```
                        c.h>
                        lloon.h>
def struct {
    int..               mode;
    ScePspFVector3 .    pos;
    ScePspFVector3 .    sbuf[3];
    int..               scnt;
    float.              t;
} BALLOONDAT;
static BALLOONDAT       balloon;
static ScePspFVector3   sphere[20];
static ScePspFVector3   pole[20];
extern void DrawSphere(ScePspFVector3 *array,float r);
extern void DrawPole(ScePspFVector3 *array,float r);
void init_balloon(void)
{
    int..   i;

    balloon.mode=MODE
    balloon.pos.x= 0.
    balloon.pos.y=-8.
    balloon.pos.z= 0.
    balloon.t=0.0f;
    balloon.scnt=2;

    for (i=0; i<3; i+
        balloon.sbuf[
        balloon.sbuf[
        balloon.sbuf[
    }
}
void draw_balloon(void)
{
    ScePspFVector3 vec;
    able(SCEGU_TEXTURE);
    ();
    loon.pos);
```

Operating Systems and C
Fall 2022, Performance-Track
6. Locality

Parallel Tracks

exciting part of the course!

performance-track lecture nr. 1:
what you'll need for perflab.
array layout, what it means for performance, cache hierarchy, how associativity is organized in the cache. (important stuff for anyone)

perflab:

- have two matrix multiplication procedures (rotate, smooth), have to rewrite it.
- about optimization techniques; blocking, loop unrolling, etc.

performance-track lecture nr. 2:
what you'll need for perflab.
optimizations. how to write code so compiler can derive performant code. manual transformations, blocking, loop unrolling

attacklab:

- you have an executable, have to attack it.
- about the stack; code injection (smashing the stack), return-oriented programming (find interesting code in other programs).

security-track lecture nr. 1:
what you'll need for attacklab.

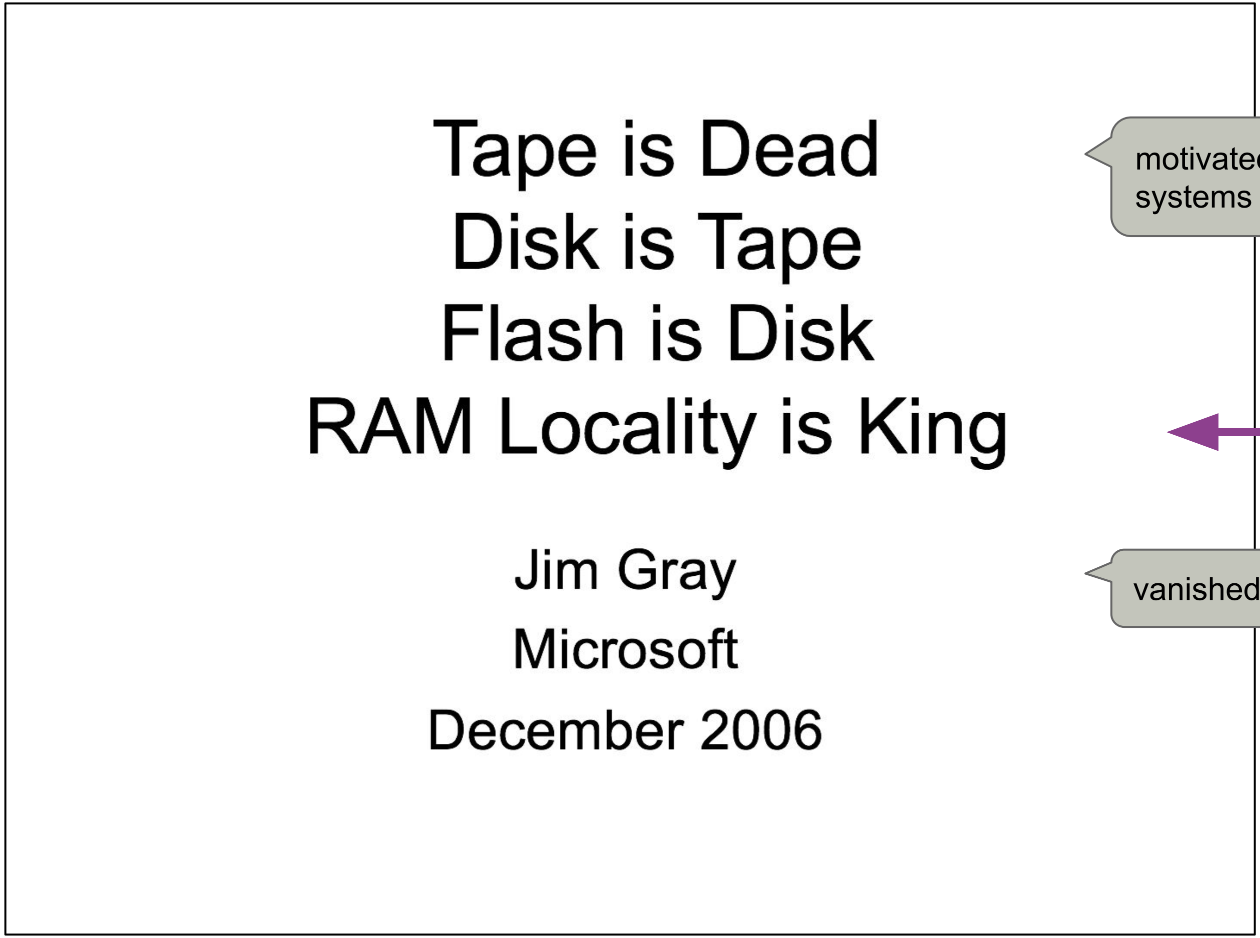
security-track lecture nr. 2:
Linux culture.

Outline

- Locality
- Memory Hierarchy
- Cache Utilization
- A note on Security

Jim Gray, 2006

http://jimgray.azurewebsites.net/jimgraytalks.htm



Tape is Dead
Disk is Tape
Flash is Disk
RAM Locality is King

Jim Gray
Microsoft
December 2006

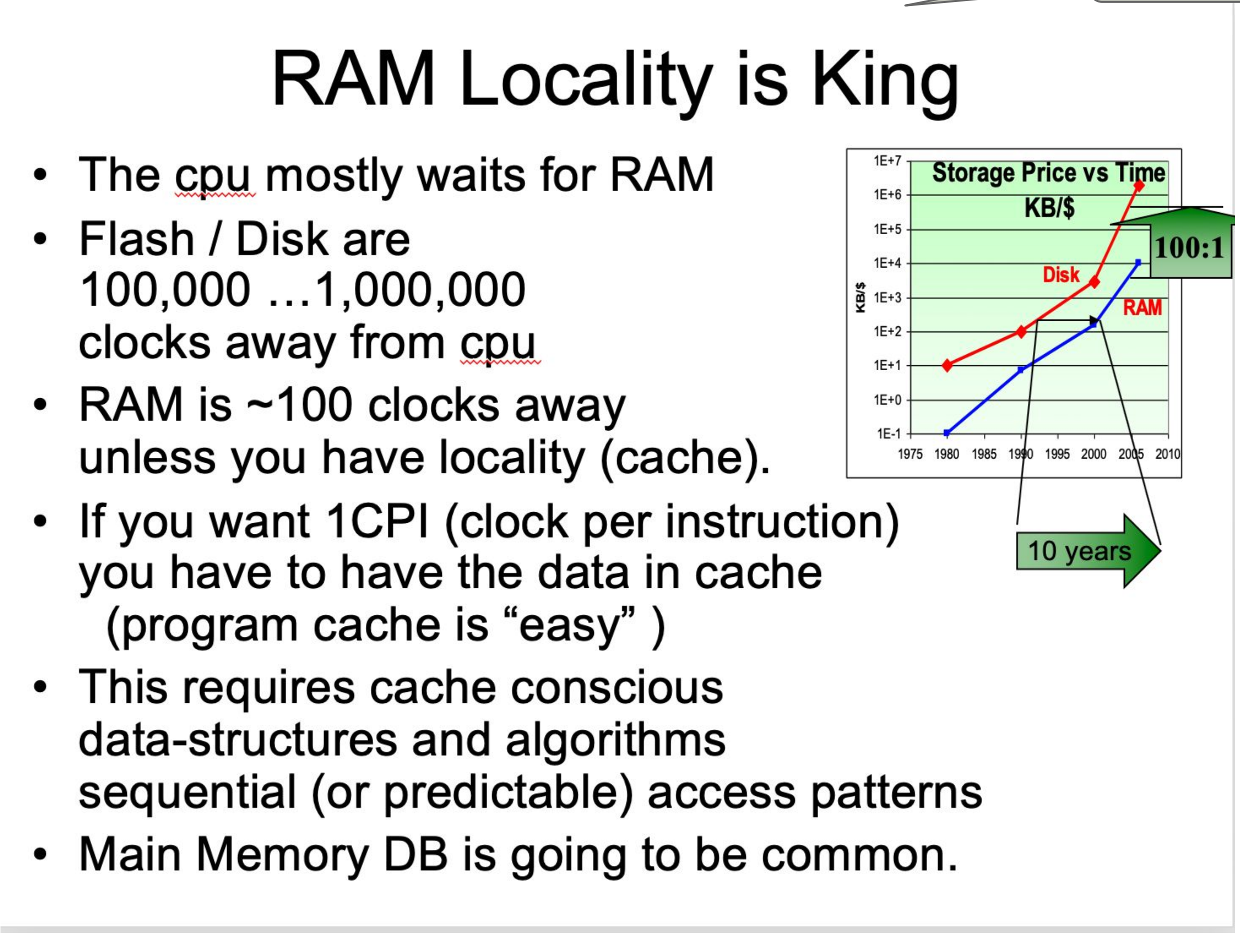
motivated & driven a lot of systems research since.

vanished w/o trace in 2007

Jim Gray, 2006

http://jimgray.azurewebsites.net/jimgraytalks.htm

layout of data in memory



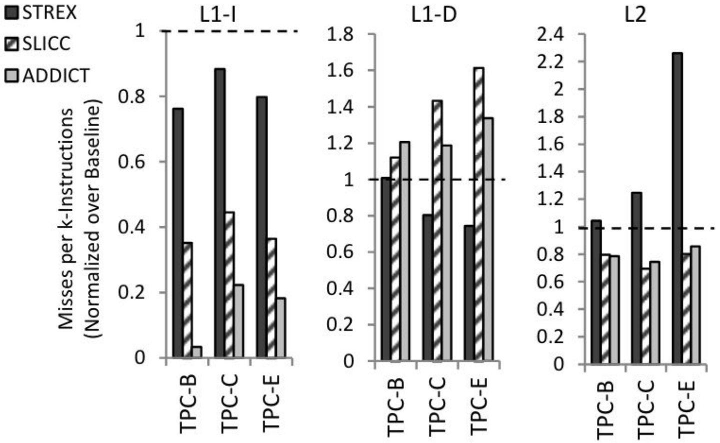
RAM Locality is King

- The cpu mostly waits for RAM
- Flash / Disk are 100,000 …1,000,000 clocks away from cpu
- RAM is ~100 clocks away unless you have locality (cache).
- If you want 1CPI (clock per instruction) you have to have the data in cache
  (program cache is "easy")
- This requires cache conscious data-structures and algorithms sequential (or predictable) access patterns
- Main Memory DB is going to be common.

Data Systems Group Promotion

microarchitectural analysis, benchmarking, …
how resources are used.
(how much time CPU spends waiting for memory?)

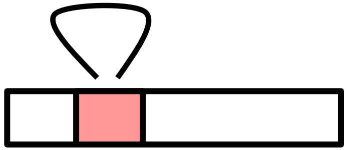
PINAR TÖZÜN
Associate Professor



Locality

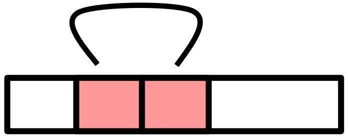
Principle of Locality: Programs tend to use data and instructions with addresses near or equal to those they have used recently

Temporal locality:



Recently referenced items are likely to be referenced again in the near future

Spatial locality:



Items with nearby addresses tend to be referenced close together in time

Locality Example

```
sum = 0;
for (i = 0; i < n; i++)
     sum += a[i];
return sum;
```

Data references

- Reference array elements in succession (stride-1 reference pattern).

  Spatial locality

- Reference variable `sum` each iteration.

  Temporal locality

Instruction references

- Reference instructions in sequence.

  Spatial locality

- Cycle through loop repeatedly.

  Temporal locality

Qualitative Estimates of Locality

Claim: Being able to look at code and get a qualitative sense of its locality is **a key skill for a professional programmer**.

Question: Does this function have good locality with respect to array `a`?

```
int sum_array_rows(int a[M][N])
{
    int i, j, sum = 0;

    for (i = 0; i < M; i++)
        for (j = 0; j < N; j++)
            sum += a[i][j];
    return sum;
}
```

```
int sum_array_cols(int a[M][N])
{
    int i, j, sum = 0;

    for (j = 0; j < N; j++)
        for (i = 0; i < M; i++)
            sum += a[i][j];
    return sum;
}
```

Q: which is faster?

Memory Hierarchies

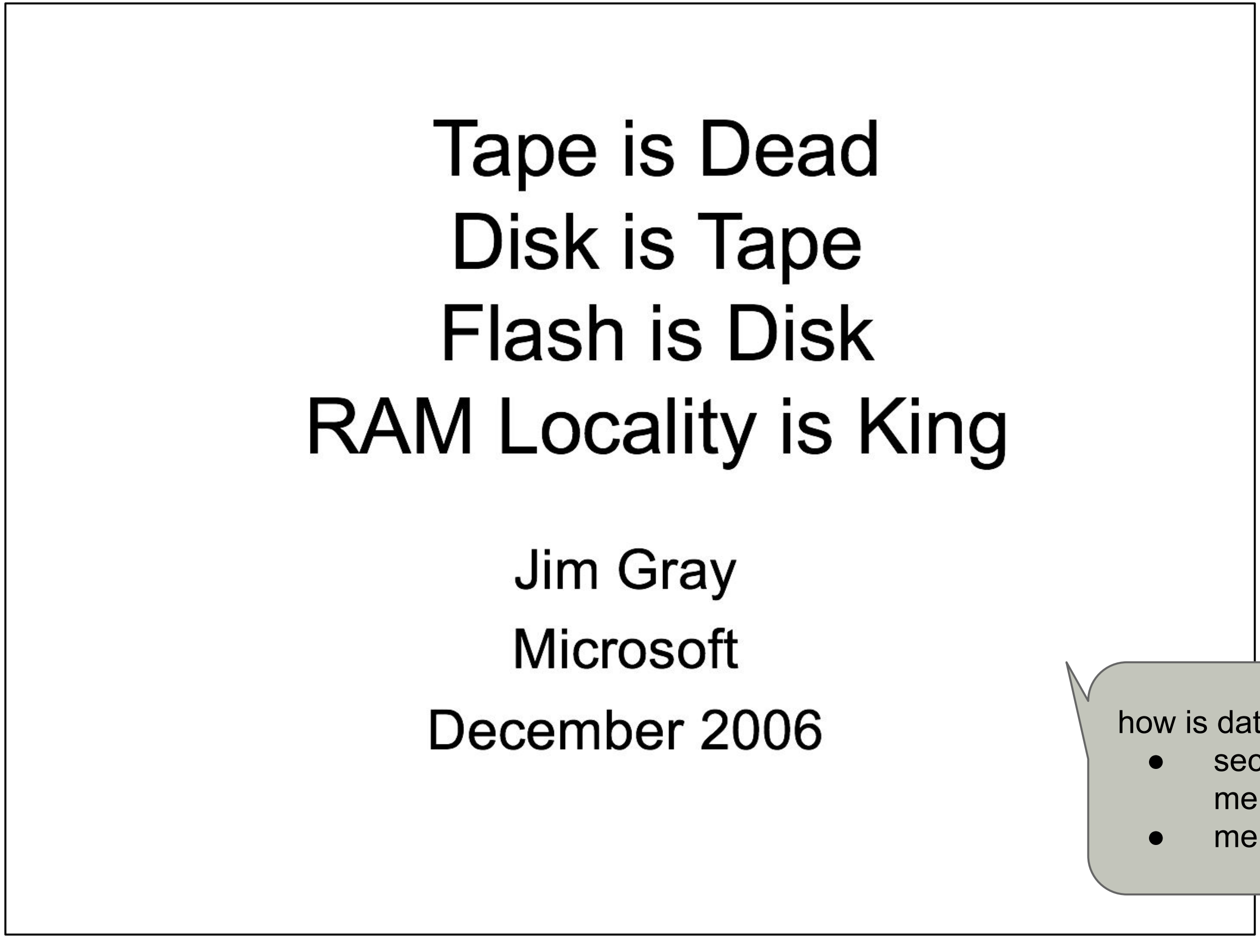
- Some fundamental and enduring properties of hardware and software:
  - Fast storage technologies cost more per byte, have less capacity, and require more power (heat!).
  - The gap between CPU and main memory *speed* is widening.
  - Well-written programs tend to exhibit good locality.

- These fundamental properties complement each other beautifully.

- They suggest an approach for organizing memory and storage systems known as a memory hierarchy.

Jim Gray, 2006

http://jimgray.azurewebsites.net/jimgraytalks.htm



Tape is Dead
Disk is Tape
Flash is Disk
RAM Locality is King

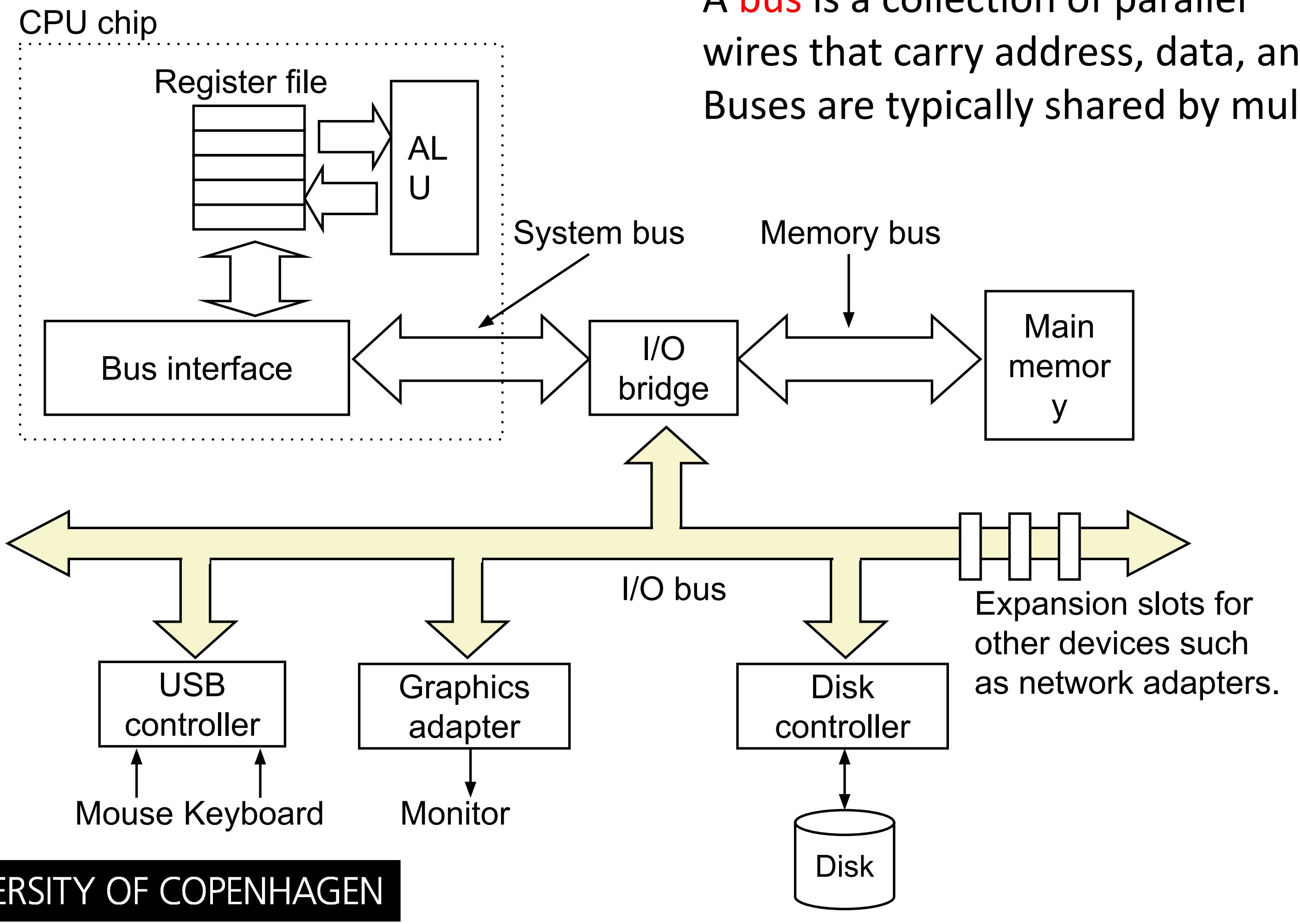
Jim Gray
Microsoft
December 2006

how is data transferred from

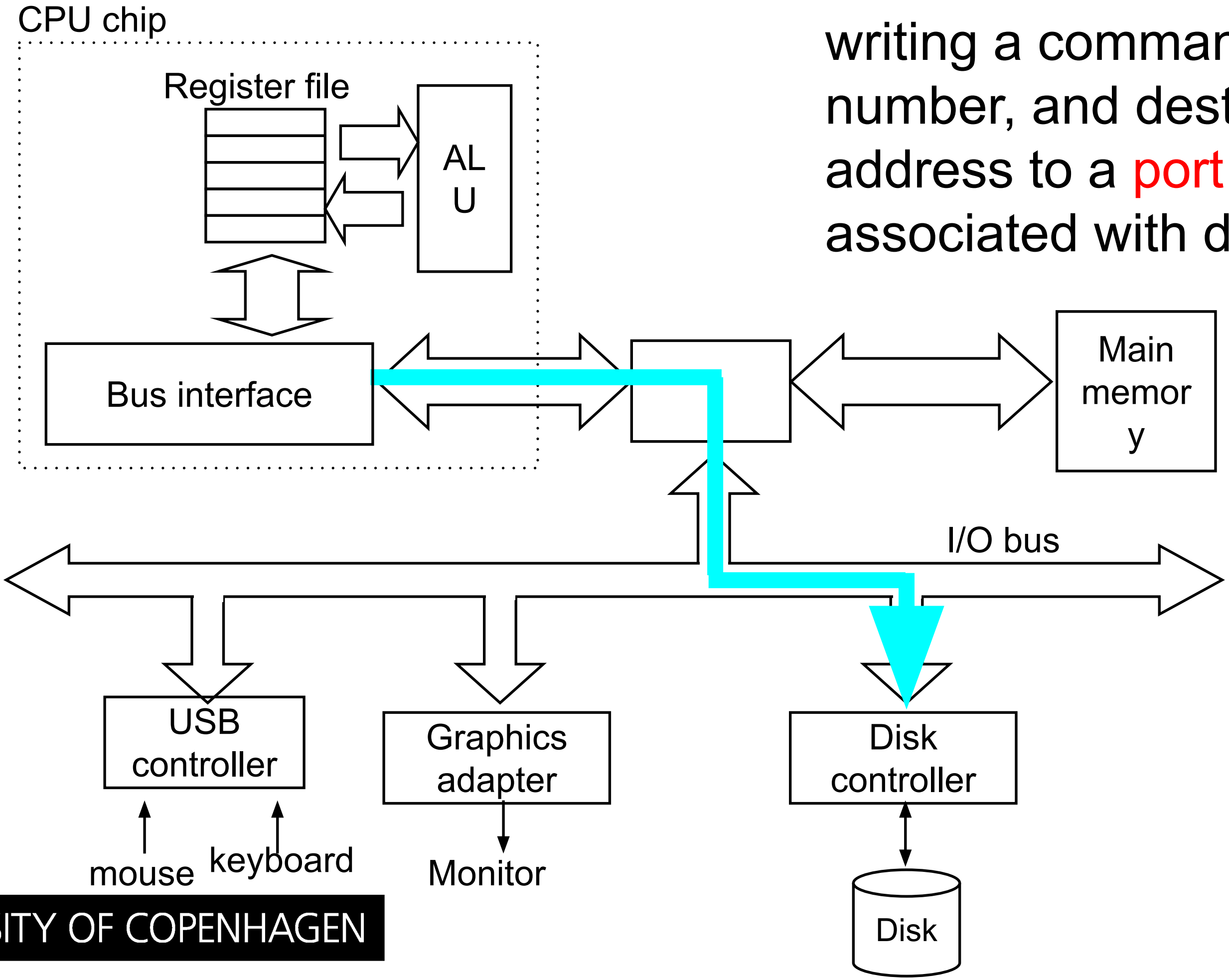
- secondary storage to memory, and
- memory to registers?

I/O Bus

A bus is a collection of parallel wires that carry address, data, and control signals. Buses are typically shared by multiple devices.

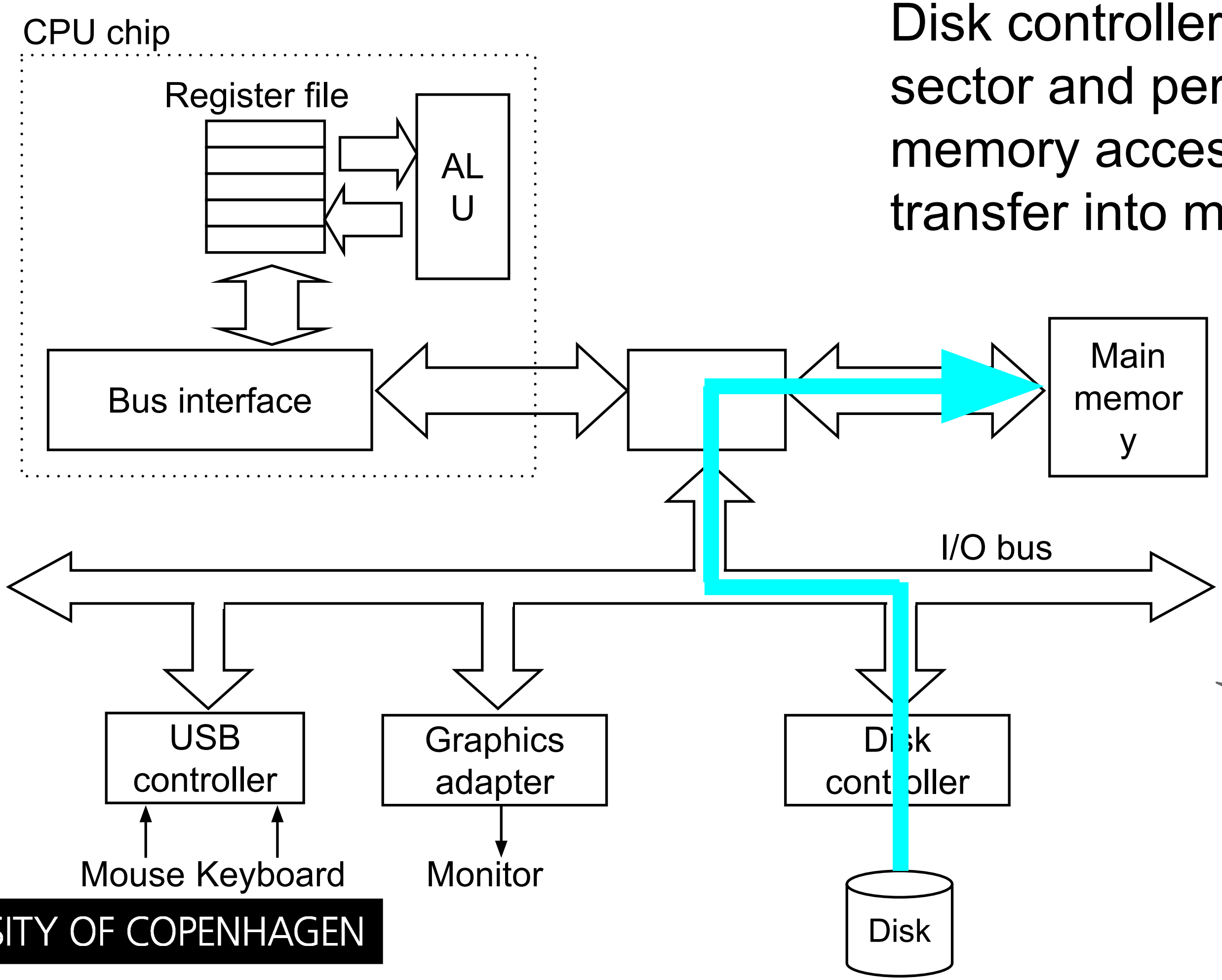


Reading a Disk Sector (1)



CPU initiates a disk read by writing a command, logical block number, and destination memory address to a port (address) associated with disk controller.

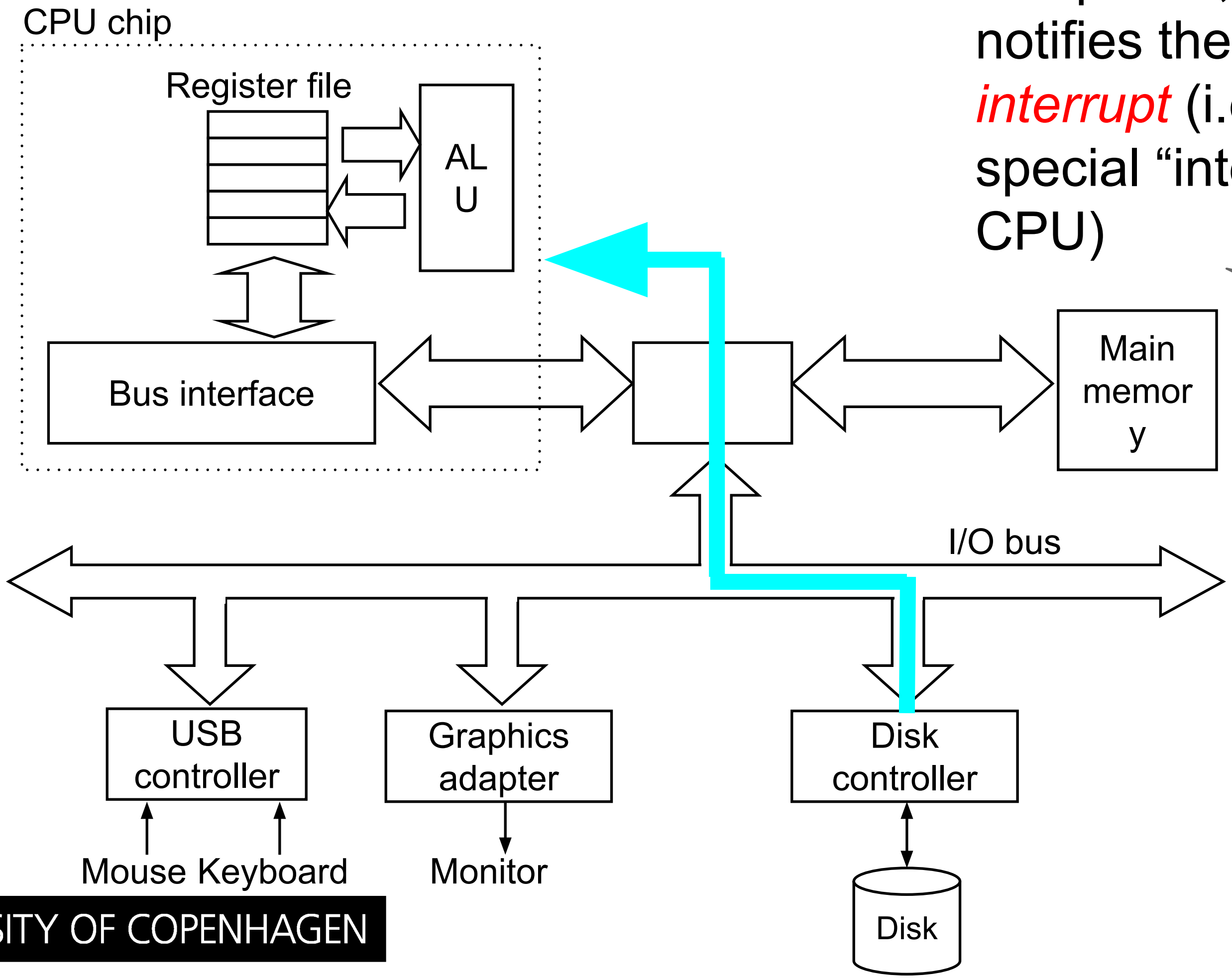
Reading a Disk Sector (2)



Disk controller reads the sector and performs a direct memory access (DMA) transfer into main memory.

note: CPU is not involved in this.

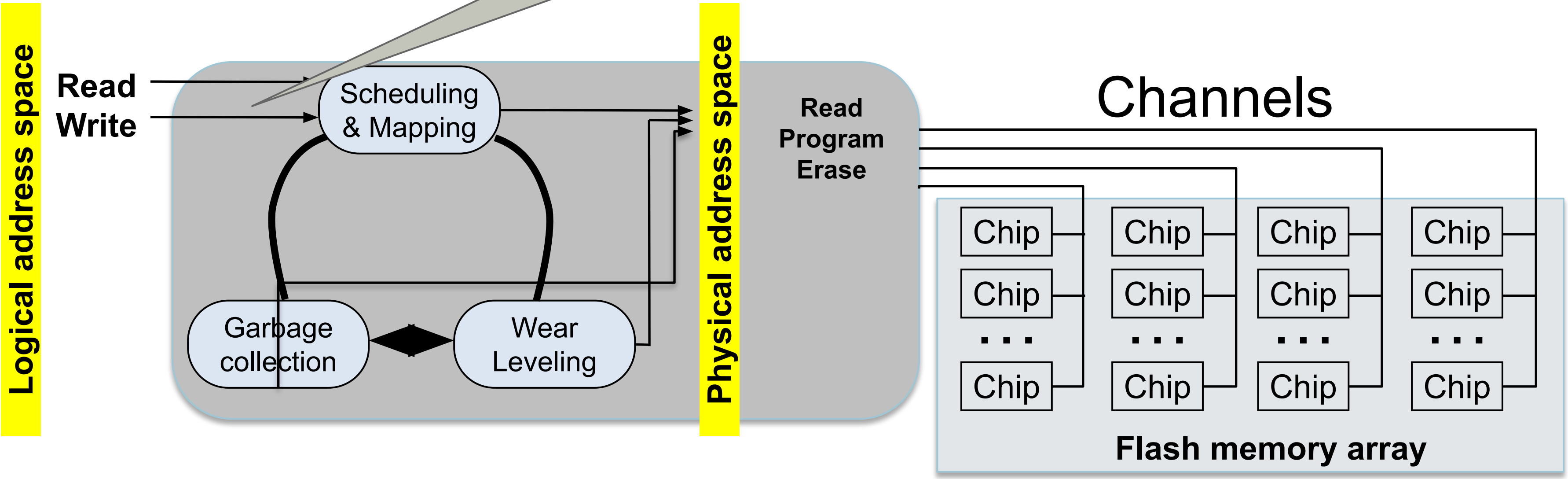
Reading a Disk Sector (3)



When the DMA transfer completes, the disk controller notifies the CPU with an *interrupt* (i.e., asserts a special “interrupt” pin on the CPU)

we’ll learn about interrupts in future lecture

Solid-State Drives

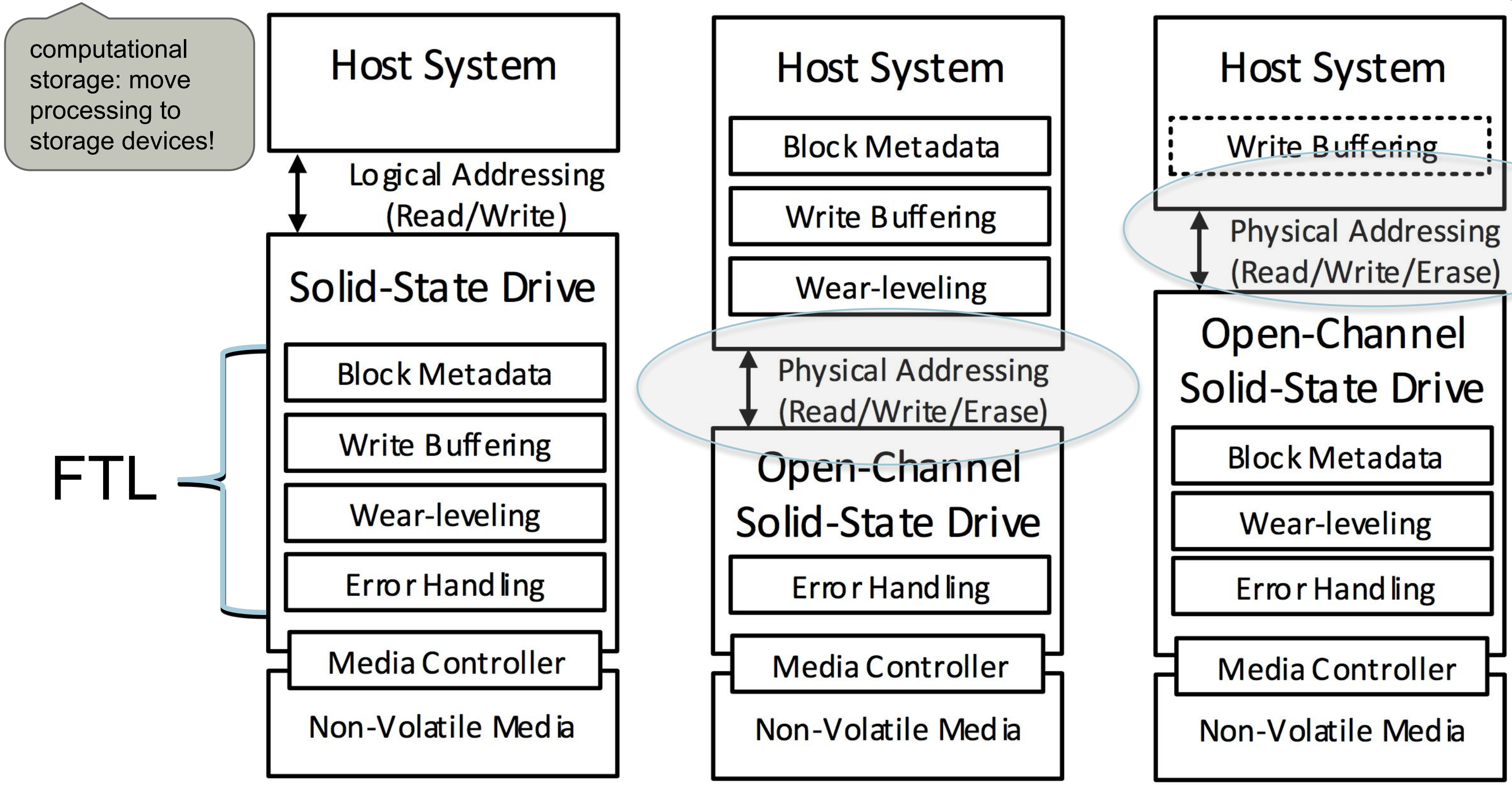


unchanged for a long time. let us replace mechanical disks w/ SSDs.

but SSDs are very different (have a controller).

Open-Channel SSDs: Design Space

computational storage: move processing to storage devices!



Mathias Bjørling, PhD at ITU, upstreamed Linux driver. Now used by Google, Amazon, Intel, Alibaba, Microsoft, …

Idea: separate front-end and back-end SSD management.

switching to… NVMe (controlled by Intel)

LightNVM separates (*application-customisable*) **front-end SSD management** from (*media-specific*) **back-end SSD management.**

Programming the Storage Controller

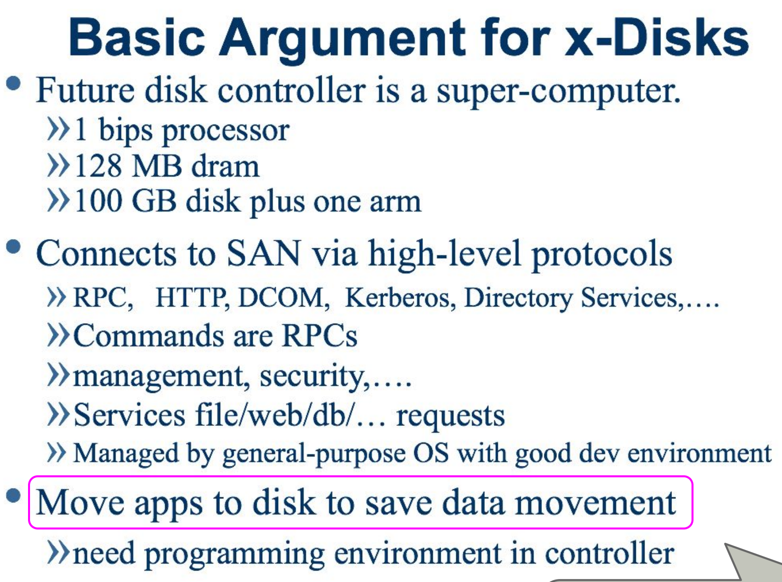
Put Everything in Future (Disk) Controllers (it's not "if", it's "when?")

Jim Gray

http://www.research.Microsoft.com/~Gray

Acknowledgements:
Dave Patterson explained this to me a year ago
Kim Keeton
Erik Riedel
Catharine Van Ingen
Helped me sharpen these arguments

1



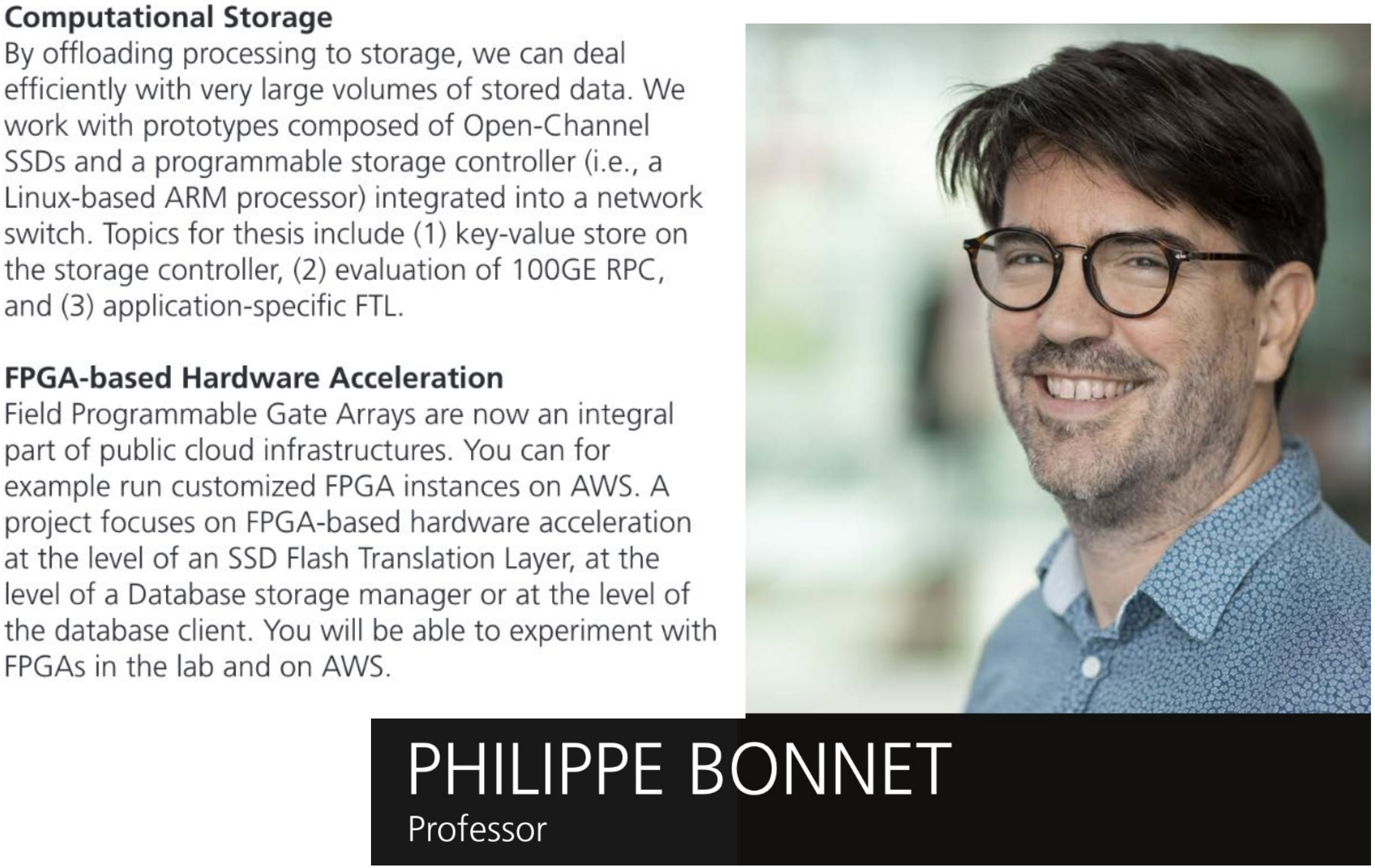
Basic Argument for x-Disks

- Future disk controller is a super-computer.
  - »1 bips processor
  - »128 MB dram
  - »100 GB disk plus one arm
- Connects to SAN via high-level protocols
  - » RPC, HTTP, DCOM, Kerberos, Directory Services,….
  - »Commands are RPCs
  - »management, security,….
  - »Services file/web/db/… requests
  - » Managed by general-purpose OS with good dev environment
- Move apps to disk to save data movement
  - »need programming environment in controller

Niclas works (worked?) on this!

Jim Gray, NASD Talk, 6/8/98
http://jimgray.azurewebsites.net/jimgraytalks.htm

Data Systems Group Promotion



Computational Storage

By offloading processing to storage, we can deal efficiently with very large volumes of stored data. We work with prototypes composed of Open-Channel SSDs and a programmable storage controller (i.e., a Linux-based ARM processor) integrated into a network switch. Topics for thesis include (1) key-value store on the storage controller, (2) evaluation of 100GE RPC, and (3) application-specific FTL.

FPGA-based Hardware Acceleration

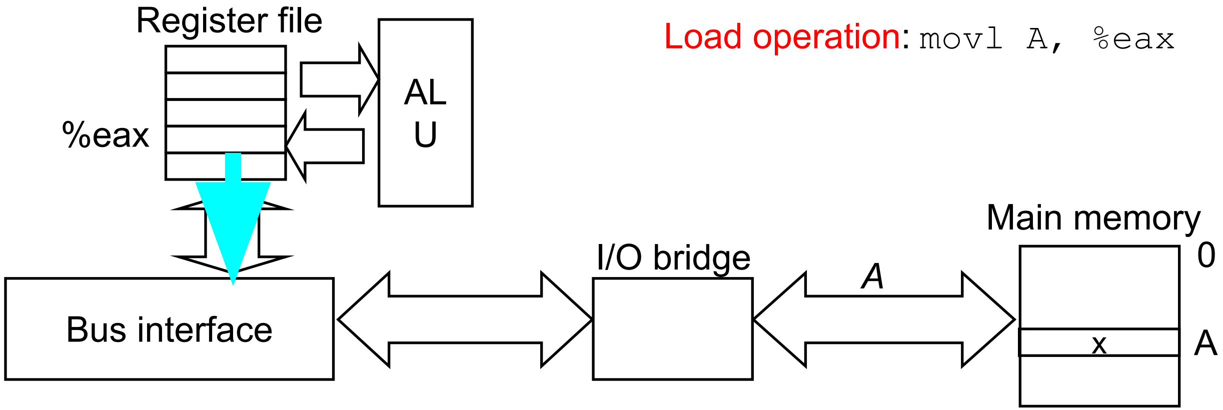
Field Programmable Gate Arrays are now an integral part of public cloud infrastructures. You can for example run customized FPGA instances on AWS. A project focuses on FPGA-based hardware acceleration at the level of an SSD Flash Translation Layer, at the level of a Database storage manager or at the level of the database client. You will be able to experiment with FPGAs in the lab and on AWS.

PHILIPPE BONNET

Professor

Memory Read Transaction (1)

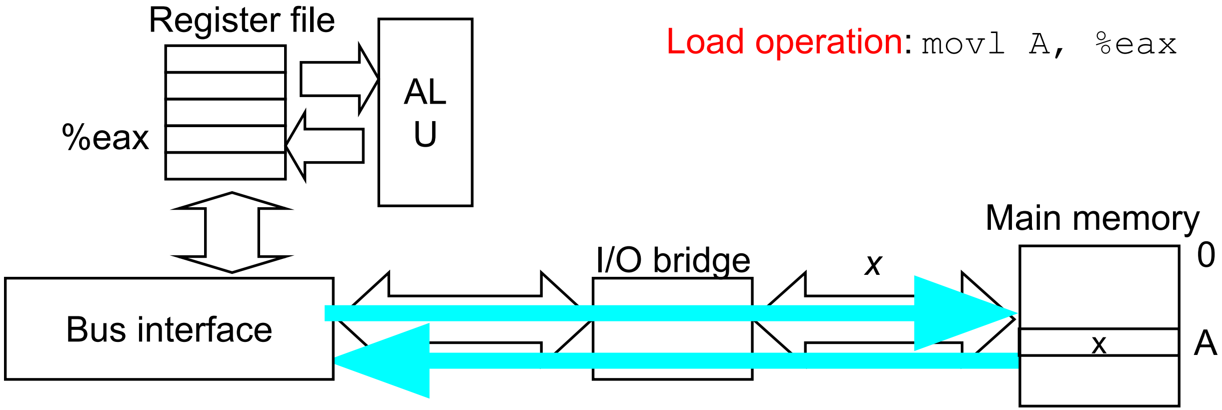
CPU places address A on the memory bus.



Load operation: `movl A, %eax`

Memory Read Transaction (2)

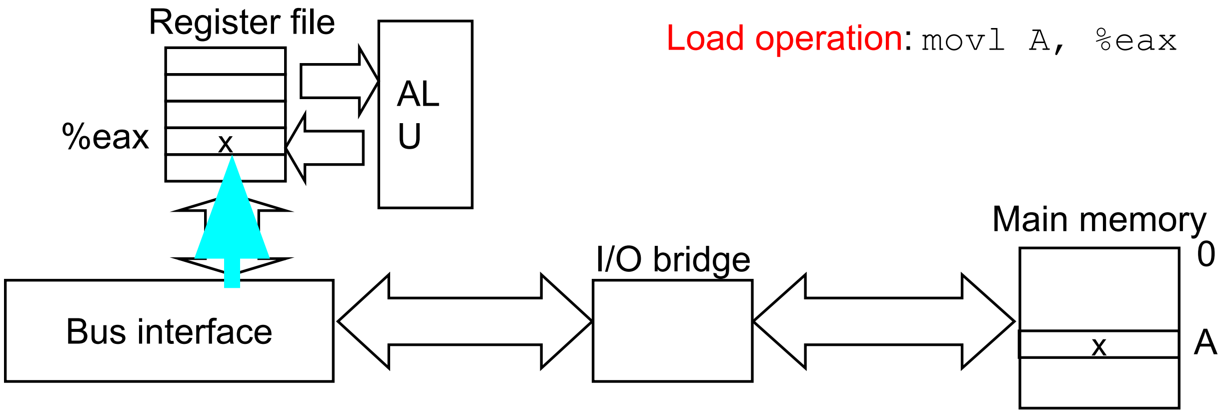
Main memory reads A from the memory bus, retrieves word x, and places it on the bus.



Load operation: `movl A, %eax`

Memory Read Transaction (3)

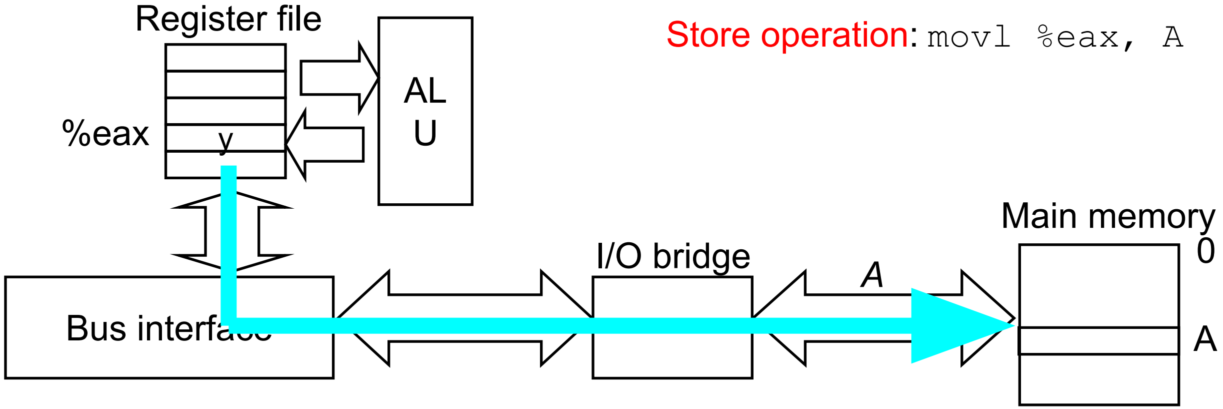
CPU read word x from the bus and copies it into register %eax.



Load operation: `movl A, %eax`

Memory Write Transaction (1)

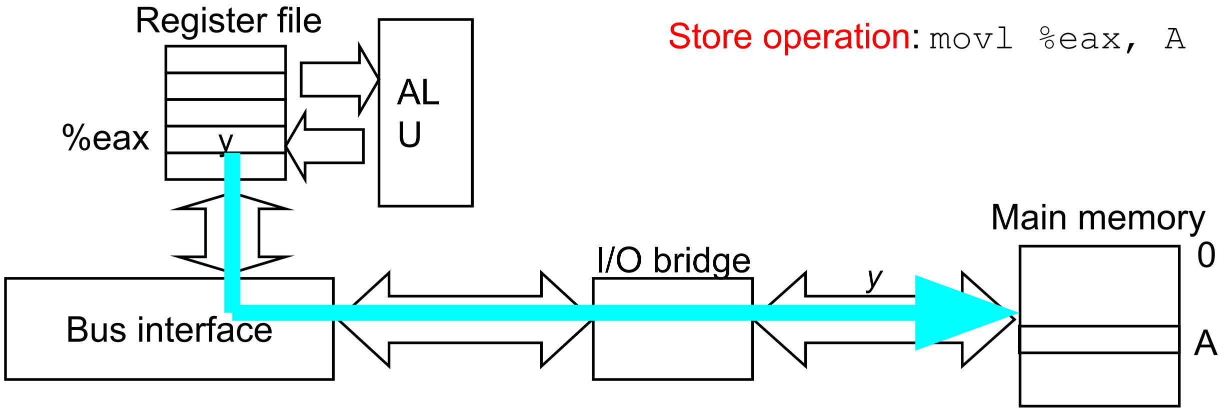
CPU places address A on bus. Main memory reads it and waits for the corresponding data word to arrive.



Store operation: `movl %eax, A`

Memory Write Transaction (2)

CPU places data word y on the bus. Main memory reads data word y from bus and stores it at address A.



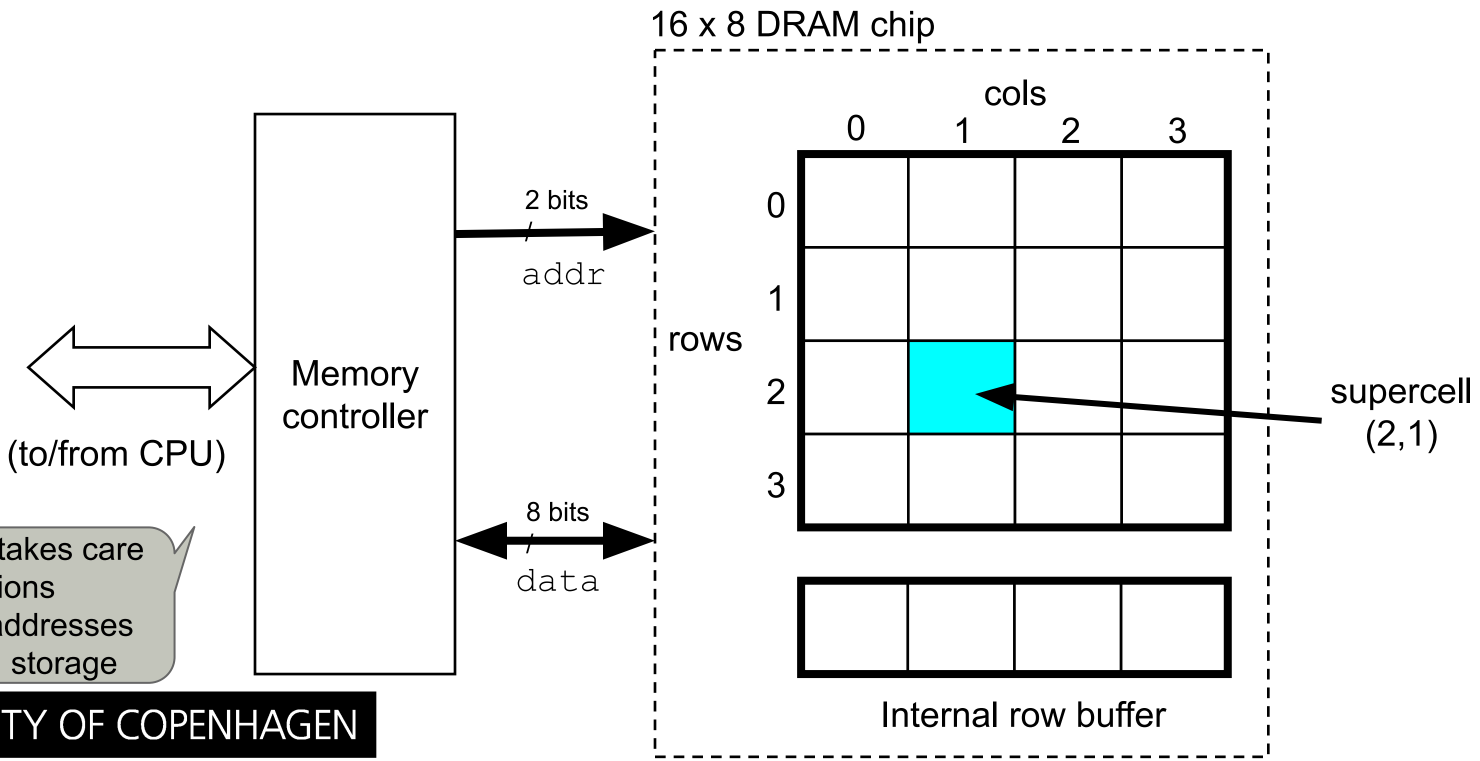
Store operation: `movl %eax, A`

Conventional DRAM Organization

the way RAM is organized is as an array of “supercells”

d * w DRAM:

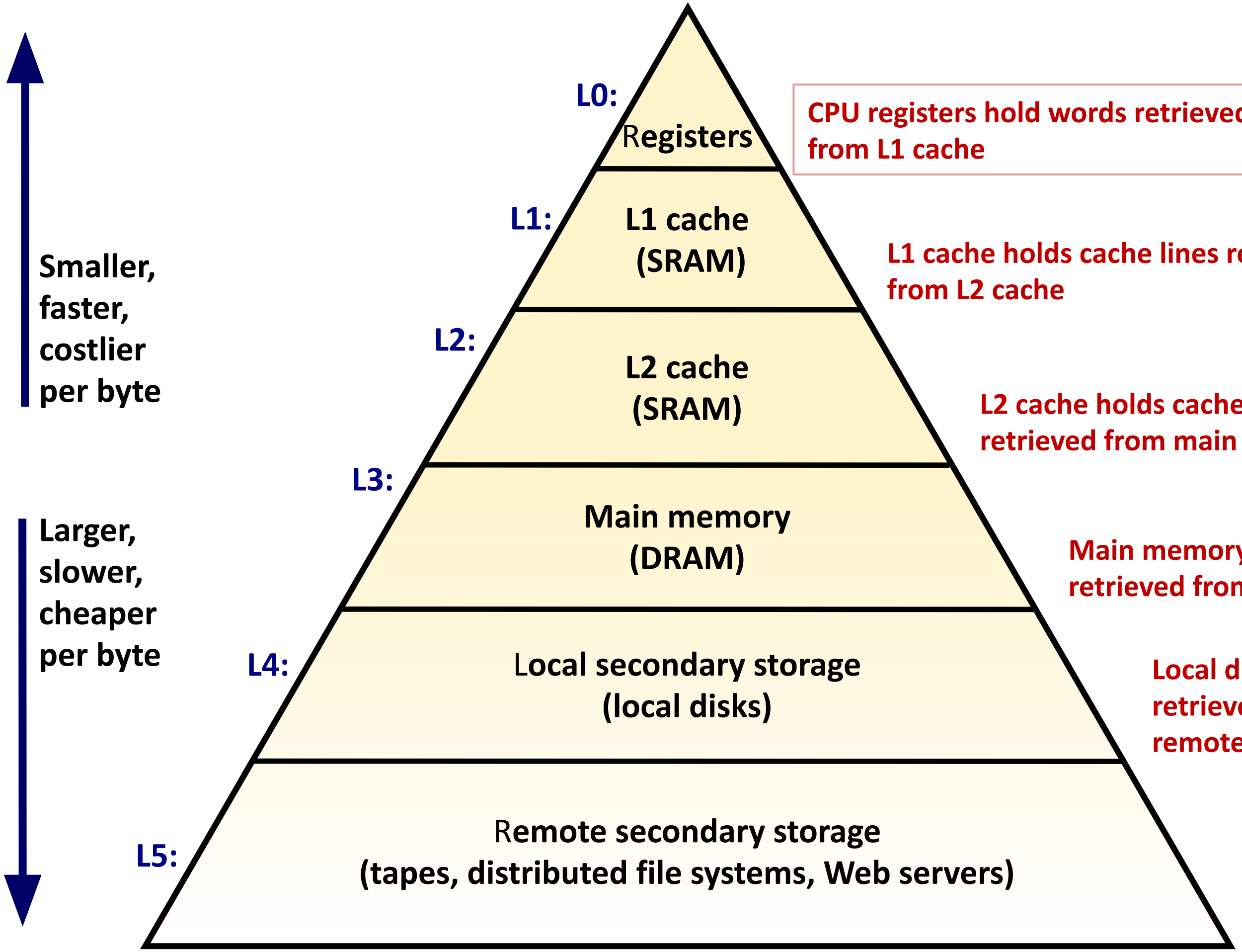
dw total bits organized as d supercells of size w bits



controller takes care of translations between addresses & physical storage

An Example Memory Hierarchy

this is actually a myth; local network faster than some local disks.



Smaller, faster, costlier per byte
Larger, slower, cheaper per byte
L0:
Registers
L1:
L1 cache (SRAM)
L2:
L2 cache (SRAM)
L3:
Main memory (DRAM)
L4:
Local secondary storage (local disks)
L5:
Remote secondary storage (tapes, distributed file systems, Web servers)

CPU registers hold words retrieved from L1 cache

L1 cache holds cache lines retrieved from L2 cache

L2 cache holds cache lines retrieved from main memory

Main memory holds disk blocks retrieved from local disks

Local disks hold files retrieved from disks on remote network servers

Caches

car mechanic analogy

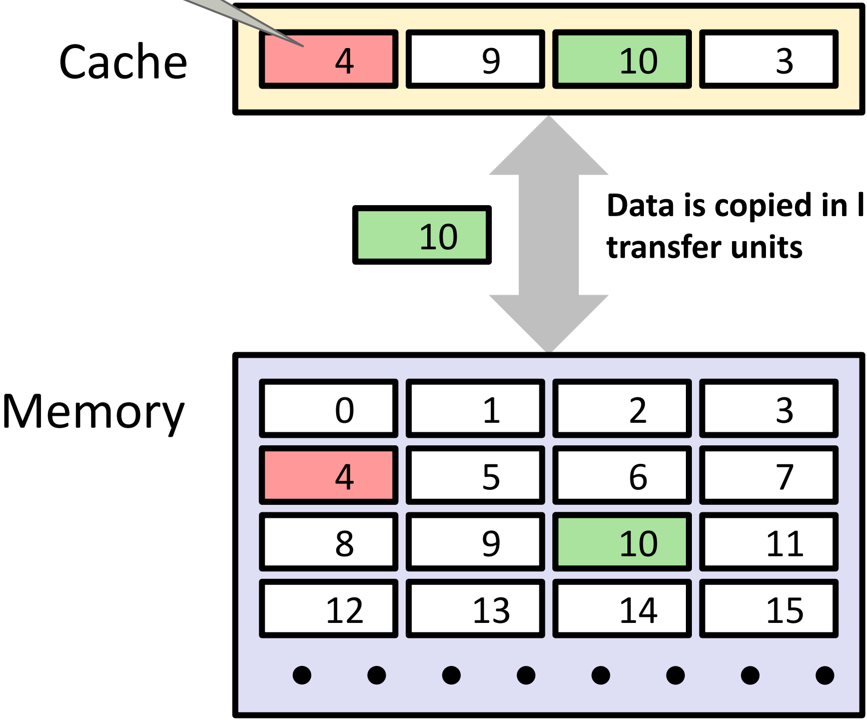
- *Cache:* A smaller, faster storage device that acts as a staging area for a subset of the data in a larger, slower device.
- Fundamental idea of a memory hierarchy:
  - For each k, the faster, smaller device at level k serves as a cache for the larger, slower device at level k+1.
- Why do memory hierarchies work? Because **locality**.
  - Programs tend to access the data at level k more often than they access the data at level k+1.
  - Thus, the storage at level k+1 can be slower, and thus larger and cheaper per bit.
- *Big Idea:* Memory hierarchy creates a large pool of storage that costs as much as the cheap storage near the bottom, but serves data to programs at the rate of the fast storage near the top.

General Cache Concepts

unit of transfer: 1 line

memory is split into blocks, aka. lines, of size 64 bytes.

number represents requested memory location

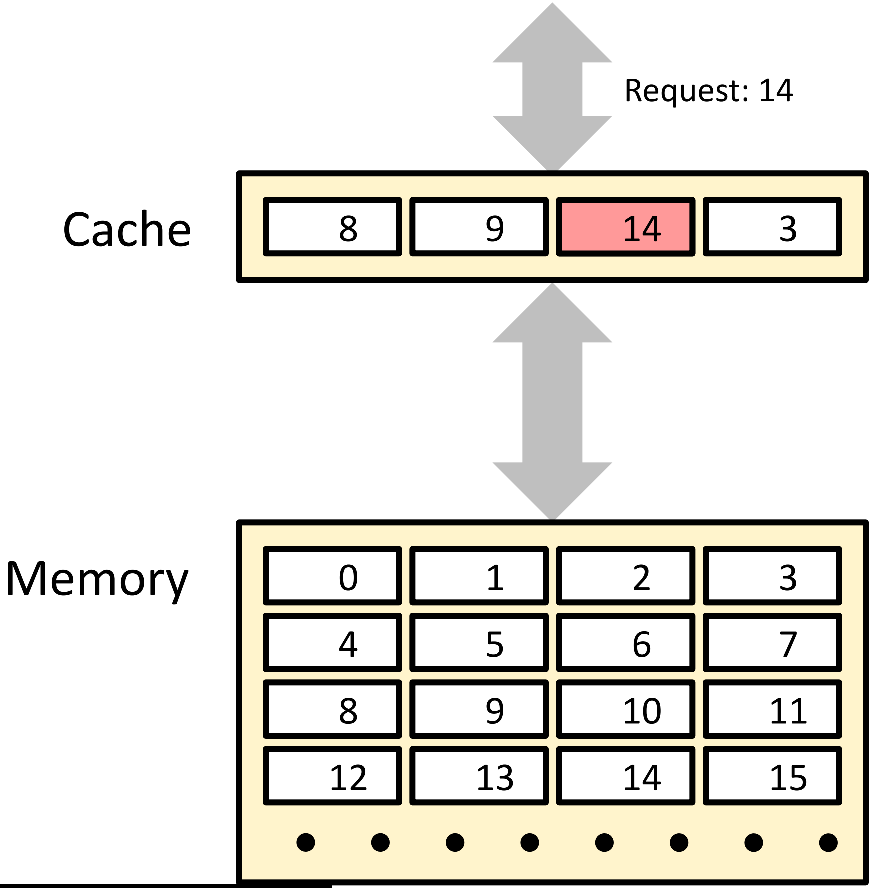


Smaller, faster, more expensive memory caches a subset of the lines

Data is copied in line-sized transfer units

Larger, slower, cheaper memory viewed as partitioned into "lines"

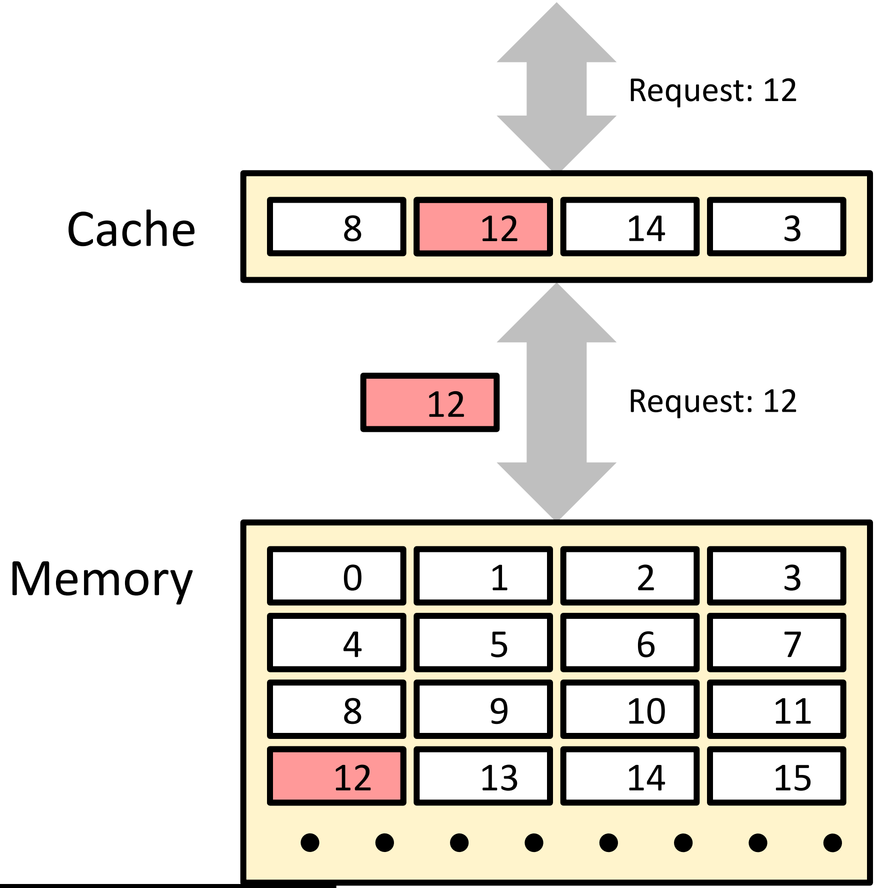
General Cache Concepts: Hit



Data in line b is needed

line b is in cache:
Hit!

General Cache Concepts: Miss



0	1	2	3
4	5	6	7
8	9	10	11
12	13	14	15
● ●	● ●	● ●	● ●

Data in line b is needed

Line b is not in cache:
Miss!

Line b is fetched from
memory

Line b is stored in cache

- Placement policy:
  determines where b goes
- Replacement policy:
  determines which block gets evicted (victim)

General Caching Concepts: Types of Cache Misses

Cold (compulsory) miss

Cold misses occur because the cache is empty.

Conflict miss

Most caches limit lines at level k+1 to a small subset (sometimes a singleton) of the line positions at level k.

- E.g. Line i at level k+1 must be placed in line (i mod 4) at level k.

Conflict misses occur when the level k cache is large enough, but multiple data objects all map to the same level k line.

- E.g. Referencing blocks 0, 8, 0, 8, 0, 8, ... would miss every time.

Capacity miss

Occurs when set of active cache lines (working set) is larger than the cache.

placement policy
avoids conflict misses
replacement policy care, to
avoid capacity misses.

thrashing = every
access misses.

Examples of Caching in the Hierarchy

ex: access to memory is 100x more expensive than L1 cache.

Cache Type	What is Cached?	Where is it Cached?	Latency (cycles)	Managed By
Registers	4-8 bytes words	CPU core	0	Compiler
TLB	Address translations	On-Chip TLB	0	Hardware
L1 cache	64-bytes line	On-Chip L1	1	Hardware
L2 cache	64-bytes line	On/Off-Chip L2	10	Hardware
Virtual Memory	4-KB page	Main memory	100	Hardware + OS
Buffer cache	Parts of files	Main memory	100	OS
Disk cache	Disk sectors	Disk controller	100,000	Disk firmware
Network buffer cache	Parts of files	Local disk	10,000,000	AFS/NFS client
Browser cache	Web pages	Local disk	10,000,000	Web browser
Web cache	Web pages	Remote server disks	1,000,000,000	Web proxy server

Cache Memories

given some data,
where is it going to be located?

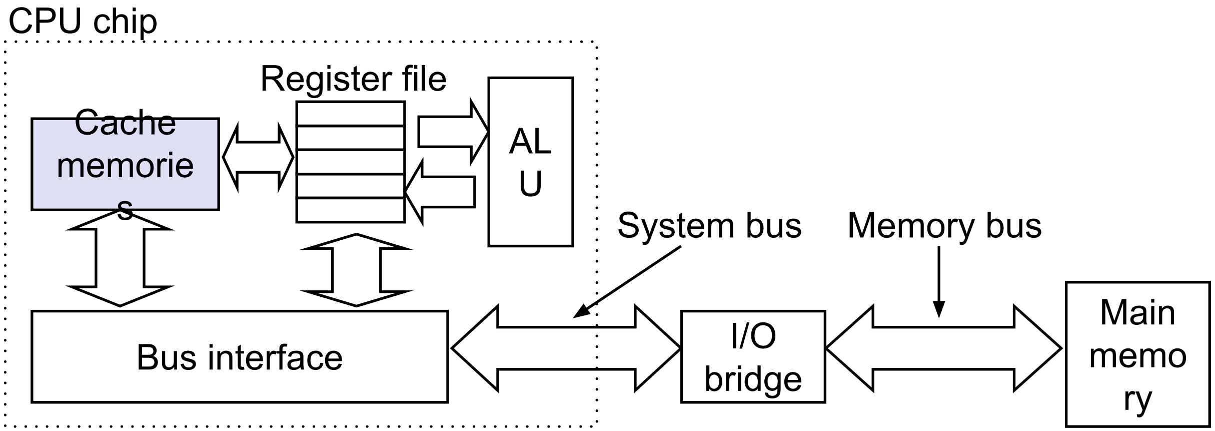
10s

Cache memories are small, fast SRAM-based memories managed automatically in hardware.

Hold frequently accessed blocks of main memory

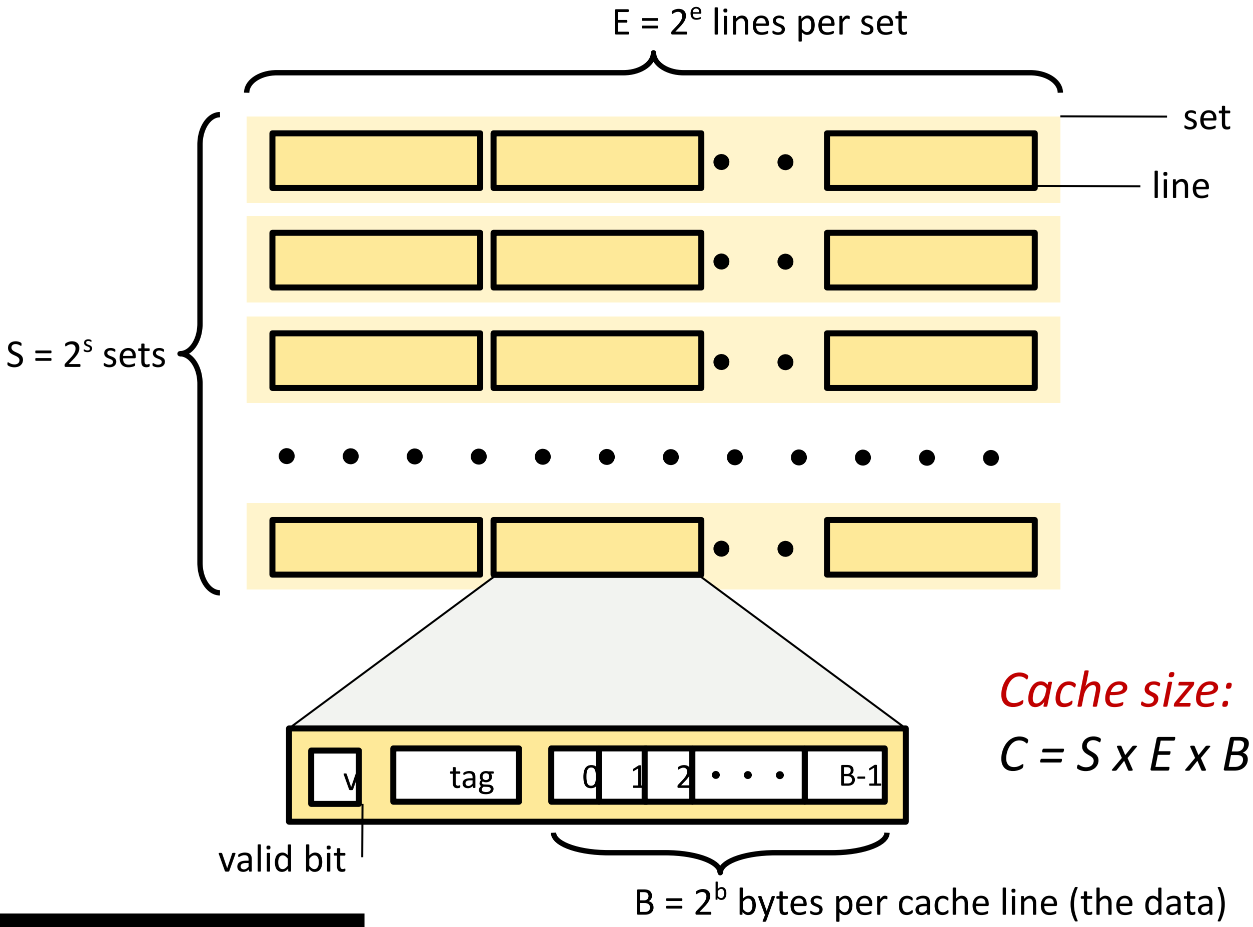
CPU looks first for data in caches (e.g., L1, L2, and L3), then in main memory.

Typical system structure:



General Cache Organization (S, E, B)

organization of a cache.
have S sets, and E lines.

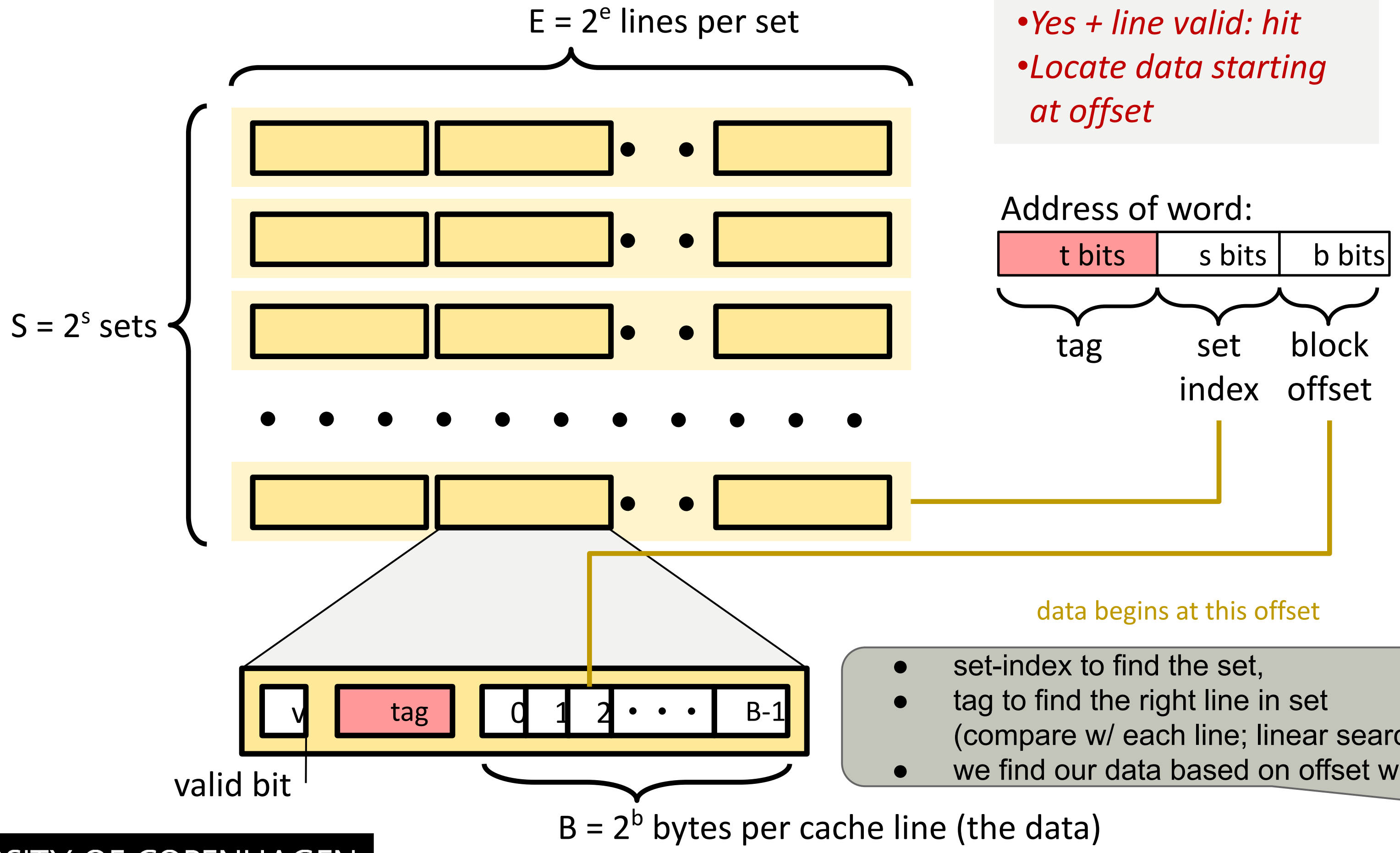


Cache size:

C = S x E x B data bytes

Cache Read

- *Locate set*
- *Check if any line in set has matching tag*
- *Yes + line valid: hit*
- *Locate data starting at offset*



- set-index to find the set,
- tag to find the right line in set (compare w/ each line; linear search)
- we find our data based on offset within line.

Direct-Mapped Cache Simulation

we ask for 1 byte.
but we transfer 1 line at a time.
here, a line is 2 bytes.

t=1	s=2	b=1
x	xx	x

M=16 byte addresses, B=2 bytes/block,
S=4 sets, E=1 Blocks/set

tag (0..1)

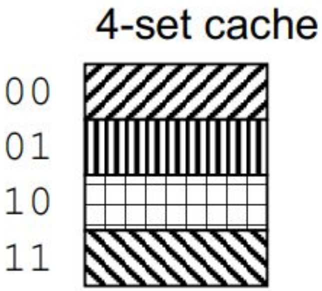
set (0..3)

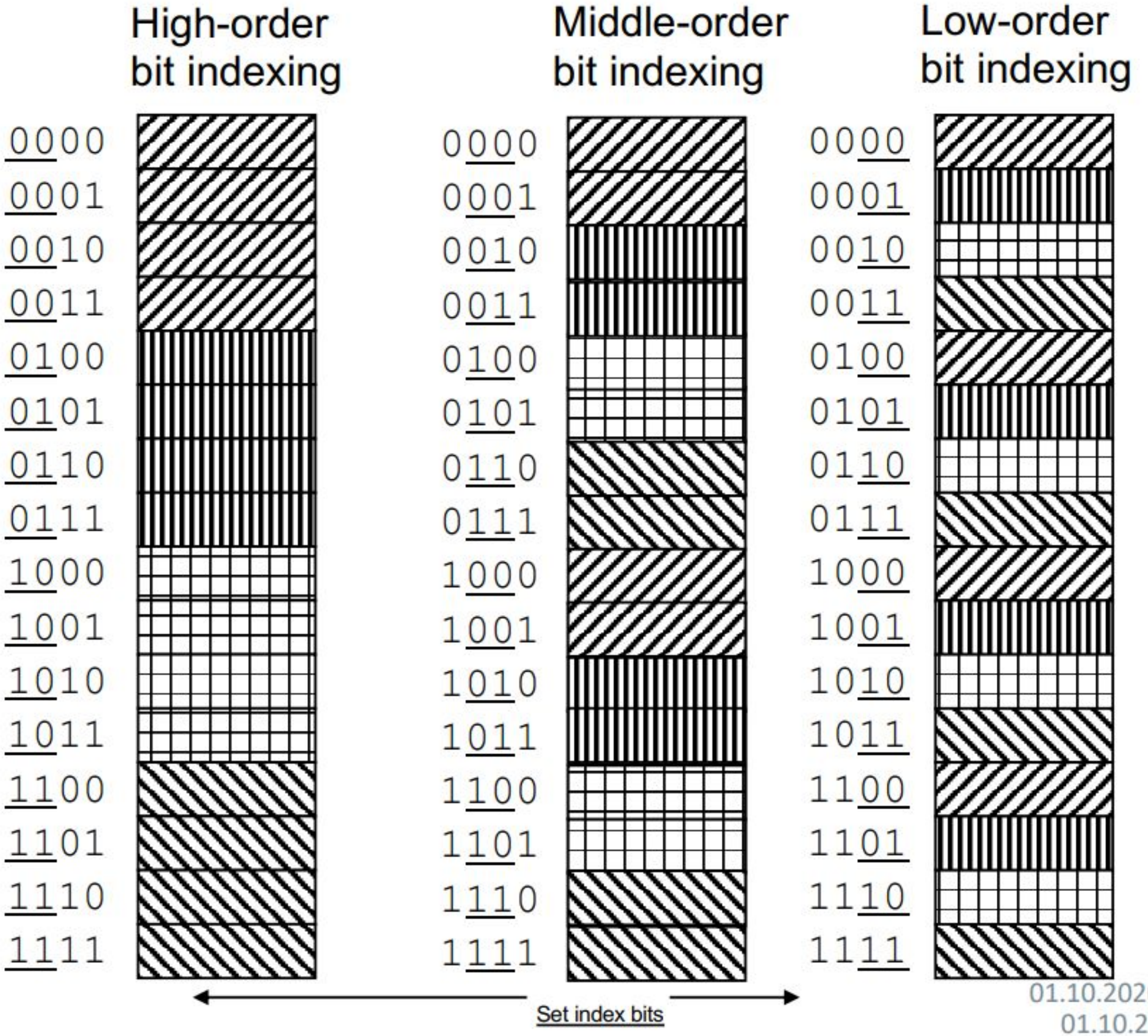
Address trace (reads, one byte per read):

0	$[0\underline{00}0_2]$,	miss
1	$[0\underline{00}1_2]$,	hit
7	$[0\underline{11}1_2]$,	miss
8	$[1\underline{00}0_2]$,	miss
0	$[0\underline{00}0_2]$	miss

	v	Tag	Block
Set 0	1	0	M[0-1]
Set 1			
Set 2			
Set 3	1	0	M[6-7]

Why index with the middle bits?

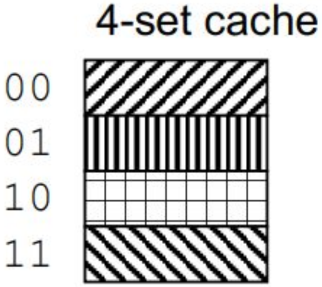


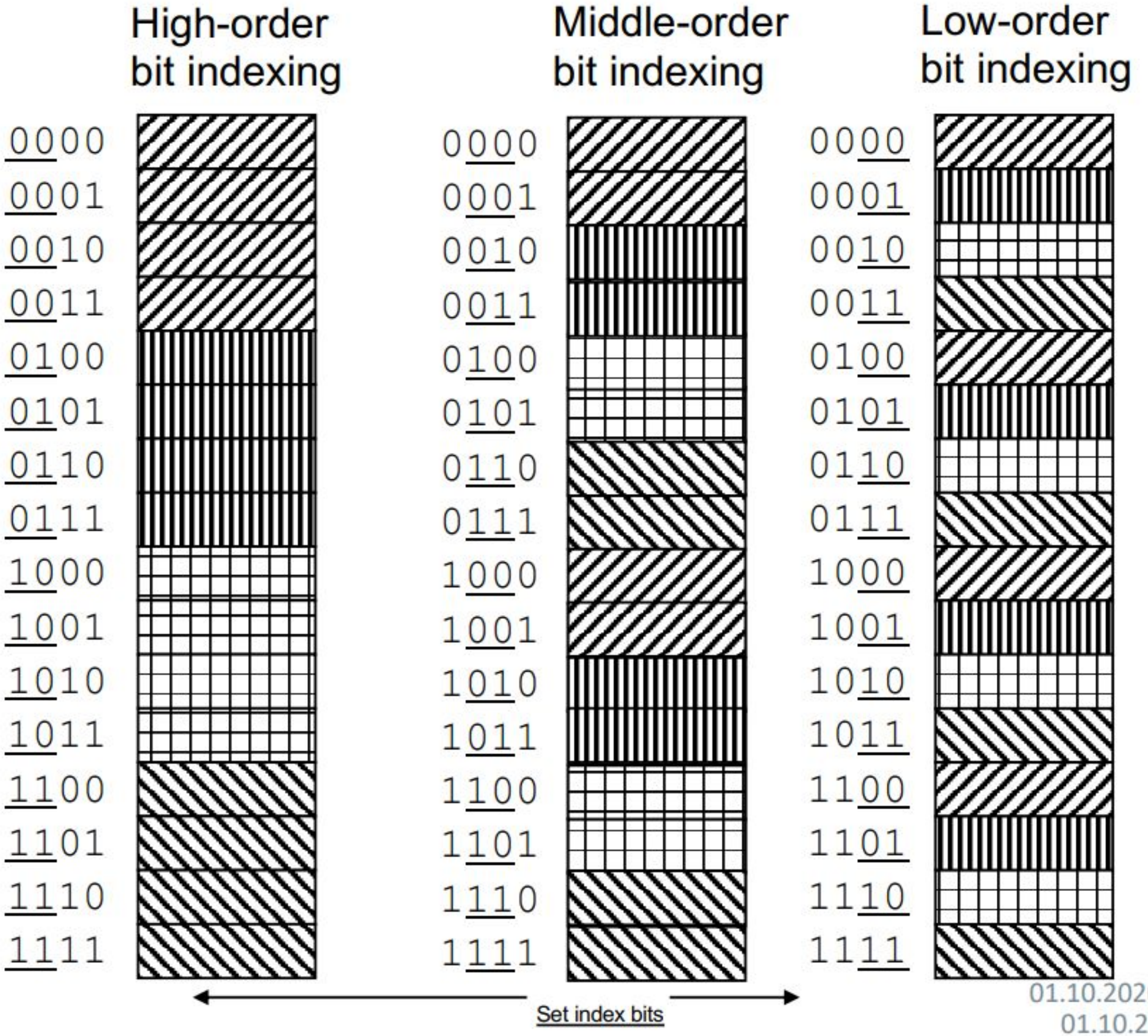


every cell on the right is 1 byte.
line size 2 bytes.

Q: which indexing strategy is best?

Why index with the middle bits?





every cell on the right is 1 byte.
line size 2 bytes.

Q: which indexing strategy is best?

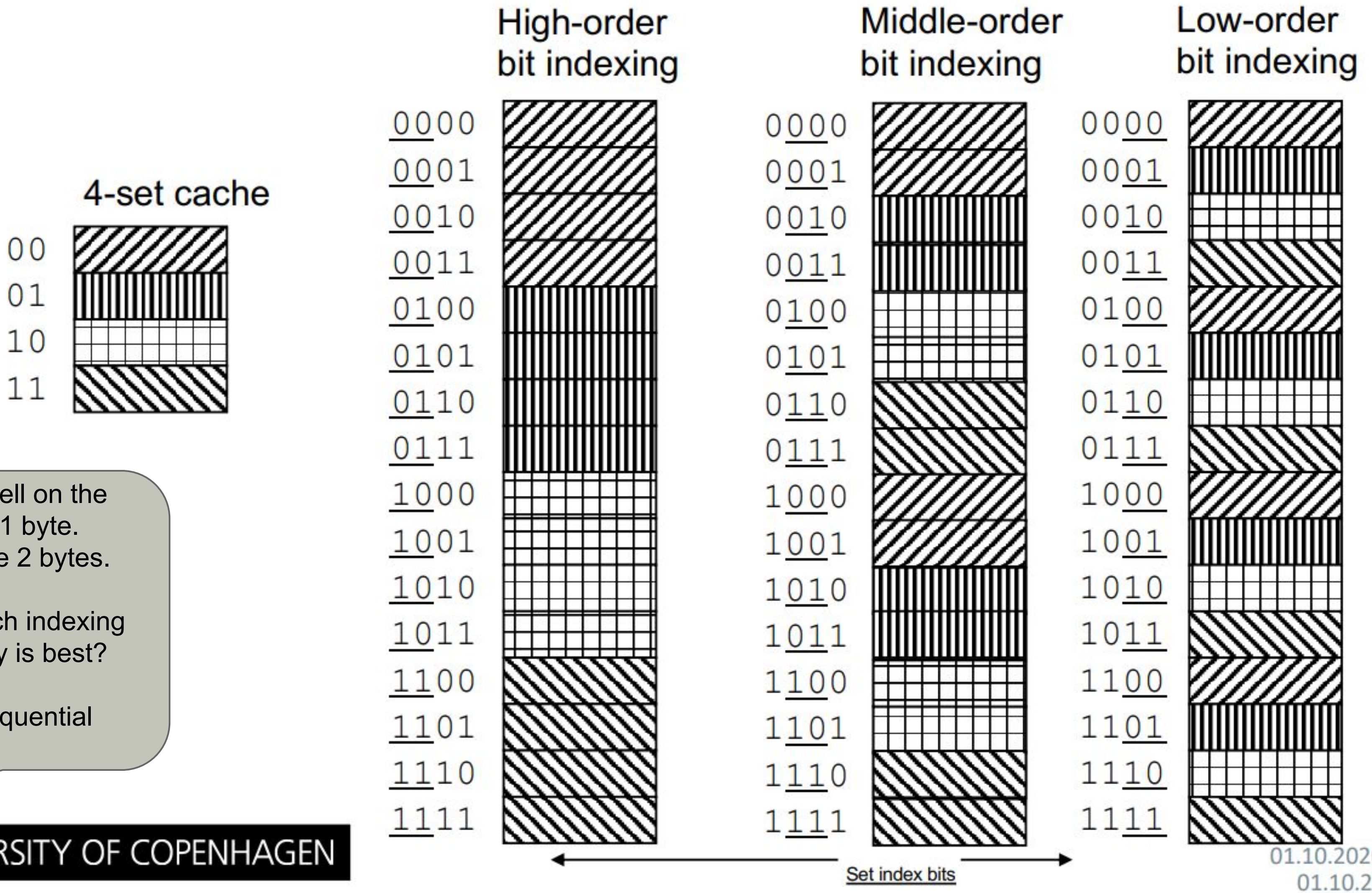
hint: sequential access

Why index with the middle bits?

50% miss

50% miss,
better temporal

underutilized
cache



every cell on the right is 1 byte.
line size 2 bytes.

Q: which indexing strategy is best?

hint: sequential access

Array Allocation

compiler is going to reserve space for the array.

Basic Principle

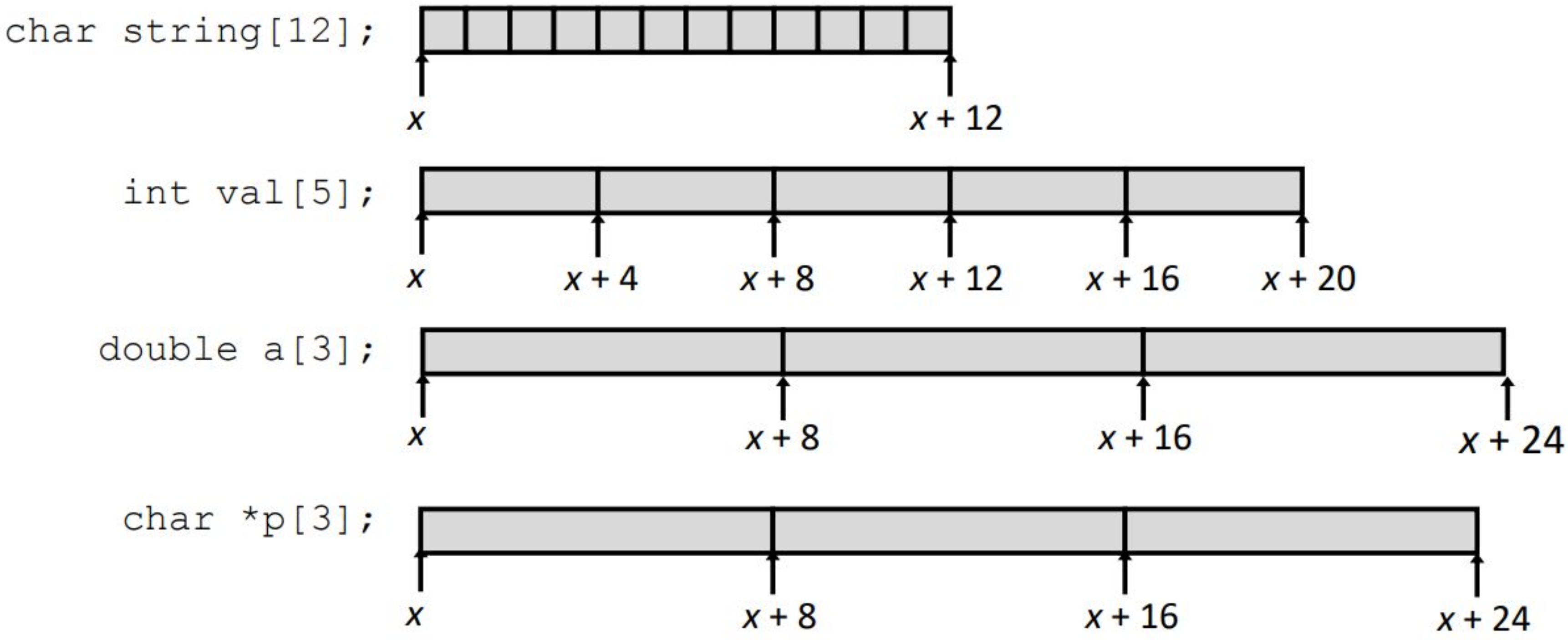
T **`A[`***L***`]`***;*

A is an Array of data type *T* and length *L*

Contiguously allocated region of *L* * **`sizeof`**（*T*） bytes

examples follow

Array Allocation



8 bytes = 64 bits (address size = word size)

Array Access

int **A[*5*] = {0, 1, 2, 3, 4};**

Array of data type *int* and length *5*

Identifier **A** can be used as a pointer to array element 0: Type *int**

val from previous slide

Reference	Type	Value
val[4]	**int**	**4**
val	**int ***	x
val+1	**int ***	$x + 4$
&val[2]	**int ***	$x + 8$
val[5]	**int**	??
***(val+1)**	**int**	**1**
val + *i*	**int ***	$x + 4\,i$

undefined behavior

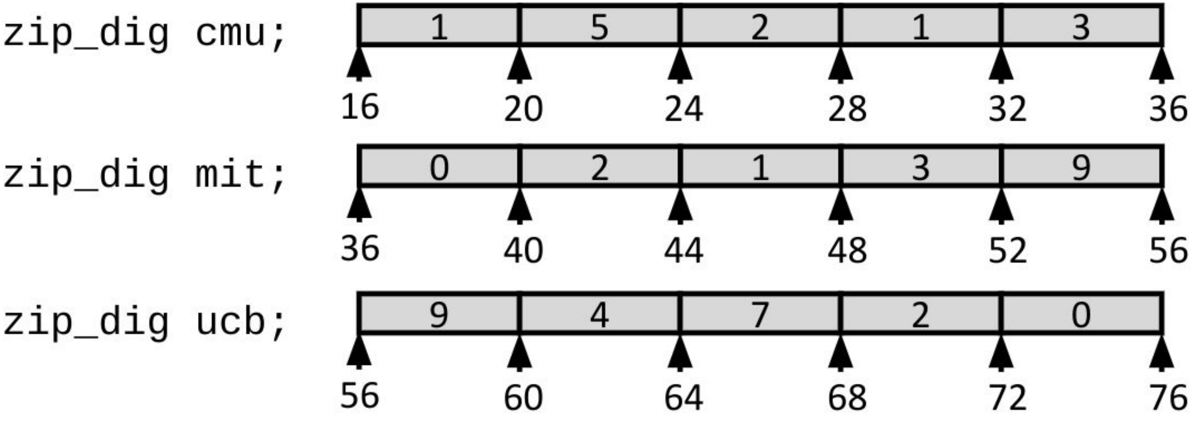
Array Example

type alias

```
#define ZLEN 5
typedef int zip_dig[ZLEN];

zip_dig cmu = { 1, 5, 2, 1, 3 };
zip_dig mit = { 0, 2, 1, 3, 9 };
zip_dig ucb = { 9, 4, 7, 2, 0 };
```

(ucb = berkeley)



20 = 5*4

Declaration “`zip_dig cmu`” equivalent to “`int cmu[5]`”

Example arrays were allocated in successive 20 byte blocks

Not guaranteed to happen in general

Array Example

```
void zincr(zip_dig z) {
  int i;
  for (i = 0; i < ZLEN; i++)
    z[i]++;
}
```

addressing mechanism:
z + 4*i

```
  # rdx = z
    movq     $0, %rax                #   %rax = i
.L4:                                 # loop:
    addq     $1, (%rdx,%rax,4)       #   z[i]++
    addq     $1, %rax                #   i++
    cmpl     $5, %rax                #   i:ZLEN (ZLEN==5)
    jne .L4                          #   if !=, goto loop
```

$1 is
constant 1

%r is
register r

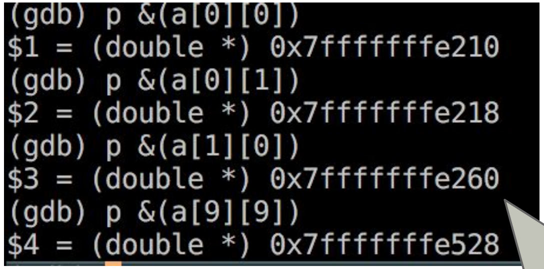
A Higher Level Example

```c
int sum_array_rows(double a[10][10])
{
    int i, j;
    double sum = 0;

    for (i = 0; i < 16; i++)
        for (j = 0; j < 16; j++)
            sum += a[i][j];
    return sum;
}
```

```c
int sum_array_cols(double a[10][10])
{
    int i, j;
    double sum = 0;

    for (j = 0; i < 16; i++)
        for (i = 0; j < 16; j++)
            sum += a[i][j];
    return sum;
}
```



```
(gdb) p &(a[0][0])
$1 = (double *) 0x7fffffffe210
(gdb) p &(a[0][1])
$2 = (double *) 0x7fffffffe218
(gdb) p &(a[1][0])
$3 = (double *) 0x7fffffffe260
(gdb) p &(a[9][9])
$4 = (double *) 0x7fffffffe528
```

not in the same line as previous two

in perflab, use debugger to find out how my data structure is layed out.

What about writes?

Multiple copies of data exist:

L1, L2, Main Memory, Disk

What to do on a write-hit?

Write-through (write immediately to memory)

faster

Write-back (defer write to memory until replacement of line)

- Need a dirty bit (line different from memory or not)

What to do on a write-miss?

Write-allocate (load into cache, update line in cache)

- Good if more writes to the location follow

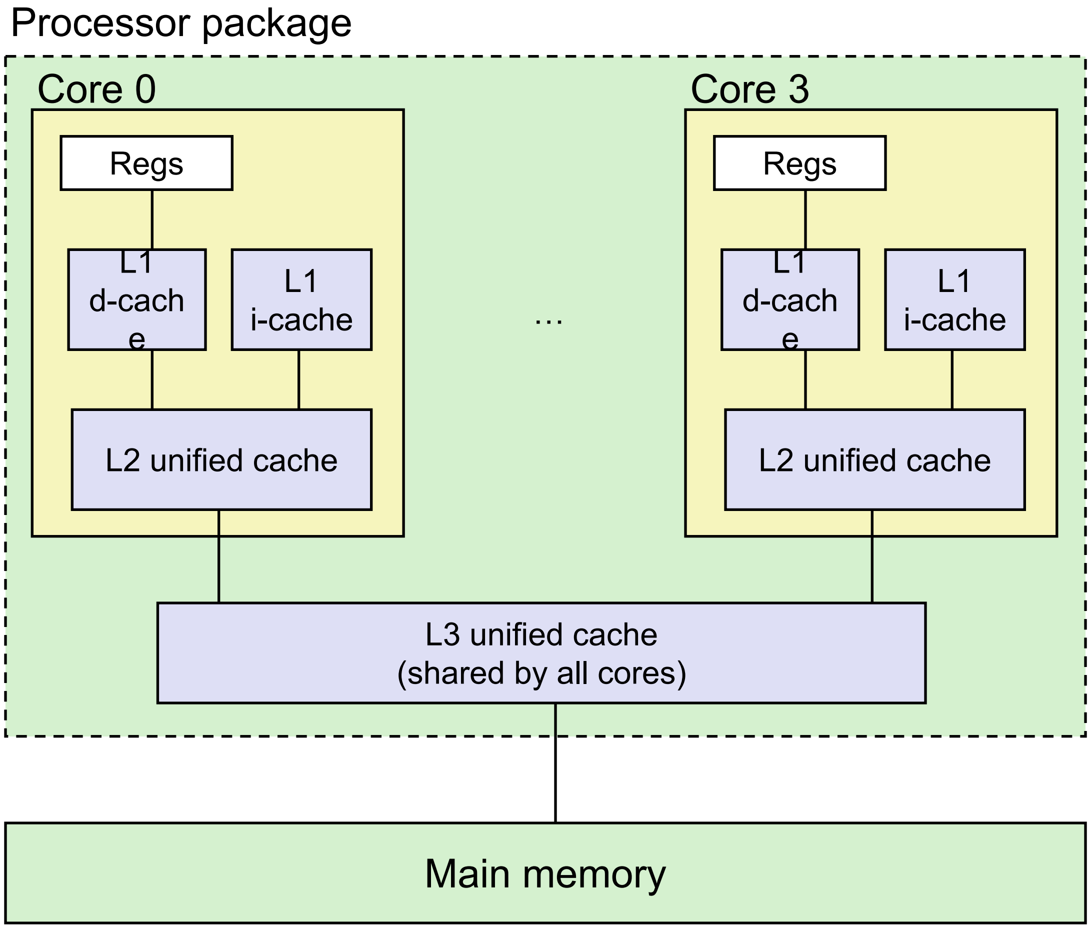
No-write-allocate (writes immediately to memory)

Typical

Write-through + No-write-allocate

Write-back + Write-allocate

Intel Core i7 Cache Hierarchy



L1 i-cache and d-cache:
32 KB, 8-way,
Access: 4 cycles

L2 unified cache:
256 KB, 8-way,
Access: 11 cycles

L3 unified cache:
8 MB, 16-way,
Access: 30-40 cycles

Line size: 64 bytes for all caches.

Cache Performance Metrics

10s

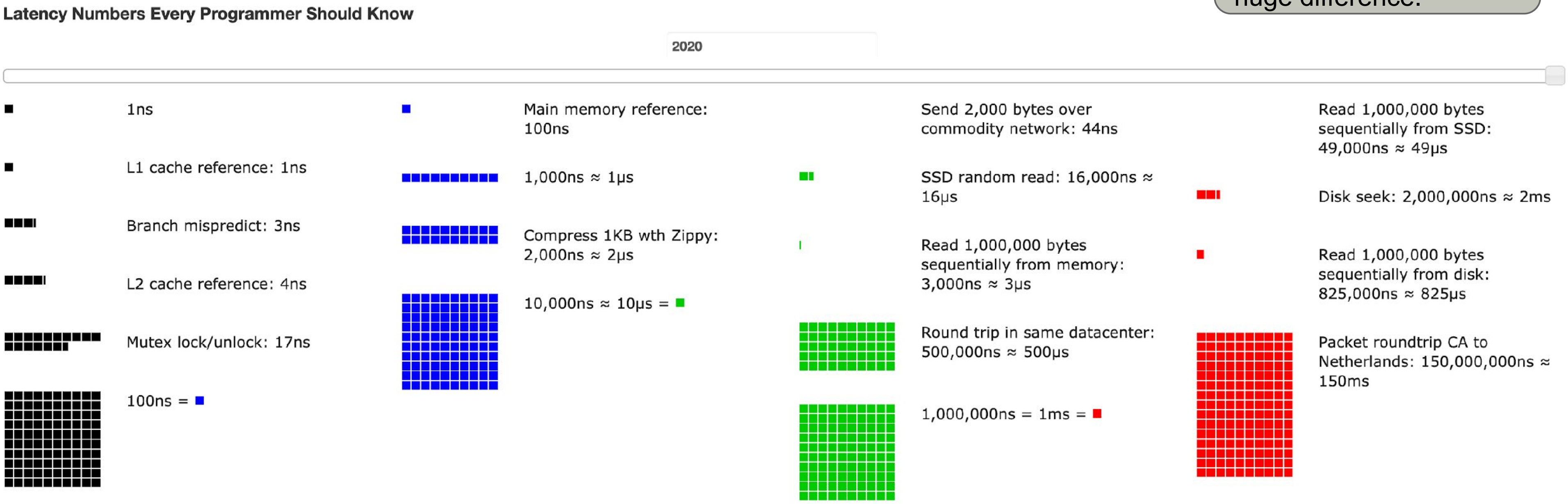
- **Miss Rate**
  - Fraction of memory references not found in cache (misses / accesses) = **1 – hit rate**
  - Typical numbers (in percentages):
  - – 3-10% for L1
  - – can be quite small (e.g., < 1%) for L2, depending on size, etc.

- **Hit Time**
  - Time to deliver a line in the cache to the processor
  - – includes time to determine whether the line is in the cache
  - Typical numbers:
  - – 1-2 clock cycle for L1
  - – 5-20 clock cycles for L2

- **Miss Penalty**
  - Additional time required because of a miss
  - – typically 50-200 cycles for main memory (Trend: increasing!)

Latency Numbers Every Programmer Should Know

we’ve seen this.
penalty: from 1 to 100.
huge difference.



Latency Numbers Every Programmer Should Know

2020

1ns

L1 cache reference: 1ns

Branch mispredict: 3ns

L2 cache reference: 4ns

Mutex lock/unlock: 17ns

100ns = 🟦

Main memory reference: 100ns

1,000ns ≈ 1μs

Compress 1KB wth Zippy: 2,000ns ≈ 2μs

10,000ns ≈ 10μs = 🟩

Send 2,000 bytes over commodity network: 44ns

SSD random read: 16,000ns ≈ 16μs

Read 1,000,000 bytes sequentially from memory: 3,000ns ≈ 3μs

Round trip in same datacenter: 500,000ns ≈ 500μs

1,000,000ns = 1ms = 🟥

Read 1,000,000 bytes sequentially from SSD: 49,000ns ≈ 49μs

Disk seek: 2,000,000ns ≈ 2ms

Read 1,000,000 bytes sequentially from disk: 825,000ns ≈ 825μs

Packet roundtrip CA to Netherlands: 150,000,000ns ≈ 150ms

https://colin-scott.github.io/personal_website/research/interactive_latency.html

Lets think about those numbers

Huge difference between a hit and a miss

Could be 100x, if just L1 and main memory

Would you believe 99% hits is twice as good as 97%?

Consider:

cache hit time of 1 cycle

miss penalty of 100 cycles

Average access time:

97% hits: 1 cycle + 0.03 * 100 cycles = **4 cycles**

99% hits: 1 cycle + 0.01 * 100 cycles = **2 cycles**

This is why “miss rate” is used instead of “hit rate”

Arrays and Cache Metrics

skip

C arrays allocated in row-major order

- each row in contiguous memory locations

Stepping through columns in one row:

-
  ```
  for (i = 0; i < N; i++)
   sum += a[0][i];
  ```
- accesses successive elements
- if block size (B) > 4 bytes, exploit spatial locality

compulsory miss rate = 4 bytes / B

Stepping through rows in one column:

-
  ```
  for (i = 0; i < n; i++)
   sum += a[i][0];
  ```
- accesses distant elements
- no spatial locality!

compulsory miss rate = 1 (i.e. 100%)

Writing Cache Friendly Code

Make the common case go fast

Focus on the inner loops of the core functions

Minimize the misses in the inner loops

Repeated references to variables are good (temporal locality)

Stride-1 reference patterns are good (spatial locality)

Key idea: Our qualitative notion of locality is quantified through our understanding of cache memories.

Locality Example #1

10s

Question: Can you permute the loops so that the function scans the 3-d array `a` with a stride-1 reference pattern (and thus has good spatial locality)?

```
int sum_array_3d(int a[M][N][N])
{
    int i, j, k, sum = 0;

    for (i = 0; i < M; i++)
        for (j = 0; j < N; j++)
            for (k = 0; k < N; k++)
                sum += a[i][k][j];
    return sum;
}
```

Locality Example #2

Assume:

Line size = 32B (big enough for four 64-bit words)

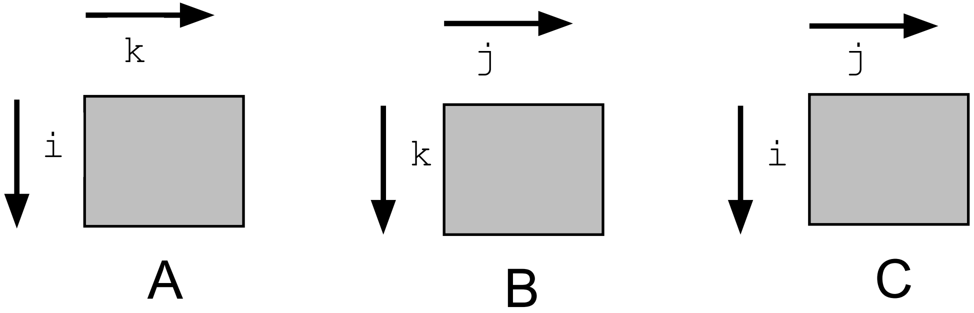
Matrix dimension (N) is very large

- Approximate 1/N as 0.0

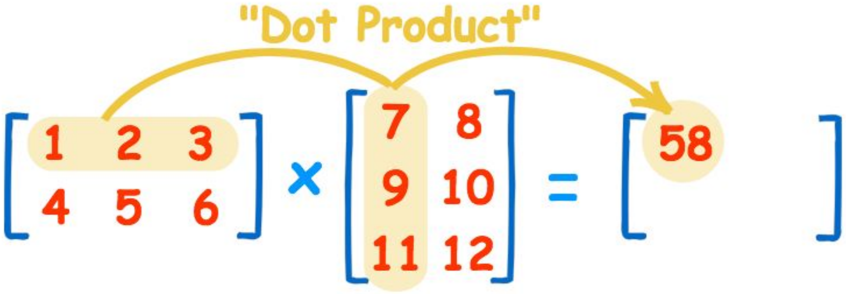
Cache is not even big enough to hold multiple rows

Analysis Method:

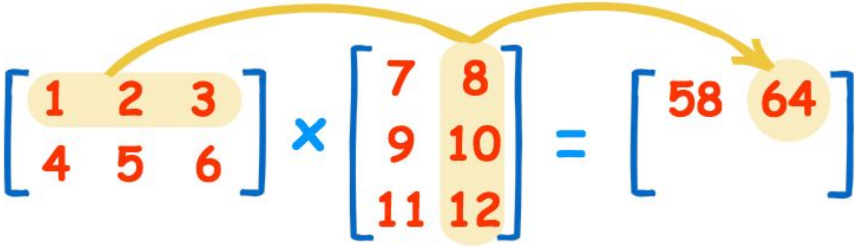
Look at access pattern of inner loop



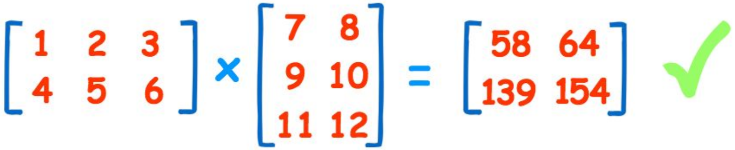
Matrix Multiplication



$$\begin{bmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \end{bmatrix} \times \begin{bmatrix} 7 & 8 \\ 9 & 10 \\ 11 & 12 \end{bmatrix} = \begin{bmatrix} 58 & \\ & \end{bmatrix}$$



$$\begin{bmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \end{bmatrix} \times \begin{bmatrix} 7 & 8 \\ 9 & 10 \\ 11 & 12 \end{bmatrix} = \begin{bmatrix} 58 & 64 \\ & \end{bmatrix}$$



$$\begin{bmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \end{bmatrix} \times \begin{bmatrix} 7 & 8 \\ 9 & 10 \\ 11 & 12 \end{bmatrix} = \begin{bmatrix} 58 & 64 \\ 139 & 154 \end{bmatrix} ✓$$

DONE!

Matrix Multiplication Example

Description:

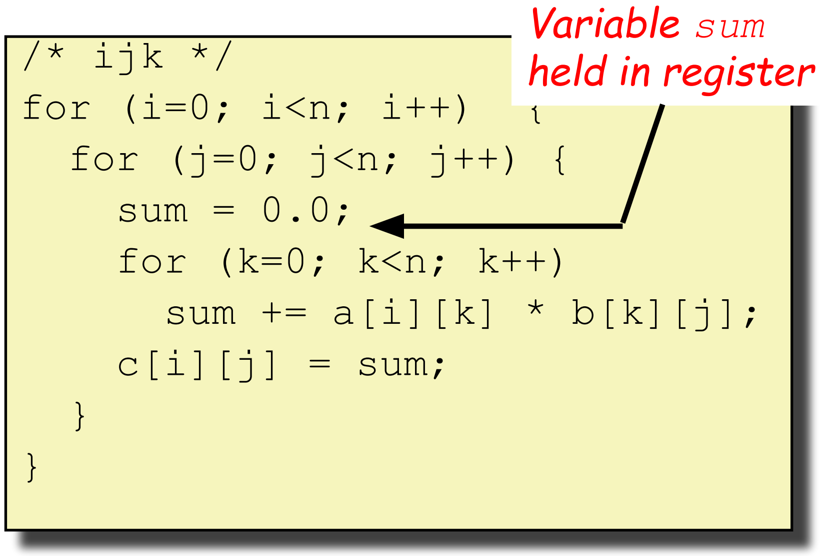
Multiply N x N matrices

O(N^3) total operations

N reads per source element

N values summed per destination

- but may be able to hold in register



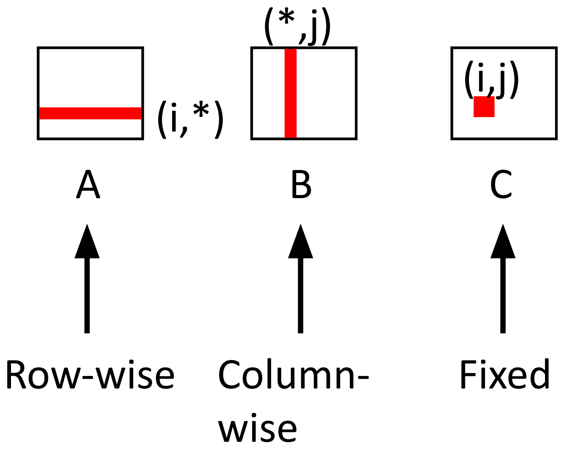
```
/* ijk */
for (i=0; i<n; i++)  {
  for (j=0; j<n; j++) {
    sum = 0.0;
    for (k=0; k<n; k++)
      sum += a[i][k] * b[k][j];
    c[i][j] = sum;
  }
}
```

Variable `sum` *held in register*

Matrix Multiplication (ijk)

```
/* ijk */
for (i=0; i<n; i++)  {
  for (j=0; j<n; j++) {
    sum = 0.0;
    for (k=0; k<n; k++)
      sum += a[i][k] * b[k][j];
    c[i][j] = sum;
  }
}
```

Inner loop:



Misses per inner loop iteration:

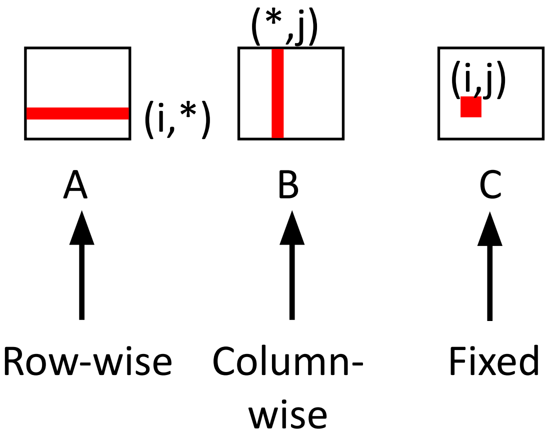
A B C

0.251.0 0.0

Matrix Multiplication (jik)

```
/* jik */
for (j=0; j<n; j++) {
  for (i=0; i<n; i++) {
    sum = 0.0;
    for (k=0; k<n; k++)
      sum += a[i][k] * b[k][j];
    c[i][j] = sum
  }
}
```

Inner loop:



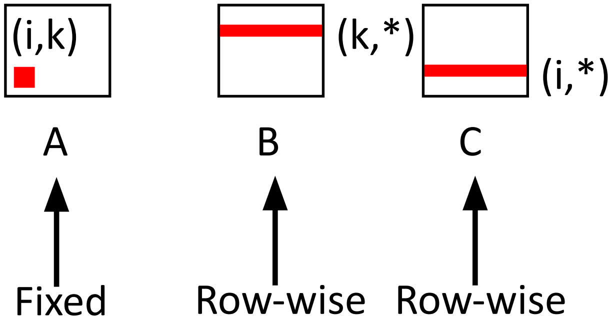
Misses per inner loop iteration:

A	B	C
0.25	1.0	0.0

Matrix Multiplication (kij)

```
/* kij */
for (k=0; k<n; k++) {
  for (i=0; i<n; i++) {
    r = a[i][k];
    for (j=0; j<n; j++)
      c[i][j] += r * b[k][j];
  }
}
```

Inner loop:



(i,k)

(k,*)

(i,*)

A

B

C

Fixed

Row-wise

Row-wise

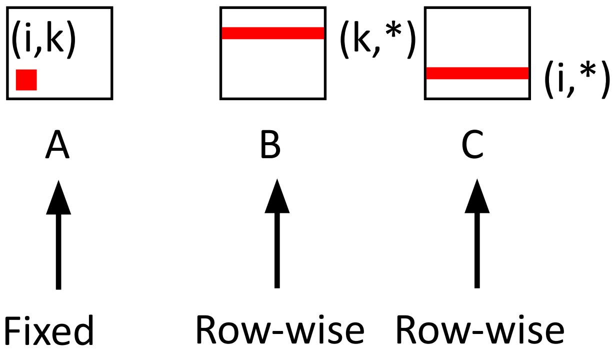
Misses per inner loop iteration:

A	B	C
0.0	0.25	0.25

Matrix Multiplication (ikj)

```
/* ikj */
for (i=0; i<n; i++) {
  for (k=0; k<n; k++) {
    r = a[i][k];
    for (j=0; j<n; j++)
      c[i][j] += r * b[k][j];
  }
}
```

Inner loop:



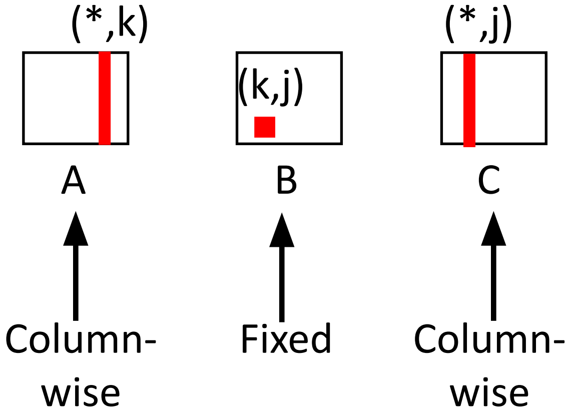
Misses per inner loop iteration:

A	B	C
0.0	0.25	0.25

Matrix Multiplication (jki)

```
/* jki */
for (j=0; j<n; j++) {
  for (k=0; k<n; k++) {
    r = b[k][j];
    for (i=0; i<n; i++)
      c[i][j] += a[i][k] * r;
  }
}
```

Inner loop:



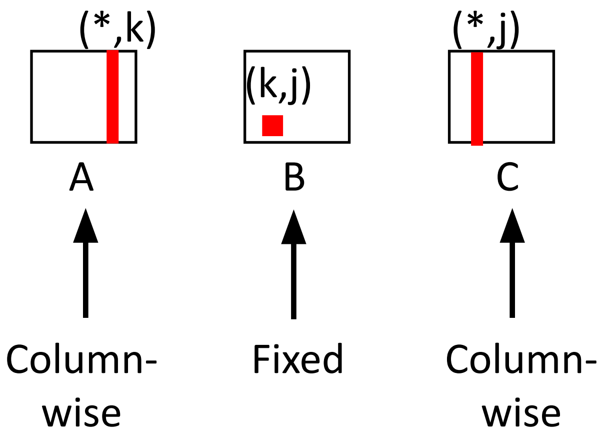
Misses per inner loop iteration:

A	B	C
1.0	0.0	1.0

Matrix Multiplication (kji)

```
/* kji */
for (k=0; k<n; k++) {
  for (j=0; j<n; j++) {
    r = b[k][j];
    for (i=0; i<n; i++)
      c[i][j] += a[i][k] * r;
  }
}
```

Inner loop:



(*,k)

(k,j)

(*,j)

A

B

C

Column-wise

Fixed

Column-wise

Misses per inner loop iteration:

A	B	C
1.0	0.0	1.0

Summary of Matrix Multiplication

k as inner loop

```
for (i=0; i<n; i++) {
  for (j=0; j<n; j++) {
   sum = 0.0;
   for (k=0; k<n; k++)
     sum += a[i][k] * b[k][j];
   c[i][j] = sum;
 }
}
```

ijk (& jik):
- 2 loads, 0 stores
- misses/iter = 1.25

j as inner loop

```
for (k=0; k<n; k++) {
 for (i=0; i<n; i++) {
  r = a[i][k];
  for (j=0; j<n; j++)
   c[i][j] += r * b[k][j];
 }
}
```

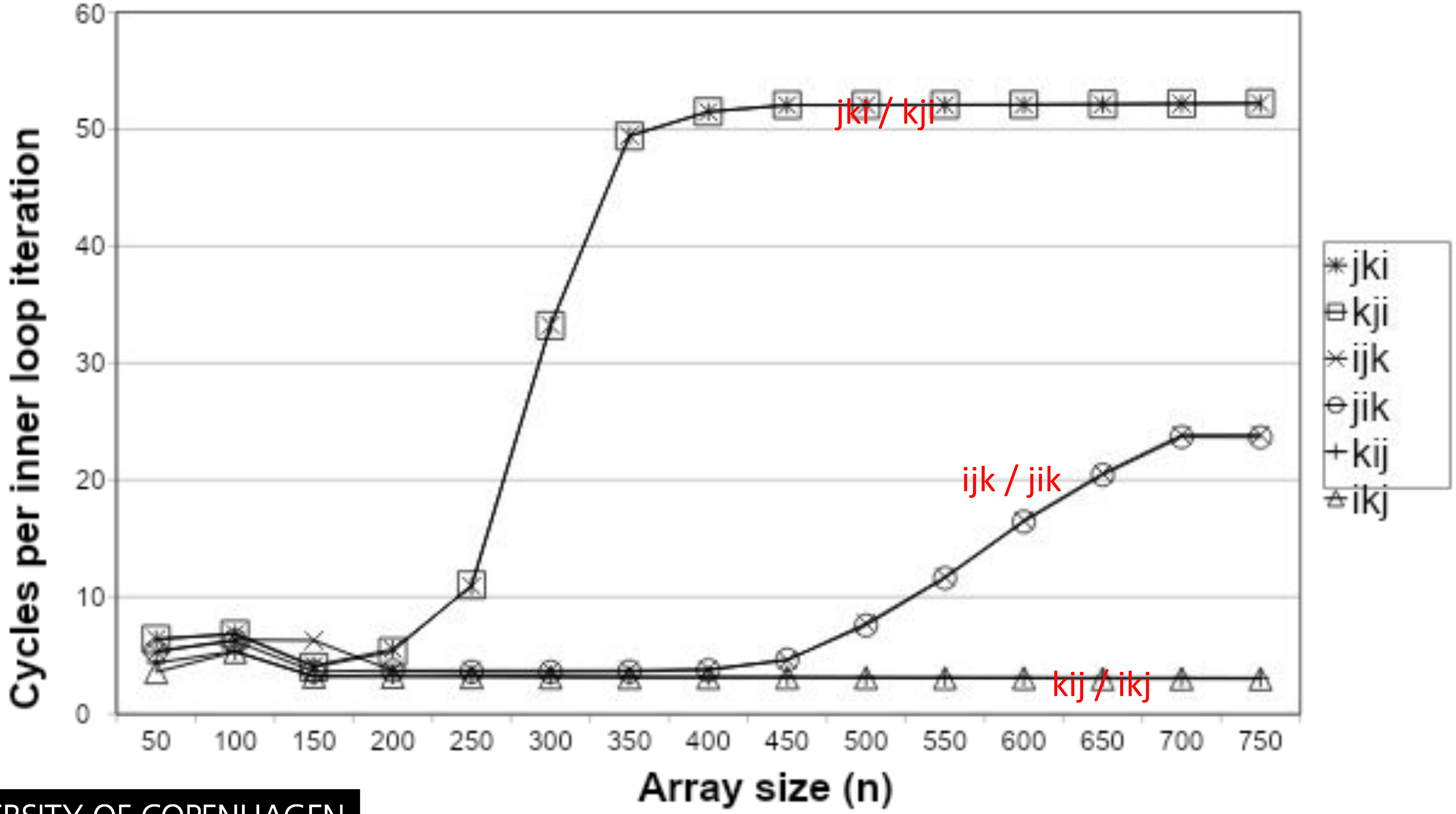
kij (& ikj):
- 2 loads, 1 store
- misses/iter = 0.5

i as inner loop

```
for (j=0; j<n; j++) {
 for (k=0; k<n; k++) {
   r = b[k][j];
   for (i=0; i<n; i++)
    c[i][j] += a[i][k] * r;
 }
}
```

jki (& kji):
- 2 loads, 1 store
- misses/iter = 2.0

Core i7 Matrix Multiply Performance



a note on security

Attacks: Side-Channel

Sharing is a Vulnerability

attackers exploit sharing.

> if A & B share E,
> A can **observe/affect** B through E.

different attacks depending on what is shared

- **hardware**
- network
- physical world (air-gap)

“somebody toucha my spaghet!”

fundamental tension:
security (isolation) vs.
performance (sharing)

Imitating the Ideal

ideal computer: infinite cores, infinite memory.

fake it

● OS multitasking	*time-sharing*
● memory hierarchy	*space-sharing*

important process requirement: *isolation*.
processes *share* resources.

isolation can be violated! (Unintended communication/interference)

How It Works

World

Sherlock

Can I use that resource?

No.

Why not?

Bob’s process is using it.

Why is Bob’s process using it?

Because Bob’s process took the **then**-branch (not **else**) in procedure-

Aha! From this, I conclude…!

CPU Timing Attacks

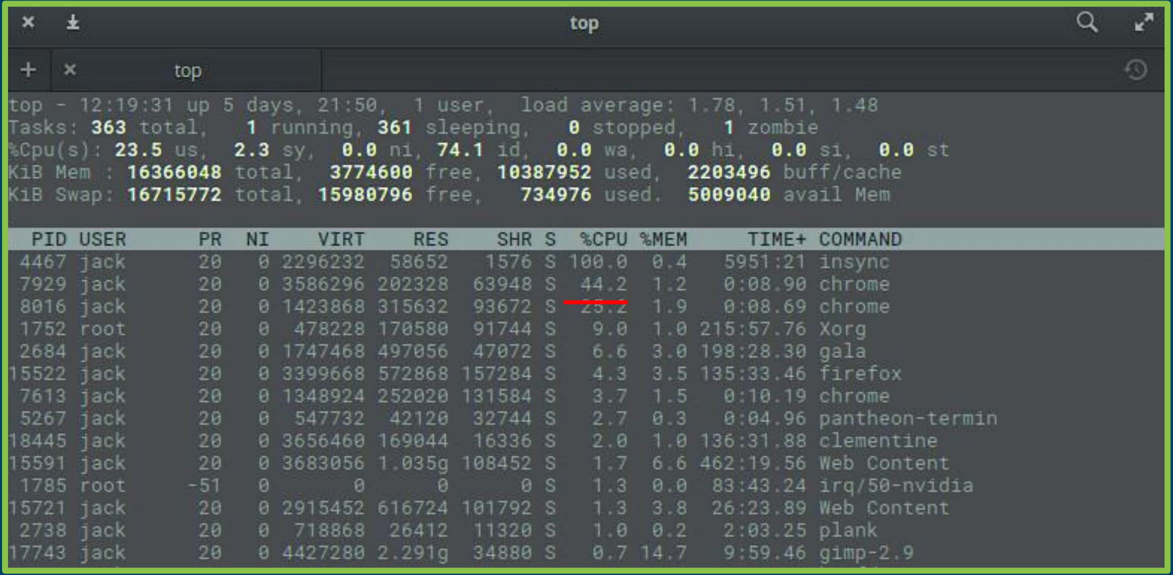
table of processes, task manager

Attack: Process A monitors the CPU load of Process B.

- High CPU load ⇒ 1
- Low CPU load ⇒ 0

Attack: Race conditions.

who writes to storage first



```
top - 12:19:31 up 5 days, 21:50,  1 user,  load average: 1.78, 1.51, 1.48
Tasks: 363 total,   1 running, 361 sleeping,   0 stopped,   1 zombie
%Cpu(s): 23.5 us,  2.3 sy,  0.0 ni, 74.1 id,  0.0 wa,  0.0 hi,  0.0 si,  0.0 st
KiB Mem : 16366048 total,  3774600 free, 10387952 used,  2203496 buff/cache
KiB Swap: 16715772 total, 15980796 free,   734976 used.  5009040 avail Mem

  PID USER      PR  NI    VIRT    RES    SHR S  %CPU %MEM     TIME+ COMMAND
 4467 jack      20   0 2296232  58652   1576 S 100.0  0.4   5951:21 insync
 7929 jack      20   0 3586296 202328  63948 S  44.2  1.2   0:08.90 chrome
 8016 jack      20   0 1423868 315632  93672 S  25.2  1.9   0:08.69 chrome
 1752 root      20   0  478228 170580  91744 S   9.0  1.0 215:57.76 Xorg
 2684 jack      20   0 1747468 497056  47072 S   6.6  3.0 198:28.30 gala
15522 jack      20   0 3399668 572868 157284 S   4.3  3.5 135:33.46 firefox
 7613 jack      20   0 1348924 252020 131584 S   3.7  1.5   0:10.19 chrome
 5267 jack      20   0  547732  42120  32744 S   2.7  0.3   0:04.96 pantheon-termin
18445 jack      20   0 3656460 169044  16336 S   2.0  1.0 136:31.88 clementine
15591 jack      20   0 3683056 1.035g 108452 S   1.7  6.6 462:19.56 Web Content
 1785 root     -51   0       0      0      0 S   1.3  0.0  83:43.24 irq/50-nvidia
15721 jack      20   0 2915452 616724 101792 S   1.3  3.8  26:23.89 Web Content
 2738 jack      20   0  718868  26412  11320 S   1.0  0.2   2:03.25 plank
17743 jack      20   0 4427280 2.291g  34880 S   0.7 14.7   9:59.46 gimp-2.9
```

Processes Share a Cache

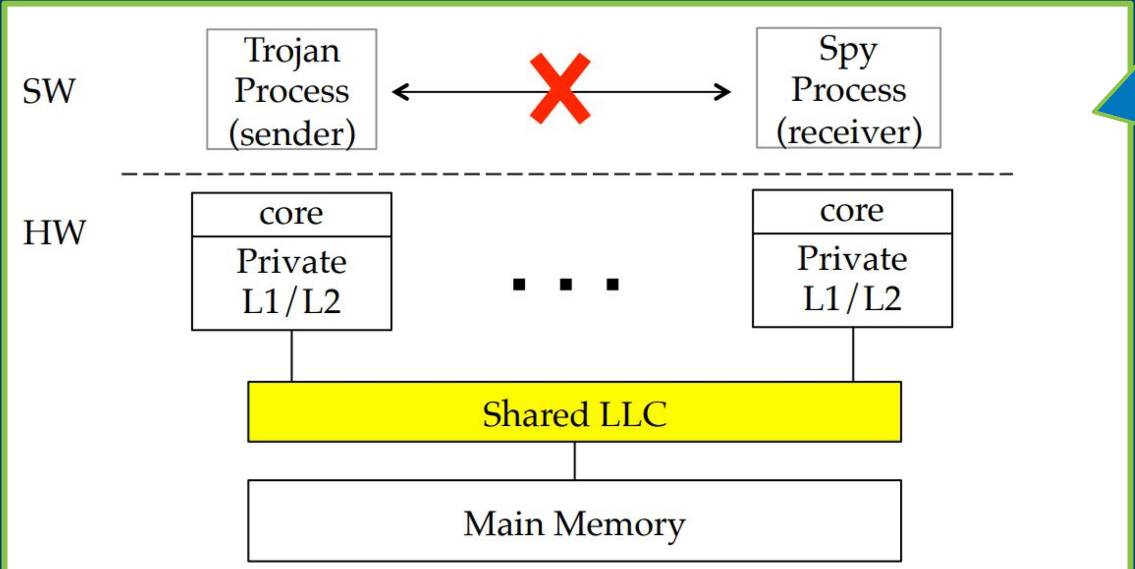


Fig. 3. LLC-based covert channel attack scenario.

different address space, though. How do they communicate?

Cache Timing Attack: Prime+Probe

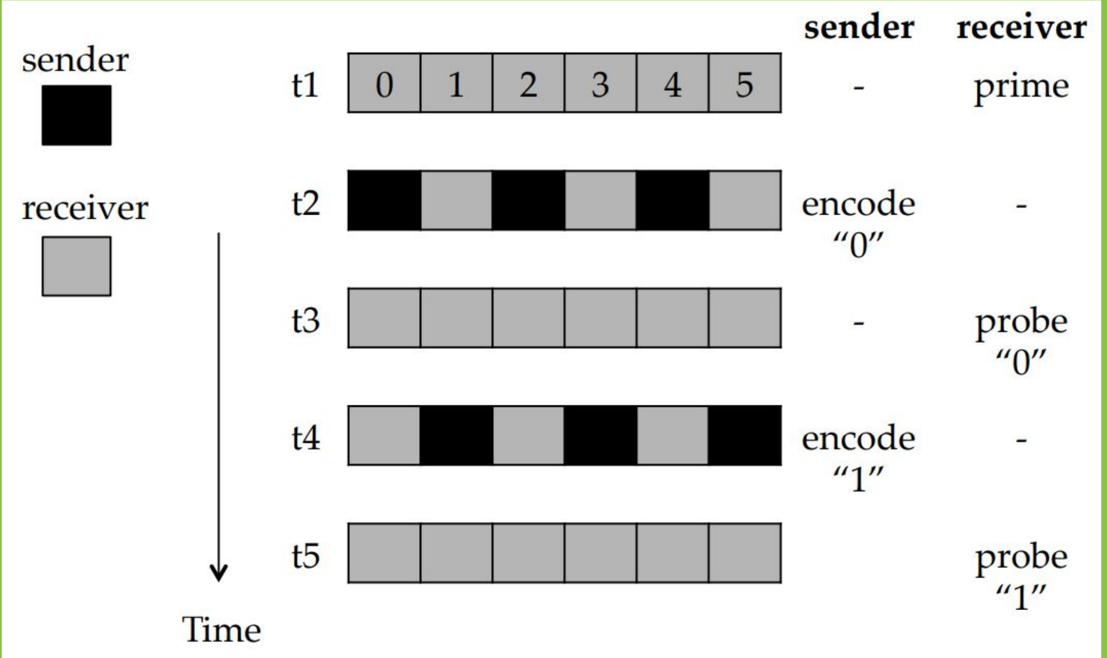


Fig. 1. Example of *Prime+Probe*.

estimate nr. of cache misses w/ a timer.

(remember memory hierarchy)

shameless self-promotion

goal: tools that developers can use to write secure SW.

sample research (past supervisions):

- analyze binaries for information leaks
- reduce timing leaks in the Linux kernel
- automatically fix vulnerabilities in JavaScript
- automatically generate (i.e. synthesize) a secure program from formal specification
- assess privacy risk in analytics programs (data scientists; Google search for “Privugger”)

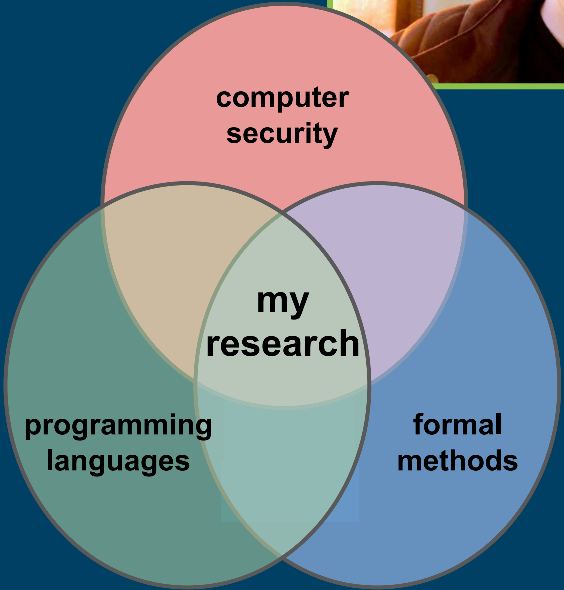
I like code, and I like proofs.

I created the “Applied Information Security” course.

I’m a barista in Analog.

Willard Rafnsson
IT University of Copenhagen

wilr@itu.dk
https://www.willardthor.com/



Take-Aways

- Locality in space / time is crucial for performance and scalability.
- Analyzing locality requires an understanding of (i) the memory hierarchy / cache memories, (ii) the layout of data structures in memory, and (iii) how loops lead to reuse of data in space and time.
- Performance & Security are fundamentally at odds (sharing vs isolation)