Caching Strategies to Improve Disk System Performance

Ramakrishna Karedla, Digital Equipment Corporation

J. Spencer Love, Independent Consultant

Bradley G. Wherry, McREL Corporation

Caching can help to alleviate I/O subsystem bottlenecks caused by mechanical latencies. This article describes a caching strategy that offers the performance of caches twice its size.

Processor speeds have increased dramatically over the last few years and are expected to continue to double every year, with main memory density doubling every two years. At the same time, I/O appetite continues to grow, especially with the emergence of applications such as multimedia and scientific modeling, which place an ever-increasing demand on the I/O subsystem.

Although rapid technological advances have doubled disk capacity every 1-1/2 years since 1990, no similar advances are expected to reduce the access times of storage devices; the difference in access times between main memory and storage devices is many orders of magnitude. As a result, I/O subsystems limit overall system response time and leave the CPU underutilized.

A typical data-storage hierarchy consists of main memory, magnetic disks, and optical and tape drives (in increasing order of capacity and access times). Typically, the host operating system routes an I/O request via the appropriate device driver to the device controller. The controller then transfers the data to host memory via the host bus adapter. For common transfer sizes, mechanical latencies are major contributors to slow device response times.

I/O subsystem manufacturers attempt to reduce latency by increasing disk rotation speeds, incorporating more intelligent disk scheduling algorithms, increasing I/O bus speed, using solid-state disks, and implementing caches at various places in the I/O stream.

In this article, we examine the use of caching as a means to increase system response time and improve the data throughput of the disk subsystem. After explaining some basic caching issues (see sidebar for an introduction to caching terminology), we examine some popular caching strategies and cache replacement algorithms, as well as the advantages and disadvantages of caching at different levels of the computer system hierarchy. Finally, we investigate the performance of three cache replacement algorithms: random replacement (RR), least recently used (LRU), and a frequency-based variation of LRU known as segmented LRU (SLRU).

0018-9162/94/$4.00 © 1994 IEEE

Cache design and implementation

Cache performance depends on a wide range of design and implementation parameters, and much effort has gone into studying the performance impact of these parameters on processor caches.[1,2] Typically, processor caches are characterized by very low access times of 10 nanoseconds, request sizes of up to 32 bytes, low miss rates of up to 3 percent, and cache sizes of up to 128 Kbytes. To achieve such low access times, they are implemented in hardware.

I/O caches, on the other hand, are faced with much larger request sizes, data sets, and backing store latencies. Because cache management for large caches involves more control logic and is both expensive and difficult to implement in hardware, I/O caches are typically implemented and managed in software with some hardware assists. We describe some of the main parameters and trade-offs in I/O cache design below.

Cache design parameters. The chief metric for cache performance is its miss rate. However, the benefit of a cache must be measured not only at the I/O subsystem level (in terms of increased I/O throughput), but also at the overall system level in terms of faster response times and more productive CPU utilization (reduced I/O wait states and increased number of completed jobs).

Cache size. A question that confronts all I/O subsystem designers is: How large should the cache be to achieve an acceptable miss rate? Although RAM costs are decreasing, cache memory is still expensive. Studies have shown that increasing the cache beyond its optimal size has diminishing returns.[3] The miss rate decreases asymptotically with increased cache size but begins to increase again after a certain point. Cost and space considerations are also factors in determining cache size. To achieve an optimum cost-to-performance ratio, system designers generally believe that the size of a cache should be at least 0.1 to 0.3 percent of the backing store. Manufacturers typically offer caches between 0.1 and 1 percent of the backing store.

Line replacement algorithm. The line replacement algorithm has a profound impact on cache performance in terms of the miss rate and thus indirectly on cache size. Vendors try to devise efficient replacement algorithms that can offer higher cache performance for relatively small cache sizes.

Caches work on the premise that data in the cache will be reused often, thus reducing the number of accesses to the backing store. To achieve this, caches exploit the principles of spatial and temporal locality of reference. Spatial locality implies that if an object is referenced, then nearby objects will also soon be accessed. Temporal locality implies that a referenced object will tend to be referenced again in the near future.

The goodness of a replacement algorithm lies in its ability to evict data that has no immediate need to reside in the cache, and thus to retain data more likely to be accessed, for as small a cache size as possible.

To decide which line to replace, the algorithm needs to closely monitor the usefulness of the lines in the cache. Thus, another measure of goodness is the algorithm's computational overhead. Since host, controller, or drive CPUs perform many other functions in addition to cache management, I/O subsystem designers tend to choose algorithms that are computationally less expensive and perform a minimal amount of bookkeeping. Since replacement algorithms give cache designers a competitive edge in the market, manufacturers either patent them or consider their implementation a trade secret. A variety of replacement algorithms for database

Basic caching terminology

A *cache buffer* is faster memory used to enhance the performance of a slower memory (a disk drive, for example), known as the backing store. By keeping copies of backing store data, caches can service some requests at faster memory speeds.

A *cache line* is a sequence of data blocks managed as a unit; it is considered "dirty" if it has been written to. The operation of writing these modified blocks to disk is called *purging* or *flushing.* Cache replacement, also known as *eviction,* is the operation of discarding data from the cache to make room for new data. If the lines to be evicted are dirty, they are purged to the backing store; otherwise, they are overwritten. A *cache replacement algorithm* decides which cache line to discard when replacement is required.

An I/O request to a storage device, especially a read request, searches the cache first. A request is said to be a *cache hit* when all of the requested data is found in the cache. The *hit rate* is the percentage of all I/O requests satisfied from the cache. A request is said to *miss the cache* when any of its data blocks are not found in the cache and must be obtained from the backing store. The *miss rate* is the percentage of all I/O requests not found in the cache.

A *direct-mapped cache* allocates a cache line to one particular area in the cache, whereas a *fully associative cache* can place the cache line anywhere in the cache. A *read- or write-through cache* is one that intercepts only read requests. In this case, write requests are commonly handled by writing the data to disk before or after adding it to the cache and signaling completion. This approach generally assures the availability of the data in the event of a failure. The drawback is that it decreases cache performance by going to disk for every write.

In a *write-back cache,* the data is written to the cache and write completion is signaled immediately. The modified cache line is later written back to the disk drive at a more opportune time. A write-back cache generally performs better than a write-through cache, but its implementation is more complex because it must guarantee data availability and validity in the event of a drive failure.

buffering, processor caching, and virtual memory paging have been studied. Listed below are some of the more popular cache replacement algorithms.

- *Random replacement (RR)*: This algorithm replaces cache lines by randomly selecting a cache line to discard. This policy is the easiest to implement but generally performs poorly, especially for small cache sizes, because it does not make use of the attributes of spatial and temporal locality that are associated with I/O request patterns.

- *Least recently used (LRU)*: The LRU algorithm is one of the most popular replacement policies, particularly in commercial implementations. It is used widely in database buffer management, virtual-memory management, file system and I/O caches. This policy exploits the principle of temporal locality and evicts the cache line used least in the recent past on the assumption that it won't be used in the near future. It is simple to implement for small caches but becomes computationally expensive for large ones. In addition, LRU uses only the time since last access and does not take into account the access frequency of lines when making replacement decisions.

- *Least frequently used (LFU)*: This algorithm uses the history of accesses to predict the probability of a subsequent reference. The cache maintains a frequency-of-access count for all its lines and replaces the line that has been used least frequently. Unfortunately, lines with large frequency counts that will not be accessed again (such as a recently active but currently cold file) tend to remain entrenched in the cache, preventing newer additions to the cache from gathering sufficient reference counts to stay in the cache. This leads to a phenomenon called *cache pollution*, in which inactive data tends to increase the miss rate and hence reduce the performance of the cache. Generally, an aging policy is used to increase the performance of LFU, by which the frequency counts of inactive cache lines are gradually reduced, making them candidates for replacement.

- *Variations on LRU*: A frequency-based LRU replacement algorithm[4] improves on LRU by partitioning the LRU stack into three sections whose sizes are tunable. On being referenced, a line is placed in the topmost section of the stack. Unlike LFU, the reference count of lines repeatedly referenced in the top section is not incremented. Thus, temporary "locality of reference" is factored out. Eventually, the line ages into the bottom part of the list by LRU replacement and is finally evicted if not referenced.

An interesting and sophisticated LRU K-page-replacement algorithm[5] for database page buffering could also find possible use in I/O caches. For each page, this algorithm maintains a history table that contains the times of the last K references, factoring out correlated references due to temporary locality as in frequency-based replacement. The cache is implemented as an LRU list, and the decision as to which cache line to replace is based on the history table and other access pattern parameters.

Although the implementation details of replacement algorithms are not made public, we have observed a growing trend among vendors toward algorithms that adapt to changing access patterns. These algorithms generally are combinations of LRU, LFU, and read-ahead strategies, with varying thresholds on the amount of data that is prefetched.

Read-ahead strategies. A read-ahead strategy known as prefetching exploits the principle of spatial locality in an attempt to minimize latency in data access by anticipating future requests for data and bringing it into the cache. Most disk drives implement prefetch in an onboard cache. However, a large prefetch can have a negative impact on small caches, because it can displace data that would have been useful in the cache.

Host caches can use this strategy more intelligently than controller or disk caches, since applications can better predict the type of data most needed in the near future. Examples include file system caches that prefetch the rest of a file that is being read, and query optimizers in database applications. Prefetching is particularly effective when read requests are sequential. Optical disk drives and CD-ROMs commonly employ large read-ahead caches to display large video or image files without interruption.

More "intelligent" disks have multiple partitions or segments to enable them to cache I/O streams to different parts of the disk. This strategy is beneficial in a multiuser or a multitasking environment. For example, the firmware in DEC drives will bunch two buffers per I/O stream upon detecting a sequential access and then alternate between them if the read is still sequential. (The assumption here is that most applications will not reuse data that they requested two tracks earlier.) The prefetch is usually done when no I/O request is pending and is typically aborted when a user request arrives. In addition to this capability, vendors also offer a tunable read-ahead threshold to minimize both cache pollution and the latency in transferring large amounts of data from the disk to the I/O bus.

Cache line size. Determining the optimal cache line size is important, because too large a line size could result in purging data from the cache that soon would have been referenced. For large cache lines, cache directory searches can be faster, reducing cache lookup time, whereas a small line size would increase cache search time. Studies show that the miss rate is lower for small line sizes in small caches and large line sizes in large caches.[3] Most vendors offer tunable line sizes of 2 Kbytes and above. Also, to avoid cache pollution, vendors offer user-selectable upper bounds on the request size that the cache will process. Requests for larger sizes bypass the cache and are routed directly to the backing store. DEC HSJ90 storage controllers use a line size of 8 Kbytes and implement a read and write threshold. One commericial host-based cache dynamically partitions the cache into three partitions for small, medium, and large transfers, depending on the request size distribution. Since the cache is built from the free page memory pool, the vendor claims that cache space is utilized more efficiently.

Write-to-cache strategy. We examine two popular policies that address the issue of adding a cache line for a write request.[6]

The *write allocate* policy dictates that, upon receiving a write request, the cache is searched for the referenced line, which is then updated in place in the cache. If the referenced line is not found, then a new cache line is allocated, and the new data is stored. This policy benefits such applications as program libraries, where the new data has a good chance of being accessed soon in a read-after-write command. On the downside, useful data can be flushed if the cache is full.

The *write invalidate* policy dictates that the cache be searched first for the referenced line, which is then invalidated by being evicted from the cache, thus making room for new data. This policy is useful in the case of database logs, system log files, and temporary files such as intermediate printer files, where the I/O stream has a tendency not to access that data again.

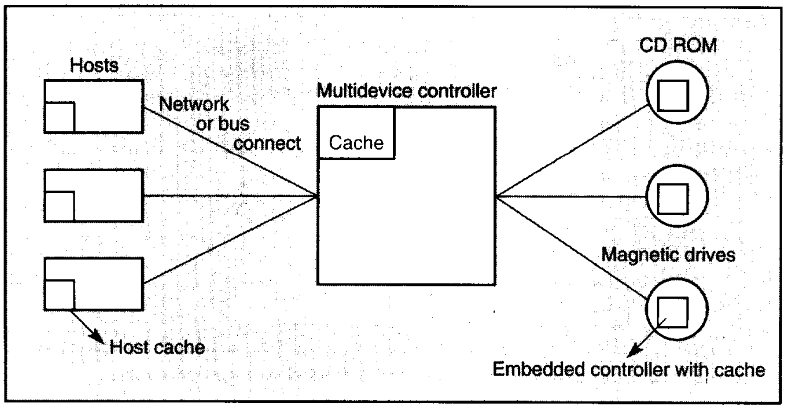


Figure 1. Block diagram of cache implementations at various levels of a system.

Writing back to disk. Various disk-scheduling algorithms exist that attempt to minimize latency by intelligently scheduling accesses to the disk.[7] With caching, disk controllers need to modify their seek optimizations to include traffic from the caches to the disk. The issue of intelligently scheduling dirty lines to the disk in the case of a write-back cache has received much attention.

One often-implemented policy is *periodic write back*, where pending writes to the disk are gathered and written to the disk in one sweep at regular intervals. For example, most Unix systems use a 30-second period. Such a scheme ensures less data volatility but is not very efficient, since other requests are kept pending during a write sweep.

Another popular policy is called *opportunistic write back*. This policy exploits read seeks by aggregating and scheduling writes pending to the same area as the read request. These writes are then "piggybacked" on the read request and written to the disk. This policy effectively cuts the cost of writing to the disk,[8] as compared with a periodic write-back policy. Opportunistic write back of dirty lines to an entire cylinder performs better than writing to a single track, but it stalls the read operation that initiates the seek. This could lead to the queuing up of read requests, thus affecting device response time and, hence, the cache's effectiveness.

Cache location. Another important issue is the optimal location of the cache along the I/O data path; that is, where should the cache be placed to maximally benefit the system? Figure 1 illustrates a typical cluster computing environment consisting of a set of hosts connected to storage devices via controllers. Three possible locations for the cache are shown: a cache in each of the hosts, a cache in the multidevice storage controller, or a cache in the drive itself. We examine the pros and cons of caching at each of these levels and briefly describe commercial products that implement such caches.

Two possible I/O subsystem configurations are common. In low-end configurations, a controller with a dedicated cache is embedded in the drive itself. Examples include DEC and Seagate Elite drives. The high-end mainframe configuration consists of a controller connected to a number of disk drives. DEC's multidevice Hierarchical Storage Controller (HSJ) is a typical example. Here, the controllers on the drive perform tasks such as reading, writing, prefetching, and transferring data, while the more complex operations such as command reordering, seek optimizations, and more intelligent caching are performed by the multidevice controller.

- *Host caching.* Most studies agree that host caching significantly reduces the host's response time resulting from such factors as network or bus latency and disk access time.[6] A cache hit in the host helps cut down data access time from tens of milliseconds to a couple of milliseconds. Applications have a better knowledge of the data needed, thus making host caches more intelligent. Database applications typically have their own optimized caches. Applications such as word processing, spreadsheets, and program development tools usually have buffers in the application that cache data and periodically save it to the disk.

To increase I/O performance, many vendors sell products that use free memory pages to implement a software cache. Such products periodically (typically every 5 seconds) monitor system memory to adaptively size the cache in accordance with the application's needs so that the system does not thrash. Cache coherency among hosts sharing a backing store is maintained by broadcasting messages to other hosts to invalidate dirty pages in their respective caches.

The advantages of caching at the host include a decrease of an order of magnitude in latency due to faster data access and lower cache pollution, and lower miss ratios due to more intelligent prefetching and caching of data by applications. The cache size can be dynamically varied to achieve a low miss rate for a given cache size and to help reduce contention for storage devices or the bus interconnect.

Some disadvantages of host caching are data volatility (unless battery backed) and CPU overhead for cache management. If the CPU is already heavily utilized, then host caching could degrade system performance. It is also more difficult to ensure cache coherency among the different hosts' caches if they are write-back caches. Modern file systems such as Windows NT's TNFS (The New File System) use a log-based recovery protocol to maintain data consistency in the event of a system failure.

- *Multidevice controller caching.* Figure 2 shows a simplified cache implementation in a DEC HSJ40 storage controller. The policy processor interprets host commands, runs the cache algorithm software, implements

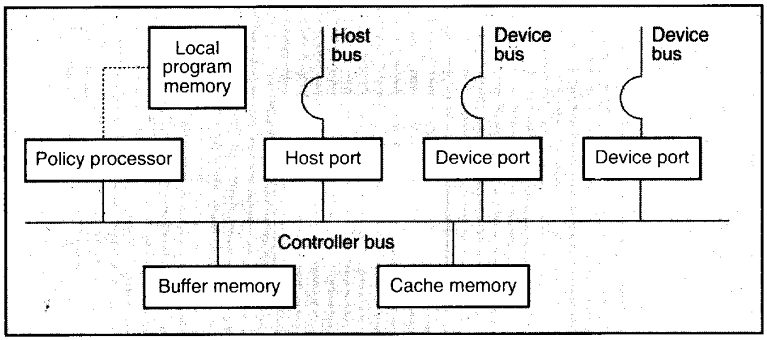


Figure 2. Simplified block diagram of DEC's HSJ40 storage controller.

cross-device functionality (such as a RAID, or redundant array of independent disks), drives devices, and implements internal diagnostics. The local program memory removes processor-related traffic from the data transfer bus. The buffer and cache memories can be combined into one contiguous piece of memory. However, they are often kept separate so that cache memory can be excluded, expanded, or made nonvolatile — independent of buffer memory.

The control structures for the cache generally consist of a control block for each cache line organized into a hash table or a tree indexed by backing-store addresses. These control blocks also contain data (such as bits and words) used to implement the cache replacement algorithm. The HSJ40 controller can have up to 64 Mbytes of write-through cache memory. Write-back caching is expected in the near future. Previous versions of the controller cache used a simplified version of the LRU algorithm, but the new generation of HSJ caches use segmented LRU.

Caching at the controller has at least five advantages. First, cache coherency is achieved, since the controller sees the I/O streams from all hosts and since data from multiple hosts resides in the controller cache. Second, functions such as command queuing, seek ordering, DMA transfers, and other optimizations are performed by the controller completely transparent to the host. The host can issue a command and then attend to other tasks. Third, because large storage controllers possess more physical space, the cache can be much larger than at the disk. Fourth, cache pollution at the controller level is minimized, because only the more frequently used portions of the prefetched data from each drive would migrate upward to the controller. Finally, one can efficiently allocate the controller cache among the different drives based on their utilization and work-load profiles.

However, there are some disadvantages. The controller sees a random mix of data requests from different hosts, resulting in a higher miss rate compared to a host cache. Optimizing for a lower miss rate would mean a larger cache size in comparison to a host cache. Write-back caching is more difficult to implement with dual-pathing because of the difficulties of keeping both HSJs' caches consistent in case of a fail-over. As in the case of host caching, a volatile cache could leave data that is shared systemwide in an inconsistent state.

- *Caching at the host-based adapter (HBA).* Many vendors sell intelligent HBAs that contain high-performance processors and large amounts of cache memory. Some of these adapters include an I/O protocol chip on the adapter card that performs similar functions as a multidevice storage controller. Data Technology Corp.[9] markets an HBA with up to 4 Mbytes of cache that intelligently combines read-ahead and write-back caching to achieve high performance. High-performance SCSI 2 I/O buses on workstation servers and high-end PCs are well suited to processing I/O requests at the HBA level. This allows better overall host CPU utilization. Intelligent HBAs can implement command tag queuing, and perform overlapped seeks, seek-data reordering, and caching transparent to the host CPU. The HBA's function is similar to that of a multidevice storage controller in a cluster environment. There is growing interest in implementing caches on HBAs for network and database servers in a LAN environment. The advantages and disadvantages of caching at the adapter level are similar to those of a multidevice controller cache.

- *Drive-level caching.* As mentioned earlier, almost all current drives have embedded controllers with on-board caches. Drive-level caching is therefore the same as caching at a single device controller. Typically, a 2-Gbyte drive comes with 1 Mbyte of cache, with prefetching being the most common cache implementation. Vendors are now beginning to offer read-ahead and write-back caches at the drive level. It remains to be seen how well such caches perform in conjunction with a multidevice controller cache or a host-based adapter cache. One vendor[10] claims near solid-state disk performance at a much lower cost per Mbyte by using a combination of magnetic storage and up to 256 Mbytes of read-ahead, write-back cache that uses very efficient caching strategies.

- *Multilevel caching*: A commonly asked question is: Is caching at multiple levels within a system more effective than a single large cache placed strategically in the I/O path? McNutt suggests[11] that given a fixed amount of memory budget, the optimal way to allocate memory is to use a relatively large cache at the host and a small- to moderate-sized cache at the controller. In a situation like this, one can intuit that the miss rate of the host cache would be lower than the controller cache, which would generally see a greater mix of random reads and writes, thus increasing its miss rate. One possible way to increase the efficiency of multilevel caches is to cache different I/O workloads at each level. For example, program libraries would best benefit by host-based caches, while some database activites such as logs would best benefit by controller-level caches. Further work is needed to evaluate the performance of multilevel I/O caches.

Simulating cache replacement algorithms

To study the relative performance of cache replacement algorithms for multidevice storage controllers, we simulated the behavior of three cache replacement algorithms: random replacement, least recently used, and segmented LRU. RR provides a baseline for evaluating LRU and SLRU performance. We reason that any algorithm based on familiarity with the I/O request stream should generally have lower miss rates than RR, which has no knowledge of the history of accesses and does not use the principle of locality of reference. Thus, the performance of RR provides us with a plausible upper bound on the acceptable miss rate for a given cache size.

Segmented LRU. During analyses of workload traces, we observed that disk blocks accessed at least twice tend to be reused many times before being removed from the cache. The problem with LRU caches is that they can be flooded by lines that are accessed only once, flushing out lines that have a higher probability of being reused. This problem becomes important when lines that are accessed only once make up a large fraction of the workload and when cache sizes are small. Recognizing this pattern of cache access led us to modify the LRU replacement algorithm to distinguish between lines that have been accessed only once and lines that have been accessed multiple times. SLRU is the simplest variation on frequency-based LRU replacement algorithms. We realize that the LRU-K algorithm[5] is very similar and more sophisticated than the one we describe, but the SLRU algorithm is easier to implement and is computationally cheaper than the LRU-K algorithm.

An SLRU cache is divided into two segments, a probationary segment and a protected segment. Lines in each segment are ordered from the most to the least recently accessed. Data from misses is added to the cache at the most recently accessed end of the probationary segment. Hits are removed from wherever they currently reside and added to the most recently accessed end of the protected segment. Lines in the protected segment have thus been accessed at least twice. The protected segment is finite, so migration of a line from the probationary segment to the protected segment may force the migration of the LRU line in the protected segment to the most recently used (MRU) end of the probationary segment, giving this line another chance to be accessed before being replaced. The size limit on the protected segment is an SLRU parameter that varies according to the I/O workload patterns. Whenever data must be discarded from the cache, lines are obtained from the LRU end of the probationary segment.

The SLRU policy protects the cache against access patterns that can flood an LRU cache with data that will not be reused. For example, data that is accessed only by backup operations or long sequential file accesses will be touched only once. Thus, this data will not enter the protected segment and displace data from it. In an LRU cache, no data is protected from displacement by these and similar access patterns because LRU does not recognize frequency of reuse.

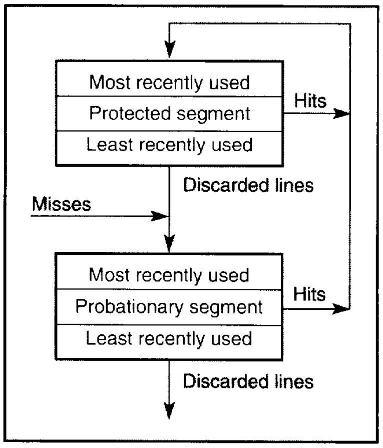


Figure 3. Logical flow of SLRU cache lines.

Figure 3 depicts the logical flow of SLRU cache lines between the two segments. In practice the segments of an SLRU cache are implemented by a single linked list. This list runs from the MRU end of the protected segment to the LRU end of the probationary segment. An additional pointer called the *boundary pointer* points to the MRU line of the probationary segment, which is generally somewhere in the middle of the list. Also, each cache line must contain a 1-bit flag indicating which segment it resides in; the flag has the values PROT and PROB for the protected and probationary segments, respectively.

When a cache miss occurs, the new line is added to the SLRU list between the MRU probationary line (as pointed to by the boundary pointer) and the LRU protected line. The flag bit of the new line is set to PROB and the boundary pointer is changed to point to the new line. If a cache entry needs to be discarded from the cache to make space for the new line, the LRU probationary entry is chosen for discarding. The limit on the size of the protected segment ensures that the probationary segment cannot be empty when the cache is full. When a cache line whose flag has the value PROT is hit, it is handled like a hit in an LRU cache, and moved to the MRU end of the list. The maintenance of the extra pointer and flags in SLRU adds very little memory and computational overhead to that of LRU.

Experimental methodology and workloads. We compare the three replacement policies RR, LRU, and SLRU in terms of their miss rate as a function of cache size. The cache we simulate is a fully associative write-through cache that performs no prefetching. Each cache line contains one disk block of 512 bytes and 8 bytes of overhead data. Lines are allocated in the cache either on a read or a write. A read request is considered a hit only if all the disk blocks requested are in the cache. The simulation is of a cold start cache; that is, the cache is initially empty and fills up as the simulation progresses. This approach is practical because the traces are long enough to completely fill the caches very early in the simulations.

Description of workload traces. To drive our simulations with realistic workloads, we use customer workload traces. The I/O workloads were collected using FSTrace, an I/O tracing package. The traces contained I/O request information such as the request type, request size, target HSJ, drive logical block address, and I/O completion time.

These traces were then used to drive our caching algorithm simulations. The cache simulator reproduces the behavior of various replacement policies in response to the various workloads. For each workload we simulate a single controller cache. This closely approximates an entire disk farm connected to a single high-capacity storage controller. All the systems traced are VAXcluster systems comprising three to five independent VAX processors on

Table 1. I/O workload profile.

Workload Type	Inventory Control	Batch Processing	Airline Reservation System	Scientific Time-Sharing
Gigabytes of data touched	5.3	5.7	1.4	2.9
Percentage touched for reads only	85.0	72.0	64.0	76.0
Avg. read request size (in Kbytes)	3.0	4.0	4.5	2.5
Avg. write request size (in Kbytes)	5.5	4.0	6.0	3.0
Read-to-write ratio	6.0	8.0	5.0	4.0
Peak request rate (in I/Os per second)	1,025.0	693.0	283.0	355.0
Peak request rate (in Mbytes per second)	3.5	2.5	1.6	2.0
Percent of total I/Os to busiest disk	12.0	29.0	16.0	25.0
Number of files on system (in thousands)	48.0	35.0	59.0	706.0

a high-speed computer interconnect. Digital HSJ storage controllers let the VAXs share access to a combination of 450-Mbyte and 620-Mbyte disk drives.

Traces were collected from three customer sites running different applications on different system configurations. Table 1 gives an I/O profile of the various workloads. Detailed file-level analyses of the workloads has been published elsewhere.[12] The four workloads were an inventory control system, a nighttime batch processing system, an airline reservation system, and a scientific data processing system.

The inventory control (IC) workload was collected from a system that supported a nationwide industrial parts distribution system. The system's primary application was a proprietary database package running an OLTP application. The daytime operation was primarily OLTP. Customer orders received from across the country were processed and inventories were updated. The database files were accessed predominantly for Read operations. This trace lasted twelve hours from 6 a.m. to 6 p.m.

A nightly batch processing (BP) workload was collected from the same system that ran the inventory control application. The night operation updated the database files in terms of orders received, type and number of parts shipped, total daily sales, and so forth. This data was used to generate management reports for the next day's operation. The night operation also included backup operations to protect against disk failures. This trace lasted approximately eight hours from 9 p.m. to 5 a.m. the next day.

The airline reservation (AR) system ran proprietary on-line transaction processing software (OLTP) using indexed sequential files on a CI (Computer Interconnect) VAXcluster. Ninety-five percent of capacity was used for this one application. This trace lasted about nine hours, from 9:30 a.m. to 6:30 p.m.

The scientific time sharing workload trace (ST) was gathered from a system for scientific data collection and processing using analytical software and modeling packages. The system also did program development in Fortran

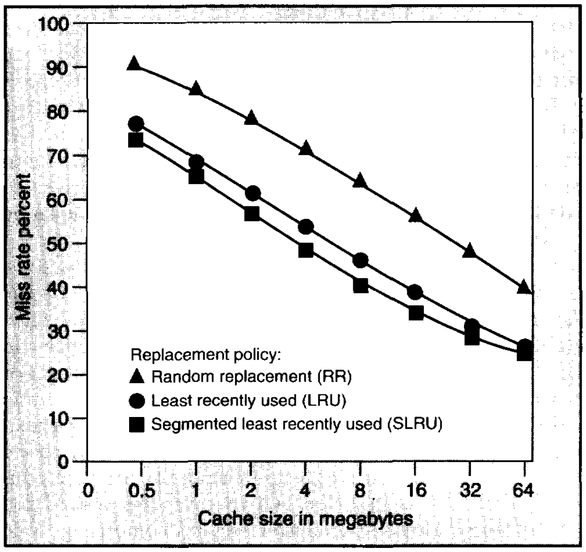


Figure 4. Replacement policy performance for the inventory control (IC) workload.

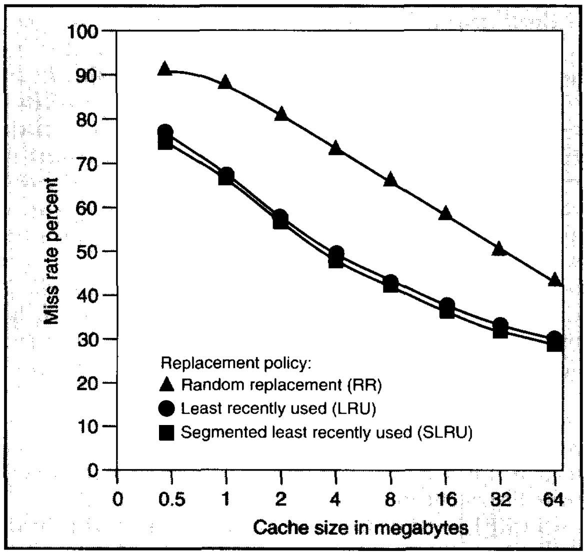


Figure 5. Replacement policy performance for the batch processing (BP) workload.

and a small amount of database processing using Digital's relational database product (Rdb). This trace was started after midnight and continued till 6:30 p.m. the next day. We selected an interval from 8 a.m. to 5 p.m. for our analysis.

Observations

The average request size for the workloads was small (up to 6 Kbytes). In all of the simulations, the caches filled up very early in the trace, typically in less than one simulated hour.

RR always performed worse than LRU, probably because most requests were for more than one cache line, so that a request would miss if even a single block had been evicted by RR. (None of the workloads exhibited such poor locality of reference that RR performed better than LRU, although RR performance did parallel LRU as the cache became very large.)

To compare the performance of RR, LRU, and SLRU, we chose a protected segment size that maximized SLRU performance. Examining Figures 4 through 7, we find that the maximal gain in cache performance is at smaller cache sizes.

The curves for the IC and BP workloads (Figures 4 and 5) look very similar, although the workloads differ in detail. IC is characterized by random-read accesses to database records, while BP accesses are more sequential because they are primarily backup and document processing tasks. For this reason, a read cache that would benefit IC would not benefit BP to the same extent because the application is different. For both applications, the greatest benefit occurs with cache sizes up to 8 Mbytes. Interestingly, the miss rate drops to about 30 percent for a 32-Mbyte IC cache as compared to a 64-Mbyte BP cache.

The curves for the AR workload (Figure 6) may indicate that small caches are overwhelmed by the nature of the workload. Each travel agency was connected to the system via dial-up lines to one user process on the system until the end of the working day. The AR workload had as many as eight paging and swapping disks to accommodate the many processes, and we believe that paging and swapping activity floods the smaller caches. Modified blocks were not immediately accessed, which shows that our policy to allocate on write was not effective for this workload. Judging from the above experience, we believe that page and swap disks do not make good candidates for caching.

For the AR and ST workloads (Figures 6 and 7), we notice that up to 8 Mbytes, an SLRU cache performs almost as well as an LRU cache twice its size. (Similar effects were noted for other workloads not discussed in this article.) An 8-Mbyte cache for the ST workload seemed to perform like a 64-Mbyte cache for the AR workload, clearly showing the dependency of cache performance on the type of application.

For the ST workload (Figure 7), the greatest benefit seemed to be for cache sizes up to 16 Mbytes. Both the LRU and SLRU performed consistently better than RR. Miss rates were around 20 percent.

In all of the workloads studied, there was a range where SLRU outperformed LRU, and in a number of cases the SLRU miss rate was close to that of an LRU cache twice its size. (Similar results have been reported for the LRU-K algorithm[5] for database page buffering.) For small caches, this may be due to skewness in accesses to database hot spots, but our analysis of the AR and ST trace characteristics suggests that such behavior is not limited to database accesses.

The SLRU parameter was varied from 1/16 to 15/16 of the total cache size in steps of 1/16th. Performance appears to increase linearly as the size of the protected segment is increased, especially for small cache sizes. We notice that the maximum benefit is obtained when the size of the protected segment is between 60 to 80 percent of the entire cache. This would tend to indicate the benefit of making access-frequency-based replacement decisions. For the AR workload, SLRU showed an improvement of 12 percent over LRU and improvements of up to 10 percent for the other workloads.

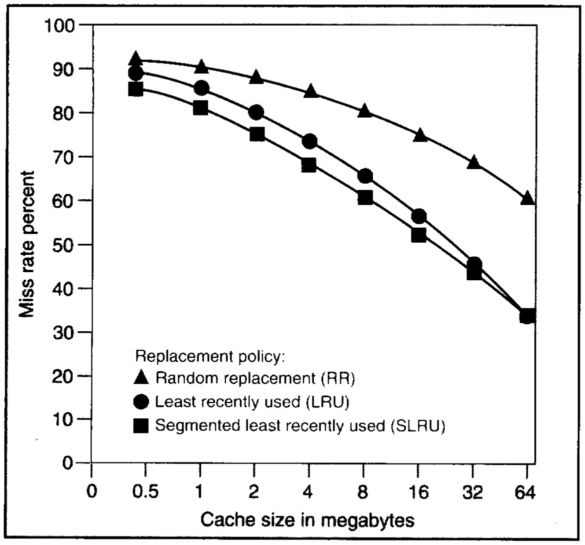


Figure 6. Replacement policy performance for the airline reservation (AR) workload.

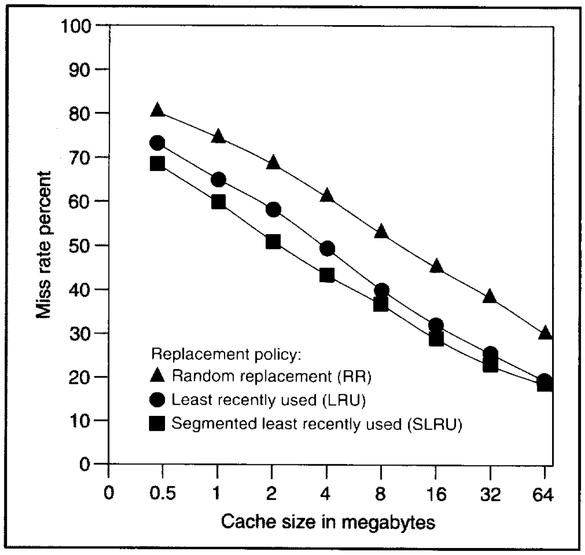


Figure 7. Replacement policy performance for the scientific time sharing (ST) workload.

Cache performance, as measured by miss rate, seems to increase with cache size. The cache size required to achieve a given performance level varies dramatically with the workload. For the workloads analyzed, the knee in the curve (where visible), occurs between 8 Mbytes and 32 Mbytes.

Although LRU is a well-known policy with low implementation overhead, we find that for small cache sizes SLRU halves the cache size for a given miss rate. SLRU is recommended for cache implementations in the range of 2 to 8 Mbytes, because SLRU performs best on workloads that overwhelm LRU caches.

However, caching is dependent on the type of system workload and should be implemented with caution. Write-intensive applications and applications that do not reuse data are not likely to benefit from caching.

As we have seen, there are no clear-cut solutions when it comes to caching. The chief metric of cache performance is its miss rate, but we must also take into account its impact on overall system performance. A good algorithm must not only show a significantly low miss rate, but must also be easy to implement (although this restraint may be relaxed in the near future with increasing processor power and falling prices for processors and RAM).

Current trends indicate that future I/O caches will be self-adaptive. More intelligence will be put into cache management, and caches will adapt to the changing access patterns of the user. A number of cache products are sold today with cache performance monitoring tools that assist one in making informed decisions about which storage devices to cache or whether to vary the read/write cache threshold, the cache line size, and so forth. Cache hints that are passed from the controller to the host and vice versa are likely to be incorporated into cache management algorithms in the near future. ■

Acknowledgments

We thank Joe Falcone for motivating this effort, developing the FSTrace tracing package for the VMS operating system, and collecting the above-mentioned workload traces. We also thank Richard Lary and Edgar Quiet for their support on this project and acknowledge other members of the Storage Systems Architecture Group at Digital Equipment for their valuable suggestions and comments. Finally, we thank the anonymous referees for their suggestions and constructive feedback.

References

1. A.J. Smith, "Cache Memories," *ACM Computing Surveys*, Vol. 14, No. 3, Sept. 1982, pp. 473-530.

2. S.A. Przybylski, *Cache and Memory-Hierarchy Design*, Morgan Kaufman Publishers, San Mateo, Calif., 1990, pp. 45-134.

3. A.J. Smith "Disk-Cache Miss-Ratio Analysis and Design Considerations," *ACM Trans. Computer Systems*, Vol. 3, No. 3, Aug. 1985, pp. 161-203.

4. J.T. Robinson and M.V. Devarakonda, "Data Cache Management Using Frequency-Based Replacement," *Performance Evaluation Rev.*, Vol. 18, No. 1, May 1990, pp. 134-142.

5. E.J. O'Neil, P.E. O'Neil, and G. Weikum, "The LRU-K Page Replacement Algorithm for Database Disk Buffering," *Proc. ACM SIGMOD Conf.*, ACM Press, New York, 1993, pp. 297-306.

6. K. Bates, *VAX I/O Subsystems: Optimizing Performance*, Professional Press Books, Horsham, Pa., 1991, pp. 81-90.

7. R. Geist and S. Daniel, "A Continuum of Disk Scheduling Algorithms," *ACM Trans. Computer Systems*, Vol. 5, No. 1, Feb. 1987, pp. 77-92.

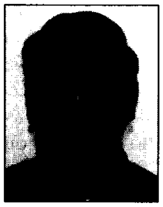
8. P. Biswas, K.K. Ramakrishnan, D. Towsley, "Trace-Driven Analysis of Write-Caching Policies For Disks," *Performance Evaluation Rev.*, Vol. 21, No. 1, June 1993, pp. 13-23.

9. Data Technology Corp., *DTC3290AS/-HD EISA to SCSI Adapter*, Milpitas, Calif., 1993.

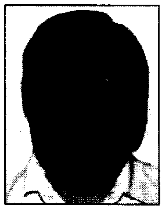
10. Zitel Corp., *CASD 1000/2000*, Fremont, Calif., 1993.

11. B. McNutt, "I/O Subsystem Configuration for ESA: New Roles for Processor Storage," *IBM Systems J.*, Vol. 32, No. 2, 1993, pp. 252-264.

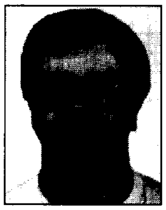
12. K.K. Ramakrishnan, P. Biswas, and R. Karedla, "Analysis of File I/O Traces in Commercial Computing Environments," *Performance Evaluation Rev.*, Vol. 20, No. 1, June 1992, pp. 78-90.



Ramakrishna Karedla is a software engineer at Digital Equipment Corporation in Shrewsbury, Massachusetts, where he has been working on the advanced development of storage products for the past six years. His technical interests include I/O subsystems, distributed file systems, performance modeling of computer systems, and speech processing. He received his master's degree in communications and signal processing from Northeastern University in 1988. He is a member of the IEEE and the ACM.



J. Spencer Love is a studio sound engineer and entrepreneur in Milford, Massachusetts. Previously, he was a principal software engineer in Digital Equipment Corporation's Storage Architecture group in Shrewsbury, Massachusetts, and was a developer of the Multics Operating System and the relational database system (RDMS) at the Massachusetts Institute of Technology. His technical interests include operating systems, networks, database systems, performance optimization, programming languages, and runtime systems. He attended M.I.T from 1973 to 1976. His memberships include the IEEE, ACM, AES, Boston Computer Society, and the League for Programming Freedom.



Bradley G. Wherry is the network services manager for McREL (Mid-continent Regional Educational Laboratory), one of 11 regional nonprofit educational research laboratories where he is investigating and implementing information services on the Internet. Prior to this he worked for US West Enhanced Services as a systems engineer. He also spent five years working for storage systems groups at Digital Equipment Corporation. He received a bachelor's degree in computer science from the Worcester Polytechnic Institute in 1988.

Readers can contact the authors through Ramakrihna Karedla at Digital Equipment Corp. 334 South St., SHR3-2/W3, Shrewsbury, Mass. 01545; e-mail ramakr@dec.com.