Scale and Performance in a Distributed File System

John H. Howord et. al.
in ACM Transactions on Computer Systems Vol. 6, No. 1, 1988

Presenter: Changyeon Jo, Jaegyoon Hahm

2013/10/22

Presentation Outline

- Andrew File System Prototype
  - Overview of Andrew File System (AFS)
  - Performance Evaluation Of AFS Prototype
  - Changes For Performance
- Revised Andrew File System
  - Effect of Changes For Performance
  - Performance Comparison of NFS and AFS
  - Changes for Operability
- Conclusion

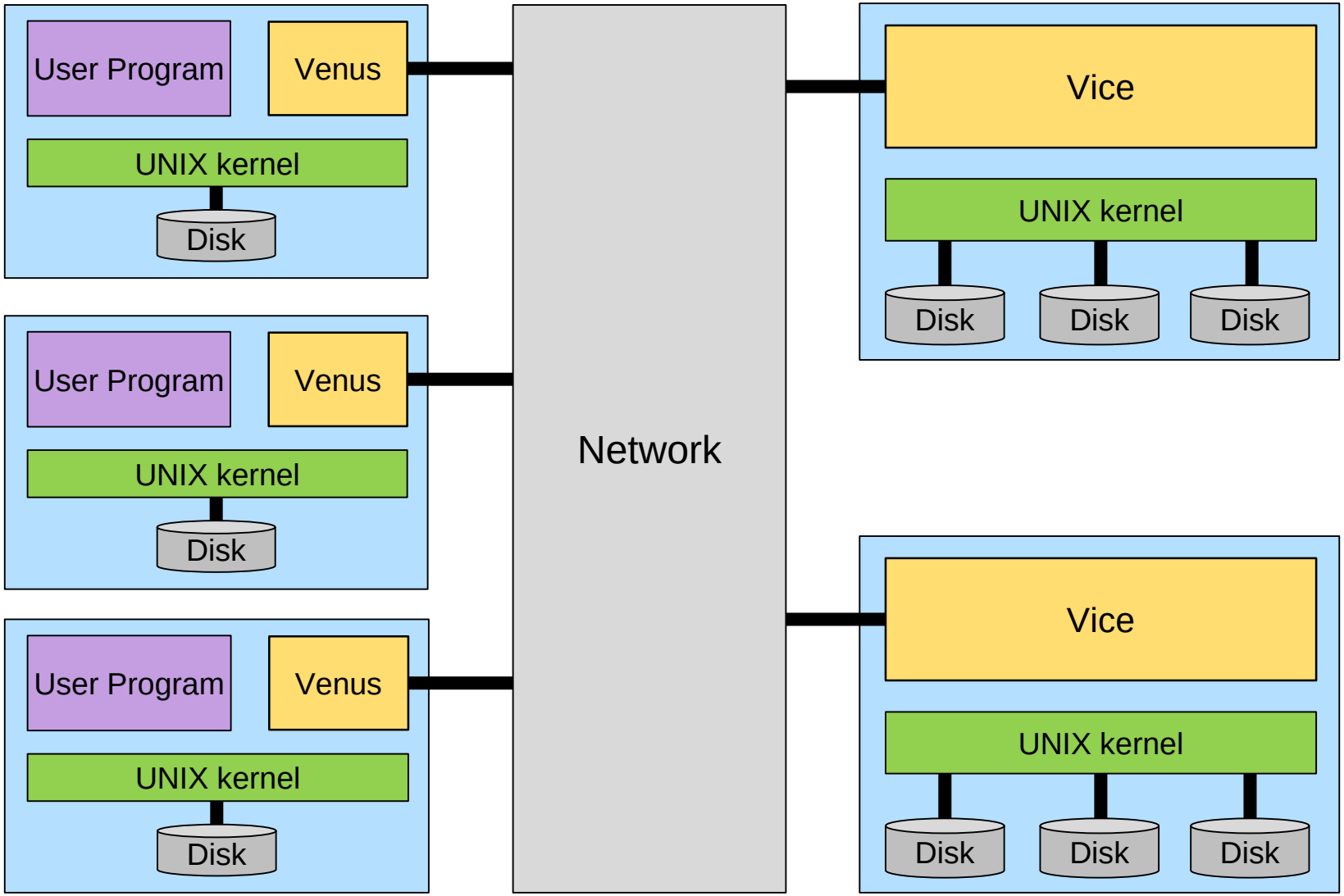
Changyeon Jo

Andrew File System Prototype

Andrew File System

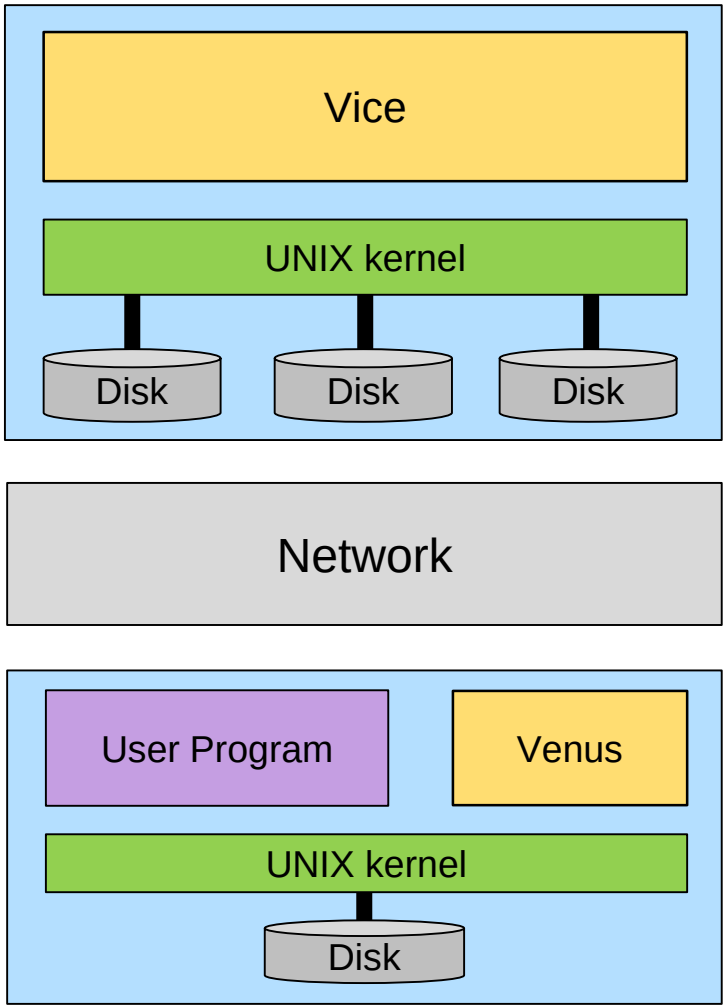
- **Distributed File System** developed in CMU
- Presents a homogeneous, location-transparent file name space
- **Scalability** is most important design goal of AFS
  - Unusual Design Features
    - Whole-file caching
- AFS architecture is based on observations …
  - Shared files are infrequently updated
  - Files are normally accessed by only a single user
  - Local caches are likely to remain valid for long periods

Andrew File System Architecture



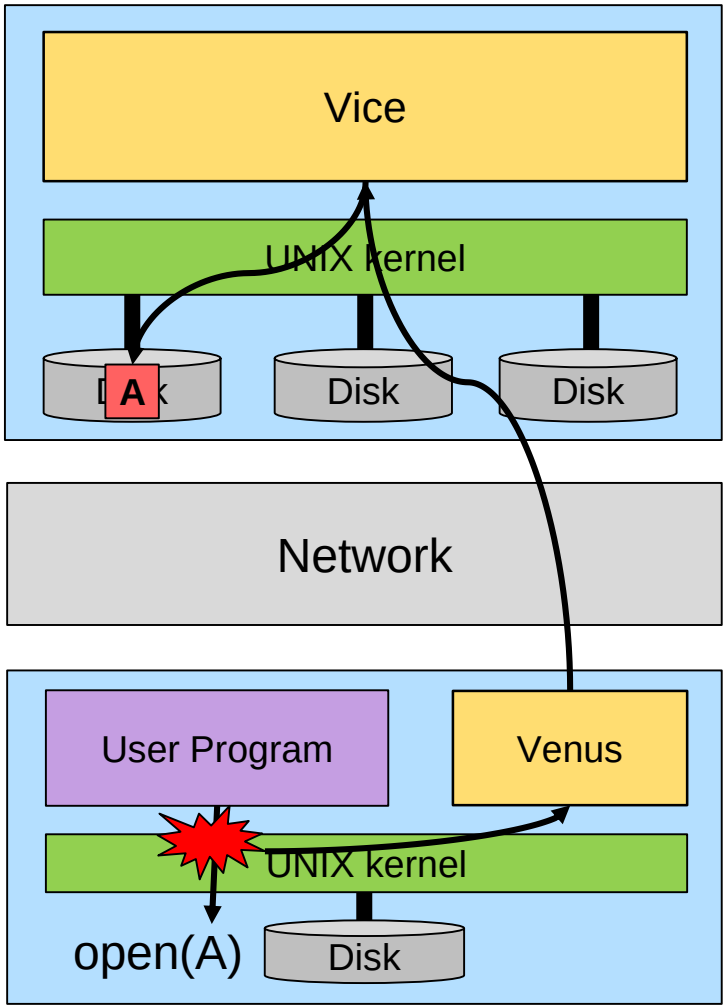
Andrew File System Prototype

- *Vice*
  - Serve files to *Venus*
  - A process running on server side
  - A *Vice* process dedicated to each *Venus* client
- *Venus*
  - Cache files from Vice
  - Contacts Vice only when a file is opened or closed
  - Reading and writing are performed directly on the cached copy



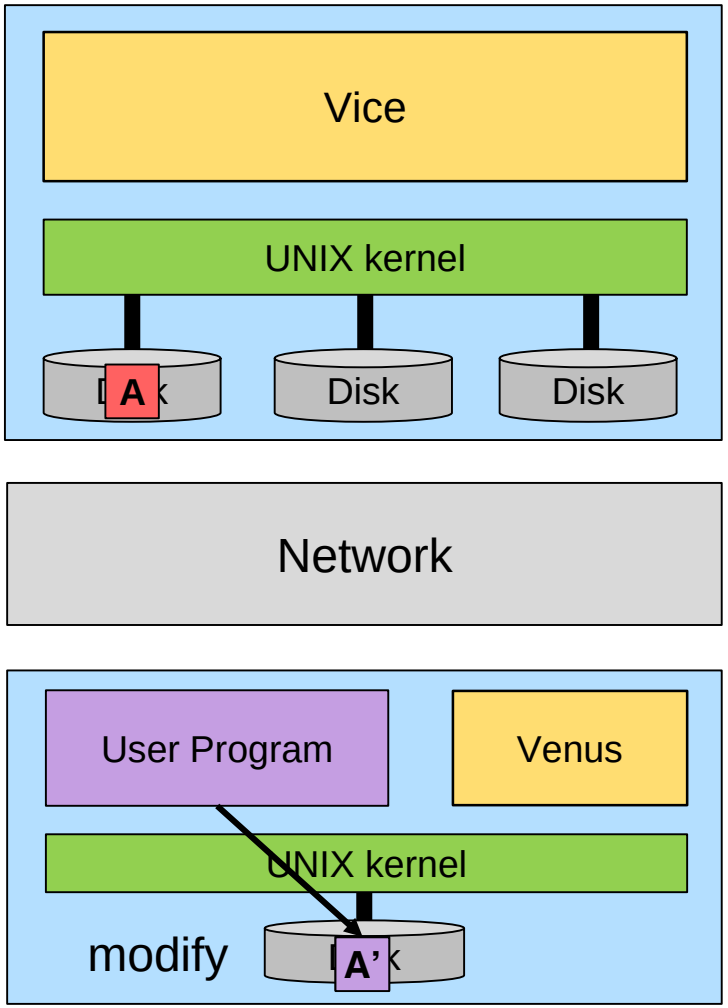
Andrew File System Prototype (cont'd)

- *Vice*
  - Serve files to *Venus*
  - A process running on server side
  - A *Vice* process dedicated to each *Venus* client
- *Venus*
  - Cache files from Vice
  - Contacts Vice only when a file is opened or closed
  - Reading and writing are performed directly on the cached copy



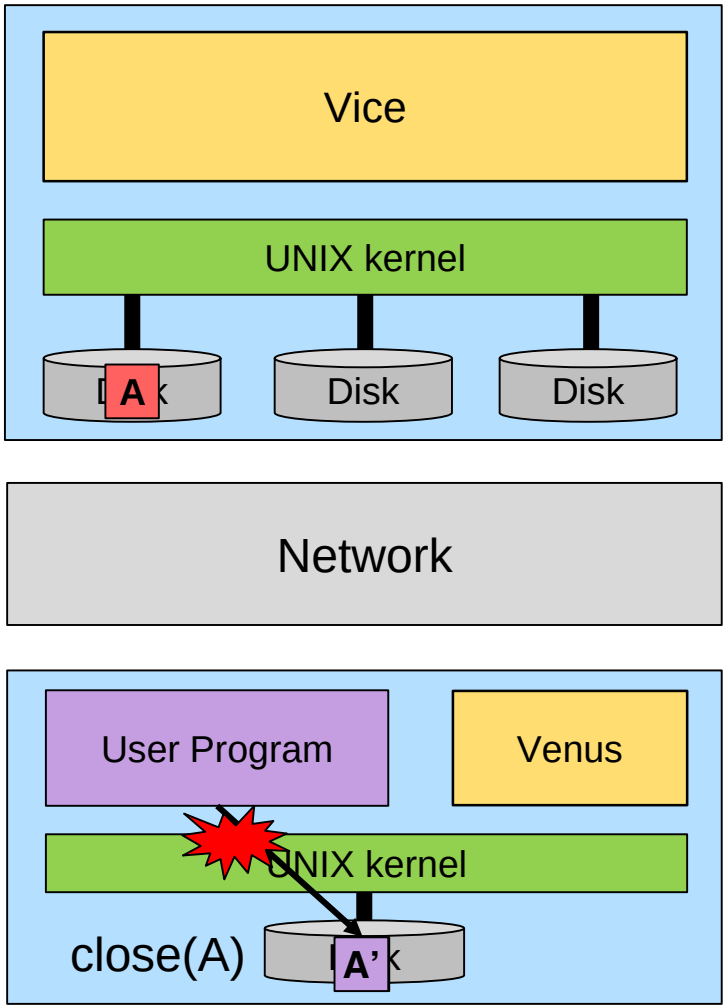
Andrew File System Prototype (cont'd)

- *Vice*
  - Serve files to *Venus*
  - A process running on server side
  - A *Vice* process dedicated to each *Venus* client
- *Venus*
  - Cache files from Vice
  - Contacts Vice only when a file is opened or closed
  - Reading and writing are performed directly on the cached copy



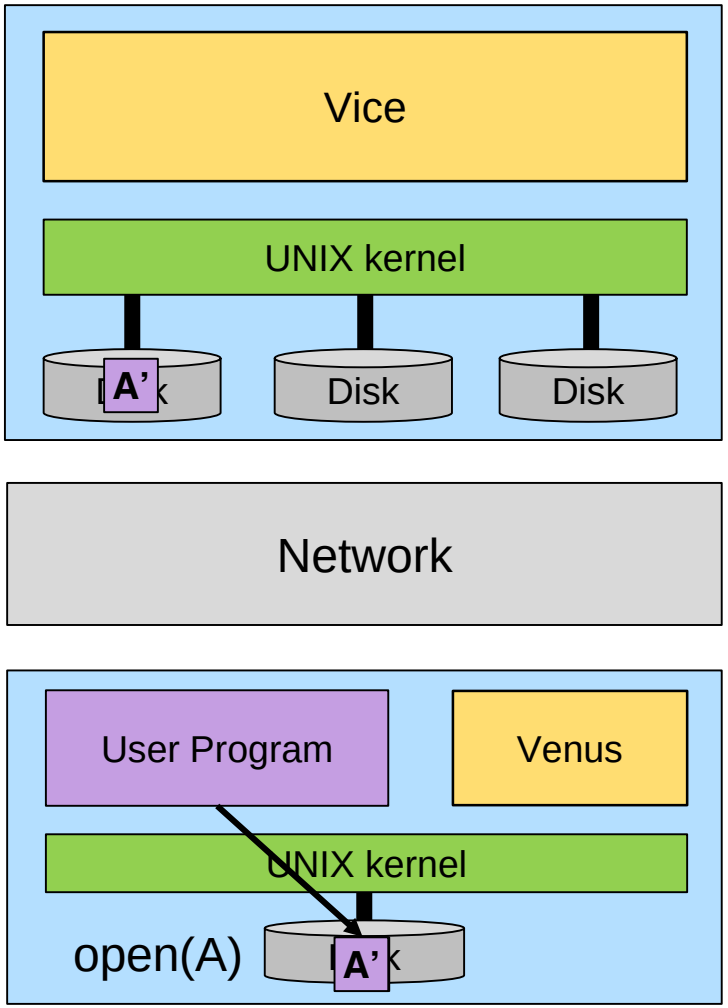
Andrew File System Prototype (cont'd)

- *Vice*
  - Serve files to *Venus*
  - A process running on server side
  - A *Vice* process dedicated to each *Venus* client
- *Venus*
  - Cache files from Vice
  - Contacts Vice only when a file is opened or closed
  - Reading and writing are performed directly on the cached copy



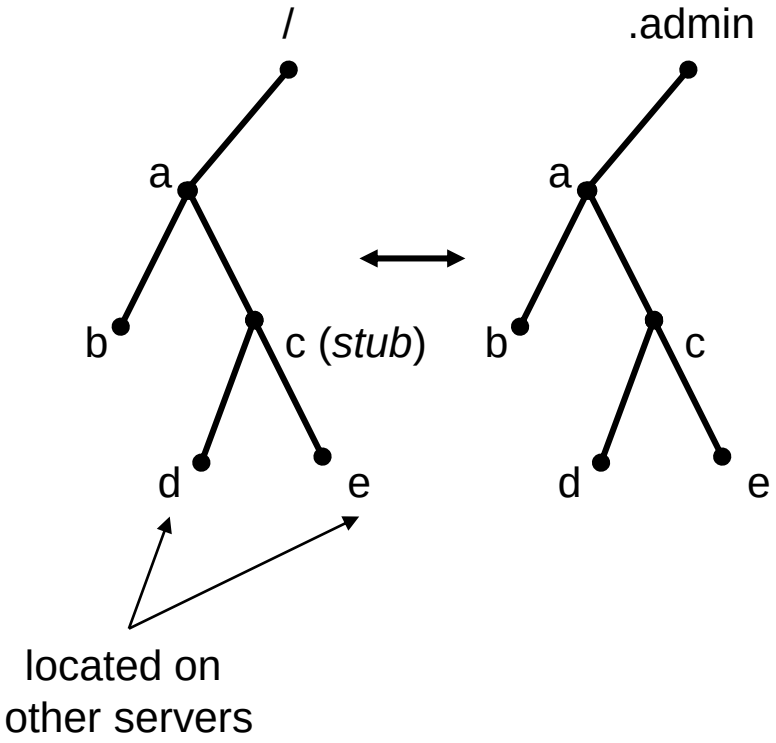
Andrew File System Prototype (cont'd)

- *Vice*
  - Serve files to *Venus*
  - A process running on server side
  - A *Vice* process dedicated to each *Venus* client
- *Venus*
  - Cache files from Vice
  - Contacts Vice only when a file is opened or closed
  - Reading and writing are performed directly on the cached copy



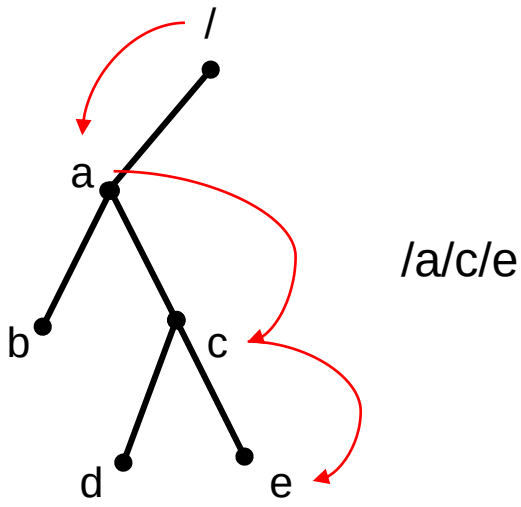
Andrew File System Prototype (cont'd)

- *Vice* maintains status information in separate files
  - *.admin* directory to store configuration data
  - mirroring original *vice* file structures
  - *Stub* directory gives a name space located on other servers



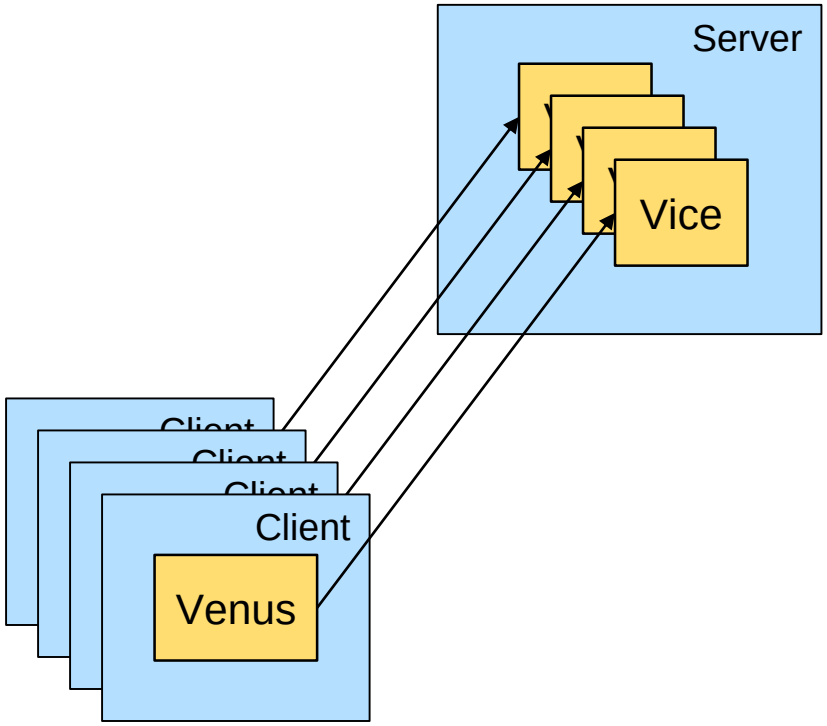
Andrew File System Prototype (cont'd)

- Vice-Venus interface names files by their full pathname
  - there is no notion of low lev el file name such as *inode*
  - full path directory working is required



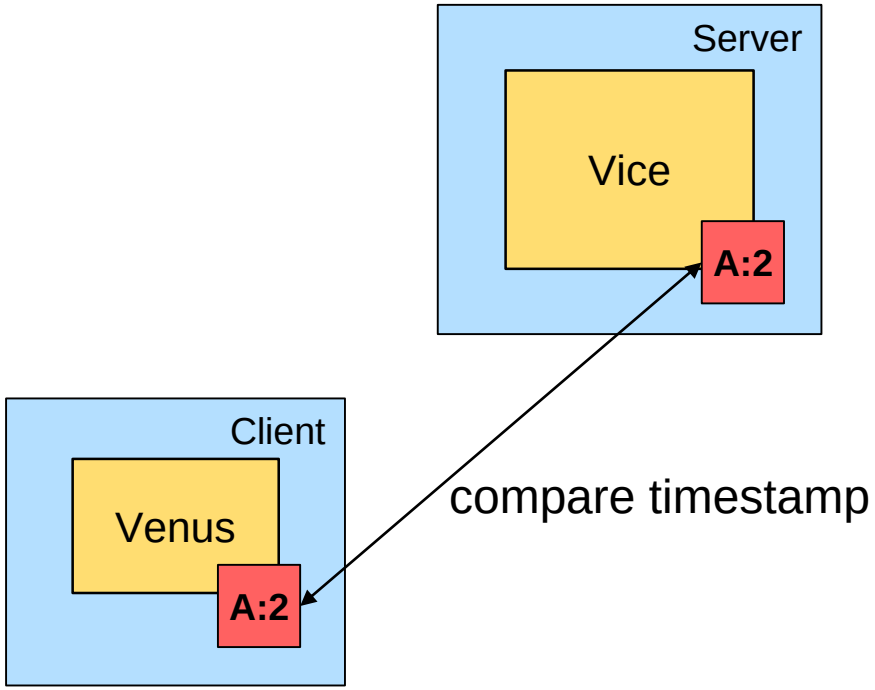
Andrew File System Prototype (cont'd)

- Vice create dedicated processes for each client
  - the dedicated process persisted until its client terminated
  - too frequent context switching



Andrew File System Prototype (cont'd)

- *Venus* verify timestamps on every open
  - Each open include at least one interaction with a server
    - even if the file were already in the cache and ***up-to-date!***

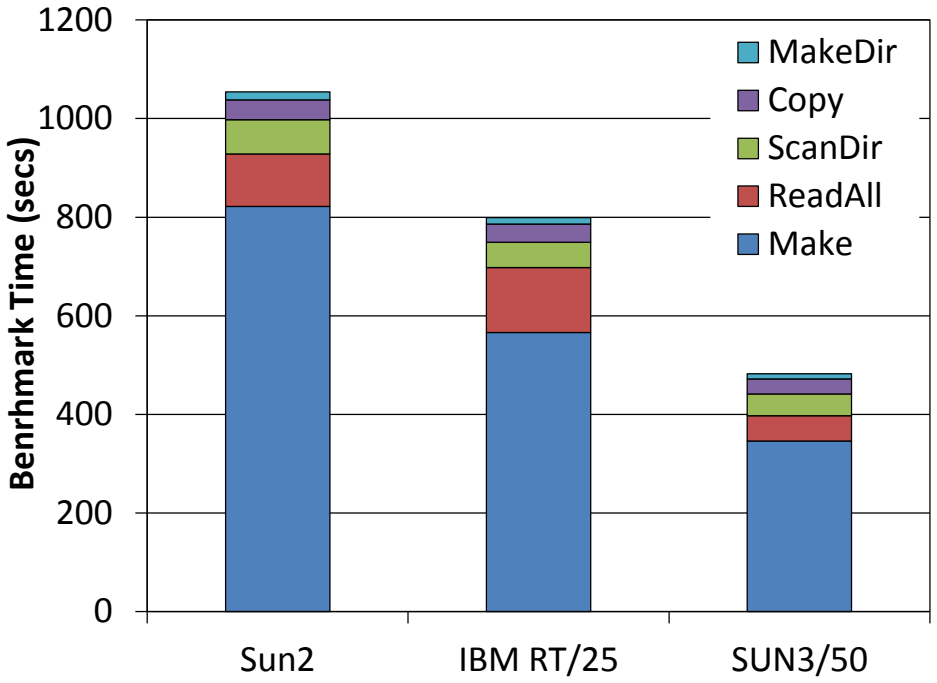


AFS Prototype Benchmark Setup

- *Load Unit*
  - Load placed on a server by a single client
  - A *Load Unit* = about five Andrew users
- Read-only source subtree
  - 70 files
  - Total 200kbytes
- Synthetic Benchmark
  - Simulate real user actions
  - MakeDir, Copy, ScanDir, ReadAll, Make

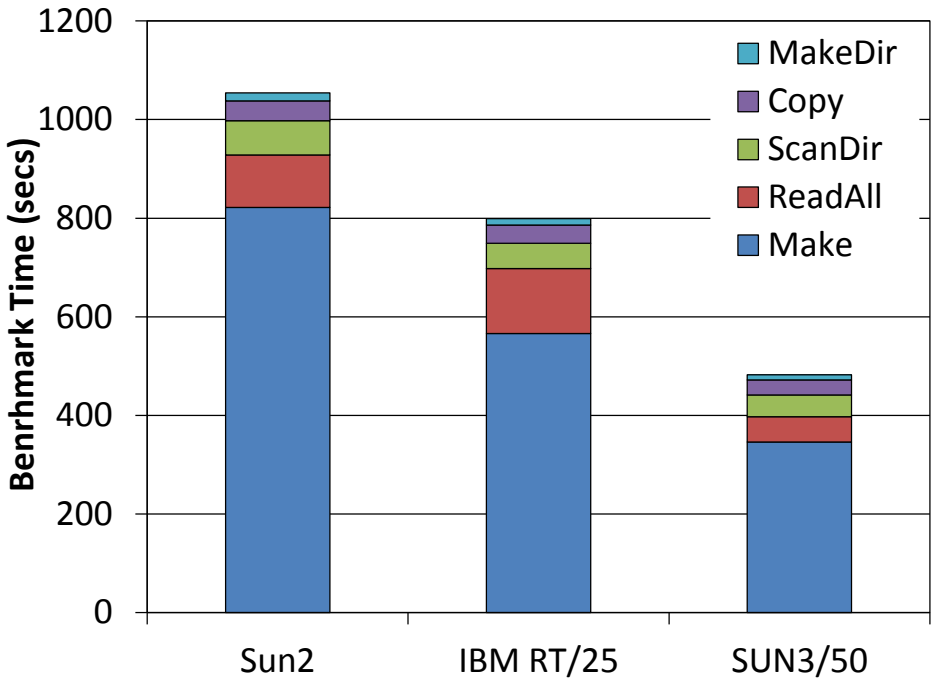
Stand-alone Benchmark Performance

- ***MakeDir***: Constructs a target subtree that is identical in structure to the source subtree
- ***Copy***: Copies every file from the source subtree to the target subtree
- ***ScanDir***: Recursively traverses the target subtree and examines the status of every file in it
- ***ReadAll***: Scans every byte of every file in the target subtree once
- ***Make***: Compiles and links all the files in the target subtree



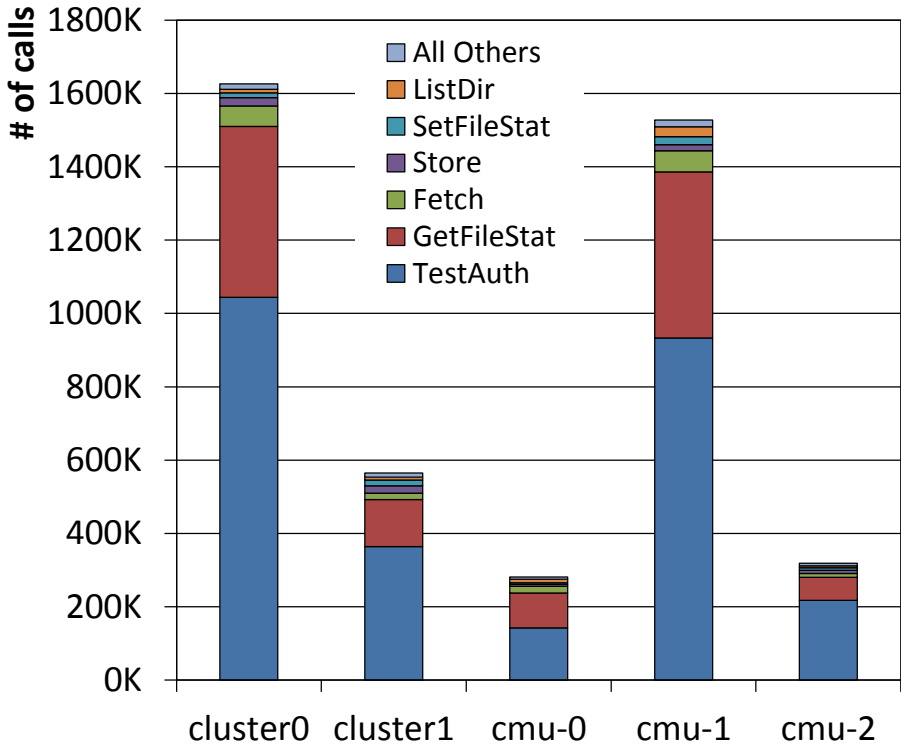
Stand-alone Benchmark Performance

- Hit Ratio of Two Caches
  - 81% for file cache
  - 82% for status cache



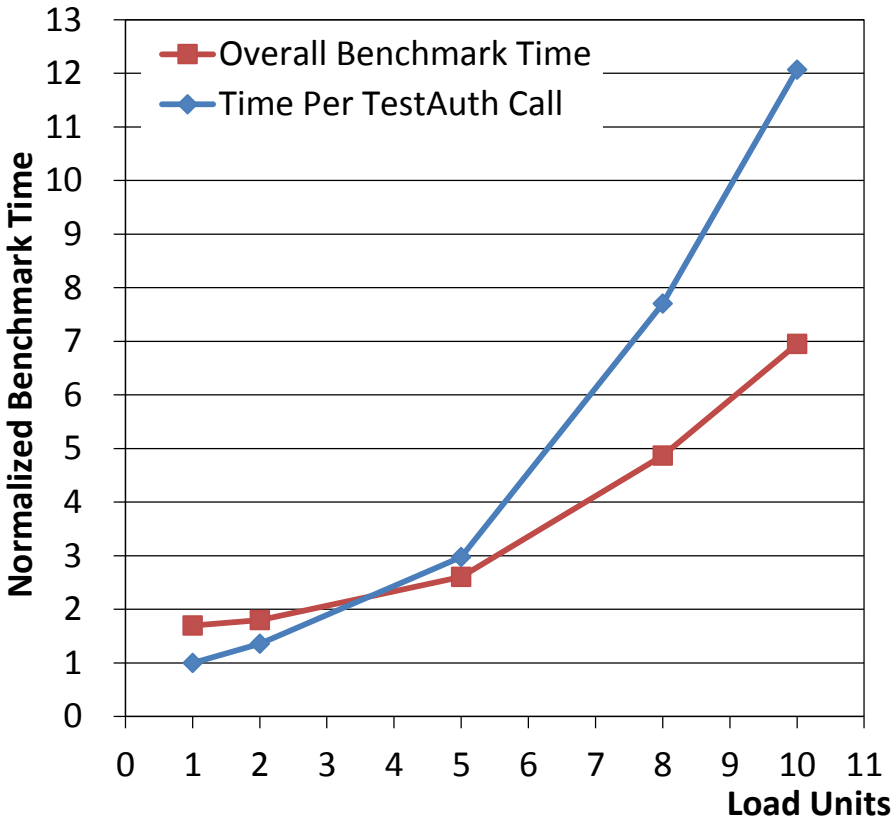
Distribution of Vice Calls in Prototype

- *TestAuth* and *GetFileStat* are accounting for nearly **90%** of the total calls!
- Caused by frequency of cache validity check



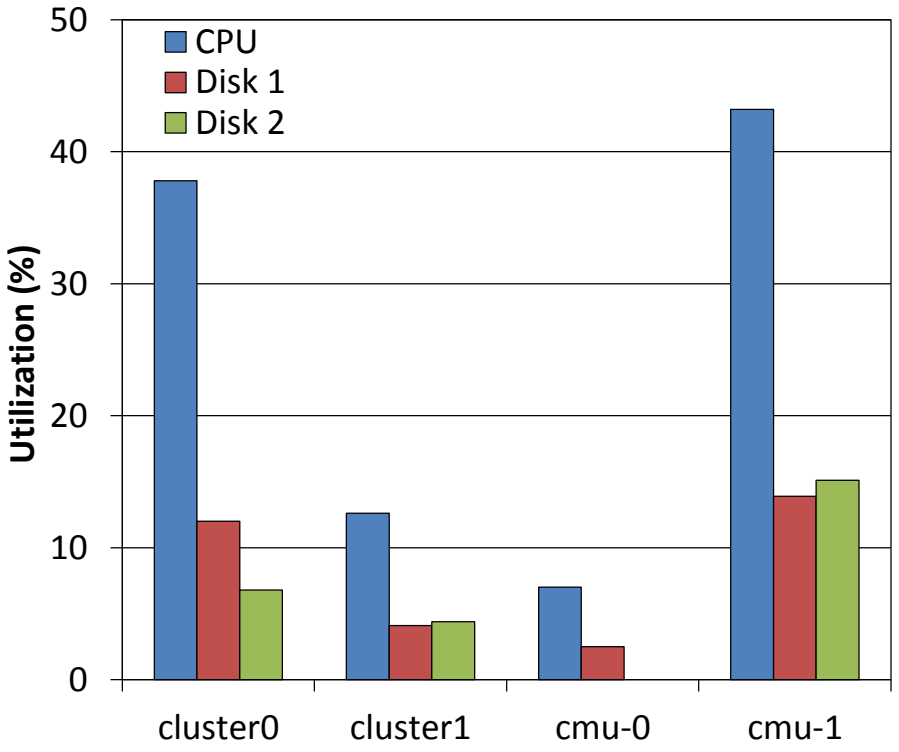
Prototype Benchmark Performance

- *Took* about **70%** longer at a load of 1 than in the stand-alone case
- *TestAuth* rose rapidly beyond a load of **5**
- Running only between **5** and **10** servers is the maximum feasible



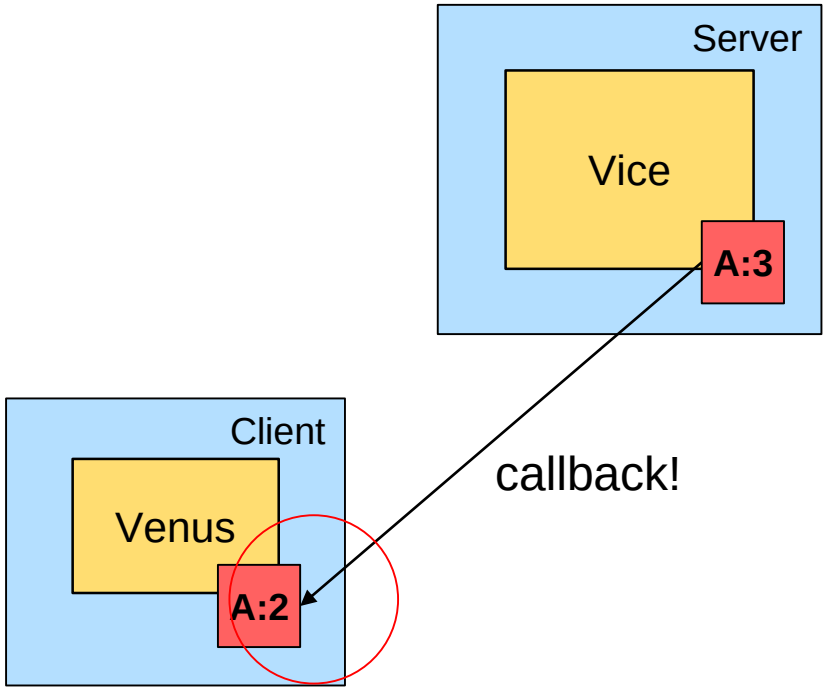
Prototype Server Usage

- CPU utilization is too high!
- Sever CPU is the performance bottleneck
  - Caused by frequent context switching
  - Traversing full pathnames
- Server loads were not evenly balanced
  - Require load balancing between servers



Changes For Performance

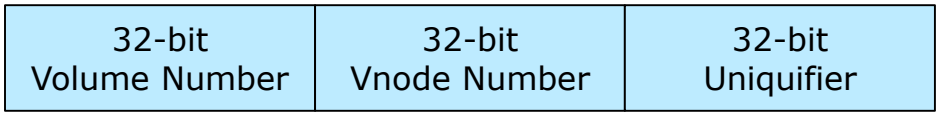
- Cache Management
  - Caches the contents of directories and symbolic links in addition to files
  - Require workstations to do path name traversals
- ***Callback***
  - *Venus* assumes that cache entries are valid unless otherwise notified
  - Server promises to notify it before allowing a modification by any other workstation



Changes For Performance (cont'd)

- Name Resolution
  - Implicit *namei* in Venus was costly
- Introduce ***fids***
  - *Volume Number*
    - Collection of files located on one server
  - *Vnode number*
    - Used as an **index** into an array containing the file storage information for the files in a sing volume
  - *Uniquifier*
    - Allows reuse of vnode numbers

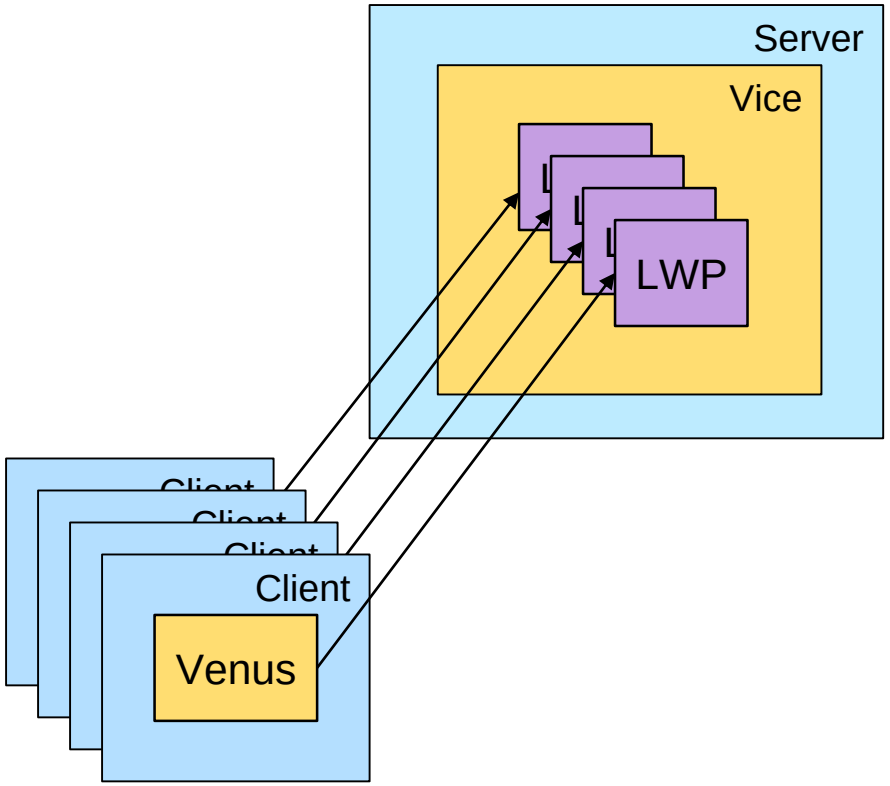
fids



No explicit location information!

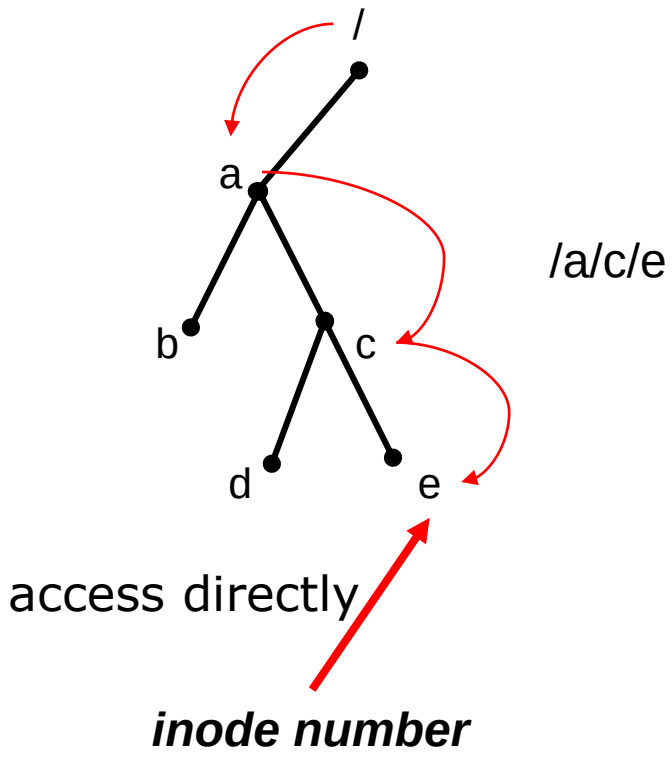
Changes For Performance (cont'd)

- Communication and Server Process Structure
  - A single process to service all clients of a server
  - Using multiple *Lightweight Processes* (*LWPs*) within one process
    - Context switching is cheap



Changes For Performance (cont'd)

- Low-Level Storage Representation
  - Access files by their inodes rather than by path names
    - Eliminated nearly all pathname lookups on workstations and servers

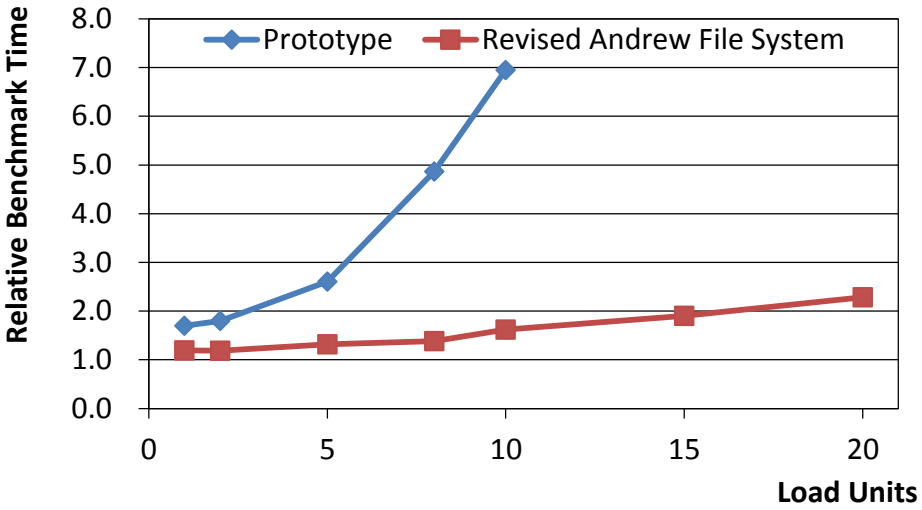


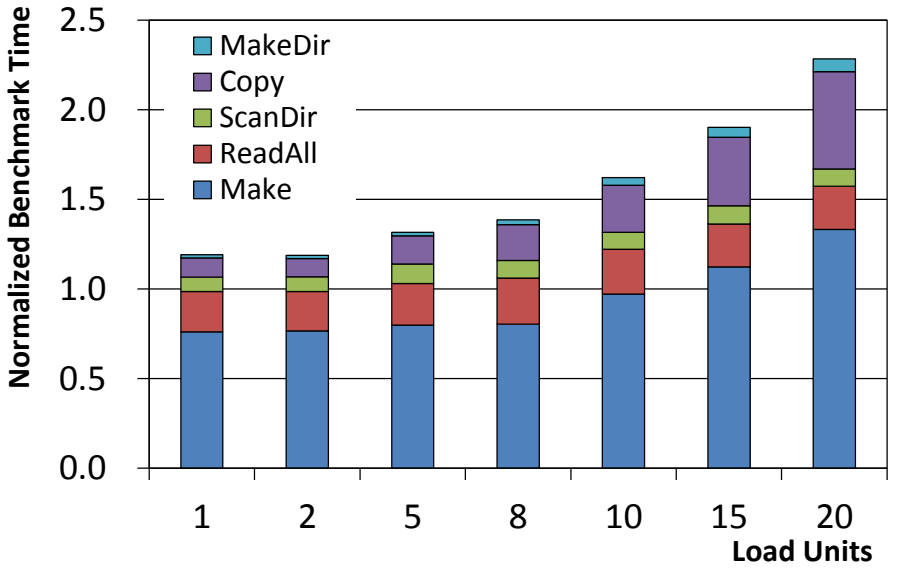
Jaegyoon Hahm

Revised Andrew File System

Revised AFS Benchmark Result

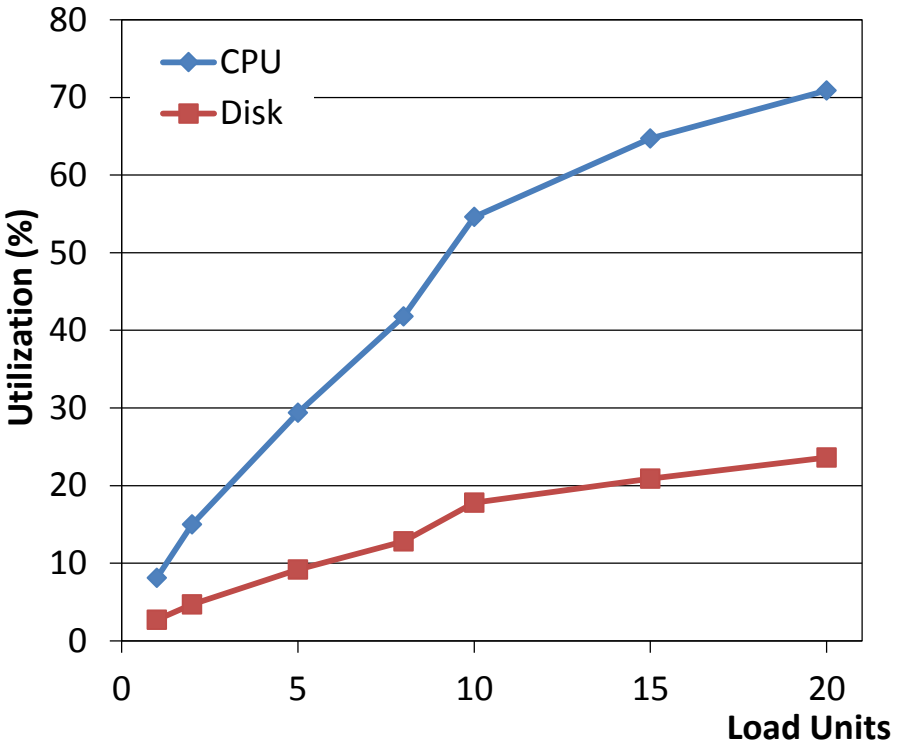
- Scalability
  - Andrew is only 19% slower than a stand-alone workstation at load 1 (prototype was 70% slower)
- Less than twice as long at a load of 15 as at a load of 1
- The design changes have improved scalability considerably!





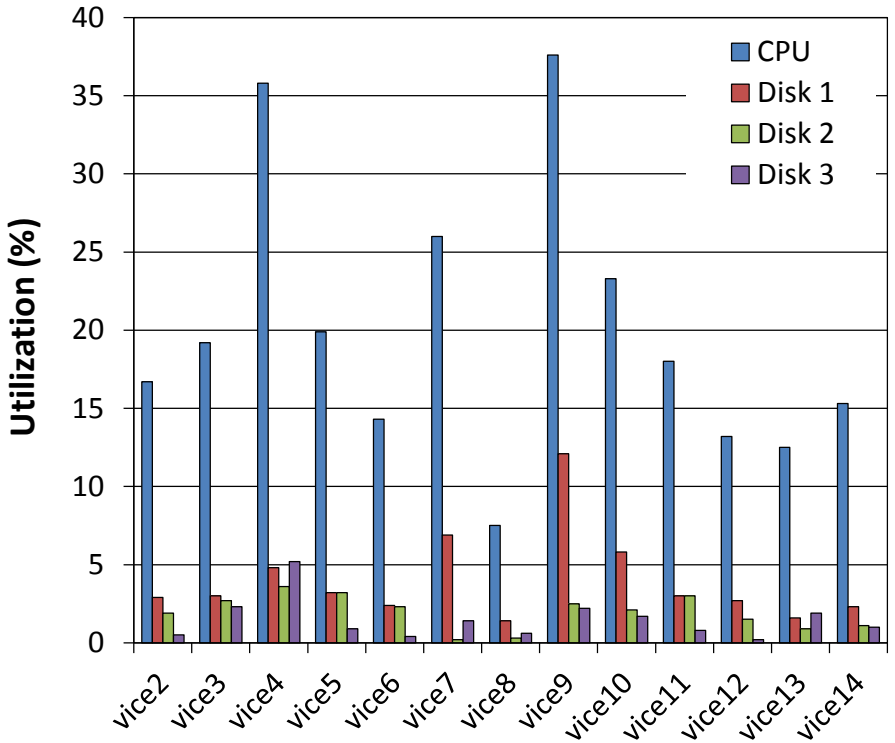
Server utilization during the benchmark

- CPU Utilization
  - 8% at load 1 to over 70% at a load of 20
  - At a load of 20 the system is still not saturated
  - But performance bound is still CPU
- Disk Utilization
  - Below 20% even at a load of 20
- Better performance requires,
  - More efficient server software
  - Faster server CPU



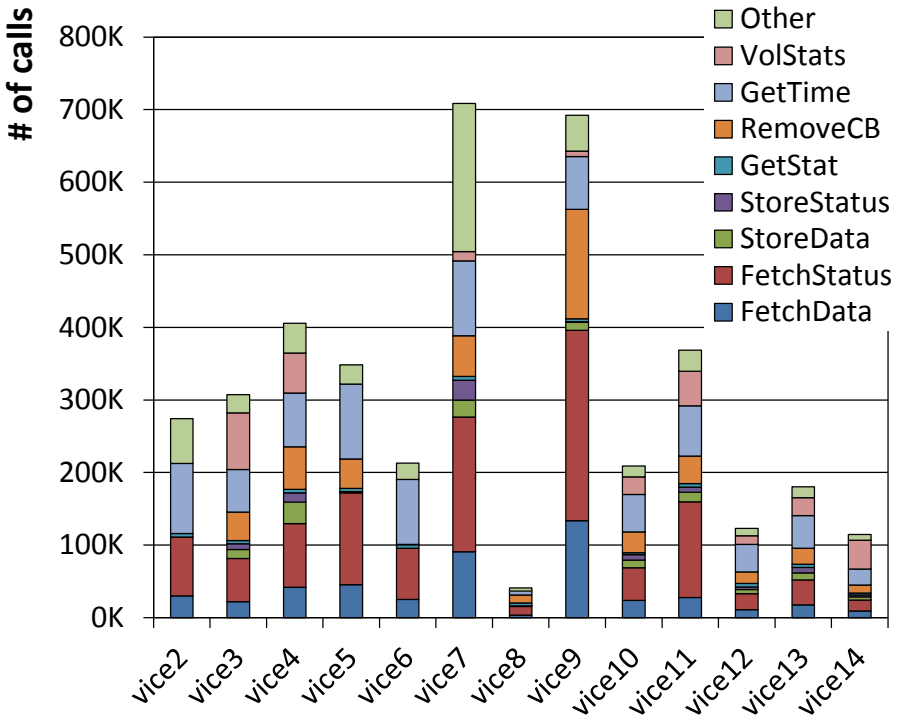
Andrew Server Usage

- Measured during 8-hour period from 9 AM to 5 PM
- Most of servers show CPU utilizations between 15% and 25%
- Vice9
  - The highest load of any server
  - Serve a bulletin board that a collection of directories that are frequently accessed and modified by many different users



Distribution of Calls to Andrew Servers

- *GetTime*
  - Most frequently called
  - Used by workstations to synchronize their clocks and as an implicit keepalive
- *FetchStatus*
  - Generated by users listing directories
- *RemoveCB*
  - Flushes a cache entry
  - Vice9: Indicates that the files it stores exhibit poor locality (mostly just one read for bulletin board)
  - Vice8: used by operation staff
  - Modification made to do RemoveCB on groups



Comparison with A Remote-Open File System

- Caching of entire file in the local disks of AFS was motivated by
  - Locality of file references by typical users makes caching attractive
  - Whole-file transfer contacts servers only on opens and closes, read-writes cause no network traffic
  - Whole-file transfer uses efficient bulk data transfer protocols.
  - The amount of data fetched after a reboot is usually small.
  - Caching of entire files simplifies cache management

Comparison with A Remote-Open File System

- Drawbacks of the entire file caching approach in AFS
  - Workstations require local disks for acceptable performance
  - Files that are larger than the local disk cache cannot be accessed at all
  - Strict emulation of 4.2BSD concurrent read and write semantics across workstations is impossible, because reads and writes are not intercepted.
- Nevertheless, our approach provides superior proformance in a large scale systems.

Comparison with A Remote-Open File System

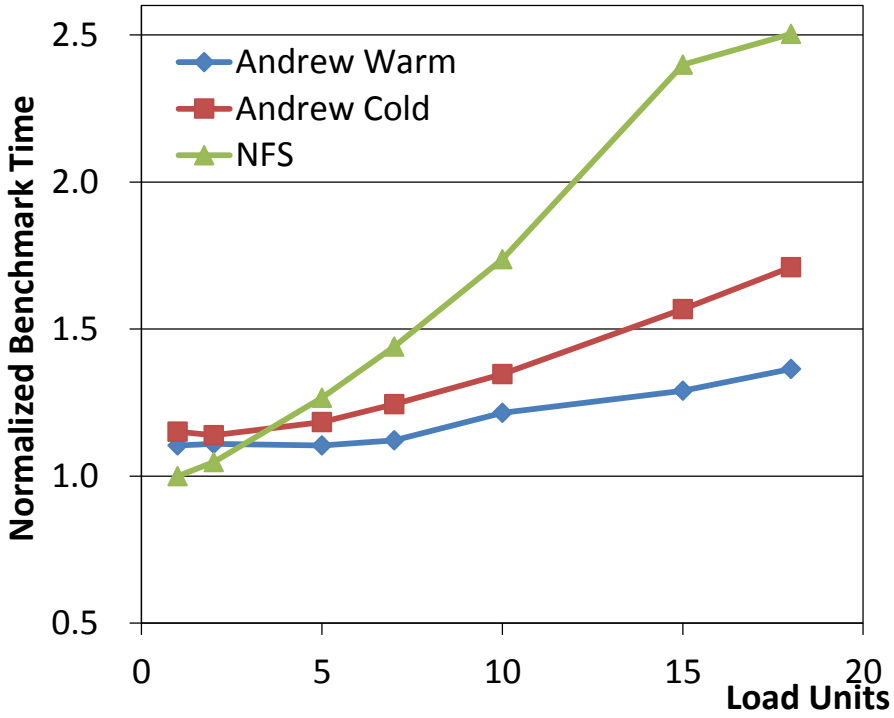
- Remote-Open File System: Sun Microsystem NFS, AT&T RFS, Locus
  - The data in a file are not fetched en masse
  - Instead, the remote site potentially participates in each individual read and write operation
  - Even though buffering and read-ahead are used to improve performance but the remote site is still conceptually involved in every I/O
- Comparison with NFS
  - Representative of Remote-Open File System
  - Mature product from a successful venter of distributed computing and de facto standard

Sun Network File System

- Servers must be identified and mounted individually: no transparent file location facility
- Both client and server components are implemented in the kernel and are more efficient than AFS
- Page caching: NFS caches inodes and individual pages of a file in memory.
  - Once a file is open, the remote site is treated like a local disk with read-ahead and write-behind of pages
- Consistency semantics of NFS
  - A new file may not be visible elsewhere for 30 seconds
  - Two processes writing to the same file could produce a different results

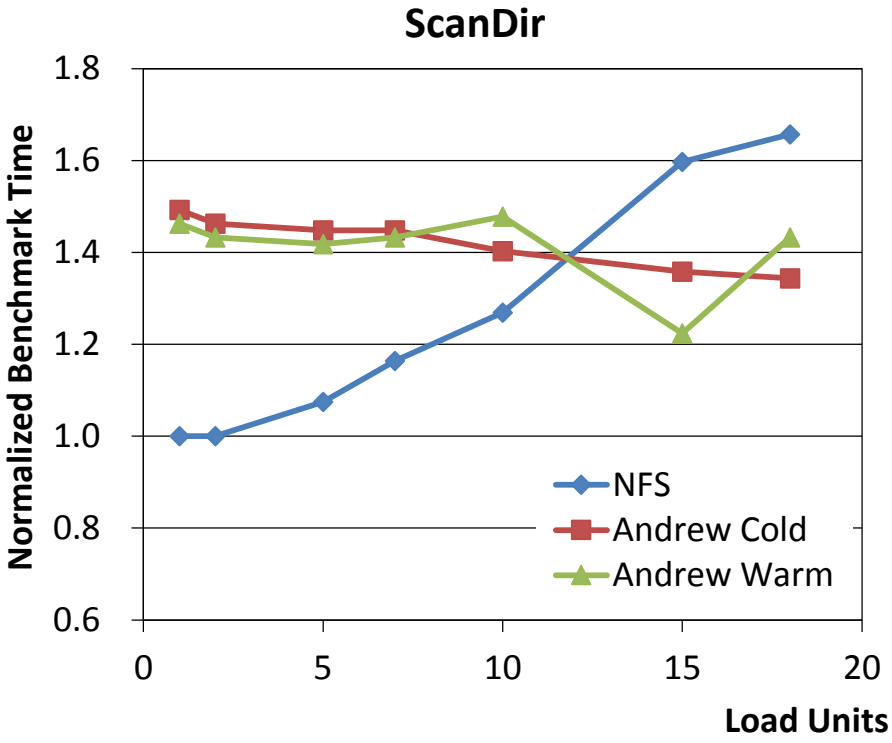
Performance

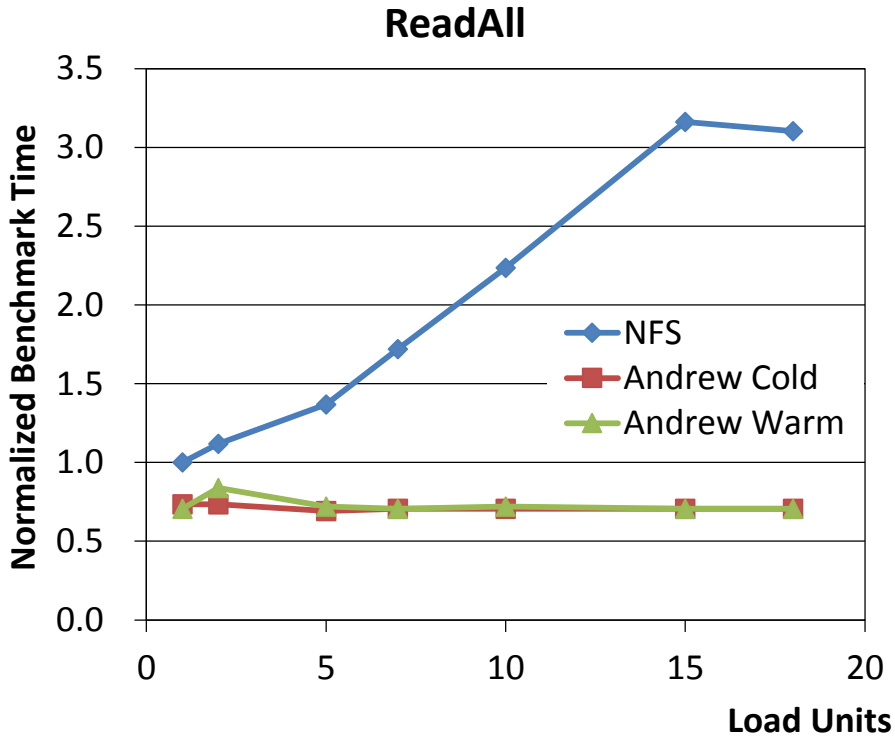
- NFS's performance degrades rapidly with increasing load
- Warm cache of AFS are better
  - Cold Cache: Workstation caches were cleared before each trial
  - Warm Cache: Caches were left unaltered



Performance

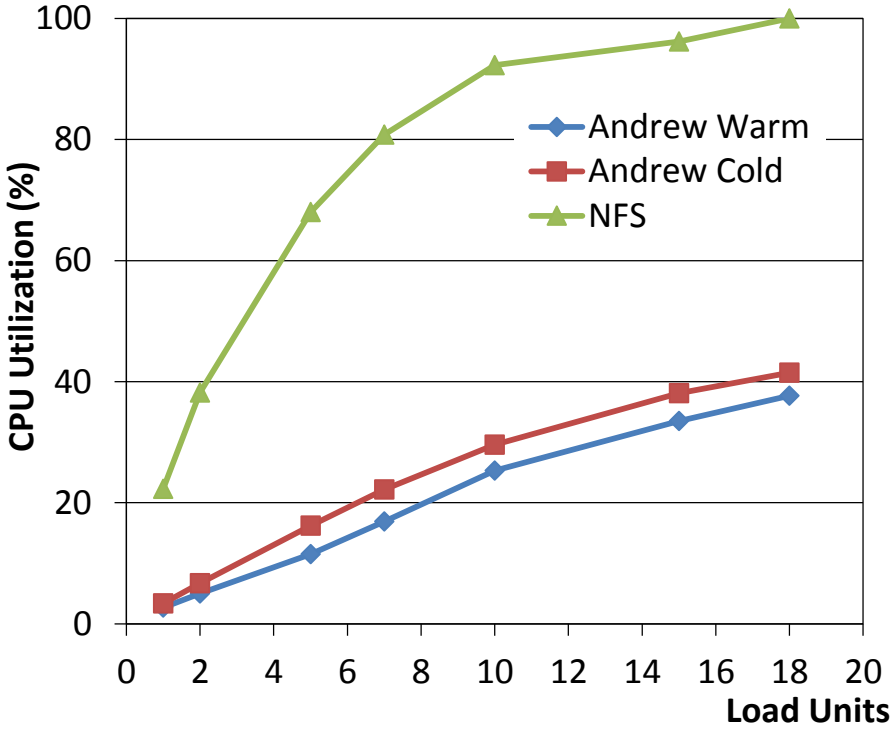
- Andrew well scales especially on *ScanDir* and *ReadAll* than NFS
- Caching and callback result this performance gain
- NFS: lack of disk cache and the need to check with the server on each file open





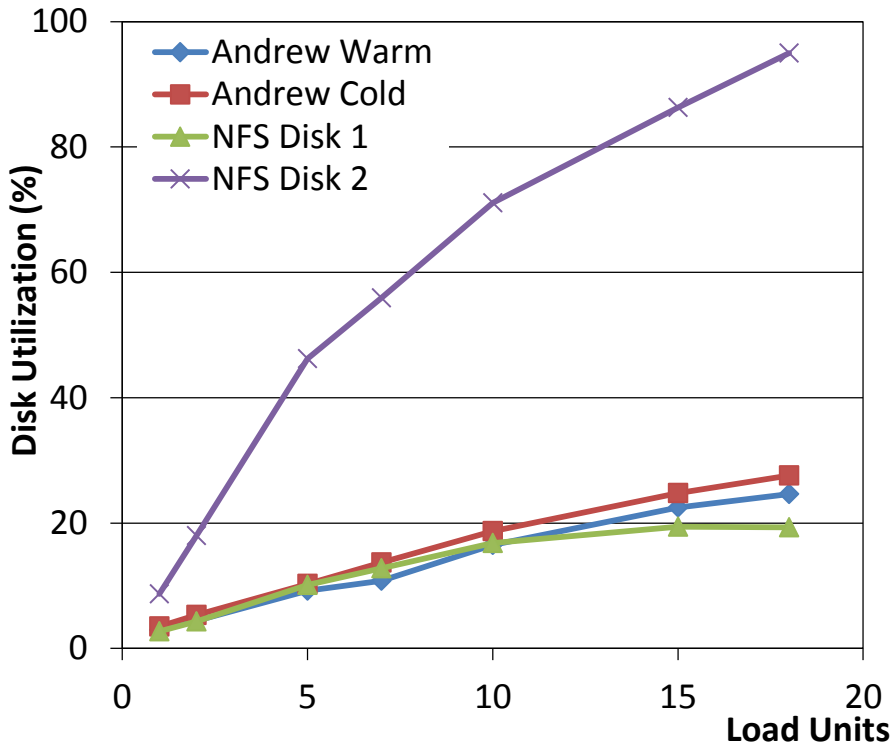
CPU Utilization

- CPU utilization of NFS is much higher than AFS
- At a load of 1, CPU utilization is about 22% in NFS but 3% in Andrew
- At a load 18, CPU utilizations saturates at 100% in NFS but for Andrew 38% in the cold cache and 42% in the warm case



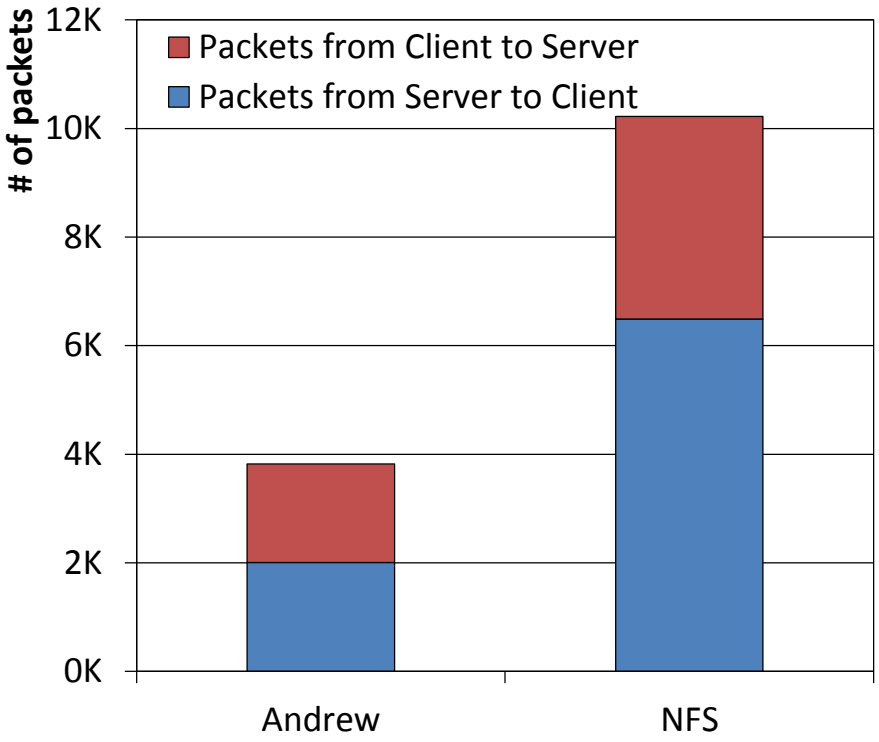
Disk Utilization

- NFS used both disks on the server, with utilizations rising from about 9% and 3% at a load of 1 to nearly 95% and 19% at a load of 18 respectively
  - Disk 1: System libs.
  - Disk 2: User data
- Andrew used only one of the server disks, with utilization rising from about 4% at a load of 1 to about 33% at a load of 18 in the cold cache case



Network Traffic

- NFS generates nearly three times as many packets as Andrew at a load of one



Changes for Operability

- Goal: to build a system that would be easy for a small operational staff to run and monitor with minimal inconvenience to users
- Problems in the prototype: inflexible mapping of Vice files to server disk storage
  - Vice was constructed out of collections of files glued together by the 4.2BSD Mount mechanism
  - Movement of files across servers was difficult (embedded file location info)
  - It was not possible to implement a quota system
  - The mechanisms of file location and file replication were cumbersome (consistency problem)
  - Standard backup utilities were not convenient for use in distributed environment
  - Backup needs the entire disk partition taken off-line

Changes for Operability

- Volumes
  - Files in Vise can be distributed over disk partitions with *Volume*
  - Collection of files forming a partial subtree of the Vice name space
  - Volumes are glued together at *Mount Points* to form the complete name space
  - A volume resides within a single disk partition on a server
  - Volumes provides a level of *Operational Transparancy*
- Volume Movement
  - Volume movement is done by creating a clone, a frozen copy-on-write snapshot of the volume
  - During movement, the volume location database is updated

Changes for Operability

- Quotas
  - Quotas are implemented on a per volume basis
- Read-Only Replication
  - Improves availability and balances load
  - No callback needed
  - The volume location database specifies the server containing the read-write copy of a volume and a list of read-only replication sites
- Backup
  - Volumes form the basis of the backup and restoration mechanism
  - Create a frozen snapshot of a read-only clone, then transfer it

Conclusions

- Design changes in the Prototype Andrew File System improved scalability considerably
  - At a load of 20, the system was still not saturated
  - A server using revised Andrew File System can serve more than 50 users
  - Changes in cache management, name resolution, server process structure, low-level storage representation
- Volumes provide a level of *Operational Transparency* that is not supported by any other file system
  - Quota
  - Read-only replication
  - Simple and efficient backup

Thank You!

Questions?