

A Dynamic Migration Algorithm for a Distributed Memory-Based File Management System *

James Griffioen, Todd A. Anderson
Department of Computer Science
University of Kentucky
Lexington, KY 40506

Yuri Breitbart
Bell Laboratories
600 Mountain Ave.
Murray Hill, NJ 07974

Abstract

Conventional migration strategies attempt to evenly balance the load across all available server machines. This paper discusses why conventional migration approaches are not necessarily appropriate for distributed memory-based file systems and presents an alternative approach that spreads data (possibly unevenly) across as few machines as possible and involves other available machines only as needed. The main advantage of our approach is that it keeps the system minimally distributed thereby reducing the failure rate among servers, the communication overhead among servers, the time needed to compute data relocation, distributed addressing costs, and the probability of unanticipated migrations (e.g., caused by, and an inconvenience to, returning users).

1. Introduction

Database and file storage systems have historically suffered from high access latencies. In fact, I/O latency is one of the major reasons application performance has not improved at the same rate as processor speeds [15]. In our previous work [11], we introduced a distributed memory-based file system (MBFS) that used the idle memory and processor cycles of a network of workstations to provide persistent storage with read and write latencies more than an order of magnitude faster than conventional disk-based storage systems. Other researchers have proposed similar systems that improve read latency (as opposed to read/write latency) by treating remote memory as additional file cache space [2, 6, 3, 7, 12].

This paper builds on our past work by examining the issue of data migration in a distributed memory-based storage system. We introduce a new data migration algorithm that

differs drastically from conventional migration algorithms used to load balance processes in a distributed system [19] or file accesses in a disk-based distributed storage system [21, 22].

Conventional process migration algorithms and load balancing algorithms attempt to (1) spread the load across all available machines, and (2) balance the load as evenly as possible. Failure to use all machines means wasted CPU cycles. In addition, if one processor is more heavily loaded than another processor, performance may suffer. Thus balancing is important. The same goals apply to disk-based file systems that allow file migration. For example, the snowball disk-based distributed file system [21, 22] showed the importance of balancing, noting that a highly balanced load (e.g., a maximum imbalance of 0.5%) performs three times better than a slightly imbalanced load (e.g., a maximum imbalance of 5%). Achieving a carefully balanced load is computationally expensive and keeping the load balanced typically means frequent migration.

Our previous work [11] illustrated two unique characteristics of distributed memory-based storage systems that differentiate them from their disk-based counterparts. These differences significantly affect the goals we set for a memory-based migration algorithm.

First, disk-based file server performance is primarily determined by client request rate. An increased request rate means longer disk queuing delays at the server resulting in higher response times. Although memory-based servers are affected by client request rate, the amount of data stored on a server also influences a server's performance. If the amount of data being accessed exceeds the server's memory capacity, page swapping increases and as a result, the server's performance can degrade by an order of magnitude or more. Consequently, memory-based servers are characterized by two loads rather than one. *CPU load* represents the number of requests processed per second by the CPU and *memory load* represents the amount of *active data*¹

*This work supported in part by NSF grant numbers IRI-92121301, CCR-9309176, CDA-9320179, and CDA-9502645.

¹Active data refers to the file data being actively accessed by the current

stored at the server. If the CPU is busy 100% of the time or if the server's memory space is 100% utilized we say the server has "reached its capacity" and is "saturated". Exceeding the memory capacity is potentially more dangerous than exceeding the CPU capacity because CPU overloads only cause performance to degrade slowly while memory overloads cause disk paging that can degrade performance by an order of magnitude or more.

A second finding from our previous work is that if two different servers are both executing below their capacity, both servers will have approximately the same average response time, regardless of their specific loads. When a server is executing below its capacity, CPU queuing delays and memory access times are insignificant compared to the network latency. Because network latency dominates a server's response time, the specific load on the server has little impact on performance. This is not true of disk-based systems. In disk-based systems, disk latency and disk queuing delays dominate server response time. Thus, small changes in request rate can severely affect a disk-based server, even if it is lightly loaded.

Memory-based storage systems can, of course, use disks for additional storage space if necessary. However, if the active file system data exceeds the aggregate physical memory capacity of the system and disk swapping ensues, queuing delays caused by high disk latencies will begin to dominate server response time just like disk-based storage systems. To differentiate disk-based models from memory-based models, we assume that memory-based storage systems have sufficient aggregate memory capacity to hold a file system's active data. Because the active data is primarily memory-resident, the number of disk faults will not be sufficient to affect average server response time.

A final characteristic of our memory-based system that impacts the migration algorithm is the fact that the system uses *idle* memory and processing power. The location of idle resources in the system will change (over time) for various reasons (e.g. users returning to their machines after an absence). If a machine undergoes a transition from idle to active, MBFS must relinquish the resources it has borrowed on that machine and find alternate idle resources. Anticipating transitions is difficult because user behavior is typically unpredictable. Consequently, transitions should be avoided if at all possible. When they do occur, other idle resources must be found quickly to minimize the inconvenience to the users and applications.

In light of our two previous observations and the goal to avoid inconvenience to client applications, this paper presents a new migration algorithm that attempts to keep the system minimally distributed while allowing load imbalances. Keeping the system minimally distributed reduces the failure rate among servers, the communication over-

head among servers, the distributed addressing costs, and the chance of a migration occurring because of unanticipated transitions from idle to active. Despite the minimal distribution and load imbalances, the system exhibits performance similar to that of a maximally distributed fully balanced system.

head among servers, the distributed addressing costs, and the chance of a migration occurring because of unanticipated transitions from idle to active. Despite the minimal distribution and load imbalances, the system exhibits performance similar to that of a maximally distributed fully balanced system.

2. Related Work

Remote memory systems have been investigated in a variety of contexts. The following briefly comments on related remote memory systems, and, when possible, describes the migration mechanism used. We also briefly mention past work in the area of load balancing.

2.1. Remote Memory Systems

Comer and Griffioen [2] introduced the *remote memory model* in which client machines accessed the memory resources of one or more dedicated *remote memory servers*. Client machines that exhaust their local memory capacity move virtual memory data to a remote server's memory and retrieve data on demand. Only dedicated remote server memory was accessible to clients. Each client's memory was private and inaccessible even if it was idle. Data migration between servers was not supported.

Felten and Zahorjan [5] enhanced the remote memory model to use idle client machines for backing store instead of dedicated memory servers. Idle client machines advertise their available memory to a centralized registry. Clients needing more space contact the centralized registry and randomly pick one of the idle clients returned by the registry. Like [2], data was not migrated among servers.

Dahlin et al. [3] describe an algorithm called N-Chance Forwarding that manages a cooperative cache. N-Chance Forwarding tries to keep as many different pages in global memory as it can by showing a preference for *singlets* (single copies of a block in some client's memory) over multiple copies. Duplicate pages, chosen for replacement, are simply discarded because the cache only stores *clean* (unmodified) pages. A singlet is forwarded from machine to machine N times before being discarded. Since the forwarding destination is picked randomly, forwarded pages could be sent to heavily loaded clients or even non-idle machines with no available memory.

Feeley et al. [7] describe the Global Memory Service (GMS) system that uses per node page age information to approximate global LRU on a cooperative cache. Like N-Chance Forwarding, GMS's page replacement algorithm only stores *clean* pages and does not consider a client's CPU or memory load when deciding the movement or replacement of pages.

Hartman and Sarkar [16] present a *hint-based* cooperative caching algorithm. Previous work such as N-Chance Forwarding [3] and GMS [7] maintain *facts* about the location of each block in the cooperative cache. Although block location hints may be incorrect, the low overhead needed to maintain hints outweighs the costs of recovering from incorrect hints. Should hints be missing or incorrect, a client can always retrieve a block from the server, to which all write requests are sent. Using hints, block migration is done in a manner similar to that of GMS [7].

The Harp file system [12] runs on a set of dedicated server nodes and uses memory, UPS, and replication to ensure persistence and availability. MBFS uses UPS in a similar fashion. However, MBFS also supports server expansion and contraction and data migration.

Franklin et al. [6] use remote memory to cache distributed database records and move data around using an algorithm similar in nature to that of N-chance forwarding. Client load was not considered by the data migration mechanism. Several other researchers have proposed non-distributed memory-resident database designs [8] which do not have to deal with data migration.

2.2. Load Balancing

Distributed process scheduling and load balancing have been studied by many researchers [19, 18]. All of these systems focus on placing processes on machines in such a way that all processors are continuously busy and the load on each processor is roughly equivalent. For most distributed applications, maintaining a balanced load is crucial in order to optimize performance. Even the slightest imbalances can severely degrade performance. Other systems treat tasks as a work heap with processors seeking-out tasks. This assumes that any processor can service any task. In MBFS, each server is responsible for only a portion of the storage space and cannot serve an arbitrary request. For each client request, MBFS uses a dynamic addressing scheme [11, 22, 13] to find the address of the server that can satisfy the request.

The topic of *disk balancing* was addressed in Snowball [22], a distributed disk-based database. Snowball demonstrated that small load imbalances (disk utilization across servers differed by only 5%) increased average server response time by a factor of three over highly-balanced loads (disk utilization differences of 0.5%). As we will show in the following sections, balanced loads that are crucial for disk-based systems are not crucial for memory-based systems.

3. A Memory-Based Storage Model

The Memory-Based File System (MBFS) uses the idle memory found in a network of workstations to store file data. In the following, we briefly outline the architecture of the MBFS system. Details of the system can be found in [11].

The MBFS system consists of general purpose workstations connected via a high-speed network. Although the network can be arbitrarily large, made up of one or more local area networks, we assume the latency between any two machines in the network is small (e.g., at least one order of magnitude faster than typical disk latencies). Users execute general purpose programs at random times on the network of workstations. We call any application executed on a workstation a *client application*. Although MBFS provides basic file storage, we will assume in this paper that the primary client application is a distributed database that needs high-performance file storage. An MBFS *server* process executes on every machine that has idle capacity. Together the servers provide the storage space for the MBFS memory storage system. MBFS servers have the lowest priority and are essentially “guests” of the machine they borrow resources from.

At any given time, each workstation operates as an MBFS client, server, neither, or both, and may change roles over time. When clients need the local machine’s resources, the server component will shrink or disappear. When no clients are active, the server will acquire the idle resources. In this way, the system dynamically adjusts to alternating periods of activity and inactivity.

The primary role of the MBFS servers is to keep all active file data memory resident, and to respond to client requests without incurring any disk accesses. MBFS stores both modified and unmodified data. Servers ensure long-term persistence of modified data by propagating newly written or modified records to their disk in the background. To ensure short-term persistence without accessing disks, MBFS temporarily writes modified data to a limited number of special server machines equipped with Uninterruptable Power Supplies called *WUPS* (Workstations with UPS). In the event of a power failure, the UPS gives the workstation adequate time to write the memory contents to disk and shutdown, ensuring that all data is reliably stored and can be recovered. Recovery from server crashes are handled with techniques such as those proposed in [1].

MBFS assigns each file to exactly one server. A server where the file is stored is called the *primary location* of the file. To find the primary location of a file, MBFS uses an addressing function that allows the address table to be dynamically expanded or contracted. Each entry in the table maps a “bucket” of file block IDs to the server where the block is stored. Our current addressing algorithm is based on dynamic hashing with multiple hash-levels [11, 22, 13].

Our dynamic addressing algorithm has several advantages. First, it minimizes the table size. Second, it allows the table size to grow or shrink in response to the addition or removal of new servers. Third, buckets in the table can be split to create smaller buckets in order to obtain a more reasonably balanced load. Finally, timestamps can be used to merge address tables on different machines to create a more up-to-date copy of the address table. Note that the primary location of a file may change over time as a result of migration. In our original MBFS system, migrations occurred only as a result of server overload. The system had no ability to expand or contract the number of servers. The migration algorithm described in this paper addresses the issue of workstations that alternate between active and idle periods, thereby causing servers to come and go over time. Our new algorithm dynamically adds new servers to the system as machines become idle and migrates data off of machines whose client applications reclaim previously idle resources.

A fundamental difference between the MBFS architecture and other remote memory systems is the fact that MBFS assumes the file system's "working set" (i.e., *active* file data) fits easily within the aggregate idle memory space of the system. All other file data (i.e. *inactive* data) is placed on disk. We base our assumption on the rapid proliferation of PCs and workstations and the decreasing cost of memory. Many industrial and academic settings already have a significant number of networked computers, the majority of which sit idle most of the time [4, 14]. In such settings, machines with hundreds of megabytes of physical memory are becoming common. Consequently, we expect that distributed systems with aggregate memory capacities of tens or hundreds of gigabytes will be common in the very near future.

To differentiate between active and inactive data, MBFS currently uses a simple time-based mechanism. If data has not been accessed within a predefined amount of time Δ , we assume the data is no longer active and does not need to be stored in memory. The parameter Δ can be defined using an updated version of Gray and Putzolo's 5 minute rule [9], but other active data definitions based on concepts like file system working sets [20] could be used as well. When inactive data is accessed again, hopefully via MBFS's automatic prefetching mechanism [10], the data will again be marked as active and placed in memory storage.

4. Migration Algorithm

The fact that MBFS assumes a massive distributed memory storage space (i.e., memory is the primary storage media) has several implications that are not valid of other remote memory systems [7, 16, 3]. These differences played an important role in the design of our migration algorithm.

First, because MBFS can differentiate between active and inactive data, it reports space used to store inactive data

as available memory instead of viewing it as "used memory". This increases the perceived available memory. Second, assuming the active data fits easily into idle memory implies that we do not expect all memory to be in use storing active data. If a machine becomes overloaded, idle resources can be found elsewhere in the system to absorb the load. Third, the system does not need to go to great lengths to ensure that memory is used efficiently. For example, the system does not need to develop complex/costly algorithms that remove duplicates or eliminate fragmentation in order to save space. Fourth, we do not need to use all available memory in the system, but instead can afford to leave some memory idle (i.e., unused and unmanaged) until it is needed.

Given these differences, we developed a set of design objectives specific to migration in a memory-based file system. The following sections outline our design goals and present our dynamic migration algorithm.

4.1. Design Goals

We identified the following design goals for our memory-based migration algorithm:

1. A server should only migrate data when saturation is imminent or when triggered by an unexpected decrease in idle resources on the machine.
2. The cost of achieving a perfectly balanced load should be considered in the light of expected performance improvements resulting from a better balance.
3. The algorithm should migrate data to as few machines as possible.
4. The algorithm should migrate data to machines least likely to be used in the near future.
5. Load redistribution should not result in migration thrashing.
6. The algorithm must consider both memory and processor load when migrating.

Our primary objective was to design a migration algorithm that prevents memory servers from becoming saturated. As described in section 1, the performance of a saturated server degrades rapidly, in some case by an order of magnitude or more. Consequently, the algorithm must ensure that server saturation does not occur. Note, this means a server does not need to get rid of load if it is not saturated or about to be saturated (goal 1).

Balancing the load among servers and keeping the load balanced is not crucial to memory-based systems. In a disk-based system, imbalances of as little as 5% can result in a 200% difference in response time between servers [22], whereas imbalances of 50% only produce differences of 11% between servers in memory-based systems [11]. Moreover, achieving a perfectly balanced load will add overhead

that in turn affects system performance. In memory-based systems, the response time improvements resulting from a balanced load are typically so small that they hardly justify achieving a perfectly balanced load. Thus, it is important that MBFS consider the cost/benefit ratio when deciding whether the load balancing algorithm should continue searching for a better solution (goal 2).

In a disk-based storage system, spreading the load across the maximal number of disks (servers) produces the best response times. In a memory-based system, a subset of servers will perform approximately as well as the maximal number of servers as long as no server becomes saturated [11]. This means that the migration algorithm can select a minimal number of servers and achieve performance similar to a system that uses the maximum number of servers (goal 3). To prevent migrations from occurring in the future, the algorithm should avoid sending data to machines that are likely to become active in the near future or to machines that are already near their capacity (goal 4).

Systems that attempt to constantly keep the load balanced often result in frequent (albeit small) migrations with data ping-ponging between machines (goal 5). Finally, unlike conventional load balancing systems that only consider CPU load, a memory-based migration algorithm must consider both the CPU load and the memory load (goal 6).

4.2. Determining When To Migrate

Conventional load balancing algorithms attempt to keep the load evenly distributed among all machines and thus invoke the load balancing algorithm whenever unacceptable load imbalances arise. Keeping the system balanced at all times is not one of MBFS's design goals. Instead, MBFS only wants to prevent servers from reaching such imbalanced loads that it could affect the average response time. This only occurs when a server reaches either its CPU or memory capacity.

Consequently, the MBFS migration algorithm introduces the concept of a *danger level* to indicate that server saturation is imminent. To identify potential server saturation, the system defines two dangers levels: one for the CPU load and one for the memory load. If either load reaches its danger level, migration is invoked to remove the overload.

The CPU danger level is defined as a percentage of the CPU's capacity. Although performance will degrade if the CPU exceeds its capacity, CPU loads slightly greater than 100% of capacity do not significantly affect the server's response time because network latency, not CPU queuing delays, still dominates a server's response time. Thus, we typically set the CPU danger level to be 100% of the CPU's capacity.

Unlike CPU overloads, memory overloads result in high latency disk accesses and queuing delays that can quickly

degrade a server's performance by several orders of magnitude. Given the enormous performance penalty for memory overload, we must ensure that memory load never approaches its capacity. On the other hand, setting the danger level too low wastes valuable memory space and can also degrade performance. To avoid these two extremes, MBFS sets the danger level as high as possible such that the server is still able to migrate data off before the growth rate causes saturation. We define the memory danger level and other parameters below.

$$MemDangerLevel = TotalMem - (\alpha \times (TimeNeededToMigrate \times GrowthRate))$$

TimeNeededToMigrate: the amount of time it takes to migrate data off of the server

TimeNeededToMigrate \times *GrowthRate*: the expected number of bytes that will be added during the migration

α : a padding constant to provide additional time ($\alpha \geq 1$).

To prevent thrashing, MBFS uses a *safe level* in conjunction with the danger level. If the load is below the safe watermark we assume the machine is underloaded and can handle additional load. The danger level represents an overloaded state while the safe level represents an underloaded state. The migration algorithm will not push any machine over its safe level. This ensures that the new load on the receiving machine will remain sufficiently below the danger level preventing any migrations in the near future.

4.3. Selecting Servers To Take The Load

Although, MBFS allows any machine to play the role of client or server, we expect future distributed systems will still categorize some machines as "servers only" for reasons of protection, security, centralization, resource sharing, and guaranteed performance. Consequently, MBFS classifies every machine as either a *restricted access* machine (which will primarily act as a server) or a *general access* machine (which will act as a client, server, both, or neither). MBFS assumes that restricted access machines are more reliable and have minimal general client application activity.

Machines acting as MBFS servers will be functioning in one of the three modes: *active* mode (the server currently stores file data), *idle* mode (the server's memory is not currently used to hold file data), or *inactive* mode (the server cannot hold file data). Thus the set of machines capable of accepting additional load are:

Restricted Access Servers: A restricted access machine with an active MBFS server.

Restricted Access Idlers: A restricted access machine with an idle MBFS server.

General Access Servers: A general access machine with an active MBFS server.

General Access Idlers: A general access machine with an idle MBFS server.

The system obtains load information from other machines at runtime via a receiver-initiated approach [18] and classifies each machine according to the above definitions.

In keeping with our design goals, the MBFS migration algorithm migrates data to as few servers as possible and selects machines that are unlikely to be used in the near future. That is, restricted access machines are preferred over general access machines, and fewer machines are preferred over many.

The migration algorithm selects server machines as follows. First, the system calculates the load that must be migrated off of the overloaded server. The algorithm then adds machines to a list until the amount of unused resources in the list can absorb the overload (i.e. unused memory space and CPU cycles below the safe levels). The algorithm adds machines to this list in the order (1) Restricted Access Servers, (2) General Access Servers, (3) Restricted Access Idlers, and (4) General Access Idlers. A parameter M defines the number of Idler machines to add each time through the loop. Setting $M = 1$ results in a minimal set of servers while settings of $M > 1$ cause servers to be added in groups, hopefully producing lower loads on all servers and increasing the time period between migrations. The server selection algorithm is shown in Figure 1a.

The above selection approach has several advantages. First, although a user's personal workstation will volunteer its unused resources, MBFS's migration algorithm will not use the resource unless absolutely necessary. Algorithms that spread data across all machines increase the probability that a newly started client application will have to wait while data migrates off the machine. The increased probability of migration also implies an increase in the processing and network load used to handle the migrations. Using fewer servers reduces the number of migrations as well as the likelihood that a client application will have to wait. Second, file data has an affinity for restricted access machines because they tend to be more reliable and predictable. In particular, they are less susceptible to accidental power cycles, crashing, or load variations resulting from client applications. Third, spreading the data across fewer machines lessens the likelihood that some part of the file storage system will be inaccessible. Tending towards a more centralized approach reduces the mean-time-to-failure of the overall storage system. Fourth, spreading the load across many machines produces larger address tables because the data is partitioned into many smaller pieces. Finally, our previous work showed that a few highly loaded servers offer similar performance as many lightly loaded servers [11]. Consequently, a minimal number of servers has approximately the same performance as the maximal number of servers.

```

SelectServers {
  calculate Overload
  add all Restricted Access Servers to ServerList
  if (available resources in ServerList > Overload) {
    compute data placement
    migrate data
    return
  }
  add all General Access Servers to ServerList
  if (available resources in ServerList > Overload) {
    compute data placement
    migrate data
    return
  }
  while (Restricted Access Idlers Remain) {
    add M largest Restricted Access Idlers to ServerList
    if (available resources in ServerList > Overload) {
      compute data placement
      migrate data
      return
    }
  }
  while (General Access Idlers Remain) {
    add M largest General Access Idlers to ServerList
    if (available resources in ServerList > Overload) {
      compute data placement
      migrate data
      return
    }
  }
}
1(a)

ComputeDataPlacementSchedule {
  while (no migration schedule has been found) {
    Revert to original machine load info
    Sort bucket list (by appropriate load metric)
    For each bucket (largest to smallest) {
      Find the server (X) that would have the
        smallest load after the addition
        of this bucket
      if (X's load over its safe-level)
        goto SPLIT
      else
        add bucket to X's load info
    }
    Compute Max and Min Server loads
    if (Max load - Min Load < split tolerance)
      return migration schedule

  SPLIT:
    Remove largest bucket (Y) from bucket list
    Split Y into L_SPLIT sub-buckets
    Add sub-buckets to bucket list
  }
}
1(b)

```

Figure 1. (a) The selection algorithm used to identify servers with available capacity. (b) The algorithm used to compute a data placement schedule.

4.4. Computing a Data Placement Schedule

Although load balancing is not a goal of the MBFS migration algorithm, balanced loads do give the best server response times and can decrease the frequency of migration occurrences. Consequently, the MBFS migration algorithm attempts to achieve an approximately balanced load whenever migration is necessary. To regulate “how balanced the load should be”, we introduce a *tolerance* parameter, T , that defines the maximum allowable difference between the maximum server’s load and the minimum server’s load. Note that the tolerance parameter T does not determine when migration occurs, nor does the migration algorithm guarantee that it will keep the load balanced within tolerance T . It only defines how close the migration algorithm should come to finding a balanced load when migration is necessary, and thereby defines the amount of effort the algorithm is willing to expend achieving a balanced load.

The objective of the tolerance parameter T is to limit the amount of effort spent balancing the load in light of the benefits obtained from a more balanced load. Small T (e.g., 0.1%) result in well-balanced loads that will hopefully increase the amount of time before another migration is needed. A tolerance of 100% provides no balancing guarantees. When T is 100% the algorithm only guarantees that each machine involved in the migration will remain below its safe level.

The data placement algorithm is given in Figure 1b. To minimize the size of the address table used by MBFS’s dynamic addressing algorithm, the data placement algorithm begins by selecting entire buckets for migration to other servers. If no distribution of the existing buckets satisfies the tolerance parameter T , the largest bucket is split into L_SPLIT buckets, where L_SPLIT is defined as a parameter of MBFS’s addressing algorithm. After splitting the largest bucket, the process is restarted and continues until an acceptable data placement schedule is achieved.

During each iteration, the entire list of buckets managed by the migrating server is sorted by memory load if memory saturation has occurred or by CPU load if processor saturation has occurred. The algorithm begins by setting the load on the migrating server to zero and then tries to distribute the entire list of buckets (likely placing some buckets back on the migrating server). Each bucket from the list is successively placed on the machine with the smallest resulting load, assuming that the machine’s safe levels are not exceeded. This continues until the entire load has been distributed or until the current bucket cannot be placed. If the load was distributed and the tolerance was satisfied, the algorithm is done. If the tolerance was not satisfied or the current bucket cannot be placed, the algorithm splits the largest bucket (creating L_SPLIT smaller buckets) and starts over.

L_SPLIT is an important parameter to the algorithm be-

cause it allows us to define how quickly the address table size grows and also how fast the placement algorithm converges. Our results, described in section 6, show that the L_SPLIT level has a significant effect on how fast the data placement algorithm converges at low split tolerance values (5% and below) but has little effect on system performance at higher split tolerance values (10% and above).

5. Simulation Model

We incorporated the MBFS file system and migration algorithm into the SunOS kernel and ran some small-scale experiments on a real system of workstations. Although the prototype provides proof of concept for the design of a real system, the limited nature of our experimental environment prohibits us from experimenting with larger number of machines and machines with different memory and processing capabilities. For these reasons we developed a simulation model to evaluate the performance of MBFS in a wider range of potential environments. The following only reports on results obtained from our simulation model.

The simulation model begins with an “empty system” in which each client starts with an address table that directs all file accesses to a single MBFS server. The initial MBFS server is a restricted access machine and is part of a simulated network of N machines with memory capacities ranging from 16 MB to 512 MB. A simulation configuration file defines the number of machines N , how many machines will be restricted access machines, how many will be general access machines, and how much idle memory each machine has. Table 1 lists the simulation parameters used in our tests.

The simulator consists of two parts: a file access generator and a migration system. To generate migrations resulting from memory overload, a file access generator simulates insert and delete operations on file blocks (the only operations that affect the memory load imposed on a server). The file access generator runs until a server’s memory becomes overloaded. While the results reported here use file accesses taken from a normal distribution [17], we have simulated several other synthetic file access distributions and compared these with traces from our prototype. The nature of the distributed hashing algorithms is that it distributes accesses evenly across the MBFS storage space. Consequently, the file access distribution, whether a simulated distribution or a real file access trace taken from our prototype, had little effect on the system’s performance.

When a server becomes overloaded, the simulator invokes the migration component of the simulation model to migrate the load. The migration component invokes the *SelectServers* algorithm and the *ComputeDataPlacementSchedule* algorithm to produce a data placement schedule. The simulator then transfers the load from the overloaded server to the other servers and updates the addressing

tables accordingly. At this point, the file access generator is invoked again and the cycle continues. Consequently, each simulation involves a long series of insert-migrate iterations. The simulation ends after a fixed number of inserts or if the inserted data exceeds the aggregate memory storage capacity of the system currently being simulated.

The simulator maintains several statistics including the number of migrations, the cost of each migration, final address table sizes, the precision of load balancing, etc. The following section presents these results.

Algorithm parameter	Value(s)
M - Number of idlers to add	3
T - Load tolerance	0.1% - 100%
L_SPLIT - # of sub-buckets in a bucket	3 - 7
Environment parameter	Value(s)
N - Number of machines	100
Big memory machines (256-512 MB)	5% of N
Medium memory machines (64-128 MB)	70% of N
Small memory machines (16-32 MB)	25% of N
# of restricted access machines	10% of N
# of general access machines	90% of N

Table 1. Simulation Model Parameters

6. Simulation Results

The migration algorithm trades balancing precision for faster computation of the data placement schedule. Intuitively, it would appear that if the algorithm generates a less balanced load, the frequency of migrations could increase. The more unbalanced the load, the sooner another server may become overloaded. Therefore, the balancing precision should be selected in such a way as to avoid the side effect of additional migrations.

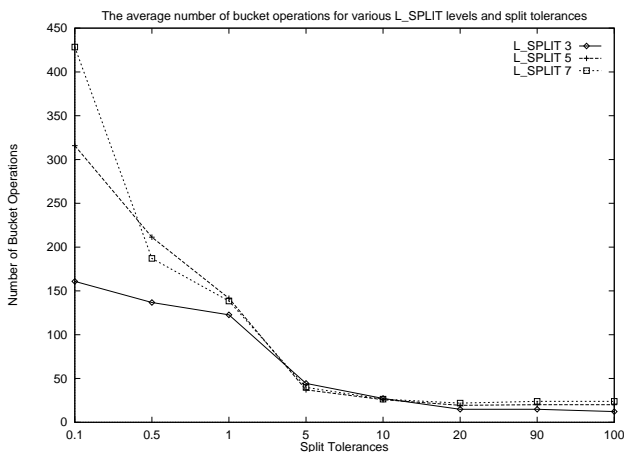


Figure 2. Average time needed to compute a data placement as measured in iterations of the for loop in Figure 1b.

To quantify this tradeoff, we measured the average time

needed to compute a data placement schedule and the frequency of migrations that occur during a fixed interval. We only report warm-start results that were recorded after the system had been sufficiently primed and all start-up effects were eliminated. We measure the computation time as the number of bucket operations (i.e., the number of iterations of the for loop in Figure 1b) that occur while computing the migration schedule. The compute times for various tolerance settings are shown in Figure 2.

As expected, computing a well-balanced load (0.1%) is an order of magnitude more costly than computing a slightly or totally imbalanced load (10% - 100%). However, imbalanced loads do not increase in any meaningful way the frequency of migrations the system experiences (see Figure 3). In fact the number of migrations is occasionally

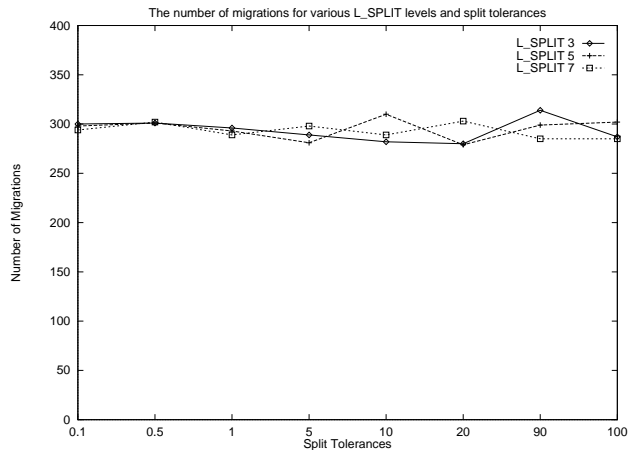


Figure 3. The number of migrations during the test period.

higher for balanced loads than unbalanced loads. This is due in part to the fact that well-balanced loads are more inclined to squeeze into existing servers and keep the server count low than their imbalanced counterparts. Despite the inclination to squeeze into fewer machines, our results show that, on average, well balanced machines (low T values) end up spreading the load across the same number of servers as systems that are not perfectly balanced (high T values).

Figure 4 illustrates how migrations are distributed in time for various tolerance settings. Low tolerances produce a rapid succession of closely-spaced migrations followed by a long period without any migrations (resulting from the addition of a new server). In this case, servers are well-balanced and become saturated at roughly the same time, triggering a rapid series of migrations that impose a significant load on the system. These heavy migrations periods consume a significant amount of network bandwidth and server CPU cycles solely for the purpose of migration. To prevent two (or more) overloaded machines from simultaneously migrating too much data to the same underloaded machine, the system

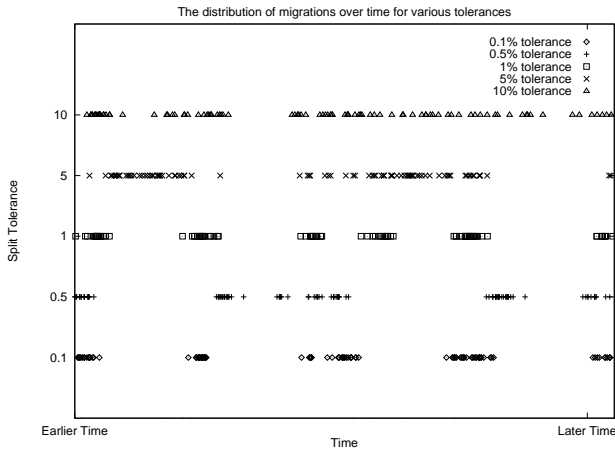


Figure 4. Migration distribution. Each point represents the time at which a migration occurred.

must ensure that migrations are synchronized and coordinated. The synchronization overhead and partial serialization of migrations affects performance when many migrations occur simultaneously. As a result, the average server response time during such heavy migration periods often increases dramatically. On the other hand, when the servers are not perfectly balanced, the servers reach their danger levels at different times and thus spread the cost of migration more evenly over time.

Our results also show that the safe/danger watermark technique used by the algorithm prevents the system from thrashing. That is, the system does not migrate data to machines that are already near their danger level. On average, the machine undergoing migration last received data from another server 26 migrations earlier.

Small address table sizes are desirable for space reasons, but also because fewer hash levels improve the dynamic hashing algorithm's performance. In addition, the delay associated with an addressing error (which involves the transmission of one or more address table correction messages) is typically less. Figure 5 shows the final size of the address tables for various split tolerances.

Note that the tolerance T only represents the maximum "allowable" difference between a heavily loaded server and a lightly loaded server. In practice, the "real" difference will be significantly less than the tolerance T . Figure 6 shows the average load difference the migration algorithm actually achieved at various tolerance levels. Although high tolerance settings could theoretically result in large imbalances, the actually load differences observed at high tolerance settings are quite low ($< 8\%$). In fact, our results show that on average the system can achieve a 10% tolerance with just two iterations of the migration algorithm and fewer than 30

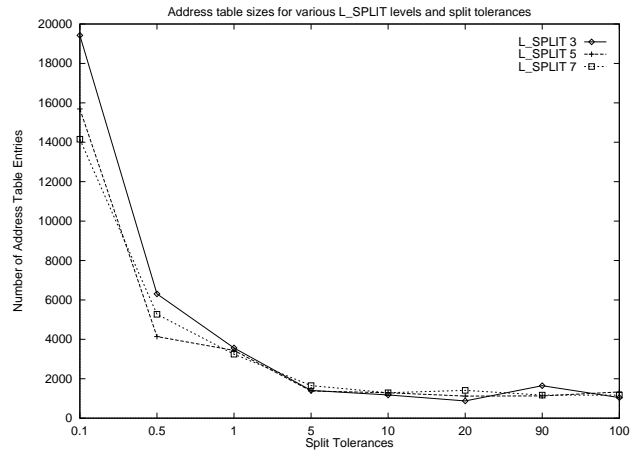


Figure 5. Address table size. Each point represents the size of the address table at the end of the simulation.

bucket operations (see figures 2 and 6).

Our algorithm tries to minimize the number of servers in an attempt to increase the mean-time-between-server-failures and to reduce the possibility of migrations caused by newly active client applications. Because we take a non-conventional approach, we wanted to compare the performance of our system against an unrealistic system where machines do not crash and client applications do not start on idle workstations. In such an environment, optimal performance is achieved by spreading the load across all machines. Consequently, we modified our simulator to spread data across all machines during the first migration. While the modified simulator performed an order of magnitude fewer migrations than the original, the minimal server approach had average server response times at most 12% higher than the modified simulator.

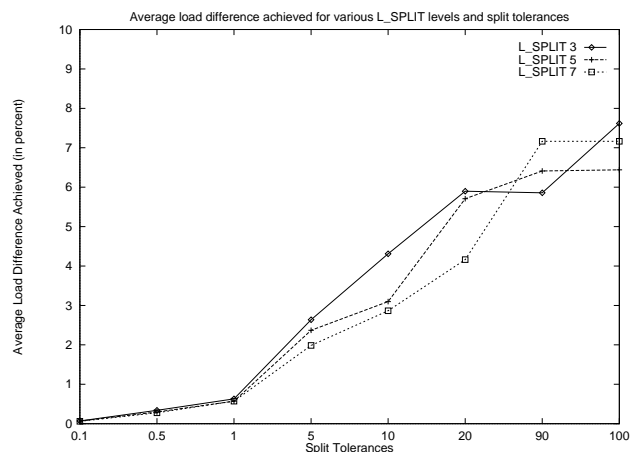


Figure 6. Average load difference achieved.

7. Conclusions

We described a dynamic migration algorithm that limits server involvement and trades poorer precision for better compute times without affecting overall system performance. We show that the algorithm's potentially imbalanced loads do not significantly affect the system's response time, can be computed an order of magnitude faster than balanced loads, do not significantly affect the number of migrations that occur, and reduces addressing costs. The use of a saturation prevention method combined with the ability to expand to other servers keeps the migration algorithm from thrashing.

References

- [1] P. M. Chen, W. T. Ng, S. Chandra, C. Aycok, G. Rajamani, and D. Lowell. The Rio File Cache: Surviving Operating System Crashes. In *1996 International Conference on Architectural Support for Programming Languages and Operating Systems*, 1996.
- [2] D. Comer and J. Griffioen. A New Design for Distributed Systems: The Remote Memory Model. In *The Proceedings of the 1990 Summer USENIX Conference*, pages 127–136. USENIX Association, June 1990.
- [3] M. D. Dahlin, R. Y. Wang, T. E. Anderson, and D. A. Patterson. Cooperative Caching: Using Remote Client Memory to Improve File System Performance. In *Proceedings of the First Symposium on Operating Systems Design and Implementation*, pages 267–280, November 1994.
- [4] K. Efe and V. Krishnamoorthy. Optimal Scheduling of Compute-Intensive Tasks on a Network of Workstations. *IEEE Transactions on Parallel and Distributed Systems*, 6(6):668–673, 1995.
- [5] E. W. Felten and J. Zahorjan. Issues in the Implementation of a Remote Memory Paging System. Technical Report 91-03-09, Department of Computer Science and Engineering, University of Washington, March 1991.
- [6] M. Franklin, M. Carey, and M. Livny. Global Memory Management in Client-Server DBMS Architectures. In *18th International Conference on Very Large Data Bases*, 1992.
- [7] M. Freeley, W. Morgan, F. Pighin, A. Karlin, and H. Levy. Implementing Global Memory Management in a Workstation Cluster. In *Proceedings of the 15th ACM Symposium on Operating Systems Principles*, December 1995.
- [8] H. Garcia-Molina and K. Salem. Main Memory Database Systems. *IEEE Transactions on Knowledge and Data Engineering*, 4(6):509–516, December 1992.
- [9] J. Gray and F. Putzolo. The 5 Minute rule for Trading Memory for Disk Access and 10 Byte Rule for Trading Memory for CPU Time. In *Proceedings of the SIGMOD Conference*, pages 395–398, 1987.
- [10] J. Griffioen and R. Appleton. Reducing File System Latency Using a Predictive Approach. In *The Proceedings of the 1994 Summer USENIX Conference*, pages 197–207. USENIX Association, June 1994.
- [11] J. Griffioen, R. Vingralek, T. Anderson, and Y. Breitbart. Derby: A Memory Management System for Distributed Main Memory Databases. In *The Proceedings of the IEEE 6th International Workshop on Research Issues in Data Engineering (RIDE '96)*, February 1996.
- [12] B. Liskov, S. Ghemawat, R. Gruber, P. Johnson, L. Shrira, and M. Williams. Replication in the Harp file system. In *Proceedings of 13th ACM Symposium on Operating Systems Principles*, pages 226–38. Association for Computing Machinery SIGOPS, October 1991.
- [13] W. Litwin, M.-A. Niemat, and D. Schneider. Rp*: A Family of Order-Preserving Scalable Distributed Data Structures. In *Very Large Data Bases Conference, Santiago, Chili, 1994*.
- [14] M. W. Mutka and M. Livny. Profiling Workstations' Available Capacity for Remote Execution. In *Proceedings of the 12th IFIP WG Symposium on Computer Performance '87*, December 1987.
- [15] J. K. Ousterhout. Why Aren't Operating Systems Getting Faster As Fast as Hardware? In *Proceedings of the Summer 1990 USENIX Conference*, pages 247–256, June 1990.
- [16] P. Sarkar and J. Hartman. Efficient Cooperative Caching using Hints. In *Proceedings of the Second Symposium on Operating Systems Design and Implementation*, pages 35–46, October 1996.
- [17] H. Schwetman. *CSIM Reference Manual (Revision 16)*. Microelectronics and Computer Technology Corporation, 1992.
- [18] M. Singhal and N. Shivaratri. *Advanced Concepts in Operating Systems: Distributed Databases and Multiprocessor Operating Systems*. McGraw Hill, 1994.
- [19] J. M. Smith. A Survey of Process Migration Mechanisms. *ACM Operating Systems Review*, 22(3):28–40, July 1988.
- [20] C. Tait and D. Duchamp. Detection and Exploitation of File Working Sets. In *Proceedings of the 1991 IEEE 11th International Conference on Distributed Computing Systems*, pages 2–9, May 1991.
- [21] R. Vingralek, Y. Breitbart, and G. Weikum. Distributed File Organization with Scalable Cost/Performance. In *ACM SIGMOD Conference*, 1994.
- [22] R. Vingralek, Y. Breitbart, and G. Weikum. SNOWBALL: Scalable Storage on Networks of Workstations with Balanced Load. Technical report, Department of Computer Science, University of Kentucky, 1995.