Netcache: A Distributed Buffercache Extension

CS262a, UC Berkeley, Fall 2002

Marco Barreno, Aaron Hurst, Adam Megacz

barreno@cs.berkeley.edu, ahurst@uclink.berkeley.edu, megacz@stat.berkeley.edu

Abstract

Netcache is a new distributed buffercache system that is designed to exploit the resources on a local area network to improve the performance of disk reads and other block I/O read accesses. The typical round-trip latency on an Ethernet segment is roughly an order of magnitude shorter than the seek time of a hard disk, and the size of the pool of unused RAM is potentially large; Netcache exploits this available storage as an extra layer of cache for block I/O operations. This paper discusses the theory, protocol, security implications, implementation details, performance analysis, and plans and ideas for future functionality enhancements of Netcache. Netcache has been implemented and tested as part of the Linux kernel's buffercache, and the results are very promising.

1. Introduction

1.1. Motivation

The exponential growth of processing power and memory capacity over the last several decades has unfortunately not been mirrored by performance improvements in other critical components. The result has been computers that are increasingly limited by bottlenecks in other aspects of the system. Hard disk accesses are a prime example. Hard drive capacity continues to increase geometrically, but the average seek time seems to have hit a plateau. When a computer initiates a disk transfer, it can be tens of millions of CPU clock cycles until the operations is complete. Improvements in other aspects of the architecture have mitigated some of the problem. For example, software parallelism allows many operations to be executed in parallel with I/O accesses, but still the penalty is present. Furthermore, there are many situations where there isn't an alternative to remaining completely idle. A virtual memory page fault must be resolved before any other tasks can resumed.

1.2. Caching

The first line of defense against slow secondary storage has always been caching. Since it appears that hard disk access time is not likely to make any tremendous leaps forward anytime soon, we should limit the need to go to the disk as much as possible. To that extent, most modern operating systems include a mechanism for keeping frequently-used disk information resident in memory. In Linux, the buffercache mechanism can use up to all available memory to keep recently accessed files quickly retrievable.

File caches have been quite successful, and the continual advancement in RAM performance makes them increasingly beneficial. However, as this disparity between memory and disk speed widens, it makes cache misses disproportionately painful. Main memory latency is currently on the order of a few hundred nanoseconds, while a typical hard disk may have an average access time of around 10-15 milliseconds—five orders of magnitude slower. The size of this latency gulf is large enough that it is worthwhile to consider the addition of another layer of cache hierarchy. A viable solution should offer an access time that is within the above range and provide storage space that is at least as big as the amount of RAM in the computer.

1.3. LANs

Local area networks (LANs) present an attractive means of filling this need. The round-trip latency on an Ethernet segment is tenths of milliseconds to milliseconds, and most LANs have an enormous amount of unused bandwidth. Plenty of storage space is available in the large memories of workstations and PCs on the network. As some workstations are in use and in need of extra disk cache, there are often others that are idle and can provide resources to the busy machines.

1.4. Netcache

Netcache is designed to exploit the available memory on a LAN for performance enhancement on busy machines. It uses the remote RAM as an additional level of read cache for filesystem blocks to reduce the number of disk accesses. It boasts a number of security features and optimizations that minimize the risk and maximize the performance gain of sending disk blocks to remote machines.

The contributions of our work come from both the direct evaluation of the Netcache design as well as the adjunct technologies associated with it. We primarily focused on integrating the core concept with the Linux kernel to test performance, but security, integrity, and efficiency were important parts of our exploration as well.

1.5. Organization

In this paper, we describe the Netcache system in depth and provide a performance analysis of Netcache compared to unmodified Linux. Section 2 details the design of Netcache, and Section 3 delves into the inner workings of the project. We present an analysis of Netcache's performance in Section 4. Section 5 summarizes some related work, and Section 6 outlines possible directions for further research in this area.

2. Design

2.1. Philosophy

The aim of Netcache is to provide filesystem performance enhancement through use of RAM on idle remote machines as an extra layer of buffercache. When a busy machine runs low on memory, it will seek out idle machines to which it can propagate blocks for later retrieval. Those blocks are encrypted for confidentiality, and the sender computes an MD5 hash for integrity checking. On the idle machine, the blocks are stored in the existing buffercache but specially marked as foreign blocks so they can be evicted first when the machine becomes active again. Netcache performs all these operations automatically and requires no configuration beyond inclusion of Netcache in the operating system.

Netcache is not an entirely independent layer of the cache hierarchy. Although main memory keeps a copy of everything that is present in the L2 cache (and so on), Netcache only stores blocks as they are evicted from RAM. Technically, it serves as a *victim cache* for data that have been discarded from the next higher level in the hierarchy. A block will never be in a higher level of the cache at the same time that it is in the Netcache, so no additional state is necessary to track whether blocks have been written. Dirty blocks are never propagated.

2.2. Assumptions

We've made several assumptions in undertaking this project. First of all, we assume that network latency is significantly less than hard disk latency including seek time. There is certainly justification that this is the case, and the trend in technology indicates that it is likely to remain so for the near future. However, significant changes in technology can not be ruled out. Cheap and abundant non-volatile RAM, non-magnetic secondary media, and micro-electromechanical circuitry all have the ability to change the existing paradigm. Network performance problems could also invalidate this assumption. Netcache is appropriate for a network with sufficient bandwidth and very low latency; high-traffic networks may not be suitable.

Next, we assume that the LAN in question is one subnet. This is very likely true for home networks, building networks, and small corporate networks, and it is a necessary assumption for our current implementation.

Third, we assume that some machines on the network will be idle and available for resource sharing at the same time as others are busy and requesting cache storage. This may not be the case in banks of servers or corporations during the workday, but there are many scenarios in which this is a fine assumption to make. In particular, we expect this assumption to hold in situations such as home networks and academic networks. The latter might be dorms, offices, or labs, since students and professors have varied schedules due to classes, meeting times, and other activities. It should be noted that having some idle machines on the LAN is not necessary for correctness, but it is a prerequisite for performance improvement.

2.3. Protocol

When a Netcache client comes online, its first action is to solicit information about any available neighbors. A `query status` message is broadcast over the subnet. Any peers that receive this message will respond with an `update status` to alert the

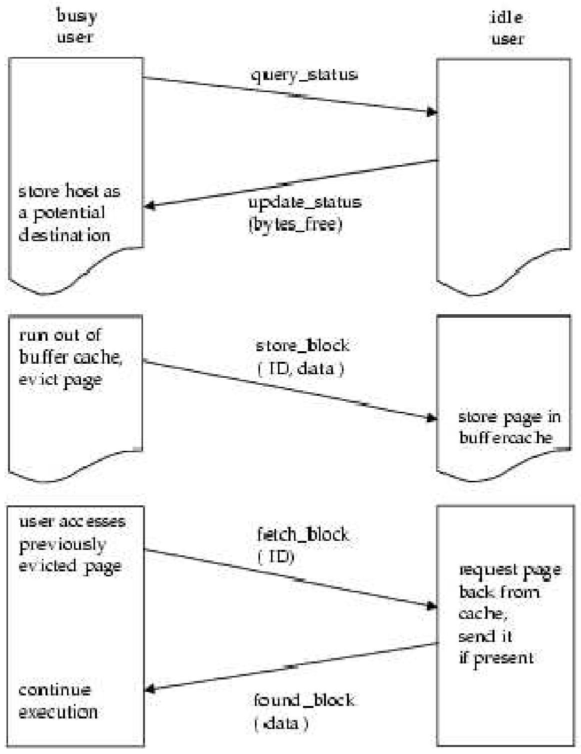
busy user
idle user
query_status
store host as a potential destination
update_status (bytes_free)
run out of buffer cache, evict page
store_block (ID, data)
store page in buffercache
user accesses previously evicted page
fetch_block (ID)
request page back from cache, send it if present
continue execution
found_block (data)

Figure 1: A graphical representation of the Netcache protocol

new member of their location and available memory. This information is then used to construct the peer information table, indexed by the last byte of the IP address. Each entry then contains the time that a peer was last contacted, its free physical memory, and its network address. This table is kept up to date by periodic `update status` broadcasts; new information is also propagated if the node's availability to accept data changes dramatically.

When a Linux machine runs low on memory, the first thing it tries to free is buffercache space. This means that the host will look for a destination for storing evicted blocks. The peer information table is scanned to find a host with an acceptable amount of available cache. If more than one qualifies, the final selection is made randomly. The randomization helps to distribute buffers among idle hosts more evenly. Next, the information is packaged into a network packet and sent asynchronously to the selected destination using a `store message`. By not blocking on the network operation, the Netcache contributes almost no additional overhead when storing packets.

Upon receiving a packet, the Netcache destination allocates a portion of its remaining free physical memory and stores the incoming data. This block will be stored as long as there is no competition for the space. The host's own file cache and applications have priority over any foreign data; if more memory is requested on the host, the foreign data will be the first to be evicted. Because propagated buffers cannot be dirty, this will mean only that the host that stored the dropped packet will probably get a cache miss but no worse. Because the host does not depend on propagated blocks for correctness, the only penalty will be the time it takes to determine that the information is no longer in the cache.

If the originating computer ever attempts to reload a packet that has been stored on the network, it will sent out a `fetch request` broadcast to attempt to locate its cached data. To avoid having to store and track the locations of all previously written blocks, we don't maintain this information; instead, we query all of the neighboring computers at once. If the block is indeed present on another machine, that computer will respond with a `fetch reply` containing the requested data. In the event that no response is received, the call times out and the block is reloaded from the disk.

2.4. Security

2.4.1. Trust

At the forefront of issues to consider when designing a system such as Netcache is security. Casually broadcasting arbitrary disk blocks over an untrusted network threatens any effort made to keep the host and its data secure. One of the key issues in designing Netcache, then, was deciding what degree of trust to place in neighboring machines on the LAN.

On the one hand, there is naturally an implicit trusting relationship between computers on one subnet. They tend to be controlled by the same organization, they are generally operated by friends or coworkers, they are all usually on the inside of an organizational firewall, and the physical and logical proximity makes it both necessary and relatively easy to put a high degree of trust in these neighbors.

On the other hand, there are some important reasons not to put full trust in the other computers. Limiting trust can limit the damage of a break-in if the intruder has difficulty using one compromised computer to exploit others. The data in the buffercache may also be very sensitive—any data stored on a block device could potentially be propagated to another machine. Personal data, classified or confidential data, and critical system files are of such a sensitive nature that not even a trustworthy coworker's machine should have access.

We examine the various security risks in terms of possible attacks on the Netcache system.

2.4.2. *Attacks*

An adversary has various potential means of attempting to subvert a Netcache-enabled computer that does not have cryptographic protection.

The most obvious is that any buffers sent in the clear to a machine controlled by the adversary could be read, possibly compromising sensitive data or exposing weaknesses on the propagating machine. Only slightly less obvious is that an adversarial computer could broadcast a fetch request for a block belonging to another computer if it could determine the device and block number, thereby accessing the data.

If an adversarial computer stores a block for a non-compromised computer, it could maliciously alter that block to produce garbage or even alternate valid content of the adversary's choice. It could also reply to a fetch request even if it had not stored the block, returning garbage or a malicious valid block.

An adversary has various Denial Of Service (DOS) attacks available as well. Any of the protocol messages—query status, update status, propagate block, fetch block—might be sent repeatedly at a fast rate in an attempt to monopolize or overwhelm the resources of the targeted machine, since each of them requires some action in the receiver of the message.

2.4.3. *Our solution*

In its current incarnation, Netcache employs a dual encryption and integrity check scheme to defend against all attacks mentioned above except DOS attacks.

Each block will be encrypted using AES[1] in CBC mode[2] before it is propagated to a foreign machine. The encryption key resides only in the memory of the propagating machine and a random IV is generated for each block. This ensures the confidentiality of each block.

The propagating machine also computes and stores an MD5 cryptographic hash of the block upon propagation. When the block is returned, the propagating machine recomputes the hash and compares it against the stored value; if they match, the block's integrity is assured with very high likelihood. If they do not match, then the block must be re-read from disk because either the propagated block has been altered remotely or the block has been changed locally since propagation.

2.4.4. *Regarding MD5 collisions*

One concern to investigate is the possibility that two blocks might have the same MD5 hash. This would be a severe problem because the storing machine might (either inadvertently or with malicious intent) return a different block that has the same hash value, thereby corrupting the data of the propagating machine.

To assess this risk, we must look at the probability that two blocks will *collide*, or hash to the same value. We will start with the estimate that there are 580.78 million people online [Nua02], which we will use for an approximate number of computers on the internet. A standard new hard drive size today is 80GB, and the majority of computers undoubtedly have much smaller drives. We will also conservatively assume that these computers use a block size of 1 KB (linux generally uses a 4 KB block size). This yields a total of 4.6×10^{16} blocks among all online computers in the world. This is undoubtedly an extremely high estimate, as a high percentage of the blocks will be identical from machine to machine. Finally, an MD5 hash is a 128-bit quantity, so it can take on approximately 3.4×10^{38} different values.

The formula for determining the probability of a collision is

$$p = 1 - \frac{m!}{(m-n)!m^n},$$

where m is the number of possible values (3.4×10^{38}) and n is the number of instances (4.6×10^{16}). In Appendix A to the online documentation for XML RFC 2938 [KM00], it is calculated that the probability of collision for $m = 3.4 \times 10^{38}$ is 1×10^{-9} for $n = 1 \times 10^{15}$

[1]AES is the Advanced Encryption Standard. For more information on the standard, see NIST FIPS-197 [Nat01]; for more information on the algorithm, see *The Rijndael Block Cipher* [DR]; for more information on performance of Rijndael and other AES candidates, see *Performance Comparison of the AES Submissions* [SKW+].

[2]Cipher Block Chaining mode is a mode for a block cipher in which blocks are chained together by taking an `xor` of each block with the ciphertext of the previous block and encrypting that result rather than simply encrypting each block independently. A random "initialization vector" (IV) is often used to `xor` with the first block. The primary reason to use CBC is to avoid the undesirable property that each block of plaintext produces the exact same block of ciphertext each time it is encountered.

and 1×10^{-3} for $n = 1 \times 10^{18}$. This means that even if every online computer in the world were to join in Netcache, and if every block on each computer were globally unique, there would still be less than a 0.1% chance of finding any MD5 collision at all!

These assumptions overestimate very heavily (especially since Netcache is designed to be used on a LAN, not the internet as a whole!), so we can conclude that the chance of MD5 collision is insignificant.

2.5. Unification

In modern computing environments, it is very common for a large number of workstations on a given subnet to have mostly-identical hard drive images; for example, in an office using Windows, almost every machine will have an identical installation of Windows, Word, Excel, Powerpoint, etc. Since files always begin on a disk-block boundary, the blocks comprising these applications will also be identical, although they will be arranged at different locations on the disk.

The full design of Netcache offers an optional *unification* mode which improves aggregate performance in this scenario by storing identical buffers from multiple hosts only once.

With optimal heuristics, the system can be expected to converge to a state where the entirety of this shared data is in memory, somewhere on the network, meaning that no host will ever have to read this data from its local disk. Other computing environments achieve this goal by putting all shared binaries on a central server with a huge amount of memory; however, this approach introduces a single point of failure and increases costs by neglecting to utilize memory on idle machines.

3. Implementation

3.1. UML

We developed and debugged our implementation using User Mode Linux (UML) [Dik01]. UML is a port of the Linux kernel which is designed to run as a process on a host Linux workstation. The guest kernel forwards all I/O requests to the host kernel, and uses system calls such as `mprotect()` to implement memory management. Guest processes are scheduled by the host scheduler as threads within the guest process' kernel.

3.2. Introduction to the Linux buffercache

The Linux buffercache is a memory cache for filesystem blocks (more generally, for blocks from any block device). It can grow to use all available memory on the system, and it shrinks whenever new memory is requested by the system. Linux uses a `buffer head` data structure to keep track of the relevant information for one buffer in memory. This structure holds a pointer to the buffer data as well as accounting information and queue pointers for the buffer.

When a program tries to read a block, Linux hashes the device and block number together to get a lookup value and uses that value to find the buffer head for the block. If no buffer head exists for that block, it creates one. Once it has obtained the buffer head, it checks to see whether the block is in memory. If not, it must schedule a disk read for the block.

When the system needs to free up some memory, it first turns to the buffer cache and asks it to free up some buffers. This is done by selecting a page of buffers, ensuring that the buffers on the page are clean (and scheduling disk writes if necessary), and then dropping those buffers and buffer heads. The page to free is selected by an approximation of LRU.

3.3. The addition of Netcache

Netcache is essentially a victim cache for the main memory buffercache. There are two primary operations for Netcache: `propagate block` and `fetch block`. These functions are easily integrated into the Linux buffercache.

Whenever a block must be evicted from the buffercache to free some memory, as part of the eviction the machine calls `propagate block`. This selects a machine on the LAN with free memory (if there is any) and sends the block to that machine before evicting it from memory. The selection happens based on values stored from updates sent out periodically from the other machines. Once a destination is selected, `propagate block` encrypts the block, serializes the encrypted block and its appropriate metadata, computes and stores an MD5 hash, and sends the serialized message in a UDP packet to the destination host. The destination host receives and unserializes the message. If this host has become busy since it last sent out an update and no longer has room to store the block, it simply drops the block. Otherwise, it stores the block in its local buffercache (see Section 3.4 for details).

If no appropriate destination is found, `propagate` block drops the block as would happen without Netcache.

When the buffercache tries to read a block and finds that it is not in memory, it calls fetch block before scheduling the disk read. This function sends a subnet broadcast asking if any host has the block in question. It waits on a reply and times out if no reply is received.

If a host on the subnet does have the block requested, it reserializes the metadata with the encrypted block and sends it back to the requesting machine. The requesting machine unserializes and decrypts the buffer, then computes the MD5 hash and checks it against the stored value. If it checks out, fetch block copies the block into the buffer memory and returns. If not, or if the timeout expires, the machine must go to disk for the block as it would without Netcache.

3.4. Storage of foreign buffers

When a host receives a foreign block to store for another machine, it is able to store it in the infrastructure already in place for the local buffercache. The buffer heads for the buffercache are stored in a hash table that hashes the device and block number to get an index into the table. We have assigned a special device number to foreign blocks so that they can take advantage of this preexisting hash table. When a block is received, the host uses its local buffercache system to create a buffer head and buffer in memory, associated with the Netcache device number rather than an actual device on the system. The block received is then copied into the local buffer and the metadata are stored in the buffer head. Memory is allocated by the system for the buffer and buffer head, as for the local buffercache. When we place the foreign blocks in the local buffercache, we give them the lowest priority so that they will be evicted first when the idle machine becomes busy once more.

When storing a block for another machine, it is not necessary to keep track of the device number on the propagating machine as part of the metadata. The reason for this is that when the propagating machine accesses a buffer in its local buffercache (and then seeks it from other machines), the request is made by device and block number, so the machine will already know the device sought when it fetches a block from the foreign host.

The block number used to store buffers propagated from other machines is a simple hash (shifted xor) of the device and block number on the propagating machine. It certainly is possible for these values to collide, but a collision here is not a significant problem. If there is a collision, the result will be that a block sent in response to a fetch request will be the wrong block. This situation is significantly different from the MD5 collision scenario described in Section 2.4.4 because in this case the MD5 acts as a check, but if the MD5 collides there is no further check. If there is a collision here and the wrong block is returned, the machine that requested the block will compute the MD5 and find that it does not match the one stored, so it will drop the erroneous block.

3.5. Implementation of Unification

We have given unification a good deal of thought and spelled out an implementation for it, though we have not yet put it into our code. We are still unsure whether we can build in adequate cryptographic protections, and it didn't seem worth spending the time on this rather than some of the other pieces we have implemented. Nevertheless, this remains the primary extension to our implementation we would like to write. Here we describe the plans for implementation of unification.

3.5.1. Unification with encryption

When unification is enabled, each evicted buffer is encrypted with the following algorithm in order to generate ciphertext:

$$\texttt{ciphertext} = AES(\texttt{buffer}, MD5(\texttt{buffer}))$$

The ciphertext is propagated across the network, and the originating host retains two hashes in a private data structure: $MD5(\texttt{buffer})$ and $MD5(\texttt{ciphertext})$.

When a host receives ciphertext, it computes $MD5(\texttt{ciphertext})$. If this value matches the corresponding hash of a different block's ciphertext, the two blocks are stored only once, since they must be identical (assuming that a highly unlikely MD5 hash collision has not occurred, as discussed in Section 2.4.4).

When a host wants to retrieve a buffer, it broadcasts a request for the buffer using $MD5(\texttt{ciphertext})$ to make the request rather than the usual $(\text{dev}, \text{block})$ combination. When the ciphertext is returned, $MD5(\texttt{buffer})$ is used to decrypt the ciphertext, yielding buffer.

3.5.2. Properties of the Unification Encryption Algorithm

This encryption technique ensures two key properties:

1. Any host which had `buffer` at some point in the past is able to decrypt `ciphertext`, even if it was not the host which propagated the buffer.
2. Only a host which had `buffer` at some point in the past is able to decrypt `ciphertext`.

These properties are ensured without the use of slow public key cryptography or cumbersome key distribution techniques. In essence, *the block itself* acts as its own encryption key.

3.5.3. Security Implications of Unification

Harmonizing unification and encryption in this fashion has two drawbacks. First, although an attacker cannot decrypt buffers being sent across the network, the attacker can *verify* a guess at an encrypted buffer.

For example, if the contents of a password file were transmitted across the network, an attacker would not be able to retrieve the passwords. However, if a block from a medical database were sent across the network, and an attacker (say, an employer) knew all the fields in the block except for one (say, a binary yes/no of whether or not a person's family had a history of heart disease), the attacker could guess values for that field and verify those guesses by encrypting both possible blocks and seeing which matches the target encrypted block.

Second, since no IV is used in the encryption, identical blocks will yield identical ciphertexts (note that this may be necessary for unification). This allows an adversary to keep track of how often some particular blocks are transmitted even if the adversary cannot decrypt those blocks.

Because of these limitations, a production implementation of Netcache should include a mechanism for marking blocks as *privacy-critical*. One simple way to do this would be to have the filesystem check for files which are unreadable even by their owner, and propagate this information to Netcache. Netcache could then generate `ciphertext` differently for these blocks, using the following algorithm:

$$\texttt{ciphertext} = AES(\texttt{buffer}, \texttt{secret})$$

Where `secret` is some random number known only to the originating host. When this algorithm is employed, the encrypted block will *not* unify with identical blocks from other machines, eliminating the performance advantage of unification. However, attackers cannot use `ciphertext` to check guesses at the contents of a block.

3.6. Adaptability of Netcache

One of the key features of Netcache is that it requires no configuration. A linux kernel compiled with Netcache will seek out peers, propagate and fetch blocks, and store blocks for other computers automatically. No effort is required on the part of the user.

4. Analysis

4.1. Disclaimer

For various reasons, porting the code from UML to the testing cluster was far more problematic than we expected. Part of this lies in the differences between the UML architecture and the standard Linux architecture, and part of it lies in the intricacies of the network booting setup we used for testing (see Section 4.2). The one significant and unfortunate consequence of this difficulty is that we had to remove all MD5-related code from Netcache in order to make it work on the testing cluster. We plan to investigate this further at a later date, but these numbers reflect a version of Netcache without MD5s.

This should not have a significant effect on the numbers. MD5 calculation and comparison does not take a significant amount of time compared to network latency, and because we perform only read tests and not write tests, propagated data cannot be stale. The only concern might be that the wrong block is returned and not caught because we don't check MD5s, but since we have only one busy host propagating files from only one block device, there can be no block id collisions.

4.2. The cluster

For our tests, we had use of the UC Berkeley Statistical Computing Facility's cluster of eight Dell Pentium II 350MHz machines. Six of the computers have 128MB RAM and two have 96MB. They are physically in the same room and are connected on a 100Mb switched ethernet.

In order to facilitate a rapid develop-compile-test-repeat cycle, we configured the machines to load `netboot` (netboot.sourceforge.net) from a floppy

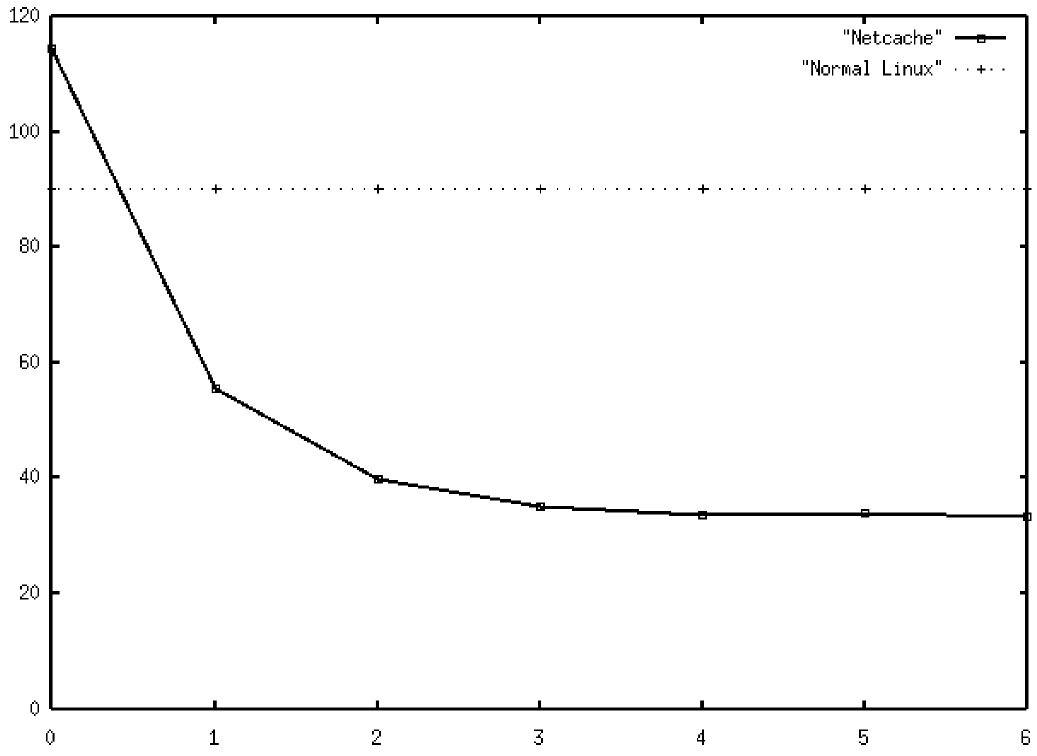

Figure 2: Seconds to complete benchmark vs. number of idle peers

disk. Netboot initializes the network card, obtains an IP address from a bootp server, and downloads the linux kernel from a tftp server. The Linux kernel then boots, mounts a root filesystem via nfs, and mounts the local hard disk on `/mnt`. A statically linked binary in `/sbin/init` contains the test code.

This setup allowed us to use a single command to recompile the kernel with modifications and place it on the tftp server. We could then toggle the power button on the cluster's power strip to reboot all the machines and instantly see our changes in action.

4.3. Methodology

Our goal for this test was twofold: to compare the overhead and speedup of Netcache to the standard Linux kernel, and to judge the effect that number of idle peers has on the performance of a busy host.

The test first creates a file of size 200 MB. This is 1.5 to 2 times as large as the memories on the cluster machines. Because we wanted to measure the optimal performance of Netcache, we primed the cache by sequentially reading the file into memory several times. It is necessary to do it more than once because the busy host only propagates a block to one peer at a time, and we wanted to ensure that there were multiple copies of each block in the network cache in case one was dropped.

The timed benchmark then consists of about 10,000 block reads from random points in the file. This benchmark is run for Linux without Netcache, as well as Linux with Netcache using 0, 1, 2, 3, 4, 5, and 6 idle peers for one busy machine. Peers were added in the same order for each test, and the machines with 96 MB memory were the 3rd and 4th peers added.

4.4. Results

The results of our testing are graphed in Figure 2. When one Netcache-enabled computer is running standalone without any peers, this preliminary implementation adds approximately 27% overhead. With even just one peer on which to store blocks, however, the Netcache-enabled computer significantly outperformed the standard Linux kernel, taking only 62% as long for its run. After three or four peers are available for storing blocks, Netcache levels out to approximately 35-40% the time of standard Linux. This should be considered a close to optimal case for this implementation of Netcache, since the cache was well primed and one busy host had several idle hosts to work with.

These are very promising results, and they demonstrate that Netcache can offer a significant speedup over standard Linux.

Given that network latencies can be on the order of ten times shorter than disk latencies or even less, however, one might ask why these benchmark times for Netcache are not even smaller. The answer, we believe, is that adding the infrastructure for Netcache creates an necessary overhead that must be taken into account. With optimization the 27% overhead demonstrated here could doubtless be lowered, but not eliminated entirely. Unavoidable sources of overhead include performing encryption, doing the bookkeeping for the peer tables, and waiting for timeouts when there is a cache miss.

The results presented here demonstrate that the Netcache distributed buffercache is a viable strategy for increasing filesystem performance and merits further research and development.

5. Related Work

5.1. Network RAM

There have been several papers published about using a network for memory paging, also aiming to take advantage of the lower network latency compared to disk latency. In 1994, Anderson and Neefe proposed a network virtual memory system that provided remote memory paging [AN94]. It too sought performance gains from using the RAM of other machines before having to go to a local disk.

The fundamental difference between network paging schemes such as this one and Netcache is dependence on remote systems. A computer using remote memory paging would fail if the remote computer failed. Netcache, on the other hand, is very tolerant of remote machine failure. In the worst case, there is still no failure or data corruption, but only some overhead impacting performance. In Network RAM's worst case, there is complete system failure.

5.2. VMware ESX Server

Our unification technique is very similar to the *content-based page sharing* technique employed by in the VMware ESX Server virtual machine monitor presented at OSDI 2002 [Wal02]. VMware unifies identical memory pages (rather than disk buffers) across multiple virtual machines on a single host, storing each page once.

Like Netcache, VMware must unify objects by content rather than address, since different hosts are likely to store the same data in different locations. Also like Netcache, VMware uses a hash function to perform page comparison in $O(1)$ time rather than $O(n^2)$.

Unlike Netcache, VMware implements copy-on-write sharing, since requests for pages are made by address rather than by content. By contrast, Netcache requests remote pages *by their hash value*, so no copy on write is needed; if a buffer is modified, its hash will change, and stale copies of the buffer will no longer match the new hash.

VMware also implements an *optimistic* hashing technique, in which a possibly-stale hash of each page is used for unification. If a match is found, the hash is recomputed and checked again to ensure that the match is valid. Netcache only computes buffer hashes when evicting a buffer from memory, and since a buffer cannot be modified once evicted, there is no need to implement a similar optimistic technique.

5.3. Networks Of Workstations

The idea of using the combined memory of workstations on a LAN as a file cache was put forward in the 1994 Berkeley paper "A Case for NOW (Networks Of Workstations)" [ACPt95]. The authors discuss the potential performance improvement of using LAN memory for file caches rather than going to disk and make a case for implementing such a system, but they do not perform the implementation.

5.4. Cooperative caching

The paper by Dahlin et al. on "cooperative caching" [DWAP94] explores the idea of using remote RAM as a buffercache, but in a different environment: there is one network file server from which many clients mount the same filesystem, and the buffercache caches only blocks from that network filesystem. This changes the problem in significant ways, perhaps most drastically in that the presence of a server may be assumed to manage the caches and direct requests. The authors mention the case where individual computers propagate and request buffers directly to and from each other (calling it "direct client cooperation") but dismiss it as lacking too many of the server's advantages.

Our work is in the same vein as this, but it is different in a few important ways. For example, Netcache works in general with any block device, while this cooperative caching is limited to a network file system. Also, cooperative caching requires one machine to function qualitatively differently from the rest as a filesystem server and cache manager. Most importantly, Netcache requires no configuration at all beyond compiling it into the kernel: peers find each other and begin to propagate and fetch when appropriate, all completely automatically.

It should also be noted that with unification our system will have most of the benefits of the cooperative caching server-based system. The same block need only be stored once, and one machine can request a block another machine has propagated. Potentially, a copy of every shared block will eventually be in some machine's memory and no request must go to disk while this state is maintained.

6. Future Work

Although the basic functionality of Netcache is complete at this time, there are several enhancements we would like to see in it, and there are some interesting research questions that might now be asked. We plan to continue research on this topic and develop Netcache further. The current state of Netcache is a system that works but has some of the hacks and compromises that commonly enter into any such project. We would like to polish Netcache to production level, adding some of the features listed here, and offer it as a public patch to the Linux kernel, perhaps someday to be merged into the main tree. We expect to tackle many of these issues in the near future.

6.1. Unification

Unification is not currently implemented in Netcache, but it is the addition we would most like to bring to fruition. More research may need to be done to find an appropriate balance of unification and our cryptographic confidentiality goals, but we expect that unification will yield strong numbers in situations such as corporate networks, where many machines have near identical sets of programs that are loaded up daily.

6.2. Performance

Netcache can undoubtedly be optimized beyond its current level. Additionally, more work needs to be done to assess the performance of Netcache in typical workloads. This is a difficult problem, especially because most of the applications that we expect Netcache to benefit most are interactive and not amenable to objective testing. Nevertheless, it is a direction that must be explored before Netcache is ready for wide release.

6.3. Expanding beyond the subnet

The current design of Netcache restricts it to one subnet. However, it may not be uncommon that, for example, one organization will have several subnets in close physical proximity and will want to utilize Netcache across the entire organization. Many of Netcache's features, such as broadcasting fetch requests, work especially well when restricted to one subnet. An interesting research question would be: how well can Netcache scale above the subnet? One key issue would be finding peers to which buffers could be sent—a simple subnet broadcast of "I'm here" no longer would work. A possible solution for this would be to have some machines (e.g. subnet gateways) act as propagate and fetch routers, communicating with other known routers to maintain lists of available hosts.

6.4. Fairness

There are issues of fairness that are not adequately addressed by the current implementation of Netcache. In many cases, such as a small household LAN, this is not a problem. In corporate environments or larger LANs with many users, however, it may become desirable to ensure that one busy machine does not use up all resources on idle machines, leaving none for other busy machines. Possible solutions to this problem include some sort of economic accounting or e-cash. Another solution for egregious cases may be blacklisting machines for a time that use more than their fair share of resources, thereby not allowing them to claim any more resources. The research questions here obviously extend past Netcache and into more general issues of fairness and common resource allocation.

6.5. Selecting a destination

A better algorithm for choosing where to propagate blocks might take into account such factors as network latency to hosts, either concentrating or more evenly distributing blocks among hosts, and the ratio of busy to idle machines. The current system of taking the first available host works, but we might be able to increase performance with more optimization here. This is also an opportunity to research the intersection of load balancing and network responsiveness.

6.6. How to choose timeout

The choice of an appropriate timeout is one of the key decisions that most influences Netcache's performance. Although it is not difficult to determine the optimal timeout for a particular LAN by experimentation (we determined a good value for our testing cluster through quick trial-and-error), it remains to be seen whether a more general solution can be found. Perhaps the optimal timeout will be proportional to average ping time, within a certain range, or perhaps it will always be an empirical factor for any given network. We would like to do more work towards resolving these issues.

6.7. Security against DOS attacks

Netcache remains vulnerable to DOS attacks, though it could perhaps be hardened against them. Dropping packets based on load would, if done correctly, lead to better graceful degradation.

7. Conclusion

Netcache implements a distributed buffercache in the Linux kernel, complete with encryption and integrity protection, as well as automatic peer discovery and cache management. In addition to the core functionality, interesting side avenues have been explored, such as encryption, peer discovery, and block unification with MD5s. The results presented here demonstrate that the Netcache distributed buffercache can outperform standard Linux at least in some cases, and it is a viable strategy for increasing filesystem performance and merits further research and development.

8. Acknowledgments

We wish to thank Eric Brewer for advising us on this project as our class professor. We would like to thank David Wagner for providing us with feedback on our cryptographic ideas and pointing out the information leakage in the unification scheme. Finally, we wish to express our great appreciation to the UC Berkeley Statistical Computing Facility for generously giving us use of their research computing cluster for testing Netcache.

9. References

[ACPt95] Thomas E. Anderson, David E. Culler, David A. Patterson, and the NOW team. A case for NOW (networks of workstations). *IEEE Micro.*, 1995.

[AN94] Eric A. Anderson and Jeanna M. Neefe. An exploration of network RAM. Computer Science Division, UC Berkeley, December 1994.

[Dik01] Jeff Dike. A user-mode port of the linux kernel. In *Proceedings of the Fourth Annual Linux Showcase & Conference*, 2001. Atlanta.

[DR] Joan Daemen and Vincent Rijmen. The Rijndael block cipher. AES Proposal: Rijndael.

[DWAP94] Michael Dahlin, Randolph Wang, Thomas E. Anderson, and David A. Patterson. Cooperative caching: Using remote client memory to improve file system performance. In *Proceedings of the 1st Symposium on Operating Systems Design and Implementation*, pages 267–280, November 1994.

[KM00] G. Klyne and L. Masinter. RFC 2938: Identifying composite media failures. http://www.zvon.org/tmRFC/RFC2938/Output/index.html, September 2000. Appendix A: The birthday paradox.

[Nat01] National Institute of Standards and Technology. *Federal Information Processing Standards Publication 197: Announcing the Advanced Encryption Standard (AES)*, November 2001.

[Nua02] Nua Ltd. How many online? (worldwide total). http://www.nua.ie/surveys/how_many_online/world.html, May 2002.

[SKW$^+$] Bruce Schneier, John Kelsey, Doug Whiting, David Wagner, Chris Hall, and Niels Ferguson. Performance comparison of the AES submissions. AES'99.

[Wal02] Carl A. Waldspurger. Memory resource management in VMware ESX server. In *Proceedings of the 5th Symposium on Operating Systems Design and Implementation*. USENIX, December 2002.

```
#define MB 200
#define NUMPASSES 10
#define FILE "/mnt/stuff"

long long T() {
  struct timeval tt;
  gettimeofday(&tt, NULL);
  return tt.tv_sec * 1000 * 1000 + tt.tv_usec;
}

main() {
  printf("linear read..\n");
  for (int j=0; j<2; j++) {
    fd = open(FILE, 0);
    for(int k=0; k<MB; k++) {
      printf("    read %d MB, pass %d\n", k, j+1);
      for(int i=0; i<1024; i++) {
        read(fd, buf, 1024);
      }
    }
    close(fd);
  }
  t = T();
  fd = open(FILE, 0);
  for(int k=0; k<NUMPASSES; k++) {
    printf("    read %d MB\n", k);
    for(int i=0; i<1024; i++) {
      long int r = random();
      r = r % (1024 * 1024 * MB);
      if (r < 0) r = -1 * r;
      lseek(fd, r, SEEK_SET);
      read(fd, buf, 1024);
    }
  }
  close(fd);

  t2 = T();
  printf("random reads took %f seconds\n",
    ((float)(t2 - t)) / (float)(1000 * 1000));
}
```

Figure 3: The benchmarking script.