Writeback-Aware Caching

Nathan Beckmann [*] Phillip B. Gibbons [*] Bernhard Haeupler [*] Charles McGuffey [*]

Abstract

The literature on cache replacement, while both detailed and extensive, neglects to account for the flow of data *to storage*. Motivated by emerging memory technologies and the increasing importance of memory bandwidth and energy consumption, we seek to fill this gap by studying the *Writeback-Aware Caching Problem*. This problem modifies traditional caching problems by explicitly accounting for the cost of writing modified data back to memory on eviction.

In the offline setting with maximum writeback cost $\omega > 0$, we show that writeback-aware caching is NP-complete and Max-SNP hard. Moreover, we show that Furthest-in-the-Future, the optimal deterministic policy when ignoring writebacks, is only $(\omega + 1)$-competitive. These negative results hold even for the simple variant of the problem in which data items have unit size, unit miss cost, and unit writeback cost ($\omega = 1$). To overcome this difficulty, we provide practical algorithms to compute upper and lower bounds for the optimal policy on real traces.

In the online setting, we present a deterministic replacement policy called *Writeback-Aware Landlord* and show that it obtains the optimal competitive ratio. Our bounds on the optimal offline policy and our optimal competitive ratio hold even for the most general variant in which data items have variable sizes, variable miss costs, and variable writeback costs. Finally, we perform an experimental study on real-world traces showing that Writeback-Aware Landlord outperforms state-of-the-art cache replacement policies when writebacks are costly, thereby illustrating the practical gains of explicitly accounting for writebacks.

1 Introduction

The long history of papers on caching problems [1, 4, 6, 7, 8, 21, 22, 26, 27, 29, 38, 39, 42, 46, 52, 54, 62, 63] 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, two key trends are causing writebacks to become increasingly important in real memory systems:

Trend 1: Memory Bandwidth and Energy. Traditionally, most 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 [16, 25] has caused power consumption to become critical for computing systems ranging from exascale computing [57] to microcomputing [23]. The practical importance of these metrics has been underscored by a significant amount of systems research [45, 56, 60]. Reducing writebacks reduces the strain on memory system bandwidth and significantly reduces power consumption [33, 45].

Trend 2: New Memory Technologies. Several new main memory technologies that store data in the physical state of material are being developed [47], such as the Intel Optane (3D-Xpoint) technology that is available today as both a solid state drive (SSD) [24] and a memory module (DIMM) [35]. These technologies offer a variety of benefits, including higher storage density, lower idle power, and non-volatility. However, writing data into these memories requires more time and energy than reading data, sometimes by an order of magnitude or more [34, 36, 43, 51, 58].

A variety of research has been done both in the systems [3, 5, 20, 44, 50, 59, 61, 64, 65] and theory [9, 10, 13, 14, 15, 19, 20, 32, 37, 59] communities investigating the effects of this cost asymmetry and how to mitigate it. For the default setting of these systems wherein the (traditional memory) cache sits in front of the new memory, reducing writebacks reduces expensive writes to the new memory.

With these two trends in mind, systems researchers

[*]Carnegie Mellon University. Contact author: Charles McGuffey, cmcguffe@cs.cmu.edu.

have begun to consider writebacks in caching. Initial work has proposed partitioning the cache into a dirty part and a clean part [64] or tracking frequently written data items [50, 61] to reduce total costs. On the theory side, we are aware of only two prior works. Back in 2000, Farach-Colton and Liberatore [28] 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 (which they called *paging with writebacks*). They showed the offline decision problem is NP-complete using a reduction from set cover. Second, Blelloch et al. [13] provided a writeback-aware online algorithm that is 3-competitive to offline optimal when given 3× 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 [54] to each half.

Our Contributions. In this paper, 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 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.

Accounting for writeback costs adds considerable challenges to the caching problem. Intuitively, traditional caching is concerned with making decisions about whether or not to keep a data item x in the cache for the interval (time period) between consecutive accesses to x—the intervals for x are disjoint, and evicting x during an interval incurs a single miss (i.e., at the end of the interval when it is next accessed). When accounting for writebacks, one must consider *competing* intervals for x, namely, the intervals between consecutive *writes* to x. Evicting x during such an interval incurs an additional writeback, whereas keeping x for the entire write interval saves not only this writeback but also all of the reads to x during the write interval.

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

THEOREM 1.1. *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 [63] to properly account for the distinction between clean and dirty items. Comparing to Blelloch et al. [13], 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× more cache to 2-competitive with 2× more cache.

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.

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 savings. 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.

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.

2 Preliminaries and Prior Work

2.1 Caching Basics.

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 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: $L(e) = 1$ [1].

2.2 Prior Work. Theoretical work on the caching problem traditionally begins with the offline version of the problem. The first to be considered was the basic-model, which was solved optimally by Belady [7] and Mattson separately [46]. Chrobak et al. [21] introduced the cost-model for caching and provided an optimal algorithm for its offline version. Albers et al. [1] provided the first algorithms approximating optimal for the offline versions of the generalized-model. Bar-Noy et al. [4] showed an algorithm that is a 4-approximation of optimal for this model. Later, Chrobak et al. [22] proved that the offline decision problem for any caching variant with multiple item sizes (bit-, fault-, and generalized-models) is NP-complete, and Brehob et al. [17] provided similar hardness proofs for several non-standard caching variants. Recently, Berger et al. [11] provided an algorithm that yields tight approximations of offline optimal in the fault-model for traces with certain statistical properties.

Initial work comparing the offline and online versions of caching problems was done by Sleator and Tarjan [54]. They provided a lower bound for the cost ratio of any deterministic online algorithm compared to the optimal offline algorithm for a worst-case trace in the basic-model (and therefore any model). Furthermore, they showed that several deterministic algorithms had matching upper bounds in that model. Fiat et al. [29] provided a lower bound for randomized online replacement policies compared to optimal and a randomized policy that matches that bound; they also showed ways of approximating online policies using other online policies. Young [62] found that the 'greedy-dual' algorithm for the cost-model had an upper bound matching Sleator and Tarjan's lower bound for the basic-model. He later generalized this algorithm to the generalized-model and obtained a matching upper bound [63] using the Landlord algorithm. Even et al. [27] 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 [30, 53] 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 [12, 55]. 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. [64], 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. [61] and Qin and Jin [50] 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.

To our knowledge, the only prior theory work related to writeback-aware caching were the two papers [13, 28] discussed in Section 1.

3 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 3.1. *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.

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

4 An Optimal Online Algorithm

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

4.1 Algorithm Description. Our algorithm is based on the classic Landlord algorithm [63]. 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 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 $V(e)$. This accounting strategy helps in the proof of optimality. The algorithm described in Figure 1 decreases the writeback credit first, but this is not necessary for optimality.

4.2 Frontloading Writeback Accounting. Caches in a writeback-aware setting pay costs at two different times: upon retrieving an item that is not in the cache, and upon evicting a dirty item. Having to consider costs upon eviction increases the complexity of analysis and encourages online policies to maintain dirty items past the point of usefulness in order to delay paying costs. To prevent these issues, in calculating the cost of a policy run on a prefix of a trace, we will charge the cost of writebacks to the write access that dirtied the item. Writes to items that are already dirty are not charged, because they do not result in additional writebacks. In other words, write accesses are charged both for loading the item (if not already in the cache) and writing it back (if it is not already dirty). This does not affect a policy's total cost for the full trace, because each charged writeback will happen later when the item is eventually evicted (recall that we must end with an empty cache).

4.3 Writeback-Aware Landlord is Optimal.

THEOREM 4.1. *Writeback-Aware Landlord with size k has a competitive ratio of $k/(k-h+1)$ to the optimal (offline) algorithm with size $h \leq k$.*

Proof. We consider the contents of two caches: the first is size h and makes optimal caching decisions (OPT), and the second is size k and runs Writeback-Aware Landlord (WALL). Both caches serve the same request trace. For the purposes of the analysis, we say that OPT uses its cache to serve the request first, and then WALL serves the request using its own cache. In Figure 2 we define a potential function Φ, which is carefully defined to capture both how resistant WALL is to change, and how far it is from the state of OPT.

```
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
```

Figure 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.

We show the following:

1. Φ is zero at the beginning of the trace.
2. Φ is never negative.
3. Each cost c paid by Writeback-Aware Landlord can be charged to a unique decrease in Φ of at least $(k-h+1)c$.
4. Φ can only ever increase by an amount kc when the optimal algorithm pays a cost c.

Facts 1 and 2 together mean that Φ can never have

$$\begin{aligned}\Phi = (h-1) \times &\sum_{f \in WALL} \big(credit_l(f) + credit_w(f)\big) \\ &+ k \times \sum_{f \in OPT} (cost_l(f) - credit_l(f)) \\ &+ k \times \sum_{f \in OPT \text{ and dirty}(f)} (cost_w(f) - credit_w(f))\end{aligned}$$

Figure 2: The potential function used to prove the competitive ratio for Writeback-Aware Landlord. Here, *WALL* refers to the contents of the cache for Writeback-Aware Landlord and *OPT* refers to the contents of the cache of the offline optimal policy. The first term is the sum of the credits of each item in WALL's cache. The second and third terms are the difference between how much cost was paid for an item to enter OPT's cache and how much credit that item retains in WALL.

decreased more than it has increased. Fact 3 limits the cost paid by WALL relative to the decrease in potential. Fact 4 limits the increase in potential relative to the cost paid by OPT. When combined, these facts prove that the cost of WALL cannot exceed the cost of the optimal algorithm by a factor larger than $k/(k-h+1)$.

We now provide proofs for the four facts above.

1. At the beginning of a trace, the cache is empty. Therefore, each summation is empty and Φ is zero.

2. Credit values always range between zero and the associated cost of that item. Therefore, Φ is always non-negative.

3. Consider any access that causes charges to WALL. This can happen if the accessed item is not in the cache, or if the access dirties the item. If the item is not in the cache, WALL performs eviction(s) to clear space, and then loads the item. Evicting an item with no credit has no effect on Φ. We apply Young's analysis [63] to the combined credit to show that Φ does not increase when WALL reduces credit.

Young's analysis is applied to the potential function used to analyze Landlord (LL), which is like that of WALL, but does not contain any terms involving writebacks. The analysis compares the total size of items in LL that decrease in credit to the total size of such items in OPT. Since decreasing credit only occurs in order to make space for a requested item and OPT processes requests before LL, we know that the requested item is in OPT's cache but not LL's at the time of the access. This means that the size of items in OPT that have their credit decreased by LL is at most the size of OPT minus the size of the requested item. Furthermore, since LL is evicting items to make space, it must contain a total size greater than its size minus the size of the requested item. The ratio of LL's effected object size to OPT's effected object size is thus greater than the ratio of the two cache sizes, which means that the decrease in potential due to the first term outweighs the increase in potential due to the second term.

When we apply Young's analysis to WALL, we see that the aggregate credit decrease in the first term will remain the same. However some of this decrease will occur in writeback credit rather than load credit. For items that are clean in OPT's cache, this decrease in credit will not show up in the second and third terms of Φ. Since these omitted reductions are to negative terms, Φ will decrease overall.

We then consider the change in credits for the accessed item after eviction. If the item was not in the cache, its load credit changes from zero to its load cost. If the access dirtied the item, the writeback credit changes from zero to the writeback cost. This means that the total credit increase i of the item is at least the cost charged to WALL by the access. Since the item has just been accessed (and OPT serves requests before WALL), it must also be in OPT and be dirty if the access was a write. This means that the second and third terms cause Φ to decrease by ki, while the first increases Φ by $(h-1)i$. Since $i \geq c$, and the potential change due to eviction is not positive, any access that causes a charge c to WALL causes Φ to decrease by at least $(k-h+1)c$.

4. We now show that any increase in Φ can be charged to costs paid by the optimal algorithm. Φ can increase due to changes in credits, items joining OPT, or items in OPT becoming dirty. Credits only decrease when WALL is evicting items, which we have already shown does not increase Φ. The credit for an item is only increased when WALL serves an access to that item. In such cases the item must also be in OPT. Thus, the decrease in Φ due to the second and third terms outweigh the increase due to the first. When an item is loaded into or becomes dirty in OPT, Φ increases by kc where c is the load cost, writeback cost, or their sum, depending on the transition. However, c is exactly the amount that the optimal algorithm is charged to load these items. □

This proof shows that Writeback-Aware Landlord can perform no worse than Sleator and Tarjan's [54] lower bound. Writeback-Aware Landlord therefore achieves the optimal competitive ratio for deterministic policies.

5 Offline Complexity Results

In this section, we show that the offline WA Caching Problem is NP-complete and Max SNP-hard, and we describe both theoretical and practical approximations.

5.1 Writeback-Aware Caching is NP-Complete. 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 [28]. We will provide a brief overview of the FL proof, and then adjust it to match our problem.

The set cover problem is: given a set of elements and non-empty subsets of that set, find a collection of subsets (a cover) of minimum size such that the union of the collection equals the original set of elements. The FL reduction generates an instance of the offline writeback-aware paging problem from an instance of set cover. The cache size is set to the number of subsets. The reduction uses one item in the trace for each element and each subset in the set cover instance. We refer to items associated with elements as element items and items associated with subsets as subset items. For each element, we call the subsets that contain it adjacent subsets, and other subsets non-adjacent subsets.

The generated trace consists of a write to each subset item, followed by a subtrace for each element. The subtrace for an element consists of a write to the associated element item, followed by a read of the element item and the non-adjacent subset items. This read pattern is repeated a total of four times.

The FL reduction shows that any solution to the set cover problem maps to a solution to the paging problem, and that any optimal solution to the paging problem can be converted to a solution to the set cover problem. The high-level idea is that writing back a subset item corresponds to choosing that subset for the cover.

There are two differences between the FL model and ours. The FL model assumes that both loads and writebacks have unit cost for all items, while we support different costs for each item and access type. In the FL model data does not need to be written back to memory if it is not evicted prior to its last use, while we assume all dirty items must eventually be propagated to storage.

To adapt the FL reduction to our data propagation model, we replace each write to an element item in the generated trace with a read to the same item, and we add a write to each set item at the end of the trace.

To support general writeback costs, we define the parameter ω as the maximum writeback cost to read cost ratio for any item. We modify the FL reduction such that the read pattern of the subtrace is repeated $\lfloor \omega + 3 \rfloor$ times rather than four times. This ensures that repeated reads in a subtrace are more valuable than the single writeback that could be saved by forgoing them.

Making these adjustments allows the FL reduction to reduce the set cover problem to the Offline WA Caching Problem. Since set cover is NP-Complete, this suffices to show that the Offline WA Caching Problem is also.

5.2 WA Caching is MaxSNP-Hard. In this section, we show that the Offline WA Caching Problem is max SNP-hard using a reduction from bounded three-dimensional (3D) matching.

The 3D Matching Problem. Consider a hypergraph $G = (V, E)$. We say that $V = \{v_1, v_2, ..., v_n\}$ is the set of vertices in G and $|V| = n$. Similarly, $E = \{e_1, e_2, ..., e_m\}$ is the set of hyperedges in G and $|E| = m$. Each hyperedge e_i consists of a subset of vertices from V that it connects. For each vertex, we call the edges that contain that vertex adjacent edges, and other edges non-adjacent edges. A hypergraph G is tripartite if the vertices can be divided into three disjoint sets $V = \{V_1 \cup V_2 \cup V_3\}$, $V_1 \cap V_2 = V_2 \cap V_3 = V_3 \cap V_1 = \emptyset$ such that no edge contains more than one item from any set. A hypergraph G is three-uniform if each hyperedge is incident upon exactly 3 vertices. A hypergraph G is B bounded if no vertex has degree greater than B.

The maximum bounded 3D matching problem, given a bounded three-uniform tripartite hypergraph G, is to find the largest collection of edges such that no edges in the subset share vertices. More formally, we define M to be a matching of $G = (V, E)$ if $M \subseteq E$ and $\forall\, e_i, e_j \in M, e_i \cap e_j = \emptyset$. We say a matching M of G is maximum if all other matchings M' of G contain at most as many edges as M, i.e. $|M| \geq |M'|$. The decision version of this problem is: given a hypergraph G and an integer k, decide if there exists a matching of cardinality k. This problem is known to be NP-Complete [41] and max SNP-hard [40].

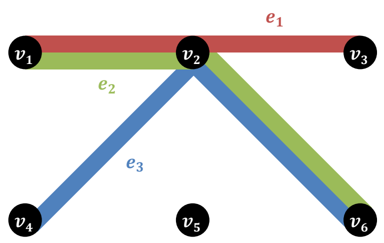
Generating the Caching Instance. Given a 3D matching instance G, we will construct an instance P of the Offline WA Caching Problem such that any valid matching in G corresponds to a solution to P.

An example bounded three-uniform tripartite hypergraph and the generated trace are shown in Figure 3. We will use this example to step through the construction.

Without loss of generality, we discard every vertex from the graph with degree zero.

The cache size of the generated instance will be the number of edges m. The trace will use one *edge item* e_i for each edge $\bar{e_i} \in E$ and $d - 1$ *filler items* $v_{(i,j)}, j \in [1, d-1]$ for each vertex $\bar{v_i}$, where d is the degree of $\bar{v_i}$. All items share a load cost of one and a writeback cost of ω, which can be any positive real. In the example, we set the cache size to three and use three edge items and four filler items (one each for $\bar{v_1}$ and $\bar{v_6}$, and two for $\bar{v_2}$). We also set $\omega = 0.5$.

Like the trace generated by the FL reduction, the trace we generate consists of a prefix and suffix, with a subtrace for each vertex in G. We will refer to the prefix and suffix as gadget $\mathbf{G_1}$. This gadget consists of one write to each edge item.



$\mathbf{G_1}$	$\mathbf{G_2^1}$			$\mathbf{G_3}$	$\mathbf{G_2^2}$			$\mathbf{G_3}$	$\mathbf{G_2^3}$			$\mathbf{G_3}$	$\mathbf{G_2^4}$			$\mathbf{G_3}$	$\mathbf{G_2^6}$			$\mathbf{G_3}$	$\mathbf{G_1}$
$W(e_1)$				e_1				e_1				e_1	e_1	e_1	e_1	e_1	e_1	e_1	e_1	e_1	$W(e_1)$
$W(e_2)$				e_2				e_2	e_2	e_2	e_2	e_2	e_2	e_2	e_2	e_2				e_2	$W(e_2)$
$W(e_3)$	e_3	e_3	e_3	e_3				e_3	e_3	e_3	e_3	e_3				e_3				e_3	$W(e_3)$
	$v_{1,1}$	$v_{1,1}$	$v_{1,1}$		$v_{2,1}$	$v_{2,1}$	$v_{2,1}$										$v_{6,1}$	$v_{6,1}$	$v_{6,1}$		
					$v_{2,2}$	$v_{2,2}$	$v_{2,2}$														

Figure 3: **Example 3D Matching to WA Caching Problem Conversion.** Performing the conversion on the graph above results in the trace below. The trace should be read column-wise from left to right, where each column is read from top to bottom. Requests are reads unless otherwise specified. Gadgets are marked above the trace.

Each subtrace will be composed of two gadgets. Gadget $\mathbf{G_2^i}$ will be created based on the vertex $\bar{v}_i$. This gadget will contain reads of every non-adjacent edge item and every filler item for $\bar{v}_i$. This read pattern will repeat $\lfloor\omega\rfloor + 3$ times. For the example, the gadget $\mathbf{G_2^1}$ generated for vertex 1 would look like $\{R(e_3); R(v_{(1,1)}); R(e_3); R(v_{(1,1)}); R(e_3); R(v_{(1,1)})\}$ for $\omega < 1$. The second gadget in the trace, $\mathbf{G_3}$, consists of reads to each edge item.

Mapping Solutions. Consider any maximum matching for G. We generate a solution for the caching instance as follows: For any time during $\mathbf{G_1}$ or $\mathbf{G_3}$, all m active items can all be kept in the cache. During $\mathbf{G_2^i}$, the $m-1$ items that are being read during the gadget are kept in cache. In addition, if an edge adjacent to $\bar{v}_i$ is in the matching (there can be at most one), that edge item is kept in the cache during the gadget. Otherwise, any of the remaining items can be chosen to remain in cache. The cost of the resulting solution is $7m - 2n + 2m\omega - \omega k$ for a matching of size k.

We now show that the solution generated for the maximum matching is the optimal solution to the caching instance. Because the only cache contention is during $\mathbf{G_2^i}$, we can ignore $\mathbf{G_1}$ and $\mathbf{G_3}$. During $\mathbf{G_2^i}$, retaining each read item for the entirety of the gadget saves $\lfloor\omega\rfloor+2$. Retaining items that are not read during the gadget can save at most $\omega + 1$ (one read and one writeback). It is thus optimal to retain all read items and one adjacent edge item. An edge item can only avoid a writeback if it is retained across all vertex subtraces. Because the matching solution will retain the edge items for each edge in the matching at all times, the matching with the most edges will have the greatest writeback savings.

To generate a solution to the matching problem from the caching solution, simply take for the matching every edge associated with an item held during the entire trace.

Lower Bounding the Size of the Matching. For any 3-uniform tripartite hypergraph G with maximum vertex degree B and m edges, the size of the maximum 3D matching $k \geq m/(3B-2)$.

Consider any edge e in the input graph G. Because G is 3-uniform, e must be incident upon exactly 3 vertices. Each of these vertices can have at most $B-1$ edges other than e incident upon them. e can be adjacent to at most $3(B-1)$ other edges. For any maximum matching M, if none of the edges adjacent to e are in M, then e must be in M. Because we consider any edge in the input graph, there can be at most $3(B-1)$ edges not in M for each edge in M. Dividing the m edges in G by the ratio of edges in the matching finishes the proof.

Generating an Approximation Algorithm. Assume there exists a $1+\epsilon$ approximation algorithm A for the Offline WA Caching Problem. Consider an instance M of the 3D matching problem with maximum matching size k. Let x and x' be the cost of the optimal solution and the solution generated by A, respectively, for the Offline WA Caching instance generated by applying the process above to M. We know from the solution mapping that $x = 7m - 2n + 2m\omega - \omega k$. By algebra, we see that $k = (7m - 2n + 2m\omega - x)/\omega$ and the same relation holds for k' and x'.

When we subtract the k' equation from the k equation, we see that $k - k' \leq (x' - x)/\omega$. By plugging in the relationship between x and x', we get $k - k' \leq \epsilon x/\omega$. By bounding x as a function of m and using the bound relating m and k above, we see that $k - k' \leq \epsilon(7+2\omega)k(3B-2)/\omega$. Rearranging, we get $k' \geq k(1 - \epsilon(7+2\omega)(3B-2)/\omega)$. As ω becomes large, this becomes $k' \geq k(1 - \epsilon 2(3B - 2))$.

This means that any constant approximation algorithm for the Offline WA Caching Problem can be used as a constant approximation to Bounded 3D Matching. Since the matching problem is max SNP-complete, the Offline WA Caching Problem is max SNP-hard.

5.3 Analyzing FitF. In this section, we analyze the performance of the Furthest-in-the-Future (FitF) policy [7, 46] in the presence of writeback costs. FitF, which evicts the item accessed furthest in the future, optimally solves the basic version of the Offline WA Caching Problem. We show how its performance changes when items have a writeback cost of ω units.

THEOREM 5.1. *FitF is an $\omega + 1$ approximation to the basic Offline WA Caching. This bound is tight.*

Proof. Consider a basic Offline WA Caching instance. Let L_B and L_A be the number of loads in the solution generated by FitF and algorithm A, respectively. Because FitF minimizes loads, $L_B \leq L_A$. The number of writebacks an algorithm suffers cannot be greater than the number of loads it suffers, so $W_B \leq L_B$. Through substitution: $Cost_B = L_B + \omega W_B \leq (1 + \omega)L_B \leq (1 + \omega)L_A \leq (1 + \omega)Cost_A$.

We now provide a family of traces where the solution generated by FitF has $\omega + 1 - \epsilon$ times the cost of the optimal solution for arbitrarily small values of ϵ. For a cache of size k, we generate a trace T using $k - 1$ dirty items and $k - 1$ clean items. T consists of a read to each clean item, followed by a write to each dirty item. The family F of traces consists of each trace that is generated by an integral number of repetitions of T.

Because FitF loads the clean items first and makes eviction decisions when they are closer to reuse than the single dirty item in cache at the time, FitF will retain all clean items for the duration of the trace. The optimal solution is to retain all dirty elements for the duration of the trace. In each iteration after the first, FitF will suffer $k - 1$ loads and $k - 1$ writebacks, while the optimal solution will suffer only $k - 1$ loads. Thus the ratio of costs for all iterations after the first will be $\omega + 1$. ☐

One reason that FitF is not optimal is that it is a so-called *stack algorithm* [46]. 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 Appendix A that stack algorithms, despite being intuitive and useful, cannot optimally solve caching problems with multiple costs, such as the WA Caching Problem.

5.4 Approximation Algorithms. Having shown that we cannot solve the Offline WA Caching Problem, we turn to approximate solutions. We provide a method with a theoretical guarantee, and practical upper and lower bounds for the optimal solution. Our practical bounds are based on the work of Berger et al. [11], modified for the writeback-aware setting.

A 2-Approximation for Savings. We provide a scheme 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 Section 5.3), it will provide a 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.

A Practical 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. [11]. 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).

A Practical Upper Bound. We similarly adapt the ideas of Berger et al. [11] 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. The MCF problem and conversion from the caching problem are described in Appendix B. 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.

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 [48]. This experimental 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 [18]. 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.

6.1 Methodology.

Workloads. For our simulations, we make use of block traces from Microsoft Research (MSR) [48]. These traces represent the access patterns experienced by various MSR servers during one week of operation, and represent many commonly seen behaviors. They are distributed in 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.

Competing Policies. We compare Writeback-Aware Landlord primarily against two policies. LRU is the simplest policy commonly used in practice, and works well on traces with high temporal locality. It treats all items equally, ignoring the size and cost of items. GDS is an efficient implementation of (non-writeback-aware) Landlord that considers item cost and size when making decisions, but does not distinguish between reads and writes. 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 2.1 describes these models). Comparing against these policies thus isolates the importance of accounting for writebacks in the WA Caching Problem.

We also compare our results to the offline optimal algorithm. Because this is difficult to compute exactly, we use WAPFOO-L, the lower bound described in Section 5.4. By comparing against WAPFOO-L, we can see how much potential for improvement exists both before and after the application of our ideas.

In Section 6.3, we further compare against GDSF, an extension of GDS that favors frequently accessed objects, and show that Writeback-Aware Landlord can also be effectively extended with such heuristics.

Unfortunately, fair comparisons against prior writeback-aware policies developed for processor caches [64, 60, 50] are not possible because these policies assume items have fixed size (as cache lines do in processors), whereas in the traces we run on, item sizes vary by orders of magnitude. This difference would cause these prior policies to perform poorly for reasons unrelated to writebacks and read-write asymmetry.

Metrics. We compare policies primarily on their total cost over the trace, as defined in Section 3. Because the traces do not specify cost, we consider the fault model, where each item is considered to have unit load cost and writeback cost ω. This represents a system where the cost of communication between the cache and storage is largely independent of the amount of data being communicated, i.e., where latency trumps bandwidth. For most of our experiments, we set $\omega = 10$, which lies between the read-write asymmetry of emerging technologies like Intel Optane [24] and the read-write asymmetry of flash memory (which can range up to $\omega \approx 50$ [31]).

Implementation. The version of Writeback-Aware Landlord described in Figure 1 simplifies the theoretical analysis, but requires work proportional to the number

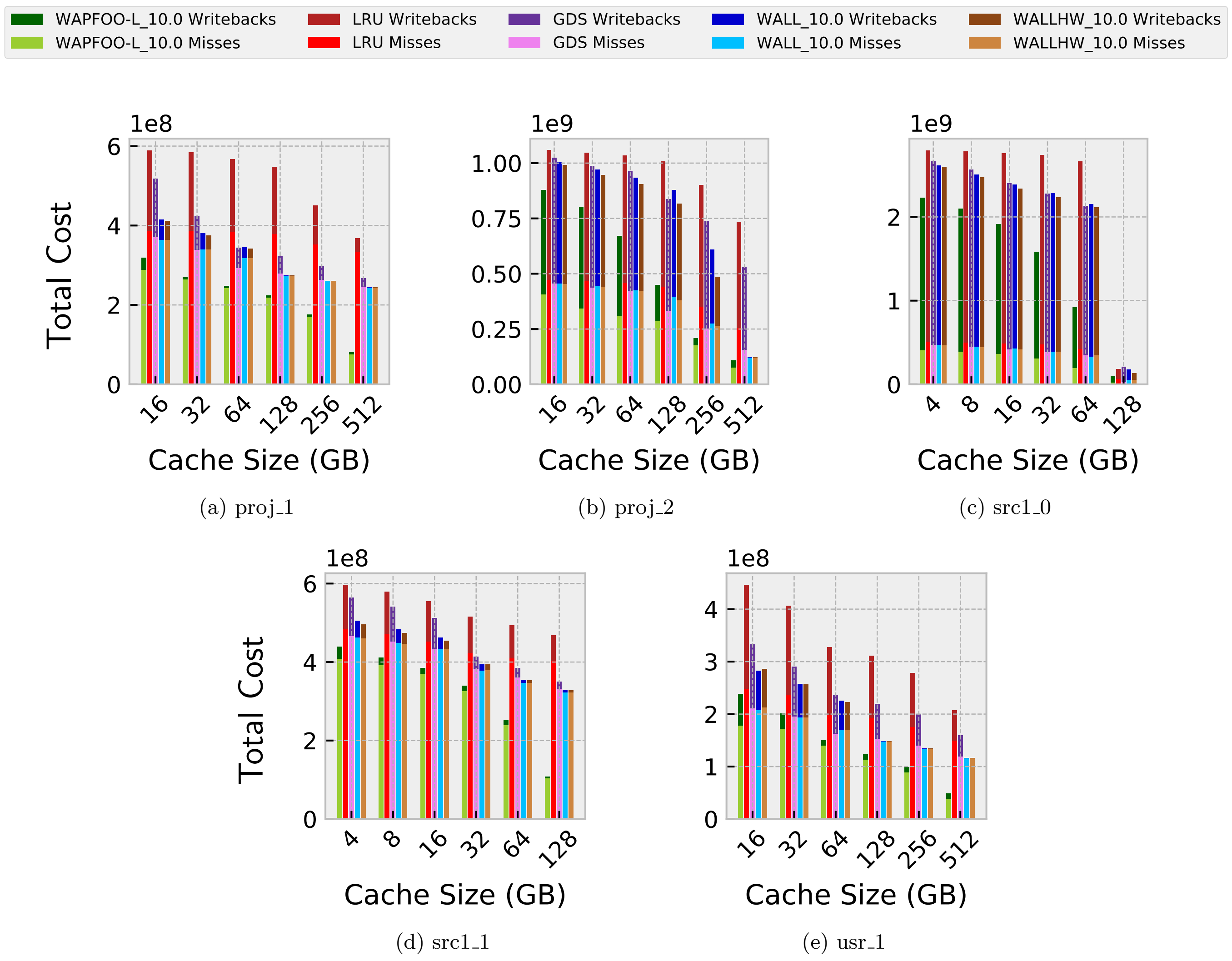


Figure 4: Total cost (misses + weighted writebacks) for different replacement policies on the five storage traces at cache sizes 4–512 GB.

of cached items for each eviction. Because this is impractical, we implement WALL in an equivalent fashion that requires only logarithmic work per eviction. Our implementation is based on Greedy Dual Size (GDS) [18]. In this policy, aging is performed by increasing a global "inflation value" L, rather than decreasing the credit of each item. To maintain credit values equivalently to Figure 1, we combine credits and scale them by the item's size during assignment, e.g.: credit $= L + \text{cost}/\text{size}$. Finally, all credits are stored in a min-heap to avoid scanning over cached items to find a victim.

We also test a version of WALL, called WALLHW, that reduces load credit before writeback credit. The optimality proof in Theorem 4.1 also holds for this policy.

6.2 WALL Reduces Caching Costs. Figure 4 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 (cost $= 1$), and costs due to writebacks (cost $= \omega$). For nearly all traces, both versions of Writeback-Aware Landlord outperform GDS and LRU.

The performance difference is fairly uniform across all traces, excluding *src1_0*. *src1_0* is an outlier: in this trace, 43% of accesses are writes, and the number of bytes written is an even larger fraction. Worse, these writes are distributed across a large number of distinct items, making it impossible for Writeback-Aware Landlord to signficantly reduce writebacks. The other traces have write percentages ranging from 5–12%, providing few enough writes for the extra credit they receive to be

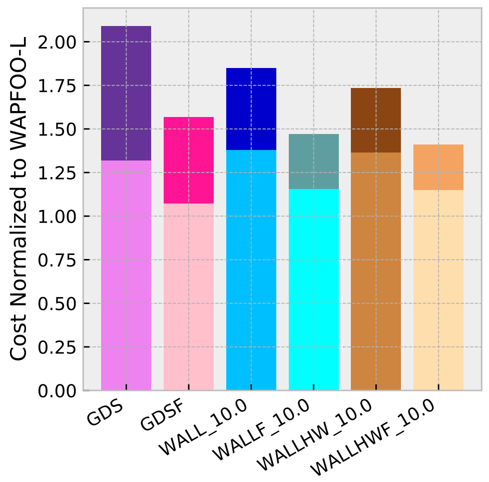


Figure 5: Total cost normalized to WAPFOO-L, averaged across all traces at 128 GB. The light and dark portions of each bar show the cost due to misses and writebacks, respectively.

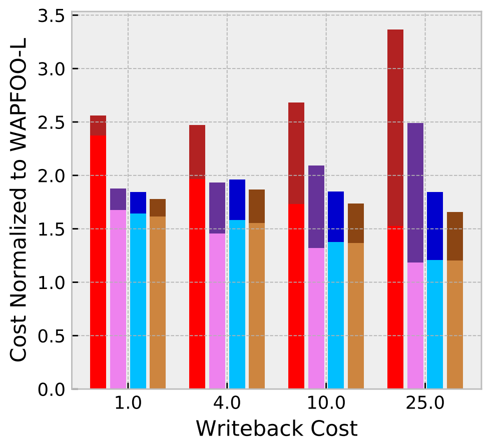


Figure 6: Total cost normalized to WAPFOO-L, averaged across all traces at 128 GB. Writeback-Aware Landlord's benefits improve as writeback cost ω varies from 1–25.

meaningful. WAPFOO-L follows the same general trends as the other policies, but performs meaningfully better. This gap shows that there are still potential gains to be made by better replacement policies.

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%. In other words, WALL significantly saves 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.

6.3 WALL Benefits from Additional Heuristics.

It is common practice for systems to augment replacement policies with heuristics. Among the most popular is frequency, which says that items that have been requested frequently will be requested again. GDSF [2] modifies GDS to account for frequency by multiplying an item's credit by the number of hits it has received while in the cache. Although this algorithm actually has worse theoretical guarantees than GDS, it performs well on real traces. We make a similar modification to Writeback-Aware Landlord, which we call WALLF.

Figure 5 shows the effect of the frequency heuristic on the costs incurred by GDS and WALL at a cache size of 128 GB. Costs are averaged across traces and, to avoid biasing results towards a particular trace, are normalized to WAPFOO-L. We see that considering frequency reduces the number of both misses and writebacks for all considered policies. These results suggest that writebacks share many of the locality patterns seen in loads, and that frequency is a useful indicator of utility. These improvements occur in both GDS and WALL, although they are more pronounced in GDS.

However, the benefits of adding frequency to writeback-aware caches may be less than adding it to writeback-oblivious caches. This could be explained by the fact that both frequency and writeback-awareness are weighting particular items more heavily, which becomes less impactful as it affects more items.

6.4 Sensitivity to Writeback Cost.

Our previous results have assumed that writebacks are 10× as expensive as reads. This cost asymmetry may have a large impact on caching decisions and the resultant costs. Figure 6 shows how the system changes with different cost ratios from $\omega = 1$ to 25, representing a reasonable range from bandwidth-sensitive DRAM systems through storage technologies with heavy read-write asymmetry [31].

GDS does not consider writebacks, so its miss cost remains constant and its writeback cost increases in proportion with ω. However, because these results are normalized to WAPFOO-L, these trends are seen as an increasing fraction of cost spent on writebacks.

WALL's results are more interesting, and show how it trades off misses and writebacks. Overall, WALL's total cost decreases relative to LRU and GDS as ω increases, primarily because it manages to reduce the number of writebacks as they become more valuable. This is at the cost of additional loads, which can be seen in the miss costs for WALL rising relative to GDS with ω. These results show that WALL effectively accounts for cost asymmetry to reduce total cost.

7 Conclusion

Going forward, parallel systems will be increasingly limited by scarce bandwidth and power. Prior work in caching has not considered how writebacks impact these constraints, especially in emerging non-volatile memory technologies with read-write asymmetry. This paper introduced the Writeback-Aware Caching Problem to fill this gap. We showed that optimally solving Writeback-Aware Caching is hard even in the simplest setting, and we developed an online replacement policy with strong theoretical guarantees and good empirical performance. We believe that these results will help build a foundation for further theoretical and empirical work in caching on systems constrained by energy or bandwidth.

Acknowledgements

This work is supported in part by NSF grants CCF-1533858, CCF-1618280, CCF-1814603, CCF-1527110, SHF-1815882, CCF-1725663, CCF-1750808, CCF-1919223, a Google Faculty Research Award, and a Sloan Research Fellowship.

References

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

[2] M. Arlitt, L. Cherkasova, J. Dilley, R. Friedrich, and T. Jin, *Evaluating content management techniques for web proxy caches*, ACM SIGMETRICS Performance Evaluation Review, 27 (2000), pp. 3–11.

[3] J. Arulraj and A. Pavlo, *How to build a non-volatile memory database management system*, in Proceedings of the 2017 ACM International Conference on Management of Data, ACM, 2017, pp. 1753–1758.

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

[5] D. Bausch, I. Petrov, and A. Buchmann, *Making cost-based query optimization asymmetry-aware*, in Proceedings of the Eighth International Workshop on Data Management on New Hardware, ACM, 2012, pp. 24–32.

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

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

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

[9] N. Ben-David, G. E. Blelloch, J. T. Fineman, P. B. Gibbons, Y. Gu, C. McGuffey, and J. Shun, *Parallel algorithms for asymmetric read-write costs*, in Proceedings of the 28th ACM Symposium on Parallelism in Algorithms and Architectures (SPAA), ACM, 2016, pp. 145–156.

[10] ——, *Implicit decomposition for write-efficient connnectivity algorithms*, in International Parallel and Distributed Processing Symposium (IPDPS), IEEE, 2018, pp. 711–722.

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

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

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

[14] ——, *Efficient algorithms with asymmetric read and write costs*, in European Symposium on Algorithms, 2016.

[15] G. E. Blelloch, Y. Gu, J. Shun, and Y. Sun, *Parallel write-efficient algorithms and data structures for computational geometry*, in Proceedings of the 30th ACM Symposium on Parallelism in Algorithms and Architectures (SPAA), ACM, 2018, pp. 235–246.

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

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

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

[19] E. Carson, J. Demmel, L. Grigori, N. Knight, P. Koanantakool, O. Schwartz, and H. V. Simhadri, *Write-avoiding algorithms*, in 2016 IEEE International Parallel and Distributed Processing Symposium (IPDPS), IEEE, 2016, pp. 648–658.

[20] S. Chen, P. B. Gibbons, and S. Nath, *Rethinking database algorithms for phase change memory*, in Proc. Conference on Innovative Data Systems Research (CIDR), 2011.

[21] M. Chrobak, H. J. Karloff, T. H. Payne, and S. Vishwanathan, *New results on server problems*, in

SIAM Journal on Discrete Mathematics, 1991, pp. 172–181.

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

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

[24] I. Corporation, *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-`.

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

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

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

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

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

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

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

[32] Y. Gu, Y. Sun, and G. E. Blelloch, *Algorithmic building blocks for asymmetric memories*, in European Symposium on Algorithms, 2018, pp. 44:1–44:15.

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

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

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

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

[37] R. Jacob and N. Sitchinava, *Lower bounds in the asymmetric external memory model*, in Proceedings of the 29th ACM Symposium on Parallelism in Algorithms and Architectures (SPAA), ACM, 2017, pp. 247–254.

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

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

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

[41] R. M. Karp, *On the computational complexity of combinatorial problems*, Networks, 5 (1975), pp. 45–68.

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

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

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

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

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

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

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

[49] J. B. Orlin, *A faster strongly polynomial minimum cost flow algorithm*, Operations research, 41 (1993), pp. 338–350.

[50] H. Qin and H. Jin, *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, IEEE, 2017, pp. 233–236.

[51] M. K. Qureshi, S. Gurumurthi, and B. Rajendran,

Phase change memory: From devices to systems, Synthesis Lectures on Computer Architecture, 6 (2011), pp. 1–134.

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

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

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

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

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

[57] Q. Tang, S. K. S. Gupta, and G. Varsamopoulos, *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 (2008), pp. 1458–1472.

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

[59] S. D. Viglas, *Write-limited sorts and joins for persistent memory*, Proceedings of the VLDB Endowment, 7 (2014), pp. 413–424.

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

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

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

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

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

[65] O. Zilberberg, S. Weiss, and S. Toledo, *Phase-change memory: An architectural perspective*, ACM Computing Surveys (CSUR), 45 (2013), p. 29.

A Stack Algorithms

Part of what makes the basic caching problem so readily tractable is the fact that most good algorithms to solve it are *stack algorithms*. Stack algorithms, defined by Mattson et al. [46], 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.

Stack algorithms are useful for several reasons. On an intuitive level, they make the problem easier to reason about, because each cache decision can be considered individually. Stack algorithms can be more easily computed using greedy algorithms or dynamic programming. They are also easy on system designers, as multiple cache sizes can be simulated on a trace simultaneously [46].

To the best of our knowledge, stack algorithms have not been investigated in any model other than the basic model. Here, we show that stack algorithms are not optimal in the presence of multiple item costs or multiple item sizes.

Consider the Offline WA Caching instance shown in Figure 7. The optimal solution for a cache of size 2 is to hold items B, D, and E in the cache. When the cache size increases to 3, a stack algorithm must keep each of these items. However, the optimal solution is to hold A, B, C, and E in the cache, dropping D. This means that the optimal solution is not a stack algorithm.

Such bad cases are not limited to small cache sizes, or to a single change in cache size. It is possible to construct an example where the optimal solution for a cache of size k is not a subset of the optimal solution for a cache of size $k+1$ for any value of k by modifying the trace in Figure 7 to replace each item and request in the trace with $k-1$ items and one request for each of the replacement items, respectively. The optimal solution for a cache of size k retains all replacement items for B, D, and E, while the optimal solution for size $k+1$ will replace one of the D items with one of the A items and one of the C items. Furthermore, as the cache increases in size from k to $2k$, the D items will gradually be replaced with A and C items.

Our construction holds for any variant of caching with multiple costs. As long as each request interval for A, B, C, and E provide more potential savings than the request intervals for D, then the cache will switch from D to A or C as soon as the space becomes available.

It is also straightforward to construct traces with multiple item sizes where stack algorithms are non-optimal. An example is having multiple items share the same time period with access frequency proportional to the square of item size. As the cache becomes large enough to accommodate larger items, these items will displace the lesser-used smaller items.

$(A, \mathbf{W})$, $(B, \mathbf{W})$, $(F, \mathbf{R})$, $(B, \mathbf{W})$, $(C, \mathbf{W})$, $(D, \mathbf{R})$, $(G, \mathbf{R})$, $(D, \mathbf{R})$, $(A, \mathbf{W})$, $(E, \mathbf{W})$, $(H, \mathbf{R})$, $(E, \mathbf{W})$, $(C, \mathbf{W})$

Figure 7: **An Example Trace that Breaks Stack Algorithms.**

$(A, \mathbf{W}), (B, \mathbf{R}), (A, \mathbf{R}), (A, \mathbf{R}), (A, \mathbf{W}), (B, \mathbf{R}), (C, \mathbf{R}), (C, \mathbf{W}), (C, \mathbf{R}), (A, \mathbf{W})$

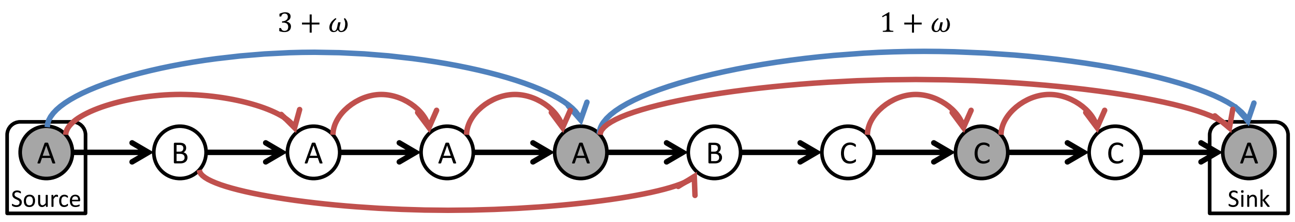


Figure 8: **Example WA Caching Problem to MCF Conversion.** The trace above is converted to the MCF problem below. All items are said to have load cost 1 and writeback cost ω. Black edges have cost 0 and capacity equal to the cache size. Red edges have cost -1 and capacity 1. Blue edges have labeled cost and capacity 1.

Because we have constructed examples that break stack algorithms for varying costs and varying sizes, we claim the following.

Theorem A.1. *For any caching problem with multiple costs or multiple sizes, the optimal solution is not a stack algorithm.*

B Writeback-Aware Caching and Minimum Cost Flow

The minimum cost flow problem is commonly used to model offline versions of caching problems. Some of our approximations make use of this technique, which we describe here.

Minimum Cost Flow. The *minimum cost flow* (MCF) problem [49] consists of a directed graph $G = \{V, E\}$ and an amount of flow f. One vertex $s \in V$ is designated as the source vertex and another vertex $t \in V$ is designated as the sink. Each edge $e \in E$ has both a cost per unit flow $c(e)$ and flow capacity $u(e)$ associated with it. The goal of the problem is to route f units of flow from the source to the sink while minimizing the total cost. Each vertex other than the source and sink must have the same amount of flow leaving and entering.

Converting Between Problems. An example trace and the generated MCF problem are shown in Figure 8. The transformation creates one vertex in the graph for each request in the trace. For simplicity, we will refer to vertices as if they were the requests they represent. The first and last requests are chosen as the source and sink, respectively. To simulate empty cache space between requests, we generate an edge from each request to the next with cost 0 and capacity k. For modeling load savings, we generate an edge between subsequent requests to the same item with cost equal to the item's load cost and capacity 1. We model writeback savings with an edge between each write and the subsequent write to the same item. Edges representing writebacks have cost equal to the item's writeback cost plus the sum of the costs of load intervals for that item that overlap with the writeback interval. In the example, we show edges representing loads and writebacks in red and blue, respectively. We set the flow from source to sink to be equal to the size of the cache. The result is a directed acyclic graph (DAG) that approximates the cost savings that can be found in the instance of the basic WA Caching Problem.

Solving the generated MCF problem provides a close approximation to the solution of the original WA Caching Problem. It is not exact, because a solution to the MCF problem can obtain savings from an item twice during same time period. However, it is a useful foundation that algorithms can build upon.