Cash: Distributed Cooperative Buffer Caching

Christopher DeCoro* Harper Langston† Jeremy Weinberger‡

Courant Institute of Mathematical Sciences
New York University

Abstract

Modern servers pay a heavy price in block access time on disk-bound workloads when the working set is greater than the size of the local buffer cache. We provide a mechanism for cooperating servers to coordinate and share their local buffer caches. The coordinated buffer cache can handle working sets on the order of the aggregate cache memory, greatly improving performance on disk-bound workloads. This facility is provided with minimal communication overhead, no penalty for local cache hits, and without any explicit kernel support.

Keywords: cooperative caching, coordinated caching, hint-based caching, buffer cache

1 Introduction

As local area networks continue to get faster and cheaper, it has become very attractive to move information from a given system to other machines a short network hop away. The argument for cluster computing in [Anderson et al. 1995] is well accepted as a good way to get more computing power for your money. As system and local bus speeds get faster, however, the cost of hitting a disk for needed blocks becomes exorbitant. When multiple servers in a cluster all need to access a common set of files, and the application working set is larger than the memory of any one server, many cycles are wasted churning blocks through the local buffer cache. Operators with excessive cash can solve this problem by spending it on RAM. Smart operators will take a look at Cash, an application-level shared buffer cache for cluster applications that access a common file repository.

We provide a library API for any application to open and read files through the shared cache. This library is a thin layer between the application and any locally visible file system. Thus, we are agnostic as to the actual file location; that is, whether the common file store is replicated on each server or visible via a shared NFS mount. We constrain our efforts to support read-only access to the file store.

When designing a coordinated caching system, there is an important trade-off between local and global performance. For certain policies, servers can pay a heavy price in local performance in order to be as altruistic as possible. The seminal analysis of these trade-offs can be found in [Dahlin et al. 1994]. We attempt to provide a framework that at least will not hurt performance, at best can improve local and global performance, and can be easily modified to support alternative policies. We specifically intend to improve cluster performance on read-only workloads where the size of the working set is greater than the main memory of any given server.

*e-mail: cdecoro@cs.nyu.edu
†e-mail: harper@cs.nyu.edu
‡e-mail: jeremy@cs.nyu.edu

2 Related Work

Generally speaking, the algorithms and problems faced in designing a distributed, cooperative buffer cache are very similar to those in designing a distributed shared memory system such as in Ivy [Li and Hudak 1986]. Our decentralized manager design is informed by the analysis of global page management in this work. Our work differs in that cooperative caching is purely an optimization. We do not have to provide perfect access to the cache, as long as cache misses occur quickly.

Our work is largely motivated by Dahlin's study on Cooperative Caching [Dahlin et al. 1994], an intended component of the Berkeley xFS [Dahlin 1996; Anderson et al. 1995] file system. We rely upon the simulations done by this group in designing an algorithm similar to N-Chance Forwarding. However, the xFS studies assume that all cache servers read blocks from a centralized file server, and that the file server provides special support for cache metadata. Our system can run over consistency-averse systems such as NFS. Our algorithm is also very similar to to the hint-based algorithm described in [Cuenca-Acuna and Nguyen 2001]. We found only simulations in this work, where we have provided a working implementation.

Other cooperative caching systems have been designed and simulated with reference to the xFS simulations. The Global Memory Service [Feeley et al. 1995; Feeley 1996] is one such technique. Sarkar and Hartman provide a comparative analysis of N-chance forwarding, GMS, and hint-based techniques in [Sarkar and Hartman 2000]. We opt for an algorithm that combines features of N-chance forwarding and hinting to allow for a simple system that can run without any support by a specialized central file server.

Our caching algorithm can also be viewed in the context of application-level caching. A target application for our code is a cluster-based web server; thus, web caching protocols such as Internet Cache Protocol (ICP) [Wessels and Claffy 1997] and Cache Array Routing Protocol (CARP) [Valloppillil and Ross 1998] are relevant. ICP is a broadcast-based query system which requires a network broadcast on each file request. This type of protocol is easy to reject for a block-based cache, due to high network overhead. CARP is a stable hash-based block location protocol. We provide more flexibility in block placement than what is possible with hash-based techniques.

We implicitly assume a round-robin or similar "fair" load-balancing request distribution policy across the member servers. This is a common approach for naïve load-balancing. However, alternative methods exist, such as Locality-Aware Request Distribution (LARD) [Pai et al. 1998]. LARD is an attempt to externally manage the cache of each server by distributing requests to servers that are likely to already have the item in cache. LARD promises to provide better performance than lower-level cooperative caching. As an application of the end to end argument [Saltzer et al. 1984], one might argue that the block level may be too low for cache services for many applications. However, LARD requires specialized kernel support in order to function at all, where our system can be provided entirely at application level. This provides a system designer with the opportunity to replace our LRU algorithm with one better suited to a given application. We can also argue via end to end that allowing the application user to provide his own caching policy as well as request distribution policy is preferred. A topic of future research would be study cooperative caching algorithms in the context of non-uniform request distribution policies.

3 Architecture

The Cash architecture on the local machine is divided into two parts: a library linked into applications and a manager process. As illustrated in Figure 1, each library instance communicates with the manager via a Unix domain socket. The manager and each library instance all mmap a single well-known shared cache file in order to avoid additional data copies between the manager and the library.

Cash is intended to be used in a cluster configuration, where all servers accessing the shared file store run an instance of Cash. As shown in Figure 2, every Cash server should communicate with every other Cash server to maximize the size of the coordinated cache.

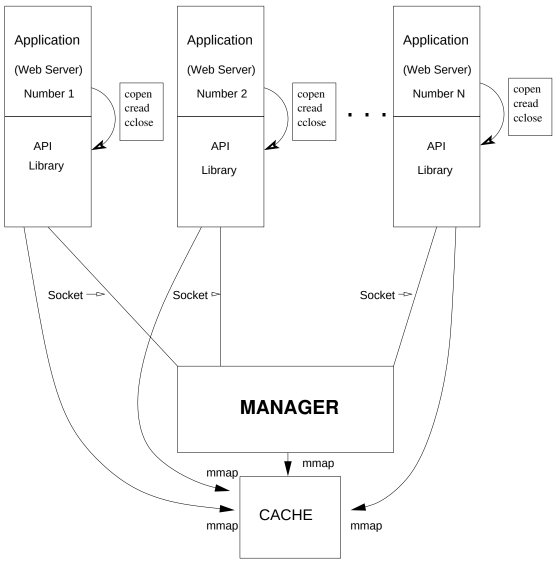

Figure 1: Cash architecture on a single local machine

4 System Interface

Clients view Cash through a standard file-system interface (similar to the fstreams interface of the C Standard Library). The only difference to the user (other than the fact that the file system is currently read-only, an artifact of our current implementation that will be changed in a future release), is faster performance through the transparent use of distributed buffer caching. No extra requirements are made of the client applications; a principal design goal of Cash, successfully realized, is that operations can be directly mapped onto Unix kernel I/O system calls. This allows us the flexibility, if desired, to integrate Cash seamlessly into kernel of a Unix-like operating system.

A Cash cluster is made of up a set of machines, each running an instance of Cash. Each instance consists of three major components:

- Client Applications, the end-user programs that make use of Cash services.

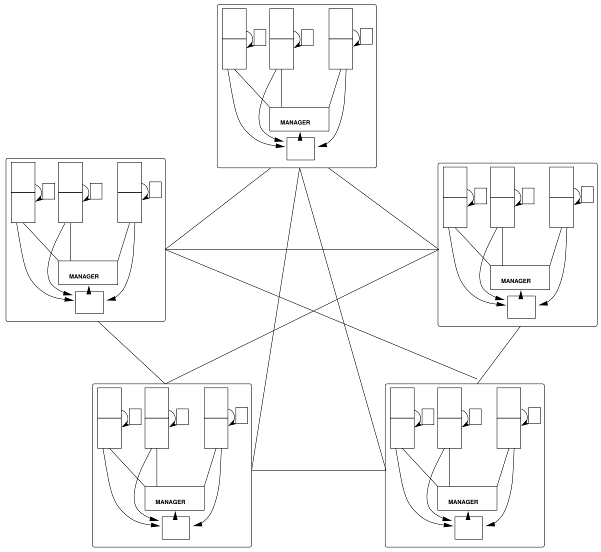

Figure 2: A cluster of Cash servers

- Application Interface Library, which translates Unix-like file I/O calls into the internal Cash interface.
- Cache Manager, the system-wide manager that controls resources on the local machine, and maintains the server's participation in the wider cluster.

Some machines may not run any client applications, existing solely to provide memory to other machines in the cluster. These are known as *pure servers*, as opposed to *standard servers*, which do run client applications.

Client applications, such as a webserver or database, will use the interface exported by the Application Interface Library to communicate with a Cache Manager process that runs on the current machine.

4.1 Programming Interface

As previously discussed, the Cash interface is all but transparent to the application programmer, and maps directly onto C Standard Library functionality. In our implementation, to avoid name clashes with those standard functions, we replace the `fopen`, `fread` and `fclose` functions with the following:

- `CFILE* copen(char* filename,char* mode)`
- `int cread(void* buffer,int size,int n,CFILE* file)`
- `void cclose(CFILE* file)`

`CFILE` is an opaque data structure that is used by the library to represent an opened file. The client acquires a `CFILE` object by calling the `copen` function and providing it with the name of a file and a mode string as in the C Standard Library function `fopen`. The function `cread` will read n elements, each of size `size`, into `buffer` from the file represented by the `file` parameter. The return value is the number of items read. The function `cclose` will deallocate a file handle and reclaim resources.

For communication between the client processes and the manager process, we use the Unix Domain Sockets method of IPC for communicating control messages and synchronizing both client and server. Additionally, we use the Memory-Mapped Files feature to share I/O data between manager and client.

We now look in detail at the implementation of each function:

```
struct CFILE {
    char* filename;        //name of the file
    char* base;            //pointer to mmap'ed file data
    int range_offset=0;    //offset of mmap'ed data in file
    int range_length=0;    //length of mmap'ed data
    int current_offset=0;  //current offset in file
};

int client_socket = -1;
CFILE * copen(char* filename, char* mode)
{
    //Once per system, initialize the manager process
    if( !file_exists( CACHE_SOCKET_NAME ) )
    {
        fork_manager_process();
    }
    //Once per process, initialize the communication socket
    if( client_socket == -1 )
    {
        client_socket = create_and_bind_socket();
    }
    CFILE* file = new CFILE;
    file->filename = filename;
    file->fd = open( CACHE_FILE_NAME );
    return file;
}
```

Figure 3: Implementation of the copen function call. The CACHE_SOCKET_NAME and CACHE_FILE_NAME correspond to the absolute file names of the well-known Unix domain socket file and the global system cache file.

```
int cread(void* buffer,int size, int n, CFILE* file)
{
    int read_bytes = size * n; //bytes still to be read
    int copy_bytes = 0;        //bytes copied so far
    //Try to avoid an IPC by using buffered data
    if( file->current_offset is between file->range_offset and
          file->range_offset + file->range_length )
    {
        copy_bytes = read_bytes - ((file->current_offset + read_bytes) -
        (file->range_offset + file->range_length));
        read_bytes -= copy_bytes;
        memcpy(buffer, file->base + (file->current_offset -
              file->range_offset), copy_bytes);

        file->current_offset += copy_bytes;
        buffer += copy_bytes;
    }
    //Continue to send messages and reading bytes
    while(read_bytes > 0)
    {
        send_read_message(file->filename, file->current_offset, read_bytes);
        int offset, length;
        get_reply_message(&offset, &length);
        file->base = mmap( file->fd, offset, length );
        file->range_offset = offset;
        file->range_length = length;
        memcpy(buffer, file->base, length);
        ptr += length;
        file->current_offset += length;
        read_bytes -= length;
        copy_bytes += length;
    }
    return copy_bytes;
}
```

Figure 4: Implementation of the cread function call

4.1.1 copen

Before reading the contents of a file, the application must call the copen function to generate a valid file handle, and initiate communication with the manager process. Because a manager process must be running on the machine, and we do not want to require applications or administrators to initialize the manager explicitly, copen will fork off a manager process if one does not yet exist on the current machine. The copen function guarantees exclusivity by testing to see if the well-known manager socket file exists, and only forks off the manager if it does not.

The copen function must also initialize the sending and receiving sockets for communication with the server. First, a Unix domain socket in datagram mode is created to send request messages to the manager's well-known address (in our implementation, this was "/var/tmp/cashsocket"). Second, a new ephemeral socket is created to receive responses from the server.

Finally, copen opens the global cache file for reading and returns a new CFILE pointer to the caller. This file will later be used to map data blocks into process address space. The CFILE pointer tracks the current position in the file (in current_offset), the location of the data file as currently mapped into memory (base), and the starting offset of that file in the mapped portion (range_offset), and length of the currently mapped portion (range_length).

A description of the copen functionality is given in a pseudocode that aims to be as C-like as possible; see Figure 3.

4.1.2 cread

Once the application has created a valid file handle with copen, it may then use cread to retrieve data from cache. The cread function will send simple messages to the server indicating the file it intends to read, as well as the starting offset and length of the read operation. The function blocks until receiving a response from the server.

We have plans for implementing an asynchronous interface for Cash. The main issue involves creating a file descriptor for use in the select function. This is necessary, because in an asynchronous, event-driven system, the application will use the select call to wait on all open file descriptors, and wakeup when activity occurs on one of those descriptors. If Cash were implemented in the kernel, we could directly implement this functionality for our descriptors. However, in its current state, there is no way to have the select function block on one of our CFILE pointers. The solution is to introduce an additional function, cgetdesc, that retrieves the file descriptor for the communication socket. This will be used in select, and will cause the application to resume when communication is detected from the manager.

The response message from the server will indicate the offset and length of the data in the system cache file. The cread function will use the mmap system call to map this data into the current process address space, and to store a pointer to this newly-mapped region in the CFILE structure. The function will then copy as much data as possible into the output buffer, and will update the other control variables appropriately. The process of sending messages, receiving responses, and accessing mapped data will continue until all of the requested bytes have been read.

It is possible that the number of bytes requested by the user is significantly smaller than the number of bytes returned by the manager, which operates on blocks of 4KB. For example, an application may chose to read the file only one byte at a time. In a naïve implementation, this would cause an IPC on each read. In our system, however, we buffer the previously read block, and if possible, we use that buffered data to service the client. If the user request can be handled solely with the data in the buffer, no IPC is needed at all. Otherwise, if the requested data is larger than the buffer, we copy the entire buffer into the output, and proceed to send requests to the manager as usual. Pseudocode for cread is given in Figure 4.

4.1.3 cclose

Finally, the cclose function will delete the CFILE pointer that was created back in copen, and reclaim its resources. Like copen, the cclose function does not need to access the manager or perform an IPC.

4.2 Administrator Interface

It is the administrators responsibility to ensure that certain settings are present before Cash-enabled programs are run. The CASH_BINDIR environment variable needs to be set to the location of the Cash manager program, in order to inform the library linked into each process of the location of the manager. The manager also reads the CASH_CACHESIZE environment variable to allow configuration of the local cache size.

The manager needs to provide a configuration file with the IP addresses of each machine that is in the cluster. This may include the current machine, as the Cache Manager will recognize its own IP address and not attempt to send requests to itself. Cash requires this information, as the lack of a single, centralized server requires some ability to determine the other machines in the cluster.

5 Implementation

5.1 Block Manager

Like other distributed shared memory systems, we need a manager for locating and relaying blocks between participating nodes. Our block manager has two critical design features: first, it is completely decentralized, and second, it does not require network communication before every block request. We provide a manager process that runs as a daemon on every participating node. This manager listens on a well-known UDP port for the following types of messages:

- `LOCATION`: "Block A is at node X."
- `BLOCK REQUEST`: "Send a copy of block A to node Y."
- `BLOCK RESPONSE`: "Here is block A" or "Nobody has a copy of block A."
- `FORWARD`: "Here is the last cached copy of block A. Please store it."

Each manager keeps a local table of hints that indicate which manager is caching or knows where to find a given block. This hinting algorithm is similar to the one described in [Sarkar and Hartman 2000]. When the manager is first started, this hint table is empty. Over time, the manager will begin to receive requests for blocks from the local machine. If a block is not found in the local cache, and there is no hint for this block, the manager assumes there are no cached copies available. At this time, the manager will begin caching the block locally. After delivering the block to the local application, the manager broadcasts a `LOCATION` message to all other known managers. The other managers will record this new hint in their tables. This is the only broadcast in our protocol.

Every block manager believes that its hint table is authoritative. If a `BLOCK REQUEST` from another manager is received, and no hint is present, the manager will respond with a false `BLOCK RESPONSE`, indicating that no manager is caching the requested block. If a hint is available and points to the current manager, the manager will issue a true `BLOCK RESPONSE` containing the bytes of the requested block. If a hint is available and points to a different manager, the manager will forward the request to the indicated manager. Thus, requests are forwarded one hop at a time until they reach a manager capable of responding true or false.

Block managers are permitted to forward blocks to one another at any time via the `FORWARD` message. Upon receiving a forwarded block, the manager is expected to keep the block in its local cache for some period of time. Global hints are not expected to remain perfect when forwarding occurs. Only the sender and receiver of a `FORWARD` message will update their hint tables. However, given that all hosts were aware of an initial manager for a given block via a broadcasted `LOCATION` message, block requests will always follow an acyclic graph across the managers to a node who can positively say "yes" or "no". Sarkar and Hartman found that simulations of a similar hint-based protocol resulted in requests being forwarded four times or less 99.998% of the time ([Sarkar and Hartman 1996]).

A sample timeline of our protocol can be found in figure 5. At time t_0, node X loads block A, and broadcasts a `LOCATION` to all nodes. At t_1, node X forwards block A to node Z. Only node X and node Z update their hint tables. At time t_2, node Y gets a local request for block A. Y's hint table says to send a `BLOCK REQUEST` to node X for block A. Node X knows that node Z probably can locate block A, and forwards the request to Z. Node Z still has the block, and responds with a true `BLOCK RESPONSE`, containing the data of block A.

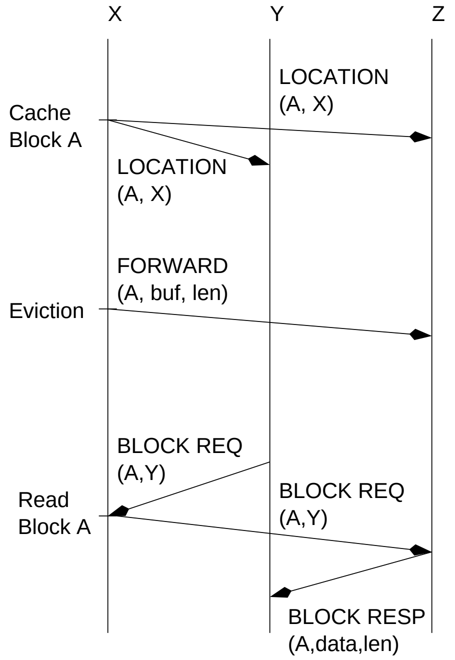

Figure 5: Sample timeline of interactions dictated by the Block Manager protocol

5.2 Local Cache

We provide a two-tiered LRU buffer cache, controlled by the distributed manager. Blocks in the cache are designated as either master or non-master, depending on whether or not there are other copies of the block. Master copies are the only known global copy of a block, and therefore have strictly higher priority than non-master copies. Non-master copies are believed to be a duplicate of blocks in another node's cache. Since duplicate copies can always be reloaded with a fast network read, non-master blocks are low in value and always evicted first. Simulations in [Leff et al. 1996] show that it is almost never worthwhile to discard the only cached copy of a block in order to keep a duplicate in the cache.

Blocks become master copies when they are loaded from disk. Our algorithm loads a block from disk if there is no hint, or the hint exists, but the response is "no", or if the response times out. Thus, the local copy is believed to be the only one in the global cache. Blocks become non-master copies when they are received from another node via a `BLOCK RESPONSE`. It is obvious that at the moment the request completed, the other block was certainly available as a master at the responding node, and so the copied block is a duplicate.

Evictions are a potential source of inaccuracies in the global hint table. When an eviction is necessary, the queue of non-master blocks is searched first for a block to discard. If no non-master blocks are available, the LRU master block is evicted and sent via a `FORWARD` to a randomly selected node. The target node is expected to cache the forwarded block. In our current implementation, the forwarded block is inserted at the front of the master LRU list, as if it had been recently referenced. If the target node needs to do an eviction to make space, it is not permitted to forward another block. This limitation is needed to ensure that forwards do not cascade through the network, causing more hints to become inaccurate, and using extra network bandwidth. Thus, the target will discard a master block instead of forwarding it if no non-master blocks are available.

6 Evaluation

In order to evaluate the effect of Cash, we built a small web server as our test application. The web server is simple and asynchronous, using `libasync` [Mazières et al. 2000]. Further, it is based on the Single Process Event Driven (SPED) architecture as described in [Pai et al. 1999] and visualized in figure 6. It is designed to use the basic calls, `fopen`, `fread`, and `fclose`, independently from Cash in order to prove the ease with which we can incorporate our new `copen`, `cread`, and `cclose`.

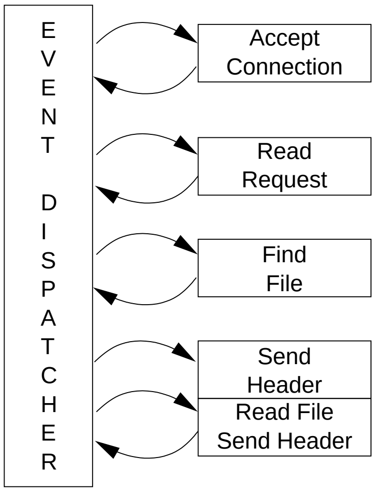

Figure 6: Single Process Event Driven (SPED) architecture for our simple web server test application

In order to evaluate the power of Cash with our test application, we look for ways to generate typical workloads over a short period of time and measure them. Based on the work of [Arlitt and Williamson 1996], we wish to model invariants such as file types, file size distribution, concentration of file requests, etc. By doing this, we can test how Cash improves the performance of our web server over time. By adding more than one Cash server, we expect dramatic improvement. Additionally, we wish to study the effect of a warm versus cold cache to see if there is increased performance as in [Kim et al. 2000].

The work of [Barford and Crovella 1998] uses the invariant models to build representative web server workloads. They use a variety of probability distribution models to build the typical workload. For example, to build the total number of requests, they observe the "concentration of requests" invariant by using Zipf's Law. For example, among 2000 files, the most popular may be requested 200,000 times when building a request model, in which case we have $\approx$ 140,000 requests that are split among the other 1999 files, some only being requested once. File sizes are built based on lognormal and Pareto distribution models, explicitly detailed in the Appendix. Further, it is predicted that 30% of files will be HTML with embedded images, 38% will be embedded images and 32% will be plain HTML files for a typical image and HTML web site. A lognormal distribution is used for modeling the frequency at which specific files will be requested. Bursts of activity are also modeled as off times, using Weibull and Pareto distributions.

We employ the use of [Barford and Crovella 1998]'s Scalable URL Reference Generator (SURGE) in order to build a typical workload for testing Cash. For our tests, we set our web site to 10,000 documents, ranging in size from 512 bytes to 1.5 MB, with a mean size of $\approx$ 20 KB (for a total web site of size $\approx$ 200 MB). Further, we make the most popular document able to be requested 50,000 times, resulting in Zipf's Law building $\approx$ 480,000 total requests for a single session. The time to complete a session depends on the speed of completed requests; i.e.,2500 KB/s transfer rate implies we can download each of the 20 KB average size requests of all 480,000 requests in approximately one hour (64 minutes). If we run a test for only five minutes, the total size requested would be $\approx$ 9600 KB. Of course we cannot complete the 480,000 requests in five minutes; however, due to the files distribution, our results show that even though we may only complete 40,000 requests in five minutes, we will see at least 5,000 distinct files if we have a transfer speed of 2500 KB/s.

Using this initial setup, we ran SURGE for five-minute intervals, and a changing number of cash-implemented servers and client processes where we fix the number of client and server machines at 4 each (each server has a replica of the web site generated by SURGE and each client requests a different distribution of files based on lognormal model using different seed values to guarantee each client does not requests the documents in the same order). We begin with each server not using Cash, then each server using Cash with no communication amongst each other. We follow by having each server in communication with only one other server, and we end by having each Cash server aware of each other, so we have all-to-all (4-4) communication. For each client machine, we increase the number of client processes from 1 to 5 where each client process runs fifty threads, each requesting documents as dictated by our setup.

Initial results with SURGE were not promising. Figure 7 shows that as we increase the number of client processes for each setup, the mean transfer delay time decreases slightly when we use one-to-one communication, but the delay increases when we have all four servers requesting documents from each other. Additionally, figure 8 shows that the KB per second transferred increases slightly for one-to-one communication, but it decreases for all-to-all communication.

Obviously, these results are not what we had hoped for. However, several aspects are encouraging. For example, we have realized that while SURGE gives a good distribution of files to be requested, the actual software is not good at handling the number of requests required to truly display the potential Cash has when overtaxing the web server. Hence, the bottleneck seems to be in SURGE being dramatically slower than our transfer rate since we were unable to reach a state of maximum transfer rate on our machines. Since all-to-all communication performs almost as well as all of the other tests, once we hit maximum transfer rate, Cash should begin to perform substantially better. As a result, we are turning to the use of *httperf* [Mosberger and Jin 1998], a more powerful tool for testing web server performance. By incorporating *httperf* with the valid file distribution models of SURGE based on web invariants, we believe we can more accurately test Cash.

Additionally, we are beginning work on incorporating Cash with a more typical web server such as Apache, since unnecessary latency may be introduced by our implementation of a simple SPED web server, and Apache may more appropriately display the awesome potential of Cash.

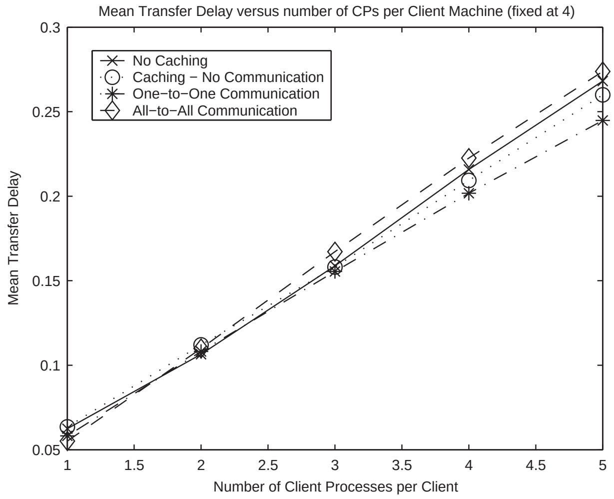

Figure 7: Mean transfer delay on all four test server configurations for varying numbers of client processes per client machine

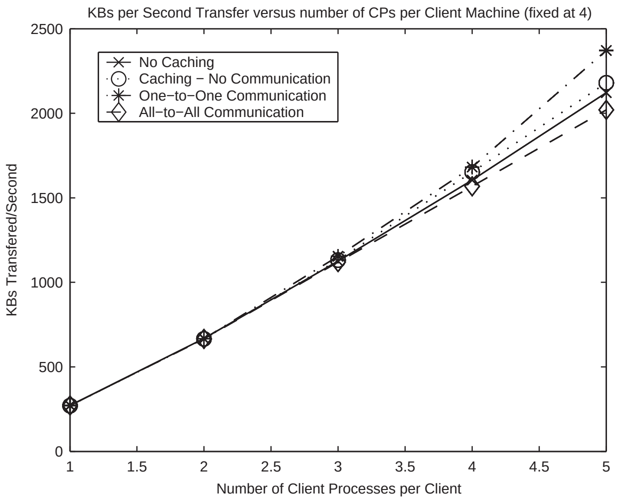

Figure 8: KB per second transfer rate on all four test server configurations for varying numbers of client processes per client machine

7 Future Work

We constrained ourselves to the case of a read-only workload on the file store, which greatly relaxes requirements for cache consistency. This makes a hint-based system much more feasible. We believe our system is extensible to a read/write workload. Using an optimistic write policy similar to [Page et al. 1997] would be an important factor in allowing writes to Cash without requiring our low-overhead location hints to become high-overhead facts.

Our design does not explicitly help in the case where a small set of items produce almost all of the load on the server. An algorithm that explicitly considers the heat on a block as a factor in its eviction is an important extension.

We did not include any support for global age hints. This would be useful to target idle servers as a destination for evicted blocks. This would also improve our behavior on receipt of a forwarded block. Instead of treating a forwarded block as recently referenced, the target node can attempt to place the block approximately in local LRU order. This would also allow behavior more like that of N-chance forwarding, where globally old blocks are discarded instead of allowing them to circulate.

Our software can easily be extended to support a dynamic list of cooperating servers. We already support timeouts to decrease the cost of misses on inaccurate hints. This timeout could be extended to maintain a weighted average round trip time, and to ignore servers that time out until we begin to receive hints from them again.

Analysis into how cache management policies interact with request distribution is an important and unaddressed topic in the literature. One could conceive of situations where both LARD and a cooperative cache might be desirable, yet this interaction is not well understood.

8 Appendix

Here we present several of the probability distribution models as used to build our test workloads, using SURGE [Barford and Crovella 1998].

- Lognormal

$$p(x) = \frac{1}{x\sigma\sqrt{2\pi}} e^{(-lnx-\mu)^2/2\sigma^2} \quad (1)$$

- Pareto

$$p(x) = \alpha k^{\alpha} x^{-\alpha+1} \quad (2)$$

- Weibull

$$p(x) = \frac{\beta x^{\beta-1}}{\alpha^{\beta}} e^{-(\frac{x}{\alpha})^{\beta}} \quad (3)$$

References

ANDERSON, T., CULLER, D., AND PATTERSON, D., 1995. A case for NOW (networks of workstations).

ARLITT, M. F., AND WILLIAMSON, C. L. 1996. Web server workload characterization: The search for invariants. In *Measurement and Modeling of Computer Systems*, 126–137.

BARFORD, P., AND CROVELLA, M. 1998. Generating representative web workloads for network and server performance evaluation. In *ACM SIGMETRICS International Conference on Measurment and Modeling of Computer Systems*, 151–160.

CUENCA-ACUNA, F. M., AND NGUYEN, T. D. 2001. Cooperative caching middleware for cluster-based servers. Tech. Rep. DCS-TR-436, Department of Computer Science, Rutgers University, Mar.

DAHLIN, M., WANG, R., ANDERSON, T. E., AND PATTERSON, D. A. 1994. Cooperative caching: Using remote client memory to improve file system performance. In *Operating Systems Design and Implementation*, 267–280.

DAHLIN, M. D. 1996. Serverless network file systems. Tech. Rep. CSD-96-900.

FEELEY, M. J., MORGAN, W. E., PIGHIN, F. H., KARLIN, A. R., LEVY, H. M., AND THEKKATH, C. A. 1995. Implementing global memory management in a workstation cluster. In *Symposium on Operating Systems Principles*, 201–212.

FEELEY, M., 1996. Global memory management for workstation networks, phd thesis, university of washington, 1996.

KIM, J., CHIO, J., KIM, J., NOH, S., MIN, S., CHO, Y., AND KIM, C., 2000. A low-overhead high-performance unified buffer management scheme that exploits sequential and looping references.

LEFF, A., WOLF, J. L., AND YU, P. S. 1996. Efficient lru-based buffering in a lan remote caching architecture. *IEEE Transactions on Parallel and Distributed Systems 7*, 2 (February), 191–206.

LI, K., AND HUDAK, P. 1986. Memory coherence in shared virtual memory systems. In *Proceedings of the Fifth Annual ACM Symposium on Principles of Distributed Computing*, 229–239.

MAZIÈRES, D., DABEK, F., AND PETERSON, E., 2000. Using TCP through sockets.

MOSBERGER, D., AND JIN, T. 1998. httperf: A tool for measuring web server performance. In *First Workshop on Internet Server Performance*, ACM, 59—67.

PAGE, T., GUY, J., HEIDEMANN, J., RATNER, D., REIHER, P., GOEL, A., KUENNING, G., AND POPEK, G., 1997. Perspectives on optimistically replicated peer-to-peer filing.

PAI, V. S., ARON, M., BANGA, G., SVENDSEN, M., DRUSCHEL, P., ZWAENEPOEL, W., AND NAHUM, E. M. 1998. Locality-aware request distribution in cluster-based network servers. In *Architectural Support for Programming Languages and Operating Systems*, 205–216.

PAI, V. S., DRUSCHEL, P., AND ZWAENEPOEL, W. 1999. Flash: An efficient and portable Web server. In *Proceedings of the USENIX 1999 Annual Technical Conference*.

SALTZER, J. H., REED, D. P., AND CLARK, D. D. 1984. End-to-end arguments in system design. *ACM Transactions on Computer Systems 2*, 4 (Nov.), 277–288.

SARKAR, P., AND HARTMAN, J. 1996. Efficient cooperative caching using hints. In *Proceeding of the 2nd ACM Symposium on Operating Systems Design and Implementation (OSDI)*.

SARKAR, P., AND HARTMAN, J. H. 2000. Hint-based cooperative caching. *ACM Transactions on Computer Systems 18*, 4, 387–419.

VALLOPPILLIL, V., AND ROSS, K. W., 1998. Cache array routing protocol v1.0. Internet draft. Microsoft Corporation.

WESSELS, D., AND CLAFFY, K., 1997. Internet cache protocol (ICP), version 2. RFC 2186.