

Design and Performance of a General-Purpose Software Cache

Arun Iyengar
IBM Research
T. J. Watson Research Center
P. O. Box 704
Yorktown Heights, NY 10598

Abstract

This paper describes a General-Purpose Software cache (GPS cache) which can improve the performance of many applications including Web servers and databases. It can service several hundred thousand cache hits per second on a uniprocessor. When used to cache data for a Web server accelerator, the overhead due to the GPS cache was an insignificant factor in the overall performance of the system. The GPS cache can store objects in memory, on disk, or both. The cache uses a new algorithm for managing expiration times of cached objects which is more efficient than previous ones. The GPS cache uses Data Update Propagation (DUP) to invalidate complex objects which is crucial for caching and maintaining updated copies of dynamic Web pages. Transactions can be logged using different buffering mechanisms in order to provide a balance between efficiency and currency of transaction log files. The GPS cache provides API functions which allow applications to directly manipulate its contents.

1 Introduction

Software caches are a critical component in improving the performance of many applications including Web servers, databases, and file systems. In order for an application to benefit from caching, it must repeatedly use data which is expensive to calculate or fetch. By caching such data, the application only needs to calculate or fetch the data once. Whenever the data is needed after it has been cached, the application can fetch the data from the cache instead of recalculating it or fetching it from a remote location.

Software caches are generally several order of magnitudes slower than processor caches. Processor caches can return data in several nanoseconds. By contrast, caches for dynamic Web pages on Web servers can provide near-optimal speedup by handling several hundred hits per second [6]. Since software caches can afford to be much slower than processor caches, software caches can provide more features such as complex invalidation and replacement strategies as well as logging cache transactions.

This paper describes a General-Purpose Software Cache (GPS Cache) designed to improve performance of a wide variety of applications such as Web servers and databases. Applications add, delete, and query the cache via a set of API function calls. The GPS cache can be configured to store data in memory, on

disk, or both. A common mode of operation is to use disk as secondary storage for cached objects which cannot fit in memory due to the presence of other cached objects which are accessed more frequently. A single processor running a GPS cache can service several hundred thousand cache hits per second.

We have used the GPS cache for storing data in a Web server accelerator. The overhead introduced by the cache was an insignificant factor in the overall performance of the Web server accelerator. A detailed analysis of the performance of the Web server accelerator is contained in [9]. We are also deploying the GPS cache for improving performance in distributed object-oriented applications.

Cached objects can have expiration times associated with them after which they are no longer valid. The GPS cache implements a new efficient algorithm for invalidating objects based on expiration times. In addition, API functions allow application programs to explicitly invalidate cached data.

The GPS cache implements the data update propagation (DUP) algorithm for invalidating cached objects [3, 7]. This feature is useful for keeping complex data objects current in the cache. DUP has proved to be extremely useful for caching dynamic Web pages.

The GPS cache allows all transactions to be logged in a file. A cache transaction is any invocation of a cache API function by the application program. In order to reduce the overhead for logging, the frequency with which buffers containing transaction information are flushed to the file system can be varied. If every transaction record is flushed to disk as soon as it is generated, log files will always be up to date and no logs will be lost if the cache process fails. The overhead for immediately flushing every transaction log is substantial, however. An alternative approach is to accumulate several transaction records in a buffer before flushing the buffer to disk. This approach has lower overhead. If the cache process fails, however, transaction records which have not yet been flushed to disk are lost.

2 Cache Architecture

The GPS cache is a POSIX-compliant C++ library. In order to use the GPS cache, an application uses the GPS cache API's to manage the cache and is linked with the GPS cache library. API functions exist to add, delete, or change objects, look up objects and

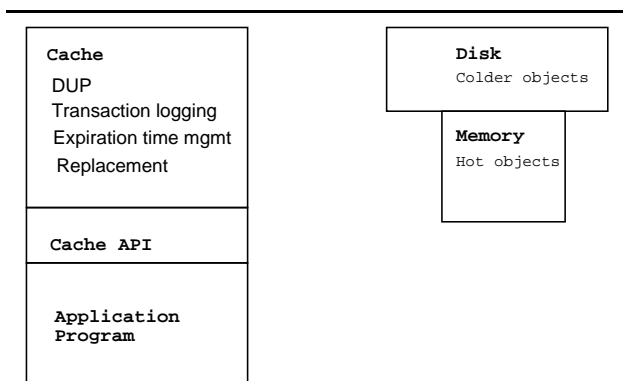


Figure 1: Key features of the GPS cache.

information about objects, change information about objects such as expiration times, and to modify cache parameters such as the maximum size of a cacheable object. API functions also exist to implement DUP.

Cache configuration parameters specify how much memory and disk to use for the cache. Objects cached in memory can be accessed much more quickly than objects cached on disk. The GPS cache attempts to keep the hottest objects in memory. It uses the Least-Recently-Used algorithm (LRU) to determine which objects should be cached in memory as well as which objects should be deleted from the cache in order to make room for new ones.

LRU is implemented by maintaining a doubly linked list ordering cached objects by the time of most recent access. Pointers to the back of the list, front of the list, and the least recently used object in memory (*lru_in_memory*) are maintained. Whenever an object is accessed, it is moved to the front of the LRU list. If the object was previously cached on disk but not in memory, the object is copied into memory. A configurable cache parameter specifies whether the copy on disk for such objects should be retained after the object is brought into memory. Retaining the copy on disk improves performance if the cached copy is pushed out of memory multiple times. Deleting the copy on disk saves disk space.

When an object is added to the cache, the application program can specify the position of the object on the LRU list. The current options are at the front of the LRU list, at the back, or adjacent to the least recently used object in memory which can be found by examining the *lru_in_memory* pointer. If caching the object results in a memory overflow, the *lru_in_memory* pointer is examined to move enough of the least recently used objects from memory to disk to make room for the new object. Transferring these objects from memory to disk may cause the disk portion of the cache to overflow as well. If so, objects are removed from the cache starting from the end of the LRU list.

When an object is cached or modified, the application program can specify a size for the buffer containing the cached object which is larger than the object

size. While this wastes space, it can save time in a situation where a variable sized object is constantly being overwritten. If the buffer size needs to be changed during each overwrite, considerable time can be spent deallocating the old buffer and allocating a new buffer. By contrast, if a buffer large enough to handle all sizes is created, it should suffice for all overwrites. When an application program modifies a cached object, the API provides two methods for changing the size of the buffer containing the cached object:

- Only allocate a new buffer for the object (and deallocate the old one) if the existing buffer is too small to contain the new data.
- Allocate a new buffer for the object (and deallocate the old one) any time the existing buffer size is different from the preferred buffer size for the new data.

Information about each cached object is maintained in an *object information block* (OIB) which is implemented as a C++ class. Information contained in an OIB includes the key used to reference an object, the size of the object, a pointer to the buffer containing the object, etc. Pointers to OIB's are maintained in a hash table indexed by object keys [1]. That way, an object can be fetched in constant time regardless of the number of objects in the cache.

The cache operates in one of two modes: *fast mode* and *safe mode*. In fast mode, a pointer to the OIB is returned without any copying. This results in extremely fast lookups. A key advantage of fast mode is that the time to return an item from the cache does not grow with the object size. While fast mode suffices for applications which read but do not modify cached objects, it is inconvenient for applications which modify cached objects because the application must make a new copy of a cached object after looking it up. Another problem with fast mode is that it exposes part of the cache's internal data structures to the application. This makes it possible for an application developer to introduce bugs by modifying data which should only be modified by the cache.

For these reasons, the GPS cache also provides safe mode which makes a copy of a cached object after a lookup. Safe mode is slower than fast mode. In addition, lookup times increase with the size of the object being returned. However, safe mode is more convenient if the application needs to modify objects returned from cache. Safe mode also protects internal cache data structures from being modified by the application program.

2.1 Managing Expiration Times

The GPS cache allows the application to optionally specify an expiration time for a cached object. After the expiration time for an object has passed, the object is no longer valid and can be deleted from the cache. Expiration times are used to invalidate objects in proxy caches on the World Wide Web today.

There are a number of methods for managing expiration times. A simple method is to maintain an expiration list of objects ordered by expiration times

and to garbage collect expired objects by scanning the list. In order to allow an object to be deleted from the cache before its expiration time in constant time, the list should be doubly linked. The drawback to this approach is its inefficiency. Objects are not necessarily added to the cache in the order in which they expire. Therefore, adding an object to the cache can become expensive because the expiration list position for the object must be determined. If there are n cached objects with expiration times, caching a new object with an expiration time requires $O(n)$ instructions.

An alternative is to use a balanced tree ordered by expiration times which we shall refer to as an *expiration tree*. Using this approach, both adding an object to the cache with an expiration time and deleting an object from the cache with an expiration time require $O(\log(n))$ instructions. The cache periodically garbage collects objects which have expired. During a garbage collection, several objects may need to be removed from the expiration tree at a time. Using balanced tree algorithms provided in textbooks, garbage collecting k objects would require $O(\log(k * n))$ instructions [5, 1, 8].

One method for obtaining more efficient garbage collections is to use a B-tree in which data are stored in leaf nodes of the expiration tree. Such B-trees are known as B^+ -trees [4]. By contrast, *basic B-trees* store data within internal nodes of the tree. By linking the leaf nodes of a B^+ -tree together, it is easy to examine cached objects in the order in which they expire. It is also possible to delete k objects from an expiration tree during a garbage collection by performing a single restructuring of the tree because all objects being deleted would correspond to consecutive leaf nodes at one end of the tree. Using this approach, deletion of the objects from the expiration tree would require $O(k + \log(n))$ instructions.

The GPS cache uses a different method for garbage collection in which objects with expiration times can be inserted and deleted from the cache in constant time. In addition, the data structures used for managing expiration times are easier to implement than balanced trees.

Cached objects are grouped by expiration times. Each group of objects having similar expiration times is known as an *expiration group*. Each expiration group corresponds to objects having expiration times between an interval starting at a time t and ending at a time $t + \text{expiration_interval_width}$, where *expiration_interval_width* is a cache parameter which can be varied. Time intervals corresponding to different expiration groups never overlap. The expiration group corresponding to the current time is known as the *current expiration group*.

Expiration groups are stored in hash tables. Empty expiration groups don't consume any space. Each nonempty expiration group is represented as a doubly linked list containing pointers to the cached objects. The double linking allows any cached object to be removed from an expiration group in constant time.

The maximum number of objects which can be reclaimed during a single garbage collection is bounded

by the cache parameter *max_reclaimable_per_gc*. This parameter limits the delay in cache response times due to garbage collections. When the cache enters a garbage collection phase, it invalidates up to *max_reclaimable_per_gc* obsolete objects by starting with objects in the oldest expiration group which has not been completely reclaimed and continuing until the expiration group corresponding to the current time is reached. No objects in the current expiration group are invalidated. This is because only some of these objects will have expired. Garbage collecting the current expiration group is more efficient if done at a later stage when the group is no longer current and the cache is certain that all of these objects have expired.

If the garbage collection succeeds in invalidating all objects in expiration groups preceding the current one, another garbage collection does not take place for a while. If, on the other hand, the garbage collection failed to invalidate all objects in expiration groups preceding the current one, another garbage collection takes place after the next request.

Garbage collections also occur after the cache cannot store a new object due to insufficient space. In this case, the garbage collection ends when enough space has been freed up for the new object. In this type of garbage collection, the cache will attempt to invalidate objects in the current expiration group if all previous expiration groups have been garbage collected and insufficient space has been reclaimed. If garbage collecting the current expiration group still fails to clear up sufficient space, the cache resorts to a replacement algorithm such as LRU.

2.2 Invalidation Algorithm

In many cases, the value of a cached object is dependent on underlying data. When the underlying data changes, one or more cached objects might become stale as a result of the changes. A method is needed to propagate the changes to the cache. For example, the cache may contain dynamic Web pages which are constructed from databases [6]. In this situation, a method is needed to determine which Web pages are affected by updates to the database. That way, caches can be synchronized with databases so that they do not contain stale data. Furthermore, the method should associate cached pages with parts of the database in as precise a fashion as possible. Otherwise, objects whose values have not changed may be mistakenly invalidated or updated from a cache after a database change. Such unnecessary updates to caches can increase miss rates and hurt performance.

The GPS cache implements an algorithm for cache invalidation known as *Data Update Propagation (DUP)* [3, 7]. DUP has been a critical component in improving performance at Web sites serving a high percentage of dynamic pages. At the official Web site for the 1998 Olympic Winter Games, DUP improved cache hit rates from 80% to near 100% [2]. The basic approach is to maintain correspondences between *objects* which are defined as items which may be cached and *underlying data* which periodically change and affect the values of objects. Although an entity may be

both an object as well as underlying data, objects and underlying data could also be different, which would mean that underlying data are not cacheable.

When DUP is being used, the GPS cache maintains data dependence information between objects and underlying data. When the cache becomes aware of a change to underlying data, it queries the dependence information which it has stored in order to determine which cached objects are affected. Caches use dependency information to determine which objects should be invalidated as a result of changes to underlying data.

The application program is responsible for communicating data dependencies between underlying data and objects to the cache. Such dependencies can be represented by a directed graph known as an *object dependence graph*, wherein a vertex usually represents an object or underlying data. An edge from a vertex v to another vertex u denoted (v, u) indicates that a change to v also affects u . Node v is known as the *source* of the edge, while u is known as the *target* of the edge. For example, if node $go2$ in Figure 2 changes, then nodes $go5$ and $go6$ also change. By transitivity, $go7$ also changes. Edges may optionally have weights associated with them which indicate the importance of data dependencies. In Figure 2, the data dependence from $go1$ to $go5$ is more important than the data dependence from $go2$ to $go5$ because the former edge has a weight which is 5 times the weight of the latter edge.

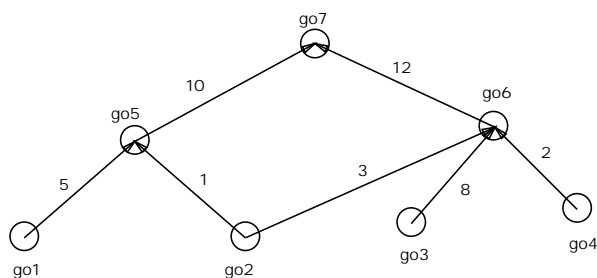


Figure 2: An object dependence graph. Weights are correlated with the importance of data dependencies.

In many cases we have encountered, the object dependence graph is a *simple object dependence graph* having the following characteristics:

- Each vertex representing underlying data does not have an incoming edge.
- Each vertex representing an object does not have an outgoing edge.
- All vertices in the graph correspond to underlying data (nodes with no incoming edges) or objects (nodes with no outgoing edges).
- None of the edges have weights associated with them.

Figure 3 depicts a simple object dependence graph. This paper explains how DUP works for simple object

dependence graphs. A method for generalizing DUP to handle any object dependence graph is contained in [3, 7].

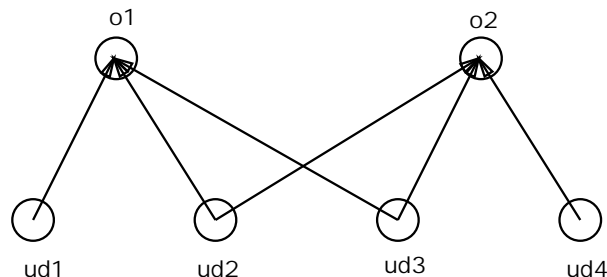


Figure 3: A simple object dependence graph.

The application program must determine an appropriate correspondence between underlying data and vertices of the object dependence graph G . For example, a vertex corresponding to underlying data may represent a database table. Another vertex corresponding to underlying data may represent portions of several database tables. There are no restrictions on how underlying data may be correlated with nodes of G . The application program has freedom to pick the most logical and/or efficient system.

Each object has a string obj_id known as the object ID or key which identifies the object. Similarly, each node representing underlying data has a string ud_id known as the underlying data ID which identifies it. An application program informs the cache that an object has a dependency on underlying data via an API function:

```
add_dependency(obj_id, ud_id)
```

Whenever underlying data corresponding to a node in G changes, the application program notifies the cache via an API function:

```
underlying_data_has_changed(ud_id)
```

The cache then invalidates all cached objects having dependencies on ud_id . Referring to Figure 3,

```
underlying_data_has_changed(ud4)
```

would cause $o2$ to be invalidated. The function call:

```
underlying_data_has_changed(ud2)
```

would cause both $o1$ and $o2$ to be invalidated.

The OIB for each object with one or more dependencies on underlying data includes the object ID and an *incoming adjacency list* containing all underlying data ID's corresponding to underlying data which affect the value of the object. Figure 4 depicts the incoming adjacency lists corresponding to the graph in Figure 3.

The cache also maintains a hash table containing pointers to *outgoing adjacency lists* for nodes in G

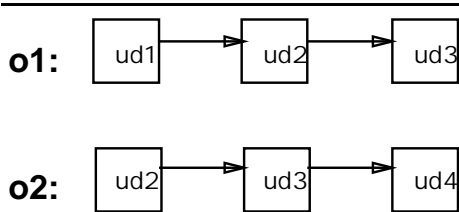


Figure 4: Incoming adjacency lists corresponding to the graph in Figure 3.

corresponding to underlying data. The hash table is indexed by underlying data ID's. Each outgoing adjacency list contains the object ID's of objects whose values depend on the underlying data represented by the underlying data ID. Figure 5 depicts the hash table corresponding to the graph in Figure 3.

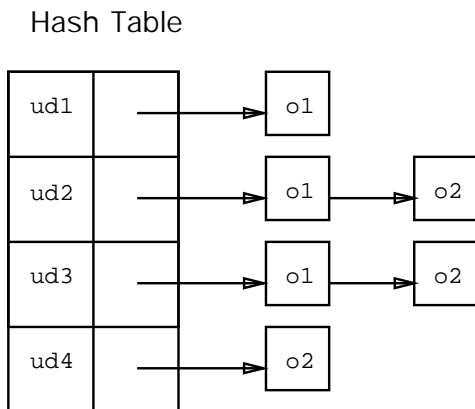


Figure 5: The hash table corresponding to the graph in Figure 3.

An invocation of the API function:

add_dependency(obj_id, ud_id)

adds a new edge to G by adding obj_id to the outgoing adjacency list for ud_id and ud_id to the incoming adjacency list for obj_id .

An invocation of the API function:

underlying_data_has_changed(ud_id)

is implemented by invalidating or updating all objects on the outgoing adjacency list for ud_id .

It is sometimes desirable to delete nodes from G . For example, all dependencies on an underlying data node could become obsolete in which case the node would no longer be needed. Similarly, an object could go away which would make its node in G unnecessary. Object nodes are removed from G by removing the object ID from outgoing adjacency lists for all nodes on

the incoming adjacency list for the object and removing the incoming adjacency list for the object. Underlying data nodes are removed from G by removing the underlying data ID from incoming adjacency lists for all nodes on the outgoing adjacency list for the underlying data node and removing the hash table entry for the underlying data node.

DUP adds very little processing overhead to the cache. In particular, CPU cycles for cache lookups are unaffected.

2.3 Transaction Logging

The GPS cache provides a flexible method for recording cache transactions in a log file. The application program can specify which cache API calls should be logged. For example, if cache reads are being logged, each read request generates a transaction record which contains the object ID requested and the size of the object found. A size of 0 would indicate a cache miss.

Cache records are written to disk using the standard C library call *fwrite*. Invoking *fwrite* does not automatically flush a record to disk. The operating system may buffer output from several *fwrite* function calls before flushing multiple transaction records to disk. The application program has the option of specifying that output should be sent to disk immediately after each call to *fwrite*. This is accomplished by calling *fflush* after each call to *fwrite*. In general, relying on the operating system to flush buffers to disk results in better performance. Flushing buffers after each call to *fwrite* will prevent transaction records from being lost in the event of a failure.

The application can also control how many transaction records need to accumulate in a buffer before calling *fwrite*. Accumulating more transaction records before calling *fwrite* results in better performance but potentially more lost records in the event of a failure.

3 Cache Performance

The performance numbers presented in this section were obtained on an IBM RS/6000 Model 590 workstation running AIX version 4.2.0.0. This machine contains a 66 Mhz POWER2 processor and comprises one node of an SP2 distributed-memory multiprocessor. The graphs in this section plot performance as the average CPU time to satisfy a request. In all of the experiments used to produce the graphs, the CPU was operating at near 100% capacity. Thus, the number of requests per unit of real time which can be satisfied is obtained by taking the reciprocal of the numbers in the graphs.

The GPS cache can return several thousand objects per second. When all objects are in memory, logging is disabled, and the cache is operating in fast mode, the GPS cache can return hundreds of thousands of objects per second. This kind of performance allows the GPS cache to store data for Web servers while consuming an insignificant number of cycles relative to the Web server. A high-performance Web server can serve a maximum of several hundred Web pages per second on an SP2 node before driving the CPU to 100% utilization.

In fast mode, the time to lookup and return objects is independent of object size because no copying takes place. Key size has some effect on performance. However, the cache results in good performance even with relatively long keys. The effect of key size on cache performance is shown in Figure 6. The curve labelled *hits* represents the average time to find and return a pointer to an object in microseconds. Unless noted otherwise, all times in this section are in microseconds. The curve labelled *miss penalty* represents the time required by the GPS cache to determine that an object is not in the cache.

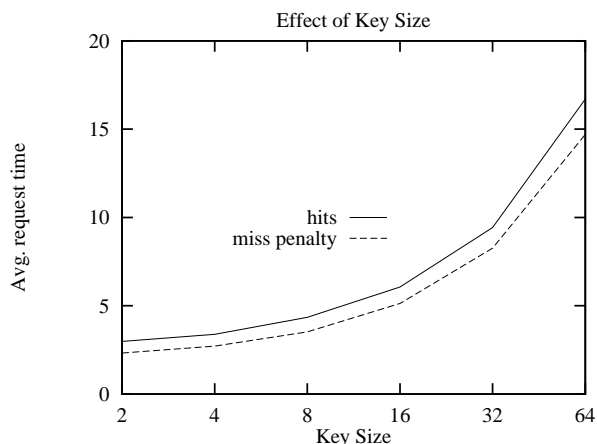


Figure 6: The effect of key size in fast mode on cache lookups when the hashing function examines each character. Times are in microseconds.

The increase in request time with increasing key size in Figure 6 is primarily due to hashing overhead; all characters of the key are used for hashing. The GPS cache uses a simple hashing algorithm which computes a numerical value for a key by examining characters of the key, computing a character number for each examined character based on both the character and its position, and adding the character numbers together (modulo the maximum integer size) to get a hash sum. The hash sum modulo the hash table size gives the hash table bucket for the key. In the graph in Figure 6, a maximum of one object was stored in each hash bucket. Figure 7 shows the effect of storing multiple objects in a hash bucket.

The overhead due to hashing is not very large, even if hashing is performed on 64 characters. In order to bound the hashing overhead, the number of characters used to calculate hash values can be set to a maximum independent of the key size. This has the negative effect of increasing the probability of two keys hashing to the same value which is known as a *collision*.

In Figure 7, only five characters are examined for hashing regardless of key size. The degradation in performance due to increasing key size is less than in Figure 6. This graph also shows the effect of storing multiple objects in the same bucket. Each nonempty

bucket of the hash table contains a linked list of pointers to objects. By choosing appropriate hashing functions and hash table sizes, the expected size of a hash table list can be kept small. The variable E used to label the curves in Figure 7 is the average number of objects examined in a bucket per request. On average, a request resulting in a cache hit examines half of the objects in the appropriate bucket which must be nonempty. If l is the average size of all nonempty lists in the hash table, we can approximate the average number of objects examined in a cache hit by:

$$E_{hit} \approx \frac{l}{2}.$$

A request resulting in a cache miss examines every object in the appropriate bucket which could be empty. If n is the number of cached objects and h is the hash table size, we can approximate the average number of objects examined in a cache miss by:

$$E_{miss} \approx \frac{n}{h}.$$

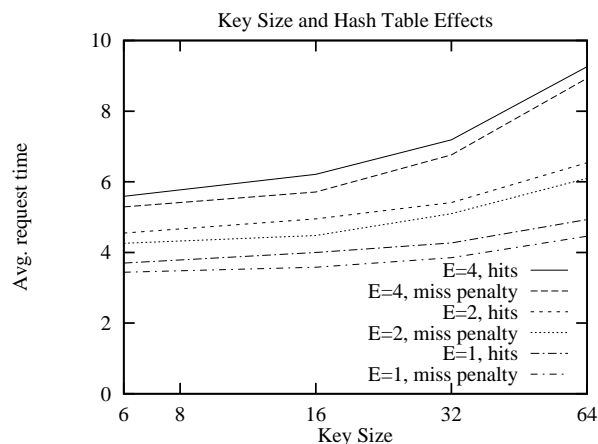


Figure 7: The effect of key size in fast mode on cache lookups when the hashing function only examines five characters. E is the average number of objects examined in a bucket per request.

The remaining performance figures in this section were generated using a key size of six, hashing by examining five of the key characters, and storing at most one object in a hash bucket. It is straightforward to calculate the effect of other choices of these parameters from Figures 6 and 7.

Figure 8 shows the average request time in microseconds for transactions which overwrite the old value of an object with a new value. Since cache writes require copying data to the cache, write times increase with object size unlike lookups in fast mode. The curve marked *new buffer* corresponds to situations in which a new buffer is created for the overwrite and the old buffer is deallocated. This might be required

if the size of the new object is different from that of the old object. The curve marked *same buffer* corresponds to situations for which the same buffer is used to hold both the old and new object. This might be the situation if the original buffer were large enough to contain both the old and new object.

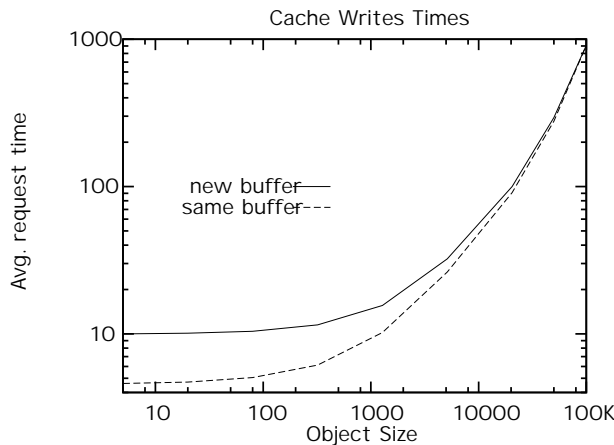


Figure 8: CPU time for cache writes.

Figure 9 shows the average request time in microseconds for cache lookups in safe mode. Copies of objects are made and returned. Because of the copying, the overhead for returning a cached object grows with object size. The overhead for looking up an object in safe mode is comparable to the overhead for overwriting an object with a new value. Thus, the graph in Figure 9 is similar to the curve marked *same buffer* in Figure 8.

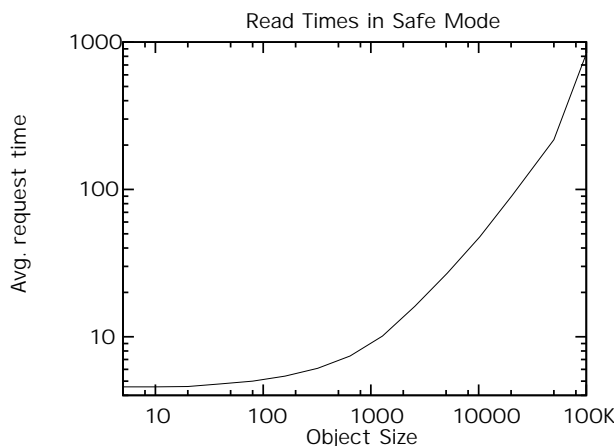


Figure 9: CPU time for cache lookups in safe mode.

Figure 10 shows the effect of transaction logging on average request times in fast mode. Records corresponding to cache transactions are placed in a buffer. After a certain number of records have accumulated,

the entire buffer is sent to disk using the standard C library function *fwrite*. The X-axis represents the number of records which must accumulate in the buffer before being written to disk via *fwrite*. The *fwrite* function doesn't immediately force the buffer to disk. In the curve labelled *immediate flush*, the cache forces the buffer to disk immediately after each *fwrite* via *fflush*. In the curve marked *deferred flush*, the cache relies on the operating system to flush data to disk.

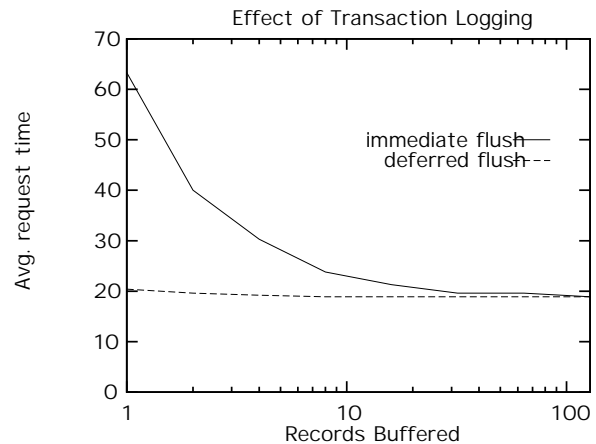


Figure 10: Effect of transaction logging.

Relying on AIX to flush transaction logs to disk results in good performance. Using this approach, little is gained by buffering several transaction records before sending them to disk via *fwrite*. The cache would explicitly flush transaction records to disk after every call to *fwrite* if it is crucial not to lose the records in the event of a failure. However, flushing records to disk after only a small number of transactions entails a high overhead.

The overhead for transaction logging is dependent on the operating system. AIX efficiently flushes output to disk. If lost transaction records in the event of a failure are not an issue, good performance can be achieved by writing single transaction records to disk without buffering and relying on AIX to flush the transaction records to disk. By contrast, on an operating system with a less efficient I/O implementation, more performance gains might be achieved by having the cache buffer several transactions before writing them to disk.

All of the performance numbers discussed so far are for situations when objects were cached in memory. Figure 11 shows the average request time for lookups in fast mode when 20% of the requests are satisfied from disk. As soon as an object cached on disk but not in memory is accessed, the object is brought into memory and moved to the front of the LRU list. Doing so may cause one or more of the least recently used objects in memory to be moved out of memory.

Most of the overhead in Figure 11 results from copying objects from disk into memory. After an object was moved into memory from disk, the copy on

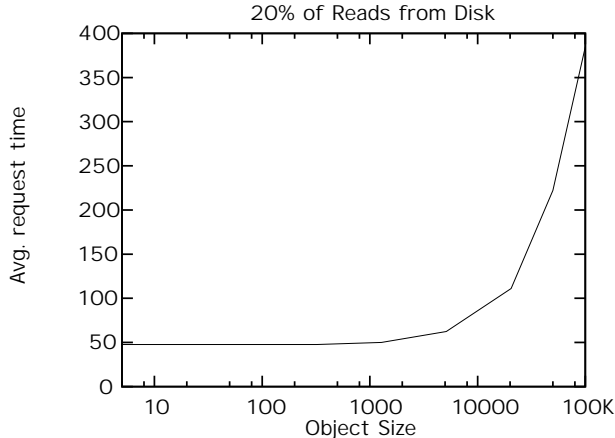


Figure 11: Lookup performance when 20% of requests are satisfied from disk.

disk was not deleted. As a result, little work was required if the object was later forced out to disk again. In the test used to generate Figure 11, files were not being created or deleted after a steady state was reached.

In the experiments used to generate Figures 10 and 11, CPU utilization was near 100% even though data were being transferred to and from disk. We did encounter situations where I/O caused CPU utilization to drop considerably. One situation was when the files being read from and written to were not local to the machine but were instead mounted remotely via NFS. Another situation was when caching objects on disk resulted in the creation and deletion of many files. In such situations, better CPU utilization can be achieved by using a multithreaded cache such as the one described in [6]. However, for most of the situations we encountered, we were able to achieve near 100% CPU utilization. The GPS cache, which is single threaded, is preferable to a multithreaded cache in situations when it can achieve high CPU utilization because of the greater efficiency, portability, and simplicity which a single threaded design entails.

4 Conclusion

The GPS cache is a high-performance cache which can service several hundred thousand cache hits per second on a single processor. It can be used to improve performance for a wide variety of applications such as Web servers and databases. When used to cache data for a Web server accelerator, the overhead due to the GPS cache was an insignificant factor in the overall performance of the system. We are also deploying the GPS cache for improving performance in distributed object-oriented applications.

The GPS cache can store objects in memory, on disk, or both. The cache uses a new algorithm for managing expiration times of cached objects which is more efficient than previous ones. The GPS cache uses DUP to invalidate complex objects which is crucial for

caching and maintaining updated copies of dynamic Web pages. Transactions can be logged using different buffering mechanisms in order to provide a balance between efficiency and currency of transaction log files.

Acknowledgements

Many of the ideas behind the GPS cache were provided by Jim Challenger. Tracy Kimbrel provided key ideas behind the expiration time algorithm used by the GPS cache.

References

- [1] A. Aho, J. Hopcroft, and J. Ullman. *The Design and Analysis of Computer Algorithms*. Addison-Wesley, 1974.
- [2] J. Challenger, P. Dantzic, and A. Iyengar. A Scalable and Highly Available System for Serving Dynamic Data at Frequently Accessed Web Sites. In *Proceedings of ACM/IEEE SC98*, November 1998.
- [3] J. Challenger, A. Iyengar, and P. Dantzic. A Scalable System for Consistently Caching Dynamic Data. To appear in *Proceedings of IEEE INFOCOM'99*, March 1999.
- [4] D. Comer. The Ubiquitous B-tree. *ACM Computing Surveys*, 11(2):121-137, June 1979.
- [5] T. Cormen, C. Leiserson, and R. Rivest. *Introduction to Algorithms*. MIT Press, Cambridge, MA, 1990.
- [6] A. Iyengar and J. Challenger. Improving Web Server Performance by Caching Dynamic Data. In *Proceedings of the USENIX Symposium on Internet Technologies and Systems*, December 1997.
- [7] A. Iyengar and J. Challenger. Data Update Propagation: A Method for Determining How Changes to Underlying Data Affect Cached Objects on the Web. Technical Report RC 21093(94368), IBM Research Division, Yorktown Heights, NY, February 1998.
- [8] D. E. Knuth. *The Art of Computer Programming*, volume 3: Sorting and Searching. Addison-Wesley, Reading, MA, second edition, 1973.
- [9] E. Levy, A. Iyengar, J. Song, and D. Dias. Design and Performance of a Web Server Accelerator. To appear in *Proceedings of IEEE INFOCOM'99*, March 1999.