Purdue University
Purdue e-Pubs

Department of Computer Science Technical Reports

Department of Computer Science

1991

GCache: A Generalized Caching Mechanism

Douglas E. Comer
Purdue University, comer@cs.purdue.edu

Shawn Ostermann

Report Number:
91-081

Comer, Douglas E. and Ostermann, Shawn, "GCache: A Generalized Caching Mechanism" (1991). *Department of Computer Science Technical Reports.* Paper 920.
https://docs.lib.purdue.edu/cstech/920

This document has been made available through Purdue e-Pubs, a service of the Purdue University Libraries. Please contact epubs@purdue.edu for additional information.

GCACHE: A GENERALIZED CACHING MECHANISM

Douglas Comer
Shawn Ostermann

CSD-TR-91--081
November 1991
(Revised March 1992)

GCache: A Generalized Caching Mechanism

Douglas Comer
Shawn Ostermann

Computer Science Department
Purdue University
West Lafayette, IN 47907

November 22, 1991
Revised March 17, 1992

CSD-TR-91-081

Abstract

This paper discusses the issues and tradeoffs involved in designing a generalized caching mechanism suitable for use in an operating system kernel. As an example of this approach, we describe GCache, a caching system implementation providing entry insertion, lookup, deletion, and automatic timeout. GCache, designed and implemented for the Xinu Operating System, interfaces transparently into existing systems and provides a clean, concurrency-protected interface. GCache is also suitable for use in large user applications. This paper serves both as an example of a practical caching system and as an overview of the tradeoffs involved.

1 Introduction

It is widely understood that caching improves operating system efficiency and performance. Unfortunately, the types of information that benefit from caching take different forms and have different characteristics. These disparate forms often lead operating system designers to consider separate caching systems for each type of information. Separate caching systems require duplicated design effort, increase the possibility for error, and lead to multiple versions of semantically similar routines.

We designed GCache for the Xinu Operating System, a small, hierarchical operating system that runs on various Sun[1], DEC[2], and PC platforms [Com84, Com87, Com88]. Xinu supports threads, virtual memory, demand-loaded executable programs, and TCP/IP network communication. Many Unix[3] applications, including the GNU project software and the X11 Window System, run over Xinu using a Unix emulation library that resides outside the kernel.

The *expense* of an item of information is a measure of the amount of CPU time expended deriving the information or waiting for it to arrive. Throughout the remainder of this paper, we will use the term *expensive* information to refer to information that can be cached to improve the performance of an application or operating system. Under this definition, network communication generally produces expensive information. Even minimal attempts to cache such information can have a dramatic effect on system performance. Examples of expensive network information that can benefit from caching are name to address translations, RPC program port number bindings, and NFS file handles.

The remainder of this paper consists of 5 sections. In section 2, we present an overview of the most important issues involved, along with the goals for our design. Section 3 outlines our implementation of GCache including its interface and data structures. Section 4 discusses alternate design possibilities along with their associated tradeoffs. Section 5 presents results that show the benefits of our approach. The final section, section 6, summarizes generalized caching and presents our conclusions.

2 Design Goals

A programmer must consider many design issues and associated tradeoffs in any attempt to build a general-purpose mechanism. The decisions made during the design phase will greatly affect the generality of the resultant system. To further clarify the problem and to assist in the design of the system, this section outlines the goals that we adopted for a general-purpose caching mechanism.

Interface

The interface must be clean and concise; simple tasks must be easy to accomplish.

Transparency

The mechanism must be transparent. Interaction between the caching routines and the surrounding application should be limited to a few well-defined access routines.

[1]Sun, as used, is a trademark of Sun Microsystems, Inc.

[2]DEC is a trademark of Digital Equipment Corporation.

[3]Unix is a registered trademark of Unix Systems Laboratories.

Efficiency

The mechanism must use data structures and algorithms whose tradeoffs of complexity versus efficiency are appropriate for the size of the problem.

Number of Entries

The programmer should not be required to know, a priori, the maximum number of entries that will be inserted. The cache must be designed in such a way that it can accommodate arbitrarily many insertions. Insertion must always succeed, implying that the last item inserted is always present in the cache.

Timeout

The cache must maintain a notion of a maximum lifetime for each cached entry. Associating a timeout with cached information allows an application to cache information that becomes invalid with time.

Robustness

We envision the cache as operating within an operating system kernel. The caching system must, therefore, be robust over arbitrarily long periods of time and must protect itself from concurrent access to its data structures.

3 GCache

This section describes our implementation of a generalized caching system, GCache. We designed and implemented GCache according to the design goals discussed in section 2. In a later section on design alternatives and tradeoffs, we cover many design issues in detail; briefly, GCache has the following properties:

Interface

GCache considers a cached entry to be a pair of buffers, a *key* and a *result*. The caller passes a buffer to the caching routines by specifying both a pointer to the buffer and a length. GCache treats the buffers as opaque blocks of data, and only uses the contents for byte-equality comparisons and hash function computations.

Storage Method

Both the *key* and the *result* are stored and passed *by value* using copy-in, copy-out semantics. Passing arguments by value makes GCache more appropriate for small items of information than for large blocks of data.

Timeouts

The lifetime of each cached entry within a single cache is passed by the caller as an argument to the cache creation routine. Each item inserted into the cache will be assigned this same lifetime. The lifetime argument "0" is reserved to mean that cached entries should have an unbounded lifetime. Entry lifetimes are not explicitly checked until lookup time.

Limiting Space Requirements

An application that creates a new cache must specify the maximum number of cached

entries that the cache can contain at any given time, allowing GCache to bound its memory utilization. GCache removes previously cached entries, guaranteeing that a cache will always contain the last entry inserted.

Aging

GCache ages entries using an LRU, *least recently used*, strategy. Insertion of a new entry into a full cache forces the deletion of the entry that was looked up the least recently.

Data Structures

GCache implements each cache as a separate hash table of buckets. Buckets are implemented as doubly linked lists of cache entry headers for easy insertion and deletion.

In the next few subsections, we present a high-level description of GCache: its user interface and data structures, its memory management scheme, and an example procedure skeleton using GCache access routines.

3.1 User Interface

This section describes the GCache interface. Throughout this section and the rest of the paper, we will refer to return value *OK*, meaning that a routine completed successfully, and *ERROR*, meaning that an error occurred. Additionally, the identifier *cid* will be used to refer to a cache id.

```
int cacreate(name,nentries,lifetime)
     char *name;        /* textual name for the cache          */
     int   nentries;    /* maximum concurrent entries          */
     int   lifetime;    /* timeout for each entry, in seconds  */
```

The *cacreate*() routine creates a new cache. The caller specifies a textual name for the cache, the maximum number of entries that the cache should contain, and the maximum lifetime of each entry in seconds. If the *lifetime* argument is 0, cached items are never timed out. *Cacreate*() returns either a cache id or *ERROR* if the cache could not be created.

```
int cadestroy(cid)
     int cid;           /* cache id                            */
```

The *cadestroy*() routine deletes the specified cache and all associated resources. *Cadestroy*() returns *OK* if the cache is valid and *ERROR* otherwise.

```
int cainsert(cid,pkey,keylen,pres,reslen)
     int   cid;         /* cache id                            */
     char *pkey;        /* pointer to the key buffer           */
     int   keylen;      /* length of the key buffer (bytes)    */
     char *pres;        /* pointer to the result buffer        */
     int   reslen;      /* length of the result buffer (bytes) */
```

Cainsert() inserts a new mapping, *key* ⇒ *res*, into the cache. It returns *OK* if the cache id is valid and *ERROR* otherwise.

```
int calookup(cid,pkey,keylen,pres,preslen)
     int   cid;          /* cache id                          */
     char *pkey;         /* pointer to the key buffer         */
     int   keylen;       /* length of the key buffer (bytes)  */
     char *pres;         /* pointer to the result buffer      */
     int  *preslen;      /* in: size of the result buffer     */
                         /* out: # bytes returned in 'pres'   */
```

Calookup() searches for a cached entry matching the key passed as an argument. It returns *OK* if cid is valid and a matching item is found, and *ERROR* otherwise. On entry, the pointer *preslen* specifies the maximum amount of data to return, which is the size of the buffer pointed to by *pres*. *Calookup*() sets **preslen* to be the size of the result copied into *pres*. If a cached item is too large to be copied into the result buffer, *Calookup*() returns *ERROR*.

```
int caremove(cid,pkey,keylen)
     int   cid;          /* cache id                          */
     char *pkey;         /* pointer to the key buffer         */
     int   keylen;       /* length of the key buffer (bytes)  */
```

Caremove() removes the cached entry whose key is given, if one exists, and returns *OK*. *Caremove*() only returns *ERROR* if the cache id is invalid.

```
int capurge(cid)
     int cid;            /* cache id                          */
```

Capurge() removes all cached entries from the cache whose handle is *cid* and returns *OK* for a valid cache id and *ERROR* otherwise.

3.2 Data Structures

This section describes the data structures that GCache uses. *Cacreate*() returns a cache id, *cid*, which is used internally as an index into an array of cache descriptors, each maintaining a single cache. Each cached entry consists of a cached entry header containing, among other fields, pointers to the key and the result buffers. GCache stores cached entry headers in buckets attached to a hash table. We describe each of these data structures in detail in the paragraphs that follow.

For greater type checking security, GCache defines the following data types:

```
typedef u_short tcelen;      /* length of a cached entry          */
typedef u_short tceix;       /* index of a cached entry           */
typedef u_int   thval;       /* type of the hashed value of a key */
typedef u_int   ttstamp;     /* type of a timestamp               */
```

A user module refers to a cache using a cache id, *cid*, returned by the *cacreate*() routine. The GCache routines use the *cid* as an index into the *cacheblk* array. Each cache block contains the name of a cache and all required bookkeeping fields. It also contains a pointer to the hash table and a pointer to a free list of cache entry headers, neither of which is allocated until cache creation time.

```
enum cb_status { CB_INUSE=1, CB_FREE=2};
typedef enum cb_status cb_status;
struct cacheblk {
    cb_status       cb_status;      /* INUSE or FREE                */
    char            cb_name[NMLEN]; /* name of the cache            */
    int             cb_mutex;       /* mutual exclusion semaphore   */
    u_short         cb_maxent;      /* maximum # of entries         */
    u_short         cb_nument;      /* number of entries            */
    u_short         cb_hashsize;    /* size of hash table           */
    u_int           cb_maxlife;     /* max life of an entry (secs)  */
    struct cacheentry *cb_cache;    /* free nodes for the cache     */
    struct hashentry *cb_hash;      /* the hash table               */
    tceix           cb_freelist;    /* list of free cacheentries    */
    u_int           cb_lookups;     /* # lookups                    */
    u_int           cb_hits;        /* # hits                       */
    u_int           cb_tos;         /* # timed out entries          */
    u_int           cb_fulls;       /* # removed, full table        */
};
```

The *cacheentry* structure maintains a single cached entry. A cache entry contains pointers and lengths for both the key and the result, and a pair of timestamps for insertion time and last access time. GCache keeps all *cacheentry* structures for an active cache either on the free list or, when they are in use, on a doubly linked list attached to a hash table slot. GCache computes a hash value as a function of the sum of the bytes in the *key* buffer. GCache then computes the hash table slot as the hash value modulo the size of the hash table. Note that, for efficiency, the field *ce_hash* holds the original value of the hash function for each cached entry. When comparing a new key against a cached entry, GCache uses the lengths of the two keys and their hash values as a fast check for inequality, only performing a full comparison of the keys when the lengths and hash values agree.

```
enum ce_status {CE_INUSE=11, CE_FREE=12};
typedef enum ce_status ce_status;
struct cacheentry {
    ce_status       ce_status;      /* INUSE or FREE            */
    char            *ce_keyptr;     /* pointer to the key       */
    tcelen          ce_keylen;      /* length of the key        */
    char            *ce_resptr;     /* pointer to the result    */
    tcelen          ce_reslen;      /* length of the result     */
    thval           ce_hash;        /* value that was hashed in */
```

```
    ttstamp         ce_tsinsert;    /* timestamp - time inserted */
    ttstamp         ce_tsaccess;    /* timestamp - last access   */
    tceix           ce_prev;        /* next entry on list        */
    tceix           ce_next;        /* prev entry on list        */
};
```

GCache implements the hash table as an array of structures representing buckets, each containing the head of a (possibly empty) doubly linked cache entry chain. A hashentry is implemented as a structure to simplify the addition of fields for debugging and performance analysis. For testing the hashing function, for example, a counter could be added to monitor the number of items hashed into each bucket. Because the caller supplies the maximum number of cached entries as an argument to *cacreate*(), an appropriate size for the hash table can be computed when the cache is created. GCache constructs a hash table whose size is prime and at least as large as the maximum number of cached entries. Such a hash table is sufficient to keep the expected length of a list, on average, to no more than 1, assuming that the hashing function produces a uniform distribution of hash values across the keys used.

```
struct hashentry {
    tceix   he_ix;
};
```

GCache allocates cache entries as an array and uses indices into this array as pointers throughout the data structures. The array implementation facilitates searching for the oldest cached entry because no list traversal is necessary. To simplify the algorithms and improve performance, we reserve cache entry 0; it becomes the null pointer and we use it as the implicit head and tail of every list.

To help clarify the data structures used, consider figure 1 representing a user application using a cache whose *cid* is 2 (for simplicity, we omit the cache block array). The figure shows a cache that can contain a maximum of 6 cached entries. The cache includes a hash table of size 11 (0-10) and currently contains 3 cached mappings: $k_1 \Rightarrow r_1$, $k_2 \Rightarrow r_2$, and $k_3 \Rightarrow r_3$. The freelist contains 3 unused cache entry headers (1, 3, and 4).

3.3 Memory Management

GCache carefully manages memory space for the hash entries, cache entries, and the cached keys and results. GCache allocates this memory when a cache is created and deallocates it when a cache is destroyed. The Xinu Operating System provides a simple memory management mechanism similar to the Unix library routines *malloc*() and *free*(), providing memory from the kernel's heap space.

3.4 Example Routine

Figure 2 shows the skeleton of the Xinu *ip2name*() system call. *Ip2name*() maps IP addresses to host names using the TCP/IP Domain Naming Service [Moc87a, Moc87b]. The use of

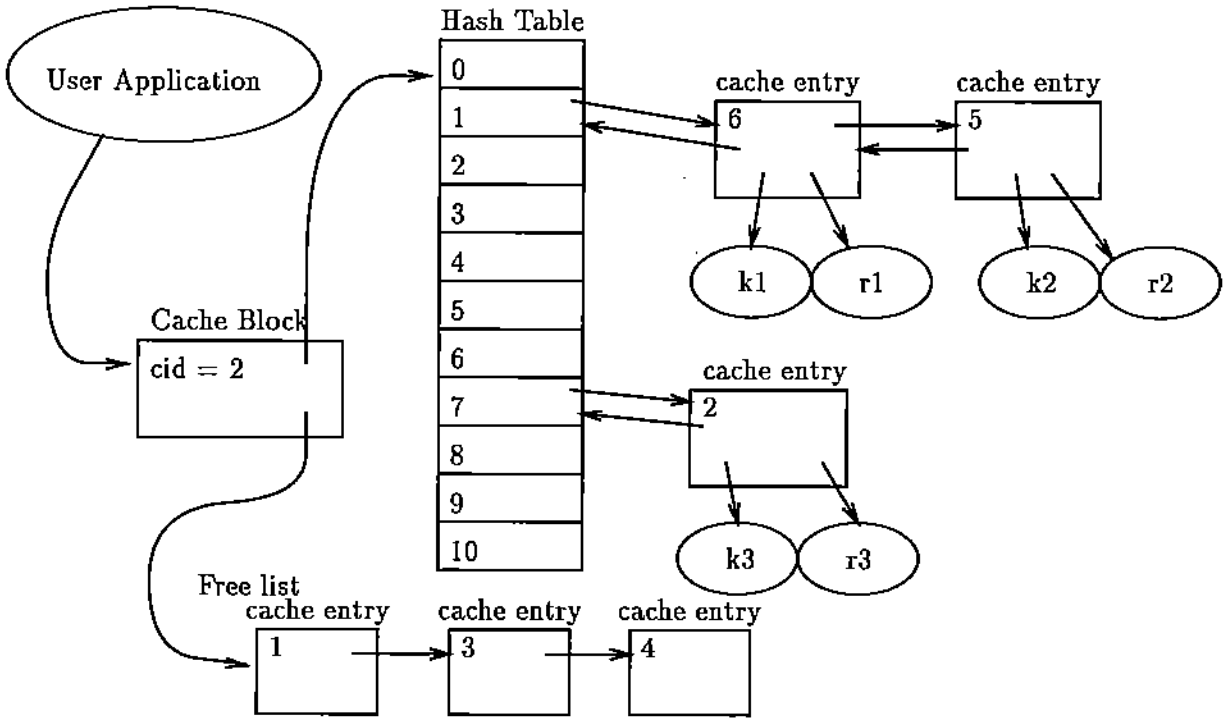

Figure 1: GCache Data Structures

dedicated GCache access routines allows the main procedure to ignore details of GCache access such as the cache id. The lookup routine uses the memory marking feature of Xinu to avoid creating the cache until it is first used. This is an important transparency issue; the alternative is to call a cache-specific initialization routine at system startup time.

4 Alternate Designs and Tradeoffs

The goals presented in section 2 lead to several design alternatives for various parts of a caching system. Each of these designs has tradeoffs associated with it. In this section, we discuss various design issues arising from each of our stated goals and identify the tradeoffs involved.

Interface

The type of cache that we envision implements a simple function mapping a key to a response, $K \Rightarrow R$. An application adds the elements of the mapping to the cache as it gathers the information. The lifetime of an individual mapping may be arbitrarily short. Each next request providing query K should either yield response R or the null response.

A more general solution allows the mapping to be arbitrary, as in $(k_1, k_2, ..., k_i) \Rightarrow (r_1, r_2, ..., r_j)$. Unfortunately, this generalization complicates the implementation while only providing a small amount of programming convenience.

```
int ip2name(ip_addr, name)
{
        /* check in the cache first */
        if (ipcachelookup(ip_addr,name) == OK) return(OK);

        /*******************************************************/
        /* Send Domain Name Server request to the network,  */
        /* wait for a reply.     (code elided)             */
        /*******************************************************/

        /* insert the result into the cache */
        ipcacheinsert(ip_addr,name);
        return(OK);
}

static int ip_cid;
static int ipcachelookup(ip_addr, name)
{
        static MARKER ipmark;
        int len = MAX_DNS_NAME;

        /* if this is the first call to this routine, we */
        /* must create the cache.                        */
        if (unmarked(ipmark)) {
                ip_cid = cacreate("ip2name", /* cache name        */
                                  50,        /* maximum entries   */
                                  60*60);    /* timeout (1 hour)  */
                mark(ipmark);
        }

        return(calookup(ip_cid,                  /* cache ID      */
                        ip_addr,sizeof(ip_addr), /* pkey, keylen  */
                        name,&len));             /* pres, preslen */
}

static void ipcacheinsert(ip_addr, name)
{
        cainsert(ip_cid,                         /* cache ID      */
                 ip_addr,sizeof(ip_addr),        /* pkey, keylen  */
                 name,strlen(name)+1);           /* pres, reslen  */
}
```

Figure 2: Example GCache Interface

Storage Method

The cache can store and retrieve the elements, K and R, of a mapping either by reference or by value. The choice of convention to use has a profound effect on the design of the system.

As an example, consider an instance of a cache in which K and R represent large structures. If the cache stores and retrieves K and R *by reference*, then the cache only needs to maintain pointers to them. This scheme makes the cache more efficient for caching large blocks of data. Unfortunately, it also introduces an unwanted interaction between the cache and the module it serves. In the absence of background garbage collection, a module cannot reclaim the space used by K and R without informing the cache. Forcing the user application to inform the cache leads to an interaction that conflicts with our requirement for transparency.

Alternately, the cache can pass and store K and R *by value*. With this design, an application can reclaim the space occupied by K and R without effecting the cache. This argument passing method, however, adds extra overhead in copying time and storage space, possibly making it inappropriate for caching large objects such as files and memory pages.

Because each of these two solutions is suitable for some class of cached information, a third possible design allows the application to specify which storage method is more appropriate for each individual cache.

GCache uses the second alternative, copying cached entries in and out, because transparency is more important than copying overhead for the types of caches for which we designed GCache.

Timeouts

Many caching systems, particularly those used for information obtained over a network, include the notion of a timeout that explicitly bounds the lifetime of a unit of expensive information. Information may have an implicit or explicit lifetime associated with it.

Network routes, for example, often include an explicit lifetime of a few minutes; they should not be used after that time. Network routes are exchanged periodically at a rate several times the maximum lifetime of the information. A host detects loss of a network gateway by timing out routes through that gateway.

Host name to address bindings also have a maximum lifetime associated with them, but this lifetime is generally implicit. A host may obtain a new address through network interface replacement or a move to a new network. Placing a maximum lifetime on host name and address bindings allows these bindings to change without requiring explicit interaction with the caches of other hosts.

A cached mapping whose lifetime has expired, therefore, should never be returned as the result of a query. The module requesting the cache should be able to specify the timeout, either for all cached entries at cache creation time, or for each entry at insertion time.

One design that provides this functionality uses a background thread that traverses the caches periodically and removes old information. A background thread solution, however, introduces extra overhead into the surrounding system and is overly complicated.

In a simpler and cleaner design chosen for GCache, each cached entry contains a timestamp encoding the insertion time. If a lookup matches an entry with an expired timestamp, that entry is removed rather than being returned.

Limiting Space Requirements

An application should be able to bound the total space requirements of an individual cache, particularly if the cache is part of an operating system. Operating system kernels generally execute in a fixed-size address space; modules that operate within a kernel environment must have bounded memory requirements.

In one possible design, a module creating a cache may specify the maximum number of cached entries it can contain. The cache insertion routine, when it discovers that the cache is full, may remove older items. GCache uses this option.

Another alternative allows the size of a cache to vary with its usage. Interface routines using this scheme can allow a heavily-used cache to expand and store more information; lightly used caches are contracted. With this design, one needs to derive appropriate policies for determining rates and thresholds for cache growth and contraction. The designer must also build a mechanism to trigger size changes; although the system can trigger cache growth at insertion time, cache contraction can be called for when the cache is never again accessed.

Aging

The above space management routines can determine that the cache is full, and must then act appropriately. One of our goals states that a cache insertion must always be successful. Therefore, when the cache is full, the caching routines must remove a previous entry. This approach forces us to design a policy for determining which old entries to remove. Several well-known policies are:

LFU

Under *least frequently used*, the lookup routines increment a counter associated with a cached entry each time that entry is accessed. The aging routines remove those entries with the lowest counter values. This simple scheme can behave badly when the items to be cached vary over time from one set of information to another.

LRU

Under *least recently used*, each cached entry contains a timestamp recording when it was last looked up. The aging routines remove those entries with the oldest LRU timestamps.

LRI

Under *least recently inserted*, each cached entry contains an insertion timestamp. The aging routines remove those entries with the oldest insertion time. Because

the timeout notion already requires an insertion timestamp, LRI aging requires no added information.

The nature of different types of expensive information may justify the use of either LRU or LRI aging. Because of this, a more general design allows a module creating a cache to pick the aging strategy most appropriate for the data it wishes to cache.

As a final issue, we must design a mechanism to search for an entry to delete from a full cache. Each of the policies above uses a timestamp within a cached entry to determine the entry to delete. For small caches, an exhaustive search for the oldest timestamp might be sufficient. For large caches, it might be more appropriate to thread a doubly linked list through the cache entries and maintain them in sorted order of their aging timestamps. This auxiliary data structure reduces the aging routine to a constant time search, but maintenance of the list increases the overhead for either insertion or lookup, depending on the aging policy.

Data Structures

The choice of an appropriate data structure depends on the expected use of the cache. We could implement a cache using many different data structures: binary trees, b-trees, heaps, hash tables, linked lists, or static arrays. Each of these data structures selects a different tradeoff between algorithm complexity, storage space, and lookup and insertion efficiency[HS76] [Knu73b] [Knu73a]. A good rule of thumb states:

tailor the complexity of the data structure to the size of the problem.

A cache that is only expected to hold a few entries could reasonably be designed using simple arrays and linear searches. For extremely large caches, however, the extra complexity of binary trees might be more appropriate. GCache uses a compromise consisting of singly hashed hash tables and buckets. Each bucket is represented as a linked list.

5 Experimental Results

Figure 3 shows statistics gathered from GCache shortly after the Xinu Operating System started. The machine involved is running an X11 server, window manager, and several local clients. The table shows statistics for 8 different caches of expensive information in the system. These caches fall into 3 categories:

IP Addresses

The first 2 caches, *ip2name* and *name2ip*, contain mappings between domain names and IP addresses. IP address mappings are small and have a long lifetime, which makes them excellent choices for caching.

NFS File Handles

The caches whose prefix is "nfs_fh" represent NFS file handle caches for a remotely mounted file system[Sun89]. When NFS parses path names, it performs an *nfslookup*() call on each component of the path, starting at the root. Unless the system caches

cid	cache name	maxent	nument	htsz	life	exp	full	finds	hits	hit%
0	ip2name	50	17	53	3600	2	0	80	44	55%
1	name2ip	50	12	53	3600	1	0	171	145	84%
2	nfs_fh_bonsai/usr	50	17	53	300	9	0	252	218	86%
3	rpcport	40	7	53	3600	2	0	147	130	88%
4	rpc_rto	50	5	53	3600	1	0	274	262	95%
5	nfs_fh_nyneve/xinu	50	39	53	300	6	0	317	258	81%
6	nfs_fh_bonsai/	50	4	53	300	0	0	16	10	62%
7	nfs_fh_ector/u12	50	2	53	300	0	0	3	0	0%

field	description	field	description
cid	cache id	exp	Number of expired entries looked up
name	cache name, from *cacreate*()	full	# entries removed, cache was full
maxent	Maximum # entries	finds	# calls to *calookup*() (and *cainsert*())
nument	Current # entries	hits	# successful lookups
htsz	hash table size	hit%	percentage of successful lookups
life	entry lifetime (seconds)		

Figure 3: Sample GCache Statistics

the results of these lookups, NFS generates a considerable amount of network traffic. Note that caching NFS file handles has security and consistency implications that are beyond the scope of this paper.

RPC Information

The *rpcport* cache stores mappings from RPC program numbers to foreign ports, originally obtained through RPC requests to remote portmappers[Sun88]. The *rpc_rto* cache contains round trip time estimations for RPC calls to remote hosts. These cached values are used to set timeouts for later datagram-based RPC calls.

The table in figure 3, generated 10 minutes after the Xinu kernel began running, shows 1260 lookup operations and 1067 valid mappings returned for a hit ratio of 85%. These figures represent a savings of over 2000 network packets, each with its corresponding cost for the client and one or more servers.

To further test the benefits of network information caching, we conducted experiments measuring the time required to invoke a *name2ip*() system call. The tests were conducted by an application program running outside the kernel. The machine used for the tests is a DECstation 3100 workstation running the Xinu Operating System. The nameserver is running on a Sun 3-160 on the same local ethernet. The results are as shown in figure 4. The first test, *Null System Call*, is for reference and shows the time required for a simple system call on this architecture. The remaining 3 tests were conducted by repeatedly requesting the IP address of a random host taken from a small set of local machine names. The cache used for name to IP address bindings holds a maximum of 50 entries. In the first such test,

experiment name	time	name set size	cache hit%
Null System Call	0.027 ms		
uncached name2ip	7.669 ms	10	0
cached name2ip (1)	0.087 ms	10	100
cached name2ip (2)	3.688 ms	100	50

Figure 4: Name2ip() System Call Test Results

file name	cached open() time	uncached open() time
/tmp/file	7.9 ms	25.9 ms
/tmp/a/file	8.4 ms	31.7 ms
/tmp/a/a/file	8.9 ms	36.4 ms
/tmp/a/a/a/file	9.4 ms	45.0 ms

Figure 5: Nfsopen System Call Test Results

uncached name2ip, the cache was disabled and the results show the time required to call the Domain Name Server over the network. The next test, *cached name2ip (1)*, shows the result of asking for the IP address of one of 10 local hosts. This test had a 100% cache hit rate and shows the overhead of using the cache. The final test, *cached name2ip (2)*, shows the effect of overflowing the cache by cycling through 100 hosts at random. Because the cache only holds 50 entries, cached mappings for the 100 hosts are repeatedly inserted into the cache and then deleted by the aging routines, resulting in a cache hit rate of 50%.

A final experiment measures caching's effect on simple NFS file operations. The test consists of a procedure that opens an NFS mounted file, */tmp/[a/]*/file*, for reading and then immediately closes it; the file system is already mounted. In the uncached case, this access results in at least 4 network calls:

1. Nameserver call to find the IP address of the NFS host
2. RPC portmapper request to bind the port used to access the NFS server
3. *nfslookup*() for the directory "*tmp*"
4. *nfslookup*() for the file "*file*"

Intervening directories, named *a*, between *tmp* and *file* result in one *nfslookup*() call each. In the cached case, only the *nfslookup*() call for *file* is required; Xinu always verifies the cached attributes of a file when opening it, but trusts cached intermediate file handles when available. We show the results of opening 4 different files in the cached and uncached cases in figure 5.

As expected, these examples show the benefits of caching. We see in the first example that operating systems make numerous network calls during operation. The next two

tests show the time that can be saved on different types of requests for expensive network information. Together, these tests support the idea that operating systems can benefit from caching. The use of a generalized caching system, therefore, is beneficial and its well-defined interface and semantics ease the task of adding caching to modules within the operating system.

6 Conclusions

Our implementation of GCache has shown that it is possible to design a general-purpose caching system. We have shown that the addition of such a cache can increase the efficiency of an operating system, particularly for expensive information obtained from a network. The example in figure 2 of a skeleton *ip2name*() system call demonstrated that caching routines can be added to existing systems with little programming effort. In production use, we have found GCache to be robust and of great utility, particularly in Xinu's networking subsystems. In addition to its usefulness in an operating system kernel, our caching approach is also suitable for use in large user applications in which caching can improve performance.

References

[Com84] Douglas Comer. *Operating System Design, the Xinu Approach.* Prentice Hall, 1984.

[Com87] Douglas Comer. *Operating System Design, Internetworking with Xinu.* Prentice Hall, 1987.

[Com88] Douglas Comer. *Internetworking with TCP/IP.* Prentice Hall, 1988.

[HS76] Ellis Horowitz and Sartaj Sahni. *Fundamentals of Data Structures.* Computer Science Press, Inc, Potomac, Maryland, 1976.

[Knu73a] D. Knuth. *The Art of Computer Programming: Fundamental Algorithms.* Addison-Wesley, Reading, Massachusetts, 1973.

[Knu73b] D. Knuth. *The Art of Computer Programming: Sorting and Searching.* Addison-Wesley, Reading, Massachusetts, 1973.

[Moc87a] P. V. Mockapetris. Domain Names - Concepts and Facilities, November 1987. RFC 1034.

[Moc87b] P. V. Mockapetris. Domain Names - Implementation and Specification, November 1987. RFC 1035.

[Sun88] Sun. RPC: Remote Procedure Call Protocol Specification Version 2, June 1988. RFC 1057.

[Sun89] Sun. NFS: Network File System Protocol Specification, March 1989. RFC 1094.