

Tunable Randomization for Load Management in Shared-Disk Clusters

CHANGXUN WU and RANDAL BURNS

Johns Hopkins University

We develop and evaluate a system for load management in shared-disk file systems built on clusters of heterogeneous computers. It balances workload by moving file sets among cluster server nodes. It responds to changing server resources that arise from failure and recovery, and dynamically adding or removing servers. It also realizes performance consistency—nearly uniform performance across all servers. The system is adaptive and self-tuning. 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, nonuniform (ANU) randomization. ANU randomization realizes the scalability and metadata reduction benefits of hash-based, randomized placement techniques, while avoiding hashing's drawbacks: load skew, inability to cope with heterogeneity, and lack of tunability. ANU randomization outperforms virtual-processor approaches to load balancing, while reducing the amount of shared state among servers and the amount of load movement.

Categories and Subject Descriptors: H.3.4 [Information Storage and Retrieval]: Systems and Software—*Distributed systems, Performance evaluation*; D.4.3 [Operating Systems]: File Systems Management—*Distributed file systems, File organization*

General Terms: Algorithms, Management, Measurement, Performance

Additional Key Words and Phrases: Load management, shared-disk file systems, computer clusters, heterogeneity

1. INTRODUCTION

Computer clusters provides a cost-effective alternative to large supercomputers [Anderson et al. 1995; Becker et al. 1995; Barak et al. 1999], due to the low cost and wide availability of commodity hardware components. Also, clusters may be progressively updated by repairing or replacing components with new hardware. Clusters are scalable in that it is possible to build systems ranging from a few computers to thousands of nodes, with proportional cost, achieving near-proportional performance increases.

Authors' addresses: Dept. of Computer Science, Johns Hopkins University, 224 New Engineering Bldg., 3400 N. Charles St., Baltimore, MD 21218; email: {wu,randal}@cs.jhu.edu.

Permission to make digital or hard copies of part or all of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or direct commercial advantage and that copies show this notice on the first page or initial screen of a display along with the full citation. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, to republish, to post on servers, to redistribute to lists, or to use any component of this work in other works requires prior specific permission and/or a fee. Permissions may be requested from Publications Dept., ACM, Inc., 1515 Broadway, New York, NY 10036 USA, fax: +1 (212) 869-0481, or permissions@acm.org.

© 2004 ACM 1553-5307/04/1200-0108 \$5.00

ACM Transactions on Storage, Vol. 1, No. 1, December 2004, Pages 108–131.

Shared-disk file systems [Schmuck and Haskin 2002; Thekkath et al. 1997] provide good solutions for managing large sets of data within a cluster. A recent trend uses storage area networks (SANs) to build shared-disk file systems [Preslan et al. 1999; Menon et al. 2003]. SANs provide all nodes with symmetric access to shared, block-storage devices. They offer fast data access, improved data availability, and increased system scalability.

The trends toward low-cost commodity hardware and open-source systems facilitate the construction of computing clusters and file systems on heterogeneous hardware rather than the proprietary parallel supercomputers [Bell and Gray 2001; Braam 2002]. As shared-disk file systems have previously been built with the assumption of homogeneous hardware, the load placement policies they employ are insensitive to heterogeneity [Thekkath et al. 1997; Schmuck and Haskin 2002]. Current methods do not translate to commodity configurations, which must adapt to changing workloads and hot spots. Also, they must address the different capabilities of different hardware components and reorganize the system in response to component failure. The trend of integrating clusters into computational grids, together with the relatively high failure rates in grid environments, makes the situation even worse, as it implies frequent configuration changes.

Heterogeneity comes in many forms and presents challenges to system management. Load management systems need to cope with spatial heterogeneity from workload units containing different amounts of data with nonuniform access properties. They also need to adapt to temporal heterogeneity by changing load placement in response to workload shifts at an arbitrarily fine time granule. They are concerned with server heterogeneity, balancing load over servers with different performance characteristics. Furthermore, they should realize future adaptability, which means upgrading hardware while the system is online, and taking full advantage of faster hardware, that is, slower hardware does not limit performance. The goal of this research is to build a self-managing resource allocation and load balancing system that adapts to heterogeneous servers and nonuniform workloads with no *a priori* knowledge of either workload or server properties.

Human administration costs of cluster systems built from commodity components is a large and growing issue. For example, Feng et al. [2002] reported that administration cost dominated the total cost of ownership for a 24-node cluster over a four-year period. In most traditional cluster management systems, human intervention is required in the cases of system configuration changes, such as adding or removing servers and failure and recovery. To reduce costs, it is crucial to build adaptive, self-tuning cluster management systems that integrate automated management functions.

Our system addresses the problem of load placement for shared-disk file systems built on heterogeneous servers. Based on a technique called adaptive, nonuniform (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 adaptive hashing strategy of Brinkmann et al. [2002]. ANU randomization makes randomized load-placement tunable by adding a layer of abstraction between servers and workload. By employing

randomization, our system minimizes the metadata needed to keep track of load, so that shared state scales with the number of servers, rather than size of the workload. The system adapts to changes in workload and hardware resources, including the addition and decommissioning of cluster components, as well as failure and recovery. Finally, the system is self-tuning: automated at all stages, including initial configuration.

Some of 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 qualitative ways in which our solution differs from existing heterogeneous load-placement schemes [Sinha and Parashar 2001; Watts et al. 1998; Li and Dorband 1997; Barak et al. 1993; Zhou et al. 1993]. When considered together, these minimize the cost of reorganizing the system to workload or configuration changes. Servers are dynamically interchangeable and reconfigurable without negatively affecting the performance of applications, facilitating the trend of building “clusters on demand” [Chase et al. 2003]. For example, the same server might be deployed in different clusters at different times during the same day.

We also built the system to achieve consistent performance. Tasks see similar latencies regardless of the node on which they are serviced. Experiments reveal this to be true for jobs assigned to servers of widely varying processing power. This property does not degrade the performance of the algorithm on other key metrics, such as aggregate latency or the amount of load moved. We assert that load management should consider the performance of each individual component of a cluster, not just aggregate average latency and throughput. We use this feature to identify weak or incompetent servers and effectively remove them from the cluster, by assigning them no tasks.

Our treatment includes experimental results in which we compare ANU randomization to several other load-balancing systems. This includes a prescient algorithm, a state-of-the-art virtual processor system, and a randomization system, similar to those used in distributed hash table (DHT) systems [Ratnasamy et al. 2001; Rowstron and Druschel 2001]. The prescient algorithm has complete knowledge of server and workload properties and uses a bin-packing approach to minimize latency. It represents an upper bound on performance metrics, such as task latency. For a virtual processor system, we implemented the RefineLB policy of the Charm++ system [Huang et al. 2003; Parallel Programming Lab 2004]. Charm++ is a widely-used parallel programming environment and represents a sound, proven virtual processor design. With virtual processors, load is assigned to abstract virtual processors and virtual processors are load-balanced across physical cluster nodes. When compared with bin-packing techniques, virtual processors reduce the amount of shared-state from the number of workload units to the number of virtual processors.

Experiments reveal that ANU randomization achieves near prescient performance while realizing all of its other merits. ANU randomization both outperforms virtual processors and uses less shared state. It provides consistent performance across heterogeneous servers. Finally, it realizes the scalability and metadata reduction benefits of hash-based, randomized placement techniques,

while avoiding hashing's drawbacks: load skew, inability to cope with heterogeneity, and lack of tunability.

2. RELATED WORK

Much work has been done in the area of load management in clusters and distributed systems. Lots of dynamic load management techniques are designed for parallel systems and homogeneous clusters [Watts and Taylor 1998; Willebeek-LeMair and Reeves 1993; Eager et al. 1986; Wu and Lau 1997; Zhou 1988]. Workload is transferred from heavily loaded servers to lightly loaded ones. Another family of techniques [Zhou et al. 1993; Zhu et al. 2000; Watts et al. 1998] take into account server heterogeneity, but require all servers to periodically broadcast load and available capacity. These techniques assume knowledge of the capacity of any given server. Utopia [Zhou et al. 1993] uses precomputed knowledge of nonuniform 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. [2000] 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.

Randomization is a powerful technique for load management in clusters and distributed systems [Mitzenmacher 1997]. Many systems, especially peer-to-peer distributed hash-table (DHT) systems [Ratnasamy et al. 2001; Rowstron and Druschel 2001], rely on randomization, or more specifically, pseudo-random hashing to uniformly distribute workload units. 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 nodes may simultaneously move together to a node that previously had available capacity [Mitzenmacher 2000]. Therefore, these systems do not actively balance load. They use simple randomized load placement, which balances load in practice, while avoiding message exchanges between the system nodes [Mitzenmacher 1997; Mitzenmacher 2001]. Although simple randomization works well in certain environments, our experiments indicate that it cannot support extreme workload and server heterogeneity.

Virtual processors are another technique widely used for load balancing in parallel systems and conventional clusters [Nedeljkovic and Quinn 1992; Kale et al. 2000]. Each virtual processor is assigned a small portion of the entire workload and the system dynamically maps these virtual processors to physical processors to achieve load balance. Peer-to-peer systems have adopted the virtual processor approach to manage load on peer nodes as well [Stoica et al. 2001; Dabek et al. 2001; Rao et al. 2003]. However, virtual processor-based load balancing requires larger shared state, and performs worse than our system.

Similar to our approach, the lookup schemes used in some DHT systems [Stoica et al. 2001; Rowstron and Druschel 2001] 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 schemes assume that peer nodes are homogeneous, and that objects have the

same size [Rao et al. 2003]. By themselves, these schemes cannot handle server or workload heterogeneity. In contrast, our technique makes the abstract hash-mapping tunable and is able to handle workload heterogeneity as well as server heterogeneity.

Much of the research in load balancing does not compare directly to our work, due to fundamental differences in workload and architecture. We develop techniques for shared-disk file serving. The workload consists of relatively short tasks, and each request must be served by a specific server. The load-balancing system uses shared-disk to move workload from server to server. Many systems redirect requests within a cluster to dynamically balance load. This is a popular form of load balancing suitable when any server can handle any request. Examples include DNS rotation [Katz et al. 1994; Cardellini et al. 2000] and request routing in Web servers [Aversa and Bestavros 2000]. Similarly, block I/O systems use striping to distribute large I/O requests across many disks [Ganger et al. 1993]. This approach applies when I/O are divisible and large. Striping is frequently used in multimedia servers [Shenoy and Vin 1997]. There has also been research in the area of functional decomposition [Amiri et al. 2000; Anderson et al. 2000]. 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 [Anderson et al. 2000]. Finally, there is a long history of load balancing through process migration [Harchol-Balter and Downey 1997; Barak et al. 1993; Petri and Langendorfer 1995; Lu and Lau 1996], in which active processes are transferred among computers. Process migration reorganizes the assignment of long running jobs among a cluster of computers.

3. LOAD-MANAGEMENT IN CLUSTERS

A brief review of the architecture of shared-disk file system clusters [Menon et al. 2003; IBM 2003; Lustre 2002; Panasas 2003] and their workload characteristics helps to facilitate our discussion and generate our solution. A shared-disk file system cluster usually uses a single global namespace, which is partitioned into file sets. A file set is a subtree of the global namespace and also the indivisible unit of workload assignment and movement. Many shared-disk file system clusters distinguish between data and metadata, which are stored and accessed separately. The shared disks hold file sets, and metadata of the file sets are assigned to file servers. In a typical access, a client sends a metadata request to a file server. The server sends the location information of the specified file(s) back to the client. Then, the client fetches data directly from the disk, across the storage area network (SAN). This architecture separates metadata workload from data workload and targets them to file servers and shared disks, respectively. File servers are loaded with a single class of metadata operations and do not serve sequential or large file I/O, which goes to shared disks. Figure 1 shows the architecture of a typical shared-disk file system cluster. Our system focuses on managing load on file servers. In shared-disk file systems, file servers are a recoverable resource. Imbalance in file servers adversely affects overall system performance, because clients acquire metadata prior to

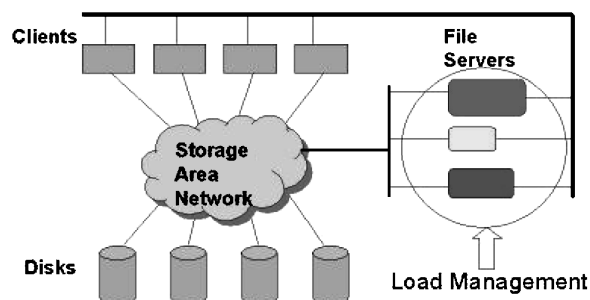


Fig. 1. Shared-disk cluster system architecture.

data. Clients blocked on metadata may leave the high bandwidth SAN underutilized. Our system does not address load management issues in shared disks, which is a separate problem in and of itself and requires different strategies [Ganger et al. 1993]. Although our load management scheme was initially designed for shared-disk file system clusters, it is suitable for any cluster system that partitions workload and has relatively short tasks.

4. ANU RANDOMIZATION

We use a technique called adaptive, nonuniform (ANU) randomization to place and balance load. ANU randomization is based on the SIEVE adaptive hashing technique described by Brinkmann et al. [2002]. 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 subregions of the same size. We assign to each server one or more subregions. A server completely occupies all but one subregion, which may be partially occupied. In the following discussion, we call these subregions partitions, and we call the segments within the unit interval occupied by a server its mapped region. Each server is 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 remapping the partitions to servers, this technique copes gracefully with hardware changes, such as adding and removing servers.

The flexibility of nonuniform randomization comes from an extra level of abstraction in the server-to-workload mapping. 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 sizes 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

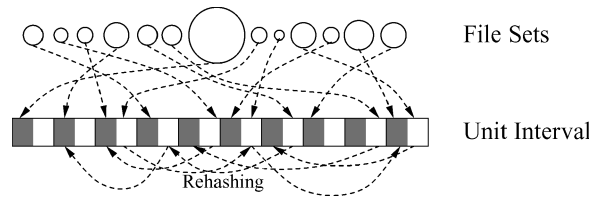


Fig. 2. Servers are assigned subregions of the unit interval. File sets of different size, representing different workloads, are hashed into this interval. File sets not mapped to servers by the first hash are re-hashed until assigned.

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, that is, 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 unmapped regions are rehashed until assigned to a mapped region. Rehashing 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

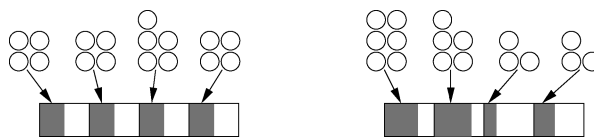


Fig. 3. Dealing with server heterogeneity. Initial configuration (left) and after reorganization (right).

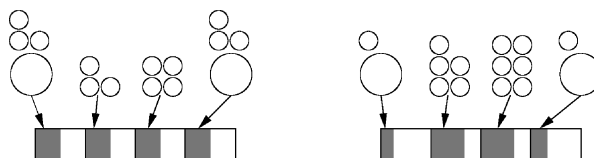


Fig. 4. Dealing with non-uniform workload. Initial configuration (left) and after reorganization (right).

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 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 nonuniform workloads. Figure 4 depicts four servers, which we assume to be uniform, serving nonuniform file sets, in which the size of the file set indicates the amount of



Fig. 5. Partitioning the unit interval when adding a server.

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 nonuniformity in workload on uniform servers.

Nonuniform 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 rehashed 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 repartitions 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 system 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 repartitions 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 noncontiguous. 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 [Litwin et al. 1996]. During failure and recovery, our system does not rehash all the file sets. Instead, it moves the minimum amount of workload possible by scaling the mapped regions of alive servers from the last configuration. Therefore, load locality is maintained, and caches of file sets are preserved.

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 [Brinkmann et al. 2002]. 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 DHT systems [Stoica et al. 2001; Rowstron and Druschel 2001], 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 rehashing 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 [Zhou et al. 1993; Watts et al. 1998] 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.

6. OVERTUNING

In the early-stages of this research, our algorithms displayed an “overtuning” 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 overtuning. 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 overtuning 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 overtuning heuristics. Thresholding is necessary to prevent overtuning, 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 overtuning 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 fails, in which case divergence cannot be evaluated by the new delegate. The ANU algorithm ignores this policy for the first round of tuning in the new configuration.

7. PERFORMANCE EVALUATION

We evaluate the performance of our load management system based on ANU randomization against three other systems using a simulator, driven by both traced and synthetic workloads. Simulation results verify that our system achieves load balance among heterogeneous server nodes, provides consistent performance for applications, and moves minimal amount of load when tuning load placement. Simulation results also indicate that our system performs comparably to a prescient system, and that virtual processor systems require a larger shared state to achieve similar performance. The experimental results helped us to locate and solve the problem of overtuning as well.

We compare our load management system with three other systems: *simple randomization*, which assigns each file set to a randomly-chosen server; dynamic *prescient* placement, which knows the processing capabilities of each server, and the workload characteristics of each file set; and *virtual processors*, which distributes file sets to virtual processors, and then dynamically maps those virtual processors to real servers. Simple-randomization allows 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. The virtual processor system first randomly distributes file sets into Nv virtual processors, in which N is the number of physical servers and v is a scaling factor, to tune the total number of virtual processors in the system. We vary the number of virtual processors based on the number of file sets in the simulation. The system then employs the RefineLB virtual processor load-balancing technique [Huang et al. 2003; Parallel Programming Lab 2004] and perfect knowledge about server capabilities to dynamically map virtual processors to servers. All systems, including the dynamic prescient system, do not realize an exact load balance, because file sets are indivisible and heterogeneous.

For the evaluation, we construct a simulator using the YACSIM toolkit, which is a C-based library for discrete event simulation. The simulator models a shared-disk server cluster, and servers use a first-in-first-out queuing discipline for workload. The simulator includes all of the load-management algorithms to be compared.

7.1 Trace-Driven Workload

We use the DFSTrace traces [Mummert and Satyanarayanan 1996] to drive our experiments. DFSTrace data were collected on about 30 different workstations running different file systems such as AFS [Howard et al. 1988], Coda [Satyanarayanan et al. 1990], NFS [Walsh et al. 1985], and the local Unix file system. DFSTrace data are 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.

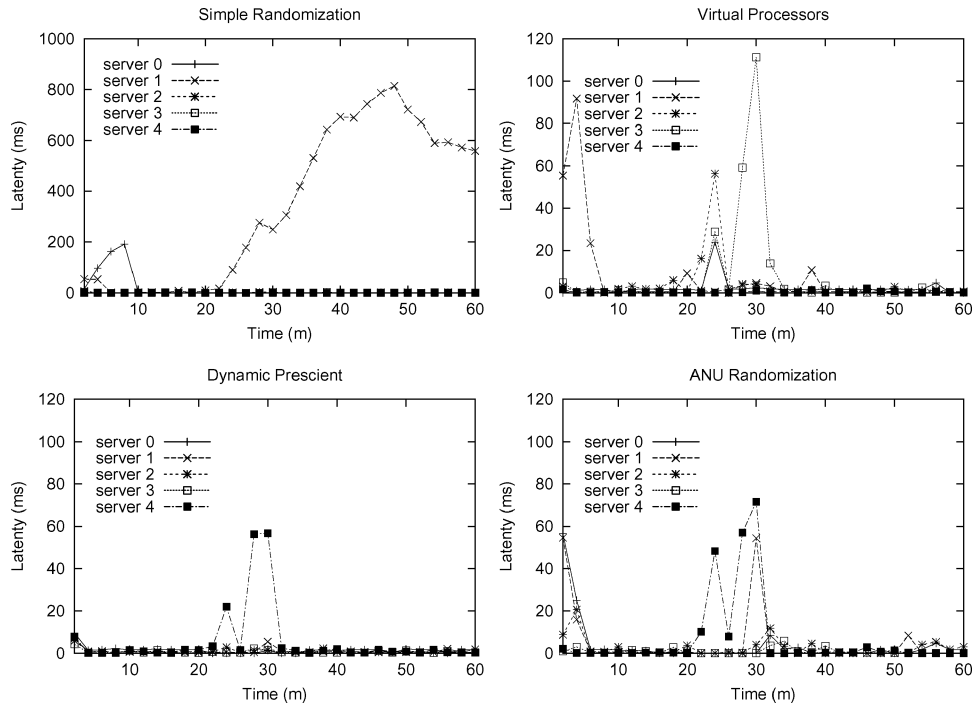


Fig. 6. Server latency for the DFSTrace workload.

To measure the performance characteristics of our load-placement system under a real workload, we run simulations based on DFSTrace (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; for example, 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 and set the virtual processor scaling factor v to be 2, resulting in 10 virtual processors for the 21 file sets. Our experiments include server heterogeneity as well. Server 0, 1, 2, 3, and 4 have processing power 1, 3, 5, 7, and 9, respectively; that is, for the same workload, if the least powerful server in our simulated 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, ANU randomization, and virtual processors update the workload configuration every two minutes. Although dynamic policies can update configurations at any time scale, we found that two minutes strikes a balance between overtuning 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 flushes its cache, writing all dirty data back to stable storage. The acquiring server initializes 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 performs poorly because it is a static algorithm. It has 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 scale on the simple-randomization graph is orders of magnitude larger than the graphs for the other policies.

The prescient and ANU randomization policies balance load over the course of the experiment. Having perfect knowledge, the prescient algorithm begins in a load-balanced state at time 0. It 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 uniform. Over the first 3 sample periods (6 minutes), ANU randomization adapts to workload and server heterogeneity, reaching a good load balance.

Both prescient and ANU randomization 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. It moves 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 another server 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.

The virtual processor system also achieves load balance during the simulation, but it performs slightly worse than the dynamic prescient and ANU randomization policies. Figure 6 shows that it converges more slowly than ANU randomization during the start-up phase, even though it has knowledge of server capacities. When dealing with the load bursts around time 22 and time 28, the virtual processor system fails to accommodate such bursts with the most powerful server and, therefore, yields a higher latency than other policies.

7.2 Synthetic Workload

We also run experiments driven by a synthetic workload, because DFSTrace data has shortcomings in driving our simulation. DFSTrace data was collected on legacy hardware with limited heterogeneity and, therefore, provides us limited ability to explore the performance of our system. To get around such limitations, we use a synthetic workload to drive the remaining experiments in

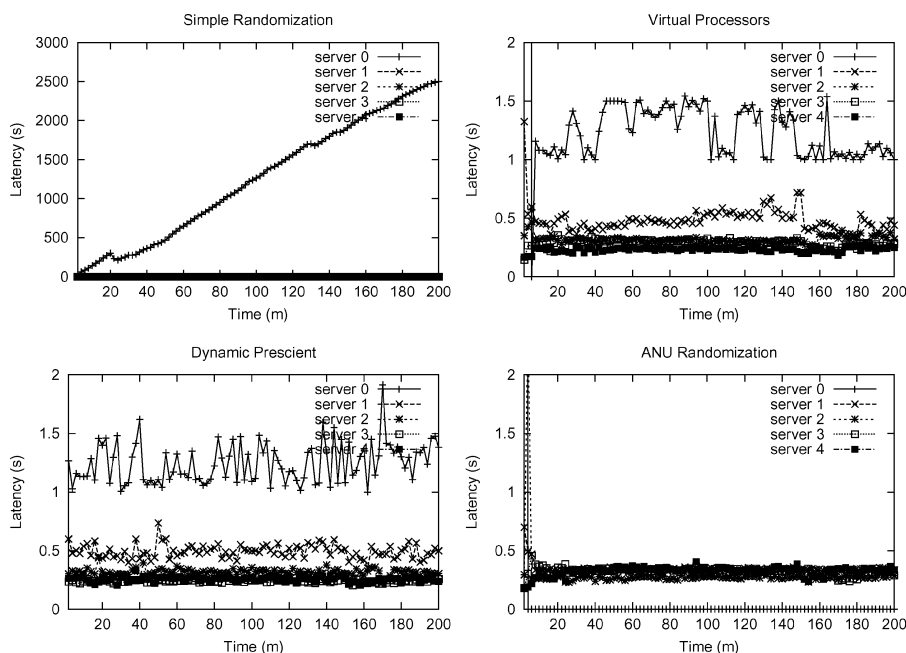
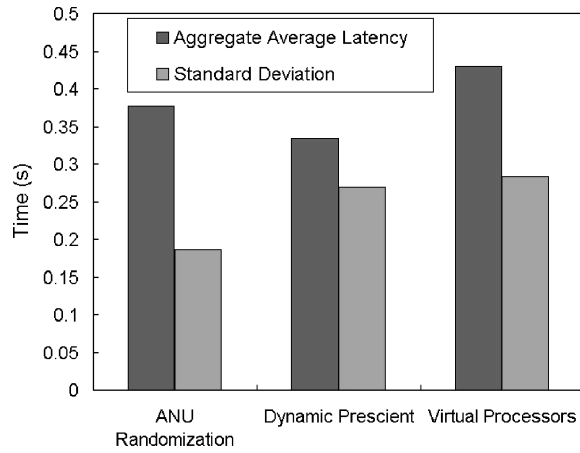


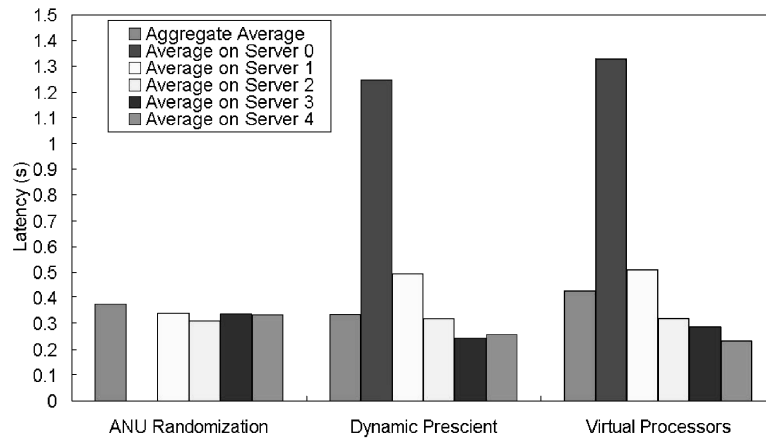
Fig. 7. Server latency for the synthetic workload.

this article. 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. The synthetic workload consists of 73,614 requests against 50 file sets during a period of 200 minutes. The amount of workload in each file set is defined as Xc , where X is randomly chosen from interval $[1,100]$, and c is a scaling factor to avoid overload of the whole system. Based on the number of file sets, we set the virtual processor scaling factor v to be 5, so that each virtual processor receives two file sets, on average. The request interarrival times in each file set are governed by a Pareto distribution that is heavy-tailed. For the purpose of comparison, we configure the same five servers as in our DFSTrace experiments.

The results of experiments driven by the synthetic workload follow our findings from DFSTrace-driven experiments. Figure 7 shows the latency of the five servers when serving the synthetic workload. Still, simple randomization is not able to handle skew and heterogeneity in the system. Having perfect knowledge of both server capacity and file set workload characteristics, dynamic prescient reaches load balance from the very beginning. ANU randomization quickly adapts to heterogeneity and maintains load balance through the simulation. Virtual processors maintain load balance in the simulation, but adapt more slowly than ANU randomization. Synthetic workload results conform to those of simulations based on file system traces (DFSTrace), showing the same scaling and tuning properties. This indicates the sanity of our synthetic workload.



(a) Aggregate average latency and standard deviation.



(b) Average latency of requests served on each server.

Fig. 8. Aggregate metrics.

7.3 Aggregate Metrics

To get a further understanding of ANU randomization's performance, we measure some aggregate performance metrics and compare these against those for the dynamic prescient and virtual processor policies. For each policy, we measure the performance metrics after the simulated policy stabilizes, that is, adapts to server and workload heterogeneity.

Figure 8(a) shows the aggregate average latency of all requests in the synthetic workload and its standard deviation. Dynamic prescient has the best aggregate average latency, because it realizes an optimal load mapping. Therefore, dynamic prescient also provides an upper bound of aggregate average latency, or equivalently, throughput. The aggregate average latency of ANU randomization is fairly close to that of dynamic prescient, indicating that ANU randomization provides throughput close to the upper bound without using α

priori knowledge of heterogeneity. Virtual processors again perform slightly worse than other policies.

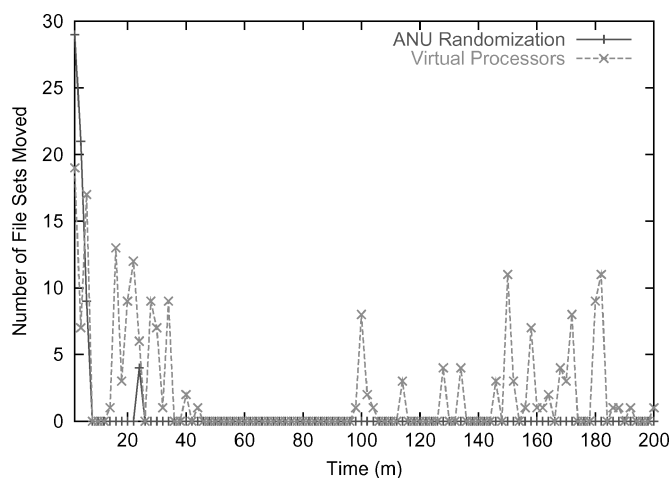
Figure 8(b) presents the average latency of tasks served by each individual server. For ANU randomization, servers exhibit consistent average latency values, with the exception of server 0—the weakest server—which receives no workload. ANU randomization enlists the most powerful servers to achieve good load balance, while allowing extremely weak servers to sit idle. The system identifies such incompetent components and notifies administrators. Applications see consistent latency over all servers once the system reaches load balance. As cluster and grid systems extend to support Service Level Agreements [Czajkowski et al. 2002; Chase et al. 2003], it is essential that application performance is consistent over different servers in a heterogeneous cluster, or even in a large-scale grid.

Results from trace-driven and synthetic workloads show that our load management scheme performs comparably to the upper bound (dynamic prescient) and provides the load balancing benefits of virtual processor schemes. Considering that our system achieves load balance without *a priori* knowledge of heterogeneity, it provides a competitive approach to load management.

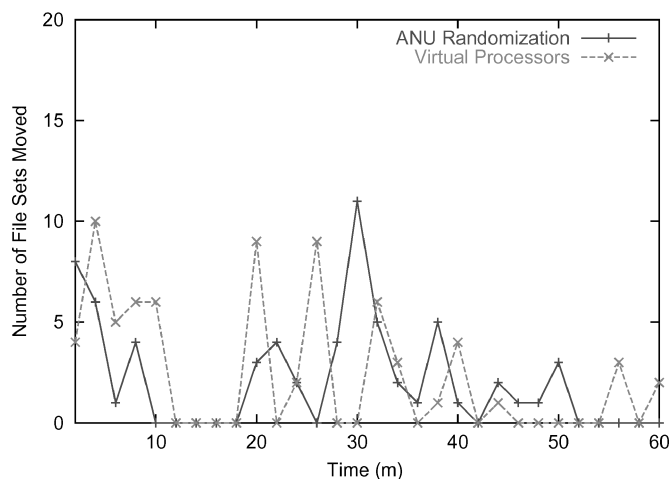
7.4 Load Movement and Stability

One of the design goals of our system is to minimize load movement when balancing load. Figure 9(a) compares the number of file sets moved by ANU randomization and virtual processors during the synthetic workload simulation. It shows that ANU randomization preserves load locality, or, in other words, minimizes load movement, during tuning. Without *a priori* knowledge of heterogeneity, ANU randomization initially assumes all servers and file sets are uniform and, therefore, assigns servers mapped regions of the same size. During the first several rounds of tuning, ANU randomization actively moves load among servers by scaling server's mapped regions to adapt to server and workload heterogeneity. Then it stabilizes around time 8, and rarely moves load after stabilizing. During the whole simulation, which consists of 100 rounds of tuning, ANU randomization moves 63 file sets, 93% of which are moved during the first three rounds of tuning. It indicates that ANU randomization is able to stabilize quickly and tolerates some amount of short-term load shift/imbalance once it reaches stability. The virtual processor system keeps moving load until the system stabilizes around time 42. However, it is not able to tolerate short-term load shift or load imbalance, and continues to move load during the simulation. The virtual processor system moves 207 file sets in 100 rounds of tuning.

We also include load movement results for the DFSTrace (Figure 9(b)), because the trace workload exhibits significant workload shifts. In 30 rounds of tuning, ANU randomization moves 64 file sets and the virtual processor system moves 71 file sets. Figure 9(b) shows that ANU randomization moved fewer file sets than virtual processors to adapt to heterogeneity when starting up. During the simulation, ANU randomization promptly responded to the workload bursts around time 22 and 28 by remapping considerable amounts of workload among



(a) Number of file sets moved during the synthetic workload simulation.



(b) Number of file sets moved during the DFSTrace simulation.

Fig. 9. Load movement.

servers. The virtual processor system also moved about 10 file sets in total, at time 24 and 26, to cope with the workload shift around time 22. However, it failed to deal with the abrupt workload shift around time 28. It was unable to redistribute enough workload among servers to absorb the burst and, therefore, produced high latencies as shown in Figure 6.

Figure 9 confirms that our system preserves load locality when tuning load placement of the system and is able to respond to workload shift promptly. Load preservation is a desirable property for load management in shared-disk clusters as well as in large-scale distributed systems.

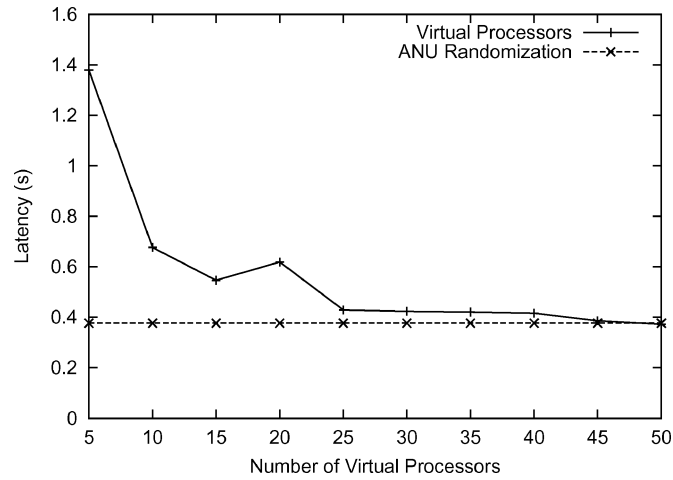


Fig. 10. Performance of virtual processor system.

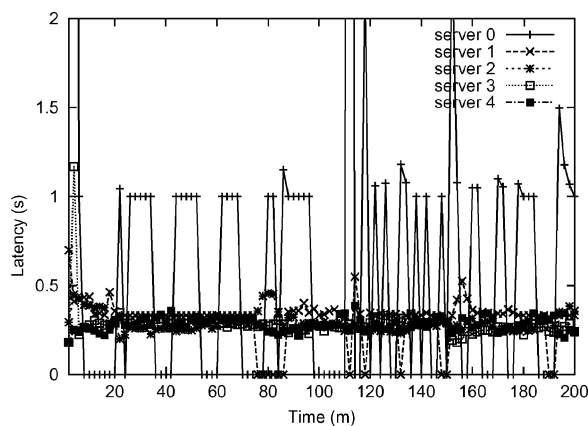
7.5 Comparison with Virtual Processors

The number of virtual processors is an important parameter that defines a trade-off between performance and shared state in virtual processor systems. With a small number of virtual processors, each will be assigned more workload, which increases the size of workload unit. With a large workload unit, it is difficult to perform fine-grained load tuning and assign to each server load proportional to its capacity, making load placement vulnerable to skew and imbalance. In contrast, a large number of virtual processors divides load finely, at the expense of increased state. Because virtual processors do not provide an efficient addressing scheme, it is essential to keep the address information for each individual virtual processor.¹ Considering large clusters consisting of tens of thousand of physical servers, maintaining the shared state for many more virtual processors becomes a serious concern.

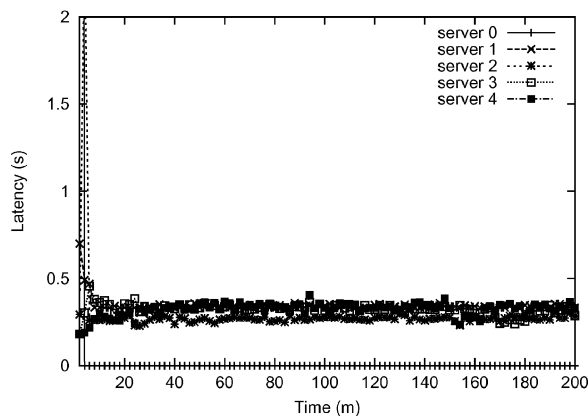
In contrast, the unit interval is the only shared state in ANU randomization. Therefore, it scales with the number of servers. The unit interval contains the information of each server's mapped region and provides an efficient addressing scheme. We hash any given file set's unique name into the unit interval to locate the server that is serving the file set, with possible rehashing if the previous hash falls into unmapped region.

Figure 10 illustrates the trade-off of virtual processor systems. We vary the number of virtual processors from 5 to 50, because we simulate 5 servers and 50 file sets. Each server can be assigned an arbitrary number of virtual processors. With a small number of virtual processors, the virtual processor system does not effectively balance the synthetic workload, yielding poor latency performance for applications. Load placement is greatly improved with a large number of virtual processors, which require more shared state.

¹Addressing information can also be implemented in the Chord ring [Dabek et al. 2001; Stoica et al. 2001], which avoids replication at the expense of $\log(n)$ probes to the data structure.



(a) Initial results exhibit overtuning.



(b) Three heuristics solve the overtuning problem.

Fig. 11. The overtuning problem—before and after.

Figure 10 also directly compares ANU randomization with virtual processor systems. The virtual processor system achieves equivalent performance to ANU randomization when using 45 virtual processors for the 50 file sets. When the number of virtual processors reaches 50, the virtual processor system outperforms ANU randomization on latency and performs comparably to the dynamic prescient system. As each virtual processor contains only one file set on average, we expect this result. The comparison in Figure 10 shows that virtual processor systems need to maintain a shared state that scales with the number of file sets to achieve similar performance to that of ANU randomization.

7.6 Overtuning

Early versions of our algorithm exhibited overtuning—an undesirable side-effect of its aggressiveness in perfecting load balance. Figure 11 shows the overtuning problem on our synthetic workload. The weakest server (server 0)

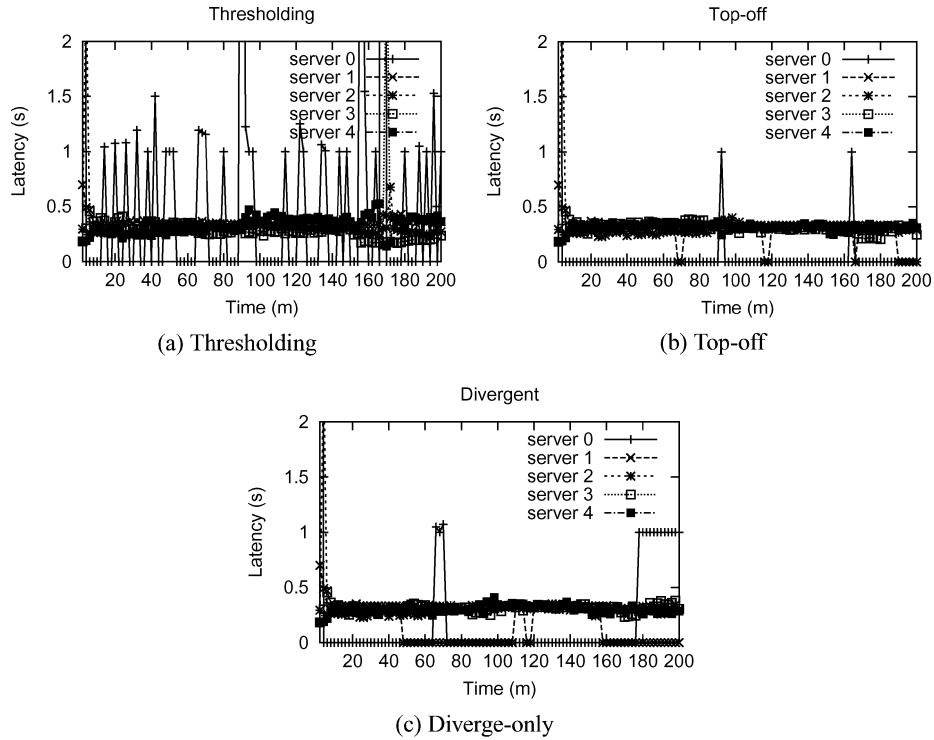


Fig. 12. The three techniques to solve overtuning.

cyclically takes on workload, exhibits high latency, releases workload, and goes to zero latency. Server heterogeneity accounts for overtuning in this case. The three heuristics—thresholding, top-off, and divergent tuning—solve the problem.

In Figure 12, 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 in server 1 in Figure 12(a), which is more stable than in Figure 11(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 most of the time (Figure 12(b)), and the second least powerful server fluctuates modestly, from having no load 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 less stably than all three policies combined.

8. CONCLUSIONS

We have described a load management and server provisioning system based on adaptive, nonuniform randomization. ANU randomization addresses several performance issues in heterogeneous, shared-disk clusters. It preserves the scalability and addressing properties of simple randomization, while avoiding the associated load-skew and inability to handle heterogeneity.

Experimental results indicate that ANU randomization deals with heterogeneity in both server and workload, and performs comparably to a prescient system. This is true for both trace-driven and synthetic workloads with extreme heterogeneity. The results also demonstrate that ANU randomization maintains performance consistency, minimizes load movement, and provides all the load balancing benefits of virtual processor systems with less shared state.

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 migrated from server to server. This includes Web servers, clustered databases, and NFS servers. ANU randomization achieves load balance quickly, without 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

- AMIRI, K., PETROU, D., GANGER, G. R., AND GIBSON, G. A. 2000. Dynamic function placement for data-intensive cluster computing. In *Proceedings of the 2000 USENIX Annual Technical Conference*.
- ANDERSON, D., CHASE, J., AND VAHDAT, A. 2000. Interposed request routing for scalable network storage. In *Proceedings of the Symposium on Operating Systems Design and Implementation*.
- ANDERSON, T. E., CULLER, D. E., AND PATTERSON, D. 1995. A Case for NOW (Network of Workstations). *IEEE Micro* 15, 1 (Feb.), 54–64.
- AVERSA, L. AND BESTAVROS, A. 2000. Load balancing a cluster of web servers using distributed packet rewriting. In *Proceedings of the IEEE International Performance, Computing, and Communications Conference*.
- BARAK, A., LA'ADAN, O., AND SHILOH, A. 1999. Scalable cluster computing with MOSIX for Linux. In *Proceedings of the 5th Annual Linux Expo*. Raleigh, N.C. (May), 95–100.
- BARAK, A., SHAI, G., AND WHEELER, R. G. 1993. *The MOSIX Distributed Operating System: Load Balancing for UNIX*. Springer Verlag.
- BECKER, D. J., STERLING, T., SAVARESE, D., DORBAND, J. E., RANAWAKE, U. A., AND PACKER, C. V. 1995. BEOWULF: A parallel workstation for scientific computation. In *Proceedings of the 24th International Conference on Parallel Processing*. Oconomowoc, WI (Aug.), 11–14.
- BELL, G. AND GRAY, J. 2001. High Performance Computing: Crays, Clusters and Centers. What Next? Technical Report MSR-TR-2001-76, Microsoft Research.
- BRAAM, P. J. 2002. The Lustre storage architecture. Technical Report available at—<http://www.lustre.org/docs.html>.
- BRINKMANN, A., SALZWEDEL, K., AND SCHEIDELER, C. 2002. Compact, adaptive placement strategies for non-uniform capacities. In *Proceedings of Symposium on Parallel Algorithms and Architectures*.

- CARDELLINI, V., COLAJANNI, M., AND YU, P. 2000. Geographic load balancing for scalable distributed web systems. In *Proceedings of the International Symposium on Modeling, Analysis and Simulation of Computer and Telecommunication Systems*.
- CHASE, J., GRIT, L., IRWIN, D., MOORE, J., AND SPRENKLE, S. 2003. Dynamic virtual clusters in a grid site manager. In *Proceedings of the International Symposium on High-Performance Distributed Computing*.
- CZAJKOWSKI, K., FOSTER, I., KESSELMAN, C., SANDER, V., AND TUECKE, S. 2002. Snap: A protocol for negotiating service level agreements and coordinating resource management in distributed systems.
- DABEK, F., KAASHOEK, M. F., KARGER, D., MORRIS, R., AND STOICA, I. 2001. Wide-area cooperative storage with CFS. In *Proceedings of the 18th ACM Symposium on Operating Systems Principles*.
- EAGER, D. L., LAZOWSKA, E. D., AND ZAHORJAN, J. 1986. Adaptive load sharing in homogeneous distributed systems. *IEEE Trans. Softw. Eng.* 12, 5.
- FENG, W., WARREN, M., AND WEIGLE, E. 2002. Honey, I shrunk the Beowulf! In *Proceedings of the International Conference on Parallel Processing*.
- GANGER, G. R., WORTHINGTON, B. L., HOU, R. Y., AND PATT, Y. N. 1993. Disk subsystem load balancing: Disk striping vs. conventional data placement. In *Proceedings of the International Conference on System Sciences*.
- HARCHOL-BALTER, M. AND DOWNEY, A. 1997. Exploiting process lifetime distributions for dynamic load balancing. *ACM Trans. Comput. Syst.* 15, 3.
- HOWARD, J. H., KAZAR, M. L., MENEES, S. G., NICHOLS, D. A., SATYANARAYANAN, M., SIDEBOTHAM, R. N., AND WEST, M. J. 1988. Scale and performance in a distributed file system. *ACM Trans. Comput. Syst.* 6, 1.
- HUANG, C., LAWLOR, O., AND KALE, L. V. 2003. Adaptive MPI. In *Proceedings of the 16th International Workshop on Languages and Compilers for Parallel Computing*.
- IBM. 2003. Total Storage SAN File System. Available at <http://www.storage.ibm.com/software/virtualization/sfs/>.
- KALE, L. V., BHANDARKAR, M., AND BRUNNER, R. 2000. Run-time support for adaptive load balancing. In *Proceedings of the 4th Workshop on Runtime Systems for Parallel Programming*.
- KATZ, E. D., BUTLER, M., AND MCGRATH, R. 1994. A scalable HTTP server: The NCSA prototype. *Comput. Netw. ISDN Syst.* 27, 2.
- LI, K. AND DORBAND, J. 1997. A task scheduling algorithm for heterogeneous processing. In *Proceedings of the Symposium on High Performance Computing*.
- LITWIN, W., NEIMAT, M., AND SCHNEIDER, D. A. 1996. LH*—a scalable, distributed data structure. *ACM Trans. Datab. Syst.* 21, 4.
- LU, C. AND LAU, S. 1996. An adaptive load balancing algorithm for heterogeneous distributed systems with multiple task classes. In *Proceedings of the International Conference on Distributed Computing Systems*.
- Lustre 2002. Lustre: A scalable, High-Performance file system. Technical Report available at—<http://www.lustre.org/docs/whitepaper.pdf>. Cluster File Systems.
- MENON, J., PEASE, D. A., REES, R., DUYANOVICH, L., AND HILLSBERG, B. 2003. IBM storage tank: A heterogeneous scalable SAN file system. *IBM Syst. J.* 42, 2.
- MITZENMACHER, M. 1997. On the analysis of randomized load balancing schemes. In *the ACM Symposium on Parallel Algorithms and Architectures*.
- MITZENMACHER, M. 2000. How useful is old information? *IEEE Trans. Parall. Distrib. Syst.* 11, 1.
- MITZENMACHER, M. 2001. The power of two choices in randomized load balancing. *IEEE Trans. Parall. Distrib. Syst.* 12, 10.
- MUMMERT, L. AND SATYANARAYANAN, M. 1996. Long term distributed file reference tracing: Implementation and experience. *Softw. Pract. Exper.* 26, 6.
- NEDELJKOVIC, N. AND QUINN, M. J. 1992. Data-parallel programming on a network of heterogeneous workstations. In *Proceedings of the 1st International Symposium on High Performance Distributed Computing*.
- PANASAS. 2003. Shared Storage Cluster Computing. (Oct.).
- PARALLEL PROGRAMMING LAB. 2004. *The CHARM++ Programming Language Manual*, UIUC, 5.8 Ed. UIUC.

- PETRI, S. AND LANGENDORFER, H. 1995. Load balancing and fault tolerance in workstation clusters migrating groups of communicating processes. *ACM SIGOPS Oper. Syst. Rev.* 29, 4.
- PRESLAN, K. W., BARRY, A. P., BRASSOW, J. E., ERICKSON, G. M., NYGAARD, E., SABOL, C. J., SOLTIS, S. R., TEIGLAND, D. C., AND O'KEEFE, M. T. 1999. A 64-bit, shared disk file system for Linux. In *Proceedings of the IEEE Mass Storage Systems Symposium*.
- RAO, A., LAKSHMINARAYANAN, K., SURANA, S., KARP, R., AND STOICA, I. 2003. Load balancing in structured P2P systems. In *Proceedings of the 2nd International Workshop on Peer-to-Peer Systems*.
- RATNASAMY, S., FRANCIS, P., HANDLEY, M., KARP, R., AND SHENKER, S. 2001. A scalable content addressable network. In *Proceedings of ACM SIGCOMM 2001*.
- ROWSTRON, A. AND DRUSCHEL, P. 2001. 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*.
- SATYANARAYANAN, M., KISTLER, J. J., KUMAR, P., OKASAKI, M. E., SIEGEL, E. H., AND STEERE, D. C. 1990. Coda: A highly available file system for a distributed workstation environment. *IEEE Trans. Comput.* 39, 4.
- SCHMUCK, F. AND HASKIN, R. 2002. GPFS: A shared-disk file system for large computing clusters. In *Proceedings of the Conference on File and Storage Technology*.
- SHENOY, P. AND VIN, H. 1997. Efficient striping techniques for multimedia file servers. In *Proceedings of the International Workshop on Network and Operating System Support for Digital Audio and Video*.
- SINHA, S. AND PARASHAR, M. 2001. Adaptive runtime partitioning of AMR applications on heterogeneous clusters. In *Proceedings of the IEEE International Conference on Cluster Computing*.
- STOICA, I., MORRIS, R., KARGER, D., KAASHOEK, M. F., AND BALAKRISHNAN, H. 2001. Chord: A scalable peer-to-peer lookup service for Internet applications. In *Proceedings of ACM SIGCOMM*.
- THEKKATH, C. A., MANN, T., AND LEE, E. K. 1997. Frangipani: A scalable distributed file system. In *Proceedings of the ACM Symposium on Operating System Principles*.
- WALSH, D., LYON, B., SAGER, G., CHANG, J., GOLDBERG, D., KLEIMAN, S., LYON, T., SANDBERG, R., AND WEISS, P. 1985. Overview of the Sun network file system. In *Proceedings of the 1985 Winter Usenix Technical Conference*. (Jan.).
- WATTS, J., RIEFFEL, M., AND TAYLOR, S. 1998. Dynamic management of heterogenous resources. In *Proceeding of the High Performance Computing Conference: Grand Challenges in Computer Simulation*.
- WATTS, J. AND TAYLOR, S. 1998. A practical approach to dynamic load balancing. *IEEE Trans. Parall. Distrib. Syst.* 9, 3.
- WILLEBEEK-LEMMAIR, M. AND REEVES, A. P. 1993. Strategies for dynamic load balancing on highly parallel computers. *IEEE Trans. Parall. Distrib. Syst.* 4, 9.
- WU, C. AND LAU, F. 1997. *Load Balancing in Parallel Computers: Theory and Practice*. Kluwer Academic Publishers, Boston, MA.
- ZHOU, S. 1988. A trace-driven simulation study of dynamic load balancing. *IEEE Trans. Softw. Engin.* 14, 9.
- ZHOU, S., WANG, J., ZHENG, X., AND DELISLE, P. 1993. Utopia: A load-sharing facility for large heterogeneous distributed computing systems. *Softw. Pract. Experi.* 23, 12.
- ZHU, H., YANG, T., ZHENG, Q., WATSON, D., IBARRA, O. H., AND SMITH, T. R. 2000. Adaptive load sharing for clustered digital library servers. *Inter. J. Digit. Librar.* 2, 4.

Received August 2004; revised August 2004; accepted August 2004