HashCache: Cache Storage for the Next Billion

Anirudh Badam*, KyoungSoo Park*,+, Vivek S. Pai* and Larry L. Peterson*
*Department of Computer Science
Princeton University
+Department of Computer Science
University of Pittsburgh

Abstract

We present HashCache, a configurable cache storage engine designed to meet the needs of cache storage in the developing world. With the advent of cheap commodity laptops geared for mass deployments, developing regions are poised to become major users of the Internet, and given the high cost of bandwidth in these parts of the world, they stand to gain significantly from network caching. However, current Web proxies are incapable of providing large storage capacities while using small resource footprints, a requirement for the integrated multi-purpose servers needed to effectively support developing-world deployments. HashCache presents a radical departure from the conventional wisdom in network cache design, and uses 6 to 20 times less memory than current techniques while still providing comparable or better performance. As such, HashCache can be deployed in configurations not attainable with current approaches, such as having multiple terabytes of external storage cache attached to low-powered machines. HashCache has been successfully deployed in two locations in Africa, and further deployments are in progress.

1 Introduction

Network caching has been used in a variety of contexts to reduce network latency and bandwidth consumption, ranging from FTP caching [31], Web caching [15, 4], redundant traffic elimination [20, 28, 29], and content distribution [1, 10, 26, 41]. All of these cases use local storage, typically disk-based, to reduce redundant data fetches over the network. Large enterprises and ISPs particularly benefit from network caches, since they can amortize their cost and management over larger user populations. Cache storage system design has been shaped by this class of users, leading to design decisions that favor first-world usage scenarios. For example, RAM consumption is proportional to disk size due to in-memory indexing of on-disk data, which was developed when disk storage was relatively more expensive than it is now. However, because disk size has been growing faster than RAM sizes, it is now much cheaper to buy terabytes of disk than a machine capable of indexing that much storage, since most low-end servers have lower memory limits.

This disk/RAM linkage makes existing cache storage systems problematic for developing world use, where it may be very desirable to have terabytes of cheap storage (available for less than US $100/TB) attached to cheap, low-power machines. However, if indexing a terabyte of storage requires 10 GB of RAM (typical for current proxy caches), then these deployments will require server-class machines, with their associated costs and infrastructure. Worse, this memory is dedicated for use by a single service, making it difficult to deploy consolidated multi-purpose servers. When low-cost laptops from the One Laptop Per Child project [22] or the Classmate from Intel [13] cost only US $200 each, spending thousands of dollars per server may exceed the cost of laptops for an entire school.

This situation is especially unfortunate, since bandwidth in developing regions is often more expensive, both in relative and absolute currency, than it is in the US and Europe. Africa, for example, has poor terrestrial connectivity, and often uses satellite connectivity, backhauled through Europe. One of our partners in Nigeria, for example, shares a 2 Mbps link, which costs $5000 per month. Even the recently-planned "Google Satellite," the O3b, is expected to drop the cost to only $500/Mbps per month by 2010 [21]. With efficient cache storage, one can reduce the network connectivity expenses.

The goal of this project is to develop network cache stores designed for developing-world usage. In this paper, we present HashCache, a configurable storage system that implements flexible indexing policies, all of which are dramatically more efficient than traditional cache designs. The most radical policy uses no main

memory for indexing, and obtains performance comparable to traditional software solutions such as the Squid Web proxy cache. The highest performance policy performs equally with commercial cache appliances while using main-memory indexes that are only one-tenth their size. Between these policies are a range of distinct policies that trade memory consumption for performance suitable for a range of workloads in developing regions.

1.1 Rationale For a New Cache Store

HashCache is designed to serve the needs of developing-world environments, starting with classrooms but working toward backbone networks. In addition to good performance with low resource consumption, HashCache provides a number of additional benefits suitable for developing-world usage: (a) many HashCache policies can be tailored to use main memory in proportion to system activity, instead of cache size; (b) unlike commercial caching appliances, HashCache does not need to be the sole application running on the machine; (c) by simply choosing the appropriate indexing scheme, the same cache software can be configured as a low-resource end-user cache appropriate for small classrooms, as well as a high-performance backbone cache for higher levels of the network; (d) in its lowest-memory configurations, HashCache can run on laptop-class hardware attached to external multi-terabyte storage (via USB, for example), a scenario not even possible with existing designs; and (e) HashCache provides a flexible caching layer, allowing it to be used not only for Web proxies, but also for other cache-oriented storage systems.

A previous analysis of Web traffic in developing regions shows great potential for improving Web performance [8]. According to the study, kiosks in Ghana and Cambodia, with 10 to 15 users per day, have downloaded over 100 GB of data within a few months, involving 12 to 14 million URLs. The authors argue for the need for applications that can perform HTTP caching, chunk caching for large downloads and other forms of caching techniques to improve the Web performance. With the introduction of personal laptops into these areas, it is reasonable to expect even higher network traffic volumes.

Since HashCache can be shared by many applications and is not HTTP-specific, it avoids the problem of diminishing returns seen with large HTTP-only caches. HashCache can be used by both a Web proxy and a WAN accelerator, which stores pieces of network traffic to provide protocol-independent network compression. This combination allows the Web cache to store static Web content, and then use the WAN accelerator to reduce redundancy in dynamically-generated content, such as news sites, Wikipedia, or even locally-generated content, all of which may be marked uncacheable, but which tend to only change slowly over time. While modern Web pages may be large, they tend to be composed of many small objects, such as dozens of small embedded images. These objects, along with tiny fragments of cached network traffic from a WAN accelerator, put pressure on traditional caching approaches using in-memory indexing.

A Web proxy running on a terabyte-sized HashCache can provide a large HTTP store, allowing us to not only cache a wide range of traffic, but also speculatively preload content during off-peak hours. Furthermore, this kind of system can be driven from a typical OLPC-class laptop, with only 256MB of total RAM. One such laptop can act as a cache server for the rest of the laptops in the deployment, eliminating the need for separate server-class hardware. In comparison, using current Web proxies, these laptops could only index 30GB of disk space.

The rest of this paper is structured as follows. Section 2 explains the current state of the art in network storage design. Section 3 explains the problem, explores a range of HashCache policies, and analyzes them. Section 4 describes our implementation of policies and the HashCache Web proxy. Section 5 presents the performance evaluation of the HashCache Web Proxy and compares it with Squid and a modern high-performance system with optimized indexing mechanisms. Section 6 describes the related work, Section 7 describes our current deployments, and Section 8 concludes with our future work.

2 Current State-of-the-Art

While typical Web proxies implement a number of features, such as HTTP protocol handling, connection management, DNS and in-memory object caching, their performance is generally dominated by their filesystem organization. As such, we focus on the filesystem component because it determines the overall performance of a proxy in terms of the peak request rate and object cacheability. With regard to filesystems, the two main optimizations employed by proxy servers are hashing and indexing objects by their URLs, and using raw disk to bypass filesystem inefficiencies. We discuss both of these aspects below.

The Harvest cache [4] introduced the design of storing objects by a hash of their URLs, and keeping an in-memory index of objects stored on disk. Typically, two levels of subdirectories were created, with the fan-out of each level configurable. The high-order bits of the hash were used to select the appropriate directories, and the file was ultimately named by the hash value. This approach not only provided a simple file organization, but it also allowed most queries for the presence of objects to be served from memory, instead of requiring disk access. The older CERN [15] proxy, by contrast, stored objects by creating directories that matched the components of the URL. By hashing the URL, Harvest was able to con-

System	Naming	Storage Management	Memory Management
CERN	URL	Regular Filesystem	Filesystem Data Structures
Harvest	Hash	Regular Filesystem	LRU, Filesystem Data Structures
Squid	Hash	Regular Filesystem	LRU & others
Commercial	Hash	Log	LRU

Table 1: System Entities for Web Caches

Entity	Memory per Object (Bytes)
Hash	4 - 20
LFS Offset	4
Size in Blocks	2
Log Generation	1
Disk Number	1
Bin Pointers	4
Chaining Pointers	8
LRU List Pointers	8
Total	32 - 48

Table 2: High Performance Cache - Memory Usage

trol both the depth and fan-out of the directories used to store objects. The CERN proxy, Harvest, and its descendant, Squid, all used the filesystems provided by the operating system, simplifying the proxy and eliminating the need for controlling the on-disk layout.

The next step in the evolution of proxy design was using raw disk and custom filesystems to eliminate multiple levels of directory traversals and disk head seeks associated with them. The in-memory index now stored the location on disk where the object was stored, eliminating the need for multiple seeks to find the start of the object. [1]

The first block of the on-disk file typically includes extra metadata that is too big to be held in memory, such as the complete URL, full response headers, and location of subsequent parts of the object, if any, and is followed by the content fetched from the origin server. In order to fully utilize the disk writing throughput, those blocks are often maintained consecutively, using a technique similar to log-structured filesystem(LFS) [30]. Unlike LFS, which is expected to retain files until deleted by the user, cache filesystems can often perform disk cache replacement in LIFO order, even if other approaches are used for main memory cache replacement. Table 1 summarizes the object lookup and storage management of various proxy implementations that have been used to build Web caches.

The upper bound on the number of cacheable objects per proxy is a function of available disk cache and physical memory size. Attempting to use more memory than the machine's physical memory can be catastrophic for caches, since unpredictable page faults in the application can degrade performance to the point of unusability. When these applications run as a service at network access points, which is typically the case, all users then suffer extra latency when page faults occur.

The components of the in-memory index vary from system to system, but a representative configuration for a high-performance proxy is given in Table 2. Each entry has some object-specific information, such as its hash value and object size. It also has some disk-related information, such as the location on disk, which disk, and which generation of log, to avoid problems with log wrapping. The entries typically are stored in a chain per hash bin, and a doubly-linked LRU list across all index entries. Finally, to shorten hash bin traversals (and the associated TLB pressure), the number of hash bins is typically set to roughly the number of entries.

Using these fields and their sizes, the total consumption per index entry can be as low as 32 bytes per object, but given that the average Web object is roughly 8KB (where a page may have tens of objects), even 32 bytes per object represents an in-memory index storage that is 1/256 the size of the on-disk storage. With a more realistic index structure, which can include a larger hash value, expiration time, and other fields, the index entry can be well over 80 bytes (as in the case of Squid), causing the in-memory index to exceed 1% of the on-disk storage size. With a single 1TB drive, the in-memory index alone would be over 10GB. Increasing performance by using multiple disks would then require tens of gigabytes of RAM.

Reducing the RAM needed for indexing is desirable for several scenarios. Since the growth in disk capacities has been exceeding the growth of RAM capacity for some time, this trend will lead to systems where the disk cannot be fully indexed due to a lack of RAM. Dedicated RAM also effectively limits the degree of multiprogramming of the system, so as processors get faster relative to network speeds, one may wish to consolidate multiple functions on a single server. WAN accelerators, for example, cache network data [5, 29, 34], so having very large storage can reduce bandwidth consumption more than HTTP proxies alone. Similarly, even in HTTP proxies, RAM may be more useful as a hot object cache than as an index, as is the case in reverse proxies (server accelerators) and content distribution networks. One goal in designing HashCache is to determine how much index memory is really necessary.

[1] This information was previously available on the iMimic Networking Web site and the Volera Cache Web site, but both have disappeared. No citable references appear to exist

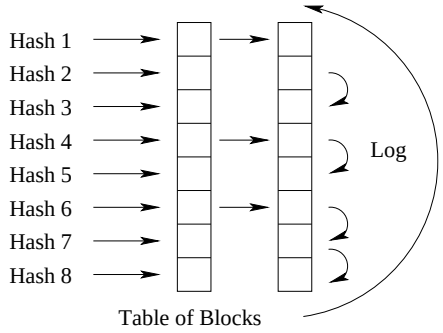

Figure 1: HashCache-Basic: objects with hash value i go to the i^{th} bin for the first block of a file. Later blocks are in the circular log.

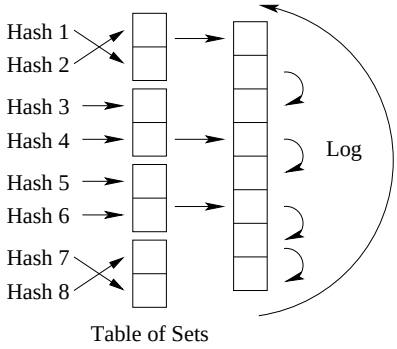

Figure 2: HashCache-Set: Objects with hash value i search through the $\frac{i}{N}^{th}$ set for the first block of a file. Later blocks are in the circular log. Some arrows are shown crossed to illustrate that objects that map on to a set can be placed anywhere in the set.

3 Design

In this section, we present the design of HashCache and show how performance can be scaled with available memory. We begin by showing how to eliminate the in-memory index while still obtaining reasonable performance, and then we show how selective use of minimal indexing can improve performance. A summary of policies is shown in Table 3.

3.1 Removing the In-Memory Index

We start by removing the in-memory index entirely, and incrementally introducing minimal metadata to systematically improve performance. To remove the in-memory index, we have to address the two functions the in-memory index serves: indicating the existence of an object and specifying its location on disk. Using filesystem directories to store objects by hash has its own performance problems, so we seek an alternative solution – treating the disk as a simple hashtable.

HashCache-Basic, the simplest design option in the HashCache family, treats part of the disk as a fixed-size, non-chained hash table, with one object stored in each bin. This portion is called the Disk Table. It hashes the object name (a URL in the case of a Web cache) and then calculates the hash value modulo the number of bins to determine the location of the corresponding file on disk. To avoid false positives from hash collisions, each stored object contains metadata, including the original object name, which is compared with the requested object name to confirm an actual match. New objects for a bin are simply written over any previous object.

Since objects may be larger than the fixed-size bins in the Disk Table, we introduce a circular log that contains the remaining portion of large objects. The object metadata stored in each Disk Table bin also includes the location in the log, the object size, and the log generation number, and is illustrated in Figure 1.

The performance impact of these decisions is as follows: in comparison to high-performance caches, HashCache-Basic will have an increase in hash collisions (reducing cache hit rates), and will require a disk access on every request, even cache misses. Storing objects will require one seek per object (due to the hash randomizing the location), and possibly an additional write to the circular log.

3.2 Collision Control Mechanism

While in-memory indexes can use hash chaining to eliminate the problem of hash values mapped to the same bin, doing so for an on-disk index would require many random disk seeks to walk a hash bin, so we devise a simpler and more efficient approach while retaining most of the benefits.

In HashCache-Set, we expand the Disk Table to become an N-way set-associative hash table, where each bin can store N elements. Each element still contains metadata with the full object name, size, and location in the circular log of any remaining part of the object. Since these locations are contiguous on disk, and since short reads have much lower latency than seeks, reading all of the members of the set takes only marginally more time than reading just one element. This approach is shown in Figure 2, and reduces the impact of popular objects mapping to the same hash bin, while only slightly increasing the time to access an object.

While HashCache-Set eliminates problems stemming from collisions in the hash bins, it still has several problems: it requires disk access for cache misses, and lacks an efficient mechanism for cache replacement within the set. Implementing something like LRU within the set using the on-disk mechanism would require a potential disk write on every cache hit, reducing performance.

3.3 Avoiding Seeks for Cache Misses

Requiring a disk seek to determine a cache miss is a major issue for workloads with low cache hit rates, since an

Policy	Bits Per Object	RAM GB per Disk TB	Read Seeks	Write Seeks	Miss Seeks	Comments
Squid	576-832	9 - 13	~ 6	~ 6	0	Harvest descendant
Commercial	256-544	4 - 8.5	< 1	~ 0	0	custom filesystem
HC-Basic	0	0	1	1	1	high collision rate
HC-Set	0	0	1	1	1	adds N-way sets to reduce collisions
HC-SetMem	11	0.17	1	1	0	small in-mem hash eliminates miss seeks
HC-SetMemLRU	< 11	< 0.17	1	1	< 1	only some sets kept in memory
HC-Log	47	0.73	1	~ 0	0	writes to log, log position added to entry
HC-LogLRU	15-47	0.23 - 0.67	$1 + \epsilon$	~ 0	0	log position for only some entries in set
HC-LogLRU + Prefetch	23-55	0.36 - 0.86	< 1	~ 0	0	reads related objects together
HC-Log + Prefetch	55	0.86	< 1	~ 0	0	reads related objects together

Table 3: Summary of HashCache policies, with Squid and commercial entries included for comparison. Main memory consumption values assume an average object size of 8KB. Squid memory data appears in http://www.comfsm.fm/computing/squid/FAQ-8.html

index-less cache would spend most of its disk time confirming cache misses. This behavior would add extra latency for the end-user, and provide no benefit. To address the problem of requiring seeks for cache misses, we introduce the first HashCache policy with any in-memory index, but employ several optimizations to keep the index much smaller than traditional approaches.

As a starting point, we consider storing in main memory an H-bit hash values for each cached object. These hash values can be stored in a two-dimensional array which corresponds to the Disk Table, with one row for each bin, and N columns corresponding to the N-way associativity. An LRU cache replacement policy would need forward and reverse pointers per object to maintain the LRU list, bringing the per-object RAM cost to (H + 64) bits assuming 32-bit pointers. However, we can reduce this storage as follows.

First, we note that all the entries in an N-entry set share the same modulo hash value (%S) where S is the number of sets in the Disk Table. We can drop the lowest log(S) bits from each hash value with no loss, reducing the hash storage to only H - log(S) bits per object.

Secondly, we note that cache replacement policies only need to be implemented within the N-entry set, so LRU can be implemented by simply ranking the entries from 0 to N-1, thereby using only log(N) bits per entry.

We can further choose to keep in-memory indexes for only some sets, not all sets, so we can restrict the number of in-memory entries based on request rate, rather than cache size. This approach keeps sets in an LRU fashion, and fetches the in-memory index for a set from disk on demand. By keeping only partial sets, we need to also keep a bin number with each set, LRU pointers per set, and a hash table to find a given set in memory.

Deciding when to use a complete two-dimensional array versus partial sets with bin numbers and LRU pointers depends on the size of the hash value and the set associativity. Assuming 8-way associativity and the 8 most significant hash bits per object, the break-even point is around 50% – once more than half the sets will be stored in memory, it is cheaper to remove the LRU pointers and bin number, and just keep all of the sets. A discussion of how to select values for these parameters is provided in Section 4.

If the full array is kept in memory, we call it HashCache-SetMem, and if only a subset are kept in memory, we call it HashCache-SetMemLRU. With a low hash collision rate, HashCache-SetMem can determine most cache misses without accessing disk, whereas HashCache-SetMemLRU, with its tunable memory consumption, will need disk accesses for some fraction of the misses. However, once a set is in memory, performing intra-set cache replacement decisions requires no disk access for policy maintenance. Writing objects to disk will still require disk access.

3.4 Optimizing Cache Writes

With the previous optimizations, cache hits require one seek for small files, and cache misses require no seeks (excluding false positives from hash collisions) if the associated set's metadata is in memory. Cache writes still require seeks, since object locations are dictated by their hash values, leaving HashCache at a performance disadvantage to high-performance caches that can write all content to a circular log. This performance problem is not an issue for caches with low request rates, but will become a problem for higher request rate workloads.

To address this problem, we introduce a new policy, HashCache-Log, that eliminates the Disk Table and treats the disk as a log, similar to the high-performance caches. For some or all objects, we store an additional offset (32 or 64 bits) specifying the location on disk. We retain the N-way set associativity and per-set LRU replacement because they eliminate disk seeks for cache misses with compact implementation. While this approach significantly increases memory consumption, it

can also yield a large performance advantage, so this tradeoff is useful in many situations. However, even when adding the log location, the in-memory index is still much smaller than traditional caches. For example, for 8-way set associativity, per-set LRU requires 3 bits per entry, and 8 bits per entry can minimize hash collisions within the set. Adding a 32-bit log position increases the per-entry size from 11 bits to 43 bits, but virtually eliminates the impact of write traffic, since all writes can now be accumulated and written in one disk seek. Additionally, we need a few bits (assume 4) to record the log generation number, driving the total to 47 bits. Even at 47 bits per entry, HashCache-Log still uses indexes that are a factor of 6-12 times smaller than current high-performance proxies.

We can reduce this overhead even further if we exploit Web object popularity, where half of the objects are rarely, if ever, re-referenced [8]. In this case, we can drop half of the log positions from the in-memory index, and just store them on disk, reducing the average per-entry size to only 31 bits, for a small loss in performance. HashCache-LogLRU allows the number of log position entries per set to be configured, typically using $\frac{N}{2}$ log positions per N-object set. The remaining log offsets in the set are stored on the disk as a small contiguous file. Keeping this file and the in-memory index in sync requires a few writes reducing the performance by a small amount. The in-memory index size, in this case, is 9-20 times smaller than traditional high-performance systems.

3.5 Prefetching Cache Reads

With all of the previous optimizations, caching storage can require as little as 1 seek per object read for small objects, with no penalty for cache misses, and virtually no cost for cache writes that are batched together and written to the end of the circular log. However, even this performance can be further improved, by noting that prefetching multiple objects per read can amortize the read cost per object.

Correlated access can arise in situations like Web pages, where multiple small objects may be embedded in the HTML of a page, resulting in many objects being accessed together during a small time period. Grouping these objects together on disk would reduce disk seeks for reading and writing. The remaining blocks for these pages can all be coalesced together in the log and written together so that reading them can be faster, ideally with one seek.

The only change necessary to support this policy is to keep a content length (in blocks) for all of the related content written at the same time, so that it can be read together in one seek. When multiple related objects are read together, the system will perform reads at less than one seek per read on average. This approach can be applied to many of the previously described HashCache policies, and only requires that the application using HashCache provide some information about which objects are related. Assuming prefetch lengths of no more than 256 blocks, this policy only requires 8 bits per index entry being read. In the case of HashCache-LogLRU, only the entries with in-memory log position information need the additional length information. Otherwise, this length can also be stored on disk. As a result, adding this prefetching to HashCache-LogLRU only increases the in-memory index size to 35 bits per object, assuming half the entries of each set contain a log position and prefetch length.

For the rest of this paper, we assume all the policies to have this optimization except HashCache-LogN which is the HashCache-Log policy without any prefetching.

Policy	Throughput
HC-Basic	$rr = \frac{t}{1+\frac{1}{rel}+(1-chr)\cdot cbr}$
HC-Set	$rr = \frac{t}{1+\frac{1}{rel}+(1-chr)\cdot cbr}$
HC-SetMem	$rr = \frac{t}{chr\cdot\left(1+\frac{1}{rel}\right)+(1-chr)\cdot cbr}$
HC-LogN	$rr = \frac{t}{2\cdot chr+(1-chr)\cdot cbr}$
HC-LogLRU	$rr = \frac{t\cdot rel}{2\cdot chr+(1-chr)\cdot cbr}$
HC-Log	$rr = \frac{t\cdot rel}{2\cdot chr+(1-chr)\cdot cbr}$
Commercial	$rr = \frac{t\cdot rel}{2\cdot chr+(1-chr)\cdot cbr}$

Table 4: Throughputs for techniques, rr = peak request rate, chr = cache hit rate, cbr = cacheability rate, rel = average number of related objects, t = peak disk seek rate – all calculations include read prefetching, so the results for Log and Grouped are the same. To exclude the effects of read prefetching, simply set rel to one.

3.6 Expected Throughput

To understand the throughput implications of the various HashCache schemes, we analyze their expected performance under various conditions using the parameters shown in Table 4.

The maximum request rate(rr) is a function of the disk seek rate, the hit rate, the miss rate, and the write rate. The write rate is required because not all objects that are fetched due to cache misses are cacheable. Table 4 presents throughputs for each system as a function of these parameters. The cache hit rate(chr) is simply a number between 0 and 1, as is the cacheability rate (cbr). Since the miss rate is (1 - chr), the write rate can be represented as (1 - chr) $\cdot$ cbr. The peak disk seek rate(t) is a measured quantity that is hardware-dependent, and the average number of related objects(rel) is always a positive number. Due to space constraints, we omit the derivations for these calculations. These throughputs are

conservative estimates because we do not take into account the in-memory hot object cache, where some portion of the main memory is used as a cache for frequently used objects, which can further improve throughput.

4 HashCache Implementation

We implement a common HashCache filesystem I/O layer so that we can easily use the same interface with different applications. We expose this interface via POSIX-like calls, such as open(), read(), write(), close(), seek(), etc., to operate on files being cached. Rather than operate directly on raw disk, HashCache uses a large file in the standard Linux ext2/ext3 filesystem, which does not require root privilege. Creating this zero-filled large file on a fresh ext2/ext3 filesystem typically creates a mostly contiguous on-disk layout. It creates large files on each physical disk and multiplexes them for performance. The HashCache filesystem is used by the HashCache Web proxy cache as well as other applications we are developing.

4.1 External Indexing Interface

HashCache provides a simple indexing interface to support other applications. Given a key as input, the interface returns a data structure containing the file descriptors for the Disk Table file and the contiguous log file (if required), the location of the requested content, and metadata such as the length of the contiguous blocks belonging to the item, etc. We implement the interface for each indexing policy we have described in the previous section. Using the data returned from the interface one can utilize the POSIX calls to handle data transfers to and from the disk. Calls to the interface can block if disk access is needed, but multiple calls can be in flight at the same time. The interface consists of roughly 600 lines of code, compared to 21000 lines for the HashCache Web Proxy.

4.2 HashCache Proxy

The HashCache Web Proxy is implemented as an event-driven main process with cooperating helper processes/threads handling all blocking operations, such as DNS lookups and disk I/Os, similar to the design of Flash [25]. When the main event loop receives a URL request from a client, it searches the in-memory hot-object cache to see if the requested content is already in memory. In case of a cache miss, it looks up the URL using one of the HashCache indexing policies. Disk I/O helper processes use the HashCache filesystem I/O interface to read the object blocks into memory or to write the fetched object to disk. To minimize inter-process communication (IPC) between the main process and the helpers, only beacons are exchanged on IPC channels and the actual data transfer is done via shared memory.

4.3 Flexible Memory Management

HTTP workloads will often have a small set of objects that are very popular, which can be cached in main memory to serve multiple requests, thus saving disk I/O. Generally, the larger the in-memory cache, the better the proxy's performance. HashCache proxies can be configured to use all the free memory on a system without unduly harming other applications. To achieve this goal, we implement the hot object cache via anonymous `mmap()` calls so that the operating system can evict pages as memory pressure dictates. Before the HashCache proxy uses the hot object cache, it checks the memory residency of the page via the `mincore()` system call, and simply treats any missing page as a miss in the hot object cache. The hot object cache is managed as an LRU list and unwanted objects or pages no longer in main memory can be unmapped. This approach allows the HashCache proxy to use the entire main memory when no other applications need it, and to seamlessly reduce its memory consumption when there is memory pressure in the system.

In order to maximize the disk writing throughput, the HashCache proxy buffers recently-downloaded objects so that many objects can be written in one batch (often to a circular log). These dirty objects can be served from memory while waiting to be written to disk. This dirty object cache reduces redundant downloads during flash crowds because many popular HTTP objects are usually requested by multiple clients.

HashCache also provides for grouping related objects to disk so that they can be read together later, providing the benefits of prefetching. The HashCache proxy uses this feature to amortize disk seeks over multiple objects, thereby obtaining higher read performance. One commercial system parses HTML to explicitly find embedded objects [7], but we use a simpler approach – simply grouping downloads by the same client that occur within a small time window and that have the same HTTP Referrer field. We have found that this approach works well in practice, with much less implementation complexity.

4.4 Parameter Selection

For the implementation, we choose some design parameters such as the block size, the set size, and the hash size. Choosing the block size is a tradeoff between space usage and the number of seeks necessary to read small objects. In Table 5, we show an analysis of object sizes from a live, widely-used Web cache called CoDeeN [41]. We see that nearly 75% of objects are less than 8KB, while 87.2% are less than 16KB. Choosing an 8KB block would yield better disk usage, but would require multiple seeks for 25% of all objects. Choosing the larger block size wastes some space, but may increase performance.

Since the choice of block size influences the set size,

Size (KB)	% of objects < size
8	74.8
16	87.2
32	93.8
64	97.1
128	98.8
256	99.5

Table 5: CDF of Web object sizes

Read Size (KB)	Seeks/sec	Latency/seek (ms)
1	78	12.5
4	76	12.9
8	76	13.1
16	74	13.3
32	72	13.7
64	70	14.1
128	53	19.2

Table 6: Disk performance statistics

we make the decisions based on the performance of current disks. Table 6 shows the average number of seeks per second of three recent SATA disks (18, 60 and 150 GB each). We notice the sharp degradation beyond 64KB, so we use that as the set size. Since 64KB can hold 4 blocks of 16KB each or 8 blocks of 8KB each, we opt for an 8KB block size to achieve 8-way set associativity. With 8 objects per set, we choose to keep 8 bits of hash value per object for the in-memory indexes, to reduce the chance of collisions. This kind of an analysis can be automatically performed during initial system configuration, and are the only parameters needed once the specific HashCache policy is chosen.

5 Performance Evaluation

In this section, we present experimental results that compare the performance of different indexing mechanisms presented in Section 3. Furthermore, we present a comparison between the HashCache Web Proxy Cache, Squid, and a high-performance commercial proxy called Tiger, using various configurations. Tiger implements the best practices outlined in Section 2 and is currently used in commercial service [6]. We also present the impact of the optimizations that we included in the HashCache Web Proxy Cache. For fair comparison, we use the same basic code base for all the HashCache variants, with differences only in the indexing mechanisms.

5.1 Workload

To evaluate these systems, we use the Web Polygraph [37] benchmarking tool, the *de facto* industry standard for testing the performance of HTTP intermediaries such as content filters and caching proxies. We use the Polymix [38] environment models, which models many key Web traffic characteristics, including: multiple content types, diurnal load spikes, URLs with transient popularity, a global URL set, flash crowd behavior, an unlimited number of objects, DNS names in URLs, object life-cycles (expiration and last-modification times), persistent connections, network packet loss, reply size variations, object popularity (recurrence), request rates and inter-arrival times, embedded objects and browser behavior, and cache validation (If-Modified-Since requests and reloads).

We use the latest standard workload, Polymix-4 [38], which was used at the Fourth Cache-off event [39] to benchmark many proxies. The Polygraph test harness uses several machines for emulating HTTP clients and others to act as Web servers. This workload offers a cache hit ratio (CHR) of 60% and a byte hit ratio (BHR) of 40% meaning that at most 60% of the objects are cache hits while 40% of bytes are cache hits. The average download latency is 2.5 seconds (including RTT). A large number of objects are smaller than 8.5 KB. HTML pages contain 10 to 20 embedded (related) objects, with an average size of 5 to 10 KB. A small number (0.1 %) of large downloads (300 KB or more) have higher cache hit rates. These numbers are very similar to the characteristics of traffic in developing regions [8].

We test three environments, reflecting the kinds of caches we expect to deploy. These are the low-end systems that reflect the proxy powered by a laptop or similar system, large-disk systems where a larger school can purchase external storage to pre-load content, and high-performance systems for ISPs and network backbones.

5.2 Low-End System Experiments

Our first test server for the proxy is designed to mimic a low-memory laptop, such as the OLPC XO Laptop, or a shared low-powered machine like an OLPC XS server. Its configuration includes a 1.4 GHz CPU with 512 KB of L2 cache, 256 MB RAM, two 60GB 7200 RPM SATA drives, and the Fedora 8 Linux OS. This machine is far from the standard commercial Web cache appliance, and is likely to be a candidate machine for the developing world [23].

Our tests for this machine configuration run at 40-275 requests per second, per disk, using either one or two disks. Figure 3 shows the results for single disk performance of the Web proxy using HashCache-Basic (HC-B), HashCache-Set (HC-S), HashCache-SetMem (HC-SM), HashCache-Log without object prefetching (HC-LN), HashCache-Log with object prefetching (HC-L), Tiger and Squid. The HashCache tests use 60 GB caches. However, Tiger and Squid were unable to index this amount of storage and still run acceptably, so were limited to using 18 GB caches. This smaller cache is still sufficient to hold the working set of the test, so Tiger and

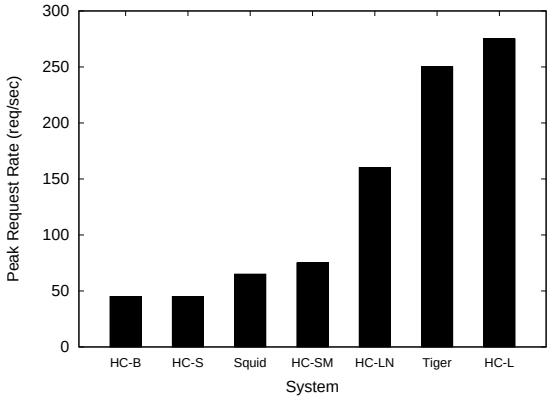

Figure 3: Peak Request Rates for Different policies for low end SATA disk.

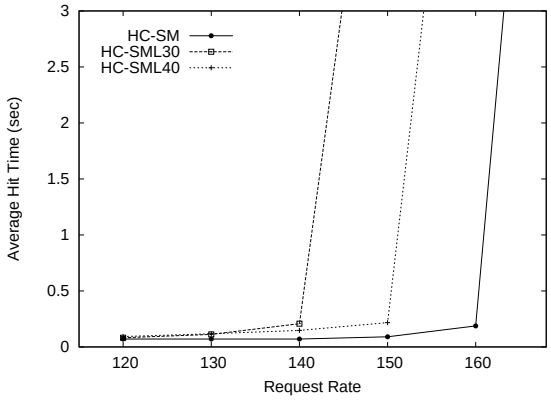

Figure 4: Peak Request Rates for Different SetMemLRU policies on low end SATA disks.

policy	SATA 7200	SCSI 10000	SCSI 15000
HC-Basic	40	50	85
HC-Set	40	50	85
HC-SetMem	66	85	140
HC-LogN	132	170	280
HC-LogLRU	264	340	560
HC-Log	264	340	560
Commercial	264	340	560

Table 7: Expected throughputs (reqs/sec) for policies for different disk speeds– all calculations include read prefetching

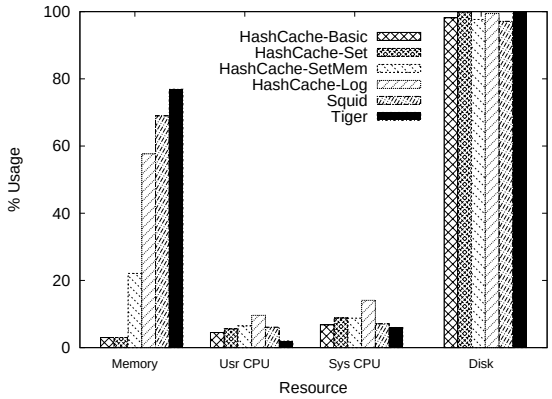

Figure 5: Resource Usage for Different Systems

Squid do not suffer in performance as a result. Table 7 gives the analytical lowerbounds for performance of each of these policies for this workload and the disk performance. The tests for HashCache-Basic and HashCache-Set achieve only 45 reqs/sec. The tests for HashCache-SetMem achieve 75 reqs/sec. Squid scales better than HashCache-Basic and HashCache-Set and achieves 60 reqs/sec. HashCache-Log (with prefetch), in comparison, achieves 275 reqs/sec. The Tiger proxy, with its optimized indexing mechanism, achieves 250 reqs/sec. This is less than HashCache-Log because Tiger's larger index size reduces the amount of hot object cache available, reducing its prefetching effectiveness.

Figure 4 shows the results from tests conducted on HashCache-SetMem and two configurations of HashCache-SetMemLRU using 2 disks. The performance of the HashCache-SetMem system scales to 160 reqs/sec, which is slightly more than double its performance with a single disk. The reason for this difference is that the second disk does not have the overhead of handling all access logging for the entire system. The two other graphs in the figure, labeled HC-SML30 and HC-SML40, are the 2 versions of HashCache-SetMemLRU where only 30% and 40% of all the set headers are cached in main memory. As mentioned earlier, the hash table and the LRU list overhead of HashCache-SetMemLRU is such that when 50% of set headers are cached, it takes about the same amount of memory when using HashCache-SetMem. These experiments serve to show that HashCache-SetMemLRU can provide further savings when working set sizes are small and one does not need all the set headers in main memory at all times to perform reasonably well.

These experiments also demonstrate HashCache's small systems footprint. Those measurements are shown in Figure 5 for the single-disk experiment. In all cases, the disk is the ultimate performance bottleneck, with nearly 100% utilization. The user and system CPU remain relatively low, with the higher system CPU levels tied to configurations with higher request rates. The most surprising metric, however, is Squid's high memory usage rate. Given that its storage size was only one-third that used by HashCache, it still exceeds HashCache's memory usage in HashCache's highest-performance configuration. In comparison, the lowest-performance HashCache configurations, which have performance comparable to Squid, barely register in terms of memory usage.

	Request Rate per sec	Throughput Mb/s	Hit Time msec	All Time msec	Miss Time msec	CHR %	BHR %
HashCache-Log	2200	116.98	77	1147	2508	56.91	41.06
Tiger	2300	121.40	98	1150	2512	56.49	41.40
Squid	400	21.38	63	1109	2509	57.25	41.22

Table 8: Performance on a high end system

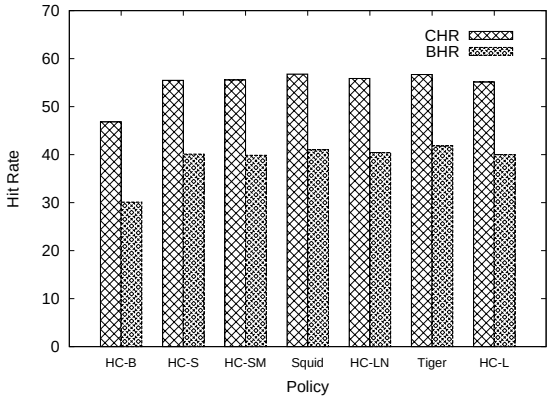

Figure 6: Low End Systems Hit Ratios

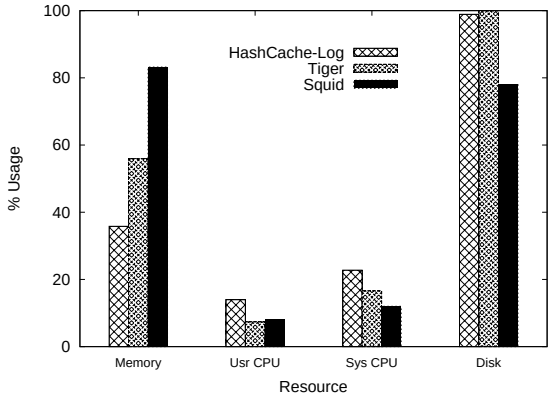

Figure 7: High End System Performance Statistics

Figure 6 shows the cache hit ratio (by object) and the byte hit ratios (bandwidth savings) for the HashCache policies at their peak request rate. Almost all configurations achieve the maximum offered hit ratios, with the exception of HashCache-Basic, which suffers from hash collision effects.

While the different policies offer different tradeoffs, one might observe that the performance jump between HashCache-SetMem and HashCache-Log is substantial. To bridge this gap one can use multiple small disks instead of one large disk to increase performance while still using the same amount of main memory. These experiments further demonstrate that for low-end machines, HashCache can not only utilize more disk storage than commercial cache designs, but can also achieve comparable performance while using less memory. The larger storage size should translate into greater network savings, and the low resource footprint ensures that the proxy machine need not be dedicated to just a single task. The HashCache-SetMem configuration can be used when one wants to index larger disks on a low-end machine with a relatively low traffic demand. The lowest-footprint configurations, which use no main-memory indexing, HashCache-Basic and HashCache-Set, would even be appropriate for caching in wireless routers or other embedded devices.

5.3 High-End System Experiments

For our high-end system experiments, we choose hardware that would be more appropriate in a datacenter. The processor is a dual-core 2GHz Xeon, with 2MB of L2 cache. The server has 3.5GB of main memory, and five 10K RPM Ultra2 SCSI disks, of 18GB each. These disks perform 90 to 95 random seeks/sec. Using our analytical models, we expect a performance of at least 320 reqs/sec/disk with HashCache-Log. On this machine we run HashCache-Log, Tiger and Squid. From the HashCache configurations, we chose only HashCache-Log because the ample main memory of this machine would dictate that it can be used for better performance rather than maximum cache size.

Figure 7 shows the resource utilization of the three systems at their peak request rates. HashCache-Log consumes just enough memory for hot object caching, write buffers and also the index, still leaving about 65% of the memory unused. At the maximum request rate, the workload becomes completely disk bound. Since the working set size is substantially larger than the main memory size, expanding the hot object cache size produces diminishing returns. Squid fails to reach 100% disk throughput simultaneously on all disks. Dynamic load imbalance among its disks causes one disk to be the system bottleneck, even though the other four disks are underutilized. The load imbalance prevents it from achieving higher request rates or higher average disk utilization.

The performance results from this test are shown in Table 8, and they confirm the expectations from the analytical models. HashCache-Log and Tiger perform comparably well at 2200-2300 reqs/sec, while Squid reaches only 400 reqs/sec. Even at these rates, HashCache-Log is purely disk-bound, while the CPU and memory consumption has ample room for growth. The per-disk performance of HashCache-Log of 440 reqs/sec/disk is in

1TB Configuration	Request Rate per sec	Throughput Mb/s	Hit Time msec	All Time msec	Miss Time msec	CHR %	BHR %
HashCache-SetMem	75	3.96	27	1142	2508	57.12	40.11
HashCache-Log	300	16.02	48	1139	2507	57.88	40.21
HashCache-LogLRU	300	16.07	68	1158	2510	57.15	40.08
2TB Configuration	Request Rate per sec	Throughput Mb/s	Hit Time msec	All Time msec	Miss Time msec	CHR %	BHR %
HashCache-SetMem	150	7.98	32	1149	2511	57.89	40.89
HashCache-Log	600	32.46	56	1163	2504	57.01	40.07
HashCache-LogLRU	600	31.78	82	1171	2507	57.67	40.82

Table 9: Performance on large disks

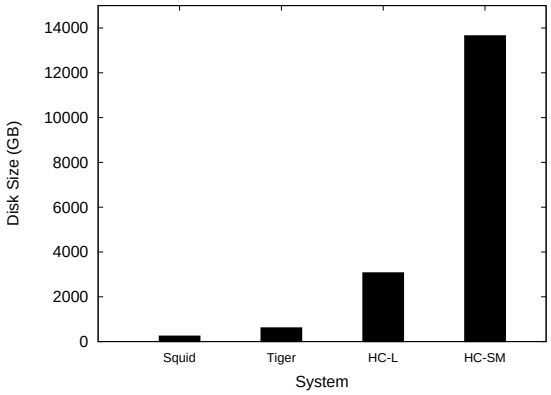

Figure 8: Sizes of disks that can be indexed by 2GB memory

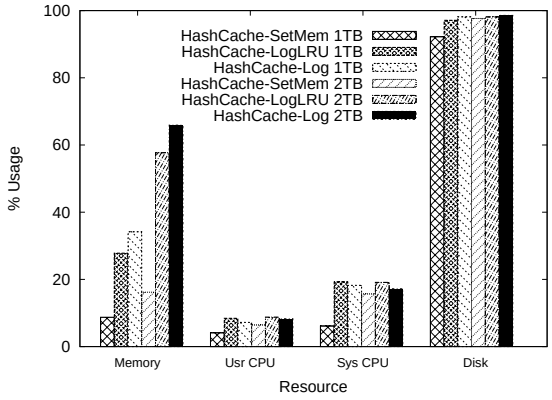

Figure 9: Large Disk System Performance Statistics

line with the best commercial showings – the highest-performing system at the Fourth Cacheoff achieved less than an average of 340 reqs/sec/disk on 10K RPM SCSI disks. The absolute best throughput that we find from the Fourth Cacheoff results is 625 reqs/sec/disk on two 15K RPM SCSI disks, and on the same speed disks HashCache-Log and Tiger both achieve 700 reqs/sec/disk, confirming the comparable performance.

These tests demonstrate that the same HashCache code base can provide good performance on low-memory machines while matching or exceeding the performance of high-end systems designed for cache appliances. Furthermore, this performance comes with a significant savings in memory, allowing room for larger storage or higher performance.

5.4 Large Disk Experiments

Our final set of experiments involves using HashCache configurations with large external storage systems. For this test, we use two 1 TB external hard drives attached to the server via USB. These drives perform 67-70 random seeks per second. Using our analytical models, we would expect a performance of 250 reqs/sec with HashCache-Log. In other respects, the server is configured comparably to our low-end machine experiment, but the memory is increased from 256MB to 2GB to accommodate some of the configurations that have larger index requirements, representative of low-end servers being deployed [24].

We compare the performance of HashCache-SetMem, HashCache-Log and HashCache-LogLRU with one or two external drives. Since the offered cache hit rate for the workload is 60%, we cache 6 out of the 8 log offsets in main memory for HashCache-LogLRU. For these experiments, the Disk Table is stored on a disk separate from the ones keeping the circular log. Also, since filling the 1TB hard drives at 300 reqs/second would take excessively long, we randomly place 50GB of data across each drive to simulate seek-limited behavior.

Unfortunately, even with 2GB of main memory, Tiger and Squid are unable to index these drives, so we were unable to test them in any meaningful way. Figure 8 shows the size of the largest disk that each of the systems can index with 2 GB of memory. In the figure, HC-SM and HC-L are HashCache-SetMem and HashCache-Log, respectively. The other HashCache configurations, Basic and Set have no practical limit on the amount of externally-attached storage.

The Polygraph results for these configurations are shown in Table 9, and the resource usage details are in Figure 9. With 2TB of external storage, both HashCache-Log and HashCache-LogLRU are able to perform 600 reqs/sec. In this configuration, HashCache-Log uses

slightly more than 60% of the system's memory, while HashCache-LogLRU uses slightly less. The hit time for HashCache-LogLRU is a little higher than HashCache-Log because in some cases it requires 2 seeks (one for the position, and one for the content) in order to perform a read. The slightly higher cache hit rates exhibited on this test versus the high-end systems test are due the Polygraph environment – without filling the cache, it has a smaller set of objects to reference, yielding a higher offered hit ratio.

The 1TB test achieves half the performance of the 2TB test, but does so with correspondingly less memory utilization. The HashCache-SetMem configuration actually uses less than 10% of the 2GB overall in this scenario, suggesting that it could have run with our original server configuration of only 256MB.

While the performance results are reassuring, these experiments prove that HashCache can index disks that are much larger than conventional policies could handle. At the same time, HashCache performance meets or exceeds what other caches would produce on much smaller disks. This scenario is particularly important for the developing world, because one can use these inexpensive high-capacity drives to host large amounts of content, such as a Wikipedia mirror, WAN accelerator chunks, HTTP cache, and any other content that can be preloaded or shipped on DVDs later.

6 Related Work

Web caching in its various forms has been studied extensively in the research and commercial communities. As mentioned earlier, the Harvest cache [4] and CERN caches [17] were the early approaches. The Harvest design persisted, especially with its transformation into the widely-used Squid Web proxy [35]. Much research has been performed on Squid, typically aimed at reorganizing the filesystem layout to improve performance [16, 18], better caching algorithms [14], or better use of peer caches [11]. Given the goals of HashCache, efficiently operating with very little memory and large storage, we have avoided more complexity in cache replacement policies, since they typically use more memory to make the decisions. In the case of working sets that dramatically exceed physical memory, cache policies are also likely to have little real impact. Disk cache replacement policies also become less effective when storage sizes grow very large. We have also avoided Bloom-filter approaches [2] that would require periodic rebuilds, since scanning terabyte-sized disks can sap disk performance for long periods. Likewise, approaches that require examining multiple disjoint locations [19, 32] are also not appropriate for this environment, since any small gain in reducing conflict misses would be offset by large losses in checking multiple locations on each cache miss.

Some information has been published about commercial caches and workloads in the past, including the design considerations for high-speed environments [3], proxy cache performance in mixed environments [9], and workload studies of enterprise user populations [12]. While these approaches have clearly been successful in the developed world, many of the design techniques have not typically transitioned to the more price-sensitive portions of the design space. We believe that HashCache demonstrates that addressing problems specific to the developing world can also open interesting research opportunities that may apply to systems that are not as price-sensitive or resource-constrained.

In terms of performance optimizations, two previous systems have used some form of prefetching, including one commercial system [7], and one research project [33]. Based on published metrics, HashCache performs comparably to the commercial system, despite using a much similar approach to grouping objects, and despite using a standard filesystem for storage instead of raw disk access. Little scalability information is presented on the research system, since it was tested only using Apache mod_proxy at 8 requests per second. Otherwise, very little information is publically available regarding how high-performance caches typically operate from the extremely competitive commercial period for proxy caches, centered around the year 2000. In that year, the Third Cache-Off [40] had a record number of vendors participate, representing a variety of different caching approaches. In terms of performance, HashCache-Log compares favorably to all of them, even when normalized for hardware.

Web caches also get used in two other contexts: server accelerators and content distribution networks (CDNs) [1, 10, 26, 41]. Server accelerators, also known as reverse proxies, typically reside in front of a Web server and offload cacheable content, allowing the Web server to focus on dynamically-generated content. CDNs geographically distribute the caches reducing latency to the client and bandwidth consumption at the server. In these cases, the proxy typically has a very high hit rate, and is often configured to serve as much content from memory as possible. We believe that HashCache is also well-suited for this approach, because in the SetMemLRU configuration, only the index entries for popular content need to be kept in memory. By freeing the main memory from storing the entire index, the extra memory can be used to expand the size of the hot object cache.

Finally, in terms of context in developing world projects, HashCache is simply one piece of the infrastructure that can help these environments. Advances in wireless network technologies, such as WiMax [42] or rural WiFi [27, 36] will help make networking available

to larger numbers of people, and as demand grows, we believe that the opportunities for caching increase. Given the low resource usage of HashCache and its suitability for operation on shared hardware, we believe it is well-suited to take advantage of networking advancements in these communities.

7 Deployments

HashCache is currently deployed at two different locations in Africa, at the Obafemi Awolowo University (OAU) in Nigeria and at the Kokrobitey Institute (KI) in Ghana. At OAU, it runs on their university server which has a 100 GB hard drive, 2 GB memory and a dual core Xeon processor. For Internet connection, they pay $5,000 per month for a 2 Mbps satellite link to an ISP in Europe and the link has a high variance ICMP ping time from Princeton ranging 500 to 1200 ms. We installed HashCache-Log on the machine but were asked to limit resource usage for HashCache to 50 GB disk space and no more than 300 MB of physical memory. The server is running other services such as a E-mail service and a firewall for the department and it is also used for general computation for the students. Due to privacy issues we were not able to analyze the logs from this deployment but the administrator has described the system as useful and also noticed the significant memory and CPU usage reduction when compared to Squid.

At KI, HashCache runs on a wireless router for a small department on a 2 Mbps LAN. The Internet connection is through a 256 Kbps sub-marine link to Europe and the link has a ping latency ranging from 200 to 500 ms. The router has a 30 GB disk and 128 MB of main memory and we were asked to use 20 GB of disk space and as little memory as possible. This prompted us to use the HashCache-Set policy as there are only 25 to 40 people using the router every day. Logging is disabled on this machine as well since we were asked not to consume network bandwidth on transferring the logs.

In both these deployments we have used HashCache policies to improve the Web performance while consuming minimum amount of resource. Other solutions like Squid would not have been able to meet these resource constraints while providing any reasonable service. People at both places told us that the idea of a faster Internet to popular Web sites seemed like a distant dream until we discussed the complete capabilities of HashCache. We are currently working with OLPC to deploy HashCache at more locations with the OLPC XS servers.

8 Conclusion and Future Work

In this paper we have presented HashCache, a high-performance configurable cache storage for the developing regions. HashCache provides a range of configurations that scale from using no memory for indexing to ones that require only one-tenth as much as current high-performance approaches. It provides this flexibility without sacrificing performance – its lowest-resource configuration has performance comparable to free software systems, while its high-end performance is comparable to the best commercial systems. These configurations allow memory consumption and performance to be tailored to application needs, and break the link between storage size and in-memory index size that has been commonly used in caching systems for the past decade. The benefits of HashCache's low resource consumption allow it to share hardware with other applications, share the filesystem, and to scale to storage sizes well beyond what present approaches provide.

On top of the HashCache storage layer, we have built a Web caching proxy, the HashCache Proxy, which can run using any of the HashCache configurations. Using industry-standard benchmarks and a range of hardware configurations, we have shown that HashCache performs competitively with existing systems across a range of workloads. This approach provides an economy of scale in HashCache deployments, allowing it to be powered from laptops, low-resource desktops, and even high-resource servers. In all cases, HashCache either performs competitively or outperforms other systems suited to that class of hardware.

With its operation flexibility and a range of available performance options, HashCache is well suited to providing the infrastructure for caching applications in developing regions. Not only does it provide competitive performance with the stringent resource constraint , but also enables new opportunities that were not possible with existing approaches. We believe that HashCache can become the basis for a number of network caching services, and are actively working toward this goal.

9 Acknowledgements

We would like to thank Jim Gettys and John Watlington for their discussions about OLPC's caching needs, and Marc Fiuczynski for arranging and coordinating our deployments in Africa. We also thank our shepherd, Michael Mitzenmacher as well as anonymous NSDI reviewers. This research was partially supported by NSF Awards CNS-0615237, CNS-0519829, and CNS-0520053. Anirudh Badam was partially supported by a Technology for Developing Regions Fellowship from Princeton University.

References

[1] AKAMAI TECHNOLOGIES INC. http://www.akamai.com/.

[2] BLOOM, B. H. Space/time trade-offs in hash coding with allowable errors. *Communications of the ACM 13* (1970), 422–426.

[3] BREWER, E., GAUTHIER, P., AND MCEVOY, D. Long-term viability of large-scale caches. In *Proceedings of the 3rd International WWW Caching Workshop* (1998).

[4] CHANKHUNTHOD, A., DANZIG, P. B., NEERDAELS, C., SCHWARTZ, M. F., AND WORRELL, K. J. A hierarchical internet object cache. In *Proceedings of the USENIX Annual Technical Conference* (1996).

[5] CITRIX SYSTEMS. http://www.citrix.com/.

[6] COBLITZ, INC. http://www.coblitz.com/.

[7] COX, A. L., HU, Y. C., PAI, V. S., PAI, V. S., AND ZWAENEPOEL, W. Storage and retrieval system for WEB cache. U.S. Patent 7231494, 2000.

[8] DU, B., DEMMER, M., AND BREWER, E. Analysis of WWW traffic in Cambodia and Ghana. In *Proceedings of the 15th International conference on World Wide Web (WWW)* (2006).

[9] FELDMANN, A., CACERES, R., DOUGLIS, F., GLASS, G., AND RABINOVICH, M. Performance of web proxy caching in heterogeneous bandwidth environments. In *Proceedings of the 18th IEEE INFOCOM* (1999).

[10] FREEDMAN, M. J., FREUDENTHAL, E., AND MAZIERES, D. Democratizing content publication with coral. In *Proceedings of the USENIX Symposium on Networked Sytems Design and Implementation (NSDI)* (2004).

[11] GADDE, S., CHASE, J., AND RABINOVICH, M. A taste of crispy Squid. In *Workshop on Internet Server Performance* (1998).

[12] GRIBBLE, S., AND BREWER, E. A. System design issues for internet middleware services: Deductions from a large client trace. In *Proceedings of the USENIX Symposium on Internet Technologies and Systems (USITS)* (1997).

[13] INTEL. Classmate PC, http://www.classmatepc.com/.

[14] JIN, S., AND BESTAVROS, A. Popularity-aware greedydual-size web proxy caching algorithms. In *Proceedings of the 20th International Conference on Distributed Computing Systems (ICDCS)* (2000).

[15] LUOTONEN, A., HENRYK, F., LEE, T. B. http://info.cern.ch/hypertext/WWW/Daemon/Status.html.

[16] MALTZAHN, C., RICHARDSON, K., AND GRUNWALD, D. Reducing the disk I/O of Web proxy server caches. In *Proceedings of the USENIX Annual Technical Conference* (1999).

[17] MALTZAHN, C., RICHARDSON, K. J., AND GRUNWALD, D. Performance issues of enterprise level web proxies. In *Proceedings of the ACM SIGMETRICS* (1997).

[18] MARKATOS, E. P., PNEVMATIKATOS, D. N., FLOURIS, M. D., AND KATEVENIS, M. G. Web-conscious storage management for web proxies. *IEEE/ACM Transactions on Networking 10*, 6 (2002), 735–748.

[19] MITZENMACHER, M. The power of two choices in randomized load balancing. *IEEE Transsactions on Parallel and Distributed Systems 12*, 10 (2001), 1094–1104.

[20] MOGUL, J. C., CHAN, Y. M., AND KELLY, T. Design, implementation, and evaluation of duplicate transfer detection in HTTP. In *Proceedings of the USENIX Symposium on Networked Systems Design and Implementation (NSDI)* (2004).

[21] O3B NETWORKS. http://www.o3bnetworks.com/.

[22] OLPC. http://www.laptop.org/.

[23] OLPC. http://wiki.laptop.org/go/Hardware_specification.

[24] OLPC. http://wiki.laptop.org/go/XS_Recommended_Hardware.

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

[26] PARK, K., AND PAI, V. S. Scale and performance in the CoBlitz large-file distribution service. In *Proceedings of the 3rd USENIX Symposium on Networked Systems Design and Implementation (NSDI)* (2006).

[27] PATRA, R., NEDEVSCHI, S., SURANA, S., SHETH, A., SUBRAMANIAN, L., AND BREWER, E. WiLDNet: Design and implementation of high performance wifi based long distance networks. In *Proceedings of the 4th Symposium on Networked Systems Design and Implementation (NSDI)* (2007).

[28] RHEA, S., LIANG, K., AND BREWER, E. Value-based web caching. In *In Proceeding of the 13th International Conference on World Wide Web (WWW)* (2003).

[29] RIVERBED TECHNOLOGY, INC. http://www.riverbed.com/.

[30] ROSENBLUM, M., AND OUSTERHOUT, J. K. The design and implementation of a log-structured file system. *ACM Transactions on Computer Systems 10*, 1 (1992), 26–52.

[31] RUSSELL, M., AND HOPKINS, T. CFTP: a caching FTP server. *Computer Networks and ISDN Systems 30*, 22–23 (1998), 2211–2222.

[32] SEZNEC, A. A case for two-way skewed-associative caches. In *Proceedings of the 20th International Symposium on Computer Architecture (ISCA)* (New York, NY, USA, 1993), ACM, pp. 169–178.

[33] SHRIVER, E. A. M., GABBER, E., HUANG, L., AND STEIN, C. A. Storage management for web proxies. In *Proceedings of the USENIX Annual Technical Conference* (2001).

[34] SILVER PEAK SYSTEMS, INC. http://www.silver-peak.com/.

[35] SQUID. http://www.squid-cache.org/.

[36] SUBRAMANIAN, L., SURANA, S., PATRA, R., NEDEVSCHI, S., HO, M., BREWER, E., AND SHETH, A. Rethinking wireless in the developing world. In *Proceedings of Hot Topics in Networks (HotNets-V)* (2006).

[37] THE MEASUREMENT FACTORY. http://www.web-polygraph.org/.

[38] THE MEASUREMENT FACTORY. http://www.web-polygraph.org/docs/workloads/polymix-4/.

[39] THE MEASUREMENT FACTORY. http://www.measurement-factory.com/results/public/cacheoff/N04/report.by-alph.html.

[40] THE MEASUREMENT FACTORY. http://www.measurement-factory.com/results/public/cacheoff/N03/report.by-alph.html.

[41] WANG, L., PARK, K., PANG, R., PAI, V., AND PETERSON, L. Reliability and security in the CoDeeN content distribution network. In *Proceedings of the USENIX Annual Technical Conference* (2004).

[42] WIMAX FORUM. http://www.wimaxforum.org/home/.