Content-Based Block Caching

Charles B. Morrey III and Dirk Grunwald
Dept. of Computer Science
University of Colorado, Boulder
Boulder, CO 80309-0430

March 13, 2006

Abstract

In this paper we propose a novel cache management mechanism termed the Content-Based Buffer Cache. The Content-Based Buffer Cache (CBBC) attempts to maintain a single copy of any block in memory according to its contents. In the presence of repeated content, this mechanism increases the effective size of the buffer cache. Overheads for maintaining this extra state information are small and bounded, providing an overall system performance improvement. Additionally, we eliminate writes to blocks where the new and old content are the same, reducing pressure on the I/O subsystems in the presence of these "Silent Writes".

We have logged traces of block-level disk access for a group of workstations over a several month period using a modified Linux kernel designed to boot off of an iSCSI target. We have analyzed single client access, as well as multiple client access to distinct logical disks using a unified block cache. There is significant replication of content and significant numbers of "Silent Writes" within a single workstation trace, improving the Content-Based Buffer Cache read hit rate as much as 80% over the traditional buffer cache design. We have also found that there is significant sharing of content between *disks, which benefits content-based caching performance in the presence of a unified cache. For our workloads, these results indicate that content-based buffer caches dramatically improve I/O performance when used to manage a cluster of similar storage.*

1 Introduction

Most, if not all, modern magnetic disk-based storage systems have some sort of DRAM buffer cache that is used to improve performance of the I/O system. These buffer caches reduce disk I/O by caching recently used blocks and decrease write latency by buffering writes to non-volatile storage. In addition, some storage systems use these buffer caches to enable disk prefetching of content that is likely to be referenced in the future. In large storage servers, the disk buffer cache may be very large, ranging from 4-30GB. Most operating systems also use a buffer cache to cache disk blocks; the size of that buffer is usually adjusted dynamically to fill memory that is otherwise unused. In both storage servers and operating systems, the size and management policy of the buffer cache is critical. If the cache is too small or makes poor use of the buffer cache items, there is little performance improvement.

Traditional buffer caches use an *address* to locate or index blocks in memory. A large storage server may export several logical disks or logical units ("LUNs"). These LUNs may either share a common buffer cache or have private caches. In either case, the blocks are simply addressed by their location on the disk, and all cache maintenance is done using that location as a unique identifier of the block.

In this paper we propose a novel cache management and indexing mechanism termed the *Content-Based Buffer Cache* (CBBC). The Content-Based Buffer Cache indexes blocks in two ways – using the traditional address based indexing *and* a second indexing mechanism that summarizes the content of a buffer cache block. The Content-Based Buffer Cache maintains a single copy of any block in the cache according to its contents. For reads, the Content-Based Buffer Cache maintains address information to quickly locate a block, if it exists, in the cache. To reduce evictions, when a write occurs the cache addresses the block by its contents, and only evicts something if the written block doesn't exist in the cache.

If many blocks have the same content, then storing only one copy of the block in the cache increases the cache's effective size. More cache space reduces the number of required cache evictions and thereby improves cache hit rate for reads. Additionally we can avoid writing to the

disk if we know that the contents already on the disk are identical to the content being written. We call these writes "silent writes" because they can be safely eliminated without changing the state of the disk contents.

The Content-Based Buffer Cache can improve buffer cache utilization if identical data is written to different locations within the same logical disk. It can also improve performance if the same data is written to locations on *different* logical disks. For example, consider a cluster of clients connected to several LUNs of a storage server in a storage area network (SAN). If those clients have similar content (*e.g.* access similar software or data), the content-based buffer cache can maintain a single copy of that data across all the clients.

Significant research has been done recently using content tracking to improve performance of systems by trading CPU time to compute hashes for other system resources that are more precious ([1], [2], [3]). The Content-Based Buffer Cache trades computation and a *small* amount of cache memory for its improved read hit rate, and elimination of silent writes.

The major contributions of this research include: a novel cache management mechanism, Content-Based Buffer Caching, which improves read hit rate by as much as 80%, while maintaining small computational overheads, an infrastructure for logging complete system block-level disk access (including swap partitions and the root system partition) on a centralized server, and an analysis of traces obtained over a several month period for a set of workstations.

The remainder of this paper is organized into five sections. In Section 2 we examine the detailed cache implementation and the logging mechanism and implementation. Section 3 describes and analyzes our workloads. We then discuss our results in Section 4; in Section 5 we detail related work and how it compares to our current research. Finally, we offer some conclusions in Section 6.

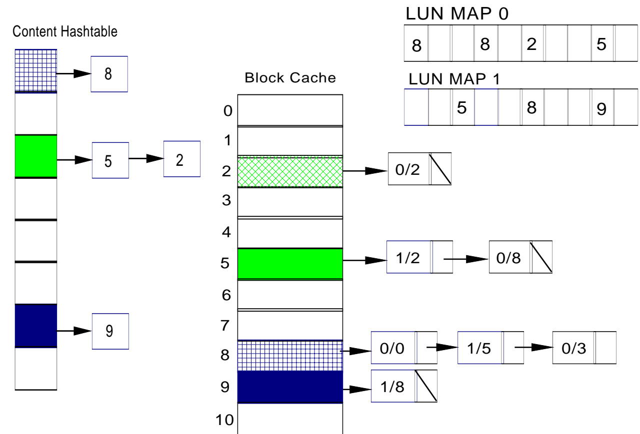


Figure 1: Data structures used in the cache simulator. Client accesses are handled by doing a lookup in the LUN map at the appropriate offset, possibly updating the content of the block cache and content hash table, and then serving or writing the content in the block cache. LUN maps hold only pointers to blocks in the block cache. content hash table entries are a chained list of pointers to all blocks in the block cache which hash to that hash entry. The block cache entries contain the blocks themselves, and a list of all LUN/offsets which point to that block. Additionally (not shown), LRU information is kept in the block cache, so block evictions can be done.

2 Detailed Design and Implementation

We implemented the design of the Content-Based Buffer Cache as a simulator to better explore the tradeoffs between computation and space overheads of the content tracking, and read hit rate. We developed two simulators. The first uses counts of specific events to estimate performance. The second uses the DiskSim [4] simulator to provide more accurate timing estimates. DiskSim has a cache module that models offset caches used in modern storage systems. This

cache module was modified to implement the CBBS data structures, while still maintaining accurate disk timing information. We used disk traces obtained over the course of several months of logging to drive these simulations.

2.1 Cache Simulator

The cache simulator takes as input a trace of block accesses (reads and writes) to one or more LUNs, a block cache size, and the fraction to be devoted to content management. The simulator reads the trace files in temporal order, performs the buffer cache algorithms described below and records statistics about specific events.

There are three main data structures created, as shown in Figure 1. The first data structure is the LUN map. Each instance of the LUN map represents the client system's view of that LUN. The LUN map can be implemented using any *map* datatype; it is currently implemented using a binary tree, with the leaves containing pointers to blocks in the block cache. A traditional offset cache has each LUN map entry point to a unique block cache entry. However, with content caching, more than one LUN map entry can point to a single block in the block cache. By recognizing when blocks with different LUN offsets have the same content, the cache can store a reference to a single cache block in each LUN map entry.

The size of each LUN map is determined by the number of valid entries in the block cache, and the access pattern to that LUN. However, for content caching, the LUN map could theoretically consume all cache space with LUN map entries pointing to a single cache block. Currently, the DiskSim implementation of the CBBC dynamically tunes this tradeoff between cache blocks and LUN map references automatically using the LRU algorithm. When the cache is full and a new LUN map reference to an existing cache block is needed, the LRU cache block's LRU LUN map reference is discarded. If LUN map reference eviction causes all references to the LRU cache block to be freed, then that cache block is freed as well, and its space becomes available either for additional references, or reallocation as a cache block.

The second data structure is the block cache itself. Each entry in the block cache consists of a 512 byte block, and a list of LUN/offset pairs, (matching the LUN map references described above), that are currently referencing that block. These block cache entries are maintained in an LRU queue of blocks that, as described above, is used when evicting blocks or LUN map references; it is also possible to use LRU approximations such as clock algorithms, which may

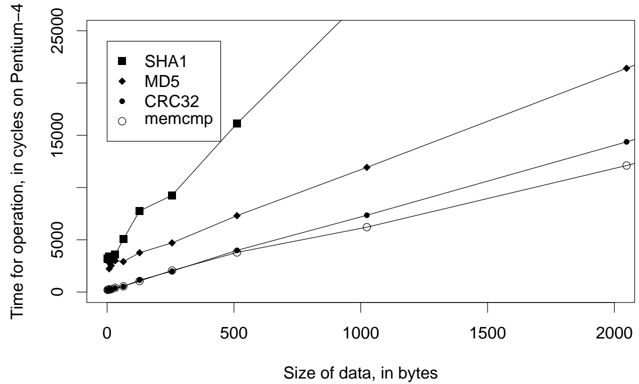


Figure 2: Time for various memory operations on a 2.4Ghz Pentium 4 CPU, in number of processor cycles. For small memory regions, the CRC32 hash is about as fast as a memory copy and significantly faster than the MD5 or SHA1 hashes.

be more space efficient. We would like to explore this in future work. The size of the block cache bounds the number of blocks and references that can remain in memory, affecting cache hit rate, and therefore performance. The ratio of LUN/offset pairs to blocks is currently controlled entirely by the LRU algorithm, and so by the request stream. We will also be exploring fixing the ratio, or bounding the maximum and minimums of the ratio to determine its effect on overall cache performance.

Finally, the third data structure is the content hash table. The content hash table only indicates that blocks *may* contain the same values; we use a full bit-wise comparison of the block to insure this is the case. If multiple blocks have different contents, but the *same* hash value, each block is stored in a different block cache entry, and a pointer to each block cache entry is stored in the chained list for that hash value. Therefore, whenever new a block arrives, it is directly compared with all other blocks that hash to the same hash value before it is inserted. Architecting the cache in this manner allows us to control the tradeoff between the number of full block comparisons which must be done on insertion, and the cache space devoted to the content hash table. By making bitwise comparisons to verify potentially matching blocks indicated by hash values, we are able to use inexpensive hash functions. Figure 2 shows the time (measured in processor cycles) needed to hash or compare regions of memory on a 2.4 Ghz Pentium 4. For a 512-byte memory block, the CRC32 hash takes about the same time as a memory comparison (about 3500 cycles) and about half the time of an MD5 hash and about a quarter of the time of the SHA1 hash. Since we do not use the

hash value without checking for collisions of content, using the CRC32 algorithm saves computational overhead versus MD5 or SHA1.

Checking a new block for insertion is done by generating a content hash of the block, and looking the hash up in the content hash table. If there is no matching hash table entry, then a new cache block to store the data is allocated or reclaimed by LRU eviction, the LUN map is updated to point to that block, and the content hash table is updated with a new entry for the new block. If there is a matching hash table entry, then each block pointed to by the chained list of pointers is compared bitwise with the new block for a match. If there is a match, then that block content already exists in the cache and the LUN map is updated to point to that new block. If there is no match, then a new cache block to store the block is allocated or reclaimed, the LUN map is updated to point to that block, and the content hash table is updated with an additional entry for that content hash value which points to the new block.

Reclaimation of blocks is done by evicting blocks from the block cache using LRU order. Eviction is done by invalidating each LUN map entry that points to the evicted block (via the list of LUN/offset pairs kept with the block in the block cache), and then removing the evicted block from the content hash table entry's list of block pointers.

The bound on the time needed to insert a new block into the cache is equal to the time necessary to calculate a CRC32 hash plus the time required to compare each block that collides in the hash table. Figure 4 is a plot of the number of comparisons required versus different hash table sizes. As the hash table consumes more of the available cache space (i.e. the hash table gets a larger portion of a fixed cache size), the number of comparisons required goes down. Generally speaking, the number of these comparisons is directly affected by the size of the hash table, since larger hash tables will have generally fewer collisions given uniform hash function distribution. However, the overhead of comparing blocks may not be significant – CPU computation is thousands of times faster than a disk access (and the gap continues to widen). Assuming a random disk access time of 10ms, more than 6,000 512 byte block memory comparisons can be done in the time required for a single disk access. (Assuming a sequential transfer rate of 30 MB/s, the disk can transfer one block every 16us, which is still greater than 10 comparisons per block transferred on a 2.4Ghz Pentium 4.)

The eviction time for clean blocks is bounded by the time required to invalidate all LUN map entries. (e.g. on the order of the amount of sharing for that block times the cost of a memory operation). Content-Based Buffer Caching reduces the number of dirty blocks by tracking silent writes and not marking those blocks dirty. The current DiskSim simulator implements write-thru caching, so not marking blocks dirty implies directly reducing the number of writes seen by the disk.

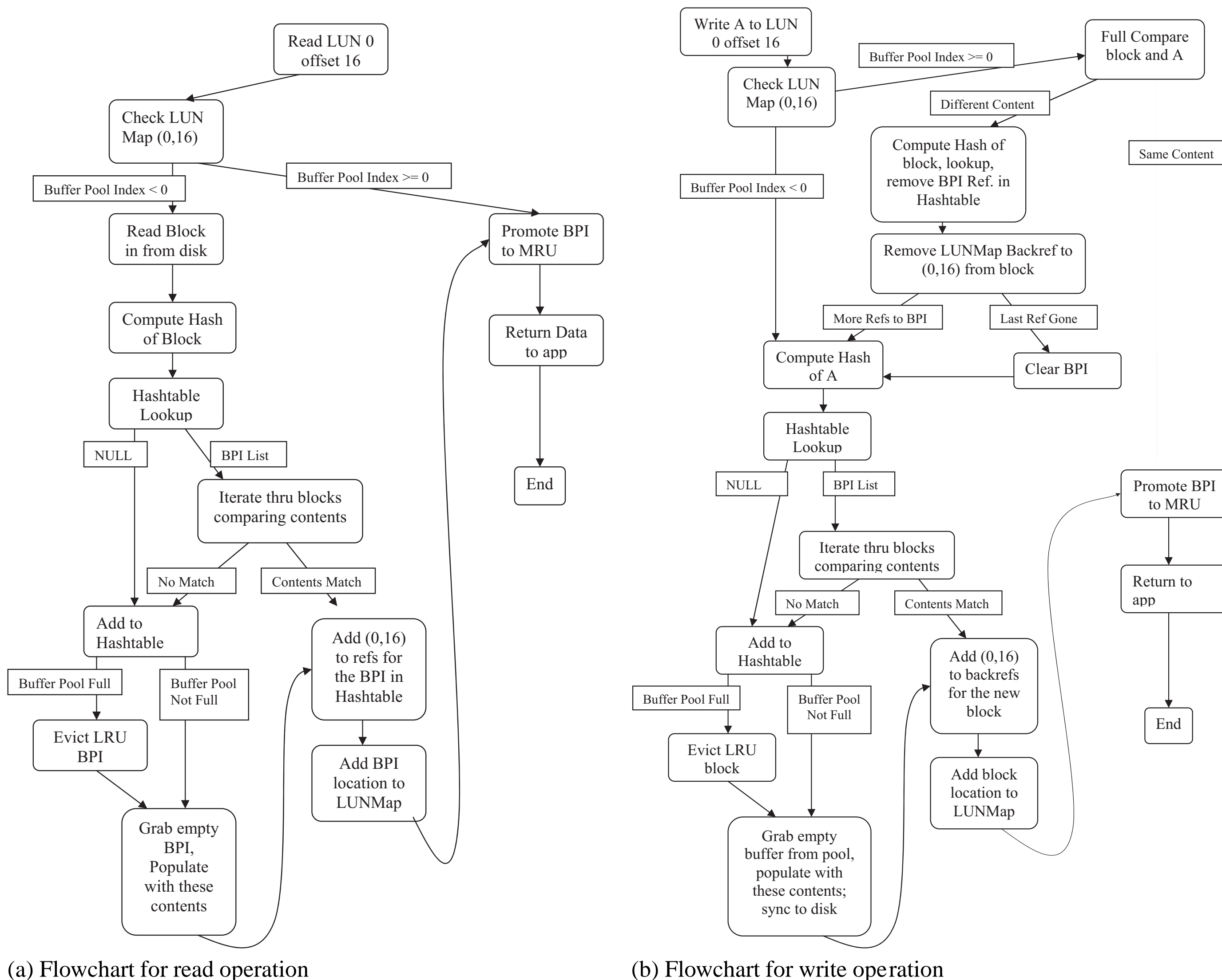
Figures 3 (a) and (b) describe the read and write operation pictorially. The operation of CBBC for the case of a read is described as follows:

The simulator checks the LUN/offset to see if there is a valid cache entry for that offset. If there is a valid cache entry, then that block in the block cache is returned, a "*read hit*" is recorded, and that entry is moved to the top of the LRU queue. If there is no valid cache entry, then the block is read from disk and its content hash is computed. If this value is in the hash table, then the contents of each block is compared with the fresh block looking for a match.

If there is a match, then that block is updated with the LUN/offset reference, it is moved to the head of the LRU queue, a "*read miss, content hit*" is recorded, and the block is returned. If no match is found, then a new block is inserted into the block cache (*i.e.* possibly evicting a block if the cache is full) the LUN map is updated with the block cache offset for the new block, the content hash entry list is updated with the new block offset, a "*read miss, content miss*" is recorded, and the block is returned.

The case for a write has more states: When the write occurs, if there is already a valid cache entry at that LUN/offset, the contents of the old block are compared with the new block. If they are the same, then a silent write is recorded, and nothing else occurs. If the contents differ, then the number of LUN/offset references for that block is checked. If this LUN/offset is the only reference to this block, then the block can be replaced in place and a private write is recorded. The old entry in the content hash must be removed, and the hash of the new block must be checked. If there is an entry for that hash, then those blocks must be compared with the new block. If the contents of any of those blocks match, then their LUN/offset lists are updated with the new block's LUN/offset, and the block cache entry for the old block is discarded. If none of the blocks match, or if there are no blocks with which to compare, then the block cache entry for the old block is reused. The key feature of a private write is that no block with other LUN/offset references is evicted, although a cache entry might be freed because of a content hit.

If there is a valid cache entry at the LUN/offset for the write, but the contents do not match and there is more than one reference to the block in the block cache, then the content hash table is searched to see if the block is somewhere



(a) Flowchart for read operation

(b) Flowchart for write operation

Figure 3: Flowcharts showing the operations on the various data structures used to maintain the content buffer pool

else in the cache. If the block is elsewhere in the cache, then a "*LUN map hit, different content, multiple reference, content hit*" is recorded, the block cache entry for the old block is updated to remove the reference to the LUN/offset, and the block cache entry for the new block is updated to reference the LUN/offset. No eviction is necessary in the preceeding case. However, if the content is not found elsewhere in the cache, then a "*LUN map hit, different content, multiple reference, content miss*" is recorded, a block is evicted if the cache is full, and the new block is inserted into the block cache.

If there is no valid cache entry at the LUN/offset, then the content hash of the new block is computed and a lookup is performed in the content hash table. If the new block's content hashes to an entry with valid block pointers, then the content of each of the blocks is compared with the new block to see if there is a match. If one of the blocks matches, then its LUN/offset list is updated, it is moved to the head of the LRU queue, and the LUN map is updated to point to that block. This is a "*LUN map miss, content hit*", and does not require an eviction. If, on the other hand, there is no valid entry in the hash table, or none of the blocks matches, then a new entry is added to the content hash table, and a new block is inserted into the block cache, evicting the least recently used block if the cache is at capacity. This is a "*LUN map miss, content miss*".

The statistics kept are outlined in Table 1. The various statistics besides read hits and write hits are used to explain improved hit rates and detection of silent writes in Section 4.

Read & Write Statistics Kept	Costs a Disk I/O	Causes an Eviction
Read Hit	No	No
Read Miss, Content Hit	Yes	No
Read Miss, Content Miss	Yes	Yes
Write Hit in LUN map, Different content, Multiple References, Content Miss	Yes	Yes
Write Miss in LUN map, Different content, Multiple References, Content Miss	Yes	Yes
Write Hit in LUN map, Different content, Multiple References, Content Hit	Yes	No
Write Miss in LUN map, Different content, Multiple References, Content Hit	Yes	No
Write Hit in LUN map, Different content, Single Reference, Private Write	Yes	No
Write Hit in LUN map, Same content, Silent Write	No	No

Table 1: Statistics maintained for the cache simulator. Multiple references implies that there are multiple LUN/offset pointers pointing to the block which is at the current LUN/offset. A Content Hit is a block found elsewhere in the buffer cache using the content hash table. Accesses which cause evictions only do so if the cache is at capacity.

2.2 Disk Logging Mechanism and Implementation

To test the hypothesis that content similarity exists at the block level, and to enable greater manageability of our experimental lab machines we designed a system where each workstation can network boot an iSCSI disk, which we then record disk traces of. The central server we used to host the iSCSI targets is a Pentium 4 2.4 Ghz server with 2.4 Terabytes of RAID5 disk on two 3ware 7500 Escalade IDE RAID cards.

To enable network booting, we installed bootPROMS containing Etherboot code on each workstation's network card. We then modified an Linux initial ramdisk to include ethernet drivers and the Cisco iSCSI initiator. Finally, we wrote a small network server which accepts authentication from clients (their MAC address in this case), and returns iSCSI configuration information which the Cisco client uses to connect and authenticate to its individual iSCSI target.

The Intel iSCSI reference implementation [5] provides an iSCSI target as a user space process. Virtual disks are provided by exporting appropriately sized files or block devices as the "backing store" for a virtual disk. In order to obtain the block level read and write traces that we require, we instrumented the Intel iSCSI target to compute the 128 bit MD5 hash of each block read from or written to each logical unit (LUN). This provides a content-based block "address" with very low probability of collision. The target then writes a transaction log that contains information about the blocks being read or written, time stamps and content summary hashes. In addition, for write operations, we copy the actual data from the backing store to a write log.

We were able to install Mandrake Linux 9.1 on a single workstation, and then copy the contents of that disk to the central server. We then simply copied the contents of the disk repeatedly to serve up disks to each client workstation. We also used VMWare 3.2 under Linux to install Windows 2000 on an iSCSI target as well because we could not get Windows booting directly off of a network attached iSCSI target as its system disk.

3 Trace Description and Analysis

For our experiments we used 10 workstation machines configured with between 128MB and 512MB of RAM, and either Pentium III, IV or Athlon processors running between 500 Mhz and 2.4 Ghz (Disks 1-10 in Table 2). All workstations were configured with full-duplex switched 100base-T Ethernet connections to the central server. We also used a dual processor Pentium III with 512 MB of RAM running Mandrake 8.0, VMWare 3.2, and Windows 2000 to trace Windows disk usage (Disk "win2k" in Table 2).

We make a distinction between "disks" and "traces". Each workstation used an emulated disk. The activity in each disk was divided into 24-hour intervals, and each of these logs is a "trace". Because of this we record the number of repeated writes within a single trace (*i.e.* only for each 24 hour period). In reality, a system may run for months at a time and one of our systems was left up and used for duration of trace collection. The Content-Based Buffer Cache will track repeated content from one day to the next in a real implementation, implying that the results shown in this paper are under-estimates of the true degree of shared content.

To obtain the data in Table 2 each trace was scanned. A read or write was counted whenever the operation per-

Disk #	Average Operations Per Day (or Trace)				# of Days
	Writes	Repeated Writes	Zero Block Writes	Reads	
1	1,476,573	1,265,093	66,781	105,227	13
2	1,545,183	1,322,575	75,710	71,898	14
3	1,556,154	1,332,916	72,537	56,449	14
4	780,208	670,227	36,983	82,446	12
5	628,096	540,547	29,431	89,734	3
6	1,432,974	1,227,173	70,582	92,678	14
7	1,174,403	1,006,937	53,712	92,761	8
8	1,251,825	1,071,016	63,778	170,958	5
9	657,768	532,934	69,530	168,617	26
10	603,793	407,582	59,448	418,060	45
win2k	142,932	69,280	2,895	3,087	16

Table 2: Workloads used in evaluation of content buffer cache. Each *trace* is the activity from a specific *disk* for a single *day*. Reads and writes are in 512 Byte blocks. Zero Block Writes are the number of 512 byte writes where the content of the block was all zeros, and is included in the repeated writes statistic. All accesses are averages over the number of days given. Individual traces were used in simulations.

formed in the trace was a read or write respectively. Additionally, whenever a block reappeared during a write, we recorded a repeated write as well. Finally, because it was the most numerous block in each trace by a significant fraction, we recorded the number of times the block containing all zeros was written out as Zero Block Writes.

4 Results and Analysis

A cache is a buffer for bursty writes, and keeps recently accessed data available for read traffic. If there are a sufficient number of clean cache entries, writes are processed without slowing down clients, however *any* cache miss on read results in a synchronous disk access. Thus, to a first approximation, the *cache read hit rate* should be a good indication of the performance of one caching policy *vs.* another. However, simply using the read hit rate ignores the access costs of disk drives.

To give an upper bound on caching effectivness, we looked at the 24 hour trace with the most read requests. That trace has 2,388,744 reads. With no caching, and assuming that each disk request would incur a random seek[1], the user perceptible stall time where every read request is serviced by the disk is 6.63 hours; if we use a 64MB cache and offset caching, the time is 6.10 hours. With the same sized cache and content-based caching, the stall time is 4.98 hours. As a lower bound, if all missed blocks were transferred sequentially off the disk in one large transfer, disk transfer time is then equal to the size of the transfer times the maximum sustained throughput. Assuming a maximum sustainable throughput of 30 MB/s, then the transfer time will be 39 seconds if no cache is used. If a 64MB offset-base cache is used, fewer reads would need to be done, and the resulting transfer time would be 36 seconds. With a 64MB cache and the CBBC policy, only 29 seconds are needed. A real cache with real disk operations will lie somewhere in between these two extremes.

[1] Estimated as 10ms

To understand the design decisions in the CBBC and to provide performance estimates for an actual implementation, we present a number of performance metrics. First, we show the effect of varying the hash size on the number of collisions in the CBBC hash table. Using a small hash table takes less memory, but results in more bit-by-bit comparisons of colliding blocks.

Next, we show the change in read hit rates when a single trace is used and then the change in read hit rate when multiple traces are combined to simulate simultaneous use of the cache. The content buffer cache algorithm can improve the utilization of the buffer cache by discovering content reuse within a single access stream or across multiple access streams. In our implementation, each LUN, or logical unit, is an independent access stream; we merge these streams to simulate the interaction of multiple concurrent users of the storage server. We first show that the content buffer cache algorithm is useful even when used by a single access stream, and then analyze the interaction of multiple access streams.

As discussed, the read hit rate may not indicate the magnitude of the performance improvement; it is possible that

offset based caching causes few seeks and thus has a lower stall time than CBBC. Rather than use an abstract metric indicating spatial locality in the set of blocks written to disk, we implemented the CBBC algorithm in the DiskSim disk simulator. This allows us to directly compare the stall time induced by the different caching policies.

4.1 Effect of Hash Size On Comparisons

When an offset miss occurs in the cache, the block that will be associated with that offset (*i.e.*, either the block read off disk for that offset, or the block to be written there) must be compared to each block that has the same content hash. We call these "*full comparisons*". As discussed in Section 2, there is a tradeoff between the size of the content hash table and the number of full comparisons required to insert a new block into the cache. This tradeoff is plotted for a sample workload from disk 10 for several cache sizes in Figure 4. While it appears there is some dependence on cache size, the overwhelming factor for determining the number of full comparisons required for a trace is how large a hash is used, because this directly controls the number of collisions to a single hash bucket. In Figure 5 the effect of the changing hash table size versus the system-wide idle time is plotted. The idle time for each simulation for each hash table size is compared to a simulation with an ideal content hash table that uses a separate dedicated space for the content hash (i.e. has no hash table overhead). From this graph, it seems that using the smallest hash table size (1/128th of the cache size) yields the performance closest to ideal (all of the cache sizes' performance were within 10% of the ideal case). We chose 1/64th of the cache size for hash table space to balance the tradeoff between full comparisons and change in simulation idle time. But this balance will be more fully explored in future work.

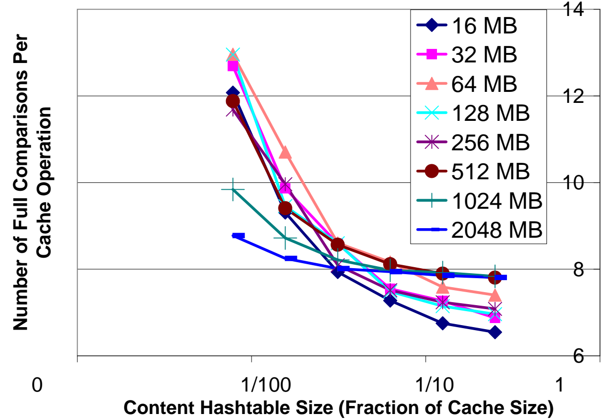


Figure 4: Number of full comparisons required due to collisions in the hash table *vs.* hash table size for a given cache size.

4.2 Read Hit Rate For Single LUN

We simulated workloads for all of our traces (See Table 2), at cache sizes of 32, 64, 128, 256, 512, and 1024 MB. We used 1/64th of this for a content hash table for all our simulations.

The results for a single logical unit are not uniform. Offset based caching had a higher read hit rate (*i.e.* better performance) at very small cache sizes (32 and 64 MB) where the overhead for the the CBBC datastructures significantly reduced the number of data block entries. When the cache size was large enough that the workload working set fit in the cache and there were no evictions, both offset and content based caching achieved the same read hit

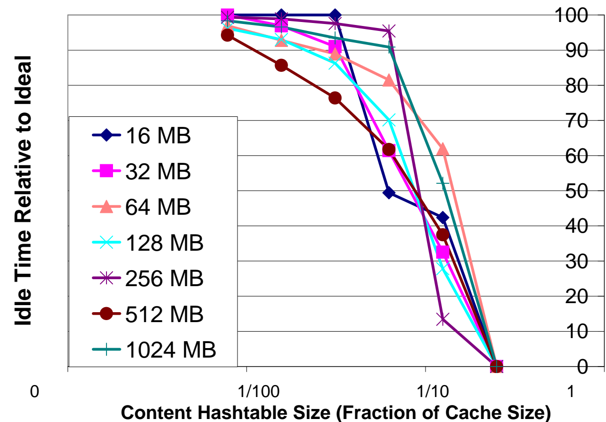


Figure 5: Trace Idle time relative to content caching wihtout cost of hash table (an idealized case) *vs.* hash table size for a given cache size. More idle time indicates better performance and capacity for additional disk traffic.

	Read Hit Rate	
CacheSize	Offset-Based	Content-Based
32	3.4%	3.2%
64	1.8%	11.2%
128	8%	25%
256	33.5%	46.9%
512	56.0%	63.7%
1024	65%	65%

Table 3: 512 byte block reads hit rates, for several cache sizes for a single trace. The Offset-Based Caching is the traditional private LRU cache which uses no content information to determine what to evict and what to keep around. The Content-Based Caching is our new algorithm which uses a content hash table to keep only single copies of blocks in memory. Both percentages are relative to *no* caching at all.

rate. In this situation, content caching would incurr more CPU overhead with no improvement in read hit rates. However, when the cache size was large enough that the CBBC data structures were not a bottleneck, and the workload was of sufficient size to cause evictions, Content-Based Buffer Caching produced consistently better read hit rates.

For example, the statistics for workload for Disk 10, Day 14, are listed in Table 3, and follow the model outlined above precisely. At 32MB, the offset cache has a higher hit rate because it has significantly more cache entries than the content based cache. However, between 64MB and 512MB, the content based cache outperforms the offset based cache. Finally, when the cache space outstrips the workload, both caches only miss on cold-start misses. This particular example shows that the CBBC policy can increase the read hit rate by ≈80%; the change in read hit rate for different traces ranges between ≈-5% to ≈100%.

4.3 Read Hit Rates For Multiple LUN's

When a second LUN is added, Content-Based Block Caching provides even better performance than the single LUN case when compared to offset-based caching. We chose a random subset of the pairs of traces, and ran the same simulation as the single LUN case (i.e. cache sizes of 32, 64, 128, 256, 512, and 1024 MB, 1/64th of this for content tracking.)

Because the number of I/O operations was proportionally larger with two traces, we more frequently encountered the results described for the single LUN case. Figure 6 shows this performance for a single two disk trace. Notice the bars

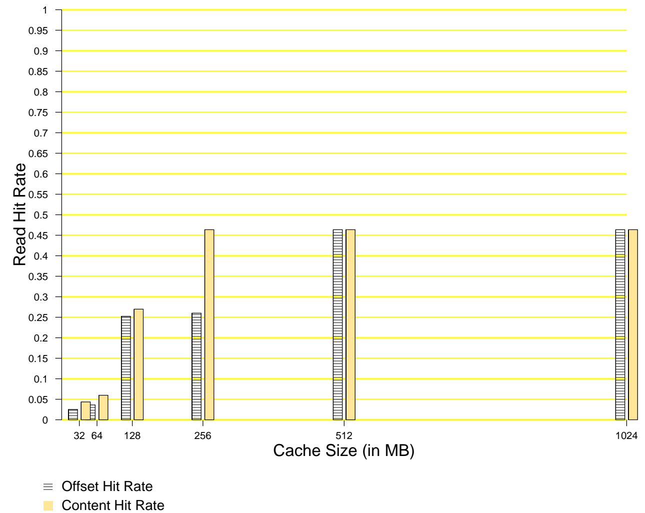


Figure 6: 512 byte block reads hit rates, for several cache sizes. The Offset-Based Caching is the traditional private LRU cache which uses no content information to determine what to evict and what to keep around. The Content-Based Caching is our new algorithm which uses a content hash table to keep only single copies of blocks in memory. Both percentages are relative to *no* caching at all.

at 256MB. Here, the content-based cache has extra cache entries and does not evict any blocks, while the offset based cache continues to evict content.

We also noticed that several traces which did not produce interesting cache behavior individually produced good read hit rate performance for content-caching when combined. This is the case for Figure 6 as each trace individually does not benefit from the content based cache.

We did not record how frequently blocks that were shared across multiple LUN's were accessed. This will be explored further in future work. However, by simply having references in multiple LUNs, we know that sharing between LUN's was occuring, and that the percentage of shared blocks at eviction time are significant.

4.4 Impact of Cache Policy on Disk Subsystem

Finally, for the same set of randomly chosen two disk simulations, we recorded the average response time, and the total simulation idle time using the DiskSim cache implementation.

Figure 7 shows distributional data for difference in sys-

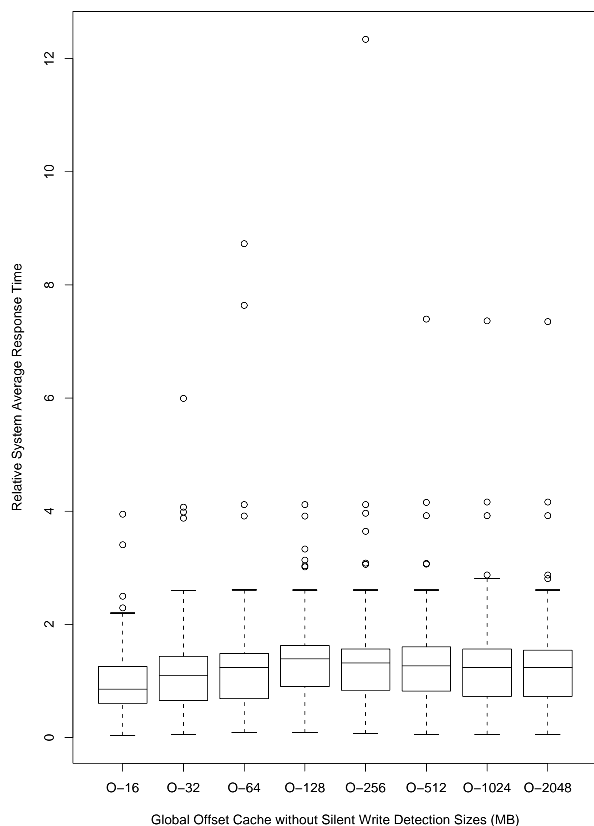


Figure 7: Boxplots showing the distribution of difference in average system response times between the offset and content based caching policies for a random sampling of dual trace runs at specific sizes of caches. The simulations using the different policies were run, and the results for average system response time (average time in milliseconds for a read or write) for content based caching were subtracted from results for offset based caching for that run. For example, the boxplot labeled "O-256" shows the distribution for the performance difference between content and offset caching with a 256MB cache. The boxplot shows the min, max, median and inter-quartile range of the data. The circles above each plot are outliers that are more than 1.5 times the value of the interquartile range of the box. For this metric, content based caching outperforms offset caching at all cache sizes.

tem response time for many different combinations of two traces for different cache sizes. For each simulation, the offset cache average system response time (average time for a read or write to complete) was subtracted from the content cache average. The content cache average response time performs an average of at least 1ms better across all simulations, and in some cases has huge performance improvement.

Figure 8 shows distributional data for difference in simulation idle time for the same set of two trace simulations as the system response time results above. For each simulation, the content cache simulation idle time was subtracted from the offset cache idle time. Simulation idle time takes into account the variance in individual requests, (which the average system response time graph does not demonstrate), showing how much additional traffic the cache could support before the cache becomes saturated. Again, for all cache sizes, content based caching outperforms traditional offset caching by a significant amount, having an average of at least 10 seconds of additional idle time for simulations that had between 400 and 700 seconds of disk activity over a 24 hour period.

5 Related Work

There is a long history of work on caching. There is work related to choosing the correct algorithm to achieve the best caching performance [6, 7, 8, 9]. Most of this work is orthogonal to work presented in this paper. Because Content-Based Block Caching aims to *reduce* evictions, it can take advantage of improvements in replacement algorithms just as a normal offset indexed cache can.

However, in [9], the issue of second level caching is brought to light, and investigated. We believe the Content-Based Block Cache may exhibit some of the qualities of a second level cache since the size of the buffer caches we are simulating are similar to the available memory in the systems we measured when collecting our traces. We will be exploring this further in future work, but we believe that Content-based caching should help *all* cache management mechanisms.

Previous work, more closely related is the work done investigating using compression to increase available cache space [10, 11, 12, 13]. In this work, the consistent theme is trading compression cost for space. There is a severe penalty if the working set of the application using the cache outgrows the uncompressed area and must continuously compress and uncompress data to get work done, however

if the working set is within the limits of the cache, compression caching can increase apparent cache size. Compressing is orthogonal to using content to increase available cache size. We can take advantage of progress in this area in our caching scheme.

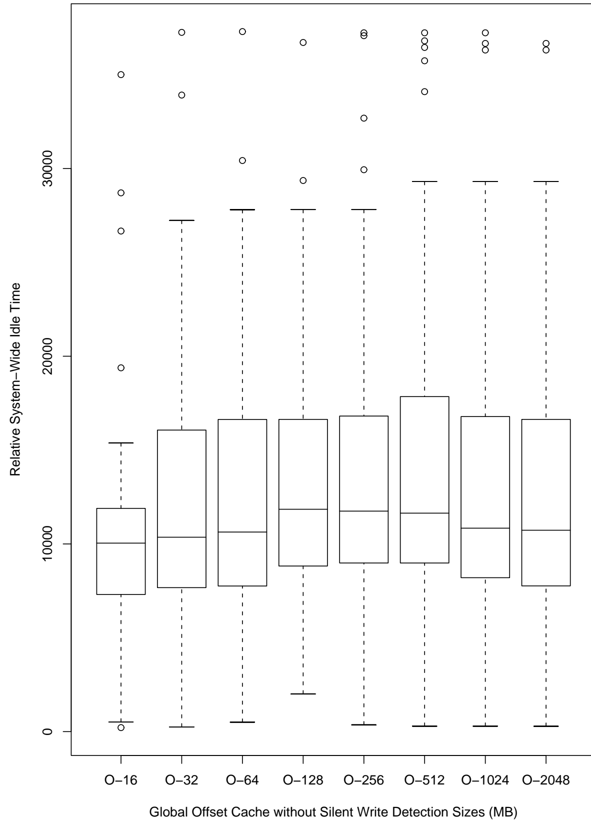


Figure 8: Boxplots showing the distribution of difference in simulation idle times between the content based and offset based caching policies for a random sampling of dual trace runs at specific sizes of caches. The simulations using the different policies were run, and the results for total simulation idle time (time in milliseconds that the disk subsystem was completely idle) for offset based caching were subtracted from results for content based caching for that run. For example, the boxplot labeled "O-256" shows the distribution for the performance difference between content and offset caching with a 256MB cache. The boxplot shows the min, max, median and inter-quartile range of the data. The circles above each plot are outliers that are more than 1.5 times the value of the interquartile range of the box. For this metric, content based caching outperforms offset caching at all cache sizes.

5.1 Block Systems Relying on Hashing

Farsite [14] is a distributed serverless file system designed to take advantage of growing workstation disk space in an enterprise network. It uses a database of hashes of files to locate and coalesce duplicate files in the system. Farsite does this coalescing to reduce disk space in the presence of replication for fault tolerance. In previous work [15], Bolosky et al. recorded file system traces of 550 Windows desktops and found that nearly 50% of space could be recovered by removing duplicate *files*. Content-Based Block Caching is not intended to cache the content of 550 machines, but saving 50% of the consumed space while only comparing data at the file granularity is further indication there is likely to be repeated content to be coalesced when caching multiple disks from similar systems.

Venti [1] is a system that uses hashes of blocks to coalesce writes to the actual data store to save space. Venti uses a block cache to store recently used hash to block translations. When a request comes in for a particular fingerprint, it is looked up in this block cache first. The Content-Based Block Cache maintains both a cache indexed by content similar to Venti, and the mappings both to the LUN map, and to the physical storage location. The Venti cache forces the user of the system to manage name to fingerprint mappings, and stores the block archivally instead of at the location indicated by the name. (For example, in Venti, to locate a file, the user must provide the hashes of the blocks of that file to the server, and the blocks are returned. Whereas in Content-Based Block Caching, all that is necessary is a LUN and offset and the block is returned. The block is stored at that LUN and offset on the physical disk, improving spacial locality of reference for reading multiple blocks not already in the cache.) Because of these mappings, Content-Based Block Caching could become a drop-in replacement for a higher level cache such as the Linux Buffer Cache, without sacrificing backwards compatibility. We will be exploring this in the future.

LBFS [2] describes using "semantic block boundaries" to collapse chunks of files which are the same to reduce write bandwidth. Whenever a file is written, the server computes block boundaries not based on offset in the file, but rather by using a fingerprinting mechanism which has a reasonably small output space. (They use Rabin Fingerprint-

ing [16] in the paper.) Each *byte* in the file is incrementally added to the potential block, and then the fingerprinting algorithm is run on that block. If the fingerprint output is some pre-chosen magical number, then that offset is output as the block boundary, and a new block is begun. The interesting property the author's claim is that insertions or deletions in the middle of a file do not cause global changes to the blocks after that block in the file. Instead, there is a local change to that block, either producing two blocks, or combining a previous block with that block, but otherwise none of the other blocks in the file will change. This is important for the LBFS implementation because they are using the MD5 hash of the resultant "semantic block" to save bandwidth between client and server by not sending blocks which haven't been altered when a file is written out.

Farsite, Venti, and LBFS are examples of "Compare By Hash" [17]. By addressing blocks using a hash of the block's contents these systems are relying on the even distribution of block to hash translations in the hash space to ensure that there are no hash collisions. Content-Based Block Caching does not rely on collision-free hash algorithms for correctness. Blocks are compared byte-wise to determine if they are the same or not when they hash to the same hash value. The hash table is used as a hint that there is something else already in the cache which might be the same. Because of this, the system can get by with a much smaller hash table than would otherwise be required, further reducing the space overhead of coalescing.

6 Conclusions and Future Work

For single logical disks, content-based block caching can reduce disk accesses by as much as 80% over traditional offset-based caching mechanisms, with manageable space and time overheads in the form of the content hash table size and full comparison costs. Additionally, for multiple logical disks, content-based block caching has improved read hit rates, significantly improved average system response times and more simulation idle time relative to offset caching. The Content-Based Block Cache has the additional savings of silent writes which traditional caching mechanisms do not provide.

Our current set of disk traces has very light workload characteristics (400-600 seconds of disk activity over 24 hours). Our next disk tracing infrastructure will be based on the Xen Virtual Machine Monitor in order to more scalably trace large numbers of workstations. We plan to include large transactional workloads such as webserver or database workloads in our test suite to more fully exert the cache at higher loads.

We would like to explore other cache replacement policies besides LRU. In particular, the Multi-Queue algorithm proposed in [9] may perform well. We are also planning to implement the Content-Based Block Caching algorithm in a real system, to evaluate speedup due to improved cache hit rate, and better tune parameters such as content hash bitsize, and LUN Map data structure choice. Finally, we have a disk-related project *Peabody* [18] for which Content-Based Block Caching forms a complementary caching front-end, and we hope to integrate the two and evaluate the performance of the complete system.

References

[1] S. Quinlan and S. Dorward. Venti: a new approach to archival storage. In *First USENIX conference on File and Storage Technologies*, Monterey, CA, 2002.

[2] Athicha Muthitacharoen, Benjie Chen, and David Mazieres. A low-bandwidth network file system. In *Symposium on Operating Systems Principles*, pages 174–187, 2001.

[3] John Kubiatowicz, David Bindel, Yan Chen, Patrick Eaton, Dennis Geels, Ramakrishna Gummadi, Sean Rhea, Hakim Weatherspoon, Westly Weimer, Christopher Wells, and Ben Zhao. Oceanstore: An architecture for global-scale persistent storage. In *Proceedings of ACM ASPLOS*. ACM, November 2000.

[4] Parallel Data Lab CMU. Disksim simulation environment v3.0. `http://www.pdl.cmu.edu/DiskSim`.

[5] Michael Mesnier. Intel iscsi reference implementation. `http://sourceforge.net/projects/intel-iscsi/`, Dec 2001.

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

[7] Theodore Johnson and Dennis Shasha. 2Q: a low overhead high performance buffer management replacement algorithm. In *Proceedings of the Twentieth International Conference on Very Large Databases*, pages 439–450, Santiago, Chile, 1994.

[8] Donghee Lee, Jongmoo Choi, Jong-Hun Kim, Sam H. Noh, Sang Lyul Min, Yookun Cho, and Chong-Sang Kim. On the existence of a spectrum of policies that subsumes the least recently used (LRU) and least frequently used (LFU) policies. In *Measurement and Modeling of Computer Systems*, pages 134–143, 1999.

[9] Yuanyuan Zhou and James F. Philbin. The multi-queue replacement algorithm for second level buffer caches. In *Proceedings of the 2001 USENIX Annual Technical Conference*, Boston, MA, June 2001.

[10] Fred Douglis. The compression cache: Using on-line compression to extend physical memory. In *USENIX Winter*, pages 519–529, 1993.

[11] Paul R. Wilson, Scott F. Kaplan, and Yannis Smaragdakis. The case for compressed caching in virtual memory systems. In *USENIX Summer Conference*, pages 101–116, Monterey, CA, June 1999.

[12] S. Kaplan. Compressed caching and modern virtual memory simulation, 1999.

[13] Toni Cortes, Yolanda Becerra, and Raúl Cervera. Swap compression: resurrecting old ideas. *Software Practice and Experience*, 30(5):567–587, 2000.

[14] A. Adya, W. J. Bolosky, M. Castro, G. Cermak, R. Chaiken, J. R. Douceur, J. Howell, J. R. Lorch, M. Theimer, and R. P. Wattenhofer. Farsite: Federated, available, and reliable storage for an incompletely trusted environment. In *Proceedings of the 5th OSDI*, Boston, MA, December 2002.

[15] W. J. Bolosky, J. R. Douceur, D. Ely, and M. Theimer. Feasibility of a serverless distributed file system deployed on an existing set of desktop pcs. In *SIGMETRICS 2000*, pages 34–43, Santa Clara, CA, June 2000.

[16] Andrei Z. Broder. Some applications of rabin's fingerprinting method. *Sequences II: Methods in Communications, Security, and Computer Science*, pages 143–152, 1993.

[17] Val Henson Sun Microsystems. An analysis of compare-by-hash. In *Hot Topics in Operating Systems IX*, Lihue, Hawaii, May 2003.

[18] Charles B. Morrey III and Dirk Grunwald. Peabody: The time travelling disk. In *Proceedings of the 20th IEEE/11th NSASA Goddard Conference on Mass Storage Systems and Technologies, MSST2003*, San Diego, CA, April 2003.