SieveStore: A Highly-Selective, Ensemble-level Disk Cache for Cost-Performance

Timothy Pritchett
School of Electrical and Computer Engineering
Purdue University
tpritche@purdue.edu

Mithuna Thottethodi
School of Electrical and Computer Engineering
Purdue University
mithuna@purdue.edu

ABSTRACT

Emerging solid-state storage media can significantly improve storage performance and energy. However, the high cost-per-byte of solid-state media has hindered wide-spread adoption in servers. This paper proposes a new, cost-effective architecture – SieveStore– which enables the use of solid-state media to significantly filter access to storage ensembles. Our paper makes three key contributions. First, we make a case for highly-selective, storage-ensemble-level disk-block caching based on the highly-skewed block popularity distribution and based on the dynamic nature of the popular block set. Second, we identify the problem of *allocation-writes* and show that selective cache allocation to reduce allocation-writes – *sieving* – is fundamental to enable efficient ensemble-level disk-caching. Third, we propose two practical variants of SieveStore. Based on week-long block-access traces from a storage ensemble of 13 servers, we find that the two components (sieving and ensemble-level caching) each contribute to SieveStore's cost-effectiveness. Compared to unsieved, ensemble-level disk-caches, SieveStore achieves significantly higher hit ratios (35%-50% more, on average) while using only $1/7^{th}$ the number of SSD drives. Further, ensemble-level caching is strictly better in cost-performance compared to per-server caching.

Categories and Subject Descriptors

D.4.2 [**Operating Systems**]: Storage Management—Storage hierarchies

General Terms

Design, Performance

1. INTRODUCTION

The storage layer is a significant factor in the capital cost, operating cost (energy), and performance of servers and data-centers. Emerging high-performance, low-energy, non-volatile, solid-state storage media (e.g., flash-based memory, phase-change memory) show enormous promise to solve two of the three problems (storage performance and energy); but cost remains a concern. Recent work has examined several ways to use flash-based solid-state media; as a buffer-cache [11, 10], as swap space for virtual memory [19], and as the storage component of custom-architectures [3]. Further, there has also been research on the organization of solid-state drives (SSDs) for improved performance [4, 7]. However, in spite of the many advances and significantly higher performance of solid-state drives (SSDs), the high cost-per-byte of solid-state media has hindered wide-spread adoption in servers. Per-server SSD deployment incurs excessive cost for modest performance improvements. To enable servers to benefit from high-performance SSDs, we propose a highly-selective, ensemble-level disk cache – SieveStore– which represents a superior cost-performance point. SieveStore enables the use of a small amount of solid-state media (∼16GB-32GB, thus reducing cost) to serve a significant fraction of accesses (thus improving performance) to larger storage ensembles that support several servers (e.g., 10+ servers, 5-10 TB total capacity, 1.5TB-2.5TB daily accesses).

SieveStore's architecture is based on the following two key observations from analysis of storage access traces of an ensemble of 13 servers over a week [14, 15].

(O1) A very small fraction (∼1%) of "popular" blocks accessed each day account for a significant fraction of accesses (between 14%-53%). Beyond the top 1% of blocks, the number of accesses per block diminishes rapidly. For example, 99% of all blocks accessed in a day see 10 or fewer accesses. The least popular 97% of all blocks accessed in a day see 4 or fewer accesses. This is not surprising because the buffer caches in memory filter out most reuse leaving very little reuse at the block device layer.

(O2) Though the degree of skew for the ensemble as a whole remains invariant (i.e., not much reuse beyond the top 1% popular blocks), the distribution of the popular blocks varies across servers, across storage volumes of the same server and over time.

Per-server disk-caches incur high cost, and may also perform poorly because the disk-cache capacity associated with servers that have few hot blocks cannot be shared by the hot blocks of other servers. In contrast, a small, shared, ensemble-level disk-cache of the popular blocks would be better because (1) the combination of small and shared capacity keeps costs low and (2) the cache would capture the popular blocks (via temporal locality) as well as the dynamic changes in the popular block set (by dynamically sharing cache space).

However, while the costs of a small, shared disk-cache are indeed low, achieving high performance is challenging

ISCA'10, June 19–23, 2010, Saint-Malo, France.

because conventional disk-cache allocation policies such as *allocate-on-demand (AOD)*, which allocates on a miss, and *write-no-allocate (WMNA)*, which allocates only on read misses, will result in poor performance because of three problems. First, because a large number of blocks have little reuse (O1), a large fraction of accesses will be misses, each of which will allocate and write a block in the cache (*allocation-writes*). Because the cache uses write-asymmetric storage media like flash-based SSDs, where writes are significantly slower than reads, such allocation-writes can have a significant impact on performance. Second, the large number of low-reuse blocks can cause cache pollution, reducing the hit ratio. Third, aggregating the hot blocks of several servers in a single disk cache may create a bandwidth bottleneck.

To avoid the cost of allocation-writes (the first problem), SieveStore uses selective cache allocation policies that aim to reduce allocation writes while improving the hit-ratio – *sieving*. One may think that sieving is simply an alternate way to perform the same function as the replacement policy (i.e., modulate the contents of the cache). However, while they may both be used to control the contents of the cache (and hence the hit-ratios), only sieving can limit allocation-writes caused by misses to low-reuse data. We show in Section 3.1 that, in the absence of sieving, even an ideal (oracle) replacement policy will not achieve significant performance improvement. We also show that using selective allocation to maximize hits does not necessarily minimize allocation-writes.

Sieving, if done accurately to capture the popular blocks, avoids the problem of cache pollution (the second problem). However, correctly identifying popular blocks poses a challenge that conventional caches do not face. Conventional caches that use unsieved allocation (i.e., AOD or WMNA) maintain metastate for only those blocks that are present in the cache because the allocation decision depends only on the current state of the cache (hit/miss) and the type of the request (read/write). In contrast, sieving fundamentally requires us to maintain state for many blocks that are *not* present in the cache. We propose two practical sieving variants that differ in how they identify popular blocks and in how they maintain metastate.

The first approach is an offline, discrete-allocation variant – SieveStore-D – which enforces selectivity using an *access-count based discrete batch-allocation (ADBA)* mechanism. SieveStore-D maintains precise metastate, by logging each access, and periodically combines the metastate in an offline pass. Blocks are allocated a frame only if their access count in an epoch exceeds a threshold. The second approach is an online, continuous allocation variant – SieveStore-C – which sieves accesses using a *hysteresis-based, lazy cache allocation* mechanism wherein disk blocks are allocated a frame on the n^{th} (for some threshold n) miss over a recent time window. To minimize the metastate of uncached blocks, SieveStore-C uses a two-tier structure; a preliminary tier that sieves with imprecise (potentially aliased) metastate, followed by an accurate tier to maintain quality of sieving.

Finally, in spite of aggregating the hot blocks of several servers in a single cache, SieveStore can handle the increased traffic, partly because SSDs that are currently in the market offer significantly higher I/O throughput (IOPS), and partly because correlated I/O bursts across independent servers in the ensemble are rare [14]. Both variants of SieveStore can handle the increased traffic with only one enterprise-class SSD over 99.9% of the time and with two SSDs 100% of the time. Further, without any special effort on the part of SieveStore to limit writes, there are no SSD wearout/lifetime problems even though SieveStore caches write-hot blocks as well.

	Sieved	Unsieved
Ensemble–Level	SieveStore–C SieveStore–D	II
Per–Server	III	IV

Figure 1: Design space

Implementation and Overheads: A SieveStore implementation can be realized as a transparent appliance (with processor, memory, SSD drives) that plugs in to existing storage ensembles at a data-center. The minimal cost of the one additional box for a small data-center has to be weighed against the benefit of significant reduction in accesses to the storage ensemble.

In summary, the major contributions of this paper are:

- Driven by our observations of (1) extreme popularity skew in real storage traces, and (2) dynamic changes in the popular block sets, we propose SieveStore – a highly-selective, ensemble-level disk-cache.
- We identify *sieving* – selective cache-allocation to reduce allocation-writes while improving the hit ratio – as a fundamental mechanism to enable cost-effective, ensemble-level disk-caching.
- We propose two variants of SieveStore – one with discrete sieving (SieveStore-D), and another with continuous sieving (SieveStore-C).

The two key ideas proposed in this paper – ensemble-level disk caching and sieving – can be applied independently of each other. Consider the quadrants in the design space, as shown in Figure 1. To show that SieveStore, which combines both the ideas (i.e., quadrant I in Figure 1), is a superior cost-performance design point, we evaluate SieveStore against other quadrants using traces of a storage-ensemble of 13 servers over a week. Our key results are:

- SieveStore-D and SieveStore-C, which lie in quadrant I, capture 35% and 50%, respectively, more accesses than the best unsieved ensemble-level disk-cache (quadrant II) while requiring $(1/7)^{th}$ the number of SSD drives.
- Both variants of SieveStore, due to to their ensemble-level caching approach, capture more accesses at the same cost (and the same number of accesses at lower cost) than an ideal per-server cache (quadrants III and IV), which is not surprising given the dynamic nature of popular blocks (O2).

The rest of the paper is organized as follows. Section 2 analyzes the traces to support our observations. Section 3 describes the two SieveStore variants. Section 4 describes our evaluation methodology. Section 5 discusses experimental results. Related work is described in Section 6. Section 7 briefly discusses some forward-looking issues on scaling and tuning. Finally, Section 8 concludes this paper.

Table 1: Trace Summary (from [14, 15])

Key	Name	Volumes	Spindles	Size (GB)
Usr	User home dirs	3	16	1367
Proj	Project dirs	5	44	2094
Prn	Print server	2	6	452
Hm	Hardware monitor	2	6	39
Rsrch	Research projects	3	24	277
Prxy	Web proxy	2	4	89
Src1	Source control	3	12	555
Src2	Source control	3	14	355
Stg	Web staging	2	6	113
Ts	Terminal server	1	2	22
Web	Web/SQL server	4	17	441
Mds	Media server	2	16	509
Wdev	Test web server	4	12	136
	Total	36	179	6449

2. STORAGE ENSEMBLE ACCESS CHARACTERISTICS

We analyze storage ensemble behavior using traces from [14]. The traces capture requests to block devices below the buffer cache, and are typical of small/medium datacenters. Table 1 reproduces the key trace characteristics (reproduced from [14]). The traces span eight calendar days for each server although the trace length is for seven days because trace collection started at 5:00pm GMT on the first day. We analyze all traces on a calendar day basis, so we treat it as an 8-day trace.

Popularity Skew.

For each day of the trace, we sort the blocks in descending order of popularity and group them into 10,000 bins such that each bin contains 0.01% (=$1/10,000$) of all blocks accessed on that day. Figure 2(a) plots the average access count of each bin (Y-axis, log-scale) against the percentile rank of the bin (X-axis, log-scale) with one curve for each day. For example, the vertical line at 1% corresponds to the 100^{th} bin of blocks which is at the top first percentile in popularity. Though the most popular 0.01th percentile bin of blocks have an average of over 1000 accesses on each day, the bin at the top 1st percentile, averages fewer than 10 accesses per day. Further, because the averages at the top 1st percentile do not hide any large variations within the bin, the maximum number of accesses (not shown) is also 10 on all days except on day 2 when it is 11. Similarly, when we exclude the top 3%, blocks have fewer than 4 accesses on average. Below the 50^{th} percentile, blocks that are accessed are never reused.

To illustrate the fraction of accesses the popular blocks account for Figure 2(b) plots for each bin (shown in ranked percentile order on the X-axis), the cumulative fraction of accesses (Y-axis) for all bins that are at higher percentiles. Because the knee of the curve occurs close to the very top 1st percentile, we show a zoomed in graph of the same data but limit the X-axis to the top 5% most popular blocks in Figure 2(c). The knee-of the curve occurs at less than 1% of blocks accessed on any given day. Though the amount of data accessed in a day varies from a minimum of 335 GB to a maximum of 1190 GB (685GB/day, on average) for the ensemble, the most popular 1% of blocks is significantly smaller (at most 11.9GB) and would fit comfortably within a modest 16-32GB SSD with room to spare.

Note, day 1's accesses are an outlier because of our decision to use traces partitioned by calendar days.

Popularity Skew Variation.

While the above analysis considered the trace of the entire storage ensemble, it is important to understand whether the observed popularity skew is truly a property that emerges at the ensemble-level even though that behavior may not be uniformly present in the traces of individual servers, or if the same behavior is also observable at the individual server level. If each server's accesses exhibits the same skewed behavior, then the per-server caching approach (quadrants III and IV from Figure 1) might be attractive (ignoring minimum drive sizes). If, on the other hand, there is significant variation in behavior at the individual server level, then an ensemble-level approach (quadrants I and II) may be well-motivated.

Our trace analysis (Figure 3) reveals that the popularity skew behavior exhibits significant variation across servers, across storage volumes of the same server and across time for the same server. Figure 3(a), which plots the cumulative access distribution (Y-axis) for block bins (in sorted, descending order of access count on the X-axis) from the proxy server (Prxy) and the source control server (Src1), illustrates server-to-server variation in popularity skew. The proxy server exhibits popularity skew with a small fraction of blocks accounting for nearly all of the days accesses. On the other hand, the source control server's near-linear cumulative access count shows that the popularity skew is minimal. Figure 3(b) uses similar axes with curves for individual storage volumes within a single server (Web, volumes 0 and 1) to compare the access characteristics of storage volumes within a server. While both volumes exhibit some degree of skew for the most popular blocks, volume-0 exhibits significantly more skew than volume-1. Even within the same server the popularity skew may vary in time as shown in Figure 3(c). For the web staging server (Stg), day 5 exhibits significant popularity skew, but day 3 does not.

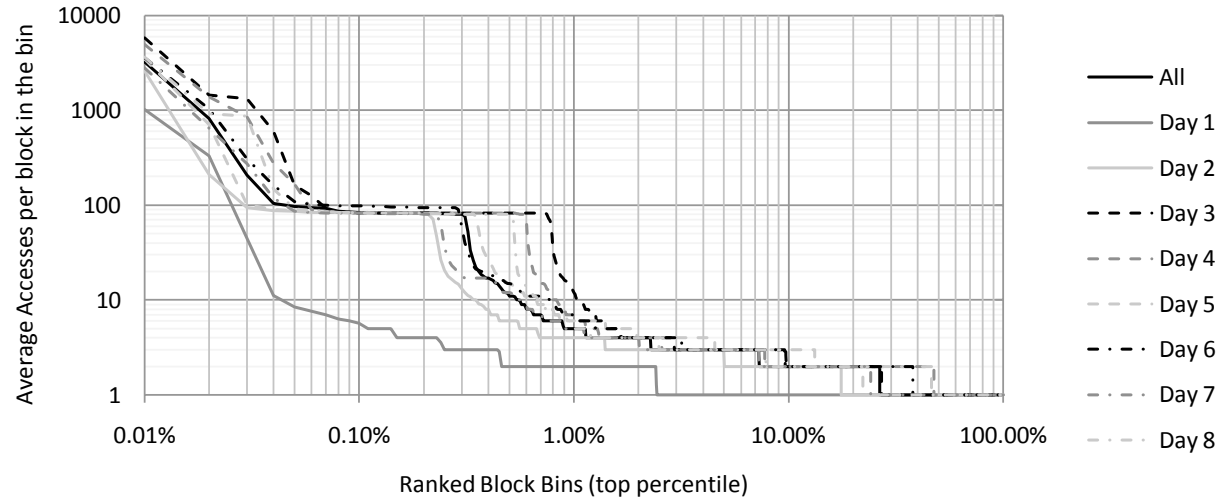
Finally, returning to the ensemble trace, Figure 3(d) plots the composition of the most popular 1% of blocks in the ensemble in terms of what fraction is contributed by each server (Y-axis) for each day of the trace (X-axis). The variation in contribution from each server demonstrates time-varying behavior that no statically (potentially unequally) partitioned per-server cache can capture.

Summary.

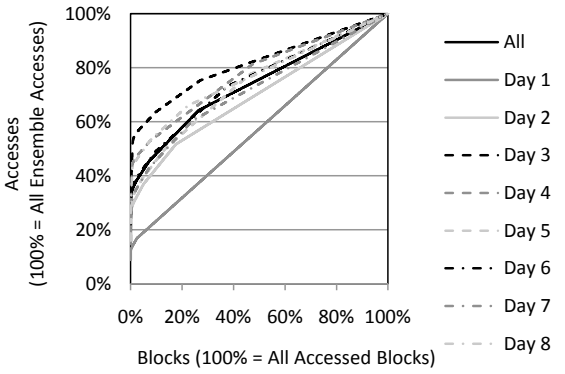
Trace analysis reveals that there is significant variation in popularity skew across servers, within servers across storage volumes, and in time for the same server. Despite such variation, the aggregate trace of the whole ensemble exhibits stable emergent behavior wherein the most-accessed 1% of blocks account for a large fraction of total accesses. The next section examines how this observation may be exploited to improve storage.

3. SieveStore

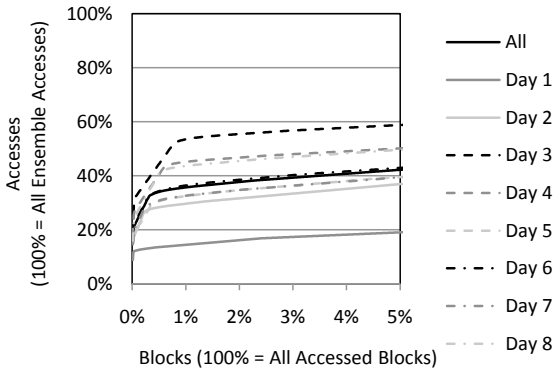
The goal of SieveStore is to capture the top 1% most popular blocks of the storage ensemble in an SSD cache to achieve cost-effective storage acceleration. There are two ways to modulate the contents of a cache to ensure that it holds the desirable blocks; either by controlling who gets in (allocation



(a) Block access count distribution



(b) Block popularity (CDF)



(c) Zoomed-in CDF for Top 5th percentile

Figure 2: Popularity Skew Characterization

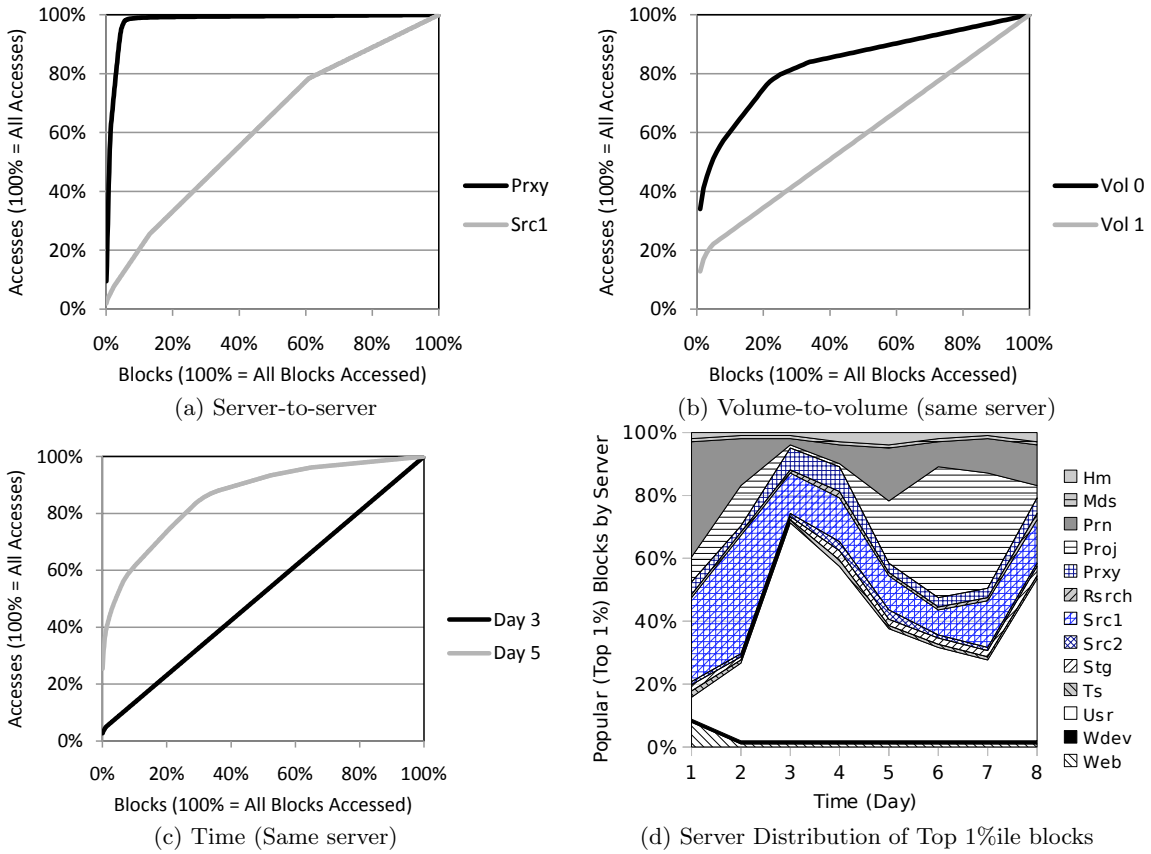
policy) or by controlling who gets evicted (replacement policy). One of the claims in this paper is that disk-caching at the ensemble-level fundamentally requires selective cache allocation to prevent low-reuse blocks from entering the cache. Section 3.1 uses a thought-experiment to justify the above claim by showing that (1) in the absence of sieving, even an ideal (oracle) replacement policy will not achieve significant performance improvement and (2) an ideal (but impractical) sieve offers significant benefits. We also show that performing selective allocation solely to maximize hits is inadequate to control the number of allocation-writes. Section 3.2 and Section 3.3 describe our two practical sieve implementations that capture most of the benefits of the ideal (impractical) sieve.

3.1 The case for sieving

Consider an oracle-replacement algorithm that magically evicts only those blocks that are not in the top 1% frequently accessed blocks on each day. Narayanan *et.al.* consider a similar oracle-replacement algorithm called LTR (Long-term Random; a policy that retains frequently accessed blocks over the long-term) at the per-server level [15]. (Note, the above oracle policy is not the same Belady's MIN/OPT oracle replacement algorithm [2] which replaces the block which is accessed farthest out in the future. We discuss MIN briefly at the end of this section.)

To isolate the cost of allocation-writes from that of misses, we conservatively assume that using the above-defined ideal (oracle) replacement policies can ensure that the top 1% of blocks are always cache-resident for both on-demand allocation (AOD) and the write-miss no-allocate (WMNA) allocation policies. We can now compare the number of allocation writes for each of the allocation policies, as compared to an ideal selective allocation policy (distinct from the ideal replacement policy). For AOD, each access (hit or miss) causes an SSD operation with a (slow) allocation-write operation for each miss. Table 2 computes the number of writes assuming a hit rate of 35% (the approximate average hit-rate for the ideal-allocation scheme over all eight calendar days), and assuming a 3:1 ratio of reads and writes in both hits and misses. The mechanism will cause 73.75% of all ensemble accesses to initiate writes to a single SSD (see first row, Table 2). The number of SSD operations increase from 35% (hits only) to 100% (all accesses), a large fraction of which are slow writes (73.75%).

With the WMNA allocation policy and the oracle replacement policy (second row of Table 2), while the hit-ratio remains unchanged, allocation-writes are avoided on write misses. (Note, changing allocation policies does affect the



(a) Server-to-server

(b) Volume-to-volume (same server)

(c) Time (Same server)

(d) Server Distribution of Top 1%ile blocks

Figure 3: Popularity skew variation

hit-ratio. However, because of our assumption that the oracle policy ensures that the most popular 1% of blocks always remains cache-resident, we conservatively assume the same hit rate.) Allocation writes will account for 48.75% (read misses $= (1 - 35\%) \times 3/4$) of all the accesses, which results in (1) more than doubling the number of SSD operations (2.4X), and (2) increasing the number of SSD writes by a factor of 5.6X. In contrast, the ideal allocation cache causes exactly 1% of accessed blocks to result in allocation-writes to bring in the top 1% of blocks. Note, the number of allocation-writes is 1% of the number of unique blocks accessed which is smaller than 1% of the accesses (and hence written as ϵ% in the last row of Table 2).

To extend the analysis to Belady's MIN algorithm, (assuming AOD allocation, as in the original paper) we attain bounds on the number of allocation-writes in terms of number of unique blocks (as opposed to number of accesses used in Table 2). Recall from Figure 2(a) that 50% of the blocks have at most 1 access and that the next 47% of blocks have 4 or fewer accesses, which implies that 97% of blocks will incur a minimum of 50% + 47%/4 = 61.75% compulsory misses, and hence allocation-writes. Note, in comparison that ideal sieving requires only 1% of blocks to be allocated.

Finally, we show that extending Belady's replacement algorithm to do selective allocation does not necessarily minimize allocation-writes. Such an extension would allocate a block only if the block's next use is earlier than the next use of at least one of the blocks in the cache. However, while such an extension will maximize the number of hits, it is not guaranteed to minimize allocation-writes. A simple example can be constructed using a 1-entry cache and the address stream `a,a,b,b,a,a,c,c,a,a,d,d,a,a,e,e,...` and so on. Belady's selective allocation would allocate blocks in `a,b,a,c,a,d,a,e,...` order with the long term hit ratio converging to 50%. Effectively, each miss causes an allocation because the block has an immediate use after that miss, resulting in 50% of accesses causing allocation-writes. In contrast, using a fixed allocation for the address `a` captures nearly the same number of hits in the long-term (nearly 50%) while also minimizing the allocation-writes to exactly 1.

While the above discussion proved that ideal sieving is effective in reducing the number of SSD write operations, ideal sieving is impractical. In the next two sections, we present two practical variants of SieveStore.

3.2 Discrete SieveStore (SieveStore-D)

SieveStore-D employs a discrete caching model wherein allocation and replacement occur in batches at discrete epoch boundaries. All blocks selected by SieveStore-D's sieving mechanism at the end of the i^{th} epoch are batch-allocated to the cache where they remain (i.e., no replacement) till the end of the $(i+1)^{th}$ epoch. Logically, each resident block is replaced at the end of the epoch and is replaced by the newly allocated blocks. In practice, if a block that is to be replaced at the end of an epoch is found to be allocated for the next epoch, the replacement and allocation cancel each other to eliminate unnecessary block moves.

SieveStore-D uses access counts to sieve cache allocation in the following way. All blocks whose access-count in the

Table 2: Impact of Allocation Policies (assuming oracle replacement policy)

Allocation Policy	Hits	Misses	Alloc.-writes	SSD operations	
				Read hits	Write hits + Alloc.-writes
Allocate-on-demand (AOD)	35%	65%	65%	26.25%	73.75% (=8.75% + 65%)
Write-no-allocate (WMNA)	35%	65%	48.75%	26.25%	57.5% (=8.75%+48.75%)
Ideal-selective-allocate (ISA)	35%	65%	ϵ%	26.25%	<9.75% (=8.75%+ϵ%)

i^{th} epoch exceeds a threshold (t) are selected for allocation at the end of the i^{th} epoch. Our design choice of access-count-based sieving in general, and the choices of the epoch as one day and the threshold $t = 10$ in particular, flow directly from our observation (O1) that 99% of blocks have low (fewer than 10/day) access counts. One may think there is a tension between our claim that the top 1% popular block set is dynamic from day-to-day and SieveStore-D's strategy of using one day's access counts to identify the hot set for the next day (which implies persistence of the hot-set). There is no real contradiction between those claims; while popular sets do drift from day-to-day (with the hot set drifting significantly with increasing time separation), there is significant overlap in successive days.

There are two potential problems that sieving must address. First, SieveStore-D must maintain access-counts for all blocks – even the blocks that are not cache resident. SieveStore-D logs all accesses for offline analysis. The analysis requires simple, per-key reductions to gather counts of all the addresses that are logged to SieveStore-D node's local storage (not the SSD-based disk-cache). Such per-key-reduction can be efficiently implemented by using a map-reduction-like structure where (1) each access is logged as a $< address, 1 >$ tuple to one of R files where the file is selected by a hash-function on the address, and (2) each of the R files are sorted, and (3) contiguous n-long "runs" of the same address are counted and emitted as a $< address, n >$ tuple. Further, such per-key reductions may be periodically performed in an incremental way to reduce the size of the logs. At the end of the epoch, tuples with n value greater than the threshold, are allocated for the next epoch.

Second, because the blocks are allocated at the end of each discrete epoch, the bulk data movement required at the end of each epoch could be a bottleneck. In practice, the movement of allocated data can be staggered by moving data when the periodic analysis of access logs identify the high-access count blocks. We show later in Section 5 that the number of blocks moved is insignificant compared to the number of accesses ($\leq 0.5\%$), and that there are significant periods of slack bandwidth in the SSD where such moves may be scheduled. Consequently, SieveStore-D moves can avoid creating a burst of heavy traffic.

3.3 Continuous SieveStore (SieveStore-C)

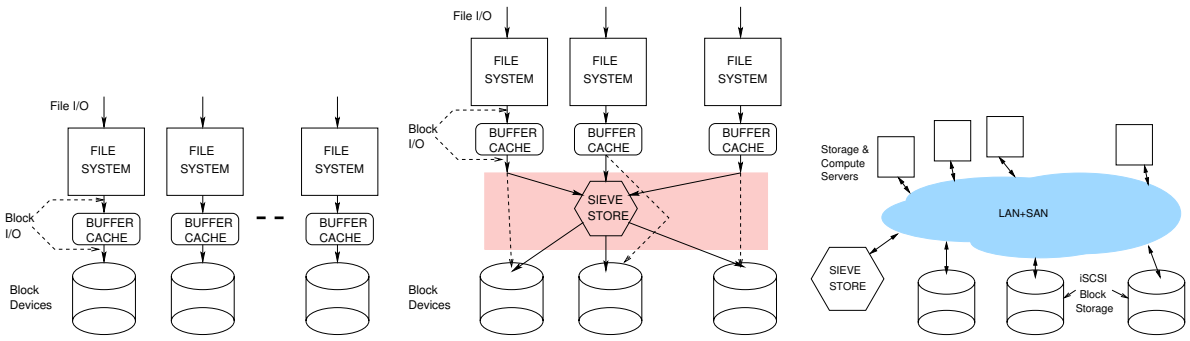
SieveStore-D's discrete allocation model limits the rate at which the system can adapt to changes in the hot set. To address that problem, we develop another variant called SieveStore-C. SieveStore-C's sieving is performed in an online fashion where each access is first checked if it is a hit or a miss. If it is a miss, it is further checked if it qualifies for allocation (i.e., sieving). If it does, the block is allocated. If not, the block is accessed directly from the underlying storage ensemble.

SieveStore-C uses hysteresis-based lazy allocation to ensure that only the n^{th} miss of a block in a recent window of time (W) results in an allocation. Such lazy allocation fundamentally requires us to maintain metastate for blocks that are accessed even if the block is not resident in the cache because block-miss-counts must be maintained. Unlike in SieveStore-D's case where the metastate was maintained in files and was never on any access critical path, SieveStore-C's continuous allocation model poses another key challenge: the metastate must be looked up on each miss to determine if the block is to be allocated. Thus the metastate must be memory resident. Unfortunately, because of the large number of unique blocks that are accessed, even assuming a small amount of state per accessed block results in state explosion that makes perfect state tracking infeasible (at least in a cost-effective way).

To address the state explosion problem, SieveStore-C uses a two-tier sieve where the first tier maintains imprecise (potentially aliased) miss-counts and the second tier maintains accurate miss-counts. At the first tier, SieveStore-C uses an *imprecise miss count table (IMCT)* of fixed size. Because the space of block addresses is significantly larger than the size of the IMCT table, the mapping from blocks to entries in the table is many-to-one. Such many-to-one mapping may be prone to aliasing, which results in access counts being inaccurate. Indeed, we found aliasing to be a significant problem because too many blocks with low-reuse were found to be piggy-backing on the miss-counts of more popular blocks and receiving undeserved cache allocations. Such cache allocations cause pollution, and allocation-writes; symptoms that single-tier sieving was not effective.

To address ineffective sieving at the imprecise layer, we employ an additional perfect Miss Count Table (MCT) which is implemented as a hash-table. With the IMCT/MCT two-tier sieve, only blocks that see a minimum number of misses in the IMCT (say threshold t_1) make it past the IMCT to the MCT. However, because IMCT is prone to aliasing, SieveStore-C further requires each block that satisfies the IMCT threshold to undergo a precise MCT threshold check. As part of the MCT check, the block has to further see an additional minimum number of misses (say threshold t_2). The two tier approach worked well in practice because the IMCT reduced the amount of perfect metastate that must be tracked and the MCT reduced the number of low-reuse blocks that are allocated cache space. We experimentally tuned t_1 and t_2 to be 9 and 4, respectively. For the traces, our implementation of IMCT and MCT occupied about 8GB of memory.

One note on the implementation of IMCT and MCT. Logically, the IMCT and MCT track the number of misses over the past W time units (say hours). However, since keeping miss counts for every time slice is impractical, we discretize the time window into k subwindows of W/k hours each. The implementation uses k counters to track the misses in each subwindow and a counter to track the last time the counters were updated. If during a miss, the current time window is larger than the last-updated counter by k or more, then



(a) Base storage-ensemble (b) Logical organization (c) Physical organization

Figure 4: SieveStore Organization

all counters are inferred to be stale and zeroed out. Periodically we prune the MCT to eliminate stale blocks. We experimentally tuned the parameters to set W to 8 hours, with k set to 4 (i.e., four subwindows of 2 hours each).

Implementation.

Both SieveStore variants may be implemented as a transparent caching appliance that can easily be deployed in existing storage-area-networks to cover a larger storage ensemble. Though the logical organization corresponds to the illustration shown in Figure 4(b) (a central cache with bypass paths), the physical realization is shown in Figure 4(c). Figure 4 assumes iSCSI block protocol and hence shows a common LAN/SAN for illustration purposes only. In practice, any connection fabric (Fibre Channel) or protocol may be used. Even if iSCSI-over-Ethernet is used, the network may use dedicated links/switches for performance.

One concern that SieveStore must address is the issue of the entire ensemble's I/O funneling through the SieveStore node. The issue may be subdivided into four different activities that are aggregated in SieveStore. First, requests are aggregated at the SieveStore node. Request processing is not a significant concern because (1) request processing is entirely in memory (e.g., cache-hit determination and sieving) except for hits and allocation-writes, and (2) the entire week's request traffic amounts to approximately 434 million multi-block requests, which is dwarfed by the amount of traffic that hits in the disk-cache. Second, the issue of handling the I/O requests that require SSD operations (hits and allocation-writes) is discussed later in Section 5.2. Third, the issue of SSD-wearout because SieveStore aggregates write-hot blocks in the SSD is also shown not to be a significant concern later in Section 5.1. Fourth, there is concern that the SieveStore node could become a network bottleneck. There are two sources of network traffic; SSD hits wherein blocks are served *from* the SieveStore node and the allocated-misses wherein blocks are copied *to* the SieveStore node. A simple worst case analysis reveals that servicing hits is well within the network bandwidth of a reasonably configured node with four Gigabit Ethernet links. Even the maximum SSD access throughput (100% sequential reads, 250MB/s) accounts for approximately 50% of the network bandwidth. In practice, the SSD accesses are neither sequential nor all-writes. Further, the SSD load is considerably lower than 100% of an SSD's bandwidth, as we show later in Section 5.2. The second source of network traffic (allocation data) is negligible because sieving is effective in reducing the number of allocation writes, as shown later in Section 5.1.

4. EXPERIMENTAL METHODOLOGY

Trace analysis, as well as analysis of the ideal case and SieveStore-D involved access-counting and was performed using simple scripts and mapreductions. (Recall, there is no need for cache-simulation for SieveStore-D because allocation/deallocation are only at epoch boundaries.) For SieveStore-C, we developed a trace-based simulator that includes the data-structures for sieving (IMCT and MCT) and for the metastate of a fully-associative, 16GB cache with LRU replacement (tags, LRU stack information). Note, LRU replacement was common for all the continuous configurations (i.e., SieveStore-C, AOD, and WMNA; but not SieveStore-D) irrespective of allocation policies.

The cache simulations operate on the trace and faithfully model the cache operation including allocation-writes. Because allocation requests can occur only after the data has been fetched from the underlying storage, each allocation request to a block was assumed to start at the time that the corresponding request in the original trace completed (as reported in the trace). We used linear interpolation to infer completion times for individual blocks in cases of large, multi-block requests.

Because SieveStore funnels all of the ensemble's hot blocks to a single node, it is important to ascertain that the IOPS and bandwidth requirements can both be satisfied at that node. At a high-level, the increased IOPS/bandwidth requirements of SieveStore is helped by the high IOPS and high bandwidth provided by SSDs. However, there may be peak bursts where the requests of 13 individual servers saturate even the high I/O throughput of the SSD, thus requiring more SSDs for parallelism. To catch such peaks, if any, our cache simulator exports the required IOPS and required bandwidth (MB/s) (separately for reads and writes) in each minute of the trace.

We assume SieveStore uses an SSD for caching blocks. The SSD's parameters are modeled after Intel's X25-E Extreme SATA SSD [8], which achieves 35,000 random read IOPS, 3300 random write IOPS, 250MB/s sustained sequential read bandwidth and 170MB/s sustained sequential write bandwidth, assuming 4KB pages. The random bandwidth (computed from random IOPS for 4KB transfers) is 140MB/s and 13.2 MB/s which is a tighter constraint than sequential bandwidth. So we evaluate the drives needed under the tighter IOPS constraint.

To assess cost, we compute the drives needed to satisfy the IOPS requirements of our workloads. To that end, we compute a *Drive IOPS occupancy* metric for each minute in the trace. We assume that each 4KB read I/O occu-

Table 3: Allocation policies

Key	Allocation Policy	When is a block allocated?
AOD	Allocate-on-demand	On a miss
WMNA	Write-no-allocate	On a read-miss
SieveStore-D	Access count-based, discrete batch-allocation with threshold=n	Blocks that are accessed at least n times in an epoch enter the cache at the end of that epoch
SieveStore-C	Lazy allocation, threshold=n, window=W	On the n^{th} miss in the previous time window.

pies the drive for $1/35000^{th}$ of a second (because read IOPS is 35000/s) and each 4KB write I/O occupies the drive for $1/3300^{th}$ of a second (because write IOPS is 3300/s). This is a rather simple model that ignores queueing. However, we will show that, with SieveStore variants, the drives are typically operating at a significantly lower load point than supported by the SSD. Consequently, queueing is unlikely to be a significant problem. The number of drives needed each minute is computed as the ceiling of the drive occupancy of all requests for that minute. Because the SSD performance parameters are specified for 4KB units, we assumed 4KB accesses. However, because there were a small number (6%) of accesses that were not 4KB-aligned, we conservatively assessed the same cost for a sub-4KB I/O as that of a 4KB I/O. (We use this conservative approximation only for assessing drive-needs. All other numbers count I/O blocks/accesses assuming 512-byte blocks for accuracy.)

Finally, we arrive at the required number of drives for the whole trace under various coverage assumptions. For 100% coverage, the number of drives needed is the maximum of the number of drives for each minute of the trace, which is a worst-case design. We also consider reducing coverage in cases where a large decrease in the number of drives may be obtained by sacrificing a little coverage.

5. RESULTS

There are three primary conclusions from our experiments.

1. We show that sieving fundamentally enables ensemble-level caching to achieve higher performance. SieveStore-D (SieveStore-C) improves the number of hits by 35% (50%) over the best unsieved ensemble-level caching, while also reducing the number of allocation-writes by over two orders of magnitude. (Section 5.1.)

2. With a single SSD, SieveStore-D (SieveStore-C) can satisfy the IOPS and bandwidth requirement of the ensemble with 100% (99.9%) time coverage. (Section 5.2.)

3. SieveStore's approach of ensemble-level caching is a superior cost-performance point compared to ideal private (per-server) caching because it can achieve higher performance at the same cost OR lower cost for the same performance. (Section 5.3.)

5.1 Sieved vs. Unsieved Ensemble-level caches

We compare the two SieveStore variants with the unsieved allocation policies for ensemble-level caches (quadrant I vs. quadrant II in Figure 1). Figure 5 plots the total number of accesses captured by ensemble-level caches (normalized to the total number of accesses, Y-axis) allocated on each day of the trace (X-axis) by each allocation technique (bars in each group). The accesses caught by the ideal SieveStore that captures the top 1% of blocks each day is the left-most bar. In addition to the ideal case, we include the sieve-based techniques SieveStore-C and SieveStore-D as well as two randomized allocation policies (RandSeive-BlkD and RandSieve-C) to illustrate the fact that SieveStore truly identifies and captures hot blocks (beyond what random sampling would achieve). The RandSieve-BlkD variant allocates a randomly chosen 1% of the *blocks* accessed each day and batch-allocates them to the cache for the next day. RandSieve-C is a continuous variant that randomly allocates 1% of all misses. We also include the two unsieved allocation policies (AOD, WMNA). For the unsieved ensemble caches, we also compare against a cache that is twice the size (32GB) of the SieveStore variants to account for the possibility that the additional hits from sieving may be captured by using the additional DRAM/storage needed to hold the sieve data-structures/logs. Like the 16GB cache, the 32GB cache is also fully-associative with LRU replacement. The accesses of the non-ideal allocation mechanisms also show the breakdown of reads and writes.

On average, SieveStore-D and SieveStore-C[1] are within 14% and 4% of the ideal case. Both SieveStore variants are, at worst, within 16.7% and 6.6% of the ideal case, with the exception of SieveStore-D on days 1 and 2. On average, SieveStore-D and SieveStore-C capture 35% and 50% more accesses (hits) than the best of AOD and WMNA.

SieveStore-D shows zero accesses on day 1 because it needs one day's access logs to bootstrap its sieve. Consequently, the average excludes the first day. The exceptional behavior of SieveStore-D on day 2 is because block popularity on day 1 is significantly more skewed than the remainder of the days because of our calendar-day-based analysis (Section 2); only 0.04% of accessed blocks have 10 or more accesses (Figure 2(a)). Consequently, SieveStore-D which allocates blocks only if their access count exceeds 10, allocates a significantly smaller set of blocks at the end of day 1. In spite of the smaller number of blocks, SieveStore-D captures more than half of accesses on day 2. In contrast, SieveStore-C can react on a continuous basis and achieves a significantly higher hit-ratio on day 2. On days 3 , 4 and 8, SieveStore-C achieves a marginally higher hit-ratio than the (day-by-day) ideal case.

Neither of the two random sieving variants (bars A and D in Figure 5) performs well, achieving only marginally higher hitrates than the unsieved allocation policies. The extremely poor hit ratio of RandSieve-BlkD is to be expected because of the low likelihood of randomly selecting the hot blocks. However, the poor performance of RandSieve-C requires some explanation. One may think that RandSieve-C should work well in principle because popular blocks occur

[1]The ideal configuration uses day-by-day discrete allocation, and hence is an upper-bound for SieveStore-D, but not for SieveStore-C because SieveStore-C can continuously change the blocks it holds.

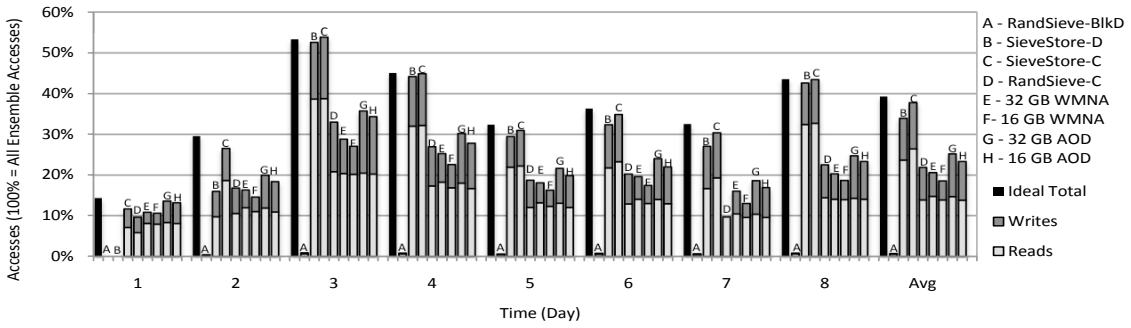


Figure 5: Sieving Effectiveness: Accesses Captured

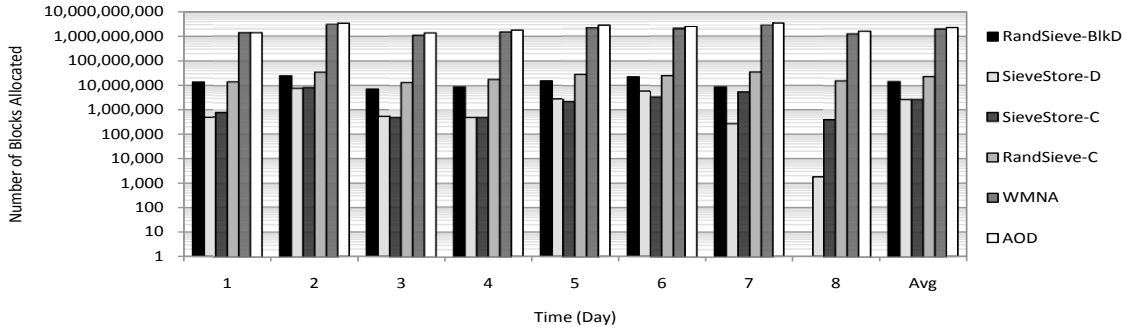


Figure 6: Sieving Effectiveness: Allocation Writes

repeatedly in the miss-stream and are thus more likely to be sampled. However, because the fraction of accesses that come from low-reuse blocks is significant (approximately 60%, on average), random sieving will result in 60% of allocations being those of low-reuse blocks. Such spurious allocations explain the lower hit ratio.

Finally, while larger disk-caches help the unsieved ensemble caches (bars labeled E,F in Figure 5), there remains a significant gap between the hit ratios of SieveStore variants and AOD/WMNA. In the rest of the paper, we use the 32GB variant of AOD/WMNA as they offer better hit ratios.

Allocation writes.

Figure 6 plots the number of allocation-writes (Y-axis, log-scale) for each day of the trace (X-axis) for our allocation mechanisms (bars). The number of allocation-writes required by SieveStore-D and SieveStore-C is a more than two orders of magnitude smaller than those of AOD and WMNA. (Though not shown in Figure 6 the number of allocation-writes for 16GB caches with AOD and WMNA were qualitatively similar.) The random sampling sieves do reduce the number of allocation writes significantly compared to AOD and WMNA. However, they are almost an order of magnitude (8.5X) worse than the SieveStore variants, on average.

SSD accesses.

In order to account for the total number of accesses to the SSD, and also to put the magnitude of the allocation-writes in the context of other accesses, Figure 7 combines the information from both Figure 5 and Figure 6. The Y-axis shows all SSD operations (at 512-byte block granularity). Each bar has three components: reads (hits), writes (hits) and allocation-writes. Three key observations can be made from Figure 7.

First, the allocation-writes bar reveals the importance of sieving. Without sieving, the allocation-writes constitute the dominant fraction of all SSD accesses. Combined with the fact that writes are slower in SSDs, such allocation-writes can have a crippling impact. For SieveStore-C and SieveStore-D, the bars for the *allocation-writes are included* as a thin (nearly-invisible at scale) bar.

Recall, from Section 1 that we do not differentiate between reads and writes in SieveStore. One may think that caching write-hot blocks can (1) reduce lifetime for SSDs and (2) impose an opportunity cost by preventing other read-hot blocks which can benefit from significantly higher IOPS. However, because the number of writes for SieveStore (= write hits + allocation-writes) never exceeds 500 million writes of 512 bytes each on any given day, and because the X25-E SSD drive can endure 1 petabyte (10^{15}B) of writes [8], the disk's endurance is over 10 years = $(10^{15}/(5 \times 10^8 \times 512 \times 365))$. Further, there is no opportunity cost in the sense that write-hot blocks preclude read-hot blocks because a 16GB disk can contain all the top 1% popular blocks with room to spare. Finally, there is the "pull" factor that makes it attractive to cache write-hot blocks; SSD write IOPS is an order of magnitude higher than the write IOPs of enterprise HDDs.

Sensitivity.

Due to lack of space, we briefly summarize the results of sensitivity analysis without providing detailed results. We

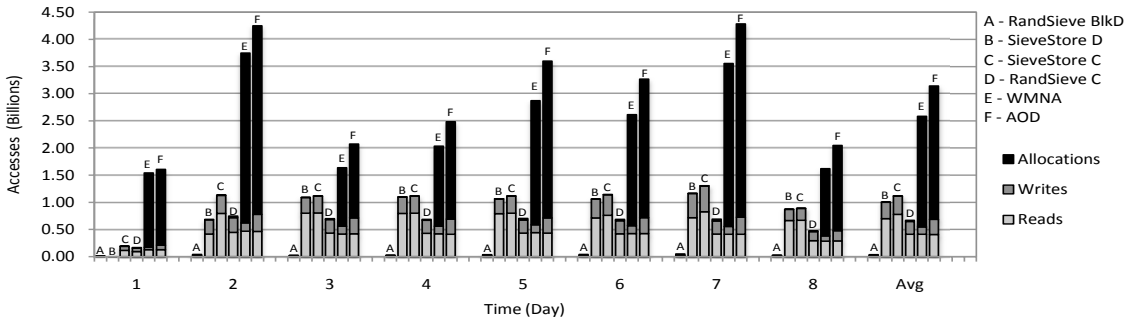


Figure 7: Total SSD Accesses

found that SieveStore is sensitive to variations in the threshold in only one direction. If the threshold is too low (e.g., below 8 for SieveStore-D), we have inadequate sieving and poor performance. But if the threshold is varied in the high range (e.g., 8-20 for SieveStore-D), the hit-rate does not vary significantly. Similarly, SieveStore was relatively insensitive to significant variations in epoch/window lengths although we observed that lengths shorter than 8 hours caused some performance degradation.

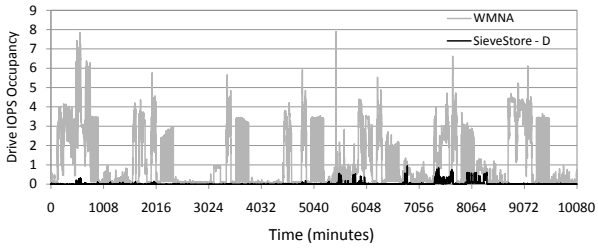
5.2 Assessing costs of SieveStore

Recall, the methodology for computing the number of drives uses IOPS rating of the Intel X25-E SSD drive and compares that to the IOPS requirement in each of the 10080 ($=7 \times 24 \times 60$) minutes of the trace.

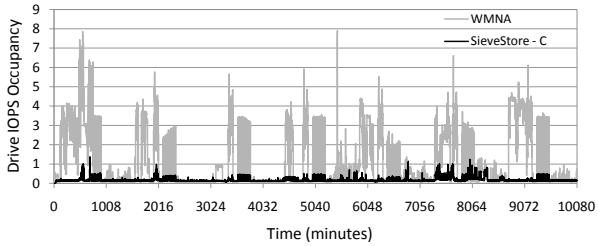
Figure 8 plots two graphs showing the drive IOPS occupancy (Y-axis) against each minute in the trace (X-axis, in chronological order). Individually, Figure 8(a) and Figure 8(b) compare WMNA allocation policy with SieveStore-D and SieveStore-C, respectively. AOD is omitted because it is strictly worse than WMNA. The cost of the large number of allocation-writes of the WMNA policy is manifested in the regions where drive IOPS occupancy peaks (gray curve in both Figure 8(a) and Figure 8(b)). In contrast, the occupancy of SieveStore variants is significantly lower and mostly under 1. SieveStore-D never requires more than one disk with the caveat that SieveStore-D assumes that batch allocation can be done during periods of low disk activity. Note, the assumption is reasonable because there is significant downtime in SSD activity. SieveStore-C, which explicitly accounts for all allocation-writes, maintains its drive IOPS occupancy under 1 more than 99.9% of the time. The requirement goes up to two drives for only 9 minutes out of a total of 10080 minutes of the trace.

Figure 9 plots the number of drives needed (Y-axis) for each minute (X-axis) of the trace for each allocation policy (curves in the figure). The Y-axis values in Figure 9 are simply the values of the curves in Figure 8 rounded up to the next integer (i.e., ceiling function). Note, the minutes are not in chronological order but in increasing order of drive-requirements. Note the small peak at the right for SieveStore-C that represents the 9 minutes where we need two drives. The WMNA policy would require 7 drives for 99.9% coverage. Even after diluting the coverage requirement to 90% WMNA would require 4 drives.

At a higher level, the result that a large fraction of the ensemble's accesses can be satisfied comfortably by a single SSD in 99.9% of the cases is not surprising because (1) the



(a) SieveStore-D vs. WMNA



(b) SieveStore-C vs. WMNA

Figure 8: Drive IOPS occupancy

IOPS offered by the SSD is two orders of magnitude higher for reads and one order of magnitude higher for writes when compared to HDDs, and (2) the probability that a large number of servers across the ensemble will experience a correlated burst of disk activity at the same time is low [14].

5.3 Ensemble vs. per-server caching

In this section, we extend the comparison to include per-server disk-caches (quadrant III and quadrant IV in Figure 1). To strengthen our argument that ensemble-level caching, in general, and SieveStore, in particular, are superior in cost-performance, we compare the cost/performance of SieveStore-C and SieveStore-D to that of two ideal per-server caching configurations. First, under the unreasonable, but conservative, assumption that SSD capacity is elastic (i.e., arbitrarily small SSDs can be built without changing the cost per byte), we compare against a per-server configuration in which each server caches the top 1% of its accessed blocks. Such a configuration is effectively an iso-capacity (and hence iso-cost, by our elasticity assumption) config-

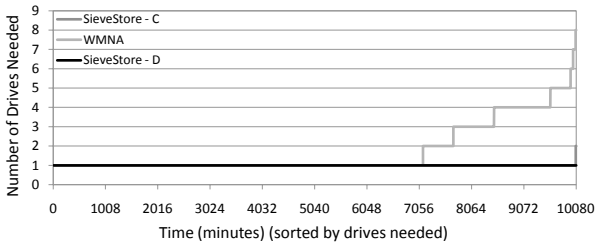


Figure 9: Number of drives needed

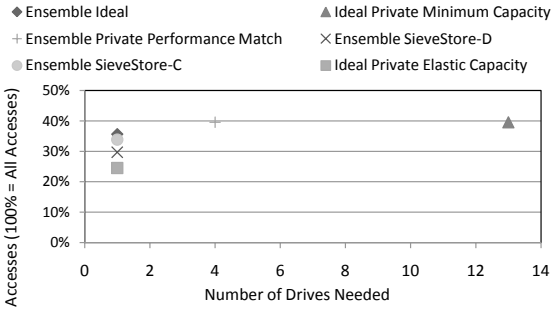


Figure 10: Cost-Performance Tradeoffs

uration with a proportional, static partition of the cache capacity among servers.

Second, under a more reasonable assumption that there is a minimum size for SSDs (i.e., non-elastic drives, we assume 16GB), the per-server caching incurs the cost of 13 SSDs. On the other hand, each server is now able to cache more data. Again, we conservatively assume that the per-server caches are filled with as many of the most popular blocks as the capacity allows.

Figure 10 compares the cost-benefit tradeoffs for the various ensemble caching configurations (ideal and SieveStore) and the two ideal per-server caching configurations. We use the number of SSDs (X-axis) as a proxy for cost and the fraction of ensemble accesses (Y-axis) that hit in the disk cache as a measure of performance.

First, focusing purely on the ideal cases with elastic SSDs, we observe that the Ideal ensemble version which caches the top 1% of blocks is significantly better than the Ideal-private configuration with elastic drives which caches the top 1% of blocks of each server. Even though the cost of the two configurations are the same, the ensemble version captures 45% more accesses than the per-server version.

Second, we compare the ideal ensemble version with ideal per-server configuration with non-elastic drives. In that case, we assume that each server has the capacity to cache as much as 1% of the entire ensemble's data, which will be a larger fraction of the server's data. This configuration increases the cost by a factor of 13X (1300%) since it needs one SSD drive per server. However, the accesses captured goes up by only 11%. This is not surprising because the benefits of increasing cache capacity beyond the 1% most popular blocks are minimal because of low-reuse (Section 2). To emphasize this point, we include an ensemble-level ideal configuration that captures the same number of accesses as the per-server configuration (labeled as Iso-performance in Figure 10). Such an ensemble configuration requires fewer SSD drives than the per-server approach (i.e., lower cost for the same benefit). More importantly, it illustrates the fact that caching more data beyond the top 1% is not cost-effective even at the ensemble-level because the cost increase (4X) is not commensurate with the benefits (11%).

Finally, SieveStore-D and SieveStore-C which are both practical, capture 21% and 37% more accesses than the ideal (impractical) iso-cost, per-server caching configuration. Thus *practical* variants of our ensemble-level caching perform better than an *ideal version* of the per-server caching technique at the same cost.

6. RELATED WORK

The advantage of dynamically sharing resources in the specific context of data-center ensembles have been widely-studied. Various researchers have described the benefits of managing power at the ensemble level [16, 5]. More recently, Lim *et.al.* proposed using a shared pool of memory that could be dynamically allocated to computers in an ensemble to alleviate the reducing memory-capacity per core [13]. Narayanan *et.al.* have shown that treating storage bandwidth of an ensemble as a pooled resource enables efficient handling of peak throughput bursts [14]. SieveStore applies similiar ensemble-level management in another context. Note, the ensemble-level approach cannot blindly be applied to other resources. Just as the observation that not all nodes in the ensemble were operating at peak power (or utilizing large amounts of memory, or placing peak demand for I/O bandwidth) at the same time was central to motivating the ensemble-level power-management (or memory-pooling, or storage-bandwidth pooling) techniques, our observations that there is extreme dynamic popularity skew in storage-ensemble block-access traces, and that sieving can filter allocation-writes, led to the SieveStore design.

There is a large body of work on replacement algorithms that achieve better hit ratios than LRU replacement by handling special cases (e.g., loop/streaming accesses) separately [6, 17], and by considering frequency in addition to recency [12]). As shown in Section 3.1, our sieving is fundamentally different in its ability to reduce allocation-writes. In the context of microprocessor caches, various static and dynamic bypassing policies have been proposed to avoid allocating cache blocks that have little reuse/value [20, 9, 18]. Because the allocation-write problem is not significant in microarchitectural SRAM caches, such bypass techniques have focused on improving hits without aiming to minimize allocation writes. In contrast, our policies aim to minimize allocation writes (while also improving hits). As shown in Section 3.1, maximizing hits need not necessarily reduce allocation writes, even when using selective allocation. Finally, Behar *et.al.* use a sampling-based selective allocation for improved tracecache performance [1]. As discussed earlier in Section 5, applying randomized sampling in ensemble-level disk caches performs poorly compared to SieveStore variants.

Prior work has examined the use of flash-based memories as extensions of the VM system (a fast swap space) [19], and in customized architectures [3]. Our work is orthogonal. We emphasize that though our work uses SSD as one possible caching mechanism, our primary contribution is the use of ensemble-level disk caching based on sieving and is independent of underlying storage medium. Indeed the storage medium could be phase-change memory. Battery-backed

DRAM could also be used. Although DRAM write-accesses are as fast as reads, sieving is still valuable because of other benefits such as improved memory bandwidth (by avoiding allocation writes) and network bandwidth (by avoiding allocation traffic). Kgil *et.al.* have proposed a flash-based per-server buffer cache [11, 10]. Recall from Section 5.3, sieve-based, ensemble-level disk-caches are superior to per-server disk-caches in cost and performance.

7. DISCUSSION: SCALING AND TUNING

While our results showed that a 16GB SieveStore was adequate for the ensemble of 13 servers, the more general question of provisioning SieveStore for larger ensembles with different capacity, IOPS, and bandwidth requirements must also be considered. If SieveStore's limited capacity is inadequate to hold all the blocks that make it past the sieve (an indicator of underprovisioning), the glut of blocks being allocated to the cache could result in cache pollution and/or degraded performance. The above problem has two possible solutions. First, multiple SieveStore nodes may be added by exploiting parallelism (1) across groups of servers, wherein smaller ensembles would each be assigned a SieveStore node, or (2) across block addresses, wherein block addresses would be striped/partitioned across the two SieveStore nodes. Second, if adding SieveStore nodes is not an option, adaptively raising the thresholds in SieveStore-D and SieveStore-C may increase selectivity and gracefully handle SieveStore underprovisioning. We leave a study of scaling and adaptive threshold tuning for future work.

8. CONCLUSIONS

The adoption of SSDs to accelerate/filter accesses to traditional magnetic hard disk-drive based storage is attractive from a power/performance point of view. But cost-effectiveness remains a serious impediment because of the high cost-per-byte of SSD capacity [15]. We propose SieveStore – a highly selective, ensemble-level, disk-cache that filters a significant fraction of disk accesses of a server ensemble with a small, shared SSD. SieveStore is driven by the observations that (1) there is extreme popularity skew among blocks that are accessed, and (2) that the popular set is dynamic. While the above observations directly make the case for ensemble-level caching, conventional cache allocation policies incur a high overhead because of a large number of allocation-writes caused by misses. We show that selective allocation to avoid such overhead – sieving – is necessary. We describe two practical SieveStore variants – SieveStore-D and SieveStore-C. Trace-based evaluations reveal that SieveStore's twin strategies of ensemble-level disk-caching with sieving is a superior cost-performance point compared to per-server caches and unsieved ensemble-level caches.

Acknowledgments.

We thank the anonymous reviewers for their feedback. We thank T. N. Vijaykumar for comments on an early draft of this paper. This work is supported in part by National Science Foundation (Grant no. CCF-0621457).

9. REFERENCES

[1] M. Behar, A. Mendelson, and A. Kolodny. Trace cache sampling filter. In *Proc. of the 14th International Conference on Parallel Architectures and Compilation Techniques*, pages 255–266, 2005.

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

[3] A. M. Caulfield, L. M. Grupp, and S. Swanson. Gordon: using flash memory to build fast, power-efficient clusters for data-intensive applications. *SIGPLAN Not.*, 44(3):217–228, 2009.

[4] C. Dirik and B. Jacob. The performance of pc solid-state disks (ssds) as a function of bandwidth, concurrency, device architecture, and system organization. In *ISCA '09: Proc. of the 36th annual international symposium on Computer architecture*, pages 279–289, 2009.

[5] X. Fan, W.-D. Weber, and L. A. Barroso. Power provisioning for a warehouse-sized computer. In *ISCA '07: Proc. of the 34th annual international symposium on Computer architecture*, pages 13–23, 2007.

[6] G. Glass and P. Cao. Adaptive page replacement based on memory reference behavior. In *Proc. of the 1997 ACM SIGMETRICS international conference on Measurement and modeling of computer systems*, pages 115–126, 1997.

[7] A. Gupta, Y. Kim, and B. Urgaonkar. Dftl: a flash translation layer employing demand-based selective caching of page-level address mappings. In *ASPLOS '09: Proc. of the 14th intl. conference on Architectural support for programming languages and operating systems*, pages 229–240, 2009.

[8] Intel. *Intel X25-E SATA Solid State Drive Datasheet*. http://download.intel.com/design/flash/nand/extreme/319984.pdf.

[9] T. L. Johnson and W.-m. W. Hwu. Run-time adaptive cache hierarchy management via reference analysis. In *Proc. of the 24th annual International Symposium on Computer architecture (ISCA '97)*, pages 315–326, 1997.

[10] T. Kgil and T. Mudge. Flashcache: a nand flash memory file cache for low power web servers. In *Proc. of the 2006 international conference on Compilers, architecture and synthesis for embedded systems*, pages 103–112, 2006.

[11] T. Kgil, D. Roberts, and T. Mudge. Improving nand flash based disk caches. In *Proc. of the 35th International Symposium on Computer Architecture*, pages 327–338, 2008.

[12] D. Lee, J. Choi, J. H. Kim, S. H. Noh, S. L. Min, Y. Cho, and C. S. Kim. Lrfu: A spectrum of policies that subsumes the least recently used and least frequently used policies. *IEEE Trans. Comput.*, 50(12):1352–1361, 2001.

[13] K. Lim, J. Chang, T. Mudge, P. Ranganathan, S. K. Reinhardt, and T. F. Wenisch. Disaggregated memory for expansion and sharing in blade servers. *SIGARCH Comput. Archit. News*, 37(3):267–278, 2009.

[14] D. Narayanan, A. Donnelly, E. Thereska, S. Elnikety, and A. I. T. Rowstron. Everest: Scaling down peak loads through i/o off-loading. In *OSDI*, pages 15–28, 2008.

[15] D. Narayanan, E. Thereska, A. Donnelly, S. Elnikety, and A. Rowstron. Migrating server storage to ssds: analysis of tradeoffs. In *EuroSys '09: Proc. of the 4th ACM European conference on Computer systems*, pages 145–158, 2009.

[16] P. Ranganathan, P. Leech, D. Irwin, and J. Chase. Ensemble-level power management for dense blade servers. *SIGARCH Comput. Archit. News*, 34(2):66–77, 2006.

[17] Y. Smaragdakis, S. Kaplan, and P. Wilson. Eelru: simple and effective adaptive page replacement. *SIGMETRICS Perform. Eval. Rev.*, 27(1):122–133, 1999.

[18] E. S. Tam, J. A. Rivers, V. Srinivasan, G. S. Tyson, and E. S. Davidson. Active management of data caches by exploiting reuse information. *IEEE Trans. Comput.*, 48(11):1244–1259, 1999.

[19] H.-W. Tseng, H.-L. Li, and C.-L. Yang. An energy-efficient virtual memory system with flash memory as the secondary storage. In *ISLPED*, pages 418–423, 2006.

[20] G. Tyson, M. Farrens, J. Matthews, and A. R. Pleszkun. A modified approach to data cache management. In *MICRO 28: Proc. of the 28th annual international symposium on Microarchitecture*, pages 93–103, 1995.