Fingerdiff: Improved Duplicate Elimination in Storage Systems.

Deepak Bobbarjung
Purdue University
drb@cs.purdue.edu

Cezary Dubnicki
NEC Laboratories America
dubnicki@nec-labs.com

Suresh Jagannathan
Purdue University
suresh@cs.purdue.edu

Abstract

Minimizing the amount of data that must be stored and managed is a key goal for any storage architecture that purports to be scalable. One way to achieve this goal is to avoid maintaining duplicate copies of the same data. Eliminating redundant data at the source by not writing data which has already been stored, not only reduces storage overheads, but can also improve bandwidth utilization. For these reasons, in the face of today's exponentially growing data volumes, redundant data elimination techniques have assumed critical significance in the design of modern storage systems.

Intelligent object partitioning techniques identify data that are new *when objects are updated, and transfer only those chunks to a storage server. In this paper, we propose a new object partitioning technique, called* fingerdiff*, that improves upon existing schemes in several important respects. Most notably* fingerdiff *dynamically chooses a partitioning strategy for a data object based on its similarities with previously stored objects in order to improve storage and bandwidth utilization. We present a detailed evaluation of* fingerdiff*, and other existing object partitioning schemes, using a set of real-world workloads. We show that for these workloads, the duplicate elimination strategies employed by* fingerdiff *improve storage utilization on average by 25%, and bandwidth utilization on average by 40% over comparable techniques.*

1. Introduction

Traditional storage systems typically divide data objects such as files into fixed-sized blocks and store these blocks on fixed locations in one or more disks. Metadata structures such as file inodes record the blocks on which a file is stored along with other relevant file-specific information, and these inodes are themselves stored on fixed-sized disk blocks. Whenever an object is modified by either inserts, deletes or in-place replacements, the new blocks in the object are written to disk, and the metadata structure is updated with the new block numbers. However due to the inability to efficiently identify those portions of the object that are actually new in the latest update, a large part of existing data must necessarily get rewritten to storage. Thus, the system incurs a cost in terms of storage space and bandwidth whenever data is created or updated. This cost depends upon the storage architecture, but is proportional to the amount of new data being created or updated.

Recently, systems have been proposed that divide objects into variable-sized chunks (Henceforth, we will use the term "chunk" to refer to variable-sized data blocks and the term "block" to refer to fixed-sized data blocks.) instead of fixed-sized blocks in order to increase the amount of duplicate data that is identified[2, 3]. These techniques enjoy greater flexibility in identifying chunk boundaries. By doing so, they can manipulate chunk boundaries around regions of object modifications so that changes in one region do not permanently affect chunks in subsequent regions.

Our contributions in this paper are the following:

- We propose a new device-level variable-sized object partitioning algorithm, *fingerdiff*, that is designed to reduce the storage and bandwidth overheads in storage systems. *Fingerdiff* improves upon the duplicate elimination capability of existing data partitioning techniques, while also reducing storage management overheads.
- Using real-world workloads, we compare storage utilization and other storage management overheads of *fingerdiff* with those of existing techniques. We evaluate the effect of chunk sizes on the performance of these techniques. We show that *fingerdiff* improves upon the storage utilization of existing data partitioning techniques by 25% on average and bandwidth utilization by 40% on average.

Our solution relies on utilizing local computational and storage resources in order to minimize the cost of writing to scalable storage networks, by reducing the amount of new data that is written with every update. This also reduces the amount of data that has to be stored and maintained in the storage system, enabling greater scalability.

2. Fingerdiff

Our system model consists of a content-addressable storage backend that is essentially a variable-sized chunk store. Applications running on various clients periodically update data objects such as files to the store using an object server. This object server divides objects into variable-sized data chunks using *fingerdiff* and sends those chunks that are identified as new in the update to the chunk store.

The application will communicate to the object server apriori the exact specification of an object. The server then maintains in its *fingerdiff* driver, a separate tree for every specified object. Examples of an object specification are a single file, all files in one directory or any group of random files that the application believes will share substantial common data. All updates to a particular object will result in the driver comparing hashes of the new update with hashes in the corresponding tree.

Typical variable-sized techniques, also referred to as *content-defined chunking* (*CDC*) employ Rabin's fingerprints to choose partition points in the object. Using fingerprints allows *CDC* to "remember" the relative points at which the object was partitioned in previous versions without maintaining any state information. By picking the same relative points in the object to be chunk boundaries, *CDC* localizes the new chunks created in every version to regions where changes have been made, keeping all other chunks the same. As a result, *CDC* outperforms fixed-sized chunking techniques in terms of storage space utilization on a content-based storage backend[4].

However the variability of chunk sizes in *CDC* is rather limited. Most chunks are within a small margin of error of an *expected_chunk_size* parameter. Since this value determines the granularity of duplicate elimination, the storage utilization achieved by *CDC* is tied to this parameter. By decreasing this parameter, we can expect better duplicate elimination since new modifications will more likely be contained in smaller sized chunks. However it has been shown[5] that, reducing the *expected_chunk_size* to fewer than 256 bytes can be counter productive as the storage space associated with the additional metadata needed for maintaining greater number of chunks nullifies the effect of storage savings obtained because of a smaller average chunk size. Further, other than storage space overheads associated with maintaining metadata information about each chunk (e.g., the hash key map), more number of chunks can lead to other system dependent management overheads as well. For example, in a distributed storage environment where nodes exchange messages on a per chunk basis, creating a greater number of chunks is likely to result in more network communication during both reads and writes.

Fingerdiff is designed to overcome the tension between improved duplicate elimination and increased overheads of smaller chunk sizes by improvising on the concept of variable-sized chunks. It does this by allowing larger flexibility in the variability of chunk sizes. Chunks no longer need to be within a margin of error of an expected chunk size. The idea is to reduce chunk sizes in regions of change to be small enough to capture these changes, while keeping chunk sizes large in regions unaffected by the changes made.

For this purpose, *fingerdiff* locally maintains information about *subchunks* - a unit of data that is smaller than a chunk. Subchunks are not directly written to the storage engine. Instead a collection of subchunks are coalesced together into chunks whenever possible and then the resultant chunk is the unit that is stored. *Fingerdiff* assumes an expected subchunk size parameter (*expected_subchunk_size*) instead of the expected chunk size parameter used in *CDC*. After calling a *CDC* implementation that returns a collection of subchunks, *fingerdiff* seeks to coalesce subchunks into larger chunks wherever possible. A *max_subchunk_size* parameter is used to determine the maximum number of subchunks that can be coalesced to a larger chunk.

For example, if an object is being written for the first time, all its subchunks are new and *fingerdiff* coalesces all subchunks into large chunks, as large as allowed by a *max_subchunk_size* parameter. If a few changes are made to the object and it is consequently written to the store again, *fingerdiff* consults a local client-side lookup and separates out those subchunks that have changed. Consecutive new subchunks are coalesced into a new chunk and written to the store. Consecutive old subchunks are recorded as a chunk or a part of a chunk that was previously written. To incorporate the notion of chunk-parts, *fingerdiff* modifies the metadata structures required to remember the chunks associated with an object. Along with the hash of a given chunk, metadata structures will also record the offset of the chunk-part within the chunk and its size.

The key intuition here is that a *fingerdiff* implementation can assume a lower *expected_subchunk_size* value than the expected chunk size assumed in an implementation of *CDC*. This is because after calling *CDC*, *fingerdiff* will merge the resultant subchunks into larger chunks wherever possible before writing them to the store. Therefore *fingerdiff* can improve duplicate elimination without incurring the overheads of small-sized chunks. Further details of the *fingerdiff* algorithm and our implementation can be found in [1].

3. Experimental Framework

An important goal of this work is to measure the effectiveness of chunking techniques including *fingerdiff* in eliminating duplicates in a content addressable storage sys-

tem with specific emphasis on applications that write consecutive versions of the same object to the storage system. But apart from storage space utilization, we also measured the bandwidth utilization, the number of chunks generated and other chunk related management overheads for different chunking techniques. In this paper, we present the results for storage and bandwidth utilization. More detailed results can be found in [1]

We used three classes of work loads to compare *fingerdiff* with *CDC*. The first one, **Sources**, contains a set of consecutive versions of source code of real software systems. This includes versions of gnu gcc, gnu gdb, gnu emacs and the linux kernel. The second class, **Databases** contains periodic snapshots of information about different music categories from the Freedb database obtained from www.freedb.org. Freedb is a database of compact disc track listings that holds information for over one million CDs. For our experiments, we obtained 11 monthly snapshots of *freedb* during the year 2003 for the jazz, classical and rock categories. These snapshots were created by processing all the updates that were made each month to the *freedb* site. The third class, **Binaries** contains executables and object files obtained by compiling daily snapshots of the *gaim* internet chat client being developed at http://sourceforge.net taken from the cvs tree for the year 2004.

We use the following terminology to define *CDC* and *fingerdiff* instantiations:

- A *cdc-x* instantiation is a content defined chunking strategy with an *expected_chunk_size* of *x* bytes;
- A *fd-x* instantiation is a *fingerdiff* instantiation with a *expected_subchunk_size* of *x* bytes and *max_subchunk_size* of 32 KB.

3.1. Total storage space consumed

We calculate storage utilization of a chunking technique instantiation for a particular benchmark by storing consecutive versions of the benchmark after chunking it into variable sized chunks using that instantiation.

The total storage space is calculated by adding the space consumed by the benchmark data on the chunk store backend (*backend storage utilization*) and the lookup space required for a given benchmark on the object server (*local storage utilization*). The backend storage space consists of data and metadata chunks for the benchmarks along with the cost of storing a pointer for each chunk. We calculate this cost to be 32 bytes(20 bytes for SHA-1 pointers plus 12 bytes to maintain variable-sized blocks). The local lookup space is used on the driver to support *fingerdiff* and *CDC* chunking. This lookup is a tree that maps hashes of subchunks of an object to information about that subchunk. This tree resides in disk persistently, but is pulled into memory when an object is being updated and has to be partitioned. As can be expected, this tree grows as more versions of the object are written to the store. We measure the size of the tree for all our *fingerdiff* instantiations. The lookup space is measured as the total space occupied by the lookup tree for each benchmark in the local disk.

Note that if a replication strategy is used for improved availability, the *backend storage utilization* will proportionately increase with the number of replicas but the *local storage utilization* will remain constant for any number of replicas.

We limit the *CDC* instantiations for which we show results to *cdc-2k*, *cdc-256*, *cdc-128*, *cdc-64* and *cdc-32*. We compare these with five *fingerdiff* instantiations namely *fd-2k*, *fd-256*, *fd-128*, *fd-64* and *fd-32*. Note that many more instantiations are possible, but we limit our presentation in order to reduce the clutter in our tables and graphs, while ensuring that the broad trends involved with changing chunk sizes are clear.

The storage space consumed by each chunking technique reflects the amount of storage space saved by leveraging duplicate elimination on the store. The technique which best utilizes duplicate elimination can be expected to consume the least storage space. Table 2 compares the total (backend+local) storage utilization achieved on account of duplicate elimination after individually storing all our benchmarks for all ten chunking instantiations.

For all benchmarks (except *gaim*) either *fd-32* of *fd-64* consumes the least and *cdc-32* the most storage. In case of *gaim* *fd-256* consumes the least storage. Among the *CDC* instantiations, either *cdc-128* or *cdc-256* gives the best storage utilization. Decreasing the chunk size of *CDC* to 64 or 32 increases total storage consumption for all benchmarks.

However for most benchmarks, reducing the expected subchunk size of *fingerdiff* to 64 or 32 bytes helps us to increase the granularity of duplicate elimination without incurring the storage space overheads of too many small chunks. The last column (% savings) in table 2 gives the savings achieved by the best *fingerdiff* (in most cases *fd-32* or *fd-64*) instantiation over the best *CDC* instantiation (either *cdc-128* or *cdc-256*). In spite of the large number of hashes for subchunks maintained in *fingerdiff* drivers, *fingerdiff* improves the storage utilization of the best *CDC* . For example, *fd-32* improves backend storage utilization of the best *CDC* by a significant percentage for all benchmarks that we measured. This improvement varied from 13% for *gaim* to up to 40% for *gcc*. The last row in table 1 gives the total storage consumed after writing all the benchmarks to the chunk store. Here, we observed that *fd-64* gives the best storage utilization. It improves upon the storage utilization of the best *CDC* technique (*cdc-128*) by 25%.

benchmark	*cdc-2k*	*cdc-256*	*cdc-128*	*cdc-64*	*cdc-32*	*fd-2k*	*fd-256*	*fd-128*	*fd-64*	*fd-32*	*% saving*
Sources											
gcc	1414	866	828	859	979	1400	799	680	579	498	40
gdb	501	363	344	358	500	498	336	293	255	255	26
emacs	327	258	259	281	457	324	239	220	199	221	25
linux	1204	708	629	692	985	1195	644	520	469	543	23
Databases											
freedb	396	348	369	442	644	370	396	317	291	290	17
Binaries											
gaim	225	245	301	447	527	213	196	208	244	246	13
Total	4067	2788	2731	3079	4090	3999	2611	2238	2038	2052	25

Table 1. Comparison of the total storage space consumed (in MB) by the ten chunking technique instantiations after writing each benchmark on a content addressable chunk store. The last column gives the % savings of the best *fingerdiff* technique over the best *CDC* technique for each benchmark.

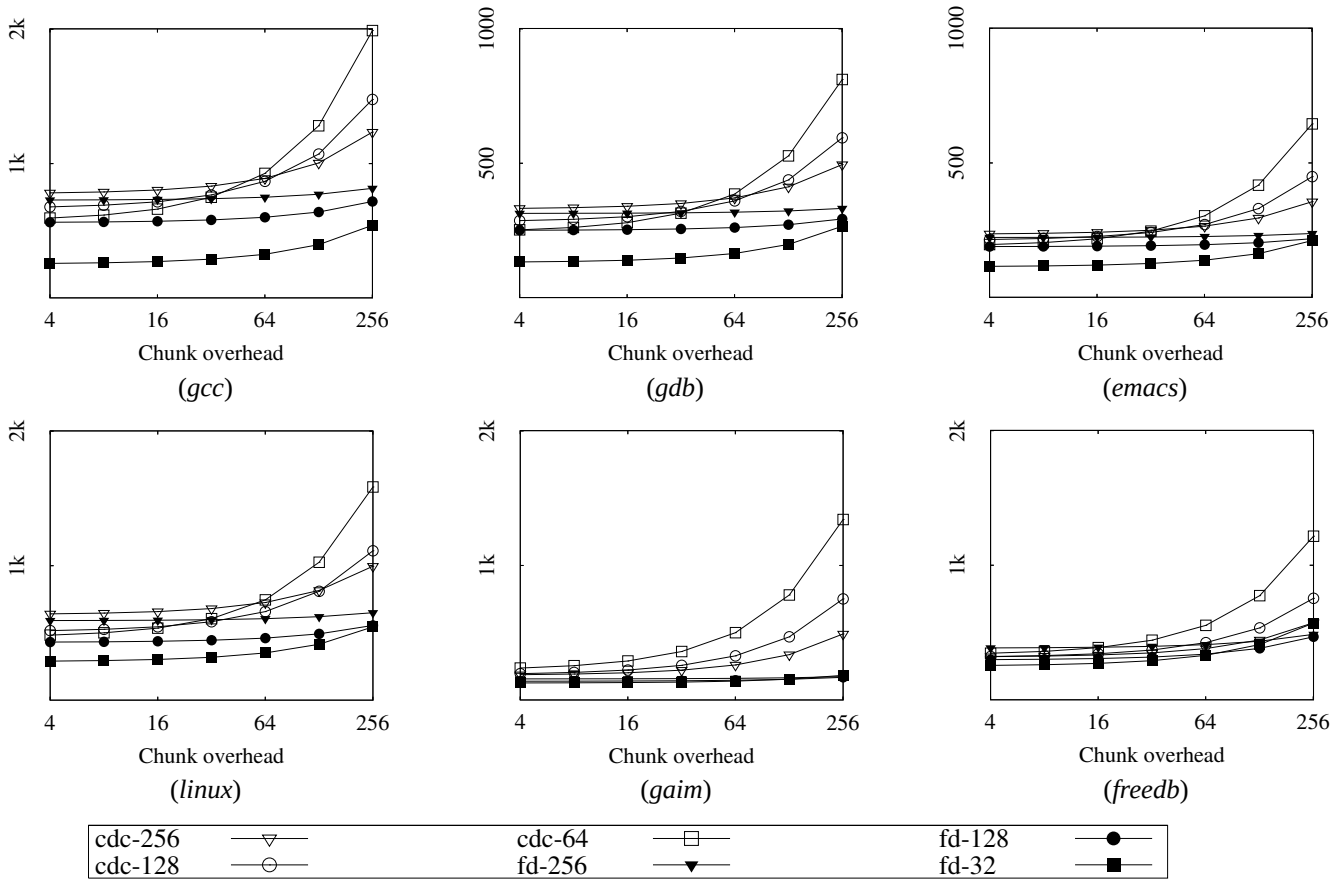

Figure 1. Comparison of the total network traffic (in MB) consumed by six of the ten chunking technique instantiations after writing each benchmark on a content addressable chunk store. The X-axis of each graph is a log plot which gives the chunk overhead; i.e the overhead in bytes associated with transferring one chunk of data from the driver to the chunk store. The network traffic measured is between the object server and the chunk store. The Y-axis gives the total network traffic generated in MB after writing each benchmark to the chunk store.

3.2. Total network bandwidth consumed

Once the object server identifies the chunks that are new in each update, it sends each new chunk to the chunk store along with necessary metadata for each chunk. In our model, this metadata must include the size of the chunk (necessary to support variable sized chunks), imposing an overhead of 4 bytes for every chunk that is sent. Based on this we calculated the average bandwidth savings of the best *fingerdiff* technique over the best *cdc* technique for all benchmarks to be 40%.

However other models might require extra metadata. For example, a model akin to the the low bandwidth file system[3] where the server also maintains object information might require the client to send the file descriptor along with each chunk. Peer to peer architectures might require the client to check the existence of each hash with the chunk store[2]. In general, chunking techniques that generate more chunks will send more traffic over the network, the exact amount of which will depend on the network protocol and the system model. Figure 7 illustrates the amount of network bandwidth consumed by different instantiations for all benchmarks for a varying amount of metadata traffic overhead per chunk. For each benchmark the per-chunk overhead is varied from 4 bytes to 256 bytes. Observe that for all benchmarks, a chunk overhead as low as 4 bytes results in substantial bandwidth savings for the best *fingerdiff* instantiations over all the *CDC* instantiations. Note that to preserve clarity of our graphs, we plot only 3 instantiations from *fingerdiff* and 3 from *CDC*. However note that we do plot *cdc-128* and *cdc-256* which formed the most efficient *CDC* instantiations for all benchmarks. Also observe that the instantiations that generate more number of chunks (i.e the *CDC* instantiations) consume more bandwidth as the per-chunk overhead is increased from 4 to 256. We conclude that *fingerdiff* substantially improves upon the bandwidth utilization of *CDC*.

4. Conclusions

Existing object partitioning techniques cannot improve storage and bandwidth utilization without significantly increasing the storage management overheads imposed on the system. This observation motivated us to discover a chunking technique that would improve duplicate elimination over existing techniques without increasing associated overheads.

We have proposed a new chunking algorithm *fingerdiff* that improves upon the best storage and bandwidth utilization of *CDC* while lowering the overheads it imposes on the storage system. We have measured storage and bandwidth consumption along with associated overheads of several *CDC* and *fingerdiff* instantiations as they write a series of versions of several real-world software systems to a content addressable store. For both these benchmarks, we show that *fingerdiff* significantly improves the storage and bandwidth utilization of the best *CDC* instantiation while also reducing the rate of increase in storage overheads(fewer number of chunks were written to the chunk store by the best *fingerdiff* instantiation than the best *cdc* instantiation for all our benchmarks).

Our contention is not that a particular *fingerdiff* technique is the best choice in all content based storage engines. But, by allowing for greater variability of block sizes, and by being able to better localize the changes made to objects into smaller chunks, *fingerdiff* is able to minimize the size of new data introduced with every update, while keeping the average size of all chunks relatively large. This in turn allows it to provide the best storage and bandwidth utilization for a given amount of management overhead.

References

[1] D. Bobbarjung, C. Dubnicki, and S. Jagannathan. A technique to detect data duplicates. Technical report IR No. 05006, NEC Laboratories America, Inc., 2005.

[2] L. Cox, C. Murray, and B. Noble. Pastiche: Making backup cheap and easy. In *Proceedings of Fifth USENIX Symposium on Operating Systems Design and Implementation*, Boston, MA, December 2002.

[3] A. Muthitacharoen, B. Chen, and D. Mazieres. A low-bandwidth network file system. In *Symposium on Operating Systems Principles*, pages 174–187, 2001.

[4] C. Policroniades and I. Pratt. Alternatives for detecting redundancy in storage systems data. In *Usenix Annual Technical Conference*, pages 73–86, 2004.

[5] L. L. You and C. Karamanolis. Evaluation of efficient archival storage techniques. In *proceedings. of the 21st IEEE Symposium on Mass Storage Systems and Technologies (MSST)*, April 2004.