DIMM: A Distributed Metadata Management for Data-Intensive HPC Environments

Brandon Szeliga, John Cavicchio and Weisong Shi
Wayne State University
{aj3638, ba6444, weisong}@wayne.edu

***Abstract*—**
Within the scientific community many high performance applications are used in order to run experiments on data sets. These data sets can be very large in size or in number. Both of these situations can cause problems to the centralized manager scheduling the system. In our approach we minimize the role of the manager by using a distributed hash table. This way all files have a given "home" location to be at if they are going to be used which reduces location maintenance. We further reduce the strain on the central manager by using a counter-based bloomfilter. This allows the central manager to quickly and easily see if a given data set exists in the system without having to use the large storage space of a database. By adding false positive detection in the form of locality checks to the bloomfilter, we can reduce the probability of a false positive causing a problem in our system. In this fashion we can move away from a centralized manager to a more distributed one while reducing the amount of additional metadata the manager needs to maintain. With our approach we show that the workload of the centralized manager directory has been reduced significantly. Not only are we capable of reducing the number of files that must be migrated into the system by up to nearly 90% compared to a centralized storage-aware scheduling but it is also possible to achieve greater hit rates than a standard cache if job placement is influenced by data location. We also show that a significant number of false positives can be detected from the bloomfilter (at least 25%) at the cost of allowing smaller false negative instances to occur at an increased rate.

I. Introduction

With the increase in availability of high performance computing platforms there has been an increase in the importance of job scheduling as increasing numbers of users utilizing system resources. This is a natural side effect of having more people using these resources. As more people use the available system resources, it becomes more pertinent to schedule their jobs in a manner such that the total elapsed time of job execution commonly known as the job makespan, or wall clock time, is minimized.

Systems such as this usually have a centralized manager that is responsible for determining this schedule of execution. This manager is then responsible for making sure the code executes on the scheduled machine, has its necessary input data, and finally retrieves the result and sends it back to the user. If the system becomes very large (in the number of machines, the number of input files, the number of jobs, and so on), this can create a bottleneck on the manager.

In recent years, the amount of scientific data collected has reached levels that no longer are capable of being stored (e.g., petabytes) on local computer centers. In several applications, such as High Energy Physics, data stored in back-end tapes is fetched using an on-demand fashion by 100s-1000s of scientists worldwide. We envision that in the foreseeable future, the access of data out of tape for analysis is usually the main bottleneck, rather than computing resources.

Within this paper we have three contributions. First we propose **DIMM**, a **DI**stributed **M**etadata **M**anagement scheme, for high performance computing environments leveraging distributed hash table (DHT) and bloomfilter. Second we systematically investigate the impact of false positives and false negatives in a bloomfilter and propose solutions respectively by using a *neighborhood based locality checks* and using a *counter-based bloomfilter*. Finally we compare DIMM with a centralized metadata scheduler in the context of storage-aware job scheduling and found a greater hit rate for data is possible, but more importantly the number of file migrations into a system can be reduced greatly.

The rest of the paper is organized as follows. First we will discuss some necessary background information about the general scheduling architecture in Section II. Then our solution will be discussed in Section III. After this our performance and evaluation will be discussed in Section IV. Related work will be mentioned in Section V. Finally we will conclude this paper with a summary of our results and give a list of future work in Section VI.

II. A General Scheduling Architecture

In a high performance computing site, many computers are connected and controlled by a single centralized manager. This manager is in charge of many aspects of this system. These functionalities include:

Scheduling Execution As jobs are submitted, it plans out on what machine these jobs execute, and how other jobs are affected by this. The computation of this execution schedule can require a lot of time to determine if the system is busy. The scheduling for jobs is the most important function that this system needs to consider.

Data Monitor The manager needs to be able to monitor where data is and if it is in use. This way it can determine if

a job that needs the data can start its execution, or if it needs to wait longer for staging in/out the data.

Data Movement The data, input and output, needs to be able to get moved around the system in a reliable way. The centralized manager takes control of this controlling machines where to send data for use.

Data Replication If multiple jobs need to access some data the central manager is also in charge of replicating the data across the system. With this it is able to have multiple jobs accessing the same data set.

Machine Monitor The manager doesn't only monitor the data, it also has to be aware of the other machines available in the system. This way it can make accurate decisions on where jobs should be scheduled in the event of system churn.

This system works well when not dealing with large data sets. However many scientific systems are beginning to (if not already) operate on sets large enough to be problematic. When the data sets become numerous the manager needs to monitor more pieces during the execution and scheduling of jobs. This can lead to an increase in the amount of computational power and storage capacity needed to maintain the database of status on all of the data sets. With this increase in computational power and storage sizes, the overall execution latency is likely to increase.

Furthermore, if the data sets become large in size then managing the data movement will bottleneck the system. In this situation, the manager will experience an increase in the latency to stage the data to the machine it needs to reside on for job execution, which will affect the entire system.

In an attempt to minimize the latencies associated with data movement and management, the goal of an ongoing project of ours is to develop a scheduler that takes data movement into account prior to making scheduling decisions. In such a system if the file movement is the major bottleneck of the system, this consideration can greatly affect the throughput of the system. This proposed scheduling system is called **Storage-Aware Job Scheduling** (SWAP) [1].

In the SWAP system, the scheduling is done with consideration to what files are already existing within the system. Therefore jobs that require popular data files already existing within the system will have the benefit to execute before jobs that require data to be migrated into the system. These systems are proposed to lower the required bandwidth in the system and allow for a better throughput.

We propose a method by which the data related workloads of the manager are reduced. It is worth noting that the proposed DIMM works perfect with SWAP, however, it works well with conventional scheduling schemes by reducing the the extra latencies incurred by managing data sets.

III. Design of DIMM

We will now introduce the major concepts of DIMM and how these concepts are capable of reducing the load on the centralized manager (a.k.a. computing farm).

In order to reduce the bottleneck from data management the key idea is to move some of the functionality away from the manager. However, these functions need to still be available to the units that need them. This way the stress these functions cause on the centralized manager will be removed and it can be used more efficiently for its remaining tasks.

There are two separate approaches we use to reduce this bottleneck on the manager. First, we intend to remove some responsibility from it by using a distributed hash table (DHT) [2]. Secondly, we intend to make some of its remaining functions easier by using a bloomfilter.

DHT A DHT is a distributed service system where the nodes are capable of doing a file lookup. In order to do this lookup a file name is hashed and then according to this output, there is a corresponding "home" location within the participating nodes. This allows all participating nodes to determine where a file will be located if it exists within the system.

Using a DHT allows us to remove the location responsibilities from the manager. The manager will no longer need to find and store a location for a file used during execution. Since every file is given a "home" location based on its hash value (which is determined by the filename), the manager can always expect the file to be at this location if it is in the system. This will allow the manager to decrease the amount of overhead metadata it will need to store within the system.

Each node in the system needs to be able to store the files for which it is the "'home"' node, but also it needs to be able to store files that it is using for which it is not the "'home"' location. For this reason each node has its disk divided into two partitions that are used for this purpose. We refer to these separate location as the home storage location and the replication storage location.

Data locality is the most important for data-intensive computing. Thus, it is preferable for a scheduler to distribute the jobs close to where the data locates. This requires the scheduler to keep track of or query another directory manager every time to figure out the location of data before scheduling, generating a lot of communication traffic, including between the scheduler to directory manager and between the directory manager and computing nodes (if there is a miss on local disk), resulting in a very complicated system. In a DHT, the location of data is determined by it's name. Thus no extra communication traffic is required to locate the files. However the problem with this is that consecutive files can be hashed to different locations and thus the locality can be lost. In order to prevent this, we propose an increased chunk size whereby multiple files will be hashed as one and thus reside on the same node. Another solution we propose is to have job migration among the locations where the files may reside.

In order to allow the above situation to work the manager still needs to be able to tell what jobs are in the system. If the manager cannot tell this, then it will be consistently requesting file movements that may not be necessary. This

will waste unnecessary bandwidth and downgrade the rate at which necessary files are moved into the system. In order to allow the manager quick and low cost access to the files located in the system it will use a bloomfilter structure as opposed to a database.

Counter-based Bloomfilter A bloomfilter is a data structure that is capable of telling what data from a set *A* also exist in a separate set *B* ($B \subset A$) [3]. 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* ($a \in A$), 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* ($a \in B$). For an example of a bloomfilter insertion see Figure 1(a).

There are two advantages to using a bloomfilter in this situation. The first one is that the storage for such a structure is low. This is because the array in use is only a bit array which makes the overhead low. 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 very 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 the manager.

However, there do exist problems with bloomfilters. One problem is if we attempt to update the list after a file is removed from the system. In this situation if the hash of file *a* and the hash of file *b* have an intersection then if one of these files is removed the other will also be ($h(a) \cap h(b) - h(a) \neq h(b)$). For an example look at how Figure 1(a) is affected by the removal of file *b* in Figure 1(b). If we check if file *a* is in the system we would receive a negative response

A simple bloomfilter can only accept entry into a set, not removal from it. The only way to remove items in a simple bloomfilter is to clean the entire set and then reinsert items, but this can be costly. In order to accommodate for this, an extended version of the bloomfilter is chosen, a counter-based bloomfilter[4]. 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. On the other hand, if an item is removed from the structure the integer value will be decreased and if the value has reached zero then that hash value no longer has a file existing in the system. 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, but it will take up more storage in the manager. However the increase in storage is negligible for a high performance machine as it is basically a large array of integers, which is validated in our simulation results in Section 4.

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 ($h(c) \subset h(a) \cup h(b)$). 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.

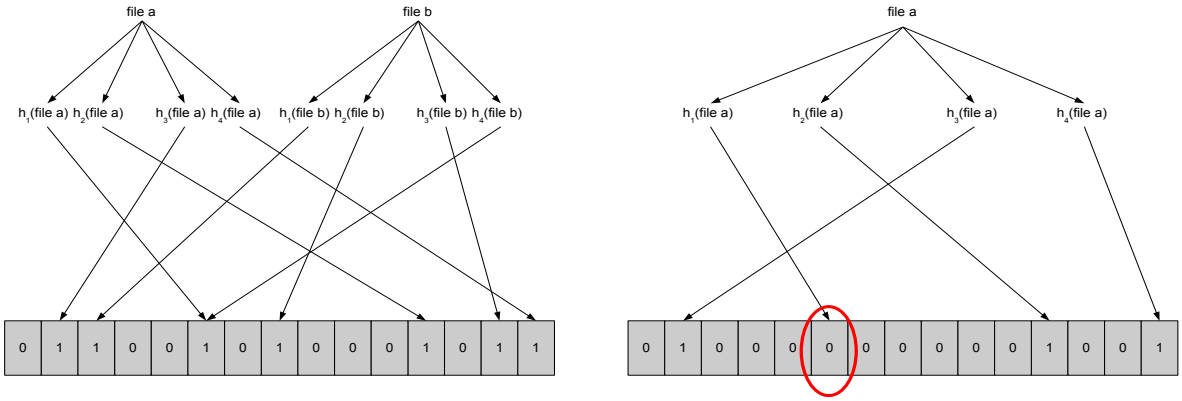
Given the size of the bloomfilter *k*, the number of hash functions *h*, and the number of items added into the set *n*, it is possible for us to compute the false positive probability.

$$\left(1 - e^{\frac{-hn}{k}}\right)^h \tag{1}$$

In order to reduce this probability there are two options; one could increase the size of the bloomfilter or one could increase the number of hash functions. However if the size of the set *A* is increasing throughout system operation, it is unrealistic to frequently halt the system in order to implement more hash functions or increase the size of the bloomfilter.

In order to control the probability of a false positive our bloomfilter also has a locality check included in it. In order to achieve this locality check, for every file that the bloomfilter says is in the system, we check a certain number, or distance, of neighboring files determined by alphabetic order of path. 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. An example of this situation would be a file in the system that is not neighboring to any other files in the system. The locality information on this file would say that since no neighbors of it are in the system, then odds are this file is not in the system either.

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 system to stage it onto its home location. Once the distributed hash table system, notices that the file is already staged in its home directory, it will not do anything. The manager will then be notified that the job is there and it can add (or correct in this case) it to the bloomfilter, at which point the job will be ready to be scheduled.



(a) Two files (a and b) are being inserted into a bloomfilter of length 15 using 4 hash functions.

(b) This figure is the result when the file *b* is removed from the Figure 1(a).

Fig. 1. An example of using counter-based bloomfilter.

IV. Performance Evaluation

A. The System Evaluation

In order to evaluate our system we have created a simulator that can be used to simulate the execution of a centralized scheduler without DIMM mechanisms included versus a scheduler with the DIMM mechanisms. In this experiment we designed over ten thousand jobs. Each of these jobs then have a set of input files associated with them. The number of input files was chosen from a normal distribution with a mean of 500 and a standard deviation of 22.[1] However the actual files associated with the job are chosen from a uniform distribution between zero and 100,000. Each of these jobs were then scheduled according to the SWAP mechanism [1] on a 400 node system. This scheduling is then used as the input for the evaluation of how our mechanism maintains the input files at each of the nodes.

For the evaluation of our mechanisms, we have set the number of files a node is capable of maintaining at 2,500. This number is obtained by assuming the local disk is 250GB, and each file is 100MB. This number is then divided into the relative ratios for the number of files in the home storage to the cache storage. In order to evaluate we compared four different data management schemes: `SWAP` with all storage being considered cache space, DIMM with only the home location being used to maintain files (`DIMM_h`), DIMM with the home location and the replication locations being used to maintain files (`DIMM_hr`), and DIMM used with a data oriented job migration technique (`JobMig`). In this technique each job will be migrated among all nodes that contain the data it requires. Thus if a file exists in the system the job will migrate to the file rather than the reverse. All of these techniques employ the least recently used (LRU) algorithm for their required replacement strategies. The last two variations are discussed in the next subsection.

As previously mentioned (Section II) the SWAP scheduling works such that jobs are scheduled on nodes in the system that achieves the smallest makespan (or maximum speedup). The algorithm considers a number of factors, most prominently the impact of the caching of existing files in the system. When a job is submitted, the scheduler considers the list of currently available nodes for each job, generating a scheduling execution estimate. During the estimation process, an estimate is generated for a reference node configuration that is intended to be representative of the average capability of a system node. Using this information, the job speedup ratio is calculated for each node. The speedup ratio allows the algorithm to position jobs on nodes that offer the greatest speedup improvement. Once we received the positioning of our jobs through this mechanism we started to compare our SWAP mechanism with and without our DIMM mechanisms. More details can be found at [1].

Figure 2 reports the comparison between these data management schemes where the x-axis represents the ratio of home to cache area storage capability and the y-axis is the percentage of data hits observed. A hit is defined as an input file existing at a node for a job without steps taken to bring it there. For the first step of the evaluation we can see that the number of hits incurred in the SWAP scheduling performed better than the DIMM mechanism when we consider keeping data only in the home storage of the node. This is because with the `DIMM_h` we are limiting the data that is capable of being stored at each node, whereas in SWAP data is capable of being stored at any and every node.

Figure 3 lists the comparison between these data management schemes where the x-axis represents the ratio of home to replication area storage capability and the y-axis is the percentage of data migrations from archival storage observed.

[1]These numbers are chosen based on our oral conversation with the people working on the STAR project.

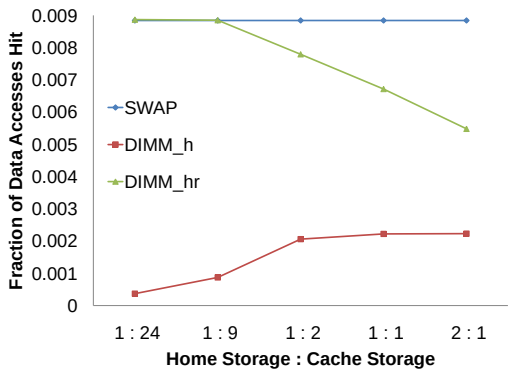


Fig. 2. The percentage of input files that hits for different home to cache ratios, represented X (home) : Y (cache) for each algorithm.

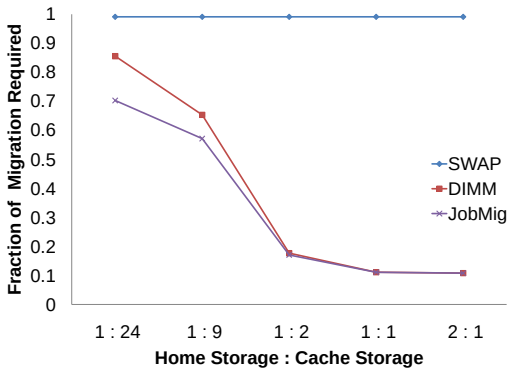


Fig. 3. The percentage of input files where migration is necessary for different home to cache ratios per each algorithm.

One interesting thing worth noting is that the number of input files that must be migrated is decreased by using the `DIMM_h`. This is due to that fact that for every file that must be brought to a node, DIMM works by bringing a copy of that file to the home location prior to sending it to the node it is going to be used at. Therefore in future accesses if the file is still available in the home node, no migration is necessary and we can achieve a 15% reduction in migrations necessary, as the increase of the home portion in local disk, the fraction of migration is reduced significantly.

B. Bloomfilter Size vs. Database Size

One of the key features of a bloomfilter is the reduction in storage size per item existing in it. However a problem exists because the total size of the bloomfilter needs to be a multiple of the total number of files possible to exist in it the system, in order to have less false positives. Both of these conditions are opposite to a database which contains a large storage size per item existing in the system and changes size depending on the total number of items in the system. In order to evaluate our system fully we need to make sure that the cost of implementing a bloomfilter is less than that of maintaining the database.

In our implementation the bloomfilter per item size is the size of a small integer. We can safely allow this to be a byte and therefore allow a single hash value to be set a total of 255 times. Whereas in a database system, we need to monitor which machine is currently holding every data item in the system. In order to do this we will need to maintain the IP address (or node id) of the machine that the file is residing on and which file that happens to be. This means that 4 bytes will be needed to store the IP address of a file and an additional $\lg(n)$ where n is the number of files in the system. This additional $\lg(n)$ is the least number of bytes possible to differentiate n possible files.

Figure 4 reports the comparison of storage requirements between a centralized database and several configurations of bloomfilter. Before 500 files exist in the system, a bloomfilter with 5 times the storage as the number of files and a bloomfilter with 10 times the storage as the number of files are less storage than a centralized database. Furthermore, it's not shown, but after 65,536 files the bloomfilter with 20 times the storage will cost less than the database. This is due to the size adjustments for the file identifications.

This shows that as the number of files increase in the system (due to storage of older data and collection of new data), the bloomfilter approach results in less storage than the minimal way to store in a database format.

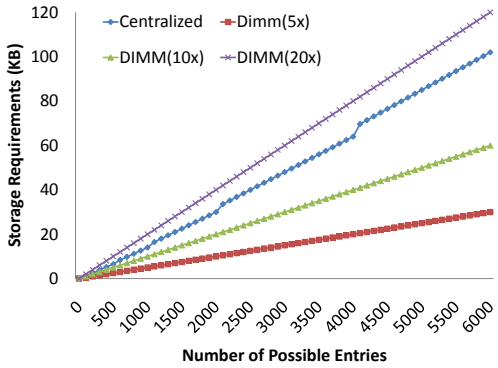


Fig. 4. The total size of a database and bloomfilters (with different size constants) relative to the number of files in the system.

C. 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.

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 percentage of there not being a false positive. We would also check the extensive list and see if there was a real false positive or was there a check on a file existing in the bloomfilter and what was the result.

The normal distribution selecting the files ran with a constant mean value of 500,000. The variance was tested with different parameters of 250, 1,000, 5,000, 10,000, 25,000, 50,000, 75,000, 100,000, 125,000, 250,000, 375,000, 500,000. By changing the variance we can influence the degree of locality a file must have with the rest of the system. Therefore a low variance implies that files are very heavily related and exhibit a great amount of locality, whereas a high variance implies the files do not exhibit a strong locality.

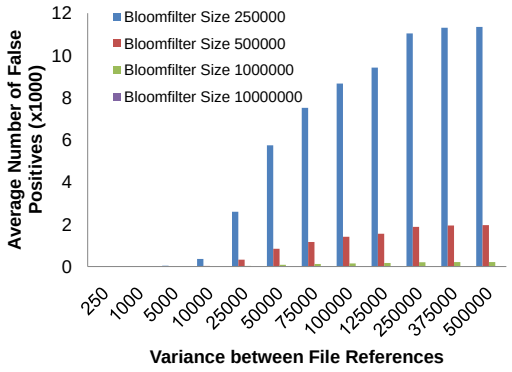


Fig. 5. The average percentage of false positives relative to the number of entries.

Figure 5 summarizes our results as to the number of false positives generated in each scenario. There are several observations that can be implied from this figure. First this supports the equation for the probability of false positives (as the size of the bloomfilter is increased the number of false positives is decreased). Second 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 (see Figure 6 which displays the average number of entries added to each bloomfilter for different variances). These additional files inserted increase

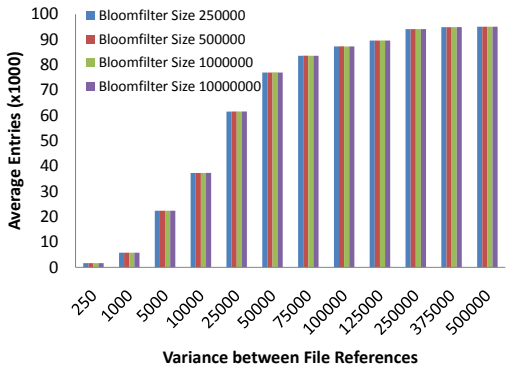


Fig. 6. The average number of entries for a given variance for varying bloomfilter sizes.

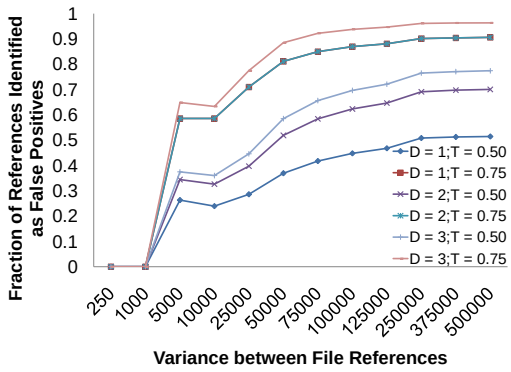
the probability of false positives in the bloomfilter.

Now that a baseline for the number of false positives generated has been created we are able to analyze the locality check for its impact on the performance. Figure 7(a) illustrates the percentage of the false positives that were correctly identified whereas Figure 7(b) displays the percentage of entries incorrectly identified as false positives. Both of these figures are for different combination of distances (D) and thresholds (T). From Figure 7(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 25-45% reduction depending on the variance among the data.

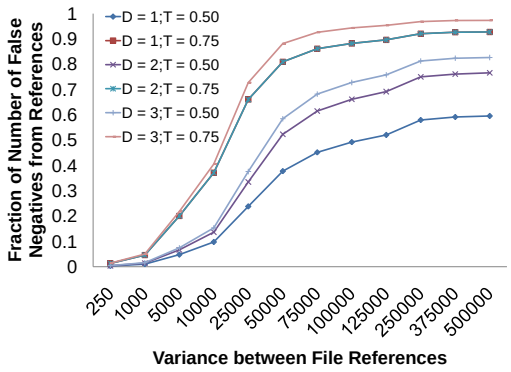
The above conclusions are only for the bloomfilter of size 250,000, but similar results have been obtained from the other bloomfilter sizes (they have been omitted due to page restrictions). However one thing worth noticing is that as the bloomfilter size increases the performance of the simplest locality check performs better. It consistently performs the same at the low variances, but as the size increases it performs better during the higher variances.

Another conclusion we can make from this data is that in order to make a more accurate detection of false positives either the distance checked or the threshold compared against need to be increased. This is expected as this is how the locality check was designed to perform.

Up until now we have only focused on the performance of this technique to catch the false positives. Looking at Figure 7(b) we can see the performance of this technique at inducing false negatives. First focusing on the variance 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. 7. The effects of locality checking in a bloomfilter of size 250,000.

of [250-10,000], we can see that the number of false negatives induced in this range is maxed at approximately 10% of the total number of references to data for a distance of 1 and a threshold of 50%. If this is compared to the maximum percentage of false positives identified in this range (25%) then we can see that by injecting more units of small traffic into the network we can account for the total amount of overhead in large network traffic when scheduling.

If we change our focus to the high end of the variance range ([100,000-500,000]) we can see that the number of false negatives is 50-60% of the total number of references. Comparing this to false positive identification we are identifying 45-50% of the total false positives. At this point the performance gain of being able to identify false positives may not outweigh the amount of false negative traffic created in the network. A situation where this could be the case is when the data sets take relatively small time to transfer or when the network is relatively slow and false negative messages take increasingly longer time.

Once again these evaluations are only based on the bloomfilter of size 250,000, but the other bloomfilter sizes behave similarly (again results have been omitted due to page length). 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.

This leads us to conclude that in order to take advantage of this approach several parameters must be known about the systems performance. First, the variance at which the files are referenced, second the time taken to transfer files into the system, and third the time taken to check if a file is in the system. Once these things are known in advance, then it will be possible for the administrator of the system to determine the best way to utilize false positive detection while not degrading the system with false negatives, i.e., choose the right parameters for bloomfilters.

D. Summary

To this end we have showed that DIMM is capable of reducing the necessary number of file migrations into the system by anywhere from 15% to up to nearly 90% depending on the setup of the system.

We have also shown that although the initial costs of using a bloomfilter is greater than that of using a database, this cost is offset after a small number of files. After the number of files in the system has been surpassed, the cost of maintaining a database actually becomes greater than that of a bloomfilter.

We have further shown that our method of detecting false positives is an efficient method if the induction of false negatives can be tolerated by the system. This method works better for a lower variance among the files where it can catch 25% of the false positives and approximately 10% false negatives, but it is still capable of use when the variance among the files becomes quite large. The parameters used in this locality check are dependent on the characteristics of the system, but are configurable in order to reach the desired degree of data checking (keeping in mind that a larger distance and a larger percentage threshold are more strict at identifying items that exist in the data set).

V. Related Work

This paper is built on top of many previous efforts, including bloomfilter, distributed hash table, and job scheduling. The former has already been discussed in the paperShankar *et al.* [5] propose additions to Condor for data caching and work flow data scheduling. However they are looking at the

data movement among a work flow and not among the entire system as we are, also the metadata contained within their system is part of a large database, whereas ours is distributed among the system in DHT.

Bright and Maier [6] propose a hybrid scheduling algorithm that takes into account data locations in the scheduling of data workflows. It does not focus on the storage of the necessary information to deliver their approach nor on how to assign the initial data storage in the workflow.

Work is also being done for data movement management. In GridFTP [7] a method of data movement is provided that is a built on top of the standard File Transfer Protocol (FTP). This system does not have a relationship with job scheduling. In HPSS [8] the system works well with moving data in a high performance system, but a problem exists with the increase in the amount of time to move data from low level storage locations to a computational node. This problem is especially apparent when the data is already located at another computational node. Also the metadata management in HPSS needs improvement [9].

Investigation into distributed job scheduling is also taking place [10]. This paper focuses on the distributed nature of scheduling whereas we are focusing on the distribution of the metadata available to the scheduler. In recent work done by Zhang *et al.* they research methods for job recovery within a scheduled environment [11]. This work complements to ours well, as it allows for jobs to be recovered in the event of failures, but it does not address the management of metadata within these scheduling environments.

There are also proposals for data management within scheduling environments. Ailamaki *et al.* propose a method of data management using databases [12]. This is orthogonal to our work as we are trying to remove the storage costs of maintaining a database on the centralized manager.

A similar work to ours is the Google BigTable [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. Having a comprehensive comparison between DIMM and Google's approach in a large scale is an interesting direction.

Perhaps the most similar work to our own is Chervnak *et al.* Giggle system [14]. This system is also concerned with controlling the replicas among a system in a distributed fashion. However, a decentralized state is not native to their system, and depends on user intervention. Another work similar to ours is the L-Store File System [15]. This work however concentrates on the files at the block level.This approach does not allow for nodes that contain these blocks to access without the cost of having to retrieve the other blocks as well.

VI. Conclusions and Future Work

Job scheduling within a high performance environment is still a very important research area, and as the knowledge of system status grow more things can be taken into account for the job scheduling. However as more things are taken into account the amount of metadata used in the scheduling increases. Therefore we proposed a method by which the typical centralized manager in a high performance environment can be turned into a more decentralized one through the use of distributed hash table. We also proposed a bloomfilter-based method by which the centralized manager is capable of getting the amount of metadata it needs to maintain decreased to allow for better execution.

Following this work, we intend to further investigate the relationship between data movement and scheduling. We are implementing DIMM using openDHT. We plan to integrate DIMM into a real product scheduler in a high performance computing environment, e.g., the STAR scheduler [16].

References

[1] 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.

[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] 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.

[4] 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.

[5] S. Shankar and D. J. DeWitt, "Data driven workflow planning in cluster management systems," in *HPDC 2007*. New York, NY, USA: ACM Press, 2007, pp. 127–136.

[6] L. Bright and D. Maier, "Efficient scheduling and execution of scientific workflow tasks," in *SSDBM'2005*. Berkeley, CA, US: Lawrence Berkeley Laboratory, 2005, pp. 65–74.

[7] B. Allcock *et al.*, "Data management and transfer in high-performance computational grid environments," *Parallel Comput.*, vol. 28, no. 5, pp. 749–771, 2002.

[8] R. A. Coyne, H. Hulen, and R. Watson, "The high performance storage system," in *Supercomputing '93*. New York, NY, USA: ACM, 1993, pp. 83–92.

[9] R. W. Watson, "High performance storage system scalability: Architecture, implementation and experience," in *MSST '05*. Washington, DC, USA: IEEE Computer Society, 2005, pp. 145–159.

[10] V. Subramani, R. Kettimuthu, S. Srinivasan, and P. Sadayappan, "Distributed job scheduling on computational grids using multiple simultaneous requests," in *HPDC 2002*. Washington, DC, USA: IEEE Computer Society, 2002, p. 359.

[11] Z. Zhang *et al.*, "Optimizing center performance through coordinated data staging, scheduling and recovery," in *Supercomputing '07*, 2007.

[12] A. Ailamaki, Y. E. Ioannidis, and M. Livny, "Scientific workflow management by database management," in *SSDBM '98*. Washington, DC, USA: IEEE Computer Society, 1998, pp. 190–199.

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

[14] A. Chervenak *et al.*, "Giggle: a framework for constructing scalable replica location services," in *Supercomputing '02*. Los Alamitos, CA, USA: IEEE Computer Society Press, 2002, pp. 1–17.

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

[16] [Online]. Available: http://www.star.bnl.gov/