HANDS: A Heuristically Arranged Non-Backup In-line Deduplication System

Technical Report UCSC-SSRC-12-03
March 2012

A. Wildani avani@cs.ucsc.edu
E. L. Miller elm@cs.ucsc.edu
O. Rodeh orodeh@us.ibm.com

Storage Systems Research Center
Baskin School of Engineering
University of California, Santa Cruz
Santa Cruz, CA 95064
`http://www.ssrc.ucsc.edu/`

HANDS: A Heuristically Arranged Non-Backup In-line Deduplication System

Avani Wildani
UC Santa Cruz
Storage Systems Research Center
Santa Cruz, California
avani@soe.ucsc.edu

Ethan L. Miller
UC Santa Cruz
Storage Systems Research Center
Santa Cruz, California
elm@soe.ucsc.edu

Ohad Rodeh
IBM Almaden Research Center
San Jose, California
orodeh@us.ibm.com

ABSTRACT

Deduplication on is rarely used on primary storage because of the *disk bottleneck problem*, which results from the need to keep an index mapping chunks of data to hash values in memory in order to detect duplicate blocks. This index grows with the number of unique data blocks, creating a scalability problem, and at current prices the cost of additional RAM approaches the cost of the indexed disks. Thus, previously, deduplication ratios had to be over 45% to see any cost benefit.

The HANDS technique that we introduce in this paper reduces the amount of in-memory index storage required by up to 99% while still achieving between 30% and 90% of the deduplication of a full memory-resident index, making primary deduplication cost effective in workloads with a low deduplication rate. We achieve this by dynamically prefetching fingerprints from disk into memory cache according to working sets derived from access patterns. We demonstrate the effectiveness of our approach using a simple neighborhood grouping that requires only timestamp and block number, making it suitable for a wide range of storage systems without the need to modify host file systems.

1. INTRODUCTION

Though the price of storage is falling rapidly, recent data from the IDC indicates that the rate of data growth is outpacing the rate of storage growth [11]. This divide is projected to increase in the near future, even before taking into account the effect of localized natural disasters on the global storage supply, which can price available storage above what many organizations can afford at scale [9]. Analysts believe that over half of the information we currently produce does not have a permanent home [15].

To address the growing need for primary space savings, researchers such as Constantinescu [6], Jones [13], the iDedup team [23], and the DBLK team [25] have examined de-duplicating primary storage in addition to backups and archival systems. At a high level, deduplication separates data into units and then compares data against an index of units to determine if the data unit in question is already on the system. Maximizing the space savings for primary storage requires the use of in-line deduplication, where every unit is checked against the index before it is written. As a result, the index must be stored on media with very fast access time to avoid affecting the perceived performance of the primary system.

Primary deduplication is rarely used because the cost of storing this index is often prohibitive. The cost of the index scales with the amount of unique data in the storage system. Though we would like to store the entire index in memory for access speed, in any system with much more disk than memory the index cache will quickly grow too large to store in memory, causing paging to and from disk; this is known as the *disk bottleneck problem* [5].

For example, to de-duplicate a 100 TB system with a segment size of 4 KB, a deduplication system would have 2.7 billion segments. At an average of 32 bytes per segment, this would result in a massive 800 GB of memory to store the index. This is a low estimate since many systems require a smaller segment size. At $100 per terabyte of disk and $10 per gigabyte of DRAM, the $10,000 disk array requires $8,000 of memory to store the entire deduplication index. This means that if the deduplication rate for the workload is under 45%, which is more than many primary deduplication systems achieve [23], we are actually paying extra to store less data with primary deduplication.

To increase the simplicity and scalability of these systems without greatly impacting the deduplication ratio, we introduce HANDS, a scalable, in-line, chunk-based deduplication method for primary storage. HANDS derives correlated sets of blocks, or working sets, based on usage patterns and places the corresponding block hashes, or fingerprints, adjacently in the index cache so they can be accessed atomically when an element is accessed. Though we propose a method for determining working sets, our technique does not rely on any particular method of generating groupings. Additionally, the method we propose for working set identification relies only on block I/O data, which is easy to collect and interpret across manifold systems, allowing the same deduplication system to be implemented across a heterogeneous environment using minimal data.

We tested our system on data from a large, multi-user enterprise grade storage system as well as a university research server and found that loading sets of data into the index cache significantly reduces the number of accesses to the on-disk index cache that the system must make while having minimal performance impact. We use HANDS to prefetch future fingerprint lookups in memory with minimal additional disk accesses, and demonstrate in-line deduplication over primary storage with an in-memory index cache of 1% or less of the total index has a deduplication rate up to 90% of the best possible rate. We also found that only 10% of the total data blocks read by a trace need to be pulled into the memory cache at all to achieve reasonable results.

Our work provides two major contributions. First, we demonstrate a dynamic, scalable method to select a portion of the deduplication index to store in memory, drastically reducing the memory cost for primary deduplication. Second, we provide an experimental evaluation of deduplication with different methods of in-memory index management and memory footprint size, demonstrating the generality and adaptability of our technique.

The rest of this paper is laid out as follows. First, in Section 2 we discuss the current state of primary deduplication and the solutions other researchers have proposed for scalability. In Sections 3 and 4, we present an overview of our working set calculation methods and system design. Then, in Section 5, we present our experimental results and an analysis of our methodology. We conclude with a discussion of how our work could be extended to different workloads and storage systems.

2. BACKGROUND AND RELATED WORK

Much of the data in large systems, generated data in particular, has a high duplication rate [6, 19]. Some workloads, such as virtual machines, have obvious massive duplication, and these images do not get cleaned up the way localized storage does [12]. Other workloads, such as scientific computing, have little file level duplication since results are unique but have a high degree of chunk level duplication as there may be very little difference in the results files between experiments.

Scalable deduplication is known to be difficult in enterprise systems at the petabyte level and beyond [5, 10]. While some groups have presented solutions for backup and archival workloads, primary storage remains relatively unstudied. The main argument against deduplication on primary storage is the performance impact on the system caused by hash calculation and additional I/O requests [6]. Scalable primary storage has become more of a concern recently as the concept of "stale data" evolves. While organizations used to be able to identify older data to store on secondary or tertiary storage, many modern datasets are accessed haphazardly and unpredictably, creating an archival-type workload where the write-once, read-maybe assumption no longer holds [1]. Researchers such as Constantinescu [6] and Storer [24] use deduplication on primary storage in addition to backups and archival systems, but they focus on compression and security, respectively, and do not directly address the disk bottleneck problem.

There have, however, been a variety of prior attempts to address the disk-bottleneck problem. These efforts have typically been limited to backup systems, though Mandagere *et al.* demonstrate the type of trade-offs that are typically made to limit a deduplication index to available memory, mainly by increasing the chunk size [18]. Jones [13] and Srinivasan *et al.* [23] propose to improve primary deduplication performance by only duplicating chunks with spatial locality. They also use temporal locality to restrict their in-memory cache to LRU. Essentially, they limit the blocks they de-duplicate to blocks that are hot and sequential. While this may be necessary for workloads with extremely high IOPS, we show in this work that it is possible to de-duplicate every block instead of restricting ourselves to blocks which have a high probability of duplication in certain workloads. Our technique is broadly applicable and results in better space savings than iDedup.

The Extreme Binning project tackles the data bottleneck problem by noting that file similarity can be determined by comparing the IDs of a subset of chunks, allowing similar files to be grouped together in backup workloads by sub-sampling larger pieces of a stream [5]. Our approach aims for the same ends with different means. First, we assume we do not know about existing chunk to file relationships when grouping our data. Our groups are purely based on chunks. We believe this is beneficial because it allows us to proceed with less system knowledge in an environment where many accesses are not sequential. Adding groups to the file similarity metric in Extreme Binning could improve their results on primary workloads. A similar technique used by Lillibridge *et al.* addresses the disk bottleneck problem by breaking up the backup streams into very large chunks and selectively de-duplicating them against similar chunks stored in a sparse index [17]. Though their throughput is very high, they rely on the large similar blocks of data that are common in backup workloads but generally absent in multi-user primary storage.

Grouping in particular has been successfully used by some projects to improve scalability in backup systems. Both Zhu *et al.* and Efstathopoulos and Guo found that pulling in data by group membership had a significant impact on the memory requirement of the index cache [30, 10]. However, their method of group detection, relying on the spatial locality of the data stream in a backup workload, does not carry over to the random accesses of a primary deduplication workload.

Zhu *et al.* also provide a comprehensive look at data deduplication on backup workloads in addition to introducing the use of Bloom filters to improve the lookup speed for testing whether a write is a duplicate [30]. While this method is much faster than a linear search over the database, it is still not fast enough to avoid affecting performance. DEBAR improved the scalability and performance of deduplication for backup systems by aggregating a set of small I/Os into large sequential blocks after passing them through a preliminary filter [29]. Our work captures this same sequentiality through working set identification and thus does not need to rely on the backup stream.

Power and reliability both become more manageable if there is less data stored on the system [4]. In archival workloads, some data can be sent to a different, slower tier of storage, leaving a smaller subset of data to be actively managed. In archival-by-accident systems, on the other hand, there is no good way to isolate large portions of data that are unlikely enough to be accessed that they can safely be sent to tape or other slower secondary storage. Data grouping has been put forward as a means to reduce the effective data for a variety of applications [3, 27], though the grouping usually relies on domain knowledge. We use a domain agnostic grouping methodology designed for conditions when a minimal amount of trace data is available [28]. There are several extant methods for working set prediction such as C-Miner [16], which uses frequent sequence matching on block I/O data, or grouping using static, pre-labeled groups as Dorimani *et al.* does [8]. There is also a large and varied body of work on file access prediction which could be used in place of working set selection [20, 26, 2].

3. WORKING SET IDENTIFICATION

Working sets are groups of block addresses or offsets that are likely to be accessed together. Identifying working sets efficiently is a key element of our deduplication system. A good working set algorithm produces groups that are small, making them less likely to churn the cache, and have high predictive value. The algorithm must also avoid overfitting to the training data. Finally, any technique used for grouping has to be fast and low impact and produce a grouping lookup table that itself does not take too much room in memory.

For our purposes, identifying working sets bears some similarity to cache pre-fetching algorithms. Instead of trying to predict the next access based on popularity, however, we co-locate elements that are likely to be accessed together regardless of whether the elements have a high probability of being accessed at all. The important distinction between working set identification and caching is that we are not limited in the size of what we can, essentially, "pre-pre-fetch." By grouping data regardless of how it is accessed, we hope to capture associations caused by the long tail of rare accesses that still occur in observable clusters.

3.1 Selecting a Grouping Technique

Our working sets need to reflect the changing workloads of large scale systems without constant maintenance. As a result, we cannot necessarily rely on semantic qualities of the data such as directory hierarchy or filetype. We also do not want to build a system that requires regular updating to keep the groupings relevant. Instead, we apply a statistical analysis to a training set to establish initial groupings and then alter these groupings based on their observed predictive capacity. Our work builds off of *neighborhood partitioning*, a working set classification technique that looks at pairwise comparisons of accesses within a window [28]. We modified neighborhood partitioning to combine the results of several overlapping windows to return a resultant grouping.

3.2 Neighborhood Partitioning

Neighborhood partitioning is a statistical method to compare data across multiple dimensions with a definable distance metric. Since primary deduplication requires that grouping can be calculated quickly without using onerous amounts of memory, we modified neighborhood partitioning to accommodate very large enterprise storage traces with limited. By aggregating incoming accesses into neighborhoods of fixed size, neighborhood partitioning is highly scalable and able to perform in real time even in systems with high IOPS. The size of neighborhoods is determined by the memory capabilities of the system calculating the working sets, though increasing the size of the neighborhood quickly meets diminishing returns [28]. The neighborhoods in our implementation also overlap by a small number of accesses to account for small groups that straddle the arbitrary breakpoints in our neighborhood selection. The choice of overlap is based on desired group size and is independent of the data.

For each neighborhood, the partitioning steps are:

1. Collect data
2. Calculate the pairwise distance matrix
3. Detect working sets in I/O stream
4. Combine new working sets with any prior working sets

3.2.1 Data Collection

Neighborhood partitioning requires a minimum of two dimensions of data. For storage systems, this is always possible to collect without impacting system performance because of the ability to directly monitor the storage bus to get block I/O access data [22]. Block I/O tracing can also easily be done in a driver or on a block device, making this method suitable for deduplication on block devices. From block I/O traces, we can extract temporal and spatial locality data. For maximum generality, we designed this implementation to only use these two axes. Though we use only spatio-temporal locality, the algorithm is trivially extendible to other dimensions such as source file, LUN, PID, *etc.*

3.2.2 Calculating the Distance Matrix

Neighborhood partitioning depends on a pre-calculated list of distances between every pair of points, where points each represent single accesses, *i.e.*, reads or writes in a block I/O trace, and are of the form $\langle time, offset \rangle$. Though we know the amount of data accessed, adding size as a feature decreased the signal to noise ratio as most fingerprints are of similar size. For n accesses, we represent pairwise distance between every pair of accesses (p_i, p_j), as an $n \times n$ matrix with $d(p_i, p_i) = 0$. We calculate the distances in this matrix using weighted Euclidean distance, defined as

$$d(p_i, p_j) = d(p_j, p_i) = \sqrt{(t_i - t_j)^2 + oscale \times (o_i - o_j)^2}$$

where a point $p_i = (t_i, o_i)$ and the variables are t = time, o = block offset, and $oscale$ is a scaling factor.

We were most interested in recurring block offset pairs that were accessed in short succession. As a result, we also calculated an $m \times m$ matrix, where m is the number of unique block offsets in our data set. This matrix was calculated by identifying all the differences in timestamps $T = [T_1 = t_{i1} - t_{j1}, T_2 = t_{i1} - t_{j2}, T_3 = t_{i2} - t_{j1}, \ldots]$ between the two offsets o_i and o_j. Weighting timestamps led to overfitting, so we decided to treat the unweighted average of these timestamp distances as the time element in our distance calculation. Thus, the distance between two offsets is:

$$d(o_i, o_j) = \sqrt{\left(\frac{\sum_{i=1}^{|T|} T_i}{|T|}\right)^2 + oscale \times (o_i - o_j)^2}$$

We generally found this to be a more accurate matrix for our test data.

3.2.3 Identifying Working Sets

Once the distance matrix is calculated, we calculate a value for the neighborhood threshold, $\check{N}$. In the online case, $\check{N}$ must be selected *a priori* and then re-calculated once enough data has entered the system to smooth out any cyclic spikes. The amount of data that constitutes "enough" depends on what is considered a normal span of activity for the workload. In the absence of workload knowledge, we found the recalculating working sets once per day was sufficient, though in section 6 we discuss more empirical techniques for determining when to re-calculate in the absence of domain

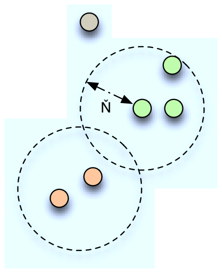


Figure 1: Each incoming access is compared to the preceding access to determine whether it falls within the threshold ($\check{N}$) to be in the same group. If it does not, a new group is formed with the incoming access.

knowledge. Once the threshold is calculated, the algorithm looks at every access in turn. The first access starts as a member of group g_1. If the next access occurs within $\check{N}$, the next access is placed into group g_1, otherwise, it is placed into a group g_2, and so on. Figure 1 illustrates a simple case.

3.2.4 Combining Neighborhoods

Once all of these neighborhood groupings are calculated we need to combine them. We do this through fuzzy set intersection between groupings and symmetric difference between groups within the groupings. So, for groupings $G_1, G_2, \ldots G_k$, the total grouping G is :

$$G = (G_i \cap G_j) \cup (G_i \Delta_g G_j) \quad \forall i,j \quad 1 \leq i,j \leq k$$

where the groupwise symmetric difference, Δ_g, is defined as every group that is not in $G_i \cap G_j$ and also shares no members with a group in $G_i \cap G_j$. For example, for two group lists $G_1 = [(x_1, x_4, x_7), (x_1, x_5), (x_8, x_7)]$ and $G_2 = [(x_1, x_3, x_7), (x_1, x_5), (x_2, x_9)]$, the resulting grouping would be $G_1 \cap G_2 = (x_1, x_5) \cup G_1 \Delta_g G_2 = (x_2, x_9)$, yielding a grouping of $[(x_1, x_5), (x_2, x_9)]$. (x_1, x_4, x_7), (x_1, x_3, x_7), and (x_8, x_7) were excluded because they share some members but not all. We choose this aggregation technique because it has a natural bias against larger groups; this in turn limits the amount of churn in our cache.

This group calculation happens in the background during periods of low activity. As accesses come in, we need to update groups to reflect a changing reality. We do this by storing a likelihood value for every group. This numerical value starts as the median intergroup distance value. If a requested fingerprint appears in multiple groups, only the group with the highest likelihood is returned. This serves to further augment the bias towards small groupings, which we have found to have a higher average likelihood. This is expected because with fewer group members, there is less chance of a group member being only loosely, or accidentally correlated with the remainder of the group, bringing the entire group likelihood down. We re-calculate the groupings and the associated likelihood values once per day, though in a system with real-time data the groups should be re-calculated when the workload shifts. For the workloads we considered, these shifts occurred about every 500,000 accesses.

Our workloads showed a working set distribution that was heavily biased towards small working sets. Figure 2 shows the working set size distributions for both data sets. Additionally, when calculating likelihood values over working sets, small sets had higher average likelihood. Given this, we arrange our working set data structure as a tiered hashtable (Figure 4). The upper tier maps working set sizes to a group of working sets while the second tier maps fingerprints to the appropriate working set.

3.3 Runtime

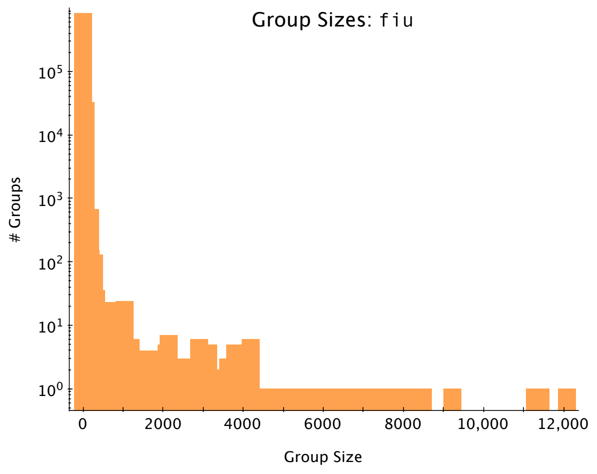
Neighborhood partitioning is especially well suited to rapidly changing usage patterns because it operates on accesses instead of offsets. When an offset occurs again in the trace, it is evaluated again, with no memory of the previous occurrence. This is also the largest disadvantage of this technique: most of the valuable information in block I/O traces lies in repeated correlations between accesses. The groups that result from neighborhood partitioning are by design myopic and will ignore long-term trend data, alleviating the problem of updated fingerprints. Neighborhood partitioning runs in $O(n)$ since it only needs to pass through each neighborhood twice: once to calculate the neighborhood threshold and again to collect the working sets. This makes it an attractive grouping mechanism for workloads with high IOPS (for example, the enterprise system we worked with can support 200,000 IOPS, though we saw far fewer in our trace), where a full $O(n^2)$ comparison is prohibitive. Additionally, we can capture groups in real time and quickly take advantage of correlations. We also can easily influence the average group size by weighting the threshold value. A main concern for us was cache churn, so we biased our grouping parameters towards smaller groups. While larger groups improve prediction immediately after groupings are calculated, over time larger groups need more re-calculation to prevent false negatives as the workload shifts. This negates their short term predictive benefit.

3.4 Validity

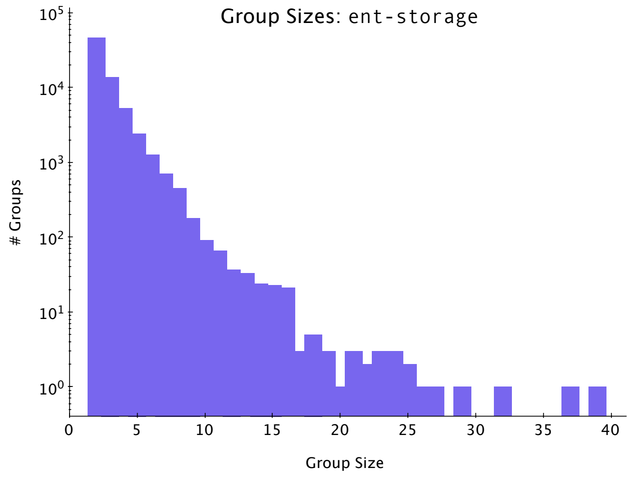
96% of group elements shared a process ID with an arbitrarily selected "first" group element in our grouped research dataset. We refer to this number as the *group purity* of this grouping. The high group purity on the research dataset indicates that this grouping does a very good job of catching interleaved groups, which prior work supports [28]. On small subsets of this data, the result is closer to 80%, likely because the groupings do not have enough occurrences to obtain high likelihood. For the enterprise data set we used, the equivalent metric showed a group purity of 100%, though the groups were generally so small that this reflects the size more than the grouping algorithm.

4. DESIGN

HANDS is a set of algorithms for content addressed inline deduplication that incorporate working sets to manage the memory footprint of the fingerprint cache. These algorithms can be interchanged modularly so, for example, another grouping technique could easily be substituted to make HANDS a better fit for a particular environment. Our high-level framework consists of three elements: the *fingerprint index* mapping fingerprints to chunks, the *index cache*, which is a subset of the fingerprint index that is kept in memory, and the *working set table* that maps fingerprints to working sets of fingerprints. The index cache is managed using LRU or LFU caching in this work, but other cache replacement algorithms can be used.



(a) fiu



(b) ent-storage

Figure 2: Group size distributions for neighborhood partitioned data sets. Both data sets have a median group size of about 3; the y axis is on a log scale.

4.1 Initialization

The first step towards deduplication is creating the working set table. Our method of compiling this table is covered in Section 3, but any grouping mechanism that results in groups that are small and likely to be co-accessed can be substituted. The next step is to allocate the two indices. The index cache is fixed size, allocated in memory, and starts out empty. We explored bootstrapping the index cache with the most frequently accessed or highest likelihood working sets, but found that neither improved our results. The working sets are then written into the on-disk fingerprint index such that every fingerprint is written adjacent to other fingerprints in its working sets.

4.2 Deduplication

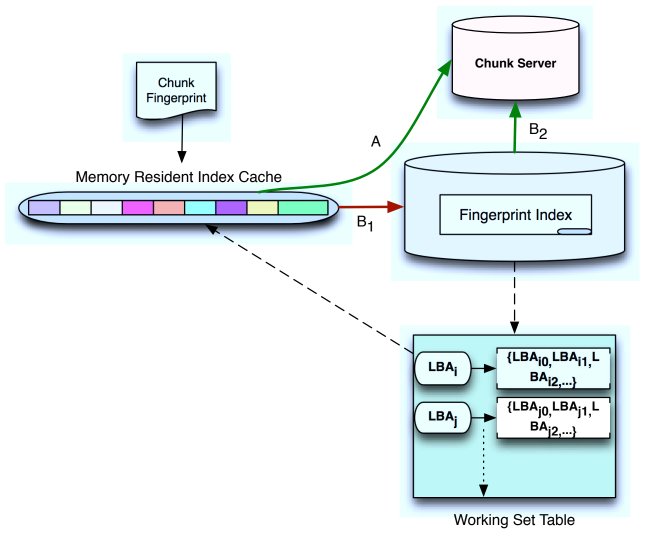
Once the working set table is established, we can begin deduplicating subsequent I/O requests. We expected most of our benefit to come from having the fingerprints for incoming write requests in cache instead of the main fingerprint index on disk. While there is some conceivable benefit to also calculating fingerprints for chunks read, that benefit is highly dependent on the underlying data retrieval mechanisms, and thus we limit the scope of this paper to write requests. We assume that all the accesses we get are post disk cache.

Figure 3 outlines the interactions of the components of our deduplication system. Our first step on receiving a write request is to calculate a hash value for the write. Any low-collision hash method works equally well; we recommend SHA-1, which for an exabyte scale system has under a 1 in 10 billion chance of hash collision in non-adversarial situations while still having a fast runtime [12, 7].

4.2.1 Duplicate Data

After the hash is computed, the next step is to compare the hash to the fingerprints in the index cache. If the hash is found in the index cache, the request is identified as a duplicate and the index cache is updated according the caching algorithm in use. We refer to this as a "cache hit."

Otherwise, on a cache miss, the request is sent to the on-disk fingerprint cache, which is arranged in tiers as shown



Chunk Fingerprint
Memory Resident Index Cache
Chunk Server
A
B_2
B_1
Fingerprint Index
LBA_i
$\{LBA_{i0}, LBA_{i1}, LBA_{i2}, ...\}$
LBA_j
$\{LBA_{j0}, LBA_{j1}, LBA_{j2}, ...\}$
Working Set Table

Figure 3: Deduplication framework. A duplicate chunk is either in the index cache (path A) or must be recovered from disk (path B). When a fingerprint enters path B, the working set for that fingerprint is pulled into the index cache from the fingerprint index, which is laid out in working set order.

in Figure 4. There is a Bloom filter across the entire index to quickly detect new data. Each tier represents groups of a given size, and they are searched from smallest to largest until a group is found. The bias towards small groups here is intentional and designed to limit cache churn.

If the fingerprint is found in the fingerprint index, in addition to serving the content the system also queries the working set table to see if the fingerprint has any known working sets. We accept the first working set match for a hash where the likelihood value is above a threshold value. This threshold is currently mean likelihood minus one standard deviation. Smaller working sets are more likely to have high likelihood, so this method should reduce the search time in very large on-disk indices. The working set returned by

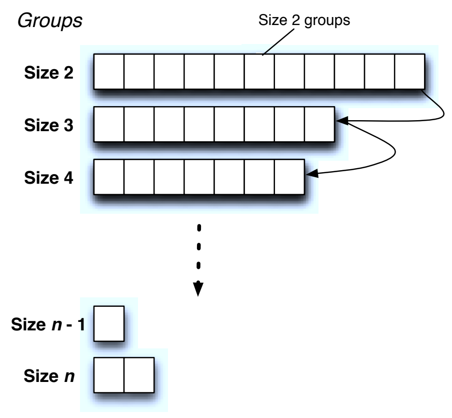


Figure 4: The fingerprint index is tiered so groups of size $n - 1$ are searched before groups of size n

the working set cache is then copied into the index cache; previous cache contents are removed based on the caching algorithm replacement policy. The hope is that this working set is a good predictor of future writes. While this results in some CPU overhead, we accept this because it is negligible compared to the penalty for the I/O.

4.2.2 New Data

If a fingerprint is not in the fingerprint index, the data is written and the hash is stored to the fingerprint index. New writes are not members of any working sets, so they can safely be placed in a temporary area without being accidently pulled into memory as a working set member. When working sets are next computed, the fingerprints will be added to working sets as appropriate. Fingerprints on disk are co-located based on group membership. Since fingerprints can be members of several groups, this could lead to duplication within the on-disk fingerprint index. We accept this because the small amount of extra disk required is inexpensive compared to the memory savings.

4.3 Design Considerations

Figure 5 shows how HANDS noticeably improves the cache hit rate. The rectangles are fingerprints, uniquely identified by color and pattern. There are three groups, identified by the letters underneath the fingerprints. An LRU cache without HANDS (Fig. 5(a)) catches most quickly repeated fingerprints and must go to disk for everything else. With HANDS (Fig. 5(b)), the cache predicts future fingerprint accesses and thus achieves considerably better cache hit rate. There is a concern that large working sets could fill the cache quickly, causing a high amount of cache churn that would push out relevant data. With the working set calculation methodology we propose, however, it is highly probable that the likelihood of a working set decreases with every additional member. This inherent bias towards smaller working sets led to less cache churn in our experimental results.

Fingerprints and blocks do not share a fixed mapping. In fact, for one of our workloads we found that over 78% of the time, consecutive accesses of the same block had different fingerprints. Thus, we group using block address but need data with actual fingerprints in order to estimate index cache performance. We found experimentally that groups tend to stay the same over time even as the fingerprints associated with blocks change [28]. Therefore, we hypothesize that our block addresses are often "unique under mutation," meaning that the usage of the data stays similar even as the actual data is modified. We discuss this shift more in Section 5.3.

5. EXPERIMENTS

To test the HANDS design, we simulated a deduplication environment where a portion of the fingerprint index is stored in memory both with and without the addition of working sets. We pass real traces with fingerprint hashes through the simulator to determine the efficacy of the working set based cache.

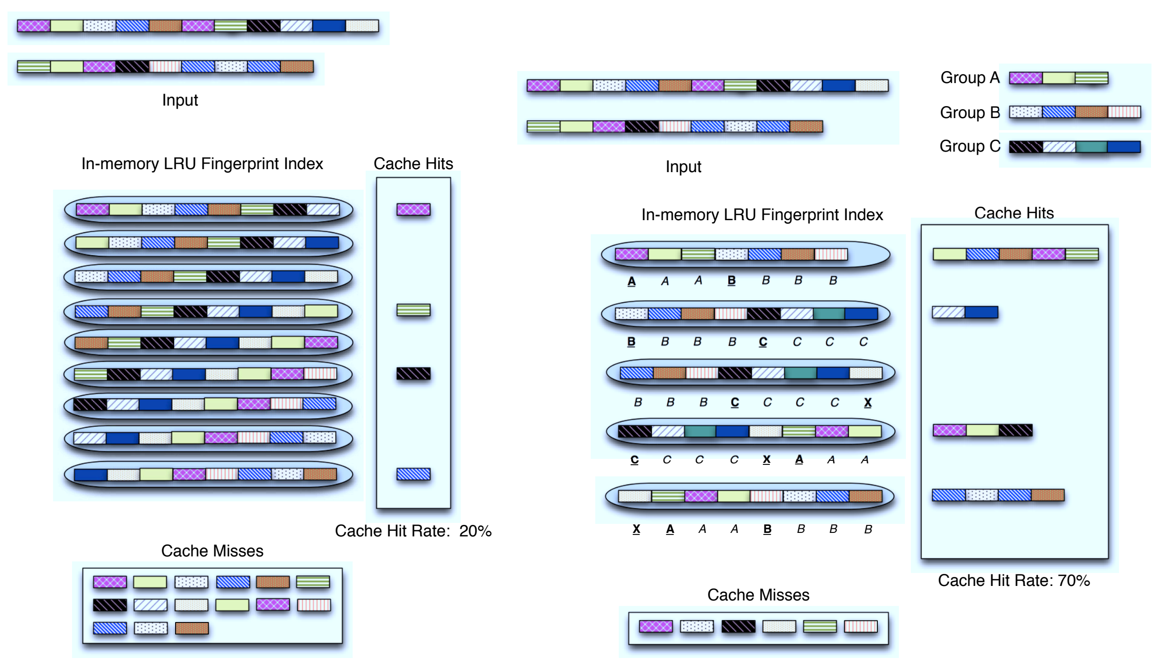
We measured the effect of working set grouping using three cache replacement algorithms: LRU, naïve LFU (LFU), and working set aware LFU (LFU-ws). Our implementation of LRU is straightforward; the oldest elements are dropped successively until there is enough room for the new element. Elements of the same age (*e.g.*, members of a working set that were pulled in together) are dropped in the order in which they appear on disk, which is preserved in the cache. Naïve LFU is an approximation of LFU that drops the least frequently used elements in the cache and is unaware of out-of-band relationships between data once the data is in the index cache. As a result, after a cache hit only the accessed element has its frequency value updated. Alternatively, when the LFU-ws algorithm records a hit to the index cache, the access frequency for every member of the accessed fingerprint's working set currently in cache is updated by .5, while the frequency of the accessed fingerprint is updated by 1 as usual. This biases the algorithm towards keeping working sets together in cache and quickly throwing out "singleton" accesses that have no working set and are not rapidly accessed.

5.1 Data

We tested our design using data sets from two real systems. Our first dataset was collected from an enterprise grade storage server at a technology company. This storage server has 120 TB of disk along with a 60 GB cache. The traces from this server are labeled `ent-storage`. Our second trace, `fiu`, is from Florida International University and traces local researchers' storage [14]. For both of these traces, we used timestamp and logical block address (LBA) to create working sets. We provide statistics about these two data sets in Table 1. For our purposes, the most important difference between these two data sets was average IOPS, meaning that over time it was harder to get predictive groups for `ent-storage` than `fiu`.

5.2 Results

We present graphs as percent of *ideal cache* versus percent of total fingerprint size. We measure ideal cache as the best a cache could do if it could always recall an element it has seen before, *i.e.*, if the cache always corresponded to the entire fingerprint index. Total fingerprint size is the sum of all of the unique fingerprints over an entire trace. We realize that this may underestimate the data size a real system needs to handle: all of our data is accessed, which is not true in many systems. However, this is the best representation that was available to us and is useful for our work because we are primarily interested in catching elements we have seen before.



(a) LRU without HANDS (b) LRU with HANDS

Figure 5: Illustration of HANDS on a toy example. The patterned rectangles correspond to fingerprints. We see that adding three working sets, or groups, improves the cache hit rate significantly by pre-fetching fingerprints into memory. In the diagram, the letters underneath the fingerprints correspond to group membership where the bolded letter is a group member that required a disk seek (a cache miss) and the italicized letters are group members that were pulled in with a bolded member.

Table 1: Workload Statistics

Trace	avg IOPS	max IOPS	R/W Ratio	# of Accesses	# Unique LBAs	# Groups	Time (h)
`fiu`	37	11897	96/4	17836701	1684407	2062671	503
`ent-storage`	75997	342142	100*	2161328	968620	70759	36

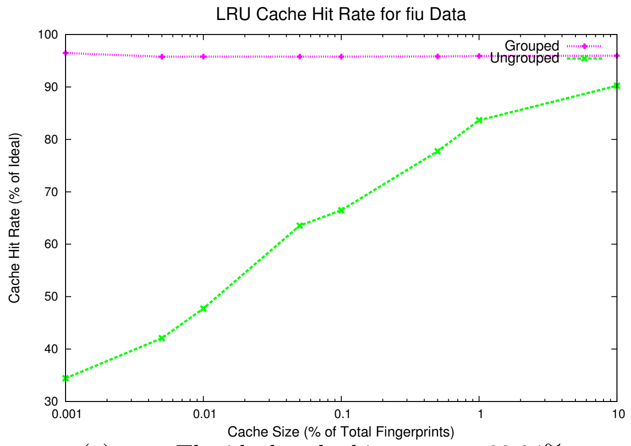
5.2.1 LRU

LRU was the best cache replacement strategy for the index cache with working sets. Figure 6(a) shows that the cache hit rate for the `fiu` dataset was almost ideal even with an index cache that was only .01% of the total fingerprint size. In contrast, without working sets the `fiu` workload had an unsurprisingly steady increase in index cache hits as the cache size increased. Adding working sets to LRU in the `fiu` case worked so well because the `fiu` workload almost entirely represents accesses by real people, and so has a very high degree of temporal locality. The `fiu` dataset also has far fewer IOPS, which helps our grouping algorithm find more predictive groups.

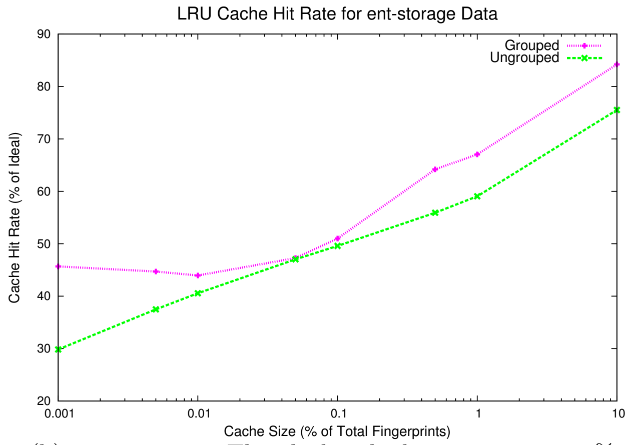
In the `ent-storage` dataset, we see a more modest but still clear improvement in cache hit rate for every cache size except about .05 after adding working sets. At that cache size, we see the beginnings of cache churn: the phenomenon where the cache is too small to hold all of the elements that are being accessed and so is passing elements in and out. We see in Figure 8(b) that our `ent-storage` cache with a cache size of .01% begins to churn at about 1.4×10^6 accesses. We believe that we can reduce this churn in the future by modifying the LRU to remove entire working sets at a time instead of just elements. We also note that the working set line never dips below the base LRU line, implying that the cache churn is not severe enough to impact the base LRU performance.

5.2.2 LFU

Figure 7 shows that for both the `fiu` and `ent-storage` datasets, there is a substantially smaller improvement in cache hit rate when grouping is added to the LFU caching algorithm as compared to LRU. This is surprising at first glance but logical when the effect that the cache replacement policies have on working sets are taken into account. The benefit of working sets for access prediction comes from a heightened probability of co-access within a working set in a given period of time. LFU evicts working set members almost as soon as they are pulled in, since they are often not used immediately. Indeed, if they were used immediately, they would be trivially easy to find and much less interesting. We attribute the slight success of LFU on the `fiu` dataset to the presence of a large number of sequential working sets. Sequential accesses were automatically filtered out of the `ent-storage` trace before it was given to us, so LFU struggles. Again, however, we see that the grouped line never falls below the ungrouped line, indicating that the working sets are not pushing enough other predictive elements out of cache to impact the base, working set free, performance.

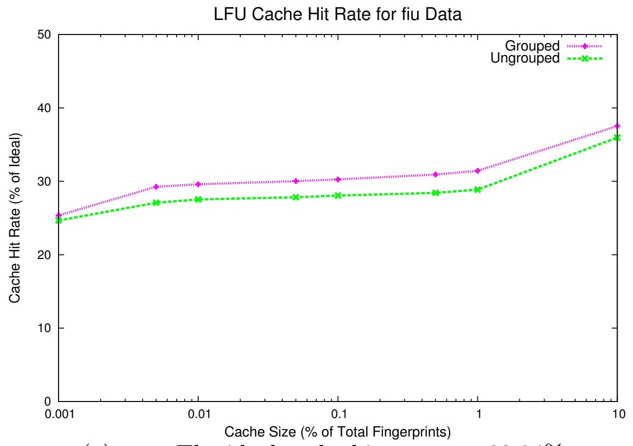


(a) **fiu**: The ideal cache hit rate was 33.94%

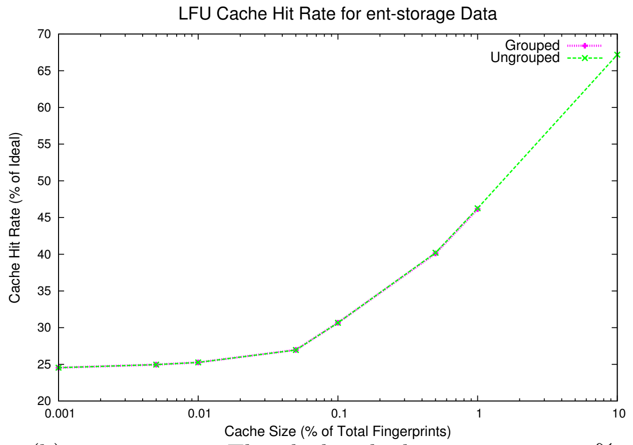


(b) `ent-storage`: The ideal cache hit rate was 35.59%

Figure 6: LRU Cache hits across cache sizes. Grouped data does consistently at least as well and often significantly better than ungrouped data.



(a) **fiu**: The ideal cache hit rate was 33.94%



(b) `ent-storage`: The ideal cache hit rate was 35.59%

Figure 7: LFU Cache hits across cache sizes. The `ent-storage` dataset sees no benefit from grouped data while the `fiu` dataset sees a modest benefit from grouping.

We also ran our simulations with LFU-ws, but saw essentially identical results compared to normal LFU, and thus do not include those in this paper. The results were likely identical because the bias provided by LFU-ws is not enough to offset the huge disadvantage of having working sets pushed out of cache immediately. While we believe it is worth investigating whether there is a balance, that is out of our scope for this work.

5.2.3 Random Working Sets

We also implemented a random working set generator to compare HANDS against. Our goal was to identify any unforeseen externalities in pulling large chunks of data into the index cache created in our deduplication system. To best mirror our observed working set distributions, we wrote a random generator sampling from a discretized Pareto distribution. The Pareto distribution is sampled from uniform using the formula:

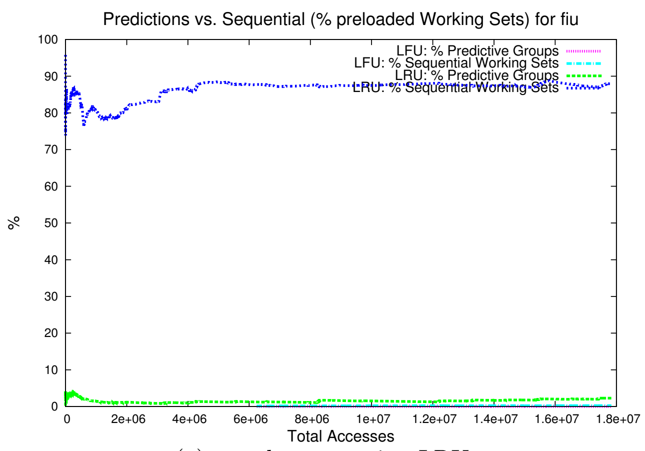
$$X = \frac{x^m}{U^{1/\alpha}}$$

Here, x^m is a parameter indicating the minimum value of X, which for groups is 1, U represents the uniform distribution between $[0,1]$, and α is a shaping parameter. We set $\alpha = 3$ to replicate our small-group bias. The resulting continuous value is then rounded to obtain an integer group size. This distribution strongly skews towards small values with few outliers, and so it is a good fit for our small working sets (Figure 2).

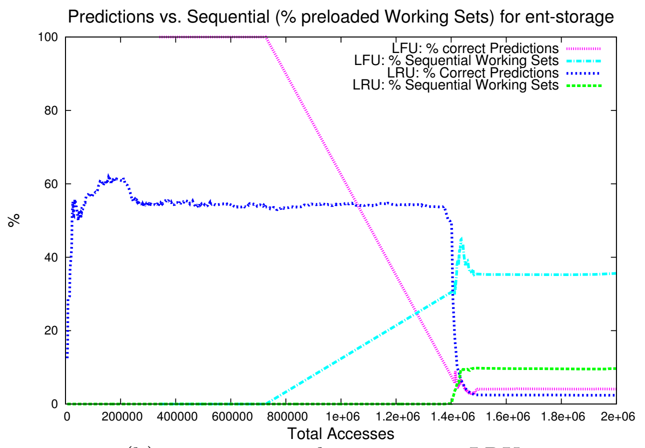
When we ran this, however, we found that the results were identical to the ungrouped results. We attribute this to the large search space of LBAs combined with the small size of groups resulting in an exceptionally low probability of successful access. Indeed, the `ent-storage` case had 0 predictive fingerprints while the `fiu` random run had under 5%. Thus, we can say with some confidence that the benefit of working sets is more than just pulling extra data into cache; the pre-computed correlation of data in working sets has value.

5.3 Analysis

Throughout this project, our technique performed as well as or better on the `fiu` dataset compared to the `ent-storage` dataset. We learned that, in the `ent-storage` dataset, sequential accesses are not part of the trace we were given because they are pre-fetched by the storage hardware. Since previous work has shown that sequential working sets are common and strong groupings, we thought that the lack of



(a) `fiu` data set using LRU



(b) `ent-storage` data set using LRU

Figure 8: For the `fiu` data set using LRU, predictive power of groups was unrelated to sequentiality. In the `ent-storage` data set using LRU, predictive power of groups fell as sequential groups increased. The percentage of predictive accesses is deceivingly low because it is calculated as a percentage of total accesses, which were an order of magnitude higher for `fiu` than `ent-storage`. The cache size was .01%

sequentiality in the `ent-storage` dataset was to blame for its relatively poor showing in both the LRU and LFU cases. However, in Figure 8(b) we see an inverse relationship in the `ent-storage` dataset for the LRU case; there is a strong correlation between groups becoming sequential and groups becoming less predictive. In the parallel figure for the `fiu` dataset, we see no relationship between the sequentiality and the percentage of predictive accesses. Instead, we believe that the difference in the two datasets in the LRU case comes from the average IOPS of `ent-storage` being high enough to make groups more transient.

We also verified our theory that the LFU `ent-storage` case suffered from cache churn by tracking correct predictions over time based on cache size. In Figure 9, we see that the few accesses that have a chance to be predictive are correct for small cache sizes before taking a precipitous drop as the cache size grows. This indicates two things: first, that working sets are being evicted early on, leading to an inflated rate of correct predictions, and second, that as the

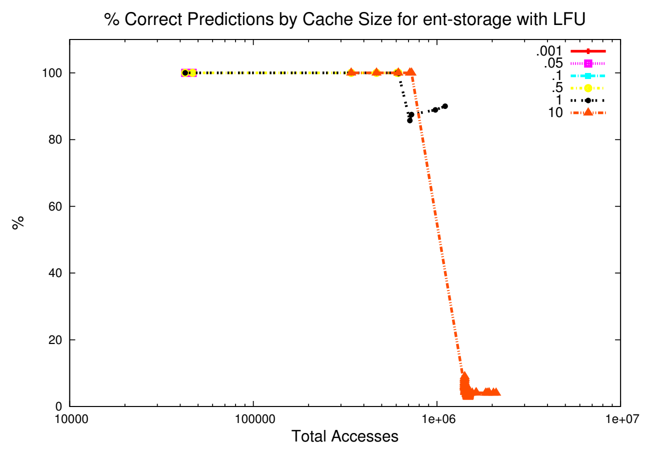


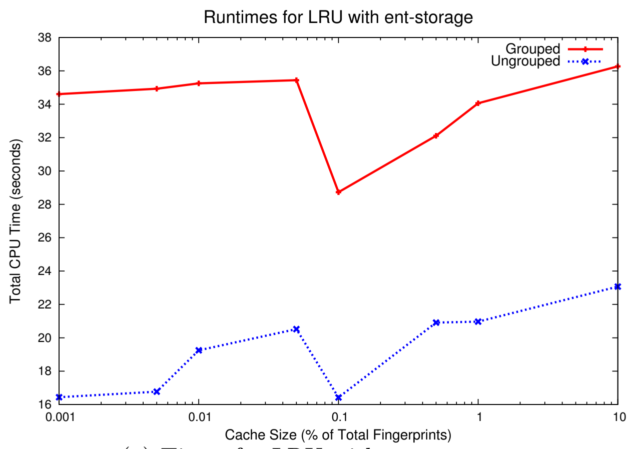
Figure 9: % of correct predictions by cache size, compared against the total fingerprints pulled into cache and not immediately replaced. We see that the few fingerprints that were pulled in were predictive at small cache sizes, while extra elements were pulled into a larger cache.

cache grows there are enough legitimate fingerprints being removed from cache that even an improvement in longevity of working set members is not enough to salvage the algorithm. LFU is simply a poor choice for a deduplication index cache with working sets.

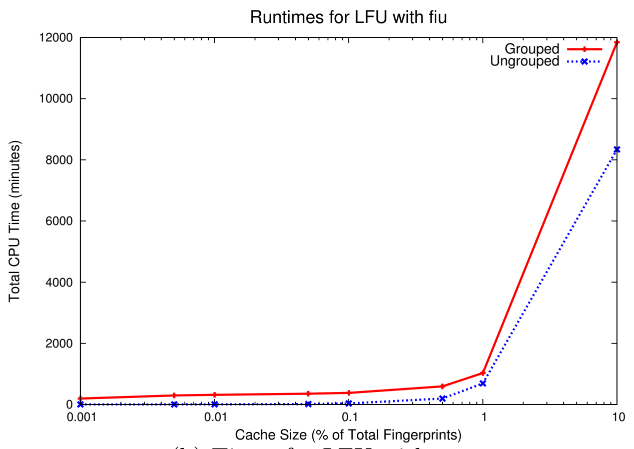
Figure 10(a) shows that smaller cache sizes pull in more LBAs early on, but over time the total number of LBAs pulled converges. Even though the average increase in cache hit rate for adding working sets to `ent-storage` for LRU was only about 10%, we accomplish this by only pulling in 3% of the total LBAs. For the `fiu` case, we see that at about one million accesses, where the `ent-storage` trace ends, about 15% of the total LBAs are pulled in. In contrast to `ent-storage`, by the end of the trace almost 80% of the total dataset LBAs had been pulled in. This high percentage of LBAs in cache is almost certainly why the cache performed so well for the `fiu` with LRU case.

One concern with this method is that the working sets are made without the content data in mind. As we see in Figure 11, the fingerprint-LBA pairing is transient. If this pairing falls away before working sets are re-calculated, the quality of predictions could decline. In this case, however, there were significantly more write accesses than read accesses, and we believe that the different fingerprints on consecutive LBAs thus represent different content with the same access characteristics. In a subset of `fiu` with a slightly higher read ratio of 9.5%, the number of data shifts drops down to 58.5%. Even with our high read ratio, we see that for both the `fiu` and `ent-storage` datasets the correlations between LBAs and fingerprints remain fairly consistent until the system sees about 500,000 accesses. We are interested in acquiring more datasets with fingerprints to determine how this compares to other types of workloads.

Determining when to recalculate the groupings will be essential to future real-time systems. Though we did not recalculate groupings often during the course of our runs because we had relatively little data, we looked for insights to determine when to recalculate our groupings. Figures 8(b) and 8(a) show that the predictive power of our groupings is strongest early on. A real system could have an auto-



(a) Times for LRU with `ent-storage`



(b) Times for LFU with `fiu`

Figure 12: A comparison of runtime vs. cache size for our algorithms. This was run in a prototyping environment; for translation to a real system the key is that the grouped performance closely tracks the ungrouped performance with a fixed overhead.

matic alert system to track the level of predictive power and re-calculate groups in the background as needed. Alternatively, working groups could be calculated even more frequently to correspond with the need to go to disk to fetch recent LBA-fingerprint pairs. As we see in Figure 11, the LBAs and fingerprints shift at about 500,000 accesses in both traces, though the `fiu` trace stops shifting for a time after. More frequent group calculation is likely to slightly improve cache hit rate numbers as the groups will more closely match the current working environment. However, since groups are based on a somewhat longer term system view, re-calculation should not provide a large bonus in the absence of a major usage shift.

5.3.1 *Performance*

While keeping the working set table up to date results in some CPU overhead, we accept this because it is negligible compared to the penalty for the I/O. It is likely that the systems' own cache policies will also place the data from the active working set into system cache, but that is outside the scope of this paper.

Our implementation was done on a personal desktop using Python for both group calculation and cache simulation. Though we used the PyPy high-speed Python implementation [21], our code is designed for prototyping and thus lacks some optimizations. As a result, the performance numbers we have should only be considered relative to each other. Figure 12 shows the performance for LRU for `ent-storage` and LFU for `fiu`. These graphs are representative of all of the experiments we did, and show that while the grouped version takes about twice as long, the grouped and ungrouped lines closely track each other. This indicates that overhead is mostly fixed and can be predicted.

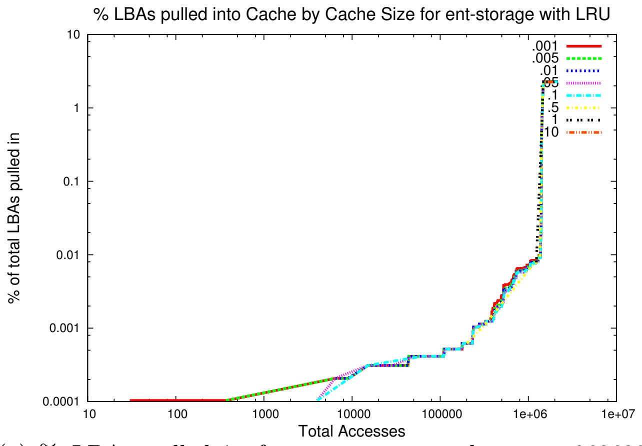
6. DISCUSSION AND FUTURE WORK

In our experiments, we only allowed the system to cache fingerprints it had seen during the course of the trace. This emulates starting from a clean slate, but in a real system once it reaches a steady state the amount of data pulled in should increase.

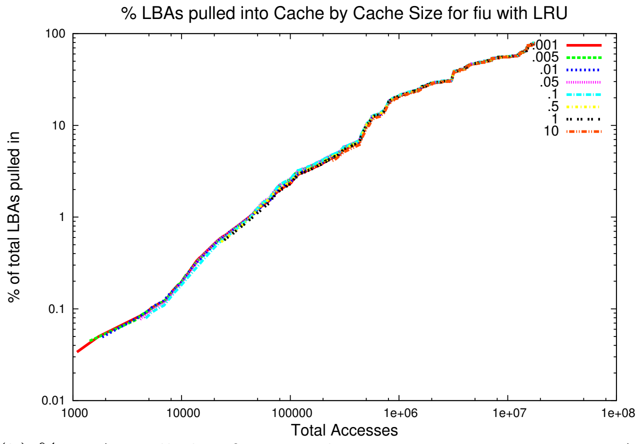
It is important to note that, contrary to our expectations, adding working sets to the index cache never reduced the cache hit rate. This indicates that there is still room for larger working sets to be pulled in before cache churn starts becoming a serious problem. Thus, real world results with some tuning could likely produce more significant improvement in cache hits. For example, if instead of periodically updating the working set table the working sets likelihoods are updated continuously the system would see more large working sets and corresponding cache hits. Additionally, we create working sets based on a bare-bone set of features, namely location and timestamp. We used these to show our method was valuable even in cases where it is not practical to extract rich system traces. Additional features such as request origin or client type could make the working sets more reflective of real workload phenomena. Additionally, any method for working set prediction or even file level access prediction can be substituted into our system with little effort, making our method a general purpose tool to improve primary deduplication.

There are also workloads where we believe this method could work significantly better than it does for our two workloads. For example, a large bank of virtual machine images should have both higher deduplication ratios and tighter working sets, since they correspond closely to individual systems. Workloads from businesses that operate based on strict timing rules such as banks and trading houses should show a cyclic usage pattern that would be amenable to our deduplication method.

In addition to large scale data, our technique could be applied to object stores where any reduction in media cost is amplified because of the relative expense of storage class memory technologies. Our method could also be used to cost-effectively address the growing problem of archival-like storage that does not obey "write-once, read maybe" semantics and instead has transient periods of primary activity. Storing exabytes of data and maintaining a usable primary deduplication index is prohibitive, but storing 1% of the index size in memory should be much more manageable and cost effective for long term storage such as Internet archives and media.



(a) % LBAs pulled in for `ent-storage`; there are 968620 unique LBAs



(b) % LBAs pulled in for `fiu`; there are 1684407 unique LBAs

Figure 10: Each line corresponds to a run with the given cache size. Adding working sets to `ent-storage` with LRU achieves an increased cache hit rate while only pulling in $< 3\%$ of the total LBAs. Conversely, `fiu` with LRU pulls in up to 80%.

We are seeking new data sets to test our algorithm on, particularly datasets with a high degree of inherent duplication. Once we gather enough of these datasets, we intend to attempt a characterization of workloads based on the most salient features for working set analysis and examine the possibility of automatically tuning working set algorithms based on the workload type. We also would like to explore the intersections of our work and other working set and file prediction technologies.

7. CONCLUSION

Traditional in-line deduplication typically requires a large in-memory index to make deduplication feasible, yet such an index is so resource-intensive that it greatly reduces the attractiveness of inline deduplication for primary storage. To address this problem, we have proposed the use of a fingerprint cache guided by algorithms to predict and then prefetch accesses. Our approach calculates working sets and pulls entire sets of fingerprints into memory when a single fingerprint is accessed, significantly increasing the number of cache hits for the in-memory deduplication index cache. We also showed that LRU is a relatively good caching algorithm for our fingerprint groupings, and we hypothesize that a similar pattern will hold for most other workloads.

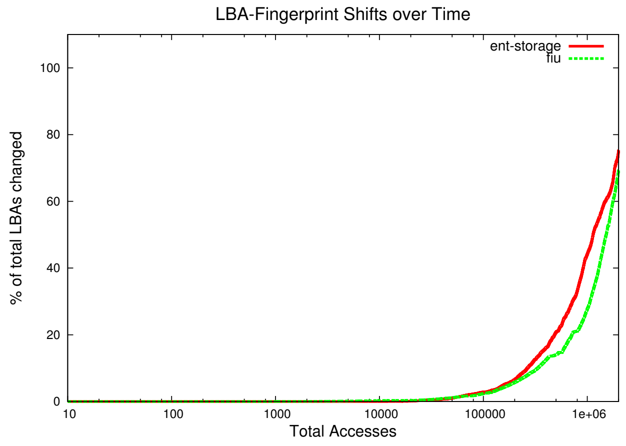


Figure 11: LBA-fingerprint correlation shifts over time. Though the data sets have very different characteristics, both show a sharp rise in shifts after about 500,000 accesses.

Our design translates directly into fewer disk accesses for inline deduplication and, from there, better performance. Our system is highly modular and adaptable to specific environmental constraints, and thus is an approach that can be deployed in nearly any system to alleviate the disk bottleneck problem. Since only a small percentage of LBAs need to be kept in cache for good performance, primary deduplication should become much more widely used to help lower costs and manage our wealth of data. Although neither of our workloads had the requisite 45% deduplication ratio to benefit from traditional primary deduplication, with HANDS they can achieve near perfect deduplication using just .01% to 10% of the memory. As server consolidation and virtualization continue to become more common, our design makes primary deduplication an affordable option for almost all real systems.

8. ACKNOWLEDGMENTS

We would like to thank David Chambliss, Jorge Guerra, and Maohua Lu for providing us with data and many valuable discussions and arguments. This work was partially conducted at and supported by IBM Almaden Research Center. This research was also supported in part by the National Science Foundation under awards CNS-0917396 (part of the American Recovery and Reinvestment Act of 2009 [Public Law 111-5]) and IIP-0934401. We also thank the industrial sponsors of the SSRC for their generous support and all SSRC members for insightful discussions and feedback.

9. REFERENCES

[1] I. Adams, E. L. Miller, and M. W. Storer. Analysis of workload behavior in scientific and historical long-term data repositories. Technical Report UCSC-SSRC-11-01, University of California, Santa Cruz, Mar. 2011.

[2] A. Amer, D. D. E. Long, J.-F. Pâris, and R. C. Burns. File access prediction with adjustable accuracy. In *Proceedings of 21st International Performance,*

Computing, and Communications Conference (IPCCC 2002), Phoenix, Arizona, 2002. IEEE.

[3] A. Arpaci-Dusseau, R. Arpaci-Dusseau, L. Bairavasundaram, T. Denehy, F. Popovici, V. Prabhakaran, and M. Sivathanu. Semantically-smart disk systems: past, present, and future. *ACM SIGMETRICS Performance Evaluation Review*, 33(4):35, 2006.

[4] B. Battles, C. Belleville, S. Grabau, and J. Maurier. Reducing data center power consumption through efficient storage. *Network Appliance, Inc*, 2007.

[5] D. Bhagwat, K. Eshghi, D. Long, and M. Lillibridge. Extreme binning: Scalable, parallel deduplication for chunk-based file backup. In *Modeling, Analysis & Simulation of Computer and Telecommunication Systems, 2009. MASCOTS'09. IEEE International Symposium on*, pages 1–9. IEEE, 2009.

[6] C. Constantinescu, J. Glider, and D. Chambliss. Mixing deduplication and compression on active data sets. In *2011 Data Compression Conference*, pages 393–402. IEEE, 2011.

[7] W. Dai. Crypto++ 5.6.0 benchmarks. 2009.

[8] S. Doraimani and A. Iamnitchi. File grouping for scientific data management: lessons from experimenting with real traces. In *Proceedings of the 17th international symposium on High performance distributed computing*, pages 153–164. ACM, 2008.

[9] T. Economist. Data, data everywhere. *The Economist Newspaper Limited*, February 2010.

[10] P. Efstathopoulos and F. Guo. Rethinking deduplication scalability. In *HotStorage10, 2nd Workshop on Hot Topics in Storage and File Systems*, 2010.

[11] J. Gantz, C. Chute, A. Manfrediz, S. Minton, D. Reinsel, W. Schlichting, et al. The Diverse and Exploding Digital Universe. *IDC White Paper*, 2, 2008.

[12] K. Jin and E. L. Miller. The effectiveness of deduplication on virtual machine disk images. May 2009.

[13] S. Jones. Online de-duplication in a log-structured file system for primary storage. Technical Report UCSC-SSRC-11-03, University of California, Santa Cruz, May 2011.

[14] R. Koller and R. Rangaswami. I/o deduplication: utilizing content similarity to improve i/o performance. *ACM Transactions on Storage (TOS)*, 6(3):1–26, 2010.

[15] K. Lawrence. Re-thinking the lamp stack: Part 2. *PINGV*, December 2010.

[16] Z. Li, Z. Chen, S. Srinivasan, and Y. Zhou. C-miner: Mining block correlations in storage systems. In *Proceedings of the 3rd USENIX Conference on File and Storage Technologies*, pages 173–186. USENIX Association, 2004.

[17] M. Lillibridge, K. Eshghi, D. Bhagwat, V. Deolalikar, G. Trezise, and P. Camble. Sparse indexing: large scale, inline deduplication using sampling and locality. In *Proceedings of the 7th conference on File and storage technologies*, pages 111–123. USENIX Association, 2009.

[18] N. Mandagere, P. Zhou, M. Smith, and S. Uttamchandani. Demystifying data deduplication. In *Proceedings of the ACM/IFIP/USENIX Middleware'08 Conference Companion*, pages 12–17. ACM, 2008.

[19] D. Meyer and W. Bolosky. A study of practical deduplication. In *Proceedings of the 9th USENIX conference on File and stroage technologies*, pages 1–1. USENIX Association, 2011.

[20] J.-F. Pâris, A. Amer, and D. D. E. Long. A stochastic approach to file access prediction. In *The International Workshop on Storage Network Architecture and Parall I/Os (SNAPI '03)*, sep 2003.

[21] A. Rigo and S. Pedroni. Pypy's approach to virtual machine construction. In *Companion to the 21st ACM SIGPLAN symposium on Object-oriented programming systems, languages, and applications*, pages 944–953. ACM, 2006.

[22] A. Riska and E. Riedel. Disk drive level workload characterization. In *Proceedings of the USENIX Annual Technical Conference*, pages 97–103, 2006.

[23] K. Srinivasan, T. Bisson, G. Goodson, and K. Voruganti. iDedup: Latency-aware, inline data deduplication for primary storage. In *Proccedings of the 10th conference on File and storage technologies.* USENIX Association, 2012.

[24] M. Storer, K. Greenan, D. Long, and E. Miller. Secure data deduplication. In *Proceedings of the 4th ACM international workshop on Storage security and survivability*, pages 1–10. ACM, 2008.

[25] Y. Tsuchiya and T. Watanabe. Dblk: Deduplication for primary block storage. In *Mass Storage Systems and Technologies (MSST), 2011 IEEE 27th Symposium on*, pages 1–5. IEEE, 2011.

[26] G. A. S. Whittle, J.-F. Pâris, A. Amer, D. D. E. Long, and R. Burns. Using multiple predictors to improve the accuracy of file access predictions. In *Proceedings of the 20th IEEE / 11th NASA Goddard Conference on Mass Storage Systems and Technologies*, pages 230–240, Apr. 2003.

[27] A. Wildani and E. Miller. Semantic data placement for power management in archival storage. In *Petascale Data Storage Workshop (PDSW), 2010 5th*, pages 1–5. IEEE, 2010.

[28] A. Wildani, E. Miller, and L. Ward. Efficiently identifying working sets in block i/o streams. In *Proceedings of the 4th Annual International Conference on Systems and Storage*, page 5, 2011.

[29] T. Yang, H. Jiang, D. Feng, Z. Niu, K. Zhou, and Y. Wan. Debar: A scalable high-performance de-duplication storage system for backup and archiving. In *Parallel & Distributed Processing (IPDPS), 2010 IEEE International Symposium on*, pages 1–12. IEEE, 2010.

[30] B. Zhu, K. Li, and H. Patterson. Avoiding the disk bottleneck in the data domain deduplication file system. In *Proceedings of the 6th USENIX Conference on File and Storage Technologies*, page 18. USENIX Association, 2008.