Architecture of The Internet Archive

Elliot Jaffe
The Selim and Rachel Benin School of
Computer Science and Engineering
The Hebrew University of Jerusalem
Givat Ram Campus
91904 Jerusalem, Israel
jaffe@cs.huji.ac.il

Scott Kirkpatrick
The Selim and Rachel Benin School of
Computer Science and Engineering
The Hebrew University of Jerusalem
Givat Ram Campus
91904 Jerusalem, Israel
kirk@cs.huji.ac.il

ABSTRACT

The Internet Archive is a live production system supporting close to a petabyte of data and delivering an average of 2.3Gb/sec of data to Internet users. We describe the architecture of this system with an emphasis on its robustness and how it is managed by a very small team of systems personnel. Notably, the current system does not employ a cache. We analyze the reasons for this decision and show that an effective cache could not be built until now. However, new solid state disk technology may offer promising new cache implementations.

Categories and Subject Descriptors

D.2 [**Software**]: Software Engineering; D.2.8 [**Software Engineering**]: Metrics—*complexity measures, performance measures*

1. INTRODUCTION

The Internet Archive (www.archive.org) is a petabyte scale public Internet library. Its collections include The Wayback Machine that provides access to approximately 500 TB of historical web pages collected from the Internet beginning in 1996, and a Media Collection that contains more than 500 TB of public domain books, audio, video, and images. The Internet Archive has been in continuous operation since 2000. In its current state, the system handles tens of millions of daily requests totaling more than 40 TB of data.

The Internet Archive is not only a good example of a large scale Internet site, but also an architectural and operational success story. One of the most interesting elements of the Internet Archive is that less than five employees are involved in operations and maintenance. Surprisingly, the Internet Archive has accomplished this feat while avoiding almost all of the popular approaches for performance enhancement and storage minimization in favor of a simplest-solution-first strategy.

Many systems [9] implement a reverse proxy cache to improve performance and to reduce the load on its storage units. The Internet Archive has no such reverse-proxy cache. We will show that implementing and operating such a cache is not cost effective using currently available technology. However, newly available Solid State Disks (SSD) are shown to be a promising option for future implementations.

The case study presented here draws on the authors' experience through access to the Internet Archive and discussions with its architects and operations staff. We would like to note that the authors have never worked in a technical capacity at the Internet Archive and are not privy to any internal operational decisions or policies. Our position is one of remote researchers with very limited observational capabilities.

2. SYSTEM ARCHITECTURE

Using an approach similar to Kruhchten's [19], we describe the System Architecture through a number of different views. The requirements section details the goals and constraints which drove and continue to drive architectural decisions. The Logical view exposes the basic system objects. The Process view presents the ongoing activities involved in delivering the basic service and maintaining its integrity. The Development view describes the implementation of these processes. Finally, the Physical view exposes the hardware and software components that implement the system.

2.1 Requirements

The Internet Archive's mission is to be an Internet Library; reliably storing large amounts of data and delivering that data to users on the Internet. The system must be scalable, storing many billions of objects and petabytes of content. The only bottleneck to content delivery should be the Internet Archives connections to the external network. That is, the internal system should be able to scale based on customer demand and available outgoing bandwidth.

The system requires a search and index mechanism to enable users to locate specific items. The designers choose to take this requirement in its minimal interpretation. Items should be searchable by title and by pre-specified keywords. Detailed search at the content level is not necessary.

A library or archival system should make an attempt to maintain its data for many years and to retain that data in the face of component failures. The designers understood that there is a direct relationship between system cost and

its level of reliability [16]. Basic reliability can be achievable with linear cost, but as the requirements grow, the cost to achieve those enhanced goals increases dramatically. The designers therefore intentionally set the minimum requirements rather low: Try not to lose data, but don't try too hard.

As a real-world system, its creator, Brewster Kahle set down a few basic design requirements orthogonal to its basic mission. These requirements are still in force today and have significantly colored the architecture and operations of the archive.

- The system should use only commodity equipment.
- The system should not rely on commercial software.
- The system should not require a PhD degree to implement or to maintain.
- The system should be as simple as possible.

2.2 Logical View

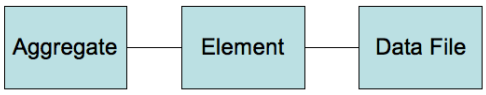

Figure 1: Logical View

The basic building block in the archive is a content element which represents a particular item such as a book, web page or video. Elements are grouped into aggregates such as collections or crawls. A collection might be a set of books as found in the Million Book Project [6], or movies such as The Prelinger Archives [24]. Other types of collections include web crawls performed by third parties or by the Internet Archive itself.

Each element may be composed of multiple data files. For example, a book may have hundreds of pages represented as images and as plain text. A video element may be retained in multiple formats for ease of access. Web pages typically reference other web objects as links or embedded objects. For web objects, each and every item is referenced as a separate element.

2.3 Process View

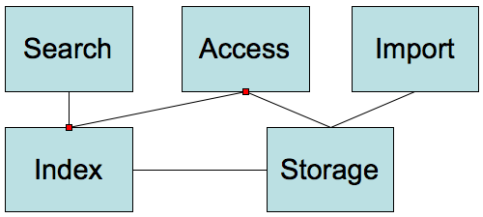

Figure 2: Process View

There are relatively few processes involved in the Internet Archive. The Storage process maintains the integrity of the elements as storage components fail or are retired from service. The Import process accepts items for storage and integrates these new items into the system. The Indexing and Search processes create and maintain the indices over items while enabling users to find specific items. Finally, the Access process delivers those items on demand. Figure 2 shows the relationships between these processes.

2.4 Development View

In this section, we describe the implementation for each of the processes listed above. The presentation is bottom up, first describing the basic components and then utilizing them in subsequent descriptions.

2.4.1 Storage

Logical elements are stored in one or more predefined root directories in the local file system on each storage node. Each root directory represents a entire hard disk, and each element has its own directory. Data files are then stored as files in their element's directory.

Web crawl elements are not stored as a directory because the number of items in that directory would stress most Linux file systems. Instead, these elements are stored as a relatively small number of ARC [4] files. Each file is a set of uncorrelated web pages usually totaling close to 100MB. For each web object, the crawler that gathers these objects appends to the ARC file a header followed by the content of that object. Note that the header appears in the ARC file directly before each item and not in some form of index. This process continues until the file reaches its maximal size at which point the crawler closes that file and opens a new file. One of the challenges in working with ARC files is that they are completely unindexed. The only way to search or access one of its web files is to sequentially scan the entire ARC file. ARC files are stored in their original unmodified form on the node.

In principal, each element is stored on at least two storage nodes. A monitoring service [12] identifies nodes that have failed and disks that are failing [23]. When an error is detected, the operators begin an automated process that copies the contents of the old node to a new node, either from a replica or from the failing node itself. In the past, some data has been lost because two nodes crashed at the same time in the lone data center. The current Internet Archive is replicated across three geographically remote sites, enabling, if necessary, retrieval from one of the remote sites.

An automated recovery system would be an obvious extension to the existing architecture. Such a system would have to manage many issues, including automated error detection and the provisioning of new hardware. In keeping with the aggressively simple implementation approach, such a system was never implemented. Another perspective would be to note that such a system was never needed. The number of failures has never been high enough to cause undue load on the administrators. More details about the Internet Archives hard disk failure rates were published by Shwarz et.al. [25].

Finally, the storage systems provides a form of load balancing. The Internet Archive contains some very popular movies. When one of these movies becomes popular, the storage nodes may be unable to keep up with the number of concurrent requests. The operations staff manually watches for these spikes and copies the files to additional nodes. When filling the nodes, some extra space is set aside for just such situations. There is no need to ever remove these duplicates.

2.4.2 Import

New items arrive at the archive by many paths. They can be uploaded by Internet users, delivered by truck to the Internet Archive's facilities, provided as bulk transfers by partners, or created internally through web crawls or book scans. Regardless of how the data arrives, the process begins by locating the current import nodes; two twin nodes are dedicated to newly imported items. Imported items are stored in parallel to these nodes until they reach a "fill level", some percentage points shy of 100%. When that level is reached, two new empty nodes are provisioned and the process continues.

Where possible, the element, its data files and any metadata are collected during the import process and integrated into the content indices. There is no need to update any indices or metadata related to the new item's location because this data is dynamically determined by the Access process.

2.4.3 Index and Search

The Index process maintains a list of each item in the Internet Archive and any available associated metadata. There are at least two different index implementations.

Each web crawl includes millions of URLs. The total number of URLs in the library is between 2 and 10 billion items. The Internet Archive may store multiple copies of a web page if it was retrieved by distinct crawls. References to each URL are tagged by date. The Wayback Machine first displays the available versions of each URL. Users can then choose to view a specific version from the Internet Archive's collections.

The original design requirements severely limiting the use of specialized software or hardware were taken to mean that the system should not use a database system. In response, The designers chose to store the URL index data in flat sorted files. The system is limited to searching for complete URLs and so the URL name space can be divided into similar size buckets. The system first removes common prefixes such as www and is then sorted by the remaining URL string. Hence the first bucket might contain all URLs that start with A, B or C. The next bucket would contain URLs that start with D, E, F and G, and so forth. Multiple index files are maintained, physically distributed across the main web servers. Requests are statically routed to the appropriate web server and index section based on the requested URL.

The web index was originally designed to be built by a batch process that was to be run each month. The process required reading every unique ARC file, extracting the URLs, sorting the results and finally splitting the index into sections. Until recently, index scans were performed very infrequently because each index scan caused the permanent loss of up to 10 hard disks. The specific cause of the disk failures seems to have been related to insufficient data center cooling capacity. Actively accessing the disks raised the machine room temperature by at least 5 degrees fahrenheit. This problem was addressed by moving the majority of nodes to a more capable data center. More recently, an improved process was developed that can incrementally update the index.

A major challenge for the index and search components are the sheer number of items in the system. There are more than a million ARC files. Assuming that the average page is only 20 kilobytes, each ARC file would then contain on average 5000 page objects. The total system would need to support more than 5 billion pages. The current index has close to that number of entries and requires more than 2 TB of storage.

The separate index is maintained for all other content, including books, movies, and audio recordings. It contains only searchable meta-data such as titles, authors and publication dates. This index is very small when compared to the Wayback Machine. It contains only a few million records and is currently implemented as a MySQL database.

2.4.4 Access

There is no central index detailing the location of elements and data files within the system. To find an item, a request for that object name is sent as a UDP broadcast to all data nodes. A small listener program on each storage node maintains the list of all local files in main memory and looks up each request. Those nodes that have the requested item reply to the broadcast with the local file path and their own node name. The requesting node then redirects the client browser to the appropriate storage node. Each storage node runs a lightweight web server that can efficiently deliver local files.

In effect, the system implements a distributed index that is very robust in the face of failures or updates. Moving a file from one node to another requires only updating the in-memory index on those two nodes. Failure of a node is invisible to the searcher. That node will simply not respond to any requests. This approach also creates a minimal form of load balancing. Heavily loaded nodes will naturally be slightly slower to respond to broadcast requests. Faster, unloaded responses will then be used instead of the subsequent responses from slower nodes.

The process is slightly different for web pages. When a particular URL is requested, the system uses a URL/ARC file index to identify the specific ARC file that contains this URL. The ARC file is then located using the broadcast mechanism and the client browser is redirected to that storage node. The light weight web server then spawns off a small process to open the ARC file, retrieve the page and return it to the browser. While this process is relatively expensive, the number of concurrent requests to ARC files per machine is small. There is sufficient local memory on each storage node to maintain an entire ARC file in RAM. This enables the kernel to prefetch the ARC file and to maintain it locally in case there are temporally close requests for items

in that same ARC file.

2.5 Physical View

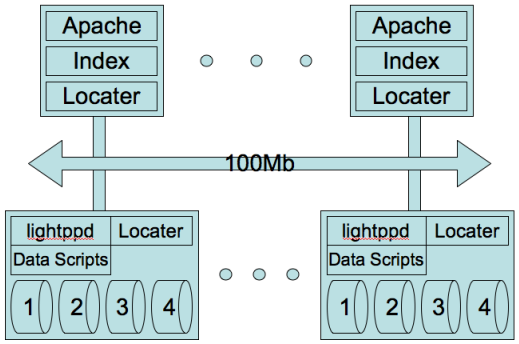

Figure 3: Physical View

The Internet Archive architecture is composed of a small number of front-end web nodes and a large number of back-end storage nodes as shown in Figure 3.

2.5.1 Web Nodes

The Web Nodes are implemented on Apache web servers running in Linux on commodity hardware. A standard load balancer is used to parcel requests between front-end nodes. Since the Web nodes never deliver bulk data, they are tuned for high volumes of short requests. With the exception of search results, all other pages are static, further reducing the computation overhead.

The URL index is stored on these same nodes. It is split into segments and grouped by the first letter of the URL. Each segment is on at least two nodes in case one node should fail. The total index is more than 2 TB of data because of the large number of archived URLs.

The number of Web nodes is dependent on the maximum number of concurrent page requests, but not on the number of concurrent downloads. There are always a minimum of three operational nodes in case of a localized node failure. In practice, the Internet Archive has approximately six web nodes running at all times. These nodes have provided sufficient capacity for all existing needs.

2.5.2 Storage Nodes

Each Storage Node consists of a low power CPU and up to four commodity disks. Each node runs Linux and a lighttpd web server. Files are stored in the local file system. Data nodes are self-contained and do not depend on the activity of any other component in the system.

A simple program implements the location responder. To make things as simple as possible, the location responder performs a name lookup on each file request. The Linux kernel caches file names and so very few searches ever require a physical disk access.

There are more than 2500 storage nodes in the current primary data center, sufficient to keep at least two copies of each data file. The Bibliotheca Alexandrina [2] in Alexandria, Egypt and the European Archive citeeuroparchive in Amsterdam and Paris act as partial replicas.

2.6 Upgrade Path

There are no architectural decisions that strictly depend on a specific software or piece of hardware. As new hardware becomes available, it can easily be integrated into the Internet Archive's network. At the same time, old hardware can be thrown away. The built-in failure management processes will replicate the data on new hardware at the beginning of its life cycle.

When the Internet Archive began major operations, the size of the largest disk was around 30 Gb. Today, it is possible to purchase 1.5 Tb disks. The architecture is independent of the size or number of disks. The operations staff has the flexibility to purchase the most cost effective disks at any given time without concern for software or hardware interactions.

There are no dependencies on the specific hardware platform. Any system that is supported by one of the popular Linux distributions will necessarily include an Apache web server and the Perl interpreter. The Internet Archive requires that all programs be written in a portable scripting language for just this reason. The Internet Archive purchases only commodity hardware and avoids any specialized cards or add-ons. Because the hardware is standard and simple, there is a very high probability that it will be supported by a recent Linux distribution.

3. ACTUAL PERFORMANCE

The Internet Archive currently includes more than 2500 nodes and more than 6000 disks. Outgoing bandwidth is more than 6 Gb/sec. The internal network consists mostly of 100Mb/sec with a 1Gb/sec network connecting the front-end web servers.

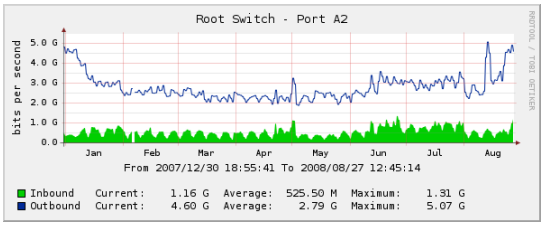

Figure 4: Network load from Jan 2008 through August 2008

Figure 4 presents the daily average network load as incoming and outgoing traffic for the entire archive. This switch is the main concentrator for the US operations. Peak loads are slightly over five gigabits per second with average loads around 2.8 gigabits per second. Data passing through this switch includes all access to the Internet Archive's main data center.

For the remainder of this section, we will focus specifically on non-webcrawl files. The Internet Archive logs accesses to URLs in the Wayback machine separately and we currently do not have access to those records.

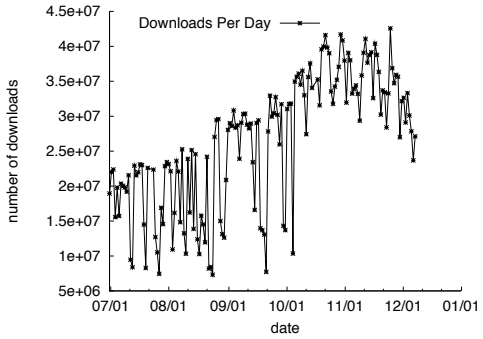

Figure 5: Downloads from July 2008 through early Dec 2008

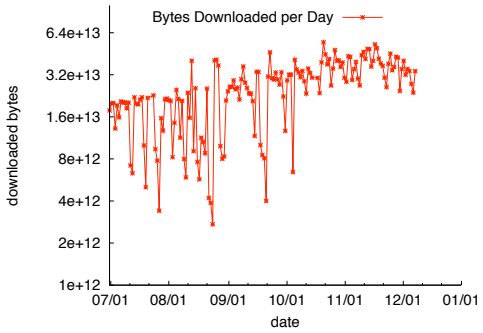

Figure 6: Bytes Downloaded from July 2008 through early Dec 2008

As seen in Figures 5 and 6, during the instrumented period, the Internet Archive served between 2.3 and 48 terabytes per day. The daily download count ranged between 7.3 million and 42.5 million downloads per day.

The local network supports 100Mb/sec and is used on a regular basis for the location broadcasts. Each request includes the name of the requested object, the IP address of the sending node, and a unique 32 bit ID for this request. The response includes the name of the responding node and the local path to the requested file. Empirically, the paths range from 10 to 300 bytes long, with 90% of requests having paths less than 160 bytes. There are on average approximately 800 requests per second with peaks around 1400 requests per second.

The Internet Archive uses a switched fabric for its networks. Thus UDP broadcasts will pass through all nodes, but responses will only travel between the requesting and responding nodes. The current load imposed by the locator mechanism is the number of requests per second times the UDP packet size. At current loads, this corresponds to less than 3% of the total capacity. The system starts to become loaded around 30,000 requests per second, or 48% of the available capacity. Upgrading to a 1Gb/sec network enables this algorithm to scale to more than 300,000 requests per second at 50% capacity.

4. ADDING A CACHE

Many Internet systems include a reverse-proxy cache for static content. Such caches are used to reduce the load on dynamic web servers and to speed up access to content. These issues are not relevant to the Internet Archive because the system already offloads static content to the storage nodes. There is no indication that the existing disk speeds, or the internal or external network bandwidths are bottlenecks to content delivery. A reverse-proxy cache might significantly reduce the load on the storage nodes, potentially enabling these disks to be idled or even shut down for extended periods of time. If the cache is operationally cost effective, it could result in significant cost savings due to wear and tear on storage nodes and due to energy savings on these same nodes.

4.1 Empirical Requirements

To understand the empirical requirements for caching static files at the Internet Archive, we analyzed Media Collection W3C logs collected between July 1, 2008 and December 1, 2008. Each compressed log is between 1Gb and 3Gb with between 7.4 million and 42 million records per file. There are more than 50 million distinct objects referenced in these files. Each object reference is a URL path between 10 and 300 characters long.

For our detailed analysis, we used a specific seven day period from November 1, 2008 through November 7, 2008. During that time, the Internet Archive served 270 million requests and delivered 240 terabytes worth of data. Using consecutive dates balances any unusual activity and improves the simulated cache performance.

For each log file, we parse the log and extract the date, URL, http status and download size. We discard all records that have an invalid size, that represent failed downloads (not a 200 or 206 return value), that were not for the HTTP protocol, or that were not a standard HTTP request (GET, HEAD, or POST). These items should never get to the storage nodes and hence would not effect a cache. In any case, these requests represent less than 4.5% of the requests and no more than 0.00015% of the total download bandwidth.

We produce three data files for each log: a mapping of object IDs to file sizes, a list of object ID's in access order, and a file containing three fields: access time, object ID, and request size. The first two files are used by the stkdst [17] program to compute the priority depth analysis. The third file is used by the webtraff [20] package to compute standard web statistics.

Converting the W3C log files into our three data files was non-trivial. The problem was to maintain a hash table with each object's URL and its mapped ID. As each URL was referenced, we could look them up in the hash table. We found that this table would not fit into the 2Gb memory footprint of most Linux programs.

Our first solution was to implement a distributed hash using memcached [11]. We deployed memcached on four machines, allocating 2Gb of memory to each server. We turned off the LRU replacement feature of memcached, thus turning it into a static hash table. The resulting 8 Gb cache was large enough to process at least 10 days worth of logs. The process took a number of hours to complete and was very fragile.

We finally implemented the conversion algorithm in Hadoop[1]. The process required two map-reduce passes. The first pass parsed the log files and generated key-value pairs where the key was the URL and the value was a vector of the record time and download size. The reduce phase ran as a single reducer and assigned each unique URL key to a new ID. The mapping was written to a secondary file in the Hadoop file system. The second map-reduce phase inverted this process, mapping each record into a time key and a vector value with the id and download size. Hadoop automatically sorts the output by key and so the reduce phase was simply the identity mapping.

The stkdst program implements an LRU stack algorithm [21], generalized to include the size of the requested objects. For each item in an access trace, the algorithm computes the size of the cache in terms of objects and bytes that would be necessary to have stored that item. By sorting the output on priority depth and then computing the cumulative distribution of number of items or item sizes as a function of priority depth, we respectively obtain the hit rate and the byte hit rate as a function of cache size.

The webtraff package is a collection of scripts to compute standard web trace analyses such as popularity, size distribution, bytes and requests per interval, and inter-arrival times.

Both stkdst and webtraff required some modifications. Neither system was designed for the large number of records and items referenced in our logs. For example, both systems limited file sizes to 2 gigabytes in size. We were able to address these problems by porting the code to use 64 bit long values for all relevant operations.

Using priority depth analysis as described in [18], we derived the cache hit rate as a function of the cache size in terms of the percentage of the files served and the percentage of bytes served. The results are shown in Figure 7. Table 1 provides a numeric summary of these charts.

4.2 Sizing the Cache

Using the byte hit rate and taking into account that each download may only be for some fraction of the total file size, we see that the maximum effective cache size is 30 TB. Such a cache would be able to serve 228 terabytes of data, accounting for 91.81% of the downloaded bytes. The other 4.91% of the hits and 8.19% of the downloaded bytes were accessed only once during this period and hence there is nothing to be gained by caching them.

The incremental improvements as the cache sizes grows begin to level off over for caches over 5 TB. A six-fold increase is cache size from 5 TB to 30 TB nets only a 8.38% increase

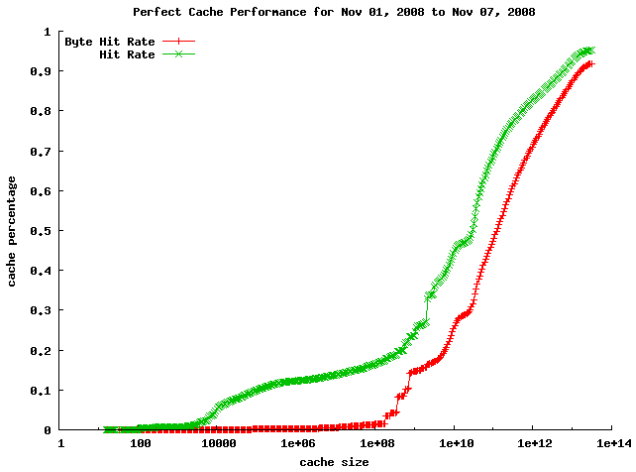

Figure 7: Cache Rates vs. Cache Size from Nov 1, 2008 to Nov 7, 2008

in the number of cached bytes, which is 21 TB of delivered data. We will come back to the cache size once we understand the I/O operations per second and the bandwidth per second that our cache will need to support. Those values are related to the number of cache hits and the size of the requests, but not to the disk footprint of the cache.

4.3 I/Os per Second

As a simplification, let us assume that the cache hits are uniformly distributed during any given period. For the period in question, there were between 107 and 1259 requests per second with a average rate of 447 requests per second. Our cache will thus serve some percentage of that value as determined by the size of the cache.

There are two scaling issues with our proposed cache: bandwidth and requests per second. Bandwidth is an issue at many architectural levels. From the network perspective, a single 10Gb ethernet would be more than the existing external network and hence would be sufficient for existing network traffic. Adding multiple such interfaces or using more than one caching node would provide for expansion room. The other two areas of concern for bandwidth are the disk access bandwidth and the bus speed.

One of the major bottlenecks in current non-memory caches is the disk subsystems. We must consider the disk transfer rates as well as the number of I/O operations per second (IOPS) to the disk subsystem. The traditional approach to limitations in disk bandwidth or IOPS is found in database systems that utilize a large number of small disks. By spreading the data, the database can use all of the disks in parallel, thus multiplying the bandwidth and IOPS by the number of disks.

In order to see if IOPS are an issue for our cache, we will need to translate the number of hits to the cache into IOPS at the disk level. For the purposes of this exposition, let us assume a worst case scenario where the local memory on the caching server cannot cache all active files. That is, each file request will result in at least one disk request.

Table 1: Coverage achieved as function of cache size.

cache size	% bytes	% hits
100 GB	48.09%	69.09%
200 GB	56.37%	74.79%
300 GB	61.09%	77.30%
400 GB	63.85%	78.63%
500 GB	65.78%	79.65%
600 GB	67.60%	80.73%
700 GB	68.77%	81.38%
800 GB	69.87%	81.96%
900 GB	70.97%	82.52%
1 TB	71.53%	82.79%
2 TB	76.79%	85.48%
3 TB	79.72%	87.17%
4 TB	81.61%	88.35%
5 TB	83.43%	89.58%
6 TB	84.32%	90.21%
7 TB	85.69%	91.21%
8 TB	86.14%	91.58%
9 TB	87.04%	92.32%
10 TB	87.93%	92.93%
15 TB	90.13%	94.35%
20 TB	91.20%	94.91%
25 TB	91.68%	95.13%
30 TB	91.81%	95.19%

In most hardware caching scenarios, the block size is fixed [7]. This means that all cache hits return exactly the same amount of data. Our cache must return a variable amount of data because our file sizes are highly variable. Furthermore, our cache must support the current HTTP 1.1 protocol, which allows the requester to download any range of bytes from within the file. Almost half of all requests to the Internet Archive are for ranges of data. Thus, while we cache whole files, we must assume that each request will perform a random seek to the beginning of the requested file segment.

Modern operating systems attempt to pre-fetch parts of the file. The Linux operating system begins with its default block size of 4KB [5]. For each subsequent sequential read, it doubles the size of the prefetch buffer up to a maximum of 128KB. The prefetch algorithm is intended to increase the throughput of the disk by performing sequential reads which do not require additional disk seeks.

Let us assume that the kernel maintains a prefetch buffer for each and every open file descriptor. That is, each open file is treated separately by the kernel and is prefetched on demand. This assumption may be true in practice given the large number of open files served by our cache.

We simulated the prefetch algorithm, counting the number of IOPS necessary per second based on the available trace data. Figure 8 shows the results on a log scale. The minimum value was 552 IOPS. The maximum value was 4021411 IOPS. The average value was 7734 per second. As can be seen, there were a significant number of seconds with more than 50000 IOPS.

It is critical to note that these values are only an approximation. The major challenge to these results is that we assume that all IO operations for each request occur immediately upon arrival of the request. In actuality, the file access is spread over the time necessary to deliver the data to the requester over the Internet. For large files, this can take hours and even days.

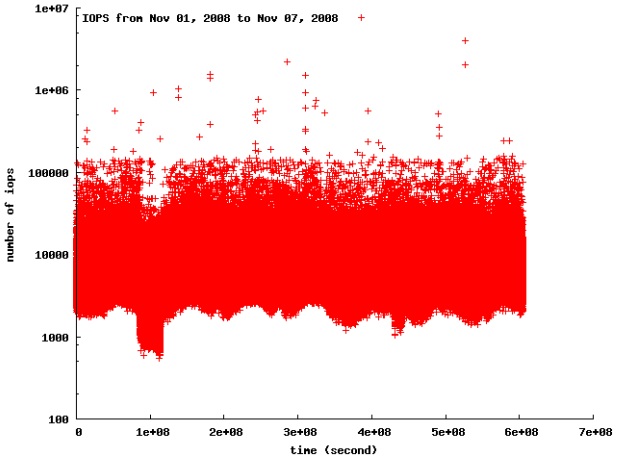

Figure 8: Estimated IOPS over time from Nov 1, 2008 to Nov 7, 2008

In support of our argument, we observe that there are no quiet times in this trace. Spreading the load over time will very likely reduce the very high IOPS rate, but it will also serve to increase the lower values. Furthermore, by spreading the load over time, we expect to see a reduction in kernel prefetch activity, once again increasing the number of IOPS.

Our calculation of IOPS also enable us to determine the required I/O bandwidth. For the period in question, the values ranged from a minimum of 0.007 Gb/sec to a maximum of 263 Gb/sec, with an average value of 0.41 Gb/sec per second. The bandwidth exceeded 1 Gb/sec approximately 5% of the time, and exceeded 2Gb/sec 0.65% of the time. Note that the total available outgoing bandwidth is less than 10 Gb/sec. Our reported value of 263 Gb/sec is a result of assuming that the entire file is downloaded at the moment of the request.

At this point, we note the now well publicized difference between traditional hard disks and solid state disks. In a traditional hard disk, data is stored on rotating platters and read by floating magnetic sensor heads. In order to perform a seek, the sensor must be moved to the correct track on this platter and the platter must complete its rotation to bring the data under the sensor head. For 5400 RPM disks, the average seek time is around 8 milliseconds. If each file access required one seek, the disk would be able to service only 125 operations per second, assuming that transfer time was negligible. In practice most commodity disks service about 100 operations per second with some enterprise disks offering up to 250 IOPS. These same enterprise disks offer sustainable transfer rates of 125MB/sec [26] (1 Gb/sec) over 3Gb/sec SATA interfaces.

Solid state disks have no moving parts and therefore are not subject to rotational seek latencies. Current solid state disks boast between 7000 and 35000 IOPS. Perhaps due to being early on the product curve, Solid states disks use the same 3Gb/sec SATA interfaces and support transfer rates very similar to traditional hard disks.

4.4 Implementation options

Let us now return to the determination of the cache size. As can be seen in Table 1, a one terabyte disk cache would satisfy 82.79% of the hits and 71.53% of the content. There are several ways that such a cache can be implemented.

Web caches are usually implemented using RAM. In-core memory would easily deliver the required IOPS, and also reduce latency relative to disks. Current operating systems limit the size of main memory to between 64 and 128 Gb. Building a 1 TB RAM cache would necessitate implementation as a distributed system, increasing the complexity of the implementation and introducing more points of failure.

An alternative would be to use the existing RAM of each node as a cache. In the Internet Archive, there are 2500 nodes and each node has 512MB of storage for a total of 1.28TB of RAM. Ignoring the fact that some portion of this RAM will be needed for the kernel and for running applications, we might consider using this memory as a in-place cache. Perhaps the biggest challenge would be that many files are larger than 512MB. To cache these files, we would need to split them up across nodes. While possible in principal, we believe that using existing RAM as a cache would violate the keep-it-simple approach in the Internet Archive and would likely reduce the performance of the existing system due to swapping and memory contention.

We might consider using enterprise-class hard disks with 250 IOPS. A single one terabyte hard disk would store all the data, but would never be able to supply the thousands of IOPS that we need. Even if we use ten 100 gigabyte disks, we would have only 2500 IOPS, which is also below our requirements. We could build a ten terabyte cache with one hundred 100 gigabyte disks. This would provide 25000 IOPS, which would support all but the the peak requirements. Unfortunately, using such a large number of disks becomes self defeating because we might as well just continue to use the original storage nodes.

It is in these circumstances that a solid state disk with 20000 IOPS would be a very good fit in terms of IOPS. Having two such disks would likely provide sufficient over capacity to handle even the peak times. There remains the question of disk bandwidth. Our simplistic calculations suggest that two solid state disks would have sufficient bandwidth for more than 99% of the sampled time periods. The proposed calculations that include the downstream bandwidth would likely lower the peak times into the supported range.

4.5 Future Work

The usefulness of a cache for systems like the Internet Archive is by no means obvious. Assuming that it could be built in a cost-effective manner and that it was sufficiently reliable, there remains the question of its impact on the underlying data store. Further research is necessary to model the impact of such a cache on the data access patterns to the backing disk nodes. We believe that such a cache will significantly reduce multiple accesses to file objects. This should allow most replicated data to remain idle and hence we should be able to turn off the disks on which those replicas reside.

We have assumed as a simplification that the cache is read-only. In practice, objects are being loaded into the cache on a regular basis. Cache writes would further increase the number of IOPS. The impact of writes to the cache should be considered when implementing such a system.

Similarly, it would be useful to model the network download performance. This model could then be used to predict the number of bytes delivered per request per second. As mentioned above, this would smooth out the download peaks by spreading the IOPS for an given file over a longer period of time.

5. RELATED WORK

Most Petabyte scale storage systems are proprietary. Published architectural details of petabyte scale storage systems exist for the Google File System[13], the Hadoop Distributed File System (HDFS)[3], and the ATLAS experiment at the Large Hadron Collider (LHC) [10]. All of these systems deal primarily with write-once data, cached data from web crawls in the GFS, log files and static content for HDFS and experimental results from the LHC.

The Google File System (GFS) stores file data in chunks, spread across distributed, commodity storage nodes. Files are stored and accessed through a centralized locater service that maintains the location of the 64MB chunks (blocks) in which the file is stored. Like the Internet Archive, each GFS storage node stores file chunks in its local file system. This is very similar to the ARC file concept in the Internet Archive, where web crawls are stored in "chunks" of 100MB. Google's approach has a centralized management service that can perform automated load balancing and recovery. We are unaware if Google uses any caching infrastructure for the GFS.

The Hadoop Distributed File System is very similar to GFS. It is an open-source file system written entirely in Java. There is no caching infrastructure currently available for HDFS. HFDS serves as the heart of The Yahoo! Search Webmap, running on more than 10,000 linux nodes[27].

The ATLAS experiment will collect 3 to 4 petabytes of data during each year of operation. The system's architects chose a traditional SAN approach to storage. There are a small number of front-end servers with 10 gigabit/sec Ethernet controllers connected to 4 gigabit/sec fiber channel controllers. Disks are managed by SAN controllers and are accessed as block devices through logical unit numbers (LUNs). The ATLAS system implements RAID 6 across the disks. Backup is performed to 800 GB tape drives in a mechanical tape library.

We believe that future large-scale storage systems will be more like GFS and the Internet Archive. Their simplicity,

low cost of hardware, lower operational overhead, and ease of expansion make them good choices for most applications. High performance file systems will still be needed to support database operations and write-frequently data. Yet, as data sets grow and become more static, archival storage systems become more cost effective and easier to maintain over time. As we have shown, the Internet Archive can deliver prodigious quantities of data even though the unit components are not themselves of particularly high performance.

In 2002, Colarelli and Grunwald proposed using massive arrays of idle disks for archive storage [8]. The concept was to spin down idle disks in order to save costs. Pinheiro and Bianchini [22] extended this approach by moving active data from low activity disks to high activity disks in order to increase the locality of reference. They argued that disks never became sufficiently inactive to shut them down and that savings could be had only with adjustable-speed disks. Both of these studies were focused on systems that had significant read and write activities. Our research looks at archival systems, where the impact of "cold" un-accessed data should be significant. We believe that these systems are prime candidates for cost savings. We assume that most accesses can be caught by a cache and that cold data can be segregated so that some significant fraction of the disks can be idled.

The late Jim Gray was one of the strongest proponents of massively over-capacity disk arrays in order to increase disk heads and hence IOPS [15]. His arguments followed from the claim that all systems were in fact databases. Most recently, in a posthumous paper, he argued that Flash disks are highly appropriate for server applications [14]. The Internet Archive shows that large scale archival storage systems are not databases. They have significant over-capacity in terms of IOPS. On the other hand, a cache for such a system would be very similar to a high performance database and would indeed be bound by IOPS and storage bandwidth.

6. CONCLUSION

The Internet Archive is a surprising example of using simple principles to implement a high capacity, highly available system. The core concept is the use of uncomplicated algorithms and simple hardware. Reliability and capacity are due to scale, not intelligence. The Internet Archive represents, perhaps for the first time, a basic example of totally unoptimized system behavior against which, in simulation, one might explore the real value of the most obvious optimizations.

The Internet Archive has been operational for almost 10 years. Its longevity is due to its simplicity. Administrators do not need to spend long months learning the intricacies of the algorithms or installations and hence make fewer mistakes. The Internet Archive is a low cost solution in all terms; software, hardware and operations.

As an extension to the Internet Archive's architecture, we have explored the requirements for a reverse-proxy cache. Using live traces from the Internet Archive, we determined that traditional hard disks are simply not a viable design alternative because they cannot keep up with the large number of I/O operations per second. Furthermore, a complete implementation will likely require a distributed approach in order to guarantee sufficient disk and I/O bus bandwidth during peak loads.

7. ACKNOWLEDGEMENT

This work was supported by Onelab2, an integrated project of the European Community No.*224263.

8. REFERENCES

[1] Apache Software Foundation. Hadoop Core, 2008. `http://hadoop.apache.org/core/`.

[2] Bibliotheca Alexandria, 2009. `http://www.bibalex.org`.

[3] D. Borthakur. *The Hadoop Distributed File System: Architecture and Design*. The Apache Software Foundation, 2007.

[4] M. Burner and B. Khale. WWW Archive File Format Specification, 2002. `http://web.archive.org/web/20021002080721/pages.alexa.com/company/arcformat.html`.

[5] A. R. Butt, C. Gniady, and Y. C. Hu. The performance impact of kernel prefetching on buffer cache replacement algorithms. *SIGMETRICS Perform. Eval. Rev.*, 33(1):157–168, 2005.

[6] Carnegie Mellon University Libraries. Frequently Asked Questions About the Million Book Project, 2008. `http://www.library.cmu.edu/Libraries/MBP_FAQ.html`.

[7] T.-F. Chen and J.-L. Baer. A performance study of software and hardware data prefetching schemes. *Computer Architecture, 1994., Proceedings the 21st Annual International Symposium on*, pages 223–232, Apr 1994.

[8] D. Colarelli and D. Grunwald. Massive arrays of idle disks for storage archives. In *Supercomputing '02: Proceedings of the 2002 ACM/IEEE conference on Supercomputing*, pages 1–11, Los Alamitos, CA, USA, 2002. IEEE Computer Society Press.

[9] B. D. Davison. A survey of proxy cache evaluation techniques. In *WCW99: Proceedings of the Fourth International Web Caching Workshop*, pages 67–77, 1999.

[10] D. Deatrich, S. Liu, C. Payne, R. Tafirout, R. Walker, A. Wong, and M. Vetterli. Managing petabyte-scale storage for the ATLAS Tier-1 Centre at TRIUMF. In *HPCS '08: Proceedings of the 2008 22nd International Symposium on High Performance Computing Systems and Applications*, pages 167–171, Washington, DC, USA, 2008. IEEE Computer Society.

[11] B. Fitzpatrick. Distributed caching with memcached. *Linux J.*, 2004(124):5, 2004.

[12] C. Gaspar. Deploying Nagios in a Large Enterprise Environment. In *LISA*. USENIX, 2007.

[13] S. Ghemawat, H. Gobioff, and S.-T. Leung. The Google file system. *SIGOPS Oper. Syst. Rev.*, 37(5):29–43, December 2003.

[14] J. Gray and B. Fitzgerald. Flash disk opportunity for server applications. *Queue*, 6(4):18–23, 2008.

[15] J. Gray and P. Shenoy. Rules of thumb in data engineering. *Data Engineering, International*

Conference on, 0:3, 2000.

[16] W. Hou and O. Okogbaa. Reliability and availability cost design tradeoffs for HA systems. *Reliability and Maintainability Symposium, 2005. Proceedings. Annual*, pages 433–438, 24-27, 2005.

[17] T. Kelly. Priority depth (generalized stack distance) implementation in ANSI C, 2000. `http://ai.eecs.umich.edu/œtpkelly/papers/`.

[18] T. Kelly and D. Reeves. Optimal web cache sizing: scalable methods for exact solutions. *Computer Communications*, 24(2):163 – 173, 2001.

[19] P. Kruchten. The 4+1 view model of architecture. *IEEE Softw.*, 12(6):42–50, 1995.

[20] N. Markatchev and C. Williamson. WebTraff: A GUI for web proxy cache workload modeling and analysis. In *MASCOTS '02: Proceedings of the 10th IEEE International Symposium on Modeling, Analysis, and Simulation of Computer and Telecommunications Systems*, page 356, Washington, DC, USA, 2002. IEEE Computer Society.

[21] R. L. Mattson, J. Gecsei, D. R. Slutz, and I. L. Traiger. Evaluation techniques for storage hierarchies. *IBM Systems Journal*, 9(2):78, 1970.

[22] E. Pinheiro and R. Bianchini. Energy conservation techniques for disk array-based servers. In *ICS '04: Proceedings of the 18th annual international conference on Supercomputing*, pages 68–78, New York, NY, USA, 2004. ACM.

[23] E. Pinheiro, W.-D. Weber, and L. A. Barroso. Failure trends in a large disk drive population. In *FAST '07: Proceedings of the 5th USENIX conference on File and Storage Technologies*, pages 2–2, Berkeley, CA, USA, 2007. USENIX Association.

[24] R. Prelinger. www.prelinger.com, 2008. `http://www.panix.com/~footage/`.

[25] T. Schwarz, M. Baker, S. Bassi, B. Baumgart, W. Flagg, C. van Ingen, K. Joste, M. Manasse, and M. Shah. Disk failure investigations at the internet archive. In *MSST2006: 23rd IEEE, 14th NASA Goddard Conference on Mass Storage Systems and Technologies*, May 2006.

[26] S. Technology. Seagate technology - cheetah hard drive family, 2009. `http://www.seagate.com/www/en-us/products/servers/cheetah/`.

[27] Yahoo!, 2008. `http://developer.yahoo.net/blogs/hadoop/2008/02/yahoo-worlds-largest-production-hadoop.html`.