STORAGE AT "BIG DATA" SCALE

Professor Ken Birman
CS4414 Lecture 22

IDEA MAP FOR TODAY

Modern applications often work with big data

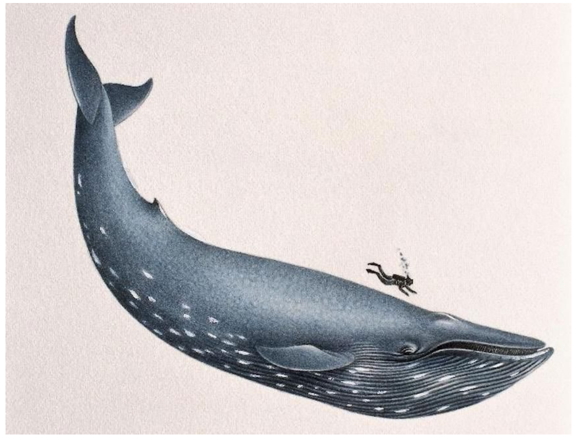
By definition, big data means "you can't fit it on your machine"

MemCacheD concept (a distributed version of std::map)

Hot and cold spots

Analogy to a distributed file system (and differences)

BIG DATA… CAN BE HUGE!



A single computer can hold gigabytes of data in memory, and many gigabytes on a local storage device.

But in modern AI/ML systems, like computer vision systems, we may need to train a model on huge data sets (like multiple photos of each student at Cornell).

It is easy to end up with data sets that won't fit on one computer.

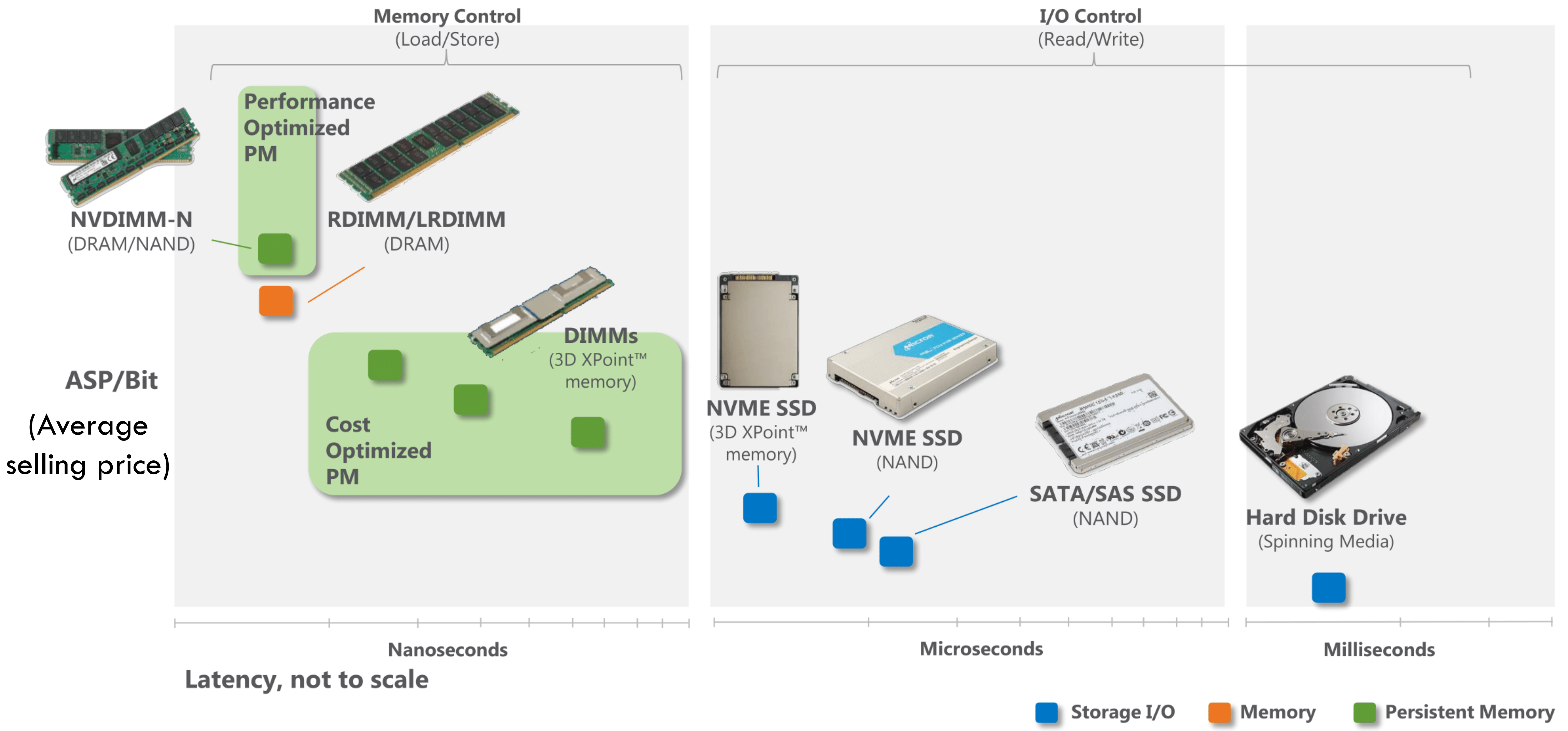
SPECIAL ISSUES WITH REALLY BIG DATA

Where does it live, physically?

If big data is *really* big, it might not fit even with hundreds or thousands of machines – the "full" data set may be on much larger (but slower) archival storage systems.

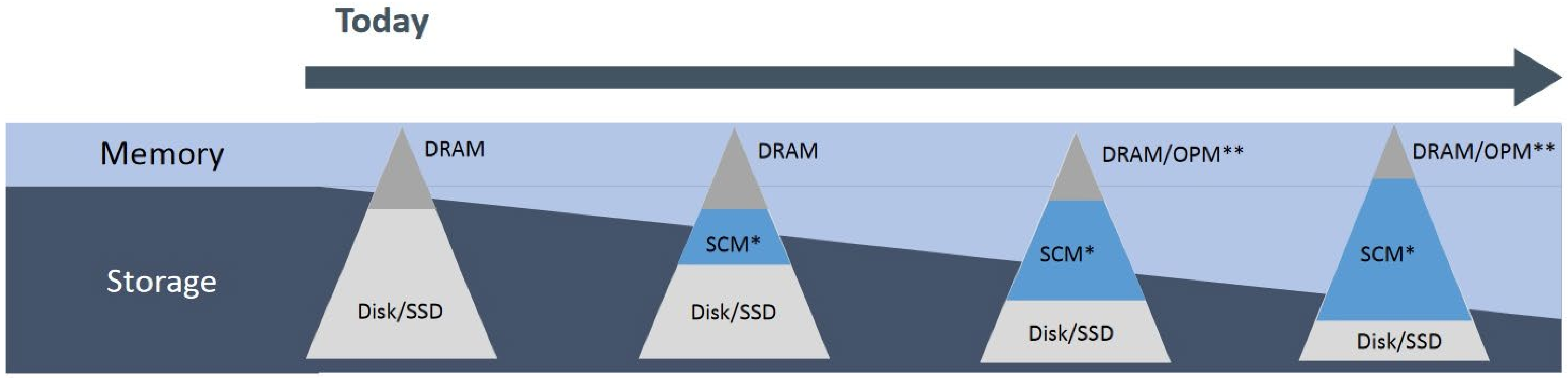
So delay for access becomes a big concern!

MEMORY/STORAGE TECHNOLOGY HIERARCHY



Memory/Storage Convergence: The Media Revolution

GEN Z



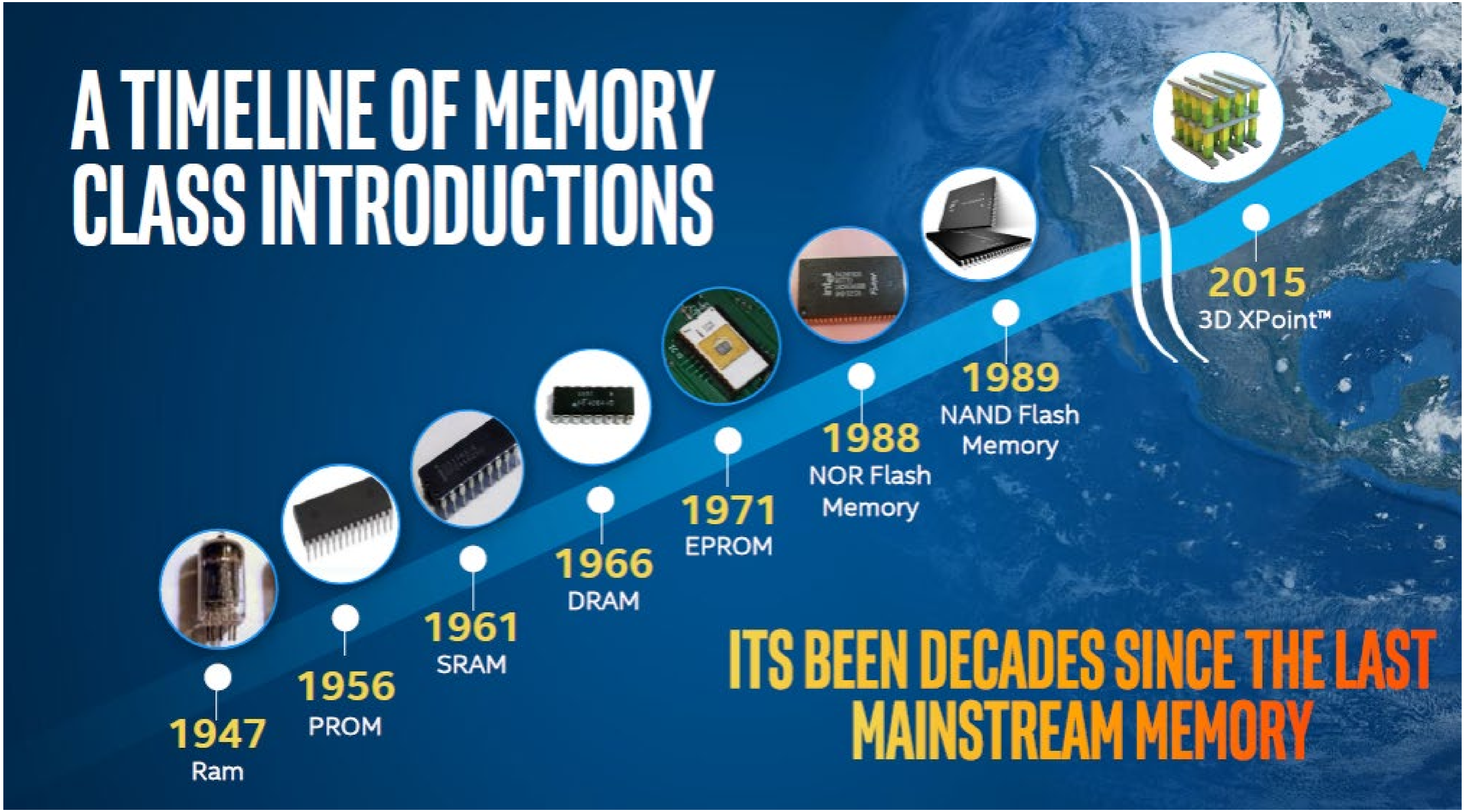
Memory Semantics will be pervasive in Volatile ***AND*** Non-Volatile Storage as these technologies continue to converge.

*SCM = Storage Class Memory
**OPM = On-Package Memory

New and Emerging Memory Technologies

HMC	3DXPoint™ Memory	Low Latency NAND
HBM	MRAM	
RRAM	PCM	Managed DRAM

10/11/2016 © Gen-Z Consortium 2016 4



A TIMELINE OF MEMORY CLASS INTRODUCTIONS

1947 Ram

1956 PROM

1961 SRAM

1966 DRAM

1971 EPROM

1988 NOR Flash Memory

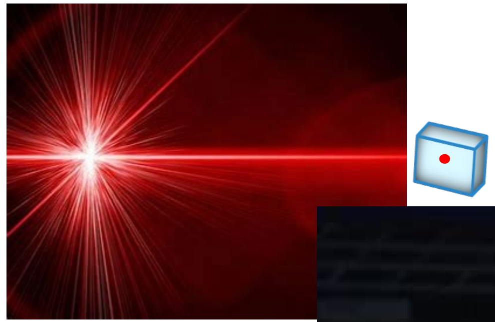
1989 NAND Flash Memory

2015 3D XPoint™

ITS BEEN DECADES SINCE THE LAST MAINSTREAM MEMORY

COMING SOON…

Laser "zaps" a tiny volume. It melts, then refreezes in a controlled way that can encode up to 6 bits per voxel



Glass (normal, inert silica) storage

Microsoft says that one cube could hold 360 terrabytes and survive for billions of years without degradation

This is Satya Nadella holding a sample

(OR MAYBE NOT SO SOON…)

DNA storage?

DNA has even more capacity! The data is encoded in powdered DNA, which is quite stable under ideal conditions. Nobody even knows what the capacity limits would be.

Reading data would require DNA sequencing hardware

GENERAL RULE…

The largest archival technologies are sometimes slow to access

Think of a “tape drive”. Incredible capacity, but you write it once and read rarely. When you do read, it can be slow.

Used for rarely accessed data, but at times highly valuable

GENERAL RULE

Memory is fastest, but as we saw, memory comes in a hierarchy

- In my address space, local NUMA memory
- In my address space, but remote NUMA memory
- On some other server, but in memory
- On my durable storage (flash memory or Optane). "The new disk"
- On the durable storage of some other server
- Archival storage… "Rotating disks are the new tape".

DOES IT MATTER?

There is roughly a 10x to 100x increase in delay and loss of bandwidth at each layer.

… this even includes the network delays of GRPC over a datacenter network. (For general applications, 100us, but for MemCacheD when heavily optimized, can drop to 25-30us)

So the value of having data in memory (somewhere) is huge!

CONNECTION TO SYSTEMS PROGRAMMING?

Up to now we focused on the single Linux box with NUMA cores, programmed with C++ processes and bash scripts and other tricks.

But the application really is a part of an ecosystem that could include many machines and the purpose may be to host and compute on huge amounts of data.

If our goal is efficiency and performance, we need to learn a new big-picture kind of perspective!

SPECIAL ISSUES WITH REALLY BIG DATA

Parallel computing is important, especially for AI/ML

But parallel algorithms really need data in memory, or "nearby". Training a modern ML model can be infeasible if data is on a slower technology.

In our last lecture we talked about caching. In-memory remote caching offers an opportunity to use those ideas!

WHAT ARE SOME REALLY BIG DATA EXAMPLES?

Companies like Apple, Microsoft, Facebook, etc. learn a lot about their users over time.

This pool of data is enormous. It includes photos, videos, cross-linked information about purchases and "click interests", friends and fans and where you live and what stores are nearby…

So this is one of the main big data use cases today.

MORE EXAMPLES

The entire web (and the “deep web”, too)

- The web would include all the web pages we can reach
- The “deep web” is the world of next-level and further pages you can reach by clicking things, or that are specialized for individuals. It also includes product prices, which are a big deal for companies!
- The web evolves, and for many organizations we also keep old copies of everything (the “Internet Archive” time machine does this too)
- Beyond all of this, the deep web also includes books and their contents, newspapers and other forms of information, etc…

MORE EXAMPLES

Think about astronomy, or particle physics, or gravity waves

The detectors often are worldwide structures, and some capture insane amounts of data, too much to process even with massive parallelism!

WHAT ABOUT THE FUTURE WORLD OF IOT?

The term is short for “Internet of Things”, often written IoT

For example, smart traffic intersections linked to create a smart city. Or smart homes that form a smart community.

- The houses could have lots of solar grids on their roofs
- If they join forces, they might produce a *lot* of electricity. And if they have batteries, we could store some, too.

All of this data (images, video, “lidar”, tracking data, “physical data”)…

SMART CITY
BUILDING TOMORROW'S CITIES
INDUSTRY
SECURITY
RETAIL
SOCIETY
HEALTHCARE
HOME
ENERGY
MOBILITY

FOR THIS LECTURE WE'LL FOCUS ON MEMCACHED

Memcached was born to "respond" to this big data need, and gave rise to a whole way of thinking about data storage and access at scale.

Companies like Facebook and Google were first to embrace it.

The idea seems trivial but gave rise to a whole world of parallel algorithms for computing on data spread over millions of computers.

CLOUD COMPUTING SOLUTIONS

The basic idea of the cloud is that someone like Amazon or Microsoft (Azure) runs a giant computing center, and you rent some of the machines in a “virtual private cluster”.

Your application basically owns this infrastructure, but you don’t have to build everything from scratch.

They offer services that are tuned to work really well at scale.

MEMCACHED CONCEPT

Originated in the 2003-2005 period.

Every programming language has some form of quick lookup class, based on the idea of hashing or a tree structure.

This suggests that we could take a very minimal API and standardize it for big data.

MEMCACHED CONCEPT (MEMORY CACHE DAEMON)

The (entire!) API of MemCacheD:

MemCacheD::put(string key, object value)

object = MemCacheD::get(key)

Put saves a copy of the pair (key,value), replacing prior value.
Get will fetch the object, if it can be found.

ISN'T THIS JUST A STD::UNORDERED_MAP?

C++ has a data structure that definitely can support the MemCacheD API.

The main difference is that the std::unordered_map is on a single computer, and is a C++ solution. Memcached might be on many computers in a data center, and is useful from many languages. Everyone "agrees" on the API.

KEY ASPECT?

MemCacheD must give “in memory” (perhaps over a fast network) performance. The data could be on durable storage as a fall-back, but everything should have a flat cost for reads.

But… a cache doesn’t need to “remember” everything. Objects can be evicted to make room.

When MemCacheD does get a cache hit, the performance should be blazingly fast. O(1) lookups: GRPC overhead + data transfer cost.

MEMCACHED CAN BE LOCAL (USEFUL WHEN DEVELOPING NEW CODE)

As noted, C++ std::unordered_map has a similar API and would be a great match to the MemCacheD standard. (std::map has an O(log) lookup cost, but std::unordered_map is O(1)).

But no single-computer solution can hold a really big data set. Your single computer only has a few 10's of GB of memory

REMOTE MEMCACHED RUNS AS A "DAEMON"

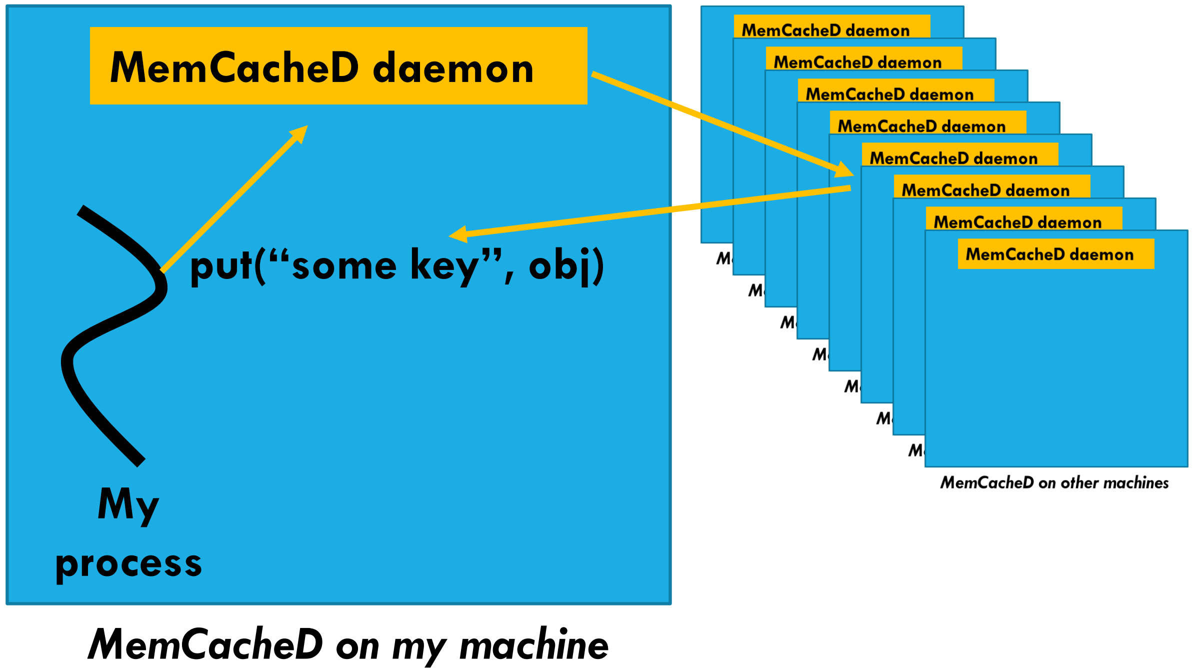
The idea is that your computer will have a way to use RPC to talk to a *pool* of Memcached servers, all automatic so that you won't need to do anything special to set this up.

The actual servers would run on cloud computing machines. The API is exactly the same. But now you get the total memory of the complete pool of machines!

A POOL OF DAEMONS…

You issue requests via your local daemon.

… but it might forward to some other daemon in the pool

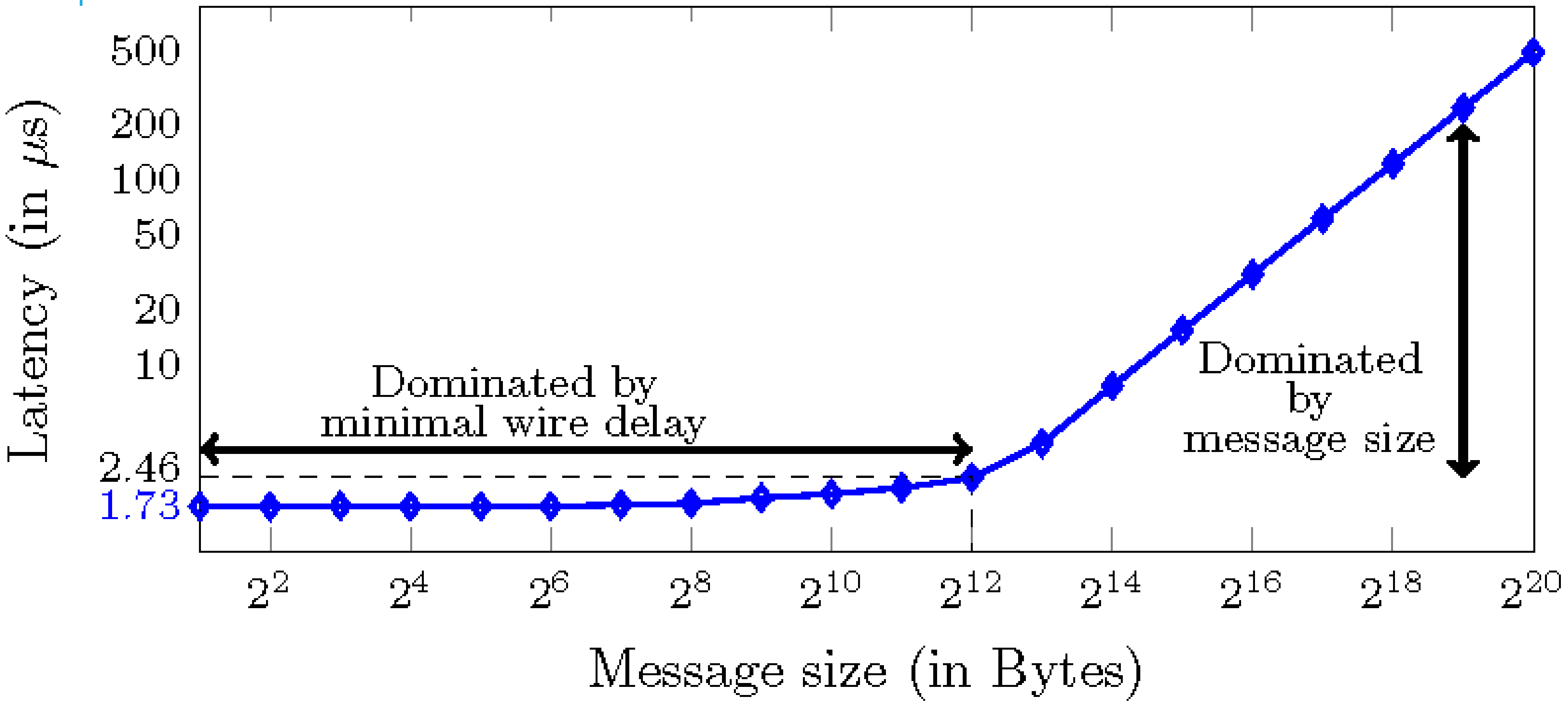


WON'T THE NETWORK BE TOO SLOW?

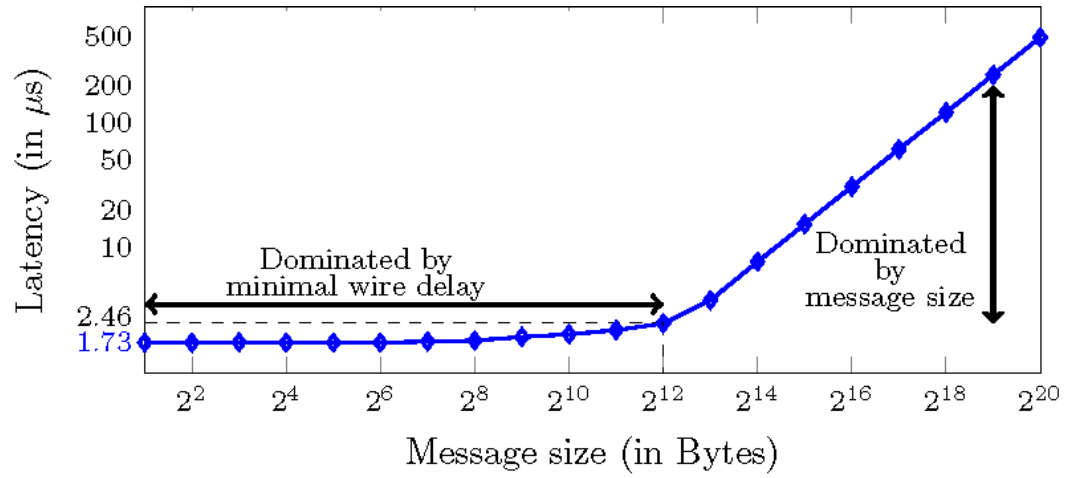
In fact a modern datacenter network runs at speeds similar to the internal "bus" between your NUMA core and one of the on-board but non-local DRAM modules.

- The only issue is that although data transfer speeds are high, delay can be a barrier.
- A modern datacenter network might have minimal delays of 1us. In contrast, accessing a DRAM module that isn't close to your core might be 125 clock cycles: about 25x faster.

SPEED OF REMOTE ACCESS IN A DATA CENTER



SO?



This tells us that MemCacheD will be awesome for large objects, like images or web pages: the overheads of getting to the server will be small compared to the data transfer times.

In contrast, if you are storing tiny objects, you might notice the delays much more, because they will be more dominating than the data transfer time.

HOW DO THEY TRACK THE MACHINES IN THE POOL OF MEMCACHED SERVERS?

You don’t really see this, but the daemon would use Zookeeper or something similar.

When you link to the MemCacheD DLL and initialize it, the library will look up the current membership of the service.

Now it has a list of all the machines running MemCacheD (IP address and port numbers for each).

WHICH MACHINE TO TALK TO?

In contrast with the case where we have an in-memory data structure, like a std::unordered_map in your own process, MemCacheD has an extra step of finding the server with the map that has your data.

Each MemCacheD server will have many data objects. We try to spread the data evenly.

So we use the concept of "hashing a key", but we do it twice.

THE USUAL APPROACH

Hashing functions are standard and built into C++.

Hash the key of the (key,value) pair, mod the number of servers

```
server_id = std::hash<std::string>{}(key) % NSERVERS;
```

Now just do an RPC (like GRPC) to the server you picked!

EXAMPLE

Perhaps your application uses keys that are strings with pathnames deliberately similar to file system paths.

/users/Alicia/SmartFarm/Animals/CowImages/Cow76512

Hash will convert this string to a number, like 0x6AF615B80DDA71C

Normally we use a uint_64 for these hash values.

STD::HASH

std::hash defines a set of templated methods, which leads to a slightly weird C++ syntax:

```
uint_64 h1 = std::hash<std::string>{}(something);
```

This part actually “generates” a templated method for hashing a std::string. Then we invoke that method and pass a string into it. That explains the {} notation.

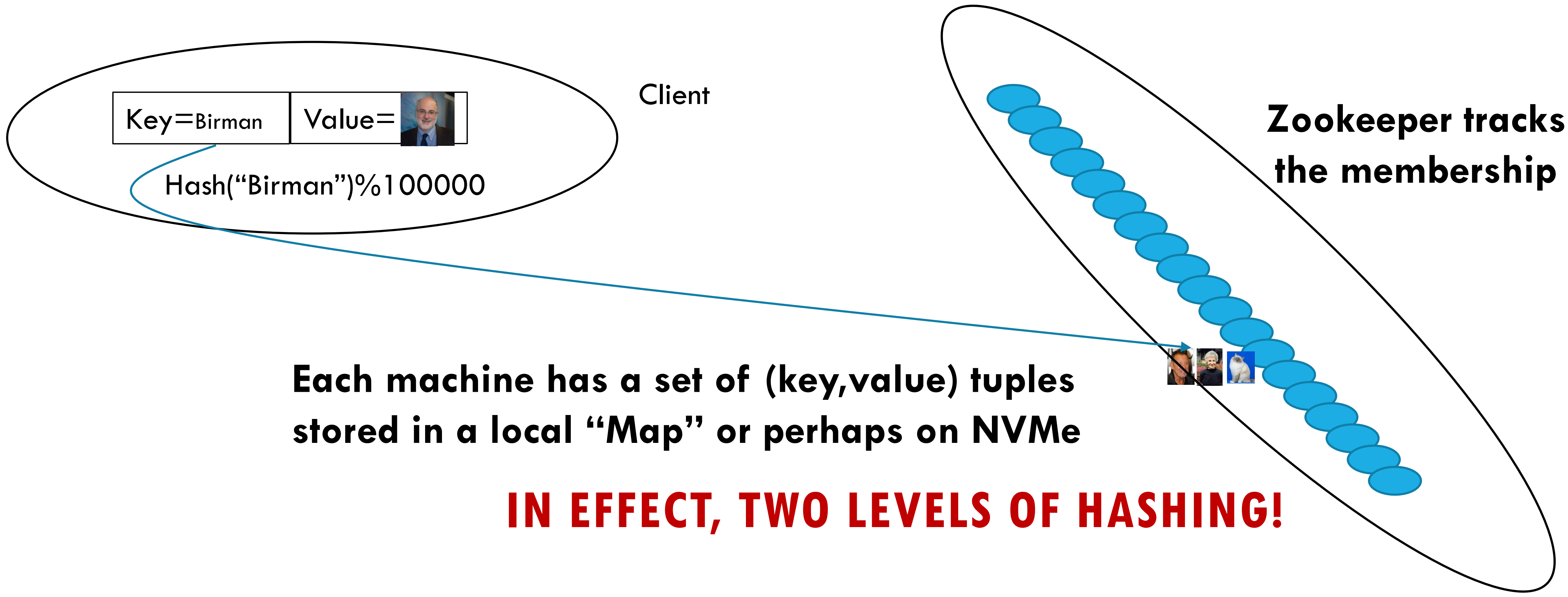
MANY KEYS MAP TO ANY SINGLE SERVER

With this approach, two very different keys might map to exactly the same server-id.

This is why the server will still need to treat the data as a pool of (key,value) pairs: it will probably use std::unordered_map!

- One O(1) hashing operation to decide which server to talk to
- One more, inside that server, to find the object
- Overhead of an RPC to send the request, and get the reply

ACCESSING (KEY,VALUE) STORAGE



Zookeeper tracks the membership

Each machine has a set of (key,value) tuples stored in a local "Map" or perhaps on NVMe

IN EFFECT, TWO LEVELS OF HASHING!

THIS IS NOT *ALWAYS* BENEFICIAL

In total, MemCacheD can hold a huge amount of data in memory

Durable storage I/O delays are larger than GRPC delays, and if we are lucky and find our data, we avoid reading or recreating the object from a file.

Payoff is best for large objects that were retrieved from some sort of expensive database service or slow media.

WHAT WOULD BE AN "EXPENSIVE" OBJECT IDEAL FOR MEMCACHED?

Think about a photo or video that someone wants to view

Each device might have its own preferred screen size.

- Resizing on the device runs the battery down
- Resizing on the cloud is a good option, but once you have the object caching it will let you avoid recreating it again and again.

WHAT MAKES THIS SO EFFECTIVE?

Many clients share the MemCacheD service: parallelism!

The service itself can hold all of those resized objects in fast memory units (or perhaps even on local storage)

Resizing is slow and energy-intensive. Transmitting on a network is quite cheap.

WHAT WOULD BE A “BAD USE” OF MEMCACHED?

It may not pay off to cache small things, like a few blocks of a file. Using memcached just wastes resources in such cases

The file system itself has effective caching mechanisms

Moreover, if the file system prefetcher anticipates the access, data might be in local cache before you request it.

MEMCACHED IS REALLY FOR TEMPORARY DATA

The intent was to create a big data cache, not a new file system.

A Memcached server is allowed to evict data if it needs room.

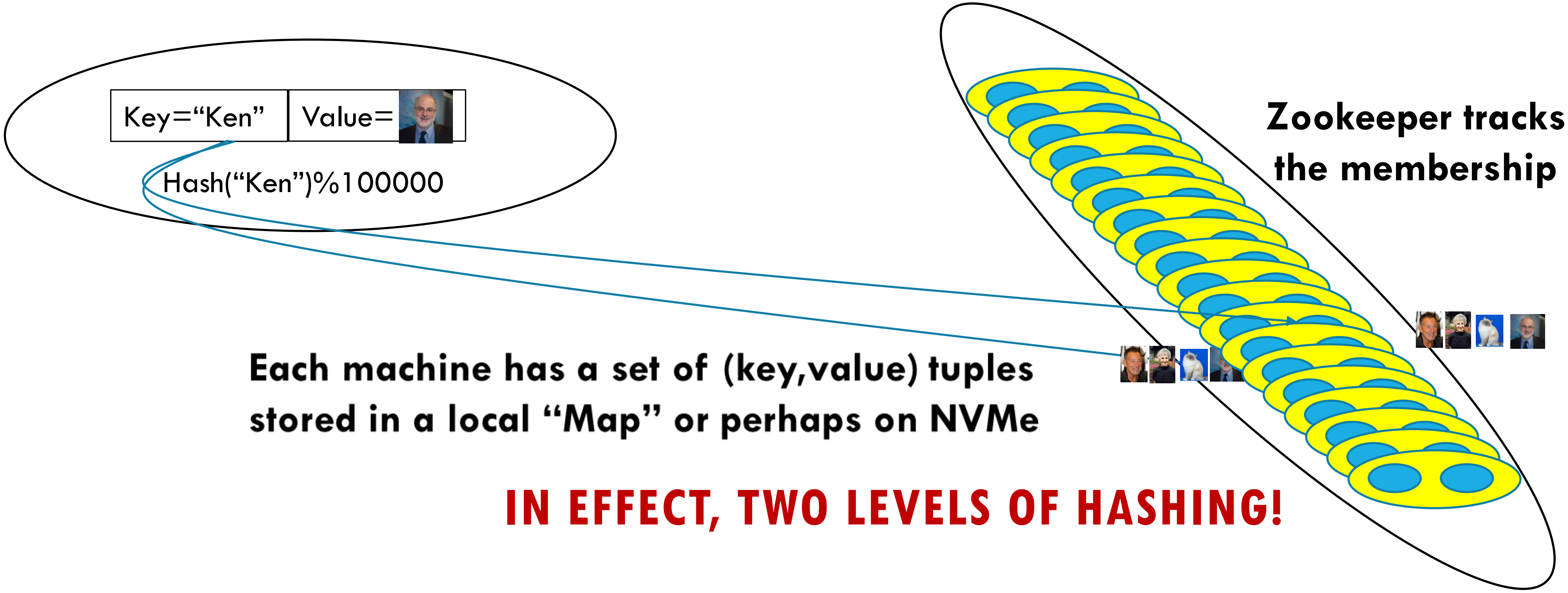
… even if a process used **put** to store some (key,value) pair, don't assume that **get** will find that object later!

CAN WE DO BETTER?

This is a bit beyond CS4414, but we can use data replication to build a MemCacheD that probably won't lose data if a server fails or a network link is transiently flaky.

For example, AWS (Amazon) has a MemCacheD service called DynamoDB. It replicates every (key,value) tuple so that even in a crash, data generally won't be lost. You can access it as a kind of database, which is why they added the "DB" part.

WITH SOME SOLUTIONS, WE CAN REQUEST REPLICATION FOR IMPROVED AVAILABILITY



Each machine has a set of (key,value) tuples stored in a local "Map" or perhaps on NVMe

IN EFFECT, TWO LEVELS OF HASHING!

HOW DID THEY BUILD IT?

… not a CS4414 topic!

The puzzle is that doing data replication correctly is harder than you might expect, because failures can leave confusing states.

The problem can be solved, and in fact Cornell is famous for working on this. But the details are covered in CS5412, cloud computing.

WHAT MAKES IT HARD?

So far it probably sounds trivial: hash twice, use GRPC!

Data replication isn't particularly hard either: you just take the key, hash it to find the "primary" server, then "add 1" and hash this new key. That can be the "backup".

Each server ends up playing two roles.

BUT MEMBERSHIP CHANGES COMPLICATE THINGS!

When servers start up, or terminate (or crash), we need to repair the Memcached server.

So these is a whole issue here of adding a new server to replace one that failed or terminated.

We might also need to re-replicate data (this new server “should” have some data it will be missing)

THE WEAK SEMANTICS OF MEMCACHED ARE A STRENGTH AND A WEAKNESS

Memcached seems very simple… partly because it makes no real promise except that **get** returns “something” from a prior **put**

Yet if we have no guarantee that **get** will find the most recent **put** results, we could actually see very old data… an “error”?

In fact staleness errors are rare (evictions are more common).

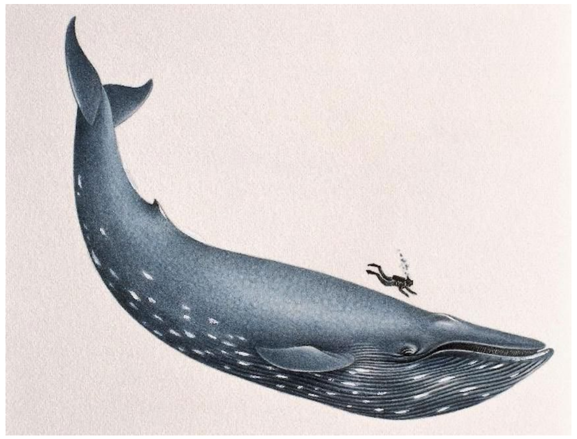
ARE THERE “COHERENT” MEMCACHED’S?

A coherent cache would guarantee that the last data put into is the value you read out, and that data won’t get lost or corrupted.

Coherent caches exist (Cornell created one!), but most products have very weak guarantees. And even so, they are complex.

There is a tension between wanting to treat remote storage solutions as local memory, and wanting them to be super fast, lightweight…

SUMMARY



We are increasingly faced with really big data scenarios

A popular option is to use a platform (like a cloud) that offers big data storage services.

Sometimes the real data would be on a slow device. But we can cache large "slices" of it in a key-value store, like MemCacheD.