Improving Restore Speed for Backup Systems that Use Inline Chunk-Based Deduplication

Mark Lillibridge[†], Kave Eshghi[†], and Deepavali Bhagwat[♯]
[†]*HP Labs* [♯]*HP Storage*
first.last@hp.com

Abstract

Slow restoration due to chunk fragmentation is a serious problem facing inline chunk-based data deduplication systems: restore speeds for the most recent backup can drop orders of magnitude over the lifetime of a system. We study three techniques—increasing cache size, container capping, and using a forward assembly area—for alleviating this problem. Container capping is an ingest-time operation that reduces chunk fragmentation at the cost of forfeiting some deduplication, while using a forward assembly area is a new restore-time caching and prefetching technique that exploits the perfect knowledge of future chunk accesses available when restoring a backup to reduce the amount of RAM required for a given level of caching at restore time.

We show that using a larger cache per stream—we see continuing benefits even up to 8 GB—can produce up to a 5–16X improvement, that giving up as little as 8% deduplication with capping can yield a 2–6X improvement, and that using a forward assembly area is strictly superior to LRU, able to yield a 2–4X improvement while holding the RAM budget constant.

1 Introduction

Inline chunk-based deduplication is a well established technique for removing data redundancy [15, 19]. It is used in commercial disk-to-disk (D2D) backup systems to greatly reduce the amount of data that must be stored [12, 27]. It works by dividing up incoming data into small ($\approx$ 4–8 KB) pieces, called chunks, and storing only chunks that do not already have an identical copy in the store. For chunks that already have an identical stored copy, only a reference to the existing copy is stored. Because daily backups are highly redundant, this removal of duplicate data can produce great savings. Savings of 10–30X are typical [3].

Unfortunately, restore speed in such systems often suffers due to *chunk fragmentation*, especially after many backups have been ingested. Chunk fragmentation, like disk fragmentation, results when the chunks of a backup become scattered all over rather than arranged in a nice compact continuous sequence. Because of modern disks' relatively poor random I/O performance compared to sequential I/O, fragmentation greatly hurts restore performance.

Chunk fragmentation primarily arises in these systems from the sharing of chunks between backups. When a backup is ingested, its new chunks are laid out in the order they arrive but its old chunks are left where they were originally deposited. Put another way, stored chunks are grouped by the first backup they appeared in so the chunks of the latest backup can be in many places. Because chunks do not arrive ordered by age, restoration must jump back and forth between different chunk groups as data of different ages is encountered.

Unlike disk fragmentation, which can be remedied by a simple re-arranging of blocks, chunk fragmentation is more problematic. Due to chunk sharing between backups, it is not possible in general to find a chunk layout that reduces the fragmentation of every or even most backups. Rearranging chunks can also be very expensive: for example, if we attempt to keep all the chunks of the latest backup together (i.e., move old chunks next to new chunks), then we need for each backup ingested to move data proportional to the pre-deduplicated backup size rather than the potentially orders-of-magnitude smaller deduplicated backup size.

Chunk fragmentation and hence restore performance gets worse as time goes by, and affects the recent backups the most. This slow down can be quite substantial: for our data sets, we see slowdowns of 4X over three months for one and 11X over two years for the other.

In this paper, in addition to investigating how fragmentation and restore speed behave over time and under different cache sizes, we investigate two approaches to

improving restore speed in these deduplication systems: First, we consider capping where we trade off deduplication for reduced chunk fragmentation by deduplicating each backup section against only a limited number of places. This limits how many places need be accessed per backup MB, which speeds up restore at the cost of missing some deduplication that could have been provided by other places. Second, we exploit the fact that we have advance knowledge of the accesses that will be required to restore the backup courtesy of the backup chunk-reference list (*recipe*) to create a new, more efficient caching and prefetching method for restoring deduplicated data, the *forward assembly area* method. Neither of these approaches require rearranging chunks.

The rest of this paper is organized as follows: in the next section, we describe the problem of slow restoration due to chunk fragmentation in more detail. In Section 3, we describe our approach to handling this problem. In Section 4, we report on various simulation experiments with real and synthetic data on the effectiveness of our approach. Finally, we describe future work in Section 5, related work in Section 6, and our conclusions in Section 7.

2 The Problem

We are concerned here with stream-based data backup using deduplication, where the backup application turns the source data into a stream that is sent to the backup appliance for storage. We are not concerned with the details of the mapping from the source data to the stream, so we talk about the storage and retrieval of backup (streams), where a backup is typically many GBs. Since we are dealing with chunk-based deduplication, we consider a backup to be a sequence of chunks generated by a chunking algorithm.

Typically, these systems store chunks in a number of files called *chunk containers*. The standard method used for storing new incoming chunks [12, 27] is as follows: at any given time, there is one container per incoming stream, called its *open container*, which is used for storing new chunks from that stream. When a new chunk arrives, it is simply added to the end of the relevant open container. When the size of an open container reaches a certain threshold, typically 4 MB, it is closed and replaced with a new empty container. Open containers are also closed when the end of their associated stream is reached. The new chunks of each backup are thus grouped into a series of new 4 MB container files.

The baseline restore algorithm is to scan down the backup's *recipe*—one or more files containing a list of references to the backup's chunks in the order they make up the backup—retrieving each chunk in order one at a time by paging in the entire chunk container containing that chunk. Although reading entire chunk containers seems wasteful at first, it makes sense for two reasons: First, with modern RAID systems, so much data can be read in the time it takes to do a single seek that it is faster to read unnecessary data than seek past it. For example, a RAID 6 group with 10 data drives @ 100 MB/s each can read an entire 4 MB container in 4 ms whereas a seek requires 10 ms.

Second, recipe references in these systems do not contain chunk offsets directly but rather indirect through indexes usually located at the start of each container. This indirection speeds up container compaction when backups are deleted because there is no need to update recipes, which can consume a substantial fraction of storage with high deduplication factors. Given that we have to seek to the start of the container anyway to read its index, it will be faster to just keep reading rather than seeking again to the first chunk we want. Because we will not generally be able to consult the index until the read finishes, stopping before the end of the container risks requiring another expensive seek. Under these conditions, it is more efficient to blindly read the entire container even when we need only one chunk.

This algorithm relies on there being sufficient caching of containers—either by the operating system or an application-specific cache—in order to achieve good performance. Because we wish to quantify the effect of cache size and caching algorithm on restore performance, we simulate this caching explicitly.

Our default cache is a simple n-slot LRU container cache, which can hold up to n chunk containers at a time and takes $n\times$container size space. We simulate restoring only one stream at a time using this cache; real systems, of course, need to be able to handle many streams at a time—some current systems can do over 100 simultaneous streams—and thus would need proportionately more cache space. Real systems have many demands on their limited RAM including being able to ingest and restore simultaneously and so cache efficiency is of great concern.

The consequences of all this can be seen in Figure 1, which shows the mean number of containers read per MB of backup restored for the backups of the data set 2year-cycle30 (described in Section 4.2) for 4 LRU cache sizes. Containers read per MB restored is a restore-relevant way of measuring a backup's fragmentation at a given scale (aka, cache size). As can be seen, even allowing a great deal of cache, each succeeding backup is more fragmented, requiring more containers to be read per MB of data restored. Because we do not rearrange chunks, a backup's fragmentation remains constant as more backups are added. More information on this and related experiments can be found in Section 4.

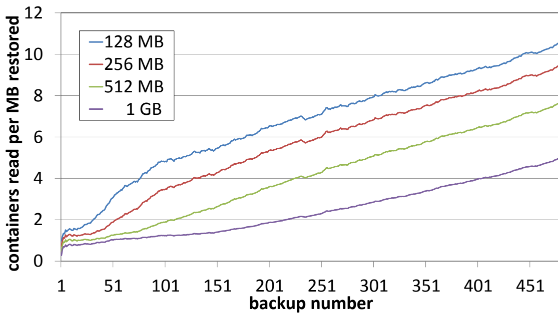


Figure 1: **Fragmentation over time** for 2year-cycle30 data set for 4 different LRU cache sizes. Shown is a simple 20 backup moving average; no capping was used.

Because reading chunk containers is by far the dominant restore cost, this means that restore speed on any of these systems is inversely proportional to this measure of fragmentation; here fragmentation increases 18X when using a 1 GB cache (11X using a 128 MB cache) over the course of two years resulting in restore speed dropping by similar factor. This large slowdown of restore performance over time due to chunk fragmentation is the problem we are concerned with in this paper.

By making assumptions about a system's I/O performance, we can estimate real-world restore speed from these numbers. If we assume reading a container takes 24 ms (2 seeks, 1 for inode and 1 for start of continuous sequential data, @ 10 ms each plus 4 ms to read 4 MB @ 1000 MB/s due to 10 data drives in a RAID 6 group), then the final speeds are 1/(24 ms $\times$ 5.0/MB) = 8 MB/s with a 1 GB cache and 3 MB/s with a 128 MB cache. Even with better hardware, these fragmentation levels are unlikely to yield acceptable speeds for restoration in an emergency.

3 Our Approach

In this section, we describe our approach to measuring restore speed and the new two techniques we investigated for improving restore performance, capping and the forward assembly area method.

3.1 Measuring restore speed

We introduce a new restore speed proxy, *speed factor*, defined as the inverse of our fragmentation measure, namely *1/mean containers read per MB of data restored*. This measure ignores restore speed variance due to filesystem physical fragmentation, faster seeks when files are closer together, and the like in favor of concentrating on the dominant cost, container reading.

If container size is kept constant (most of this paper), then relative differences in speed factor observed on one system, say due to caching changes, should be similar on another system even though their raw I/O performance (seek and sequential read speeds) may be quite different. This breaks down if container size is allowed to vary, however (see Section 4.7).

Given assumptions about raw I/O performance and knowledge of the container size used to generate the speed factor, an estimate of absolute system performance can be generated from the speed factor. For example, under the example assumptions of the previous section, one unit of speed factor translates to 1 MB/24 ms = 41 MB/s using standard 4 MB chunk containers.

3.2 Container capping

One way to limit chunk fragmentation and thus increase restore speed is to limit how many containers need be read at restore time for each section of the backup. We can do this by capping how many old containers each section of the recipe can refer to. In order to use fewer old containers, we will have to give up deduplication: instead of using a reference to an existing chunk copy in an old container, we will have to store a duplicate copy of that chunk in an open container and point to that copy. Capping, unlike using the forward assembly area method, thus trades off deduplication for faster restore speed. This basic idea of trading off deduplication for faster restore speed is not new; see Section 6 for earlier uses.

Capping requires breaking up the backup stream into sections, which we call *segments*. By default, we use 20 MB "fixed"-size segments, which works out to about 5000 4 KB mean-size chunks per segment. Our segments are actually slightly shorter than the stated fixed size in order to align chunk and segment boundaries: we end each segment one chunk before it would become too large.

Ingestion with capping works a segment at a time; we process each segment as follows: First, we read in a segment's worth of chunks to an I/O buffer. Second, we determine which of these chunks are already stored and in which containers. How chunks are found does not matter for capping; one possible method might be that of Zhu *et al.* [27], which uses a full chunk index on disk combined with a Bloom filter and caching of chunk container indexes. Note that capping requires an extra segment-sized buffer in RAM for each stream being ingested.

Third, we choose up to T old containers to use (a *capping level* of T containers per segment size) based on the information we have about which containers contain which chunks of the segment. We would like to lose

as little deduplication as possible here while remaining within our cap of T old containers per segment. Accordingly, we rank order the containers by how many chunks of the segment they contain, breaking ties in favor of more recent containers, and choose the top T containers, which contain the most chunks.

Finally, we compute the recipe section for the segment and append any "new" chunks to the open container. In doing this, old chunks found in one of the chosen T containers generate references to existing chunks as usual but old chunks found in other containers are treated as if they were new chunks. That is, both the later old chunks and any new chunks are appended to the current open container and references are made to those new copies. Recipe and open container data is flushed to disk in a buffered manner, with open containers usually only written once full.

This process guarantees that the recipe for each segment refers to at most T old containers plus 1 to 5 (= 20 MB/4 MB) new containers containing "new" chunks. Except for segments containing mostly new data such as in initial backups, most segments refer to only one or, in case of filling up the existing open container, two new containers. Clearly, this caps fragmentation at the scale of our segment size, 20 MB; as we shall see in Section 4.4, it also reduces fragmentation and thus improves restore performance at higher scales.

Capping level, although expressed as T containers per S MB segment, is not really a ratio as the choice of denominator (segment size) matters independent of T/S. Levels of 20 containers per 20 MB and 40 containers per 40 MB are not equivalent: generally the latter allows more containers per 20 MB section because containers used in both of a 40 MB segment's halves count only once towards its cap but twice (once per half) when 20 MB segments are being used.

3.3 Forward assembly area

The restore problem for deduplicated backup streams differs in two important ways from the virtual memory paging problem: First, the effective unit of I/O is an entire chunk container (4 MB) whereas the unit of use is a much smaller variable-size chunk (≈4–8 KB). Second, at the time of starting the restore we have perfect knowledge of the exact sequence of chunks that will be used thanks to the backup's recipe.

We have devised a new restore algorithm that exploits these differences to improve performance over traditional paging algorithms like LRU. As past work has shown [4,5,18], future knowledge of accesses can be exploited to improve both caching and prefetching. Careful arrangement of data can also save space and reduce memory copies.

#	hash	CID	length	start	end
0	64aeca58	51	4037	0	4036
1	3f886668	13	6772	4037	10708
2	b52bed44	47	1900	10709	12608
3	8d39de3b	13	2199	12609	14807
4	64aeca58	51	4037	14808	18844
5	c8fd7a94	28	4041	18845	22885

Table 1: **Allocation of forward assembly area** from recipe. The recipe is on the left (CID = container ID, unrealistically short hashes used, one chunk per line) and the resulting allocation on the right.

We page in chunk containers to a single buffer but cache chunks rather than containers to avoid keeping around chunks that will never be used and consult the next part of the recipe to make better decisions about what chunks from the paged-in containers to retain. Our algorithm uses one chunk-container–sized I/O buffer, a large *forward assembly area* where the next M bytes of the restored backup will be assembled, and a recipe buffer big enough to hold the part of the recipe that describes the piece being assembled.

In the simplest case, we restore M-byte *slices* of the backup at a time by first assembling each M-byte backup slice in the forward assembly area and then sending it out in a single piece. To restore a M-byte slice, we first read in the corresponding part of the recipe into the recipe buffer. From that recipe part, we can determine which chunks are needed to fill which byte ranges (*chunk spots*) of the forward assembly area. See Table 1 for an example of this.

We then repeatedly find the earliest unfilled chunk spot in the assembly area and load the corresponding container into our I/O buffer then fill all parts of the assembly area that need chunks from that container. For table 1, this results in loading container 51, filling chunk spots 0 and 4 with the same data (note duplicated chunk); loading container 13, filling spots 1 and 3; loading container 47, filling spot 2; and finally loading container 28 and filling spot 5.

With this method, we need load each container only once per M-byte slice and do not keep around chunks that will not be needed during this slice, unlike with container caching. We do waste a small amount of space when chunks occur multiple times (e.g., chunk 64aeca58 above), however. This could be avoided at the cost of more complicated bookkeeping and extra memory copies. Note that our method handles variable-size chunks well, without requiring padding or complicated memory allocation schemes that waste memory due to memory fragmentation.

In addition to the simple version described above, which we call *fixed*, we also consider a variant, *rolling*, where we use a ring buffer for the forward assembly area. Here, after we have finished with each container, we send out the continuous filled-in part at the (logical) start of the forward assembly area and then rotate the ring buffer to put that now unused space at the end of the forward assembly area. In the example, we would send out 0–4036 then make chunk spot 1 (old offset 4037) the start of the ring buffer, adding 4037 bytes to the end of the ring buffer, allowing the allocation of further chunks. By incrementally loading, filling, sending, rotating, and allocating new chunk spots, we can use our memory more efficiently and ensure that we need load each container at most once every M bytes, no matter where slice boundaries would have been. By contrast, the non-rotating variant requires reloading containers whose uses, while nearby, span a slice boundary.

4 Experimental Results

In order to test our approach, we modified one of our deduplication simulators. We apply it to two primary data sets and report on deduplication, fragmentation and restore performance under various caching strategies, and the effects of capping and container size.

4.1 Simulator

The simulator, a 9,000 line C++ program, is capable of simulating many different modes and styles of deduplication. Here, we have set it to simulate a system with a full chunk index that maps the hashes of stored chunks to the chunk containers they occur in. In the absence of capping, this allows the system to achieve so-called perfect deduplication where no chunk is ever duplicated.

Capping when enabled is done as described in section 3.2 using 20 MB fixed-size segments by default. Using it, duplicate chunks are intentionally sometimes stored and the full chunk index is updated to point to the most recent copy of each chunk. When a chunk is deduplicated, the resulting backup recipe points to the particular copy of that chunk that was found via the index.

Except when we say otherwise, we are using 4 MB chunk containers per Zhu *et al.*; a chunk container becomes full when adding the next chunk would put it over the maximum container size. Thus, in the absence of deletion almost all chunk containers will be essentially the same size, namely 4 MB.

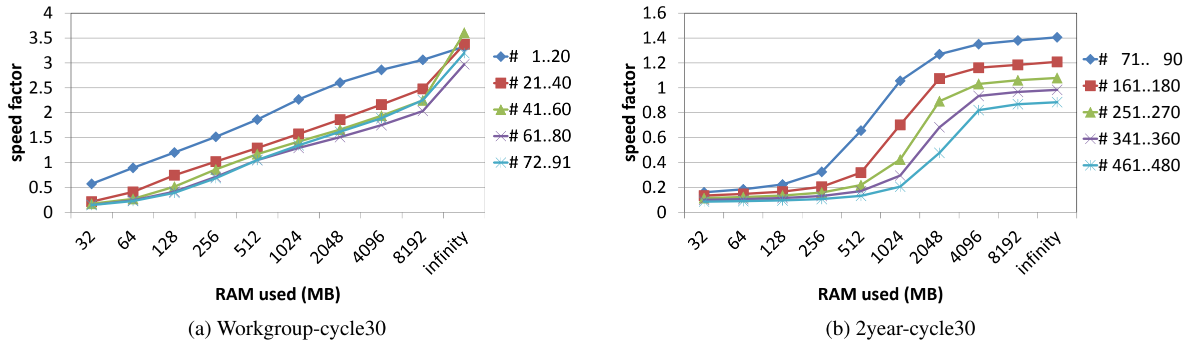
We simulate deletion of backups by removing their recipes and the chunk copies that are now garbage (e.g., no longer pointed to by any backup recipe) from their containers. We do not simulate the merging of now-too-small containers after deletion or other kinds of housekeeping that involve moving chunks between containers or merging duplicate copies of a chunk. When a chunk pointed to by the index is garbage collected, the entry in the index for that hash is removed even if another copy of that chunk exists. We do this even though it can cause loss of deduplication because we expect this to occur rarely—usually older copies are deleted first—and because avoiding it in a real system is likely to be expensive.

The simulator produces various statistics. In this paper, we are primarily concerned with the *(estimated) cumulative deduplication factor* (total raw/total deduplicated = sum of every non-deleted input chunk's length / sum of every currently-stored chunk copy's length) and a measure of restore speed, *speed factor* = 1/containers read per MB (restored). Note that the estimated cumulative deduplication factor does not take into account metadata overhead (causing it to overestimate) or local compression of chunks (causing it to underestimate total compaction). In general, we ignore local compression for this paper; thus, for example, our maximum container size limit is a bound on the sum of the uncompressed lengths of the chunks contained in a container.

4.2 Data sets

We report results for two main data sets. The first data set, which we call *Workgroup*, is created from a semi-regular series of backups of the desktop PCs of a group of 20 engineers taken over a period of four months. The backups were taken using uncompressed tar. Although the original collection included only an initial full and later weekday incrementals for each machine, we have generated synthetic fulls at the end of each week for which incrementals are available by applying that week's incrementals to the last full. The synthetic fulls replace the last incremental for their week; in this way, we simulate a more typical daily incremental and weekly full backup schedule. We are unable to simulate file deletions because this information is missing from the collection.

There are 154 fulls and 392 incrementals in this 3.8 TB collection, with the fulls ranging from 3 GB to 56 GB, with a mean size of 21 GB. Note that because these machines were only powered up during workdays and because the synthetic fulls replace the last day of the week's back up, the ratio of incrementals to fulls (2.5) is lower than would be the case for a server (6 or 7). We generate the Workgroup data set from these backups by grouping backups taken on the same day into a single backup. This grouping results in 91 "system-wide" backups; we group the original backups this way so that tape cycling (see below) treats backups taken the same day as a unit. The data is chunked into variable-size chunks with mean size 4 KB using the TTTD chunking algorithm [6].



(a) Workgroup-cycle30

(b) 2year-cycle30

Figure 2: **Speed as LRU cache size varies** for two data sets. Each line shows the average speed of restoring the 20 backups whose numbers are given on the right. No capping was used.

The second data set, which we call *2year*, is a synthetic data set provided to us by HP Storage that they have designed to mimic the important characteristics of the data from a past customer escalation involving high fragmentation. This fragmentation stress test is known to produce problematic fragmentation and is used by HP Storage to test their products. It starts from a 10 GB filesystem snapshot and then each simulated weekday randomly selects 2% of the files and synthetically changes 10% of each selected file by seeking to a random offset and overwriting that many bytes; it also adds 200 MB of original files each simulated weekday. During each simulated week, one full backup and four incremental backups, one for each other weekday, are taken via uncompressed tar for a total of 480 backups covering 96 simulated weeks (1.9 years). These backups are chunked via TTTD using a slightly smaller mean chunk size of 3850 bytes.[1]

These data sets make for a good test of the ability to handle fragmentation because they contain the features known to cause greater fragmentation: within backup sharing, incrementals, and most especially a great number of backups. Because these data sets consist of so many backups, it would be unrealistic to simply simulate ingesting one after another—even with deduplication, companies can only afford to keep a limited number of backups on disk in practice.

Instead, we simulate deleting backups according to a schedule, *cycle30*, where backups are kept for only 30 "days". More precisely, we delete the $n-30$th backup before ingesting the nth backup. Although we only report on this schedule, we have experimented with other kinds of schedules, including grandfather-father-son [24]. We find that the exact schedule for deleting backups does not particularly affect our results other than to change the overall deduplication factor.

[1] HP Storage uses this chunking size because chunking can be done faster when the TTTD derived divisors work out to be powers of two, avoiding the need for division.

4.3 Baseline (LRU)

As with any use of caching, we expect restore performance to improve as cache size increases. Figure 2 shows how restore performance for our baseline system (i.e., LRU caching, no capping) varies under LRU caching for a variety of cache sizes for selected 20 backup periods. We show averages of 20 backups here and elsewhere to smooth out the effects of incrementals versus fulls.

You will immediately see that restore speed varies by a huge amount as we change our cache size: switching from 32 MB to 8 GB for Workgroup-cycle30 increases restore speed by 5–16X (more as more backups accumulate) and switching from 32 MB to 8 GB for 2year-cycle30 similarly increases restore speed by 9–10X. Even a smaller change like from 128 MB to 1 GB (8X) increases restore speed by 1.9–3.5X (Workgroup-cycle30) or 4.7–2.2X (2year-cycle30).

Notice that restore speed generally decreases as backups accumulate due to increased fragmentation no matter what cache size is used. How fragmentation increases over time can be seen more clearly in Figures 1 and 3, which plot a measure of fragmentation, containers read per MB restored, as more backups accumulate for four different values of LRU cache size. Fragmentation appears to increase linearly over time: applying linear regression gives R^2 values of over 0.96 for 2year-cycle30

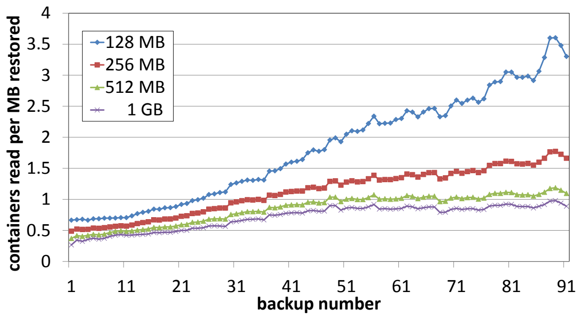


Figure 3: **Fragmentation over time** for Workgroup-cycle30 for 4 different LRU cache sizes. Shown is a simple 20 backup moving average; no capping was used.

---

and 0.98, 0.98, 0.90, and 0.86 for Workgroup-cycle30. As this measure of fragmentation is the inverse of our speed proxy, this means that, given enough backups, doubling the total number of backups halves the final backup restore speed.

If you look carefully at Figure 2, you will see that the shapes of the speed versus cache size curves are different for the two data sets. This is because their levels of fragmentation differ differently at each scale, not just overall. In hindsight, this is unsurprising when we consider the analogy to memory caches and memory traces: a memory trace does not have a single "locality" value, but rather has a locality value for each possible cache size. Knowing how fast a memory trace executes with a cache of one size does not tell you much about how fast it will run at a substantially different cache size unless the working set fits in the first size.

Put another way, fragmentation like memory locality is not a scalar value but rather a one-dimensional curve. Changes that affect fragmentation like capping may affect different scales differently; indeed, it is conceivable that some methods of capping might improve fragmentation at one scale while hurting it at another scale. Accordingly, we will show results at a variety of cache sizes.

4.4 Capping

Capping trades off deduplication for reduced fragmentation and hence restore speed. How useful this trade-off is in practice depends greatly on the shape of the trade-off curve; ideally, giving up only a little deduplication will give us a great deal of speed. Figure 4 shows this trade-off curve for our data sets.

To make this trade-off clearer, in Figure 5 we have plotted for each of several LRU cache sizes how giving up a certain percentage of cumulative deduplication factor via capping increases restore speed. For example, using a 128 MB cache size with Workgroup-cycle30, we can give up 2% of our deduplication in return for a 4.0X speedup or we could give up 8% in return for a 6.4X speed up. The relative improvement of capping when using a 1 GB cache is less: 2% gives us only 1.7X and 8% only 2.3X.

With 2year-cycle30, we need to give up more deduplication to get sizable speedups. For example with 128 MB, giving up 8% yields only 1.7X but 15% yields 3.9X and 23% yields 8.8X. Using 1 GB, 8% yields 3.1X, 15% yields 4.2X, and 23% yields 5.0X. These results demonstrate that it is possible to give up a relatively small percentage of deduplication in practice and get quite substantial speedups.

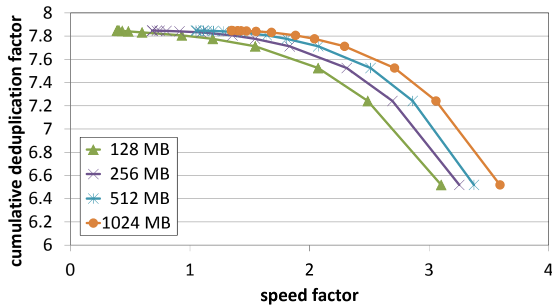
How does capping affect fragmentation over time? Figure 6 shows that capping alters the rate at which fragmentation accrues but does not stop it from accumulating. The more capping applied, the slower fragmentation grows. For example, with 2year-cycle30 and 128 MB LRU, fragmentation grows 11X over 480 backups without capping, but only 6X with cap 90 (90 containers per 20 MB segment), 3X with cap 50, 1.8X with cap 20, and 1.4X with cap 10. Results for other cache sizes are broadly similar, but with differing amounts of vertical spread where higher cache sizes produce less spread.

Our segment size sets the scale of fragmentation we are capping; Figure 7 shows what happens to the capping trade-off curve as we increase this parameter for two LRU cache sizes. At 1 GB, the curve shifts right as deduplication decreases, allowing increased speeds for a given deduplication loss. We estimate that by increasing our segment size to 160 MB we could increase restore speed at 1 GB by 15% (Workgroup-cycle30) or 50% (2year-cycle30) while holding deduplication loss constant.

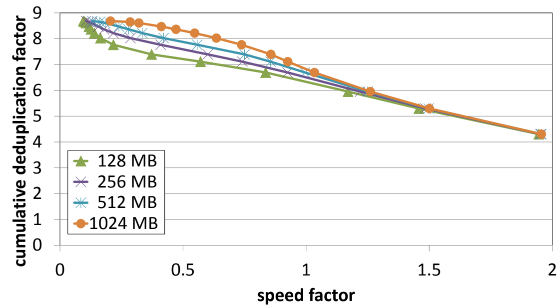
At 128 MB, by contrast, increasing the segment size can produce small or negative speed increases, especially for low levels of deduplication loss or large segment sizes. 40 MB segments do not produce better results than 20 MB megabytes for 2year-cycle30 until deduplication loss exceeds 20%, for example. We are still investigating, but we believe these effects come from capping at segment size S being at a scale equivalent to an forward assembly area of S and thus, as we shall see, a much higher LRU cache size. Capping at a scale substantially above the scale being restored at is unlikely to be particularly effective.

4.5 Assembly

How does our new forward-assembly-area restore method compare to the state-of-the-art? Figure 8 compares it to our baseline system (LRU, no capping). Shown are the two different forward assembly area variants described in Section 3.3: fixed alternates filling

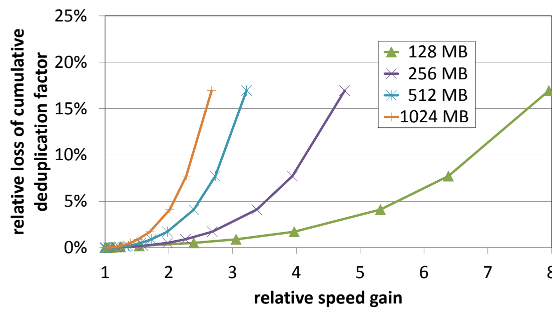


(a) Workgroup-cycle30

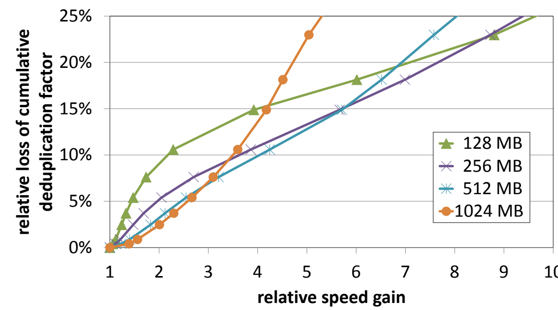


(b) 2year-cycle30

Figure 4: **Deduplication versus speed as capping level and cache size vary** for two data sets. Deduplication and speed values are the average of the last 20 backups using LRU caching with the shown cache size. Each curve shows varying capping levels of from left to right: none, 250, 200, 150, 130, 110, 90, 70, 50, 40, 30, 20, 15, and 10 containers per 20 MB segment. Y-axis of Workgroup-cycle30 starts at 6 to provide more detail.

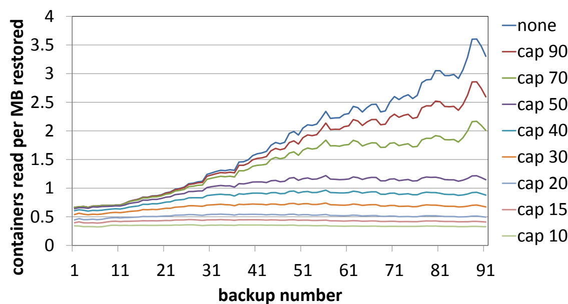


(a) Workgroup-cycle30

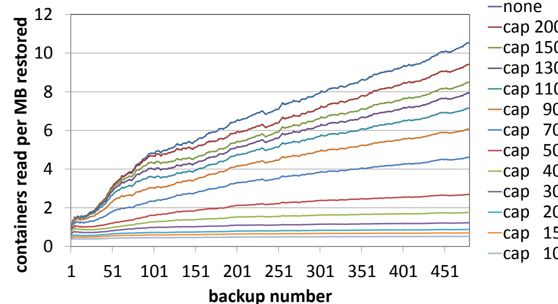


(b) 2year-cycle30

Figure 5: **Relative deduplication loss versus relative speed gain as a result of capping** for selected capping levels and cache sizes for two data sets. Deduplication and speed factor are calculated using the average of the last 20 backups using LRU caching with the shown cache size. Each curve shows varying capping levels of from left to right: none, 250, 200, 150, 130, 110, 90, 70, 50, 40, 30, 20, 15, and 10 containers per 20 MB segment.

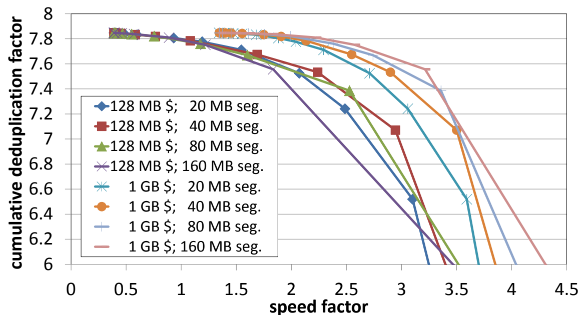


(a) Workgroup-cycle30

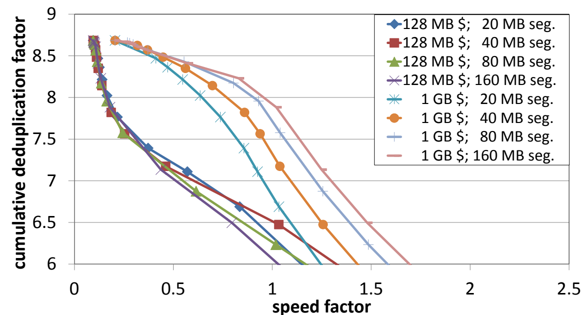


(b) 2year-cycle30

Figure 6: **Effect of varying capping level on fragmentation** over time for two data sets. Shown is a simple 20 backup moving average using 128 MB LRU caching and 20 MB segments. Cap T denotes a capping level of T containers per 20 MB segment.

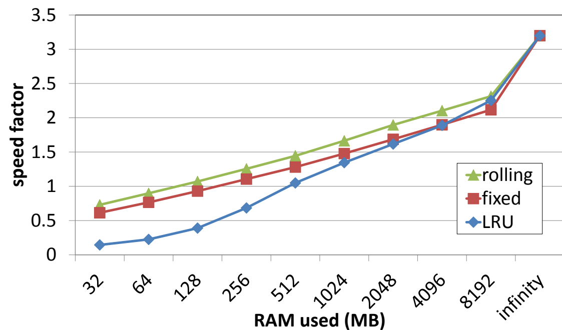


(a) Workgroup-cycle30

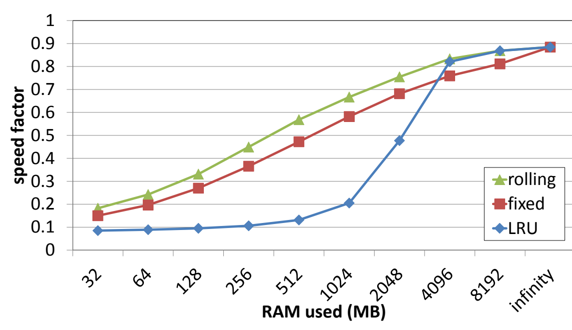


(b) 2year-cycle30

Figure 7: **Effect of varying segment size on deduplication and speed for selected capping ratios** for two data sets for two sizes of LRU caching and 4 sizes of segments. Deduplication and speed factors are the average of the last 20 backups. Each curve shows varying capping ratios (capping level as a ratio) of from left to right: infinity, 150/20, 130/20, 110/20, 90/20, 70/20, 50/20, 40/20, 30/20, 20/20, 15/20, 10/20, 5/20, and 2/20 containers per MB. Y-axes start at 6 to provide more detail.

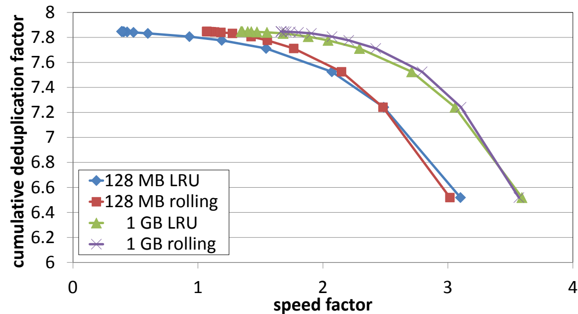


(a) Workgroup-cycle30

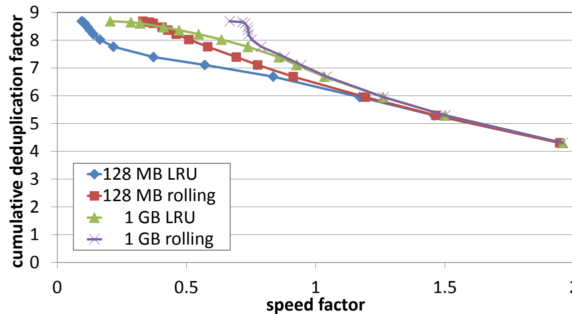


(b) 2year-cycle30

Figure 8: **Speed as RAM usage varies for 3 restore methods** for two data sets. Shown is the average speed for the last 20 backups. No capping was used.



(a) Workgroup-cycle30



(b) 2year-cycle30

Figure 9: **Comparing deduplication and speed as capping level varies between LRU and assembly** for two cache sizes and two data sets. Deduplication and speed values are the average of the last 20 backups. Each curve shows varying capping levels of from left to right: none, 250, 200, 150, 130, 110, 90, 70, 50, 40, 30, 20, 15, and 10 containers per 20 MB segment. Y-axis of Workgroup-cycle30 starts at 6 to provide more detail.

up the entire forward assembly area then sending it out whereas rolling sends out data incrementally using a ring buffer. As expected, because of its more efficient memory usage, rolling outperforms fixed by 7–23%, with greater improvement at smaller RAM sizes.

RAM used for LRU is simply the number of container slots in the cache times the container size (4 MB). For the forward assembly area methods, it is the size of the forward assembly area. In neither case do we count the chunk container I/O buffer (4 MB) or the recipe buffer (≈1–2% of the forward assembly area size).

The assembly methods produce substantial speedups over LRU for sub-4 GB RAM sizes. If we compare over what we regard as practical[2] RAM sizes per stream, namely 128 MB to 1 GB, rolling is 1.2–2.7X (Workgroup-cycle30) or 3.3–4.3X (2year-cycle30) faster. The above results assume no local compression of chunks or chunk containers; modifying LRU to use such compression may make it more competitive. If we generously assume compression reduces LRU RAM usage by 2, then for our practical RAM usages, rolling is 1.03–1.56X (Workgroup-cycle30) or 1.4–3.4X (2year-cycle30) faster.

At very high RAM sizes, LRU catches up with (rolling) assembly, presumably because pretty much everything cacheable has been cached at that point. It is likely that with larger individual backups LRU will not catch up to rolling until still higher RAM sizes. LRU outperforms fixed when everything fits in cache because fixed effectively evicts everything at the end of a slice and then has to reload it again.

4.6 Capping and assembly

Combining capping with assembly can produce even bigger speedups as Figure 9 shows. The benefit of assembly is greatest for smaller amounts of capping (i.e., allowing a larger number of containers) and tapers off as capping increases. We believe this is because increased capping reduces fragmentation and hence the needed "working set" size; although assembly makes more effective use of its memory than LRU, given enough memory both can hold the working set.

4.7 Container size

We have been using standard-size chunk containers of 4 MB per Zhu *et al.* [27]. If a system can read an extra 4 MB faster than it can start reading a new container then in theory it may pay to use a larger container size. This turns out to be largely false in practice when using

[2]Remember that these systems may have hundreds of concurrent streams in normal operation (often including restoring backups in order to write them to tape) so even 128 MB per stream adds up quickly.

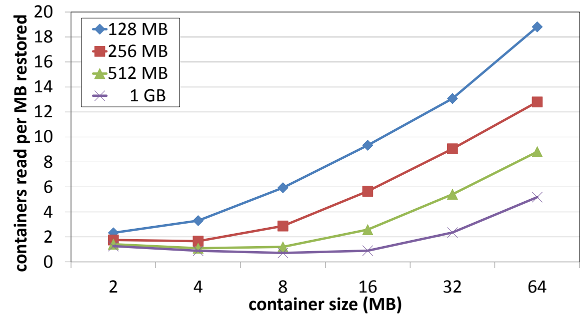


Figure 10: **Containers read using LRU as container size is varied** for Workgroup-cycle30 for 4 cache sizes. Containers read per MB restored is the average of the last 20 backups; no capping was used.

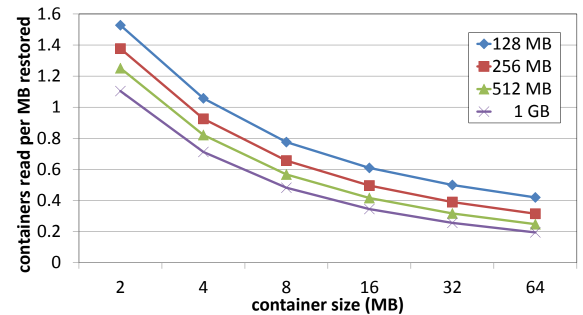


Figure 11: **Containers read using assembly (rolling) as container size is varied** for Workgroup-cycle30 for 4 cache sizes. Containers read per MB restored is the average of the last 20 backups; no capping was used.

LRU, however, because LRU cache performance quickly drops as we increase the container size: fewer large containers fit in the same size cache causing a higher miss rate, which in turn causes so many extra container reads that the savings from the data fitting in fewer containers is swamped by the cost of the extra reads. As an example, Figure 10 shows this for Workgroup-cycle30.

Assembly, by contrast does not suffer from this problem because it keeps individual chunks not entire chunk containers. This can be seen in Figure 11 where increasing the container size results in strictly fewer containers read per MB of restored data. However, reading larger containers takes longer so the optimal container size for restore depends on the ratio of cost between opening a container and sequentially reading a MB. For example, if we assume opening a container takes 20 ms (2 seeks @ 10 ms each) and reading 1 MB takes 1 ms (1 RAID group with 10 data drives @ 100 MB/s each) then we are indifferent between opening another container and reading another 20 MB. Under these assumptions if we ignore

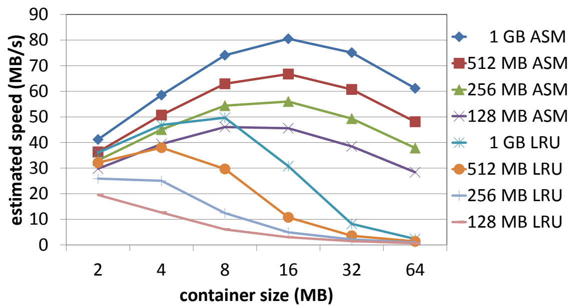


Figure 12: **Estimated speed for a sample system as container size is varied** for Workgroup-cycle30 for 4 cache sizes and assembly (rolling) and caching. Based on a system that reads at 1000 MB/s and opens a container in 20 ms, and the average of the last 20 backups; no capping was used and all containers were assumed to be full.

other considerations like ingest or deletion performance and assume all containers are full size, the optimal container size for restore without capping under assembly (rolling) is 8 MB for the 128 MB cache size and 16 MB for the 1 GB cache size (see Figure 12).

4.8 Ingest performance of capping

We did not simulate the I/O performance of backup ingestion but our experience with these systems is that the dominant cost of ingestion is reading chunk container indexes and/or on-disk full chunk index lookups. This involves a lot of reads per MB ingested, which are effectively random accesses, while writing out new containers is sequential. Thus, we do not expect the extra work of writing out a small amount of extra duplicates during ingestion to noticeably affect ingestion performance.

The number of index reads per MB ingested is proportional for most of these systems to the fragmentation of the backup being ingested *before* capping is applied. This is because any capping is applied after (provisional) deduplication is done to determine where chunks are already stored. Essentially, we need to load a chunk container index in order to determine if it is worth deduplicating against it. Capping may nonetheless improve ingest speed because it indirectly reduces the pre-capping fragmentation of the backup being ingested. This is because this fragmentation depends in part on the fragmentation of the previously stored backups, which capping does reduce.

Capping has been implemented in HP Storage's StoreOnce™ backup products, which are based on sparse indexing [12]. Sparse indexing deduplicates a segment at a time, deduplicating it against only a limited number of places found using sampling, and already has to cope with occasional extra stored copies of chunks due to the limited number of places that are searched. Capping, thus, fits easily into a sparse indexing system.

As originally described, sparse indexing indexes and searches *manifests* (recipe pieces). While sparse indexing's limitation on the number of places searched during ingest protects it from fragmentation during ingest (the number of places searched per MB is capped, guaranteeing a minimum ingest speed), it provides limited protection against fragmentation during restore. This is because each manifest may refer to many different containers and thus the segment may be deduplicated against far more containers than the limitation, each of which may have to be accessed during restore.

Given that with capping we are going to limit the number of containers we use anyway, it turns out to be more efficient to modify sparse indexing to index and search container indexes directly. That is, we look up samples of the segment to be deduplicated in a sparse index that maps them to the containers that contain them; from this information alone we can directly choose the top T containers to use (i.e., pick those containing the most samples). This efficiently limits the number of containers that must be accessed both during restore and ingestion, greatly improving both restore and ingest performance.

5 Future Work

Our current capping mechanism makes its decisions on a purely local basis, looking only at the current segment; while simple, this sometimes leads to suboptimal choices. We are investigating two methods of extending its context. First, we are exploring *adaptive capping* where the current capping level is adjusted at the start of each segment based on the actual number of containers used by recent segments so as to cap a running average of containers used rather than the number of containers used by the current segment. This produces better results with unevenly fragmented backups where the unused container access capacity from lightly fragmented spots can be used to improve deduplication of highly fragmented spots.

Second, we are investigating remembering at ingest time the containers that we have used for recent segments and not counting the use of these containers towards the cap. Such containers almost certainly will be in the cache at restore time given a reasonable cache size, and thus do not affect fragmentation at that scale. We hope this will give us the same performance as using a larger segment size without actually needing to increase our segment buffer space. We also plan to investigate the effect of merging small containers after deletion.

6 Related Work

Measuring fragmentation. Chunk fragmentation level (CFL) has been proposed as an effective indicator of restore performance [16, 17]. In our terms (generalizing to the non-LRU case), it is equal to the maximum[3] of our speed factor divided by the container size and 1. This ensures a maximum value of 1 for backups with high internal duplication, which otherwise would have values above 1. CFL's creators claim they do this because "the read performance is high enough" in that case but this makes CFL unable to predict the performance of such streams. For cases where the maximum is not hit, CFL like our speed factor is proportional to the actual restore speed if container size is held constant. If container size varies, its accuracy falls because it assumes the cost of reading a container is proportional to its size, which is only approximately true for typical systems and container sizes because of the substantial cost of opening a container relative to the cost of reading that container.

Trading off deduplication. A number of systems also trade off deduplication for lower chunk fragmentation and improved read performance. The earliest described, iDedup, does so in the context of primary storage deduplication [20]. Chunks there are ordinary fixed-sized filesystem blocks that may be shared rather than variable-sized chunks stored in container files. To reduce fragmentation, iDedup deduplicates a sequence of duplicate blocks only when its corresponding already stored blocks are sequentially laid out on disk and exceed a minimum length threshold. This limits how often seeks must be done when sequentially reading back data.

Nam *et al.* [17] suggest using a similar approach for backup systems: while a calculated fragmentation level is too high, only deduplicate sequences of duplicate chunks (ignoring any intermixed non-duplicate chunks) stored in the same chunk container if the sequence is long enough. In the presence of caching, this is suboptimal because it penalizes switching back to a recently used container. For example, it fails to deduplicate two sizable runs of duplicate chunks, each from a different container, if they are intermixed rather than occur one after the other. Capping and CBR (below) do not make this mistake because they ignore the order of chunks within a segment/context. Nam *et al.* do not report on how much deduplication is lost under their approach in order to reduce fragmentation by a given amount.

Context-based rewriting (CBR) [10] is the most similar to Capping in spirit. Modulo differences due to CBR assuming a single continuous storage region rather than discrete chunk containers, CBR essentially amounts to deduplicating a segment against a container if and only if it contains at least a minimum number of chunks from that segment. By cleverly adjusting this minimum over time, CBR limits the overall amount of deduplication lost to a small value (e.g., 5%). This comes at the cost of potentially unbounded fragmentation and restore time slowdown. Capping, by contrast, is designed to make it easy to guarantee a minimum restore speed—just set the capping level based on how many containers can be read per second—at the cost of potentially arbitrary deduplication loss. We believe that capping's trade-off is better because it is more important in practice for the latest backup to be able to be restored in a predictably-fast manner then to be able to store a few more old backups on the system. CBR has been only cursorily evaluated to date, being tested against only one extremely short data set with any real fragmentation (2X slowdown over 14 backups) [10].

Caching. Previous studies of chunk fragmentation in backup systems [10,16,17,23] have all used LRU caches. Wallace *et al.* [23], a study of backup workloads in production systems, reports the hit ratio for a read cache when restoring final backups, from which fragmentation can be estimated, for a wide range of cache sizes. Their numbers also show that increasing cache size substantially decreases measured fragmentation—in their case well into the terabyte cache size range. In addition to caching entire containers, they consider caching compression regions ($\approx$ 128 KB regions) or individual chunks. As expected, these produce substantially worse hit rates at all cache sizes and thus poorer restore performance.

Belady [4] shows that it is possible to construct a provably-optimal paging algorithm given perfect knowledge of future accesses. Cao *et al.* [5] consider how to integrate prefetching and caching of blocks in the presence of perfect future knowledge, giving four rules that an optimal such system must follow. They do not apply directly to our system because it does not use fixed-size blocks, but suggest we are suboptimal if memory copies are free because that would allow storing more chunks by using a sparse representation for the forward assembly area. Nam *et al.* [16] point out that information about future chunks needed is available in advance when restoring backups.

Patterson *et al.* [18] show how to prefetch and cache fixed-size file blocks in the presence of imperfect future knowledge provided by applications in the form of hints, showing how to dynamically allocate resources between hinted and unhinted accesses. Unlike us, they consider multiple outstanding prefetch requests, which improves performance and could be added to our method at the cost of more container buffers.

[3] The formula in the paper [17] says min but the accompanying text and figure 4 clearly indicate that max was intended.

Other work. Backlog [13] is an efficient implementation of metadata, specifically back-reference pointers, to ease defragmentation-like data re-organization tasks in write-anywhere file systems such as WAFL [8], Btrfs [1], and ZFS [2]. Cumulus [22] is a file backup system that uses large blocks so that many files can be packed into a block to reduce unused storage space. A cleaner minimizes fragmentation that occurs when files are deleted. SORT [21] suggests that taking ownership information into account when making routing decisions in multi-node deduplication systems may improve restore performance with little loss of deduplication.

Almost all file systems provide defragmentation utilities whose objective is to move objects so as to maximize contiguous free space [25]. Various heuristics have been proposed for grouping together, on disk, all data that are expected to be accessed together [7, 11, 14]. Locality-Improving Storage (ALIS) [9] is a storage system that clusters together frequently accessed data blocks while largely preserving the original block sequence to reduce random I/Os and increase read performance. Yu *et al.* [26] propose introducing several copies of data and then using metrics such as expected access time dynamically to reduce seek and rotational delay.

7 Conclusions

Poor restore performance due to chunk fragmentation can be a serious problem for inline, chunk-based deduplicating backup systems: if nothing is done, restore performance can slow down orders of magnitude over the life of a system.

Chunk fragmentation, like memory locality, is not a scalar value but rather a curve: one data set may have much more fragmentation at a lower scale than another yet have the same level of fragmentation at a higher scale. Because restore speed depends on the fragmentation at the scale of caching used and chunk fragmentation decreases as the scale it is measured at increases, restore performance can be increased by using more cache memory. Although expensive, adding more RAM continues to produce substantial improvements for real data sets well through the point where 8 GB of RAM is being used per stream.

The kind of caching matters as well. Switching from LRU to our new forward assembly area method provides substantially faster restore performance for the same RAM budget for practical RAM sizes (i.e., 128 MB to 1 GB). This has essentially no downside and demonstrates the power of using the perfect information about future accesses provided by recipes to keep only chunks that we are sure are going to be used in the near future. Because it keeps only these chunks, the forward assembly area method is able to take advantage of larger chunk containers than LRU, which may provide better performance on some systems.

Restore performance can also be improved by decreasing chunk fragmentation itself. Our container capping accomplishes this at the cost of decreased deduplication by selectively duplicating chunks to avoid accessing containers that we only need a small number of chunks from. This can result in substantial speedups for real data sets while giving up only a small amount of deduplication. Container capping is an ingest-time approach and does not require rearranging chunks later.

Given our results, we recommend that system designers use all available RAM for restoring a stream (including operating system cache space) for a single large forward assembly area and associated buffers. If the number of streams being restored at a time can vary, we recommend using more RAM per stream when fewer streams are being restored. Unless deduplication is at a great premium, at least a small amount of capping should be employed.

Acknowledgments

We would like to thank our shepherd, Fred Douglis, and the anonymous referees for their many useful suggestions.

References

[1] Btrfs project wiki. http://btrfs.wiki.kernel.org. Viewed February 10, 2012.

[2] Oracle Solaris 11 ZFS technology. http://opensolaris.org/os/community/zfs/. Viewed February 10, 2012.

[3] ASARO, T., AND BIGGAR, H. Data De-duplication and Disk-to-Disk Backup Systems: Technical and Business Considerations. *The Enterprise Strategy Group* (July 2007).

[4] BELADY, L. A. A study of replacement algorithms for virtual storage computers. *IBM Systems Journal 5*, 2 (1966), 78–101.

[5] CAO, P., FELTEN, E. W., KARLIN, A. R., AND LI, K. A study of integrated prefetching and caching strategies. In *Proceedings of the Joint International Conference on Measurement and Modeling of Computer Systems (SIGMETRICS)* (May 1995), pp. 188–197.

[6] ESHGHI, K. A framework for analyzing and improving content-based chunking algorithms. Tech. Rep. HPL-2005-30(R.1), Hewlett Packard Laboratories, Palo Alto, 2005.

[7] GANGER, G. R., AND KAASHOEK, M. F. Embedded inodes and explicit grouping: exploiting disk bandwidth for small files. In *Proceedings of the 1997 USENIX Annual Technical Conference* (Berkeley, CA, USA, 1997), USENIX Association, pp. 1–17.

[8] HITZ, D., LAU, J., AND MALCOLM, M. File system design for an NFS file server appliance. In *Proceedings of the Winter 1994 USENIX Technical Conference* (Berkeley, CA, USA, 1994), USENIX Association, pp. 235–246.

[9] HSU, W. W., SMITH, A. J., AND YOUNG, H. C. The automatic improvement of locality in storage systems. *ACM Transactions on Computer Systems 23*, 4 (2005), 424–473.

[10] KACZMARCZYK, M., BARCZYNSKI, M., KILIAN, W., AND DUBNICKI, C. Reducing impact of data fragmentation caused by in-line deduplication. In *Proceedings of the 5th Annual International Systems and Storage Conference (SYSTOR '12)* (Haifa, Israel, June 2012), ACM, pp. 11:1–11:12.

[11] KROEGER, T. M., AND LONG, D. D. E. Predicting file system actions from prior events. In *Proceedings of the 1996 USENIX Annual Technical Conference* (Berkeley, CA, USA, 1996), USENIX Association, pp. 319–328.

[12] LILLIBRIDGE, M., ESHGHI, K., BHAGWAT, D., DEOLALIKAR, V., TREZISE, G., AND CAMPBELL, P. Sparse Indexing: Large scale, inline deduplication using sampling and locality. In *Proceedings of the Eighth USENIX Conference on File and Storage Technologies (FAST '09)* (Feb. 2009), pp. 111–123.

[13] MACKO, P., SELTZER, M., AND SMITH, K. A. Tracking back references in a write-anywhere file system. In *Proceedings of the 8th USENIX Conference on File and Storage Technologies (FAST '10)* (Feb. 2010), pp. 15–28.

[14] MCKUSICK, M. K., JOY, W. N., LEFFLER, S. J., AND FABRY, R. S. A fast file system for UNIX. *ACM Transactions on Computer Systems 2*, 3 (1984), 181–197.

[15] MUTHITACHAROEN, A., CHEN, B., AND MAZIÈRES, D. A low-bandwidth network file system. In *Proceedings of the 18th ACM Symposium on Operating Systems Principles (SOSP '01)* (Banff, Alberta, Canada, October 2001), ACM Press, pp. 174–187.

[16] NAM, Y., LU, G., PARK, N., XIAO, W., AND DU, D. H. Chunk Fragmentation Level: An effective indicator for read performance degradation in deduplication storage. In *IEEE 13th International Symposium on High Performance Computing and Communications (HPCC '11)* (Banff, AB, Sept. 2011), pp. 581–586.

[17] NAM, Y. J., PARK, D., AND DU, D. H. C. Assuring demanded read performance of data deduplication storage with backup datasets. In *Proceedings of the 20th IEEE International Symposium on Modeling, Analysis, and Simulation of Computer and Telecommunication Systems (MASCOTS 2012)* (Washington DC, Aug. 2012).

[18] PATTERSON, R. H., GIBSON, G. A., GINTING, E., STODOLSKY, D., AND ZELENKA, J. Informed prefetching and caching. In *Proceedings of the Fifteenth ACM Symposium on Operating Systems Principles* (Dec. 1995), ACM Press, pp. 79–95.

[19] QUINLAN, S., AND DORWARD, S. Venti: A new approach to archival storage. In *Proceedings of the FAST 2002 Conference on File and Storage Technologies* (Monterey, CA, USA, January 2002), USENIX Association, pp. 89–101.

[20] SRINIVASAN, K., BISSON, T., GOODSON, G., AND VORUGANTI, K. iDedup: Latency-aware, inline data deduplication for primary storage. In *Proceedings of the 10th USENIX Conference on File and Storage Technologies (FAST '12)* (Feb. 2012), pp. 299–312.

[21] TAN, Y., FENG, D., HUANG, F., AND YAN, Z. SORT: A similarity-ownership based routing scheme to improve data read performance for deduplication clusters. *International Journal of Advancements in Computing Technology (IJACT) 3*, 9 (2011), 270–277.

[22] VRABLE, M., SAVAGE, S., AND VOELKER, G. M. Cumulus: Filesystem backup to the cloud. In *Proceedings of the 7th USENIX Conference on File and Storage Technologies (FAST '09)* (Feb. 2009), pp. 225–238.

[23] WALLACE, G., DOUGLIS, F., QIAN, H., SHILANE, P., SMALDONE, S., CHAMNESS, M., AND HSU, W. Characteristics of backup workloads in production systems. In *Proceedings of the 10th USENIX Conference on File and Storage Technologies (FAST '12)* (Feb. 2012), pp. 33–48.

[24] WIKIPEDIA. Backup rotation scheme. http://en.wikipedia.org/wiki/Backup_rotation_scheme#

Grandfather-father-son_backup. Viewed February 5, 2012.

[25] WIKIPEDIA. Defragmentation. http://en.wikipedia.org/wiki/Defragmentation. Viewed February 5, 2012.

[26] YU, X., GUM, B., CHEN, Y., WANG, R. Y., LI, K., KRISHNAMURTHY, A., AND ANDERSON, T. E. Trading capacity for performance in a disk array. In *Proceedings of the 4th conference on Symposium on Operating System Design & Implementation (OSDI)* (Berkeley, CA, USA, 2000), USENIX Association, pp. 243–258.

[27] ZHU, B., LI, K., AND PATTERSON, H. Avoiding the disk bottleneck in the Data Domain deduplication file system. In *Proceedings of the 6th USENIX Conference on File and Storage Technologies (FAST '08)* (San Jose, CA, USA, February 2008), USENIX Association, pp. 269–282.