6.172

PERFORMANCE ENGINEERING OF SOFTWARE SYSTEMS

Distributed Systems

Saman Amarasinghe

Fall 2010

Final Project

Design review with your Masters

Competition on Dec 9th

Akamai Prize for the winning team

- Celebration / demonstration at Akamai HQ
- iPOD nano for each team member!

Scaling Up

Cluster Scale

Data Center Scale

Planet Scale

Cluster Scale

Running your program in Multiple Machines

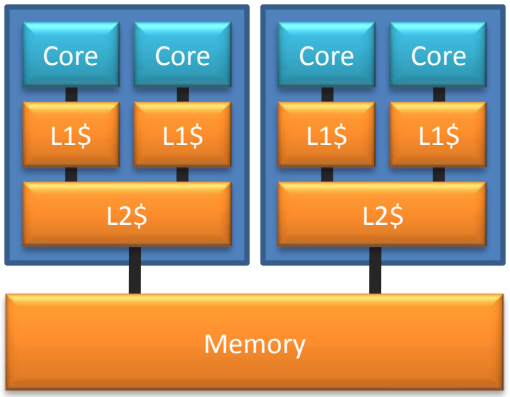
Why?

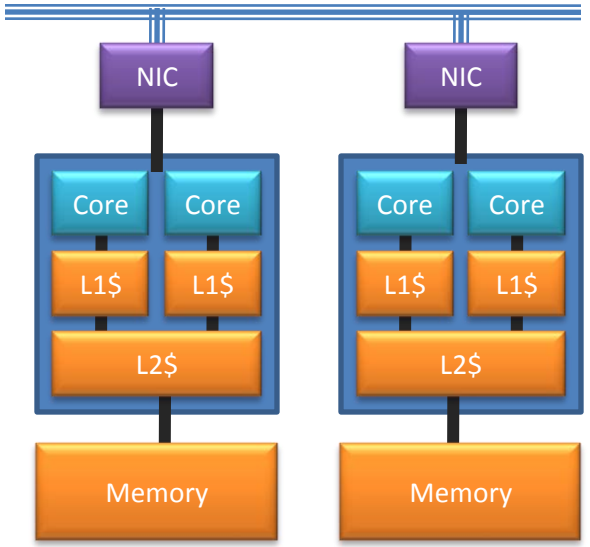
- Parallelism → Higher Throughput and Latency
- Robustness → No single point of failure
- Cost savings → Multiple PCs are lot cheaper than a mainframe

Programming Issues

- Parallel programming with message passing
- Robustness → tolerating failure

Shared vs. Distributed Memory





Memory Layer	Access Time (cycles)	Relative
Register	1	1
Cache	1–10	10
DRAM Memory	1000	100
Remote Memory (with MPI)	10000	10

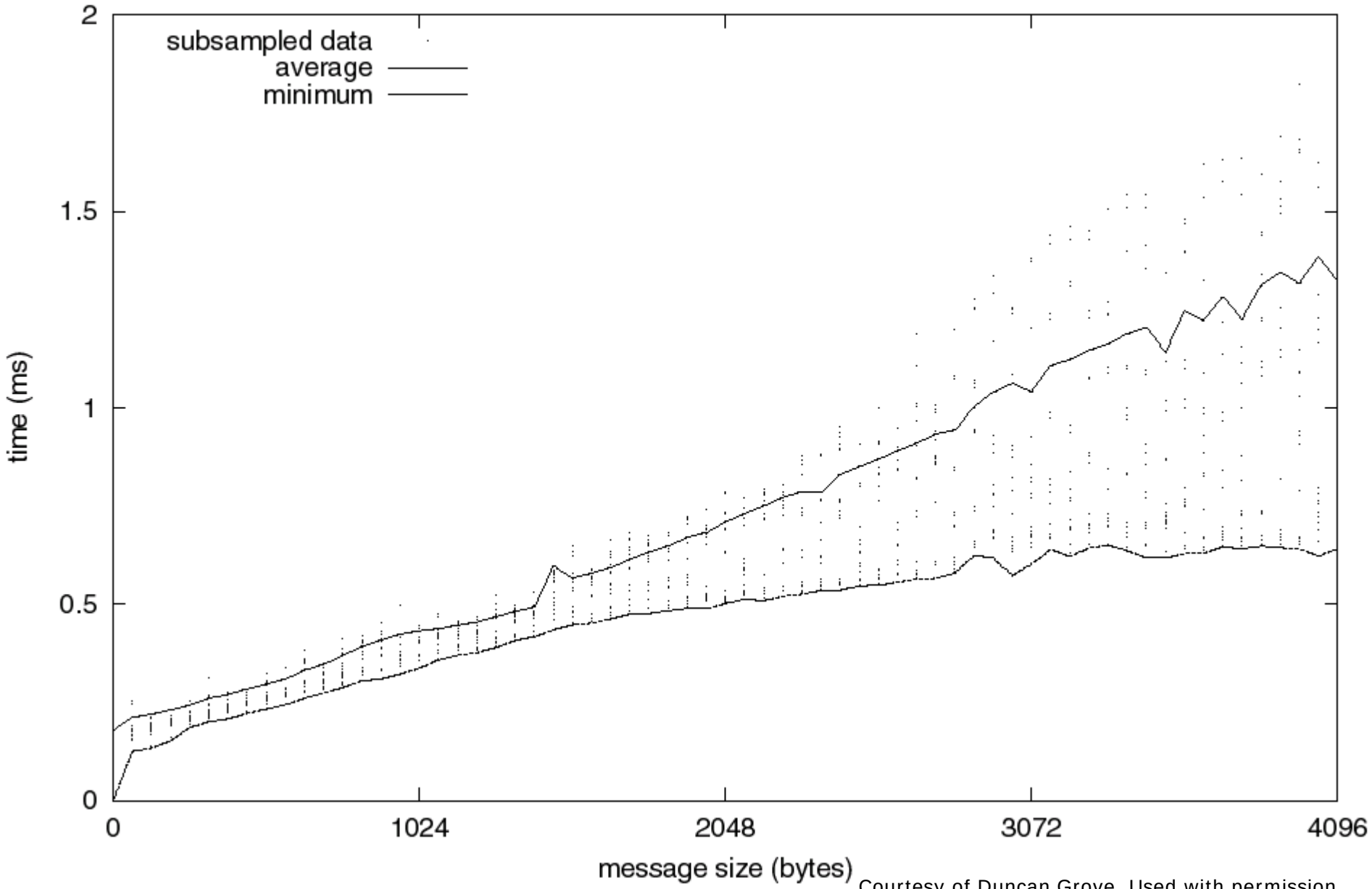
Shared Memory vs. Message Passing

Shared Memory

- All Communication via. Memory
- Synchronization via. Locks
  - Locks get translated into memory actions

Message Passing

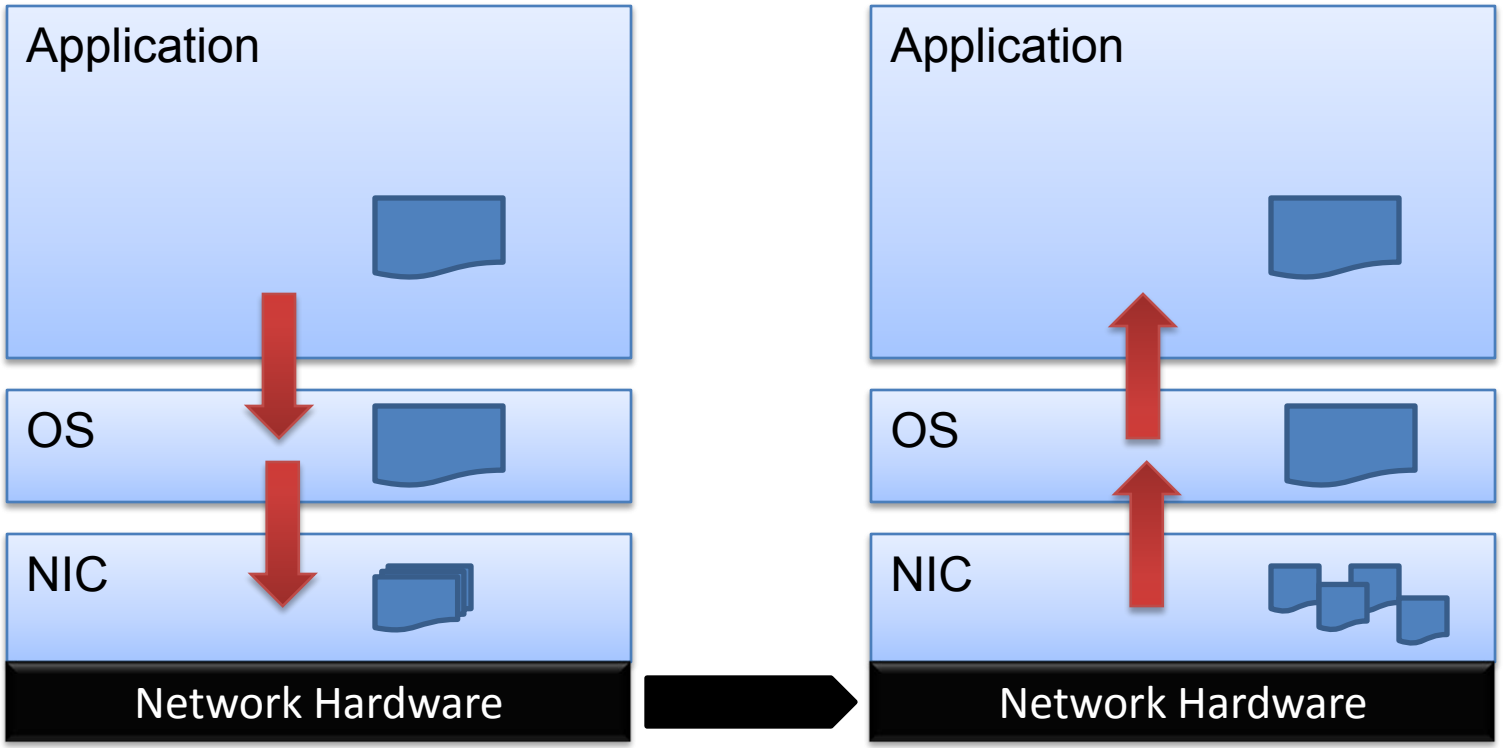
- Communication via. explicit messages
- Synchronization via. synchronous messages



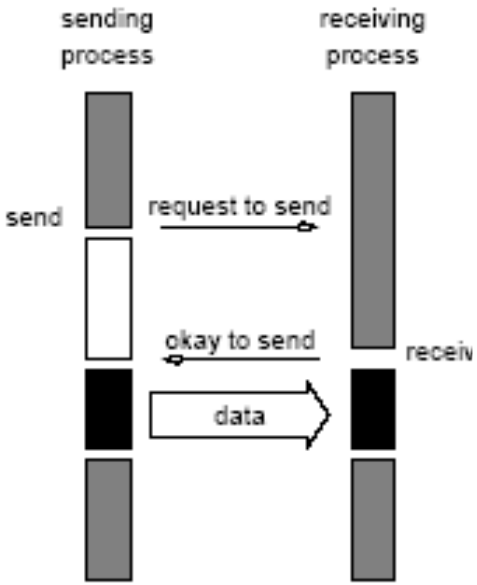
Courtesy of Duncan Grove. Used with permission.

From a slide by Duncan Grove @ Adelaide University

Anatomy of a message



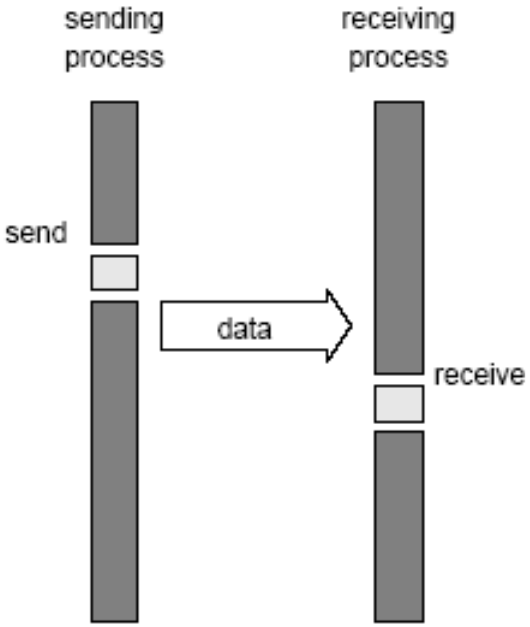
Non-Buffered Blocking Message Passing Operations



(a) Sender comes first; idling at sender

When sender and receiver do not reach communication point at similar times, there can be considerable idling overheads.

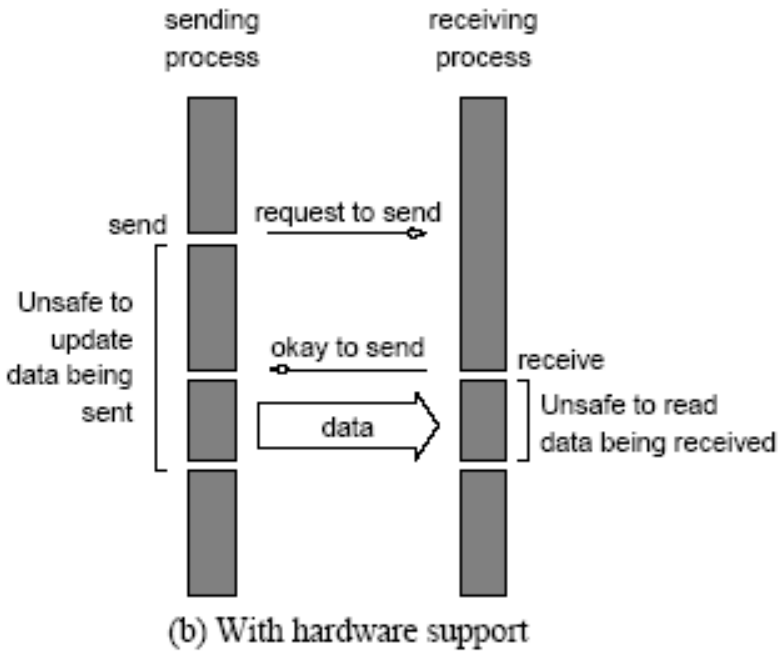
Buffered Blocking Message Passing Operations



Blocking buffered transfer protocols:

- (a) in the presence of communication hardware with buffers at send and receive ends
- (b) in the absence of communication hardware, sender interrupts receiver and deposits data in buffer at receiver end.

Non-Blocking Message Passing Operations



(b) With hardware support

Non-blocking non-buffered send and receive operations

- (a) in absence of communication hardware;
- (b) in presence of communication hardware.

MPI Language

Emerging standard language for cluster programming

- Machine independent → portable

Features

- Each machine has a process
  - Its own thread of control
  - Its own memory
- Each process communicate via messages
  - Data that need to be communicated will get packaged into a message and sent
  - Addresses in each process may be different
    - Cannot communicate pointers

```
#include "mpi.h"
#include <stdio.h>
int main(int argc, char * argv[])
{
   int numtasks, myid, dest, source, rc, count, tag=1;
   char inmsg, outmsg='x';
   MPI_Status Stat;
   MPI_Init(&argc,&argv);
   MPI_Comm_size(MPI_COMM_WORLD,  &numtasks);
   MPI_Comm_rank(MPI_COMM_WORLD,  &myid);
   if (myid== 0) {
      dest = 1;
      source = 1;
      rc = MPI_Send(&outmsg, 1, MPI_CHAR, dest, tag, MPI_COMM_WORLD);
      rc = MPI_Recv(&inmsg, 1, MPI_CHAR, source, tag, MPI_COMM_WORLD,  &Stat);
   } else if (myid== 1) {
      dest = 0;
      source = 0;
      rc = MPI_Recv(&inmsg, 1, MPI_CHAR, source, tag, MPI_COMM_WORLD,  &Stat);
      rc = MPI_Send(&outmsg, 1, MPI_CHAR, dest, tag, MPI_COMM_WORLD);
   }
   rc = MPI_Get_count(&Stat, MPI_CHAR, &count);
   MPI_Finalize();
}
```

```
#include "mpi.h"
#include <stdio.h>
int main(int argc, char * argv[])
{
   int numtasks, myid, next, prev, buf[2], tag1=1, tag2=2;
   MPI_Request recv_reqs[2], send_reqs[2];
   MPI_Status stats[4];
   MPI_Init(&argc,&argv);
   MPI_Comm_size(MPI_COMM_WORLD,  &numtasks);
   MPI_Comm_rank(MPI_COMM_WORLD,  &myid);
   prev = (myid-1)%numtasks;
   next = (myid+1)%numtasks;

   MPI_Irecv(&buf[0], 1, MPI_INT, prev, tag1, MPI_COMM_WORLD, &recv_reqs[0]);
   MPI_Irecv(&buf[1], 1, MPI_INT, next, tag2, MPI_COMM_WORLD, &recv_reqs[1]);
   MPI_Isend(&rank, 1, MPI_INT, prev, tag2, MPI_COMM_WORLD, &send_reqs[0]);
   MPI_Isend(&rank, 1, MPI_INT, next, tag1, MPI_COMM_WORLD, &send_reqs[1]);

   MPI_Waitall(2, recv_reqs, stats);
   { do some work }
   MPI_Waitall(2, send_reqs, stats);
    MPI_Finalize();
}
```

Example: PI in C - 1

```
#include "mpi.h"
#include <math.h>
int main(int argc, char *argv[])
{
 int done = 0, n, myid, numprocs, i, rc;
 double PI25DT = 3.141592653589793238462643;
 double mypi, pi, h, sum, x, a;
 MPI_Init(&argc,&argv);
 MPI_Comm_size(MPI_COMM_WORLD,&numprocs);
 MPI_Comm_rank(MPI_COMM_WORLD,&myid);
 while (!done)  {
   if (myid == 0) {
     printf("Enter the number of intervals: (0 quits) ");
     scanf("%d",&n);
   }
   MPI_Bcast(&n, 1, MPI_INT, 0, MPI_COMM_WORLD);
   if (n == 0) break;
```

Example: PI in C - 2

```
    h   = 1.0 / (double) n;
    sum = 0.0;
    for (i = myid + 1; i <= n; i += numprocs) {
      x = h * ((double)i - 0.5);
      sum += 4.0 / (1.0 + x*x);
    }
    mypi = h * sum;
    MPI_Reduce(&mypi, &pi, 1, MPI_DOUBLE, MPI_SUM, 0,
               MPI_COMM_WORLD);
    if (myid == 0)
      printf("pi is approximately %.16f, Error is %.16f\n",
             pi, fabs(pi - PI25DT));
  }
  MPI_Finalize();
    return 0;
}
```

Courtesy of William Gropp. Used with permission.

Correctness Issues

Deadlocks

- Blocking send/receives can lead to deadlocks
- Exhaustion of resources can also lead to deadlocks (next slides)

Stale data

- Need to make sure that up-to-date information is communicated

Robustness

- Single box is very reliable. And when fails it is catastrophic
- A cluster has a lot more failures
  - But you have a chance of making a program more robust

Sources of Deadlocks

Send a large message from process 0 to process 1

- If there is insufficient storage at the destination, the send must wait for the user to provide the memory space (through a receive)

What happens with

Process 0	Process 1
`Send(1)`	`Send(0)`
`Recv(1)`	`Recv(0)`

- This is called “unsafe” because it depends on the availability of system buffers

Some Solutions to the "unsafe" Problem

Order the operations more carefully:

Process 0	Process 1
`Send(1)`	`Recv(0)`
`Recv(1)`	`Send(0)`

- Use non-blocking operations:

Process 0	Process 1
`Isend(1)`	`Isend(0)`
`Irecv(1)`	`Irecv(0)`
`Waitall`	`Waitall`

Performance Issues

Occupancy Costs

Latency Tolerance

Network Bottleneck

Occupancy Cost

Each message is expensive

- Context switch, buffer copy, network protocol stack processing at the sender
- NIC to OS interrupt and buffer copy, OS to application signal and context switch and buffer copy at the receiver

Message setup overhead is high

- Send small amount of large messages

Latency Tolerance

Communication is slow

- Memory systems have 100+ to 1 latency to CPU
- Cluster interconnects have 10,000+ to 1 latency to CPU
- Grid interconnects have 10,000,000+ to 1 latency to CPU

Split operations into a separate initiation and completion step

- Programmers rarely good at writing programs with split operations

Latency Tolerance in MPI

Example: Point-to-point "Rendezvous"

- Typical 3-way:
  - Sender requests
  - Receiver acks with ok to send
  - Sender delivers data
- Alternative: "Receiver requests" 2-way
  - Receiver sends "request to receive" to designated sender
  - Sender delivers data
  - MPI_ANY_SOURCE receives interfere
- MPI RMA: sender delivers data to previously agreed location

Network Bottlenecks

Network Storms

- Bursty behavior can clog the networks
  - TCP timeouts can be very expensive
- Trying to stuff too much data can lead to big slowdowns
  - Too much data enters a overloaded switch/router/computer
  - A packet gets dropped
  - Waits for the packet until timeout
  - TCP backoff kicks in → adds a big delay

Messages are not streams

- User buffer can be sent in any order
- Allows aggressive (but good-citizen) UDP based communication
  - Aggregate acks/nacks
  - Compare to "Infinite Window" TCP (receive buffer)
- 80%+ of bandwidth achievable on long-haul system
  - Contention management can maintain "good Internet behavior"
  - Actually *reduces* network load by reducing the number of acks and retransmits; makes better use of network bandwidth (use it or lose it)

Data Center Scale

Some programs need to scale-up

- A lot of users
- A lot of data
- A lot of processing

Examples of Need to Scale

Airline Reservation System

Stock Trading System

Web Page Analysis

Scene Completion

Web Search

Example: Web Page Analysis

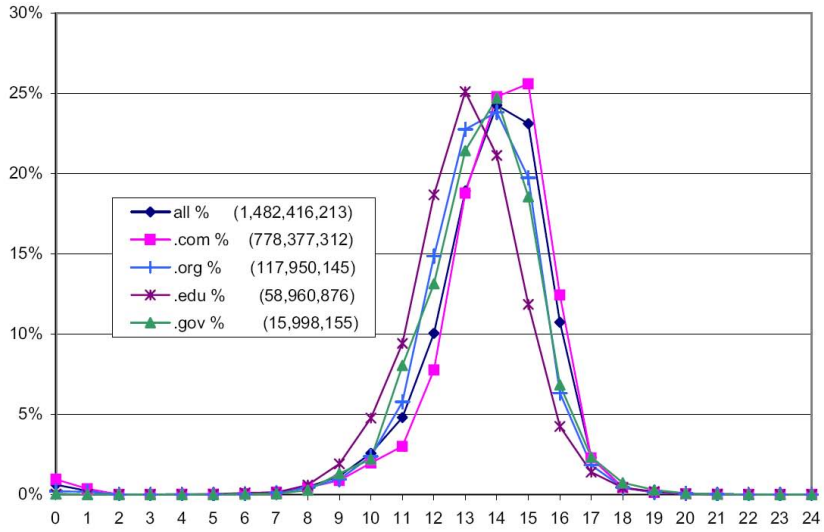


Figure 2. Distribution of document lengths overall and for selected top-level domains.

Fetterly, Manasse, Najork, Wiener (Microsoft, HP), "A Large-Scale Study of the Evolution of Web Pages," Software-Practice & Experience, 2004

Experiment

- Use web crawler to gather 151M HTML pages weekly 11 times
  - Generated 1.2 TB log information
- Analyze page statistics and change frequencies

Slide courtesy of Randal Bryant. Used with permission.

From: www.cs.cmu.edu/~bryant/presentations/DISC-FCRC07.ppt

Example: Scene Completion

Images courtesy of James Hays and Alexei Efros. Used with permission.

Hays, Efros (CMU), "Scene Completion Using Millions of Photographs" SIGGRAPH, 2007

Image Database Grouped by Semantic Content

- 30 different Flickr.com groups
- 2.3 M images total (396 GB).

Select Candidate Images Most Suitable for Filling Hole

- Classify images with gist scene detector [Torralba]
- Color similarity
- Local context matching

Computation

- Index images offline
- 50 min. scene matching, 20 min. local matching, 4 min. compositing
- Reduces to 5 minutes total by using 5 machines

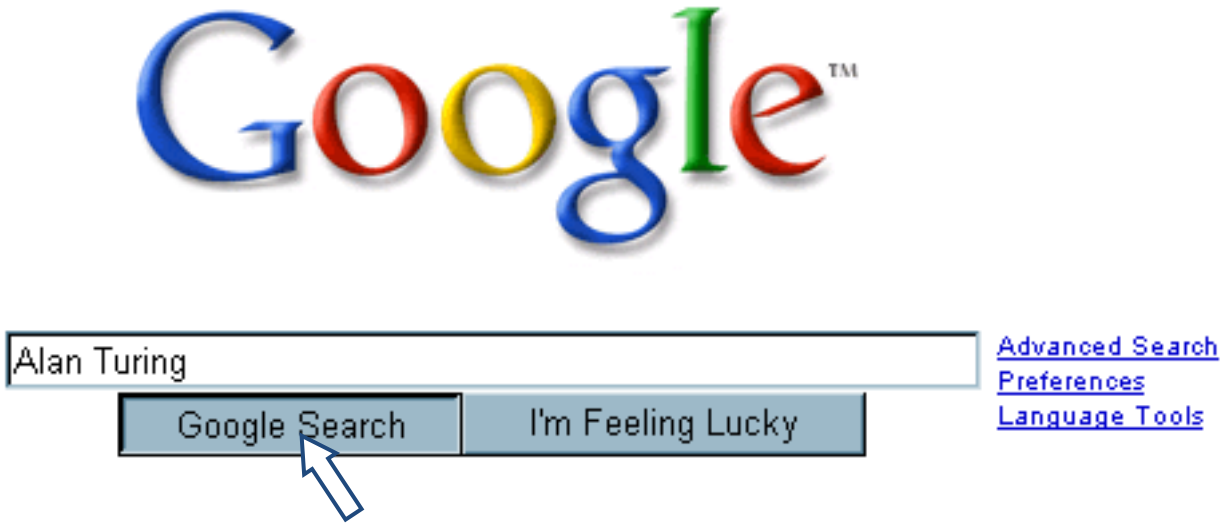
Extension

- Flickr.com has over 500 million images …

Slide courtesy of Randal Bryant. Used with permission.

From: www.cs.cmu.edu/~bryant/presentations/DISC-FCRC07.ppt

Example: Web Search



- 2000+ processors participate in a single query
- 200+ terabyte database
- 10^{10} total clock cycles
- 0.1 second response time
- 5¢ average advertising revenue

Slide courtesy of Randal Bryant. Used with permission.

From: www.cs.cmu.edu/~bryant/presentations/DISC-FCRC07.ppt

Google's Computing Infrastructure

System

- ~ 3 million processors in clusters of ~2000 processors each
- Commodity parts
  - x86 processors, IDE disks, Ethernet communications
  - Gain reliability through redundancy & software management
- Partitioned workload
  - Data: Web pages, indices distributed across processors
  - Function: crawling, index generation, index search, document retrieval, Ad placement

Barroso, Dean, Hölzle, "Web Search for a Planet: The Google Cluster Architecture" IEEE Micro 2003

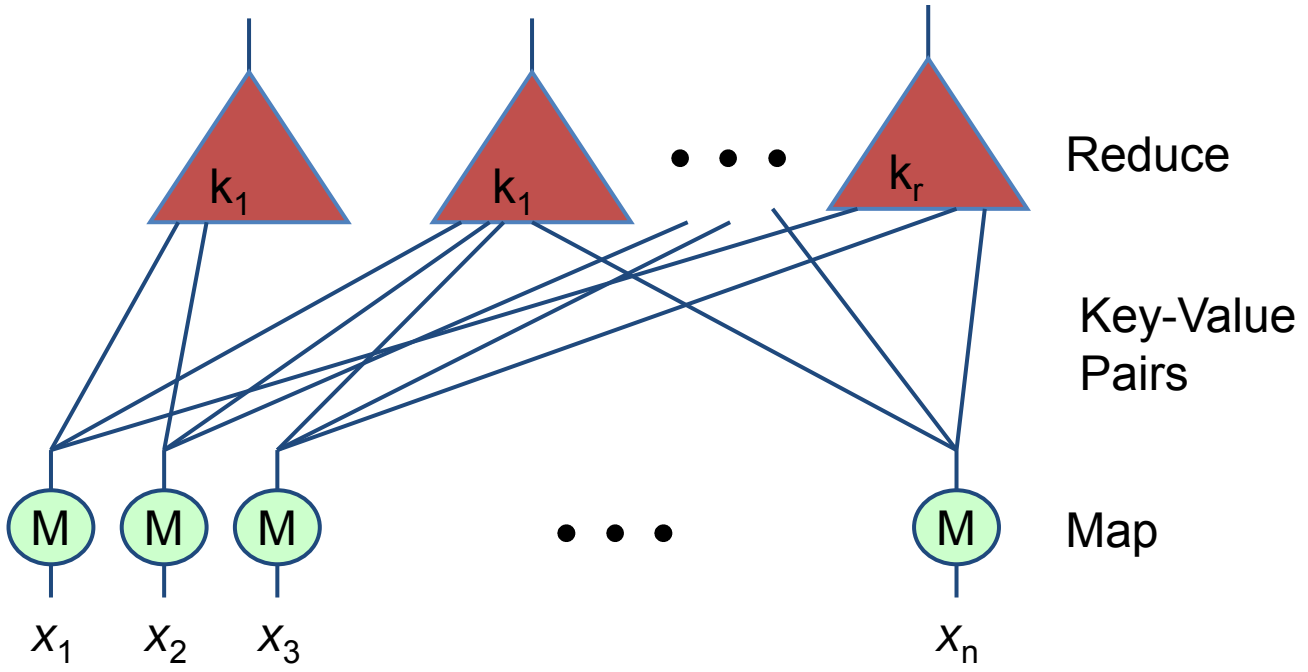
- Similar systems at Microsoft & Yahoo

Slide courtesy of Randal Bryant. Used with permission.

From: www.cs.cmu.edu/~bryant/presentations/DISC-FCRC07.ppt

Google's Programming Model

MapReduce



- Map computation across many objects
  - E.g., 10^{10} Internet web pages
- Aggregate results in many different ways
- System deals with issues of resource allocation & reliability

Dean & Ghemawat: "MapReduce: Simplified Data Processing on Large Clusters", OSDI 2004

Programming Model

Borrows from functional programming

Users implement interface of two functions:

- `map (in_key, in_value) ->`
  `(out_key, intermediate_value) list`

- `reduce (out_key, intermediate_value list) ->`
  `out_value list`

From: Mass Data Processing Technology on Large Scale Clusters Summer, 2007, Tsinghua University

map

Records from the data source

- (lines out of files, rows of a database, etc) are fed into the map function as key-value pairs: e.g., <filename, line>.

map() produces

- one or more *intermediate* values
- along with an output key from the input.

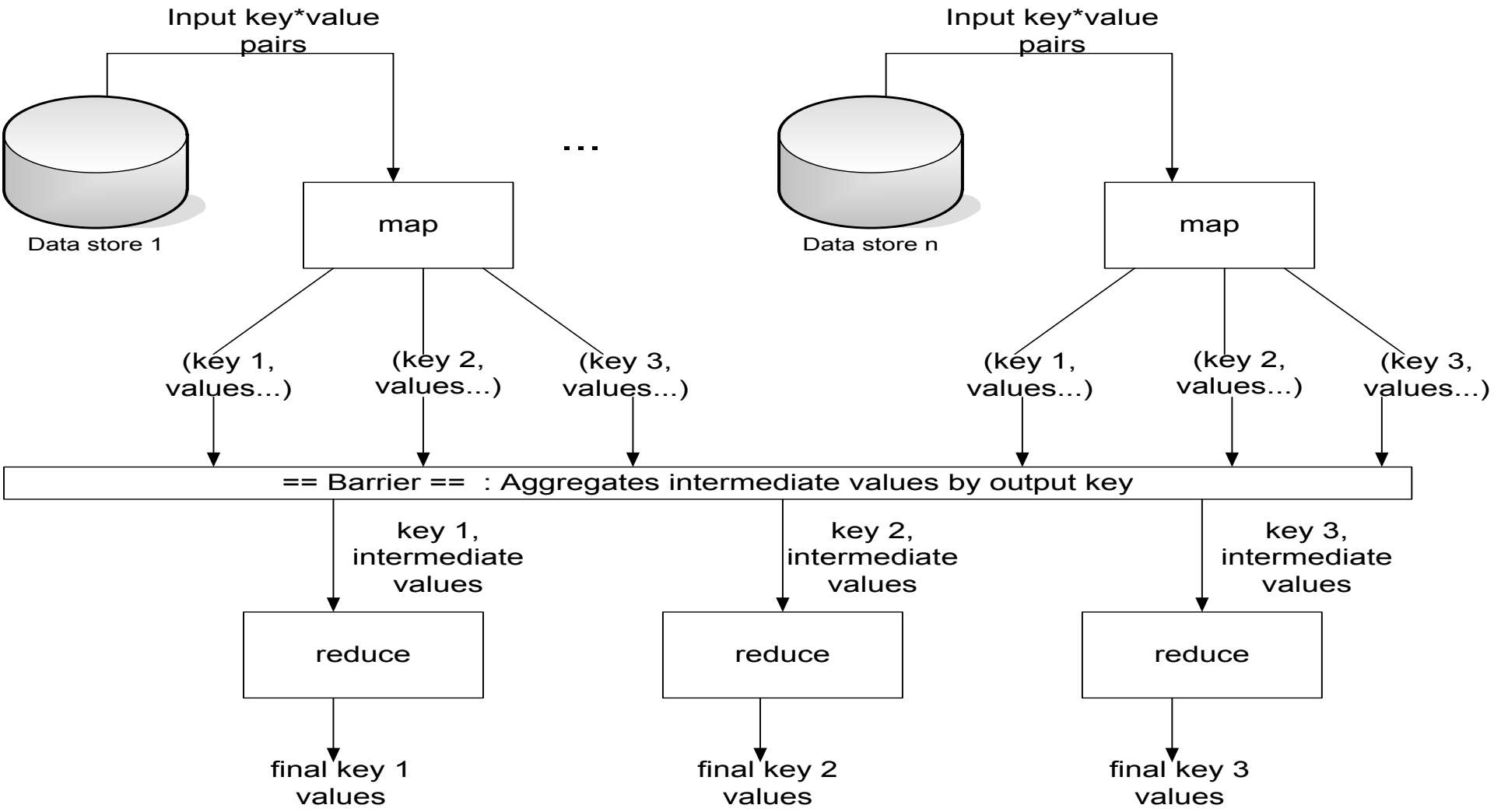
reduce

Combine data

- After the map phase is over,
- all the intermediate values for a given output key are combined together into a list
- reduce() combines those intermediate values into one or more *final values* for that same output key
- (in practice, usually only one final value per key)

From: Mass Data Processing Technology on Large Scale Clusters Summer, 2007, Tsinghua University

Architecture



From: Mass Data Processing Technology on Large Scale Clusters Summer, 2007, Tsinghua University

Parallelism

map() functions

- run in parallel, creating different intermediate values from different input data sets

reduce() functions

- also run in parallel, each working on a different output key

All values are processed *independently*

Bottleneck:

- reduce phase can't start until map phase is completely finished.

From: Mass Data Processing Technology on Large Scale Clusters Summer, 2007, Tsinghua University

Example: Count word occurrences

```
map(String input_key, String input_value):
  // input_key: document name
  // input_value: document contents
  for each word w in input_value:
    EmitIntermediate(w, "1");


reduce(String output_key, Iterator intermediate_values):
  // output_key: a word
  // output_values: a list of counts
  int result = 0;
  for each v in intermediate_values:
    result += ParseInt(v);
 Emit(AsString(result));
```


From: Mass Data Processing Technology on Large Scale Clusters Summer, 2007, Tsinghua University

How to Scale?

Distribute

- Parallelize
- Distribute data

Approximate

- Get to a sufficiently close answer, not the exact
- A little stale data might be sufficient

Transact

- If exactness is required, use transactions

Planet Scale

Some programs need to scale-up

- A lot of users
- A lot of data
- A lot of processing

Examples:

- Seti@Home
- Napster
- BitTorrent

Scaling Planet Wide

Truly Distributed

- No global operations
- No single bottleneck
- Distributed view → stale data
- Adaptive load distribution is a must

Case Study – The Bonsai System

Case study from VMware Inc.

A Prototype for "Deduplication" at Global Scale

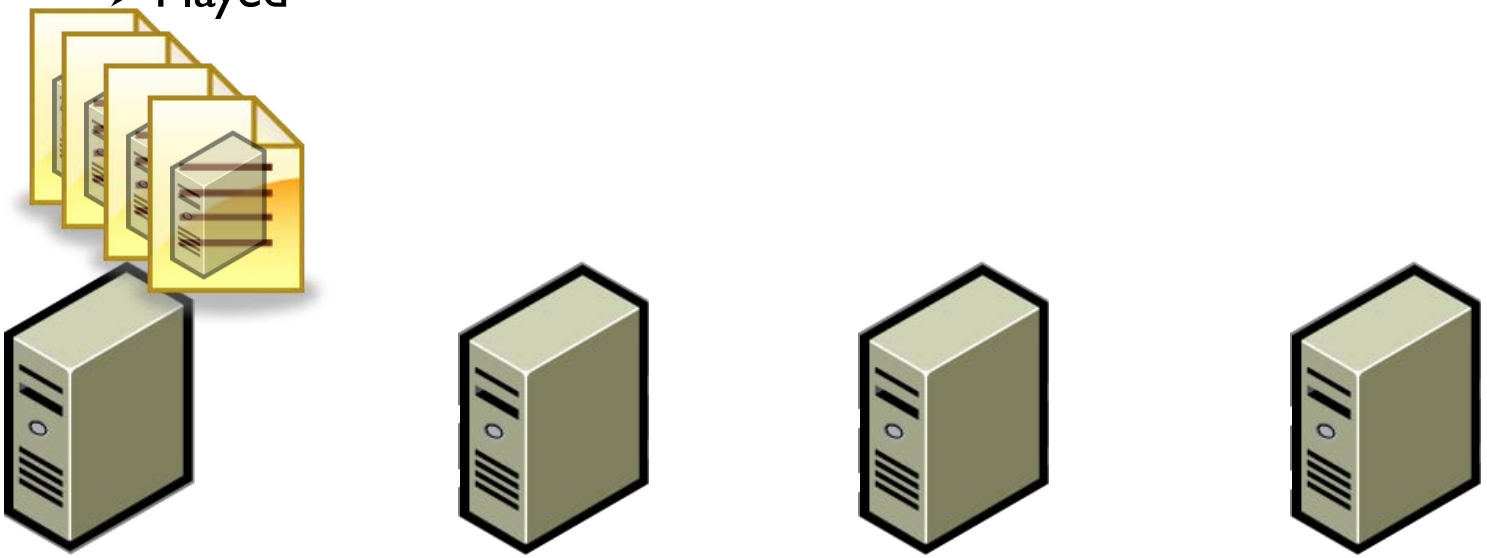
Why? For Moving Virtual Machines Across the World

What is the Virtualization Revolution

Decouple the "machine" from the physical machi
file

Virtual Machines can be..

- Replicated
- Moved
- Played

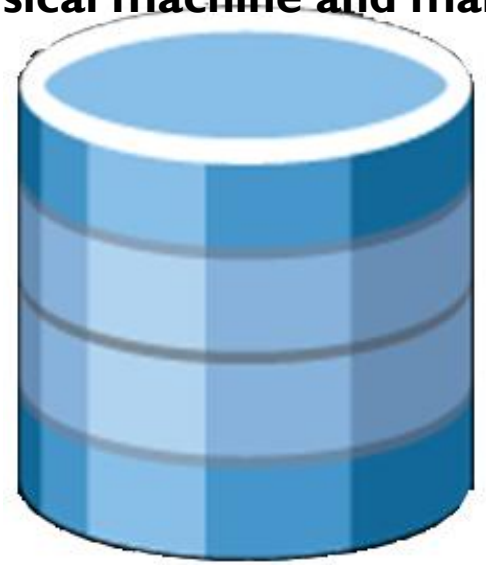


What is the Virtualization Revolution

Decouple the "machine" from the physical machine and make it a file

Virtual Machines can be..

- Replicated
- Moved
- Played
- Stored



Cloud Computing

Vision: Global marketplace of computing power

Work migrates as needed

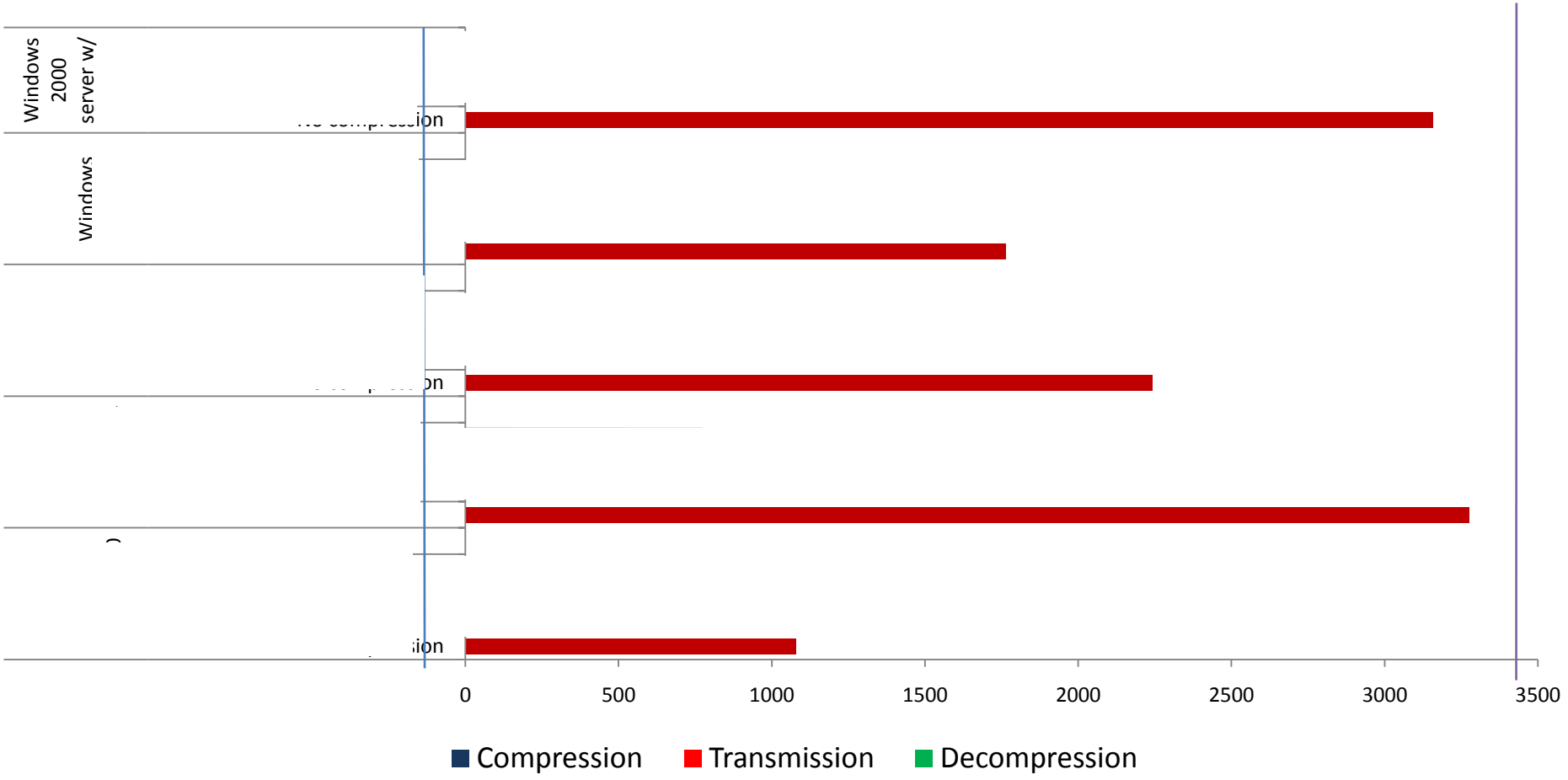
- To find more computing resources
- To be near data and/or users
- To find a cheaper provider of resources
- To amortize the risk of catastrophic failure

Issues

- Mostly applications are encapsulated as virtual machines
- They are hefty to move

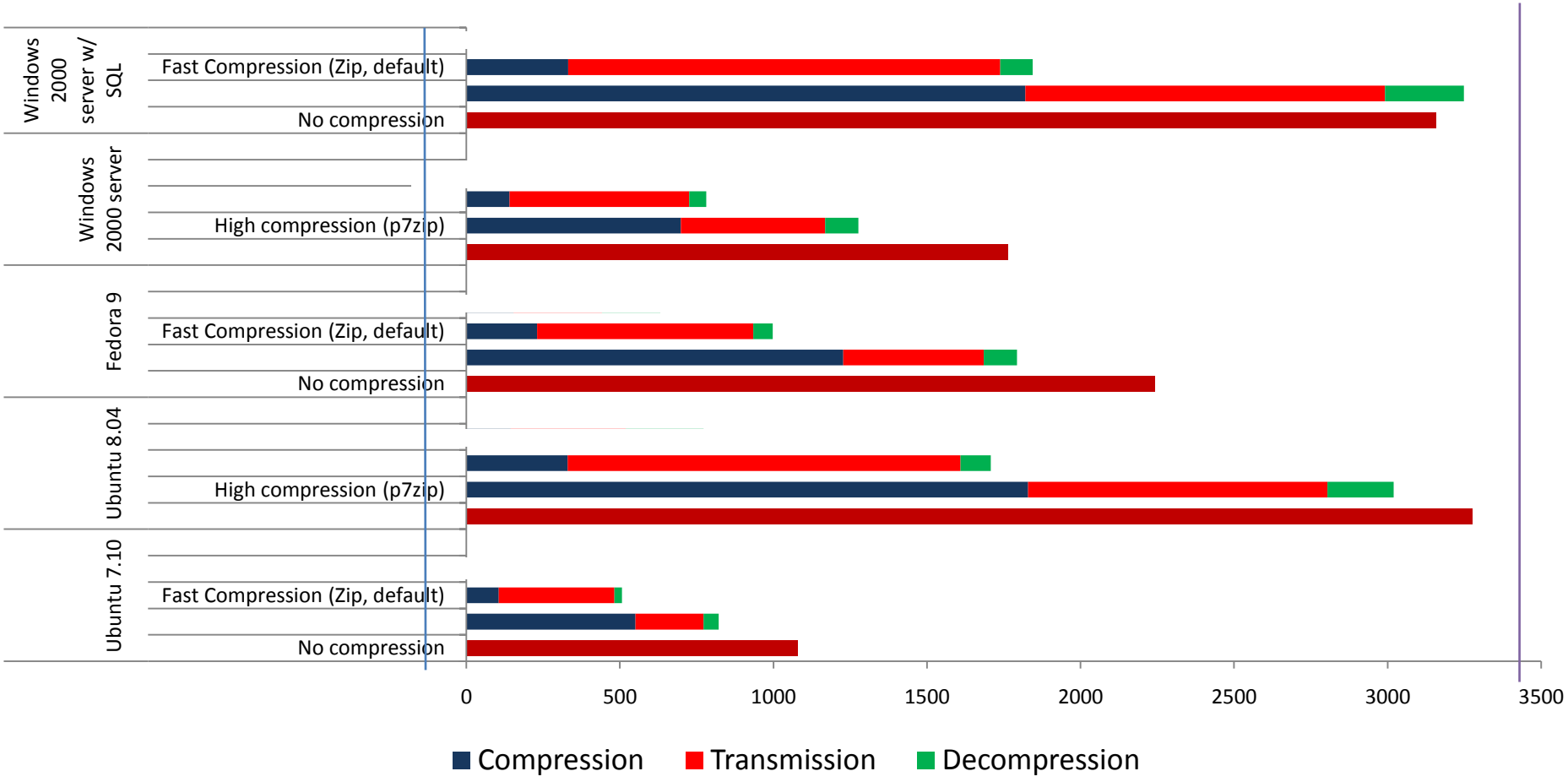
Time to Move a VM Disk file

A typical Boston desktop to Palo Alto desktop (2mbps network bandwidth) copying of a VM file



Time to Move a VM Disk file

A typical Boston desktop to Palo Alto desktop (2mbps network bandwidth) copying of a VM file



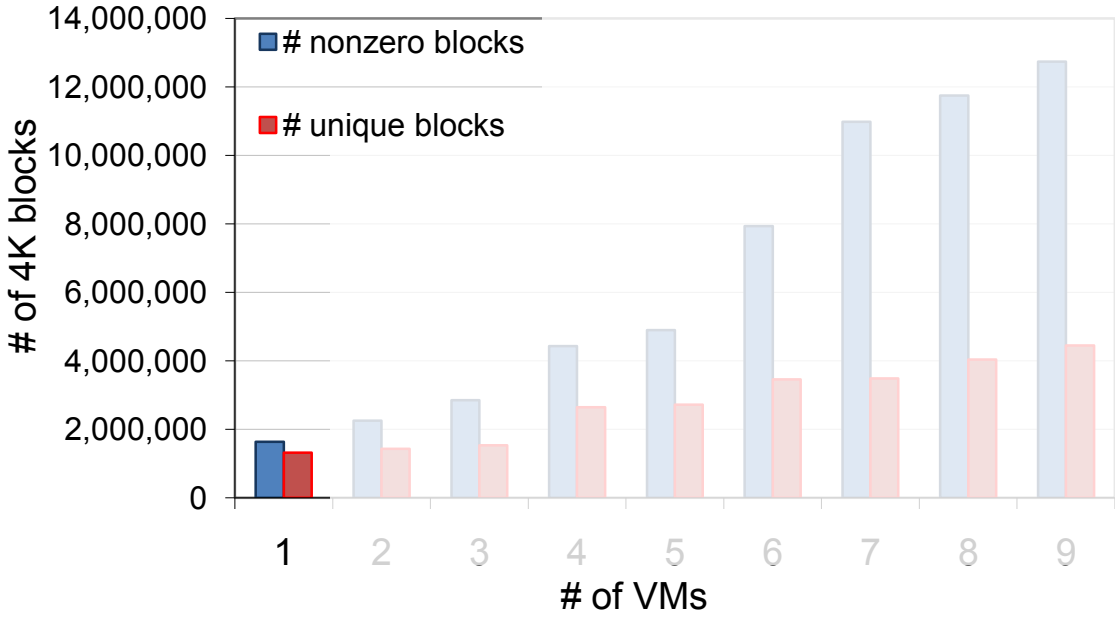
Data Redundancy – A Key Observation

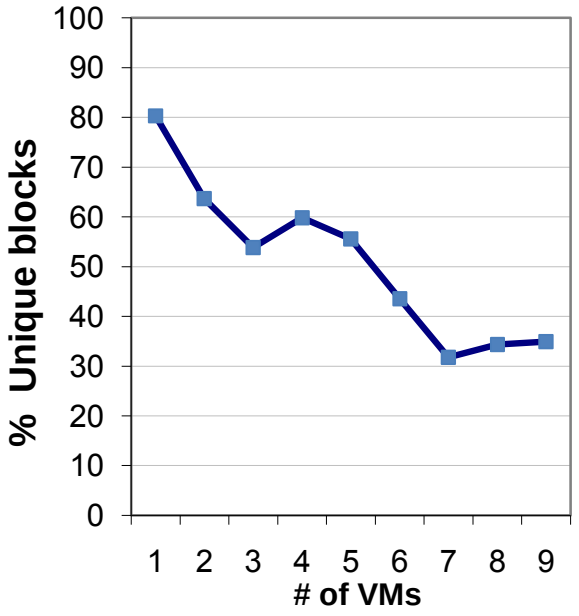
Observation 1: Large part of each VMDK is executables

Observation 2: A few applications dominate the world and are in every machine (eg: XP and Office on desktops)

Observation 3: Substantial redundancy even within a single disk (eg: DLL cache, install and repair info)

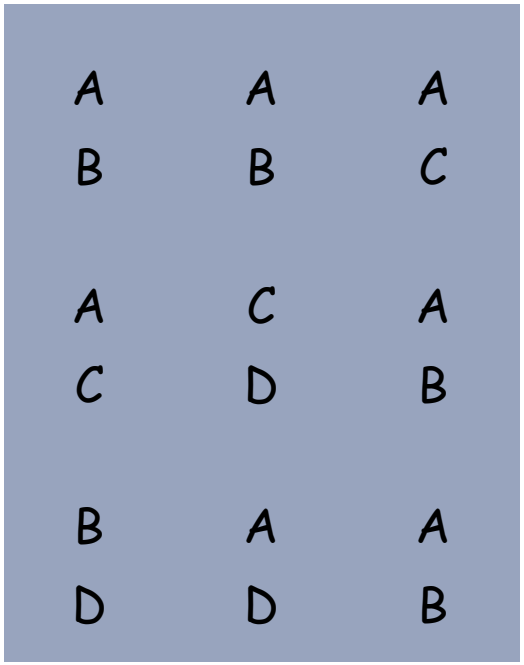
Observation 4: Many Disks have a lot of zero blocks!





Basic De-Duplication

A lot of data redundancy

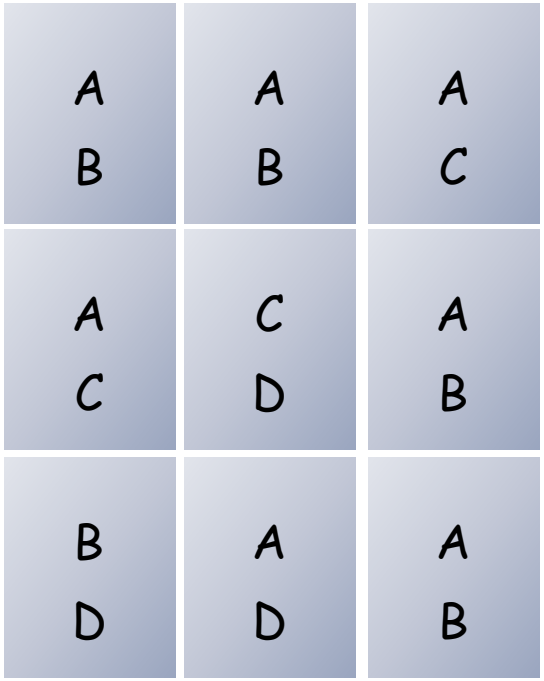


Basic De-Duplication

A lot of data redundancy

Break them into blocks

- Eg: 4K byte disk blocks



Basic De-Duplication

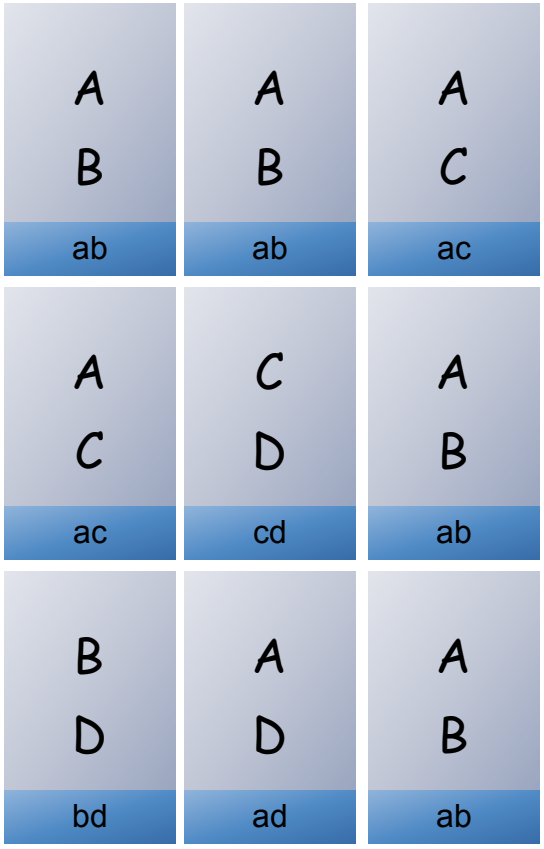
A lot of data redundancy

Break them into blocks

- Eg: 4K byte disk blocks

Calculate a hash value per block

- Eg: SHA-256 hash (32 bytes)



Basic De-Duplication

A lot of data redundancy

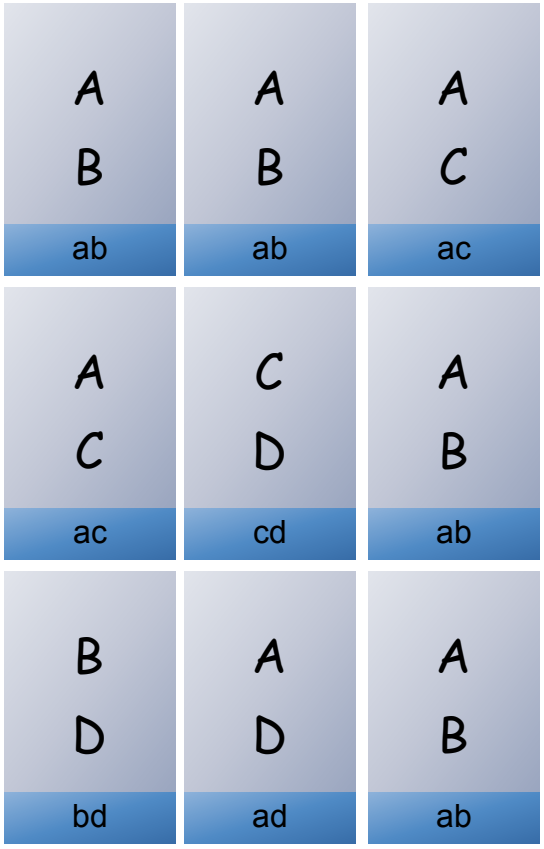
Break them into blocks

- Eg: 4K byte disk blocks

Calculate a hash value per block

- Eg: SHA-256 hash (32 bytes)

Identify similar blocks by comparing the hash values



Basic De-Duplication

A lot of data redundancy

Break them into blocks

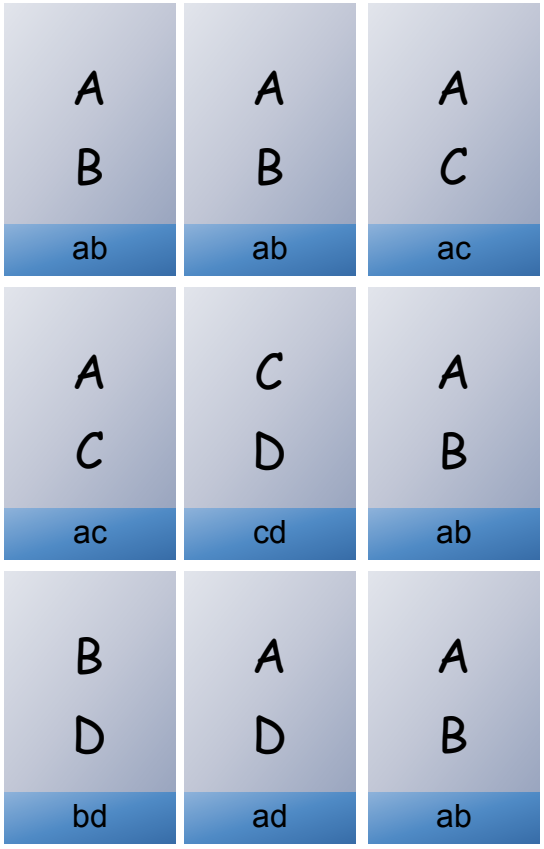
- Eg: 4K byte disk blocks

Calculate a hash value per block

- Eg: SHA-256 hash (32 bytes)

Identify similar blocks by comparing the hash values

Eliminate copies and keep only the hash as an index



Basic De-Duplication

A lot of data redundancy

Break them into blocks

- Eg: 4K byte disk blocks

Calculate a hash value per block

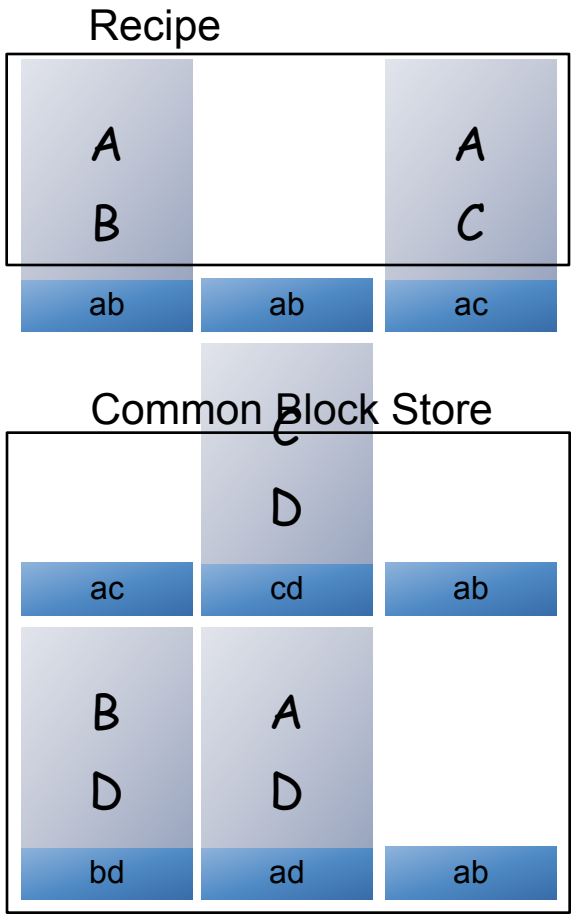
- Eg: SHA-256 hash (32 bytes)

Identify similar blocks by comparing the hash values

Eliminate copies and keep only the hash as an index

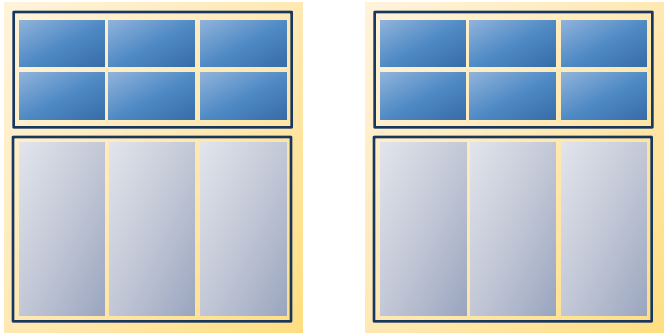
Much more compact storage

- Recipe table and common block store can be separated



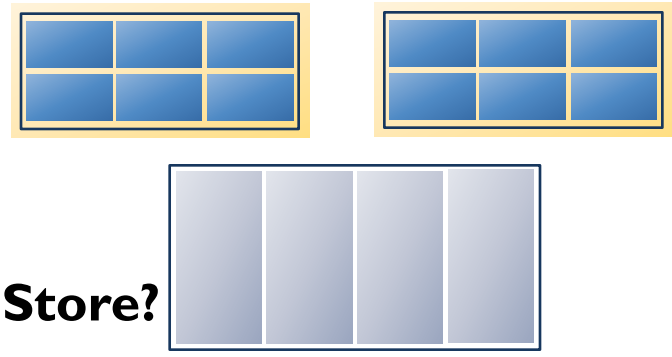
Inter. vs. Intra. Deduplication

Recipe and Common Block Store in same "system" → Traditional deduplication



Multiple Recipes for One Common Block Store

- Pro: Single copy of common data blocks across systems → Higher compression
- Cons: Lack of universal mobility
- Cons: Inability to guarantee data availability
- Cons: Inability to guarantee data integrity



Who owns and manages the Common Block Store?

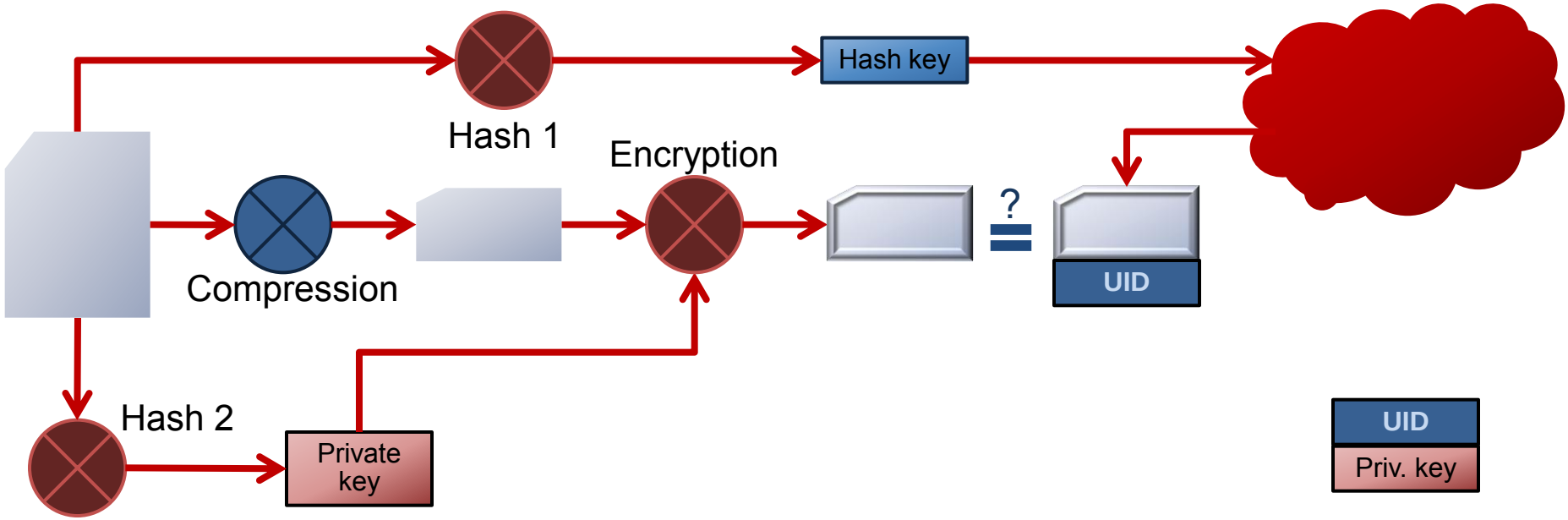
Bonsai: A Global Store for Common Disk Blocks

Take Advantage of the Monoculture

Store the common blocks in a global store

“Google” or “Akamai” or “VeriSign” for disk blocks

Bonsai Flow



Same original block from all the systems will have the identical encrypted block

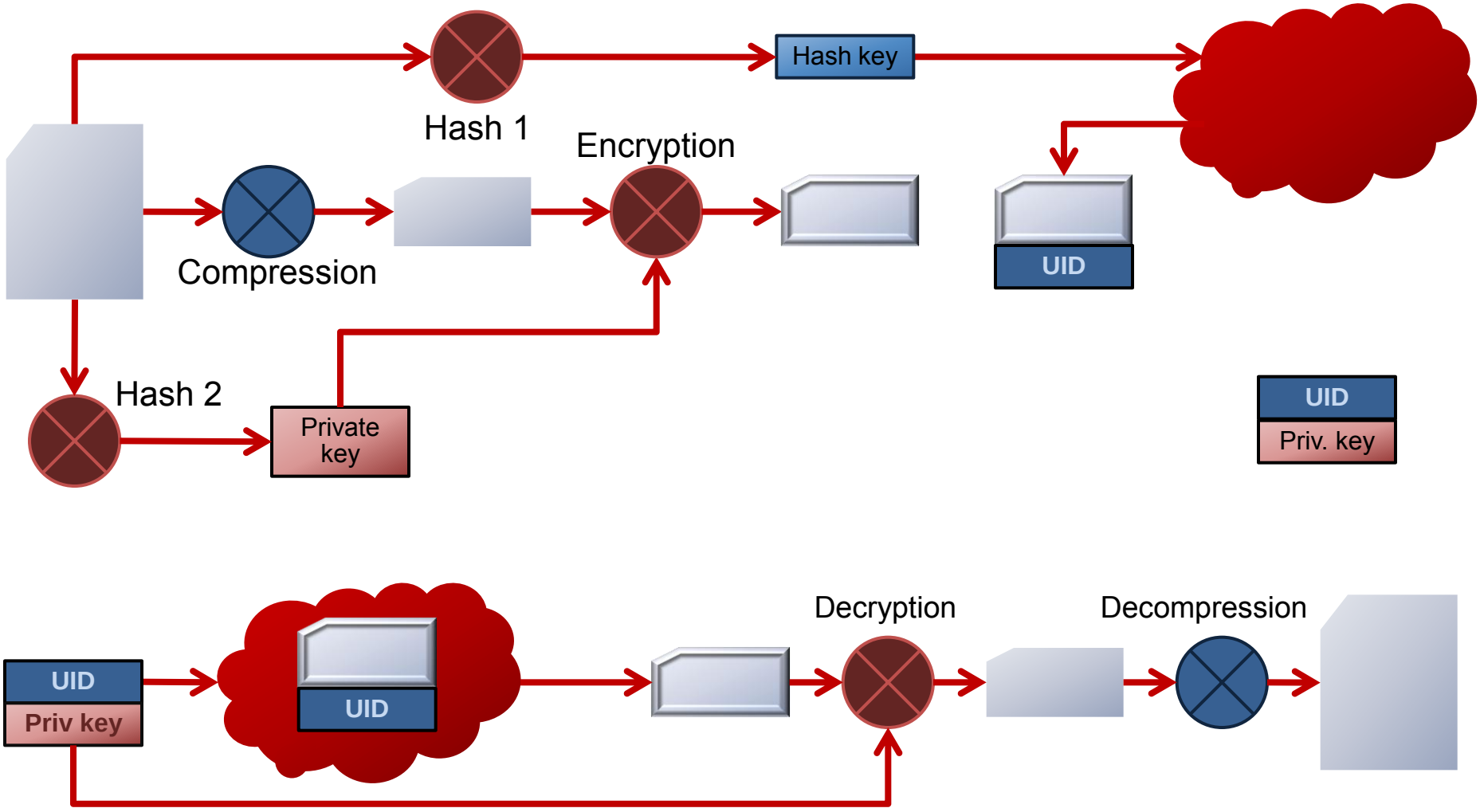
- Gets deduplicated

No one can read the content of the block unless the original block was seen at one time

- Requires the hash key to read the text
- Requires the original block to calculate the hash key

Search by UID → No possibility of getting the wrong block due to a hash collision

Bonsai Flow


Hash 1
Hash key
Encryption
Compression
UID
Hash 2
Private key
UID
Priv. key
UID
Priv key
UID
Decryption
Decompression

Hash Key vs UID

Hash Key

Reliability

- Hash check is inexpensive
- 1 in 18,446,744,073,709,600,000 (2^{64}) chances that a different block will match the hash key

Efficiency

- Lookup is random → costly

Integrity

- Can be a P2P system

Unique ID

Reliability

- Optional hash check + full page check
  - Full page check can be done later
  - No errors possible in a match

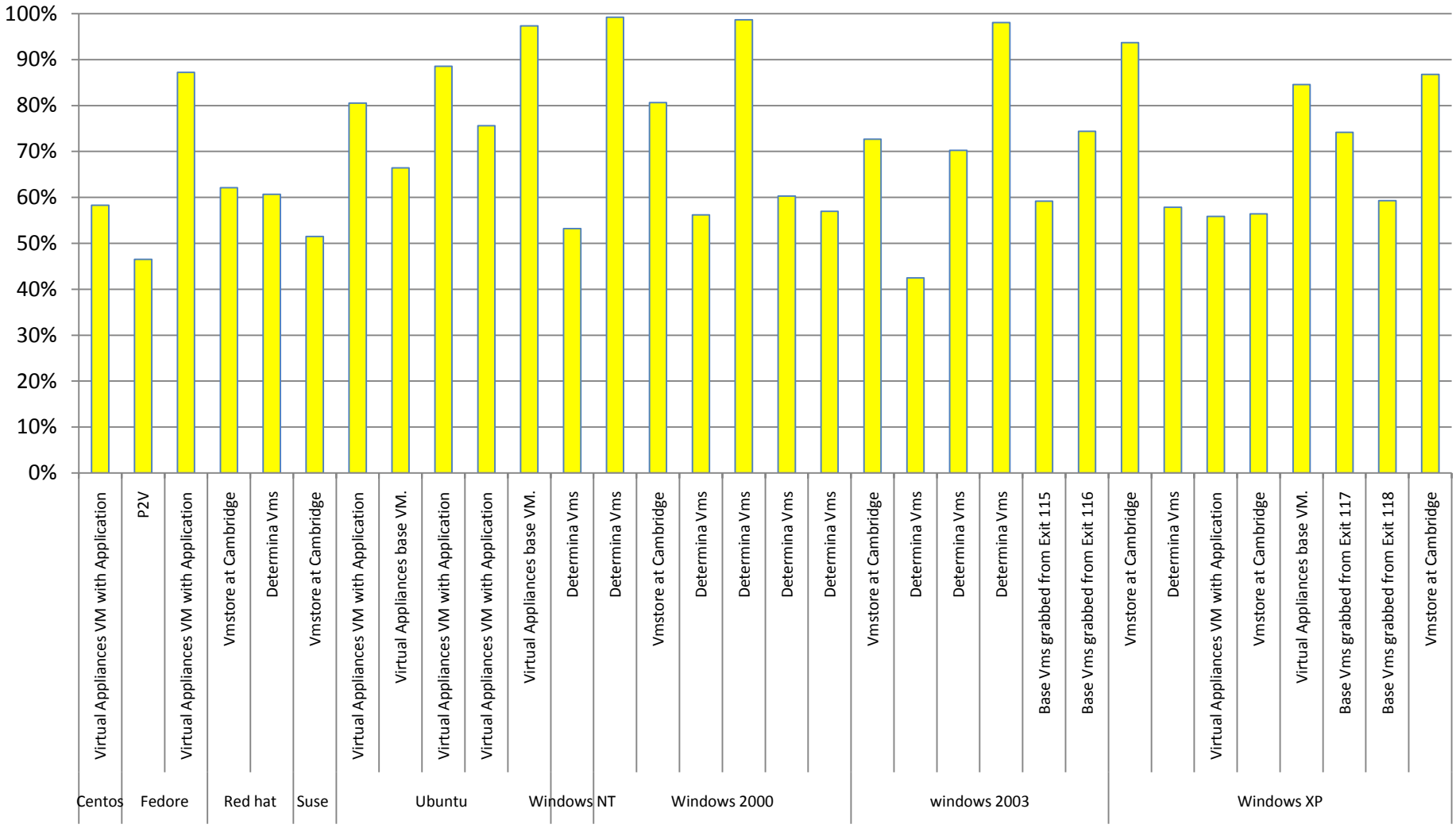
Efficiency

- UID layout has good special locality

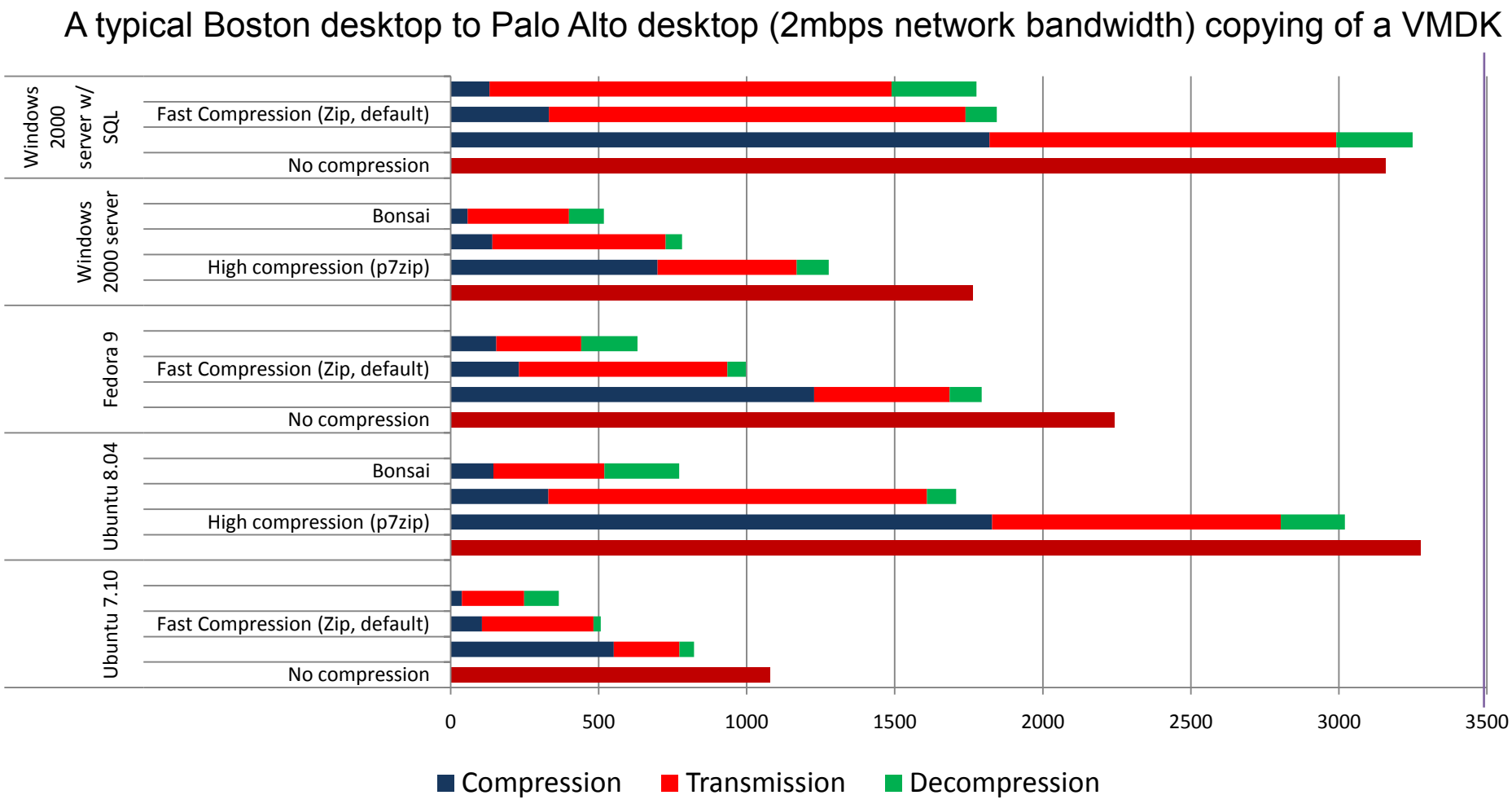
Integrity

- Central/global authority to assign UIDs
  - Guarantee block integrity and availability

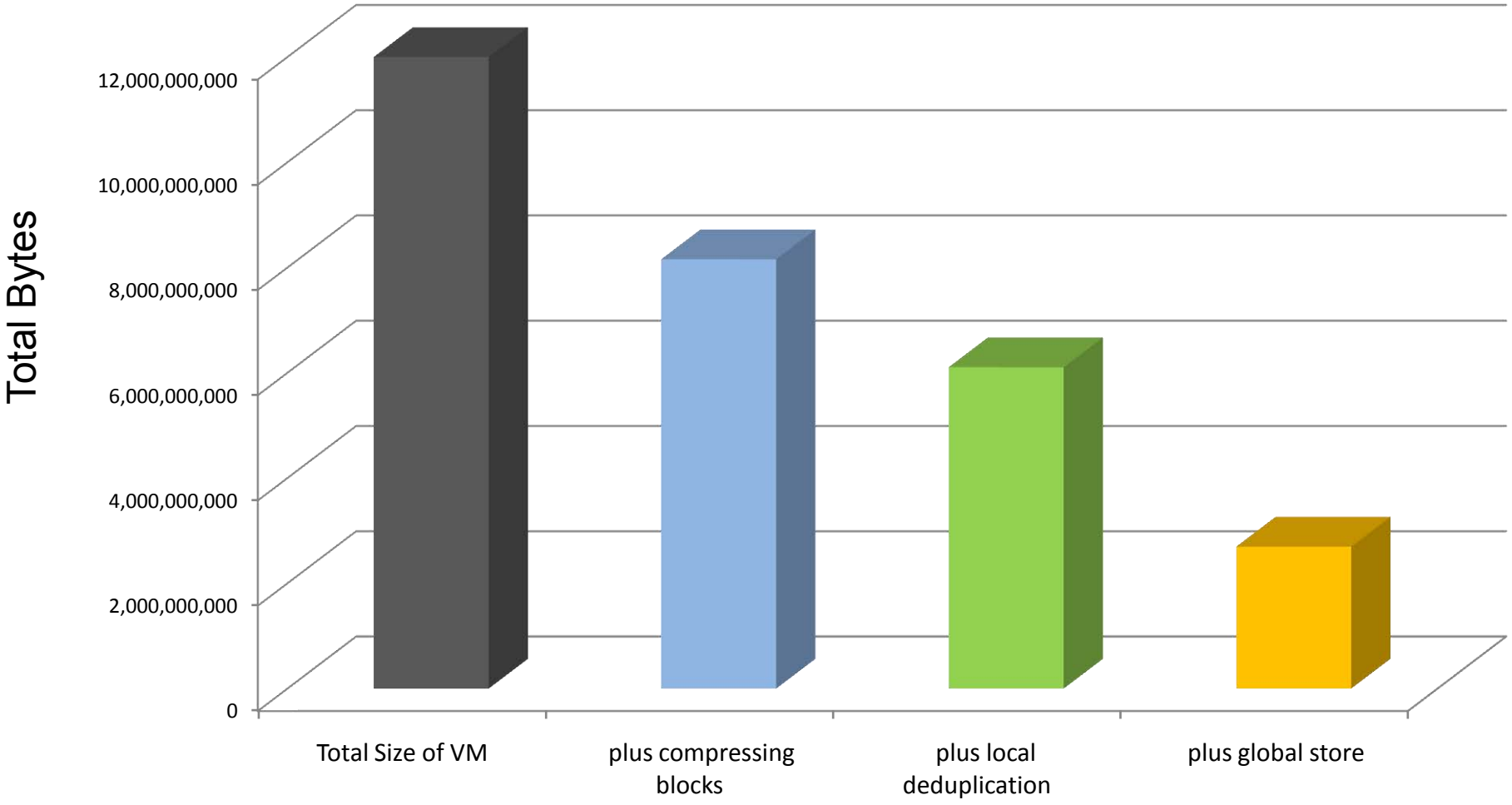
Compression Ratios



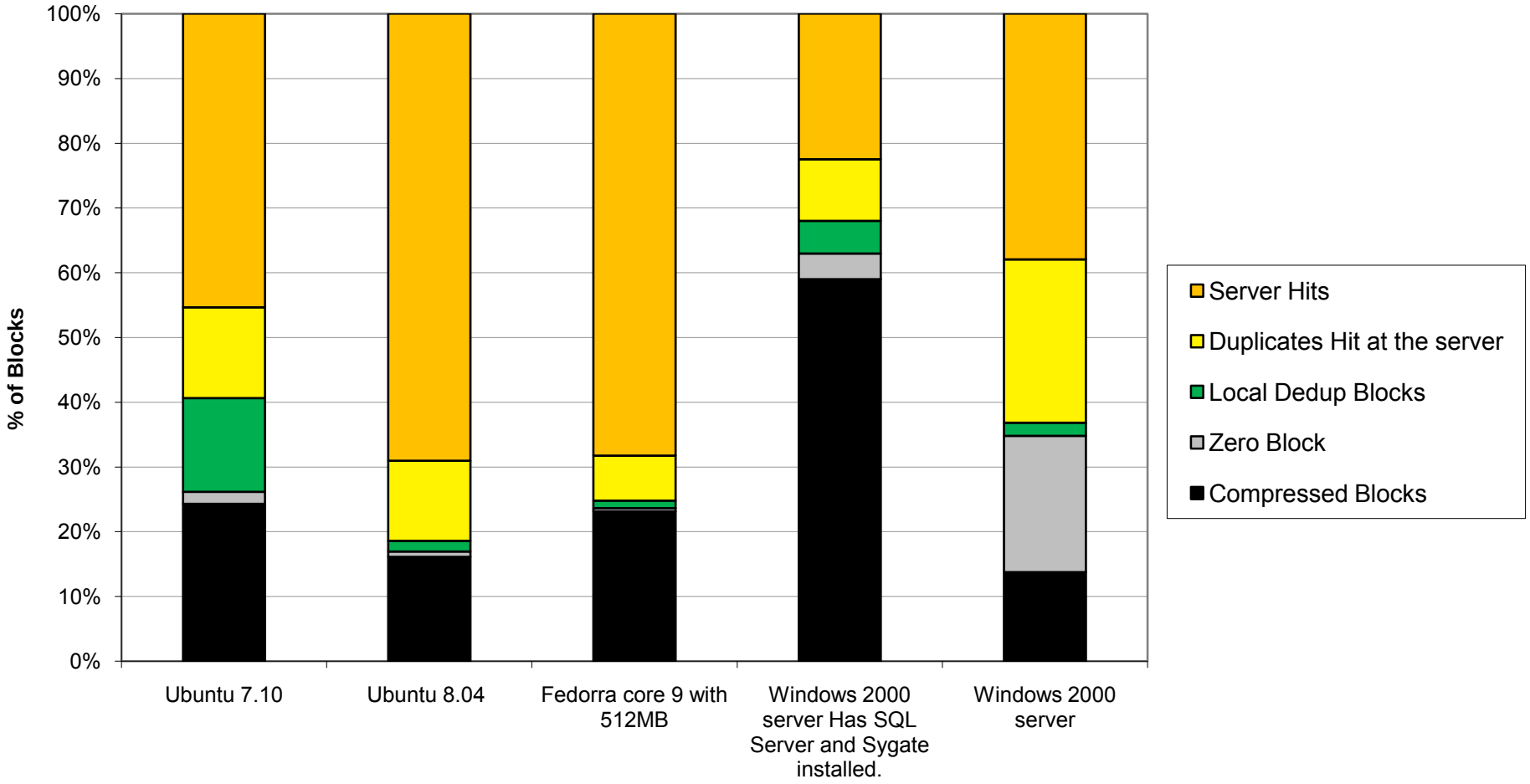
End-to-End Time to Move a VMDK



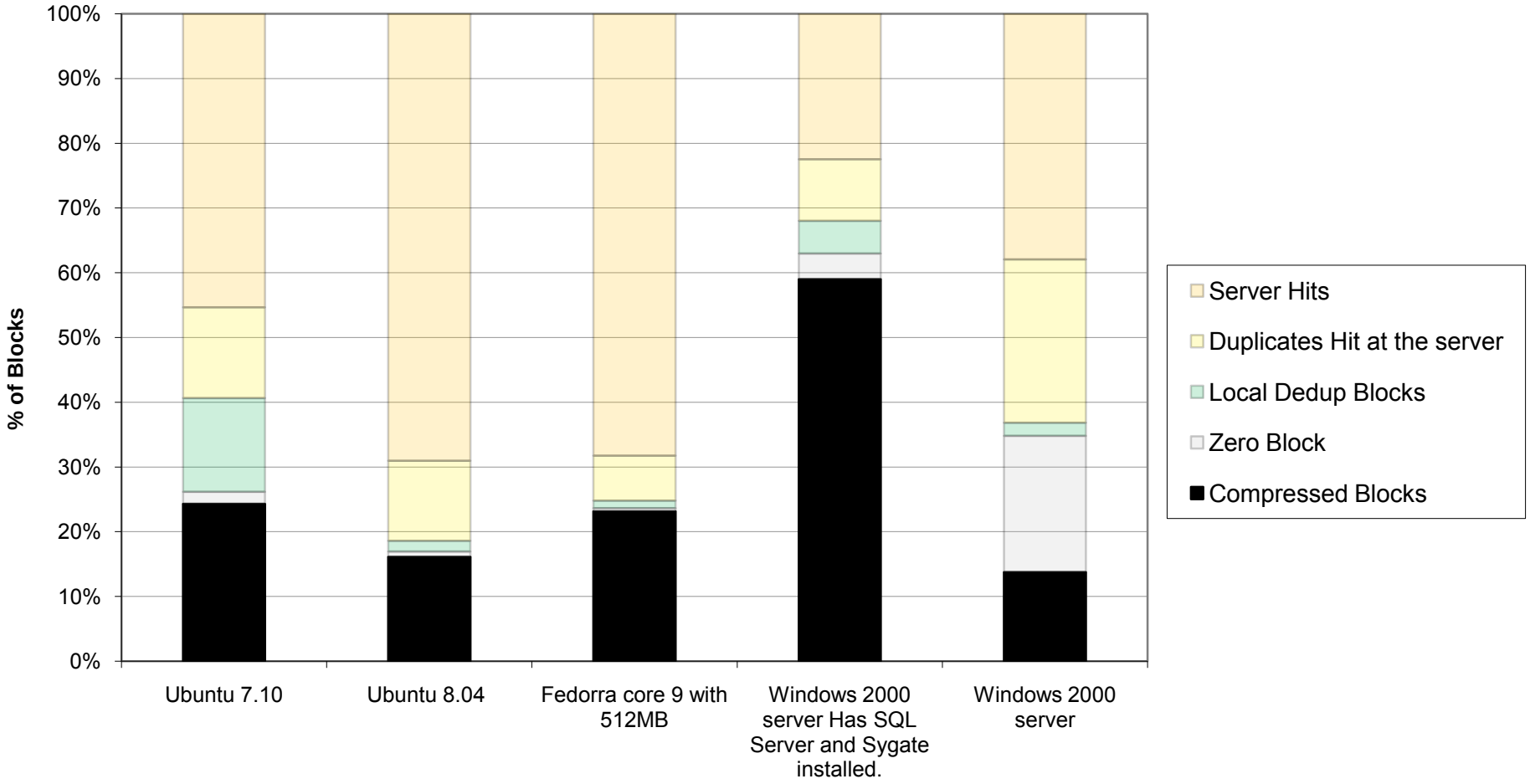
Different Levels of Compression



Contribution of Each Component to Compression



Contribution of Each Component to Compression



Size of the compressed blocks > 99% of the size of the Bonsai VMDK

Technical Challenges

Adaptive Store

Robust and Scalable Truly-Global Store

Integration with the Product Line

Improve the Compression Rate

Security and Privacy

6.172 Performance Engineering of Software Systems

Fall 2010