DiSK: A Distributed Shared Disk Cache for HPC Environments

Brandon Szeliga, Tung Nguyen, and Weisong Shi
Wayne State University
{aj3638, nttung, weisong}@wayne.edu

Abstract—**Data movement within high performance environments can be a large bottleneck to the overall performance of programs. With the addition of continuous storage and usage of older data, the back end storage is becoming a larger problem than the improving network and computational nodes. This has led us to develop a Distributed Shared Disk Cache, DiSK, to reduce the dependence on these back end storage systems. With DiSK requested files will be distributed across nodes in order to reduce the amount of requests directed to the archives. DiSK has two key components. One is a Distributed Metadata Management, DIMM, scheme that allows a centralized manager to access what data is available in the system. This is accomplished through the use of a counter-based bloomfilter with locality checks in order to reduce false positives and false negatives. The second component is a method of replication called Differentiable Replication, DiR. The novelty of DiR is that the requirements of the files and capabilities of underlying nodes are taken into consideration for replication. This allows for a varying degree of replication depending on the file. This customization of DiSK yields better performance than the conventional archive system.**

I. Introduction

In many scientific areas experiments are being conducted continuously. These experiments yield results that must continually be stored and maintained. Currently data is on the order of gigabytes or even terabytes, so it won't be long until this data is on the order of petabytes or larger. However scientists continually rely on this archived data for analysis with their current data. How is this explosion of data going to impact the way scientists write their applications and configure their systems for use?

Currently with these systems, the approach for application scientists to utilize collected data is to have the files stored in their archive system, and migrated to nodes as needed. When finished using this data, the nodes act as a cache keeping the data in hopes of minimizing extra transfer times needed in the future. Although this is better, a problems still exists. If two different nodes are accessing a similar set of files cold (not having any of these files in their local cache), both of them need to access the back end storage in order retrieve the file sets. This can both increase unnecessary work on the back end and also delay the time until execution. Furthermore if the number of files involved in job execution is too large, then the maintenance involved in controlling where these files are staged can become a bottleneck in the process. Another problem with this approach is that as disks and network resources increase their abilities, the archival back end for these systems are increasingly becoming a bottleneck for system execution. These back ends no longer are able to keep pace with the other aspects of the system, and thus create a bottleneck without hardware updates. However hardware upgrades may not be desirable due to the amount of data stored (a typical example, Google is still using old Pentium III machines for data storage). In order to allow these legacy archives to continue their use we propose **DiSK**, a Distributed Shared Disk Cache system. With this system older archival systems can continue to be used without frequent or recent updates and not suffer from a bottleneck in data access.

DiSK uses hierarchy of "caches" (in disk storage) available to all nodes in the system. By having caches distributed across the system and open to all nodes, file sets that are similar do not need to be remigrated from the archive. Instead, nodes can check this hierarchy and migrate files from the cache. Furthermore in order to allow centralized schedulers a chance to monitor the data available, a counter-based bloomfilter with locality-checks is used.

Using simulation via DiskSim, with DiSK it is possible to improve the average time to migrate data by up to 36% and at the same time lower the amount of migrations required from archives. In the best case, the amount of migrations can be lowered to the number of files in the archive. However even in non-optimal settings it is possible to reduce this.

To allow further customization available to DiSK, we present a new method of replication called Differentiable Replication, or simply **DiR**. The novelty of DiR is that the requirements of the files and capabilities of underlying nodes are taken into consideration for replication. DiR can be flexibly customized at the file granularity. We show that by using this method, files with a reliability requirement can achieve a lower level of migration from the archives compared to files without this requirement (40-70% compared to 85-93%).

The main contribution of this paper is an extensive analysis of the performance impact of DiSK usage using the DiskSim environment. A secondary contribution is the replication management scheme DiR, which allows for a varying degree of replication depending on the file. Thirdly, we present a preliminary prototype of DiSK built on top of Chord/DHash

[1]. A fourth contribution is a version of DiSK that is capable of being combined with a centralized scheduler for monitoring the system. This is the Distributed Metadata Management, or **DIMM**. DIMM acts as a small part of DiSK, requiring a bloomfilter to do so. However due to the possibility of false positives in bloomfilter we develop a locality check for it. Our final contribution in this paper is a systematic analysis of the impact of neighborhood based locality check in counter-based bloomfilter on false positives and false negatives.

The rest of this paper is organized as follows. Section II explains the DiSK mechanism and its components. In Section III our simulations of DiSK's performance are discussed. The current DiSK implementation is described in Section IV along with preliminary results obtained. Next, Section V will discuss the differences between our approach and other related work. Finally, Section VI summarizes our approach and concludes this paper with our future work.

II. Our Approach: DiSK

DiSK is composed of several components, a set of APIs for high level users, a distributed hash table based distributed cache, and a metadata management DIMM. DIMM uses a counter-based bloomfilter with locality check for this management. Next, we will discuss the component details.

A. APIs

The users of our system may be domain experts, or systems software developers, such as job scheduling. All they want is to get the data from the huge and very slow responding archive in a faster way. Therefore, from their point of view, we only need to supply a `retrieve` method with the filename as its input. Secondly, our users may also be system software developers. Beside `retrieve`, these users may wish to have at least two other methods such as `stage-in` and `has`. Method `stage-in` is used to migrate the file from archive to its home node. To provide differentiated services to different files, DiSK also provide a `service_type` parameter to the `stage-in` method. Method `has` checks whether the requested file is at its home node. It returns the node ID if the file is in the system. For example, the central scheduler in Figure 2 is a user of DiSK. It may want to call the `has` to check if the data needed for a submitted job is already in the DiSK and where it is.

B. DiSK Design

In the design of DiSK each file is mapped to a “home” node and stored in a home cache. Other nodes that access this file cache it within their local cache. Overall the system is shown in Figure 2. In this diagram the nodes are displayed as a combination of “H” and “C”. These represent the home and local cache, respectively. Notice how the backend storage of the system is displayed disjoint from the nodes. This shows how the archives are not a part of this mapping.

In order to accomplish this mapping, a distributed hash table is used. A Distributed Hash Table, **DHT** [2], is a mechanism whereby a file is capable of being stored at a specific location according to the file's hash value (this location is referred to as the “home” location through the rest of this paper). Nodes are arranged on a ring of all possible hash values. Each node is responsible for the range of hash values occurring prior to its predecessor.

If we fix the hash to be based on the file's name, then we have a mechanism whereby any node requesting the file is capable of determining the home location of a file within the DHT. Furthermore, a functionality provided by DHTs is the ability to store and retrieve a copy of the data. Thus, any node is capable of retrieving a copy of the data that may be stored. However in order to store the data we need another mechanism this leads us to the Archival Manager.

The Archival Manager is the portion of the system responsible for managing the data requested out of the archive system. For every request of data the Archive Manager grants the request by storing a a copy of the data in the DHT. This allows the DHT to act as a distributed cache for requested files. By acting as a lower layer of cache (a higher level being the local cache), requests for similar files from different machines can be satisfied via the DHT rather than by the archives each time. Thus reducing the dependency on the older archive system.

A functionality the DHT provides is the ability to retrieve files from it. Thus by accessing this method users are able to retrieve files from the DHT without having to access the back end archives. Another benefit that can be gained from using the DHT is if the set of nodes participating in the DHT are the same as the set of computation nodes. In this situation if the user selected to perform computation on a file's home location no additional cost will be necessary to retrieve the file if it is already in the DHT.

C. Differentiable Replication

Further improvements are possible via replication management within the DHT. The most obvious one provided by the DHT is to distribute files to the following *k-successors* of their ”home”. The benefit is in the event of the default home leaving the DHT, because of failures or upgrades, the new home for the file is already set correctly by the inherent consistent hashing feature, and no additional set up cost is incurred. The problem with this approach however is that all files will be replicated in this fashion and can waste storage space for files that will be accessed less frequently. Since different files may require different amounts of replication with different requirements on this

Function name	Description
`retrieve(filename)`	retrieve data from the DiSK
`stage-in(filename, service_type)`	insert data from the archive to its home node in DiSK with specified service type for DiR use
`has(filename)`	check whether the requested file is already in DiSK

TABLE I
THE DISK APIS.

replication, we wanted to create a method which could accommodate this. This is what led to the design of Differentiable Replication, or simply **DiR**.

The key idea in DiR is that not all files have the same requirements on their replication. Files that are brand new might have stricter requirements on their replication as opposed to older files which may not be used as often. Thus the performance gains of caching these older files might be minimal, or even decrease the overall performance. DiR allows the users to control the replication of files by specifying how important a file is.

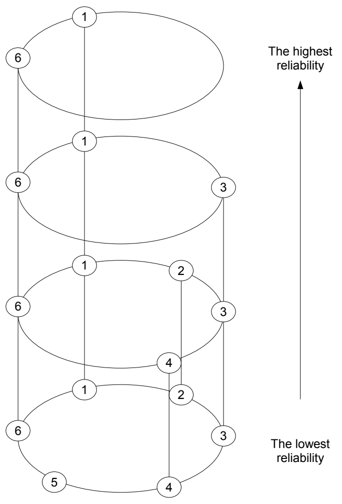


Fig. 1: A view of the DHT with DiR.

By integrating DiR as a portion of the DHT a simple method is available for controlling file replication. By having the user indicate how important a file is when it is being moved into the DHT, this can be used as a direct indicator of the degree of replication to use for the file for the k-successor case. However this is not the only method available for use.

Another possible application of DiR could be to rank locations for the replications to be placed. In this scenario if all the nodes in the DHT are ranked, this ranking can be used with the user indicated importance to determine where the replicas of a file should be placed. For example this ranking can be based on disk size (nodes capable of storing more files), node reliability, or other factors defined by the system administrator. To this end we argue, DiR provides a method of replication that depends on both the requirements of a file and the capabilities of the nodes.

Looking at Figure 1 one can see the differences between the normal DHT and one with a DiR implementation. With the DiR implementation the DHT can be treated as several levels of DHT. For example, in the top of Figure 1 one can see that nodes 1 and 6 maintain the highest reliability. Therefore if a file is hashed between nodes 1 and 2 and requires this level of reliability, the home for the file would be on node 2 (as before), but the replication of the file would follow the DHT for the level of reliability. Thus the replication would be placed on node 6, since it is the successor node for the location of the file hash.

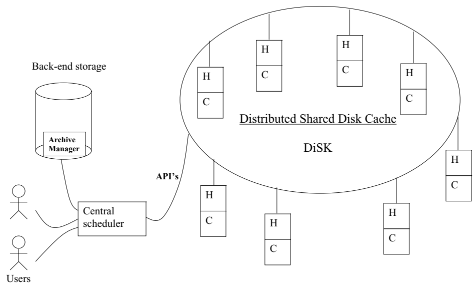


Fig. 2. An overview of the whole DiSK system. The "H" and "C" represent the home and local caches, respectively, on the nodes.

D. DIMM

With centralized scheduling of jobs in these high performance environments new obstacles are discovered. One such obstacle, is in the event of storage-aware job scheduling [3]. In this method, a centralized scheduler (or manager) assumes that the bottleneck for job execution is the stagging of data to the nodes. Thus to minimize this, jobs are placed when their data exists in the system.

In order to allow for this future of schedulers, we give the centralized scheduler a way for it to keep track of what files exist within DiSK at a time. This is done with a bloomfilter in DIMM, which stands for distributed metadata management.

A bloomfilter is a data structure that is capable of telling what data from a set A also exist in a separate set B [4]. It consists of a bit array that is a constant size (k) larger than the size of A all set to 0. For every item a that exists in A, if a wants to be added into B then it is hashed by h hash functions. For each hash function, the corresponding bit in the array will be set to 1. When these bits have been set it can be recognized that item a may exist in B.

There are two advantages to using a bloomfilter. The first one is that the storage for the structure is low. This is because the array in use is a bit array, which makes for low overhead. The second advantage is that the access times for an item in the bloomfilter is fast. This is because the hash functions are designed to be fast and the rest of the structure is setting a bit or checking to see if it is set. This allows for the fast access times that will be needed by a manager.

A simple bloomfilter can accept entry into a set, not removal from it. To remove items in a simple bloomfilter is to clean the entire set and then reinsert items, but this is costly. To allow removals, an extended version of the bloomfilter is chosen, a counter-based bloomfilter [5]. Instead of containing just a bit, this structure has an integer field. As items are added to the structure, the integer value will be incremented. Similarly, removed items with be decremented. This way we can see how many times a bit has been set and allow the system to remove files from the system as they become no longer used. This bloomfilter will continually allow for fast lookups, inserts, and removals from a system at the cost of a small amount of additional storage.

A second problem with any bloomfilter structure is its tendency to have false positives. An example of a false positive is when a file is said to exist in the system when it really does not. This can occur when the hash values from file *a* and file *b* have the hash values from file *c* as a subset. This situation is a large concern in our system, because the manager and machines will proceed like normal until the point where the job is ready to execute on the designated node, and then it becomes clear the file is not already there for execution. In this case, the system will face a major slowdown as everything needs to stop and receive the file before normal execution can take place.

In order to control the probability of a false positive, our bloomfilter also has a locality check included. The locality check operates by taking every initial query to the bloomfilter and obtaining a result. If the query responds that the file exists in the bloomfilter set, then the locality check queries the previous and following *d distance* of alphabetic neighbors. For each of these checked files, if they are also in the system, then a certain degree of confidence can be determined for the original file. With this locality check, a system can now make a more informed choice as to whether the desired dataset exists in the system or whether it is a false positive by comparing it to an established threshold (confidence above the threshold implies the file is likely in the system, and vice versa). The problem is in doing so we are actually introducing the possibility of a false negative into the system.

A false negative scenario is not a concern when using distributed hash table to move the files. This is because a false positive is a much larger bottleneck than a false negative. If a false negative does occur, the scheduling node will believe that the file does not exist within the system. It will then try to move the file into the system by contacting the distributed hash table to stage it to its home location where it already exists. The manager will then be notified that the job is already there and it will not have to update the bloomfilter. At which point the job will be ready to be scheduled.

III. Experiment and Results

In order to demonstrate the effectiveness of DiSK as opposed to a conventional system, we have designed the following simulation. Using the DiskSim simulator (version 2.0) [6] we have designed a simulation of DiSK. In this simulation the archives are composed from a RAID-5 system and all the computation nodes to be simple disks. All network connections are a randomly assigned uniform time between 1 and 10 milliseconds. The trace used in all evaluations has been generated by using a random distribution for the node and for the file needed and contains a total of nearly 5 million requests.

In these simulations we concentrate evaluation on several metrics of the system. First, the ratio of the average time per request in the tested system to the basic system to compare the average amount of time required per file request (denoted as *Access Ratio*). The migration percentage compares the relative number of migrations from the archive to the system (denoted as *Migration %*). Our final metric is the relative number of local file hits (no data movement required) for each file request (denoted as *Hit %*).

A. Raw Cache Performance

In our first simulation we ran our trace on a set of 32 nodes without any additional replication involved. In these experiments the only impact is the DiSK architecture compared to a conventional approach and the relative sizes of the caches for the nodes. Table II displays a comparison between various versions of DiSK and its counterpart system.

In Table II the system are capable of holding a thousand files. From the table, one can see that DiSK significantly outperforms the conventional system configuration. It is capable of reducing the access time ratio by at least 15% while maintaining less than 1% of data needed to be migration percentage. We also did the experiment on the system with smaller capacity, the result shows that as the capabilities of the nodes involved in the system grow, the nodes are able to maintain all the files for longer and thus not require migrating them from the archive as often.

Now that we know some of the capabilities that DiSK is capable of without replication, we can investigate what performance impact replication may have. If we adjust the configuration of DiSK slightly to allow for *k-successors* replication we can see the results displayed in Table III. Although the use of replication increases the average file request time for DiSK, a drastic increase is not apparent until 3-successors of replication is used. Prior to this, the performance is still approximately 35% better with a higher hit percentage. After this point however the performance drops to 53% worse. The results in the percentage of required migrations from back end follow a similar pattern. Once again this performance change occurs because the size of the cache is no longer capable of supporting this degree of replication without impacting the surrounding nodes greatly. So prior to this saturation of the cache, replication helps, but only limitedly before it starts to downgrade the performance.

Configuration	Cache Size		**Access Ratio**	**Migration %**	**Hit %**
Basic	1000		1.00	90.02%	9.98%
	Home Size	Local Size			
DiSK	500	500	0.64	0.20%	8.11%
DiSK	400	600	0.64	0.20%	9.11%
DiSK	600	400	0.65	0.20%	7.11%
DiSK	750	250	0.65	0.20%	5.61%
DiSK	250	750	0.85	19.97%	9.98%

TABLE II
TABLE OF RESULTS FOR VARIOUS CONFIGURATIONS OF HPC ENVIRONMENTS.

Configuration	Cache Size		Replication	**Access Ratio**	**Migration %**	**Hit %**
Basic	1000		0	1.00	90.02%	9.98%
	Home Size	Local Size				
DiSK	500	500	1	0.64	0.21%	9.99%
DiSK	500	500	2	0.64	0.38%	9.98%
DiSK	500	500	3	1.53	46.10%	9.99%

TABLE III
TABLE OF RESULTS FOR VARIOUS CONFIGURATIONS OF HPC ENVIRONMENTS.

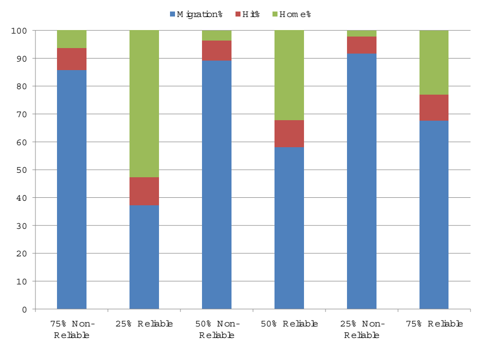


Fig. 3. The relative percentage of each occurrence of different files that do and do not require reliability constraints. All simulations were run on DiSK with 500 files for home and local cache, no *k-successors* used, and 4 reliable nodes.

	2 Reliable Nodes		4 Reliable Nodes		8 Reliable Nodes	
File Type	**Migration %**	**Hit %**	**Migration %**	**Hit %**	**Migration %**	**Hit %**
Normal Files	90.35%	7.87%	85.81%	7.82%	71.27%	7.99%
Reliable Files	61.56%	9.46%	37.28%	10.06%	6.59%	10.25%

TABLE IV
THE PERFORMANCE DIFFERENCE BETWEEN FILES THAT DO AND DO NOT REQUIRE RELIABILITY CONSTRAINTS. ALL SIMULATIONS WERE RUN ON DISK WITH 500 FILES FOR HOME AND LOCAL CACHE, WITH NO *k-successors*, AND USED 25% FILES REQUIRING RELIABILITY.

Using DiSK it is possible to increase performance compared to that of a conventional system. However the drawback is that the setup of DiSK for a system is highly evolved. The larger the caches, the better the performance and less migrations required from the back end. Ideally, the home cache across the nodes should be large enough to hold their share of the files. This will make sure that the nodes can contain all the files and limit the total amount of migrations from the storage.

B. *Differentiable Replication*

We will now analyze the performance impacts posed by using Differentiable Replication. In these simulations there are two degrees of replications, either the files have a requirement on their replication or they do not. Similarly there are only two degrees of reliability among the nodes, reliable and unreliable nodes. Reliable nodes are distributed around the ring, and files that are consider reliable are only a select percentage of the total number of files.

Figure 3 display the normalized values for each variation of files requiring reliability. This is run on the 32 node configuration with 4 reliable nodes. All nodes are using a home and local cache size of 500 files. As the number of files that require reliability increases their performance starts to decrease. The migration percentage for the reliable files changes from under 40% to nearly 70%. This result is expected, however what was not expected was the decrease in non-reliable performance. Throughout the simulations the migrations percentage remains in a small range, but steadily increasing (from 85% to over 90% migrations). Both of these are due to more replacements occurring in the cache of the reliable nodes and thus more migrations and less hits.

The reason the change in the number of reliable files impacting normal files is that reliable files are being replicated to a select few reliable nodes. If we change the number of nodes available as reliable nodes then there should be a change in the performance of the reliable files. This simulation with 25% files requiring reliability is summarized in Table IV. As previously mentioned, reliable files were causing collisions with non-reliable files, and thus files were being replaced from their cache causing an increase in migrations needed and a decrease in hits. However if the number of reliable nodes increases, this results in reliable files being more distributed across the system, and thus less migrations and more hits for both reliable and non-reliable files.

An important thing to notice in this simulations is that the reliable files, have a better hit and migration percentage compared to the other files. Thus in principal the idea of Differentiable Replication works when applied to reliability.

C. *Bloomfilter Performance*

Now we will focus on the various performance aspects related to DIMM and the choices to use a bloomfilter compared to a database and the choice of the locality checks.

1) Bloomfilter with Locality Checks: In order to test the effectiveness of locality checks on false positives/negatives, we created a pool of one million files for bloomfilter array sizes of 10,000,000, 1,000,000, 500,000, and 250,000. The bloomfilter used three SHA-1 functions with different seeds and a MD5 function to create the four hash values. These hash values were then used to determine the position in the bloomfilter to set.

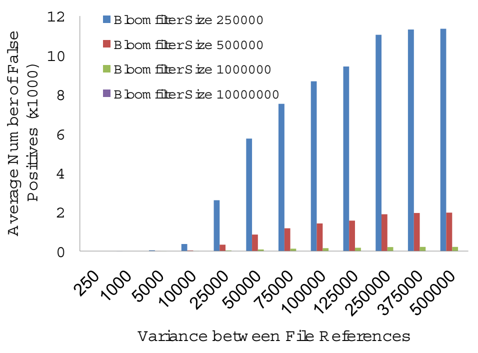
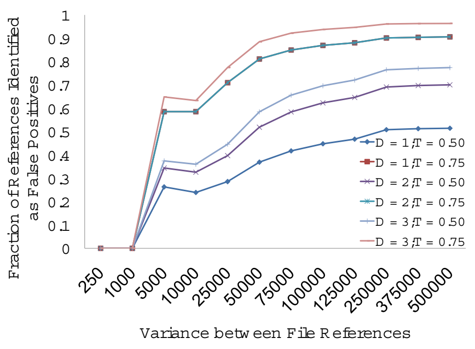


Fig. 4. The average number of false positives relative to the number of entries.

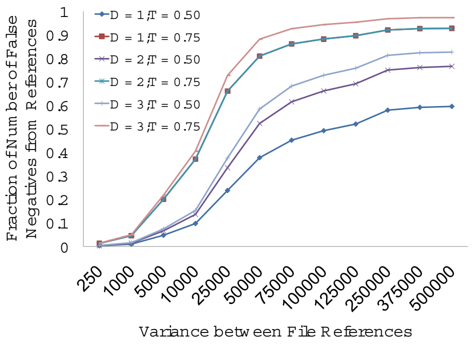
We then ran a normal distribution selecting a total of a 100,000 files from this pool. If the selected file doesn't exist in the bloomfilter, we would input it into the bloomfilter and mark it in an extensive list of all the files (an array consisting of all the files and an on/off bit). On the other hand if the file does exist we would carry out the locality check and determine the degree of confidence. We would also check the extensive list and see if the file was a false positive or was it a reference to a file existing in the bloomfilter. From here we could compare the locality check with the list conclusion and determine the effectiveness of the locality check. We could also identify items as false positives and false negatives.

Figure 4 summarizes the number of false positives generated in different scenarios. There are several observations that can be implied from this figure. First, this supports the idea that as the size of the bloomfilter is increased the number of false positives is decreased. Secondly, we can see that as the variance increases more false positives are incurred. This is due to the fact that more files are available to be inserted into the bloomfilter. These additional files increase the probability of false positives occurring in the bloomfilter.

Now that a baseline for the number of false positives generated has been created, we are able to analyze the impact of the locality check. Figure 5(a) illustrates the percentage of the false positives that were correctly identified as false positives (entries that failed the locality check and are not in the system) whereas Figure 5(b) displays the percentage of entries incorrectly identified as false positives (entries that failed the locality check, but really are in the system). Both of these figures are for different combinations of distances (D) and thresholds (T). From Figure 5(a) it is possible to see that even in our weakest test of locality (a distance of 1 and a threshold of 50%) it is possible to identify over half of the false positives (when the number of entries and the variance among them is large). In the worse case, however, this locality test can backfire and not identify any of the false positives (as in the case of low number of entries and low variance among them). Both of these situations are extreme and we can see that generally the test performs in the range



(a) The average percentage of false positives identified for distance (D) and Threshold (T).



(b) The average percentage of false negatives identified for distance (D) and Threshold (T).

Fig. 5. The effects of locality checking in a bloomfilter of size 250,000.

25-45% reduction depending on the variance among the data.

The above conclusions are only for the bloomfilter of size 250,000, but the other bloomfilter sizes behave similarly (they have been omitted due to page restrictions). All of the graphs have similar low false negatives identified for low variances, and higher false negative rates for higher variances. Comparing across the graphs it is interesting to note that as the bloomfilter size increases the false negative rate also increases.

IV. DiSK Implementation

In addition to the simulations above, we also implemented a prototype of the DiSK system. Figure 6 shows the general diagram of the prototype with main classes, methods, and properties. One thing to notice here is that the links in the figure are used to express function calls between blocks or classes, not the class relationships (parent-child relation for example). From the figure we can see that the system is composed of several units. There is the archive manager, the modified Chord DHT [1], and the end-user library. Each of these components will be discussed below.

The archive manager is responsible for managing the archive node for user file requests. This is accomplished through the use of ARPC (Asynchronous Remote Procedure

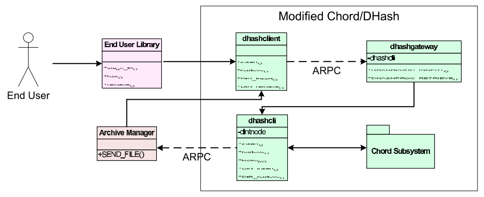


Fig. 6. The implemented system prototype.

Calls), obtained in the sfslite library package [7], and also the dhashclient library, part of the Chord/DHash library. In our system structure in Figure 2, this archive manager is located within the back-end storage block. The archive manager functions like a delivery service inserting files into the DHT when called upon.

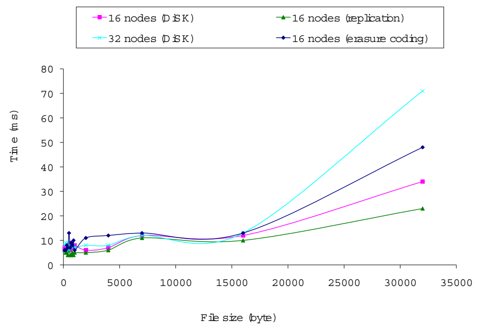
The main DHT used is from the Chord/DHash open-source from MIT. This has been modified so that the `insert` and `retrieve` methods provided by the DHT are updated with `DiR_insert` and `DiR_retrieve` for `stage-in` and replication placement (see Section II-C for details). These functions operate on a block level, so extra processing was added to store files instead of blocks.

The final portion of our implementation is the End User Library. This library is responsible for managing the local cache copies of data, along with allowing users access to bring new files in and retrieve them for access.

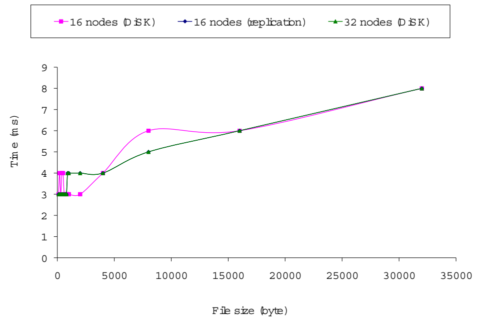
To create a comparison for our changes we installed Chord/Dhash (snapshot 20080217 and SFSlite 0.8.16) onto a cluster of 4 nodes, each node has 4 virtual nodes since the original Chord/DHash requires at least 16 nodes. The original Chord/DHash provides 2 redundancy mechanisms: replication and erasure coding. Therefore, to have an equal comparison, we configured the tested systems so that they all have the same degree of redundancy. We then inserted/retrieved files with different sizes ranging from 100 bytes to 32KB into/out-of the system. In order to vary the system size, we did the same experiments on a cluster of 8 nodes, each has 4 virtual nodes. Next, we deployed our developed prototype onto these two clusters and repeated the experiments.

Figure 7(a) shows the execution time of the file insert operations with different file sizes. From this figure, we can see that the execution time of the insert using the original replication mechanism on the 16-vnode system is the smallest. Our replication method introduced a little more overhead because we need to spend time on finding the reliable nodes not just the successor nodes to put our replicas.Also from the figure, the execution time of the erasure coding method is higher than the replication. This is reasonable because of the overhead in the encoding and decoding process. Moreover, there is no surprise as we see the latency of the insert function increases as system size increases (32 vnodes).

Figure 7(b) shows the execution time of file retrieve methods on different file sizes and in different systems. According to the figure, the execution time of the retrieve methods



(a) The execution time of the insert (write) operation.



(b) The execution time of the retrieve (read) operation.

Fig. 7. Comparison of the execution time of the DiSK prototype and conventional Chord/DHash [1].

are in general smaller than their insert counterparts. This shows there is no significant difference between replication methods, since there is no failure. Second, the execution times in two different size systems are comparable. This may be because our test systems and file sizes were not large enough to see the difference.

V. Related Work

This paper is a combination of different areas of system work: replication, distributed hash table, and data management. However no previous work has combined these areas with the same intent in mind.

Remote caching was first introduced by Dahlin et. al [8]. Our work is an extension of this work, as we design it to be distributed as well as decentralized from the archive while available to all nodes.

There are many distributed file systems which are related to our work. One of them, for example, is the L-Store file system [9] developing at Vanderbilt University. The authors of L-Store want it to be a complete virtual file system that can take into account the varied QoS issues in the distributed environment. With that goal in their mind, L-Store was developed in the lower level. The files in L-Store are broken into many slices and put on different media. In DiSK, we work on a higher level and put all blocks of a file at the same site. Perhaps the closest work to ours is Bhagwan et. al in their Total Recall system [10]. Although their work deals with replication and uses a DHT to accomplish this, they have put their work together in an entirely different fashion. Their work concentrates on the automation of replication within the DHT at the block level, whereas our method proposes a new method for replication using the DHT to help manage it on a file level. Also their system is concentrated as a file system, whereas our system is more of an application level system.

Work on the Cooperative File System [11] shares the similar goal as ours. But again this work is concentrating on using the DHT to work as a file system rather than the application level. Their work assumes that if nodes reach their limitation in storage and some members leave the system, the data may be lost. In this work we assume that the archive is persistent, and thus loss of data is not an issue. In our system, when saturated, the system removes the least recently used files before inserting new data from the archive. Therefore, the read operations always work and data never lost. Another similar work is the Google BigTable an application of the Google File System [12], [13]. In this work the goal is to manage the metadata associated with data in a reliable and efficient manner while distributing these management roles across multiple nodes. However in their approach, they utilize large amounts of metadata stored on a centralized node. This is contrary to our approach which utilizes a DHT in order to avoid this extra metadata management. Having a comprehensive comparison between DiSK and Google's approach in a large scale is an interesting direction.

A lot of previous work has been done in the area of replication. Previous work includes the development of erasure codes [14]. This work is contrary to ours because we wish to use the file directly if it is stored locally without any extra overhead for decoding the file. A project that has used erasure codes is RobuSTore [15]. Similar to our project, RobuSTore cached files across disks. However the goal with the erasure codes was to provide a robust (low variation) performance, not to improve performance.

Some work with replication strategies has been specifically designed for the peer to peer environments. Cohen and Shenker developed a method for replicating data in unstructured peer to peer environments [16]. However this work is concerned with reducing the time for queries to find the data. In our system by utilizing the DHT for data placement queries are reduced to checking a handful of nodes, as opposed to all neighbors.

VI. Conclusion and Future Work

Data management and replication within HPC environments is an area of research where many things need to be investigated. In this work we proposed a method of distributing a shared disk cache using a DHT. With this approach it is possible for different nodes to access the cached file without having to access the back end archive. This result is capable of reducing the amount of time required for the files by at most 36% in our simulations. At the same time we were capable of reducing the amount of migrations required to the minimum in the optimal case, while in non optimal cases a reduction was still achieved (from 99% in the basic system to 76-91% in DiSK, depending on the configuration).

Furthermore we analyzed different mechanisms which can be used in order to further increase the replication of a file. The first mechanism is the *k-successors* replication which is a part of the DHT. The second mechanism is our Differentiable Replication (DiR). In DiR we claim that different files can have different degrees of replication required. With these different required degrees we can cater the replication strategy to the file. Future work in this project is looking at creating a fully functional version of DiSK and creating a version of DiR that is capable of monitoring machine reliability and dynamically adjusting file requirements as the usage frequency changes.

References

[1] I. Stoica, R. Morris, D. Karger, M. F. Kaashoek, and H. Balakrishnan, "Chord: A scalable peer-to-peer lookup service for internet applications," in *ACM SIGCOMM'2001*, 2001.

[2] D. Karger, E. Lehman, T. Leighton, M. Levin, D. Lewin, and R. Panigrahy, "Consistent hashing and random trees: Distributed caching protocols for relieving hot spots on the world wide web," in *Proc. of ACM STOC*, 1997.

[3] J. Cavicchio, B. Szeliga, and W. Shi, "Stroage-aware job scheduling for data-intensive applications," Wayne State University, Tech. Rep. Technical Report MIST-TR-2007-012, Nov 2007.

[4] B. Bloom, "Space/time trade-offs in hash coding with allowable errors," *Communications of the ACM*, vol. 13, no. 7, pp. 422–426, July 1970.

[5] L. Fan, P. Cao, J. Almeida, and A. Z. Broder, "Summary cache: a scalable wide-area web cache sharing protocol," *IEEE/ACM Trans. Netw.*, vol. 8, no. 3, pp. 281–293, 2000.

[6] B. W. Greg Ganger and Y. Patt, "The disksim simulation environment version 2.0 reference manual," 1999.

[7] M. Krohn, "Sfslite library," 2006. [Online]. Available: http://www.okws.org/doku.php?id=sfslite#sfslite

[8] M. Dahlin, R. Wang, T. Anderson, and D. Patterson, "Cooperative caching: Using remote client memory to improve file system performance," in *Proc. of the First USENIX Symposium on Operating Systems Design and Implementation*, Nov. 1994.

[9] A. Tackett *et al.*, "Qos issues with the l-store distributed file system," Oct 2006.

[10] R. Bhagwan, K. Tati, Y.-C. Cheng, S. Savage, and G. M. Voelker, "Total recall: System support for automated availability management," in *Proc. of NSDI'04*. Berkeley, CA, USA: USENIX Association, 2004, pp. 25–25.

[11] F. Dabek, M. F. Kaashoek, D. Karger, R. Morris, and I. Stocia, "Wide-area cooperative storage with CFS," in *Proc. of the 18th ACM Symp. on Operating Systems Principles (SOSP-18)*, Oct. 2001.

[12] F. Chang *et al.*, "Bigtable: A distributed storage system for structured data," in *OSDI 2006*, Nov. 2006.

[13] S. Ghemawat, H. Gobioff, and S. Leung, "The google file system," in *Proceedings of SOSP'03*, Lake George, NY, Oct. 2003.

[14] L. Rizzo, "Effective erasure codes for reliable computer communication protocols," *SIGCOMM Comput. Commun. Rev.*, vol. 27, no. 2, pp. 24–36, 1997.

[15] H. Xia and A. A. Chien, "Robustore: a distributed storage architecture with robust and high performance," in *Supercomputing '07*, 2007.

[16] E. Cohen and S. Shenker, "Replication strategies in unstructured peer-to-peer networks," in *Proc. of ACM SIGCOMM'02*, Aug. 2002.