposium on Parallelism in algorithms and architectures (SPAA)* (2008).

[7] Arora, N. S., Blumofe, R. D., and Plaxton, C. G. Thread scheduling for multiprogrammed multiprocessors. *Theory of Computing Systems 34*, 2 (Apr 2001).

[8] Arthur, D., and Vassilvitskii, S. k-means++: The advantages of careful seeding. Tech. rep., Stanford, 2006.

[9] Aumann, Y., and Ben-Or, M. Asymptotically optimal PRAM emulation on faulty hypercubes. In *IEEE Symposium on Foundations of Computer Science (FOCS)* (1991).

[10] Balsamo, D., Weddell, A. S., Merrett, G. V., Al-Hashimi, B. M., Brunelli, D., and Benini, L. Hibernus: Sustaining computation during intermittent supply for energy-harvesting systems. *IEEE Embedded Systems Letters 7*, 1 (2015).

[11] Bar-Noy, A., Bar-Yehuda, R., Freund, A., Naor, J., and Schieber, B. A unified approach to approximating resource allocation and scheduling. *Journal of the ACM (JACM) 48*, 5 (2001), 1069–1090.

[12] Beckmann, N., Chen, H., and Cidon, A. {LHD}: Improving cache hit rate by maximizing hit density. In *15th {USENIX} Symposium on Networked Systems Design and Implementation ({NSDI} 18)* (2018), pp. 389–403.

[13] Beckmann, N., Gibbons, P. B., Haeupler, B., and McGuffey, C. Writeback-aware caching. In *Symposium on Algorithmic Principles of Computer Systems* (2020), Society for Industrial and Applied Mathematics, pp. 1–15.

[14] Beckmann, N., and Sanchez, D. Modeling cache performance beyond lru. In *2016 IEEE International Symposium on High Performance Computer Architecture (HPCA)* (2016), IEEE, pp. 225–236.

[15] Beckmann, N., and Sanchez, D. Maximizing cache performance under uncertainty. In *High Performance Computer Architecture (HPCA), 2017 IEEE International Symposium on* (2017), IEEE, pp. 109–120.

[16] Belady, L. A. A study of replacement algorithms for a virtual-storage computer. *IBM Systems journal 5*, 2 (1966), 78–101.

[17] Belady, L. A., and Palermo, F. P. On-line measurement of paging behavior by the multivalued min algorithm. *IBM Journal of Research and Development 18*, 1 (1974), 2–19.

[18] Ben-David, N., Blelloch, G. E., Friedman, M., and Wei, Y. Delay-free concurrency on faulty persistent memory. In *The 31st ACM Symposium on Parallelism in Algorithms and Architectures* (2019), pp. 253–264.

[19] Berger, D. S., Beckmann, N., and Harchol-Balter, M. Practical bounds on optimal caching with variable object sizes. *Proc. ACM Meas. Anal. Comput. Syst. (SIGMETRICS'18)* (2018).

[20] Berryhill, R., Golab, W., and Tripunitara, M. Robust shared objects for non-volatile main memory. In *LIPIcs-Leibniz International Proceedings in Informatics* (2016), vol. 46, Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik.

[21] Bhadkamkar, M., Guerra, J., Useche, L., Burnett, S., Liptak, J., Rangaswami, R., and Hristidis, V. Borg: Block-reorganization for self-optimizing storage systems. In *FAST* (2009), vol. 9, Citeseer, pp. 183–196.

[22] Bhandari, K., Chakrabarti, D. R., and Boehm, H.-J. Makalu: Fast recoverable allocation of non-volatile memory. In *ACM SIGPLAN Notices* (2016), vol. 51, ACM, pp. 677–694.

[23] Blelloch, G. E., Fineman, J. T., Gibbons, P. B., Gu, Y., and Shun, J. Sorting with asymmetric read and write costs. In *Proceedings of the 27th ACM Symposium on Parallelism in Algorithms and Architectures (SPAA)* (2015), ACM, pp. 1–12.

[24] Blelloch, G. E., Gibbons, P. B., Gu, Y., McGuffey, C., and Shun, J. The parallel persistent memory model. In *ACM symposium on Parallelism in algorithms and architectures (SPAA)* (2018).

[25] Blelloch, G. E., Gibbons, P. B., and Simhadri, H. V. Low depth cache-oblivious algorithms. In *ACM symposium on Parallelism in algorithms and architectures (SPAA)* (2010).

[26] Bohr, M. A 30 year retrospective on dennard's mosfet scaling paper. *IEEE Solid-State Circuits Society Newsletter 12*, 1 (2007), 11–13.

[27] Brehob, M., Wagner, S., Torng, E., and Enbody, R. Optimal replacement is np-hard for nonstandard caches. *IEEE Transactions on computers 53*, 1 (2004), 73–76.

[28] Buettner, M., Greenstein, B., and Wetherall, D. Dewdrop: an energy-aware runtime for computational RFID. In *NSDI* (2011).

[29] Cao, P., and Irani, S. Cost-aware www proxy caching algorithms. In *Usenix symposium on internet technologies and systems* (1997), vol. 12, pp. 193–206.

[30] Cappello, F., Al, G., Gropp, W., Kale, S., Kramer, B., and Snir, M. Toward exascale resilience: 2014 update. *Supercomput. Front. Innov.: Int. J. 1*, 1 (Apr. 2014).

[31] Chakrabarti, D. R., Boehm, H.-J., and Bhandari, K. Atlas: Leveraging locks for non-volatile memory consistency. In *OOPSLA* (2014).

[32] Chauhan, H., Calciu, I., Chidambaram, V., Schkufza, E., Mutlu, O., and Subrahmanyam, P. NVMove: Helping programmers move to byte-based persistence. In *INFLOW* (2016).

[33] Chen, S., and Jin, Q. Persistent b+-trees in non-volatile main memory. *Proceedings of the VLDB Endowment 8*, 7 (2015), 786–797.

[34] Chlebus, B. S., Gambin, A., and Indyk, P. PRAM computations resilient to memory faults. In *European Symposium on Algorithms (ESA)* (1994).

[35] Chrobak, M., Karloff, H. J., Payne, T. H., and Vishwanathan, S. New results on server problems. In *SIAM Journal on Discrete Mathematics* (1991), pp. 172–181.

[36] Chrobak, M., Woeginger, G. J., Makino, K., and Xu, H. Caching is hard-even in the fault model. *Algorithmica 63*, 4 (2012), 781–794.

[37] Coburn, J., Caulfield, A. M., Akel, A., Grupp, L. M., Gupta, R. K., Jhala, R., and Swanson, S. NV-Heaps: Making persistent objects fast and safe with next-generation, non-volatile memories. In *ASPLOS* (2011).

[38] Colin, A., and Lucia, B. Chain: tasks and channels for reliable intermittent programs. *OOPSLA* (2016).

[39] Colin, A., and Lucia, B. Termination checking and task decomposition for task-based intermittent programs. In *Proceedings of the 27th International Conference on Compiler Construction* (2018), ACM, pp. 116–127.

[40] Corporation, I. Optane ssd dc p4800x series, 2018. Retrieved online on 11 Jan 2019 at `https://ark.intel.com/products/97161/Intel-Optane-SSD-DC-P4800X-Series-375GB-2-5in-PCIe-x4-3D-XPoint-`.

[41] David, T., Dragojevic, A., Guerraoui, R., and Zablotchi, I. Log-free concurrent data structures. EPFL Technical Report, 2017.

[42] de Kruijf, M. A., Sankaralingam, K., and Jha, S. Static analysis and compiler design for idempotent processing. In *PLDI* (2012).

[43] Dennard, R. H., Gaensslen, F. H., Rideout, V. L., Bassous, E., and LeBlanc, A. R. Design of ion-implanted mosfet's with very small physical dimensions. *IEEE Journal of Solid-State Circuits 9*, 5 (1974), 256–268.

[44] Duong, N., Zhao, D., Kim, T., Cammarota, R., Valero, M., and Veidenbaum, A. V. Improving cache management policies using dynamic reuse distances. In *Microarchitecture (MICRO), 2012 45th Annual IEEE/ACM International Symposium on* (2012), IEEE, pp. 389–400.

[45] Eisenman, A., Naumov, M., Gardner, D., Smelyanskiy, M., Pupyrev, S., Hazelwood, K., Cidon, A., and Katti, S. Bandana: Using non-volatile memory for storing deep learning models. *arXiv preprint arXiv:1811.05922* (2018).

[46] Even, G., Medina, M., and Rawitz, D. Online generalized caching with varying weights and costs. In *Proceedings of the 30th ACM Symposium on Parallelism in Algorithms and Architectures (SPAA)* (2018), ACM, pp. 205–212.

[47] Farach-Colton, M., and Liberatore, V. On local register allocation. *Journal of Algorithms 37*, 1 (2000), 37–65.

[48] Fiat, A., Karp, R. M., Luby, M., McGeoch, L. A., Sleator, D. D., and Young, N. E. Competitive paging algorithms. *Journal of Algorithms 12*, 4 (1991), 685–699.

[49] Finocchi, I., and Italiano, G. F. Sorting and searching in the presence of memory faults (without redundancy). In *ACM Symposium on the Theory of Computing (STOC)* (2004).

[50] Friedman, M., Herlihy, M., Marathe, V. J., and Petrank, E. A persistent lock-free queue for non-volatile memory. In *ACM Symposium on Principles and Practice of Parallel Programming (PPOPP)* (2018).

[51] Frigo, M., Leiserson, C. E., Prokop, H., and Ramachandran, S. Cache-oblivious algorithms. In *IEEE Symposium on Foundations of Computer Science (FOCS)* (1999).

[52] Gill, B. S., and Modha, D. S. Wow: Wise ordering for writes-combining spatial and temporal locality in non-volatile caches. In *FAST* (2005).

[53] Grove, D., Hamouda, S. S., Herta, B., Iyengar, A., Kawachiya, K., Milthorpe, J., Saraswat, V., Shinnar, A., Takeuchi, M., and Tardieu, O. Failure recovery in resilient X10. Tech. Rep. RC25660 (WAT1707-028), IBM Research, Computer Science, 2017.

[54] Grupp, L. M., Caulfield, A. M., Coburn, J., Swanson, S., Yaakobi, E., Siegel, P. H., and Wolf, J. K. Characterizing flash memory: anomalies, observations, and applications. In *Microarchitecture, 2009. MICRO-42. 42nd Annual IEEE/ACM International Symposium on* (2009), IEEE, pp. 24–33.

[55] Guerraoui, R., and Levy, R. R. Robust emulations of shared memory in a crash-recovery model. In *Inter. Conference on Distributed Computing Systems (ICDCS)* (2004), IEEE.

[56] Herlihy, M., and Shavit, N. *The Art of Multiprocessor Programming*. Morgan Kaufmann, 2012.

[57] Hester, J., Storer, K., and Sorber, J. Timely execution on intermittently powered batteryless sensors. In *Proc. ACM Conference on Embedded Network Sensor Systems* (2017).

[58] Horowitz, M. Computing's energy problem (and what we can do about it). In *Proc. of the IEEE Intl. Solid-State Circuits Conf. (ISSCC)* (2014).

[59] Hsu, T. C.-H., Bruegner, H., Roy, I., Keeton, K., and Eugster, P. NVthreads: Practical persistence for multi-threaded applications. In *EuroSys* (2017).

[60] www.slideshare.net/IBMZRL/theseus-pss-nvmw2014, 2014.

[61] Intel. Intel NVM library. https://github.com/pmem/nvml/.

[62] Intel. Intel architecture instruction set extensions programming reference. Technical Report 3319433-029, Intel Corporation, April 2017.

[63] Intel. `www.intel.com/content/www/us/en/products/docs/processors/xeon/xeon-scalable-platform-where-to-buy.html`, 2019.

[64] IOTTA, S. Storage networking industry association's input/output traces, tools, and analysis.

[65] Izraelevitz, J., Kelly, T., and Kolli, A. Failure-atomic persistent memory updates via JUSTDO logging. In *ASPLOS* (2016).

[66] Izraelevitz, J., Mendes, H., and Scott, M. L. Brief announcement: Preserving happens-before in persistent memory. In *ACM symposium on Parallelism in algorithms and architectures (SPAA)* (2016).

[67] Izraelevitz, J., Mendes, H., and Scott, M. L. Linearizability of persistent memory objects under a full-system-crash failure model. In *DISC* (2016).

[68] Izraelevitz, J., Yang, J., Zhang, L., Kim, J., Liu, X., Memaripour, A., Soh, Y. J., Wang, Z., Xu, Y., Dulloor, S. R., et al. Basic performance measurements of the Intel Optane DC persistent memory module. *arXiv preprint arXiv:1903.05714* (2019).

[69] Jain, A., and Lin, C. Back to the future: leveraging belady's algorithm for improved cache replacement. In *Computer Architecture (ISCA), 2016 ACM/IEEE 43rd Annual International Symposium on* (2016), IEEE, pp. 78–89.

[70] Jain, A., and Lin, C. Hawkeye: Leveraging belady's algorithm for improved cache replacement.

[71] JaJa, J. *Introduction to Parallel Algorithms*. Addison-Wesley Professional, 1992.

[72] Jaleel, A., Theobald, K. B., Steely Jr, S. C., and Emer, J. High performance cache replacement using re-reference interval prediction (rrip). In *ACM SIGARCH Computer Architecture News* (2010), vol. 38, ACM, pp. 60–71.

[73] Kabiljo, I., Karrer, B., Pundir, M., Pupyrev, S., Shalita, A., Presta, A., and Akhremtsev, Y. Social hash partitioner: a scalable distributed hypergraph partitioner. *arXiv preprint arXiv:1707.06665* (2017).

[74] Kann, V. Maximum bounded 3-dimensional matching in max snp-complete. *Inf. Process. Lett. 37*, 1 (1991), 27–35.

[75] Keramidas, G., Petoumenos, P., and Kaxiras, S. Cache replacement based on reuse-distance prediction. In *Computer Design, 2007. ICCD 2007. 25th International Conference on* (2007), IEEE, pp. 245–250.

[76] Kim, H., Seshadri, S., Dickey, C. L., and Chiu, L. Evaluating phase change memory for enterprise storage systems: A study of caching and tiering approaches. *ACM Transactions on Storage (TOS) 10*, 4 (2014), 15.

[77] Kim, W.-H., Kim, J., Baek, W., Nam, B., and Won, Y. NVWAL: exploiting NVRAM in write-ahead logging. In *ASPLOS* (2016).

[78] Kolli, A., Pelley, S., Saidi, A., Chen, P. M., and Wenisch, T. F. High-performance transactions for persistent memories. In *ASPLOS* (2016).

[79] Koutsoupias, E., and Papadimitriou, C. H. Beyond competitive analysis. *SIAM Journal on Computing 30*, 1 (2000), 300–317.

[80] Kumar, R., Purohit, M., Svitkina, Z., and Vee, E. Interleaved caching with access graphs. In *Proceedings of the Fourteenth Annual ACM-SIAM Symposium on Discrete Algorithms* (2020), SIAM, pp. 1846–1858.

[81] Lee, C. J., Narasiman, V., Ebrahimi, E., Mutlu, O., and Patt, Y. N. Dram-aware last-level cache writeback: Reducing write-caused interference in memory systems. Tech. rep., U.T. Austin, 2010.

[82] Lee, S. K., Lim, K. H., Song, H., Nam, B., and Noh, S. H. Wort: Write optimal radix tree for persistent memory storage systems. In *USENIX Conference on File and Storage Technologies (FAST)* (2017).

[83] Liu, M., Zhang, M., Chen, K., Qian, X., Wu, Y., Zheng, W., and Ren, J. DudeTM: Building durable transactions with decoupling for persistent memory. In *ASPLOS* (2017).

[84] Lucia, B., and Ransford, B. A simpler, safer programming and execution model for intermittent systems. *PLDI* (2015).

[85] Lykouris, T., and Vassilvitskii, S. Competitive caching with machine learned advice. *arXiv preprint arXiv:1802.05399* (2018).

[86] Maddah-Ali, M. A., and Niesen, U. Fundamental limits of caching. *IEEE Transactions on Information Theory 60*, 5 (2014), 2856–2867.

[87] Maeng, K., Colin, A., and Lucia, B. Alpaca: intermittent execution without checkpoints. *OOPSLA* (2017).

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

[89] Meena, J. S., Sze, S. M., Chand, U., and Tseng, T.-Y. Overview of emerging nonvolatile memory technologies. *Nanoscale research letters 9*, 1 (2014), 526.

[90] Memaripour, A., Badam, A., Phanishayee, A., Zhou, Y., Alagappan, R., Strauss, K., and Swanson, S. Atomic in-place updates for non-volatile main memories with Kamino-Tx. In *EuroSys* (2017).

[91] Narayanan, D., Donnelly, A., and Rowstron, A. Write off-loading: Practical power management for enterprise storage. *ACM Transactions on Storage (TOS) 4*, 3 (2008), 10.

[92] Nawab, F., Izraelevitz, J., Kelly, T., Morrey III, C. B., and amd Michael L. Scott, D. R. C. Dali: A periodically persistent hash map. In *DISC* (2017).

[93] Pandurangan, G., and Upfal, E. Can entropy characterize performance of online algorithms? In *Proceedings of the twelfth annual ACM-SIAM symposium on Discrete algorithms* (2001), Society for Industrial and Applied Mathematics, pp. 727–734.

[94] Pandurangan, G., and Upfal, E. Entropy-based bounds for online algorithms. *ACM Transactions on Algorithms (TALG) 3*, 1 (2007), 1–19.

[95] Pelley, S., Chen, P. M., and Wenisch, T. F. Memory persistency. In *ISCA* (2014).

[96] Phalke, V., and Gopinath, B. Compression-based program characterization for improving cache memory performance. *IEEE Transactions on Computers 46*, 11 (1997), 1174–1186.

[97] Qin, H., and Jin, H. Warstack: Improving llc replacement for nvm with a writeback-aware reuse stack. In *Parallel, Distributed and Network-based Processing (PDP), 2017 25th Euromicro International Conference on* (2017), IEEE, pp. 233–236.

[98] Qureshi, M. K., Gurumurthi, S., and Rajendran, B. Phase change memory: From devices to systems. *Synthesis Lectures on Computer Architecture 6*, 4 (2011), 1–134.

[99] Qureshi, M. K., Jaleel, A., Patt, Y. N., Steely, S. C., and Emer, J. Adaptive insertion policies for high performance caching. In *ACM SIGARCH Computer Architecture News* (2007), vol. 35, ACM, pp. 381–391.

[100] Rohatgi, D. Near-optimal bounds for online caching with machine learned advice. In *Proceedings of the Fourteenth Annual ACM-SIAM Symposium on Discrete Algorithms* (2020), SIAM, pp. 1834–1845.

[101] Schindler, J., Griffin, J. L., Lumb, C. R., and Ganger, G. R. Track-aligned extents: Matching access patterns to disk drive characteristics. In *FAST* (2002), vol. 2, pp. 259–274.

[102] Sleator, D. D., and Tarjan, R. E. Amortized efficiency of list update and paging rules. *Communications of the ACM 28*, 2 (1985), 202–208.

[103] Soundararajan, G., Prabhakaran, V., Balakrishnan, M., and Wobber, T. Extending ssd lifetimes with disk-based write caches. In *FAST* (2010), vol. 10, pp. 101–114.

[104] Spielman, D. A., and Teng, S.-H. Smoothed analysis of algorithms: Why the simplex algorithm usually takes polynomial time. *Journal of the ACM (JACM) 51*, 3 (2004), 385–463.

[105] Spielman, D. A., and Teng, S.-H. Smoothed analysis: an attempt to explain the behavior of algorithms in practice. *Communications of the ACM 52*, 10 (2009), 76–84.

[106] Stuecheli, J., Kaseridis, D., Daly, D., Hunter, H. C., and John, L. K. The virtual write queue: Coordinating dram and last-level cache policies. *ACM SIGARCH Computer Architecture News 38*, 3 (2010), 72–82.

[107] Tang, Q., Gupta, S. K. S., and Varsamopoulos, G. Energy-efficient thermal-aware task scheduling for homogeneous high-performance computing data centers: A cyber-physical approach. *IEEE Transactions on Parallel and Distributed Systems 19*, 11 (2008), 1458–1472.

[108] Van Der Woude, J., and Hicks, M. Intermittent computation without hardware support or programmer intervention. In *OSDI* (2016).

[109] van Renen, A., Vogel, L., Leis, V., Neumann, T., and Kemper, A. Persistent memory i/o primitives. In *International Workshop on Data Management on New Hardware* (2019), pp. 12:1–12:7.

[110] Volos, H., Tack, A. J., and Swift, M. M. Mnemosyne: Lightweight persistent memory. In *ASPLOS* (2011).

[111] Wang, Z., Khan, S. M., and Jiménez, D. A. Improving writeback efficiency with decoupled last-write prediction. In *ACM SIGARCH Computer Architecture News* (2012), vol. 40, IEEE Computer Society, pp. 309–320.

[112] Wang, Z., Shan, S., Cao, T., Gu, J., Xu, Y., Mu, S., Xie, Y., and Jiménez, D. A. Wade: Writeback-aware dynamic cache management for nvm-based main memory system. *ACM Transactions on Architecture and Code Optimization (TACO) 10*, 4 (2013), 51.

[113] Wu, K., Arpaci-Dusseau, A., Arpaci-Dusseau, R., Sen, R., and Park, K. Exploiting intel optane ssd for microsoft sql server. In *Proceedings of the 15th International Workshop on Data Management on New Hardware* (2019), pp. 1–3.

[114] Young, N. The k-server dual and loose competitiveness for paging. *Algorithmica 11*, 6 (1994), 525–541.

[115] Young, N. E. On-line file caching. *Algorithmica 33*, 3 (2002), 371–383.

[116] Zhou, M., Du, Y., Childers, B., Melhem, R., and Mossé, D. Writeback-aware partitioning and replacement for last-level caches in phase change main memory systems. *ACM Transactions on Architecture and Code Optimization (TACO) 8*, 4 (2012), 53.