Handling Heterogeneity in Shared-Disk File Systems

Changxun Wu and Randal Burns
Department of Computer Science
Johns Hopkins University
{wu,randal}@cs.jhu.edu

Abstract

We develop and evaluate a system for load management in shared-disk file systems built on clusters of heterogeneous computers. The system generalizes load balancing and server provisioning. It balances file metadata workload by moving file sets among cluster server nodes. It also responds to changing server resources that arise from failure and recovery and dynamically adding or removing servers. The system is adaptive and self-managing. It operates without any *a-priori* knowledge of workload properties or the capabilities of the servers. Rather, it continuously tunes load placement using a technique called adaptive, non-uniform (ANU) randomization. ANU randomization realizes the scalability and metadata reduction benefits of hash-based, randomized placement techniques. It also avoids hashing's drawbacks: load skew, inability to cope with heterogeneity, and lack of tunability. Simulation results show that our load-management algorithm performs comparably to a prescient algorithm.

1 Introduction

Shared-disk file systems [26, 30, 34] provide good solutions for managing large sets of data among a group of computers, because they offer fast data access, improved data availability, and increased system scalability. However, shared-disk file systems built on heterogeneous servers are vulnerable to load imbalance, which results in performance degradation. We develop a load placement and resource management system to handle server and workload heterogeneity in shared-disk file systems.

As file systems evolve from running on a single computer to a cluster, the problem of efficient and highly-available data management becomes crucial. Shared-disk file systems solve this problem through a decentralized architecture and the use of network-attached block storage devices [26, 30, 34]. Shared access to network-attached disks makes storage devices accessible from all cluster nodes. Many file systems employ storage area networks (SANs) [26], such as Fibre Channel or iSCSI, for shared-disk connectivity. SANs provide a network implementation of block storage protocols.

The problem of load-placement in a shared-disk file system differs from what is considered "dynamic load balancing." Dynamic load balancing frequently refers to routing individual requests so that all servers experience equal load [28, 42]. Since every request can be routed to any node, servers share all workload at all times. In contrast, a shared-disk file system consists of sets of files that are assigned to cluster server nodes. At runtime, a single server owns some sets of files and serves all metadata requests for files in those sets. The system uses shared-disk to move file sets from server to server in response to load imbalance or the set of servers changing. All servers can access the disk image of any file set, which facilitates error recovery if a server fails. Thus, the load-placement problem resembles the data allocation problem in shared-nothing databases [19].

Today, cluster file systems are shifting from expensive dedicated parallel machines to commodity hardware. The trend toward low-cost commodity hardware and open-source systems facilitates the construction of computing clusters on heterogeneous hardware rather than the proprietary parallel supercomputers [5, 6]. As cluster file systems have previously been built with the assumption of homogeneous hardware, the load placement policies they employ are insensitive to heterogeneity [30, 34]. Current methods do not translate to commodity configurations.

Our system addresses the problem of load placement for shared-disk file systems built on heterogeneous servers. Based on a technique called adaptive, non-uniform (ANU) randomization, our system handles not only the heterogeneity of cluster hardware, but also heterogeneity in user workload. ANU randomization is derived from the SIEVE

SC'03 November 15-21, 2003, Phoenix, Arizona, USA

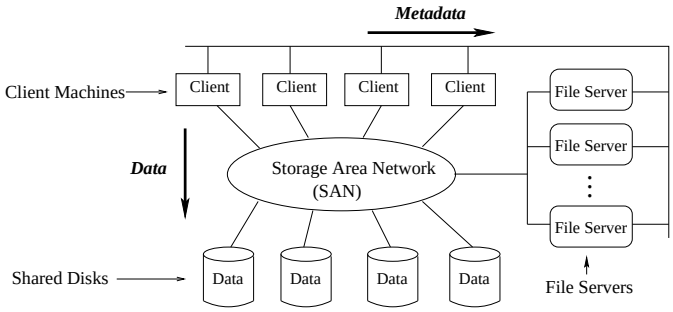


Figure 1: Shared-disk file system architecture.

adaptive hashing strategy of Brinkmann *et al* [7]. ANU randomization makes randomized load-placement tunable by adding a layer of abstraction between servers and file sets. The unique name of a file set is hashed to a unit interval. Servers are assigned to regions of the unit interval. Our system scales the server regions to add or remove load. We manage the mapping of file sets and servers so that configuration changes result in a minimum amount of data movement. Our system has advantages in managing heterogeneous workloads:

- temporal heterogeneity – changing load placement in response to workload shifts;
- spatial heterogeneity – coping with file sets that contain different workloads.

It has advantages when dealing with server hardware:

- server heterogeneity – balancing load over servers with different performance characteristics;
- self-configuring – no knowledge of hardware capabilities is needed;
- future adaptability – upgrading hardware while the system is on-line and taking full advantage of faster hardware.

It also has benefits during failure and recovery:

- self-organizing – placing, moving, and balancing workload without human intervention;
- cache preserving – maintaining load locality by moving the minimum amount of data.

The best properties of our load-placement technique are (1) the ability to tune the system without knowledge of application behavior or of server capabilities and (2) the preservation of load locality during failure and recovery. These are also the ways our solution differs from existing heterogeneous load placement schemes [4, 16, 32, 36, 41]. When considered together, these minimize the cost of reorganizing the system to workload or configuration changes. Servers are dynamically interchangeable and reconfigurable. For example, the same server might be deployed in different clusters at different times during the same day or hour, as needed in enterprise hosting.

2 System Architecture

A brief review of a shared-disk file system, IBM Storage Tank [8, 20], and its workload characteristics helps to motivate our solution. Load placement using ANU randomization was designed for Storage Tank. However, our algorithms for cluster resource management are suitable for any file system that partitions workload [2, 13].

The Storage Tank shared-disk file system (Figure 1) consists of a SAN, shared disks, file servers, and client machines. Storage Tank distinguishes between meta-data and data, which are stored and accessed separately. File servers store, serve, and write file system metadata, grant file/data locks, and detect and recover failed clients. Metadata are stored on shared disks accessible to all servers. The data (associated with metadata) are stored on shared disks available to all computers, including clients. In a typical file access, the client first obtains metadata and locks for a file from the Storage Tank servers and then fetches data by sending I/O requests directly to shared disks on the SAN. To better manage file system metadata, files are partitioned into file sets, each of which is assigned to an individual file server. In this way, file servers share nothing at runtime and Storage Tank does not require expensive server-to-server operations, such as memory sharing or distributed commit. A file set is a subtree of the global file system namespace. In Storage Tank, file sets are indivisible and, therefore, the unit of workload assignment and load movement.

This architecture separates metadata workload from data workload and targets them to file servers and shared-disk respectively. File servers are loaded with the single class of metadata operations – small reads and writes. The file server workload does not include sequential or large file I/O, which goes to shared disks.

The simplicity of the server workload makes it reasonable to use a simple performance metric to evaluate server performance. We use request latency, because all requests are short and service time variance is low. In contrast, throughput would be a more suitable metric for load-balancing large, sequential I/Os. Based on the observed request latencies, our system adaptively moves file sets among servers to balance the metadata workload on servers.

Although our system does not address all load balancing issues in shared-disk file systems, keeping file metadata servers balanced is an important task. In shared-disk file systems, file servers are recoverable resources. Imbalance in file metadata servers adversely affects overall system performance, because clients acquire metadata prior to data. Clients blocked on metadata may leave the high bandwidth SAN underutilized.

3 Related Work

Many dynamic load management techniques for parallel systems are designed for homogeneous clusters [10, 37–40]. Workload is transferred from heavily loaded servers to lightly loaded ones.

Another family of techniques [36, 41, 42] takes into account server heterogeneity but requires all servers to periodically broadcast load and available capacity. These techniques assume knowledge of the capacity of any given server. Utopia [41] uses pre-computed knowledge of non-uniform server capabilities and makes load adjustment decisions based on the available CPU cycles, free memory size, and disk bandwidth updates of each server. Zhu *et al* [42] use knowledge of server capacity and employ a metric that combines available CPU cycles and disk capacity of each server to select a node for processing an incoming request.

Many systems, especially peer-to-peer systems [27, 33], rely on randomization for load balancing. There are challenges in balancing load dynamically in global scale networks. Message exchanges may have to travel across the Internet, which makes it difficult to move load. Also, herds of tasks from many peer nodes may simultaneously move together to a node that previously had available capacity [22]. Therefore, peer-to-peer systems do not actively balance load. They use simple randomized load placement, which balances load in practice while avoiding message exchanges between the peer nodes [21, 23]. Although simple randomization works well in certain environments, our experiments indicate that it cannot support extreme workload and server heterogeneity.

I/O systems use striping to distribute a large I/O request across many disks in a load-balanced fashion [11]. This approach applies when jobs are divisible and large, and, thus, are not suitable for metadata operations. Striping is frequently used in multimedia servers [31].

Many systems redirect requests within a cluster to dynamically balance load. This is a popular form of load balancing suitable for systems in which any server can handle any request. Examples include DNS rotation [9, 15] and request routing in Web servers [3].

There is a long history of load balancing through process migration [4, 12, 18, 25], in which active processes are transferred among computers. Process migration reorganizes the assignment of long running jobs among a cluster of computers. These techniques do not apply to Web serving and file serving, which consist of relatively short tasks.

Recently, there has been research in the area of functional decomposition [1, 2]. Instead of dividing workload among cluster servers, the system places different functions at different servers. For example, splitting an interactive file metadata workload from a throughput oriented large-file I/O workload [2].

Some peer-to-peer systems [27, 33] use hashing for distributed lookup services. Similar to our approach, such systems also map values (keys and nodes in these cases) to an artificial one-dimensional space and try to match them by their offsets within the artificial one-dimensional space. However, these systems are able to handle distributed lookup only and rely on the underlying hash functions to provide load balancing. As the underlying hash functions in these systems only guarantee the uniform distribution of the hashed values, these systems are not sensitive to object workload heterogeneity and cannot maintain load balancing in the situation where objects have heterogeneous access costs and frequencies.

4 Algorithm

We use a technique called adaptive, non-uniform (ANU) randomization to place and balance load. ANU randomization is based on the SIEVE adaptive hashing technique described by Brinkmann *et al* [7]. The technique employs a pseudo-random hash function to map file sets to offsets in a unit interval (Figure 2). The unit interval is partitioned into multiple sub-regions of the same size. We assign to each server one or more sub-regions. A server completely occupies all but one sub-region, which may be partially occupied. In the following discussion, we call these sub-regions partitions and we call the segments within the unit interval occupied by a server its mapped region. Each server is

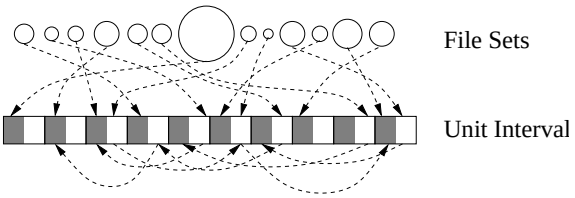


Figure 2: Servers are assigned sub-regions of the unit interval. File sets of different size, representing different workloads, are hashed into this interval. File sets not mapped to servers by the first hash are re-hashed until assigned.

then assigned the file sets for which the hash of the unique name of that file set lies within its mapped region. In the target architecture, the unique name is assigned by an administrator. In other systems, it might be the pathname in a global namespace or some fingerprint of the data contents. Our technique balances load by changing the sizes of the server mapped regions based on a simple performance metrics – observed latency. By repartitioning the interval and re-mapping the partitions to servers, this technique copes gracefully with hardware changes, such as adding and removing servers.

The flexibility of non-uniform randomization comes from an extra level of abstraction in the server to workload mapping. Non-uniform randomization employs a unit interval to which it maps both workload (file sets) and servers. The ability to change the mapping of the servers allows us to tune the system in a fashion that is not possible when mapping file sets directly to servers (simple randomization). File sets are placed into the unit interval using a pseudo-random hash function. In figure 2, we show file sets of different size to indicate different amounts of workload (not data). For a system with k servers, we divide the unit interval into $2^{\lceil \lg k \rceil + 1}$ partitions of equal size. Servers are mapped to portions of one or more of these partitions and a server handles all file sets that fall into the shaded region that it occupies. The system does not assign servers to all portions of the unit interval. Instead, the system assigns servers to half of the unit interval, *i.e.*, the total of all portions of partitions assigned sums to half of the total unit interval. Half occupancy is maintained as an invariant to make sure that there is always an assignment of partitions satisfying the needs of all servers as well as an unassigned partition available to a recovered server. File sets that hash into un-mapped regions are re-hashed until assigned to a mapped region. Re-hashing is performed using the next hash function among an agreed upon family of hash functions. File sets that have not been assigned in a fixed number of rounds are hashed to a server (rather than the unit interval). This bounds the number of rounds and does not introduce significant skew into the system, because it occurs with low probability, 2^{-r} for r rounds. On average, the system requires two probes to assign a file set, but we note that a hash probe does no I/O when determining where a file set is served. Successive hash probes incur negligible costs.

The system manages server and workload heterogeneity by changing the sizes of the mapped regions. Each server monitors its performance and produces a performance metric over a chosen time interval. We use latency as the performance metric – a natural choice as the metadata workload consists of little data and short-lived transactions. At the end of each interval, each server computes its latency in the past interval and reports it to an elected delegate server. The delegate server examines all latencies and comes up with an "average" value for the whole system. The delegate scales down the mapped regions for servers above the average and scales up the mapped regions for servers below the average. The description of the algorithm at this point is conceptual. There are many parameters that we vary and several heuristics that we implement to tune the algorithm, but all variants of the algorithm conform to this framework.

An appropriate average is difficult to determine. For simplicity's sake, we are using a weighted average of the current latencies. However, we also ran experiments using a median. Results verify that our system is robust to the choice of an average and operates well using different techniques. We note that in a perfectly balanced system, the mean, median, and mode of server latency are identical.

We make the load update algorithm stateless in order to gracefully handle failures of the delegate server. The delegate determines the new load configuration based solely on latencies reported from the current configuration. If the delegate fails, the next elected delegate runs the same protocol with the same information.

The system reorganizes load to conform to configuration changes made by the delegate. The delegate distributes a new mapping of servers to the unit interval to all servers. This is the only replicated state needed by our algorithm. Upon receiving updates to its mapped regions, a server identifies "shed" file sets – file sets that it served in the previous configuration that are served by another server in the current configuration. The shedding server flushes its cache with respect to shed file sets to create a consistent disk image. Then, the server hashes each shed file set to locate a new

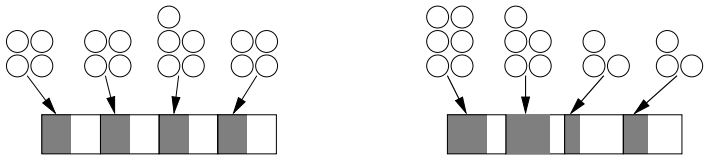


Figure 3: Dealing with server heterogeneity. Initial configuration (left) and after reorganization (right).

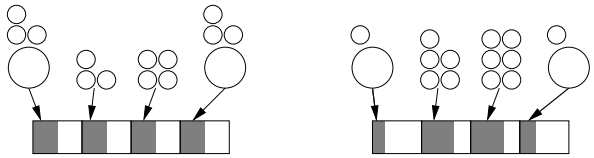


Figure 4: Dealing with non-uniform workload. Initial configuration (left) and after reorganization (right).

server and notifies the new server that it is gaining workload. If necessary, the new server initializes the file set as required by the file system.

Scaling the server mapped regions handles server heterogeneity. Figure 3 shows four servers. Assume the first two servers (from the left) to be twice as fast as the third and fourth, and assume all file sets hold identical amounts of workload. In a balanced system, servers one and two have twice the load of servers three and four. Because the system has no knowledge of server capabilities, the initial configuration places the same number of file sets at each server, minus hashing variance and discrete effects. During reorganization, servers one and two increase their mapped regions to gain load and servers three and four decrease their mapped regions to shed load.

Scaling the servers deals gracefully with variance in hashing. In Figure 3, we might expect servers three and four to have mapped regions of the same size, because they are equally fast. However, hashing variance placed more load at three than four initially. Three reduced its mapped region by a larger factor to shed more load. Interestingly, server scaling results in better load balance than simple randomization even when all servers and all file sets are homogeneous.

Scaling the server mapped regions also balances non-uniform workloads. Figure 4 depicts four servers, which we assume to be uniform, serving non-uniform file sets, in which the size of the file set indicates the amount of workload. In the initial configuration, all servers receive an equal (less variance and discrete effects) number of file sets. Servers with large file sets are overloaded and decrease their mapped regions in the next reorganization. Other servers take on workload, because their mapped regions are increased proportionally in order to maintain the half-occupancy invariant. Scaling mapped regions balances non-uniformity in workload on uniform servers.

Non-uniform randomization performs well when servers fail or recover or when servers are installed or removed, maintaining good load balance and preserving load locality. So far we have discussed load placement in systems with a constant number of servers and therefore a constant number of partitions in the unit interval. When a server fails, the load it can take effectively goes to zero. Other servers increase their mapped regions to preserve the half-occupancy invariant of the unit interval. Only the file set(s) that were served previously by the failed server are re-hashed to locate a new server. When a server recovers or is added, it is assigned to a free partition and all other servers are scaled back to preserve the half-occupancy invariant. The framework treats commissioning (installing) or decommissioning servers the same as a recovery or failure respectively. If the added server increases k (the number of servers) such that there are fewer than $2^{\lceil \lg k \rceil + 1}$ partitions, the algorithm re-partitions the unit interval. The combination of the number of partitions and the half-occupancy invariant ensures that there is always an available partition into which a recovered or added server may be placed. We present an example in Figure 5. The systems starts with 4 servers in 8 partitions with a highly skewed workload. The first server occupies almost all of four partitions, while the remaining three servers have little pieces in the remaining four partitions. Adding a fifth server re-partitions the unit interval, creating new partitions for more servers to be added. As long as each server occupies at most one single partial partition, free partitions are always available. We comment that a server may occupy multiple partitions that are non-contiguous. Further partitioning the unit interval does not move any existing load and does not change the hash functions that address load, as does linear hashing [17]. During failure and recovery, our system does not re-hash all the file sets. Instead, it moves the minimum amount of workload possible by scaling the mapped regions of alive servers from last configuration. Therefore, load locality is maintained and caches of file sets are preserved.

Figure 5: Partitioning the unit interval when adding a server.

Load balance in this scheme is within a small constant factor of optimal. For n servers and m file sets, each server contains load $\lceil \frac{m}{n} + 1 \rceil$ with high probability. This result depends on several factors including a multiple choice heuristic that we have not described [7]. This variance is as small as any known bound for randomized placement and compares favorably to simple randomization in which load is bounded by $\lceil \frac{m}{n} + \Theta(\frac{\lg n}{\lg \lg n}) + 1 \rceil$.

5 Discussion

In addition to balancing load, the presented algorithm has several properties that make it attractive for parallel and distributed systems. These include scalability, efficient addressing, and load-preservation. The algorithm has much in common with the distributed directory schemes used in peer-to-peer systems [27, 33], in that both systems use hashing to map requests to an abstract address space. In contrast to directory schemes, ANU randomization adds the ability to tune the mapping of servers to the address space so that load placement is tunable.

Hashing offers a scalable, lightweight addressing mechanism that our system uses to locate file sets within a cluster. Hashing is deterministic. The hash function used to place load during a configuration change is also used to locate a file set at runtime and route requests to the appropriate server. Addressing and locating load through hashing and re-hashing requires no I/O.

The shared state in ANU randomization scales with the number of servers, rather than the number of file sets. The algorithm replicates the mapped region of each server. A server has no concept of the number of or names of the file sets managed at other sites. When a server sees an unknown unique name, it hashes it and routes the request to the appropriate server. These scaling properties are particularly valuable when there are many more file sets than servers.

The scalability and addressing features of ANU randomization compare favorably to bin-packing load balancing schemes [36, 41] in which any workload unit can be placed onto any server. To locate file sets, each computer must maintain a table that maps file sets to a particular server. This can represent a large amount of state to maintain and replicate when dealing with large numbers of file sets, increasing the time to reconfigure servers when moving load or recovering from a failure.

Experimental results (Section 7) show that for balancing load, ANU randomization compares favorably to bin-packing schemes. During re-partitioning, the algorithms move a minimum number of file sets, preserving the cache contents of servers across reconfiguration and reducing restart recovery times. To make this point, we compare our algorithm to a prescient bin-packing algorithm that has complete knowledge of future workload.

Although ANU randomization has been developed for cluster computing, small changes make it suitable for distributed environments, such as peer-to-peer and global scale systems. Currently, load movement and addressing are completely decentralized. The only centralized step is the collection of latency information and the redistribution of the server to unit interval mapping. For future work, we are modifying the algorithm, replacing centralized re-scaling of server mapped regions with pair-wise interactions in which servers scale their mapped regions in peer-to-peer exchanges. Completely decentralizing the algorithm also requires a technique to modify a server's mapped region without replicating this information to all other servers.

6 Over-tuning

In the early-stages of this research, our algorithms displayed an "over-tuning" side-effect. Preliminary simulation results showed that load placement did not converge. The system continued to tune load, moving file sets from server to server, without improving load balance. Several factors lead to this effect. First, file set workloads are indivisible and cannot be balanced exactly. Consider two servers managing three file sets with equal workload. The "balanced" configuration has one server with twice the workload of the other. Moving a file set to the server with lower load merely changes the location of the imbalance, rather than resolving it. Also, server heterogeneity leads to over-tuning. Consider two servers managing two file sets, in which one server is ten times more powerful than the other. The "balanced" configuration places both file sets on the more powerful server. But, the less powerful server has no workload, and, therefore, might continue to attempt to acquire load. On acquiring load, latency on the lower-powered

server spikes and the load is moved back to the higher powered server. Both examples are cyclic, tuning themselves into and out of "balance" repeatedly.

To solve this problem, we design and evaluate three techniques: *thresholding*, *top-off* tuning, and *divergent* tuning. These are combined and integrated into the ANU algorithm. Experimental results (Section 7) show that these techniques eliminate over-tuning and decompose their contributions.

Thresholding permits a certain degree of imbalance to exist among the servers. Using this policy, the delegate server updates only those servers whose latency lies outside the range $[(1-k)\mathcal{L}, (1+k)\mathcal{L}]$, where $\mathcal{L}$ is the average latency and k is a tuning parameter. The proper choice of k depends on workload heterogeneity, on the number of file sets, and on the combination of thresholding with other over-tuning heuristics. Thresholding is necessary to prevent over-tuning, because this policy allows the algorithm to decide that it has reached a "balanced" state even though the system is not in perfect balance. Fairly large values of k are necessary to cope with workload heterogeneity in our experiments and we choose $k = 0.5$ based on experience.

Top-off tuning restricts the algorithm to decreasing the mapped regions of overloaded servers. The algorithm no longer explicitly increases the mapped regions of underloaded servers. Underloaded servers do increase their load implicitly because (1) overloaded servers shed workload and (2) when reducing one server mapped region all other server mapped regions are increased to preserve the half-occupancy invariant. We call this top-off tuning, because it looks for latency peaks (above average latency) and cuts the tops off these peaks. Top-off tuning can be thought of as an extension to thresholding in which the threshold interval is $[0, (1+k)\mathcal{L}]$.

Top-off tuning manages extreme server heterogeneity by allowing some servers to sit idle. In our early experiments, we observed workload thrashing; the slowest server would go from zero latency to high latency based on acquiring and shedding a single file set. The top-off policy allows the most powerful servers to achieve good load balance, while ignoring the weakest servers.

Divergent tuning tackles over-tuning due to overshooting the latency target. A configuration update immediately changes the workload distribution in the system, but it does not immediately change the latency. Latency converges to its equilibrium value more slowly due to tasks left in the server queue from the previous configuration. These "memento" tasks artificially increase the latency of subsequent tasks, particularly when workload is being decreased. Without divergent tuning, the system can tune down a server and then tune the server down again before the first tuning reaches an equilibrium. Successive reductions result in much lower latencies than desired.

To avoid overshoot, divergent tuning scales only the mapped regions for servers that are above the average latency and increasing or below the average latency and decreasing. This policy makes ANU randomization less aggressive. In exchange, it prevents cyclic overshoot. Divergent tuning does give up the stateless property of the ANU algorithm, because scaling server mapped regions in the next configuration depends on the current and previous configurations. The ANU algorithm still works when the delegate server crashes, in which case divergence cannot be evaluated and the ANU algorithm ignores this policy.

7 Experimental Results

We evaluate our algorithms using a simulator driven by both trace workload and synthetic workload. The simulator models a shared-disk file system server cluster. Experimental results verify the effectiveness of our load placement system in handling heterogeneity and show that our system performs comparably to a prescient load placement system. The experimental results helped us to locate and solve the problem of over-tuning as well.

We have constructed a simulator using the YACSIM [14] toolkit. YACSIM is a C-based library for discrete-event simulation. In the simulator, servers use a first-in-first-out queuing discipline. The latency of each server is collected over a specified interval of time and written into a log file.

We use the DFSTrace traces [24] to drive our experiments. DFSTrace data were collected on about 30 different workstations running different file systems such as AFS [13], Coda [29], NFS [35], and the local Unix file system. DFSTrace data is naturally partitioned along workstation boundaries. Therefore, the metadata portion of the DFSTrace workload is equivalent to the workload of a file set; DFSTrace data match our shared-nothing architecture well.

DFSTrace data has shortcomings in driving our simulation. DFSTrace data was collected on legacy hardware, The data provides limited ability to explore the performance of our system because it represents a fixed hardware configuration. To get around the limitations of DFSTrace, we perform experiments driven by a synthetic workload as well. Synthetic workload drives our simulation to the hardware capacity, and, explores different hardware/workload configurations. Synthetic workload lets us represent and experiment with an arbitrary amount of heterogeneity.

We compare our load management system with three other systems: *simple randomization*, which assigns each file

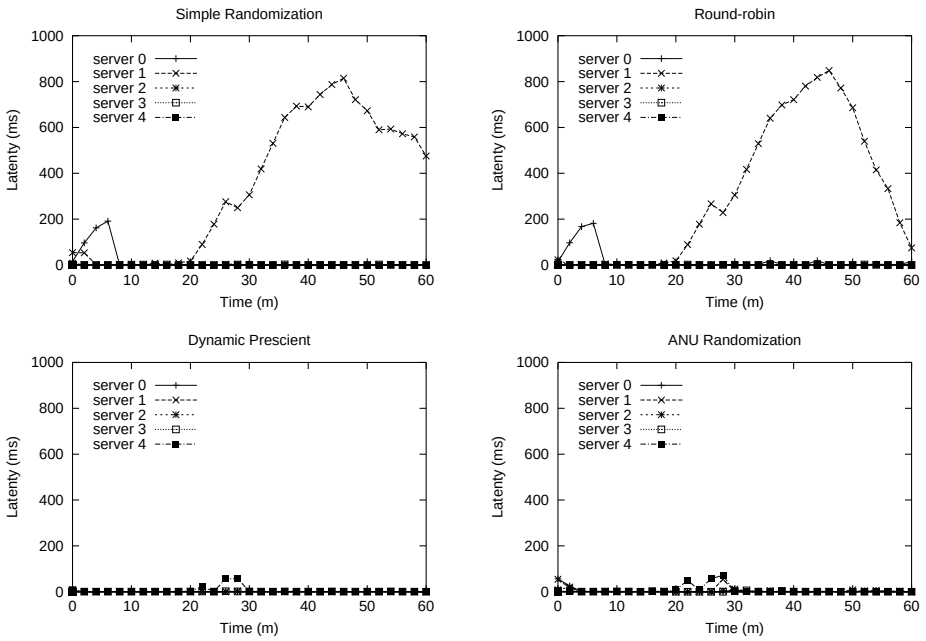


Figure 6: Server latency for DFSTrace workloads.

set to a randomly-chosen server; *round-robin* placement, which assigns the same number of file sets to each server; and dynamic *prescient* placement, which knows the processing capabilities of each server and the workload characteristics of each file set. Simple-randomization and round-robin allow us to compare ANU randomization with static, offline policies used in heterogeneous clusters. The dynamic prescient system provides an upper bound for load balancing; it realizes the best possible load balance, because it uses perfect knowledge of server capabilities and identifies the permutation of file sets onto servers that minimizes load skew. We note that even dynamic prescient load placement does not result in exact load balance because file sets are indivisible and heterogeneous.

To measure the performance characteristics of our load placement system under a real workload, we run simulations based on DFSTrace data (Figure 6). We pick a high-activity one hour interval from the traces to test the performance of our system and the other three systems described above. There are 112,590 client requests in total and 21 file sets accessed in the one hour trace. The file sets display highly heterogeneous workload characteristics; *e.g.* the most active file set has more than one hundred times as many requests as many of the least active file sets. Based on the number of file sets, we choose to simulate a cluster of five servers. Our experiments include server heterogeneity as well. Server 0, 1, 2, 3, and 4 have processing power 1, 3, 5, 7, and 9 respectively. More specifically, for the same workload, if the least powerful server in our simulated five-server cluster (server 0) consumes time T to complete a metadata request, then the most powerful server (server 4) consumes time $T/9$.

Figure 6 shows the latency of the five servers when serving metadata requests over a one hour period. The prescient policy and ANU randomization update the workload configuration every two minutes. Although dynamic policies can update configurations at any time scale, we found two minutes to strike a balance between over-tuning and responsiveness. We note that it takes five to ten seconds to move a file set from one server to another in our target system. The releasing server needs to flush its cache, writing all dirty data back to stable storage. The acquiring server must initialize the file set. Furthermore, the acquiring file server starts with a cold cache, which hinders performance initially. Therefore, our system is relatively conservative in moving data in response to short-term bursts in workload.

Simple randomization and round-robin systems perform poorly because they are static algorithms. They have no knowledge of server or workload heterogeneity and cannot respond to skew when it occurs. Over the hour, the least powerful server's performance degrades, even though the more powerful servers have unused capacity.

The prescient and ANU randomization policies balance load over the course of the experiment. Figure 7 shows a closeup of the prescient and ANU randomization results of Figure 6. Having perfect knowledge, the prescient algorithm begins in a load-balanced state at time 0. The adaptive prescient algorithm looks forward into the trace, identifying the best load balance before the workload occurs and configuring the servers to best handle that workload. ANU randomization has no *a-priori* knowledge and, therefore, assumes initially that all file sets and all servers are

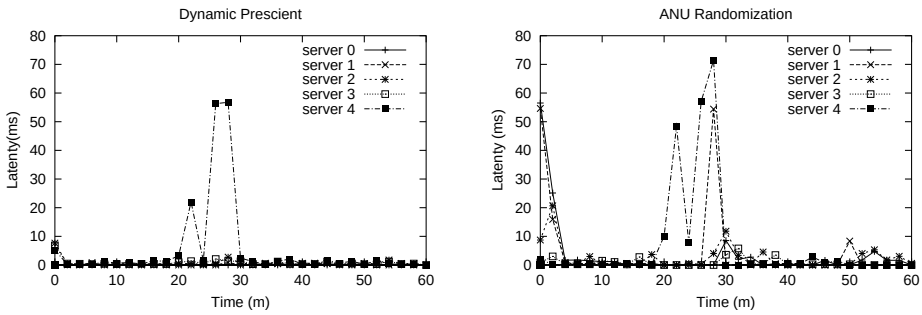


Figure 7: Dynamic Prescient vs. ANU Randomization for DFSTrace workloads.

uniform. Over the first 3 sample periods (6 minutes), ANU randomization adapts to workload and server heterogeneity, reaching a good load balance.

Both systems show increases in latency on the most powerful servers under periods of heavy load. The bursts of load occur in few file sets and both algorithms localize those bursts to the most powerful servers. The prescient algorithm does so more effectively because it can permute file sets, moving any file set to any server to get the best fit between workload and server capabilities. ANU randomization is restricted by the cache-preserving property and minimizing load movement to move file sets incrementally and does not achieve as "good a fit" of workload to servers. However, ANU randomization does perform comparably. After experiencing one load increase on server 4, ANU randomization enlists the next most powerful server (server 3) in the next time step. One of the limitations of ANU randomization is that the algorithm does not directly move load. Rather, it scales mapped regions. We cannot predict the result of a configuration change under ANU randomization. The combination of scaling and randomization may result in no load moving or more load than expected moving.

To better understand the impact of our system under extreme workload heterogeneity, we conduct experiments using synthetic workload. The synthetic workload consists of 100,000 client requests against 500 file sets during a period of 10,000 seconds. Although workload inter-arrival times in each file set are governed by a Poisson process, the distribution of requests from each file set is stable for the duration of the simulation. To ensure file set workload heterogeneity, the workload of each file set is defined as Xc where X is randomly chosen from interval $[1, 10]$ and c is a scaling factor. For the purpose of comparison, we configure the same five servers as in our DFSTrace experiments and tune c so that the system is below peak load.

Our trace-driven experimental results follow our findings from DFSTrace. Figure 8 shows results from the experiments driven by synthetic workload and Figure 9 shows a closeup comparison of the prescient and ANU randomization policies. The prescient policy retains the same configuration for the duration of the experiment, because the workload for each file set does not vary with time. We again observe the simple randomization and round-robin policies cannot deal with heterogeneity. As with DFSTrace, the prescient policy starts off balanced and ANU randomization takes some time to discover heterogeneities and converge. For the most part, the prescient algorithm and ANU randomization are comparable.

Synthetic workload results illustrate the difference between the power of randomization and bin-packing approaches, of which the prescient algorithm is one example. The prescient algorithm places a single, small file set (workload = c) on the least powerful server. This is the optimal configuration. ANU randomization cannot pick which file sets get assigned to which server, and, therefore, cannot place a small file set on the least powerful server. Instead, the least powerful server has no load in the steady state. Its efforts to place a file set on the least of all servers, at times 4 and 18, result in much larger latency than is tolerable, because the file set in question was too big. ANU randomization exchanges the fine-grained tuning of bin-packing approaches for addressing and scalability benefits described in Section 5.

Early versions of our algorithm exhibited over-tuning – an undesirable side-effect of its aggressiveness in perfecting load balance. Figure 10 shows the over-tuning problem on our synthetic workload. The weakest server (server 0) cyclically takes on workload, exhibits high latency, releases workload, and goes to zero latency. Server heterogeneity accounts for over-tuning in this case. The three heuristics – thresholding, top-off, and divergent tuning – solve the problem.

In Figure 11, we decompose the contributions of each of the three heuristics. Each graph shows the effect of using only one of the policies. Thresholding prevents the algorithm from tuning as aggressively. This can be seen

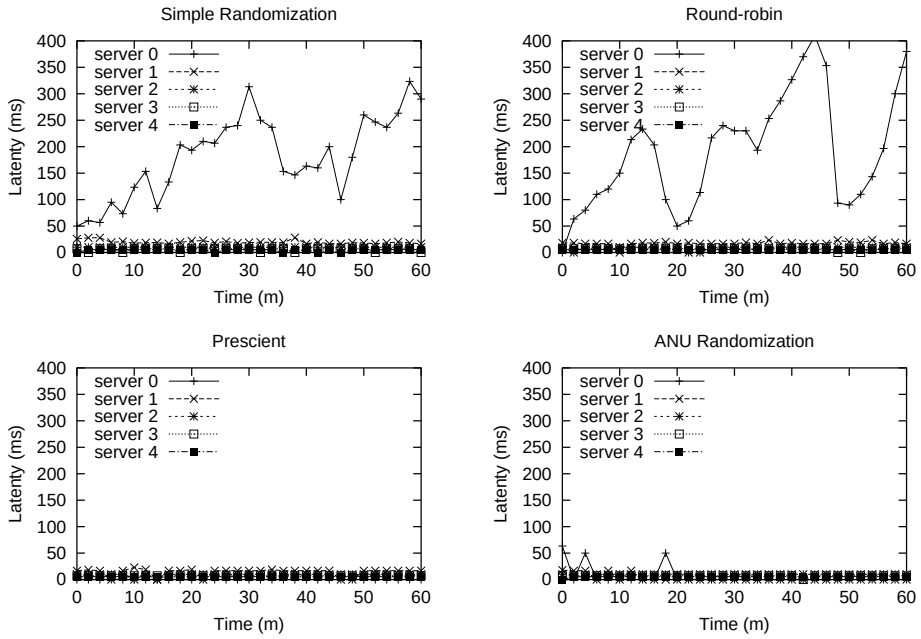


Figure 8: Server latency for synthetic workload.

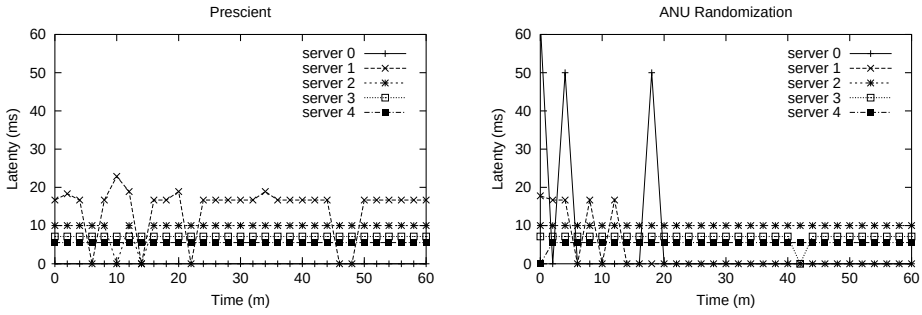
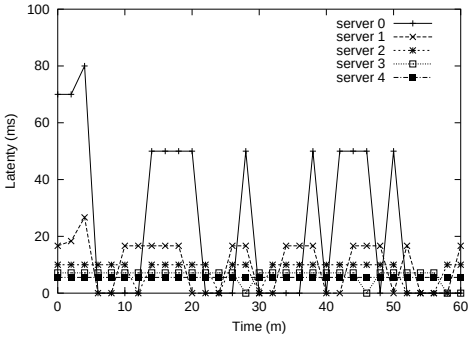


Figure 9: Prescient vs. ANU Randomization for synthetic workload.

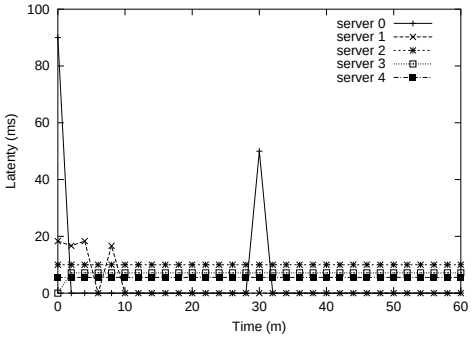
in server 1 in Figure 11(a), which is more stable than in Figure 10(a). However, server 0 fluctuates above and below the threshold. So while thresholding stabilizes the system, it does not address extreme server heterogeneity, in which load jumps between 0 and values above the high threshold. Top-off tuning extends thresholding, allowing servers to sit idle at zero latency, with no workload. Top-off tuning only tunes down the mapped regions of overloaded servers. Underloaded servers gain load implicitly by "catching" load shed from overloaded servers. Top-off tuning is the single most effective of the three policies. It tunes the least powerful server down to no workload (Figure 11(b)) and the second least powerful server fluctuates modestly from having no workload to having a small amount of workload within the threshold latency range. Divergent tuning stabilizes the workload by only tuning servers whose load is moving away from a balanced average. Divergent tuning reaches a load-balance, but does so more slowly than all three policies combined.

8 Conclusions

We have described a load-management and server provisioning system based on adaptive, non-uniform randomization. ANU randomization balances the scalability and addressing properties of distributed hashing with load-balancing performance comparable to a prescient algorithm. ANU randomization also maintains as much of the current configuration as possible when balancing load, preserving the cache contents of servers. The system extends randomized load-balancing approaches to make them tunable. Thus, the system can handle arbitrary amounts of heterogeneity in server capability and workload.

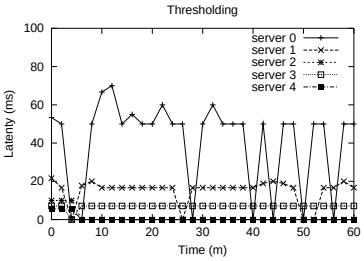


(a) Initial results exhibit over-tuning.

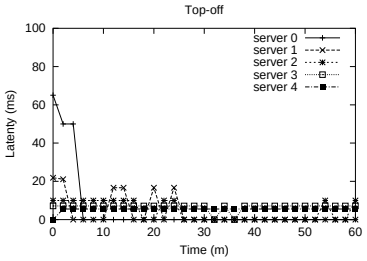


(b) Three heuristics solve the over-tuning problem.

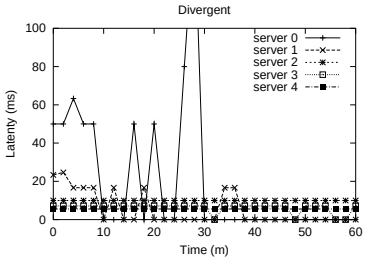
Figure 10: The over-tuning problem – before and after.



(a) Thresholding



(b) Top-off



(c) Diverge-only

Figure 11: The three techniques to solve over-tuning.

Experimental results show that ANU randomization performs similarly to an optimal prescient algorithm. It handles workload skew and server heterogeneity under both trace-driven workloads and synthetic workloads. Results also show that ANU randomization over-tunes a system and verifies the effectiveness of three heuristics we develop to prevent over-tuning.

Our experience with ANU randomization indicates that it is a powerful technique for managing load in large-scale, shared-disk architectures. ANU randomization places indivisible workload units onto a set of servers. Although it is designed for a shared-disk file system, it suits any architecture in which data are partitioned among servers at runtime, but can be moved from server to server. This includes Web servers, clustered databases, and NFS servers. ANU randomization achieves load balance quickly without any foreknowledge of the workload or servers. The ability to manage a cluster without knowledge is equivalent to a system being "self-managing," in that no administration inputs are required. This allows clusters to scale to sizes that were previously unmanageable and allows a cluster to enlist unknown heterogeneous resources automatically.

References

[1] K. Amiri, D. Petrou, G. R. Ganger, and G. A. Gibson. Dynamic function placement for data-intensive cluster computing. In *Proceedings of the 2000 USENIX Annual Technical Conference*, 2000.

[2] D. Anderson, J. Chase, and A. Vahdat. Interposed request routing for scalable network storage. In *Proceedings of the Symposium on Operating Systems Design and Implementation*, 2000.

[3] L. Aversa and A. Bestavros. Load balancing a cluster of web servers using distributed packet rewriting. In *Proceedings of the IEEE International Performance, Computing, and Communications Conference*, 2000.

[4] A. Barak, G. Shai, and R. G. Wheeler. *The MOSIX Distributed Operating System: Load Balancing for UNIX*. Springer Verlag, 1993.

[5] G. Bell and J. Gray. High Performance Computing: Crays, Clusters and Centers. What Next? Technical Report MSR-TR-2001-76, Microsoft Research, 2001.

[6] P. J. Braam. The Lustre storage architecture. Technical Report available at – http://www.lustre.org/docs.html, Lustre, 2002.

[7] A. Brinkmann, K. Salzwedel, and C. Scheideler. Compact, adaptive placement strategies for non-uniform capacities. In *Proceedings of Symposium on Parallel Algorithms and Architectures*, 2002.

[8] R. Burns. *Data management in a distributed file system for Storage Area Networks*. Ph.D. dissertation, University of California at Santa Cruz, 2000.

[9] V. Cardellini, M. Colajanni, and P. Yu. Geographic load balancing for scalable distributed web systems. In *Proceedings of the International Symposium on Modeling, Analysis and Simulation of Computer and Telecommunication Systems*, 2000.

[10] D. L. Eager, E. D. Lazowska, and J. Zahorjan. Adaptive load sharing in homogeneous distributed systems. *IEEE Transactions on Software Engineering*, 12(5), 1986.

[11] G. R. Ganger, B. L. Worthington, R. Y. Hou, and Y. N. Patt. Disk subsystem load balancing: Disk striping vs. conventional data placement. In *Proceedings of the International Conference on System Sciences*, 1993.

[12] M. Harchol-Balter and A. Downey. Exploiting process lifetime distributions for dynamic load balancing. *ACM Transactions on Computer Systems*, 15(3), 1997.

[13] J. H. Howard, M. L. Kazar, S. G. Menees, D. A. Nichols, M. Satyanarayanan, R. N. Sidebotham, and M. J. West. Scale and performance in a distributed file system. *ACM Transactions on Computer Systems*, 6(1), 1988.

[14] J. R. Jump. *YACSIM reference manual*. Rice University, version 2.1.1 edition, 1993.

[15] E. D. Katz, M. Butler, and R. McGrath. A scalable HTTP server: The NCSA prototype. *Computer Networks and ISDN Systems*, 27(2), 1994.

[16] K. Li and J. Dorband. A task scheduling algorithm for heterogeneous processing. In *Proceedings of the Symposium on High Performance Computing*, 1997.

[17] W. Litwin, M. Neimat, and D. A. Schneider. LH* — a scalable, distributed data structure. *ACM Transactions on Database Systems*, 21(4), 1996.

[18] C. Lu and S. Lau. An adaptive load balancing algorithm for heterogeneous distributed systems with multiple task classes. In *Proceedings of the International Conference on Distributed Computing Systems*, 1996.

[19] M. Mehta and D. J. DeWitt. Data placement in shared-nothing parallel database systems. *VLDB Journal*, 6(1), 1997.

[20] J. Menon, D. A. Pease, R. Rees, L. Duyanovich, and B. Hillsberg. IBM storage tank: A heterogeneous scalable SAN file system. *IBM Systems Journal*, 42(2), 2003.

[21] M. Mitzenmacher. On the analysis of randomized load balancing schemes. In *the ACM Symposium on Parallel Algorithms and Architectures*, 1997.

[22] M. Mitzenmacher. How useful is old information? *IEEE Transactions on Parallel and Distributed Systems*, 11(1), 2000.

[23] M. Mitzenmacher. The power of two choices in randomized load balancing. *IEEE Transactions on Parallel and Distributed Systems*, 12(10), 2001.

[24] L. Mummert and M. Satyanarayanan. Long term distributed file reference tracing: Implementation and experience. *Software - Practice and Experience*, 26(6), 1996.

[25] S. Petri and H. Langendorfer. Load balancing and fault tolerance in workstation clusters migrating groups of communicating processes. *ACM SIGOPS Operating Systems Review*, 29(4), 1995.

[26] K. W. Preslan, A. P. Barry, J. E. Brassow, G. M. Erickson, E. Nygaard, C. J. Sabol, S. R. Soltis, D. C. Teigland, and M. T. O'Keefe. A 64-bit, shared disk file system for Linux. In *Proceedings of the IEEE Mass Storage Systems Symposium*, 1999.

[27] A. Rowston and P. Druschel. Pastry: Scalable, distributed object location and routing for large-scale peer-to-peer systems. In *Proceedings of the IFIP/ACM International Conference on Distributed Systems Platforms*, 2001.

[28] Y. Saito, B. N. Bershad, and H. M. Levy. Manageability, availability and performance in porcupine: A highly scalable, cluster-based mail service. In *Proceeddings of the Symposium on Operating Systems Principles*, 1999.

[29] M. Satyanarayanan, J. J. Kistler, P. Kumar, M. E. Okasaki, E. H. Siegel, and D. C. Steere. Coda: A highly available file system for a distributed workstation environment. *IEEE Transactions on Computers*, 39(4), 1990.

[30] F. Schmuck and R. Haskin. GPFS: A shared-disk file system for large computing clusters. In *Proceedings of the Conference on File and Storage Technology*, 2002.

[31] P. Shenoy and H. Vin. Efficient striping techniques for multimedia file servers. In *Proceedings of the International Workshop on Network and Operating System Support for Digital Audio and Video*, 1997.

[32] S. Sinha and M. Parashar. Adaptive runtime partitioning of AMR applications on heterogeneous clusters. In *Proceedings of the IEEE International Conference on Cluster Computing*, 2001.

[33] I. Stoica, R. Morris, D. Karger, M. F. Kaashoek, and H. Balakrishnan. Chord: A scalable peer-to-peer lookup service for Internet applications. In *Proceedings of ACM SIGCOMM*, 2001.

[34] C. A. Thekkath, T. Mann, and E. K. Lee. Frangipani: A scalable distributed file system. In *Proceedings of the ACM Symposium on Operating System Principles*, 1997.

[35] D. Walsh, B. Lyon, G. Sager, J. Chang, D. Goldberg, S. Kleiman, T. Lyon, R. Sandberg, and P. Weiss. Overview of the Sun network file system. In *Proceedings of the 1985 Winter Usenix Technical Conference*, January 1985.

[36] J. Watts, M. Rieffel, and S. Taylor. Dynamic management of heterogenous resources. In *Proceeding of the High Performance Computing Conference: Grand Challenges in Computer Simulation*, 1998.

[37] J. Watts and S. Taylor. A practical approach to dynamic load balancing. *IEEE Transactions on Parallel and Distributed Systems*, 9(3), 1998.

[38] M. Willebeek-LeMair and A. P. Reeves. Strategies for dynamic load balancing on highly parallel computers. *IEEE Transactions on Parallel and Distributed Systems*, 4(9), 1993.

[39] C. Wu and F. Lau. *Load Balancing in Parallel Computers: Theory and Practice.* Kluwer Academic Publishers, Boston, Mass, 1997.

[40] S. Zhou. A trace-driven simulation study of dynamic load balancing. *IEEE Transactions on Software Engineering*, 14(9), 1988.

[41] S. Zhou, J. Wang, X. Zheng, and P. Delisle. Utopia: A load-sharing facility for large heterogeneous distributed computing systems. *Software – Practice and Experience*, 23(12), 1993.

[42] H. Zhu, T. Yang, Q. Zheng, D. Watson, O. H. Ibarra, and T. R. Smith. Adaptive load sharing for clustered digital library servers. *International Journal on Digital Libraries*, 2(4), 2000.