CS698L: Write Cache-Friendly Code

Swarnendu Biswas

Semester 2019-2020-I

CSE, IIT Kanpur

Content influenced by many excellent references, see References slide for acknowledgements.

Things important with Computer Systems:

- Correctness (obvious!)
- Performance
- Power

Correctness is Important!

- **AT&T hangs up its long-distance service (1990)**
  - For nine hours in January 1990 no AT&T customer could make a long-distance call. The problem was the software that controlled the company's long-distance relay switches—software that had just been updated. AT&T wound up losing $60 million in charges that day—a very expensive bug.

- **The Pentium chip's math error (1993)**

- **The Mars Climate Orbiter disintegrates in space (1998)**
  - NASA's $655-million robotic space probe plowed into Mars's upper atmosphere at the wrong angle, burning up in the process. The problem? In the software that ran the ground computers the thrusters' output was calculated in the wrong units (pound–seconds instead of newton–seconds).

What is Performance?

Execution time = Time (s) taken by a program to execute

What is Performance?

Execution time = Time (s) taken by a program to execute

$$Exec\ time = Time\ to\ execute\ \#\ instrs\ in\ the\ program$$

What is Performance?

Execution time = Time (s) taken by a program to execute

$$Exec\ time = Time\ to\ execute\ \#\ instrs\ in\ the\ program$$

$$Exec\ time = \frac{\#\ instrs}{program} * Time\ to\ execute\ 1\ instr$$

What is Performance?

Execution time = Time (s) taken by a program to execute

$$Exec\ time = \frac{\#\ instrs}{program} * Time\ to\ execute\ 1\ instr$$

$$Exec\ time = \frac{\#\ instrs}{program} * \frac{\#\ cycles}{instr} * Time\ to\ execute\ 1\ cycle$$

What is Performance?

Execution time = Time (s) taken by a program to execute

$$Exec\ time = \frac{\#\ instrs}{program} * \frac{\#\ cycles}{instr} * Time\ to\ execute\ 1\ cycle$$

$$Exec\ time = \frac{\#\ instrs}{program} * \frac{\#\ cycles}{instr} * \frac{\text{time (s)}}{\text{cycle}}$$

What is Performance?

Execution time = Time (s) taken by a program to execute

$$Exec\ time = \frac{\#\ instrs}{program} * CPI * \frac{1}{freq}$$

What is Performance?

Ex

Performance $\propto frequency$

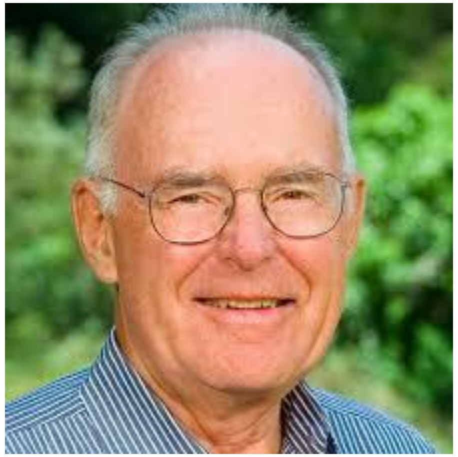
E

Performance $\propto \frac{1}{CPI}$

Buying Performance with Technological Innovations

- 1986 – 2005
  - Performance of microprocessors increased by ~50% per year
- Programs ran faster by themselves
  - We did not worry about performance
- Parallel computing, concurrent programming, and HPC were jobs for specialists

Moore’s Law



- Number of transistors on chip doubles every year
  - 1965
  - Recalibrated it later in 70’s to say “doubles every two years”

- David House from Intel said “improvements would cause performance to double every 18 months”

Moore’s Law

- Number of transistors on chip doubles every year
  - 1965
  - Recalibrated it later in 70’s to say “doubles every two years”

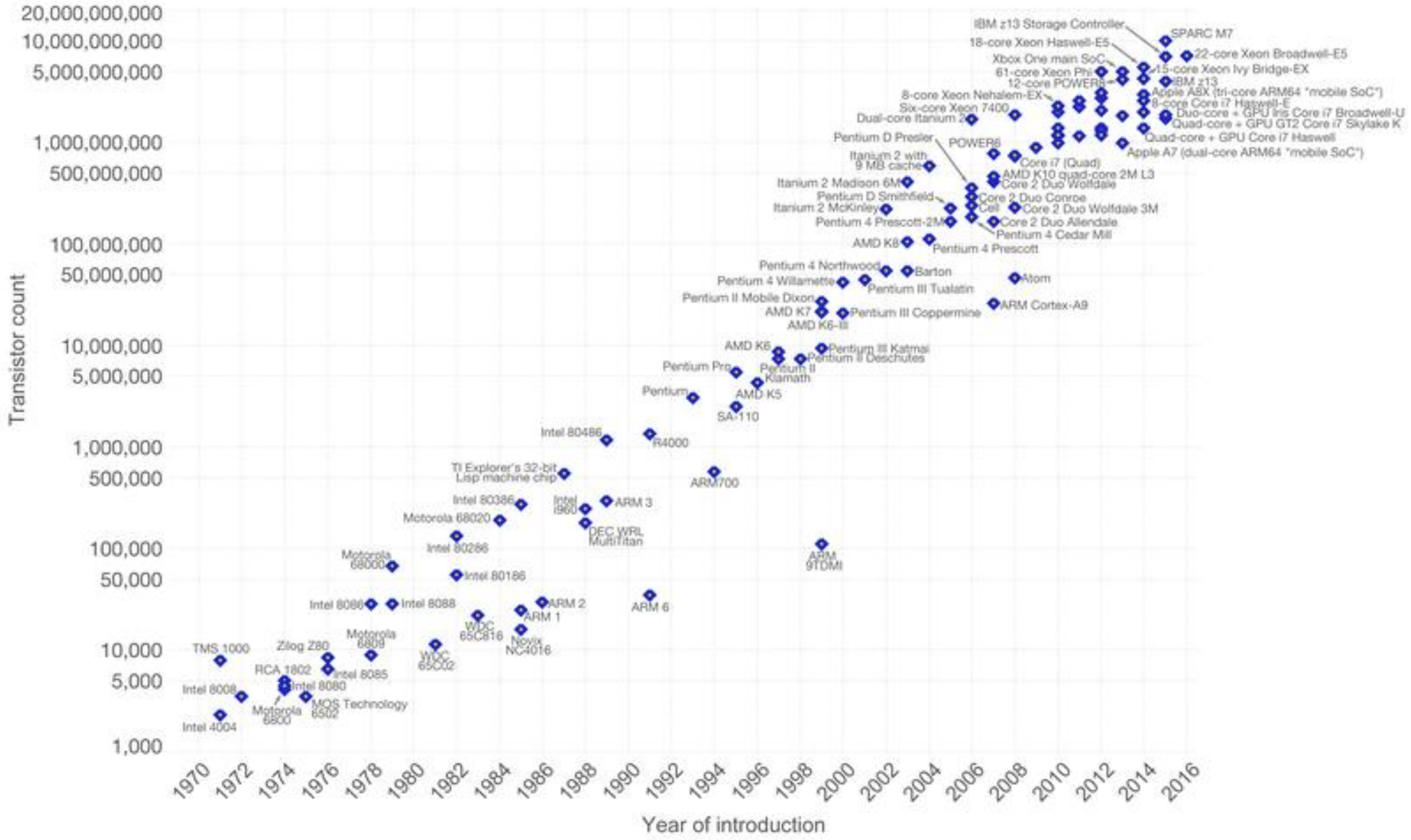
- David House from Intel said “improvements would cause performance to double every 18 months” 😯

> “Moore’s law is a violation of Murphy’s law.”

Moore's Law – The number of transistors on integrated circuit chips (1971-2016)

Our World in Data

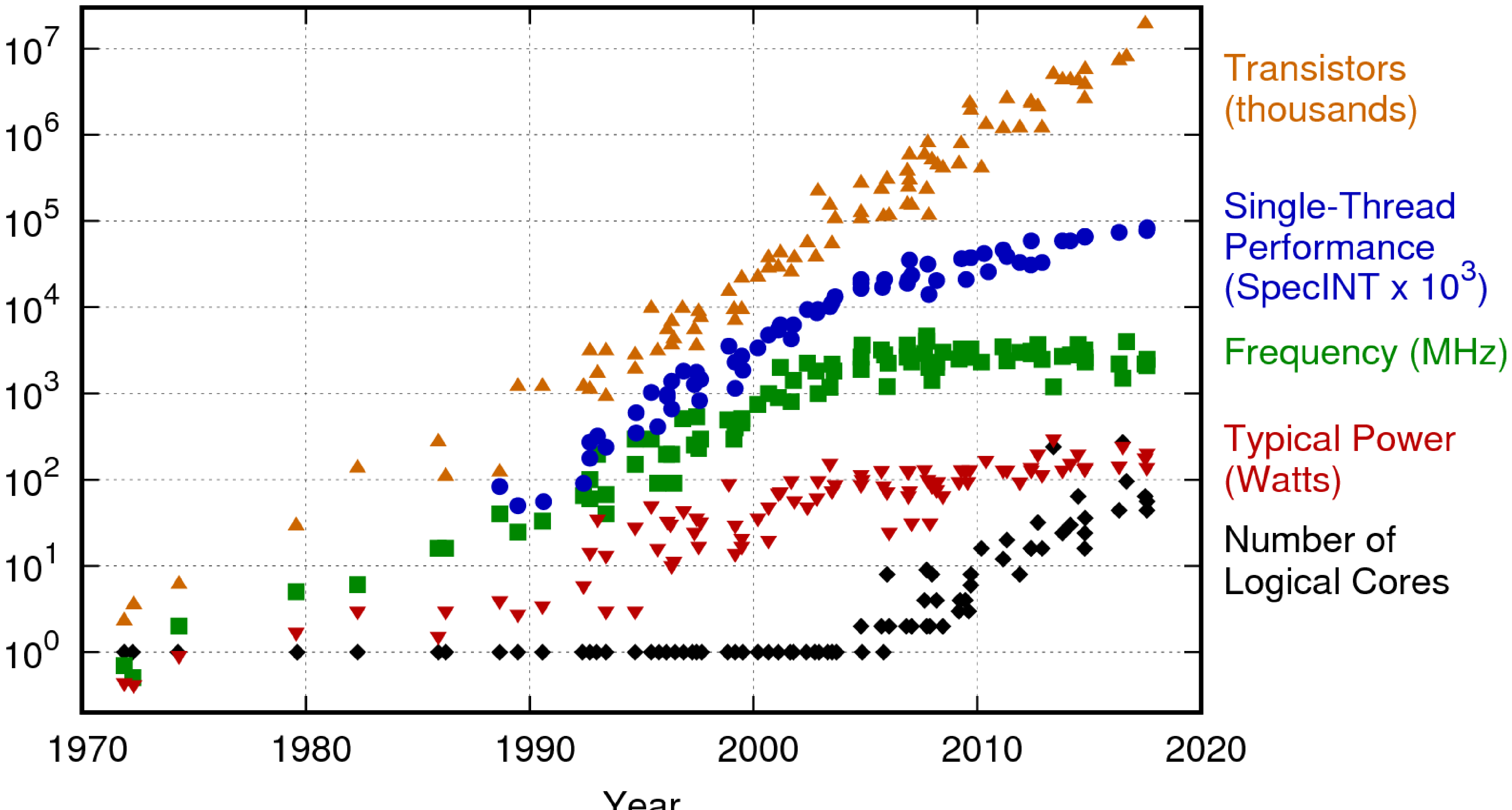
Moore's law describes the empirical regularity that the number of transistors on integrated circuits doubles approximately every two years. This advancement is important as other aspects of technological progress – such as processing speed or the price of electronic products – are strongly linked to Moore's law.



Data source: Wikipedia (https://en.wikipedia.org/wiki/Transistor_count)

The data visualization is available at OurWorldinData.org. There you find more visualizations and research on this topic.

42 Years of Microprocessor Trend Data

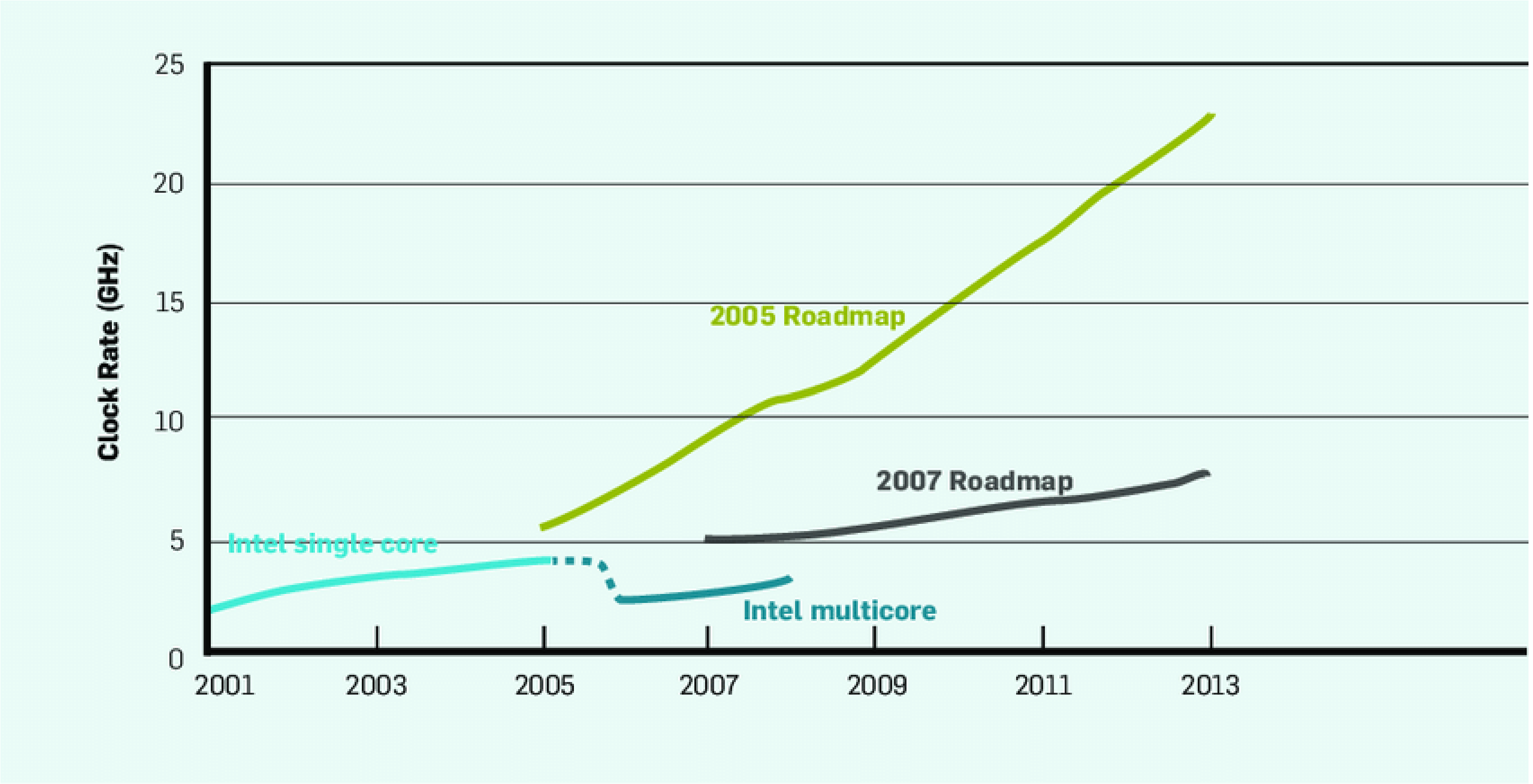


Original data up to the year 2010 collected and plotted by M. Horowitz, F. Labonte, O. Shacham, K. Olukotun, L. Hammond, and C. Batten
New plot and data collected for 2010-2017 by K. Rupp

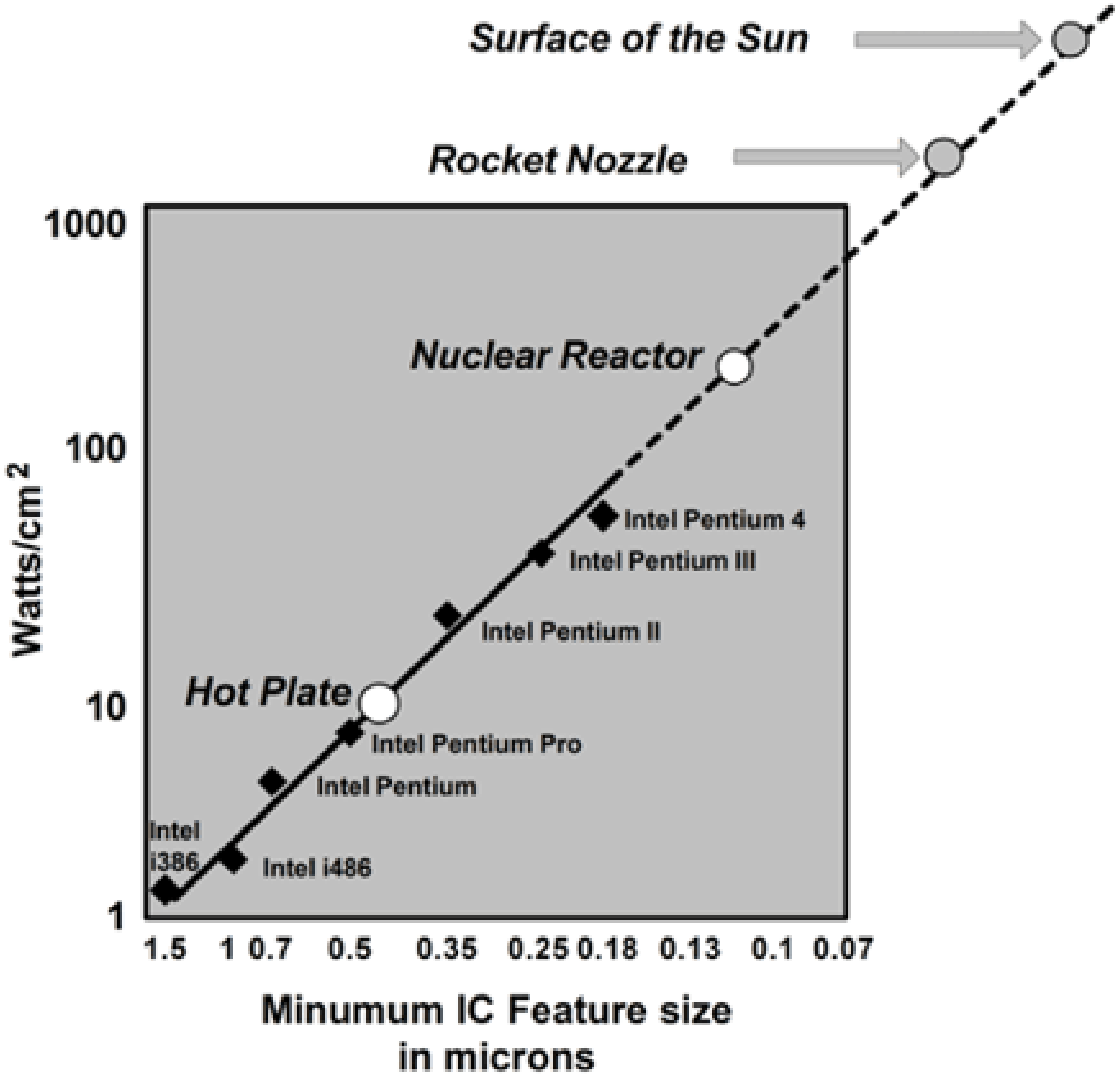
Challenges to Growth in Performance

Clock speeds are not increasing any more ?

Clock speeds are stagnating!



K. Asanovic et al. A View of the Parallel Computing Landscape. CACM, Oct 2009.



Mark Smotherman. https://people.cs.clemson.edu/~mark/330/power_density.gif

Power Wall

- Power, and not manufacturing, limits microarchitectural improvements – F. Pollack

Dynamic power ∝ Capacitive load x Voltage^2 x Frequency

Hardware Trends in the Last Ten Years!

- 2005 – 2018
  - Single core performance increase is ~20%
- **Programs do not run any faster by themselves**

Programs Do Not Run Any Faster by Themselves!

- Microarchitectural techniques
  - Add more functional units to improve ILP
    - Superscalar architecture, VLIW, more cache structures (e.g., L4 caches), deeper pipelines

Programs Do Not Run Any Faster by Themselves!

- Microarchitectural techniques
  - Add more functional units to improve ILP
  - pipelines

Law of diminishing returns!

There is little or no more hidden parallelism (ILP) to be found

Multicore Architecture

- Make effective use of the extra transistors
  - Chip density is continuing to double every two years
- New prediction: # cores will double every two years
- We now have manycore machines

What is the software side of the story?

Develop Parallel Programs

From my perspective, parallelism is the biggest challenge since high-level programming languages. It’s the biggest thing in 50 years because industry is betting its future that parallel programming will be useful.

…

Industry is building parallel hardware, assuming people can use it. And I think there’s a chance they’ll fail since the software is not necessarily in place. So this is a gigantic challenge facing the computer science community.

– David Patterson, ACM Queue, 2006.

Develop Parallel Programs

To save the IT industry, researchers must demonstrate greater end-user value of from an increasing number of cores –
A View of Parallel Computing Landscape, CACM 2009.

New Challenges in Software Development

- Adapt to the changing hardware landscape
- Most applications are single-threaded

How can we develop software that makes effective use of the extra hardware?

Challenges in Developing Parallel Programs

- Programmers tend to **think sequentially**
  - Correctness issues – concurrency bugs like data races and deadlocks
  - Performance issues – minimize communication across cores

Programmer’s tend to think sequentially

Atomicity Violation

Thread 1	**Thread 2**
`if (thd->proc_info)`	
	`thd->proc_info = NULL;`
`  fputs(thd->proc_info, …)`	

MySQL
ha_innodb.cc

Order Violation

Thread 1

```
void init(…) {
  …
  mThread=
      PR_CreateThread(mMain, …);
  …
}
```

Thread 2

```
void mMain( ) {
  mState=mThread->State;
}
```

Mozilla
nsthread.cpp

Deadlock

```
public class Account {
  int bal = 0;
  synchronized void transfer(int x, Account trg) {
    this.bal -= x;
    trg.deposit(x);
  }
  synchronized void deposit(int x) {
    this.bal += x;
  }
}
```

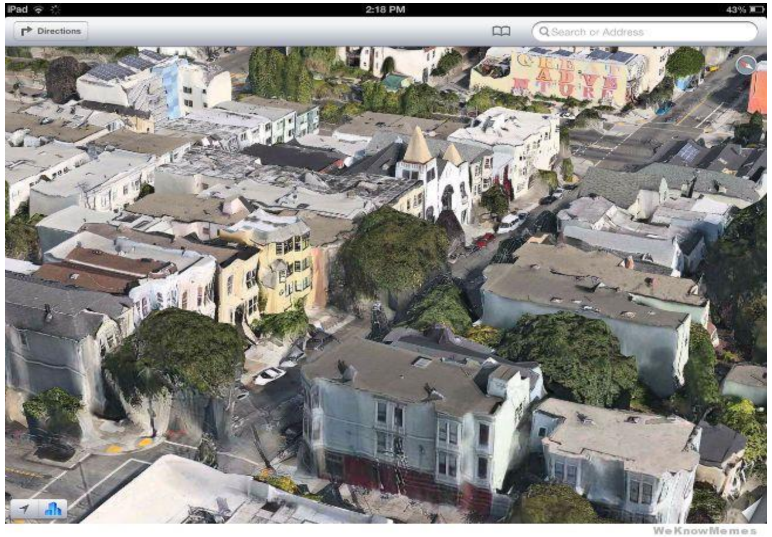
Starvation and Livelock

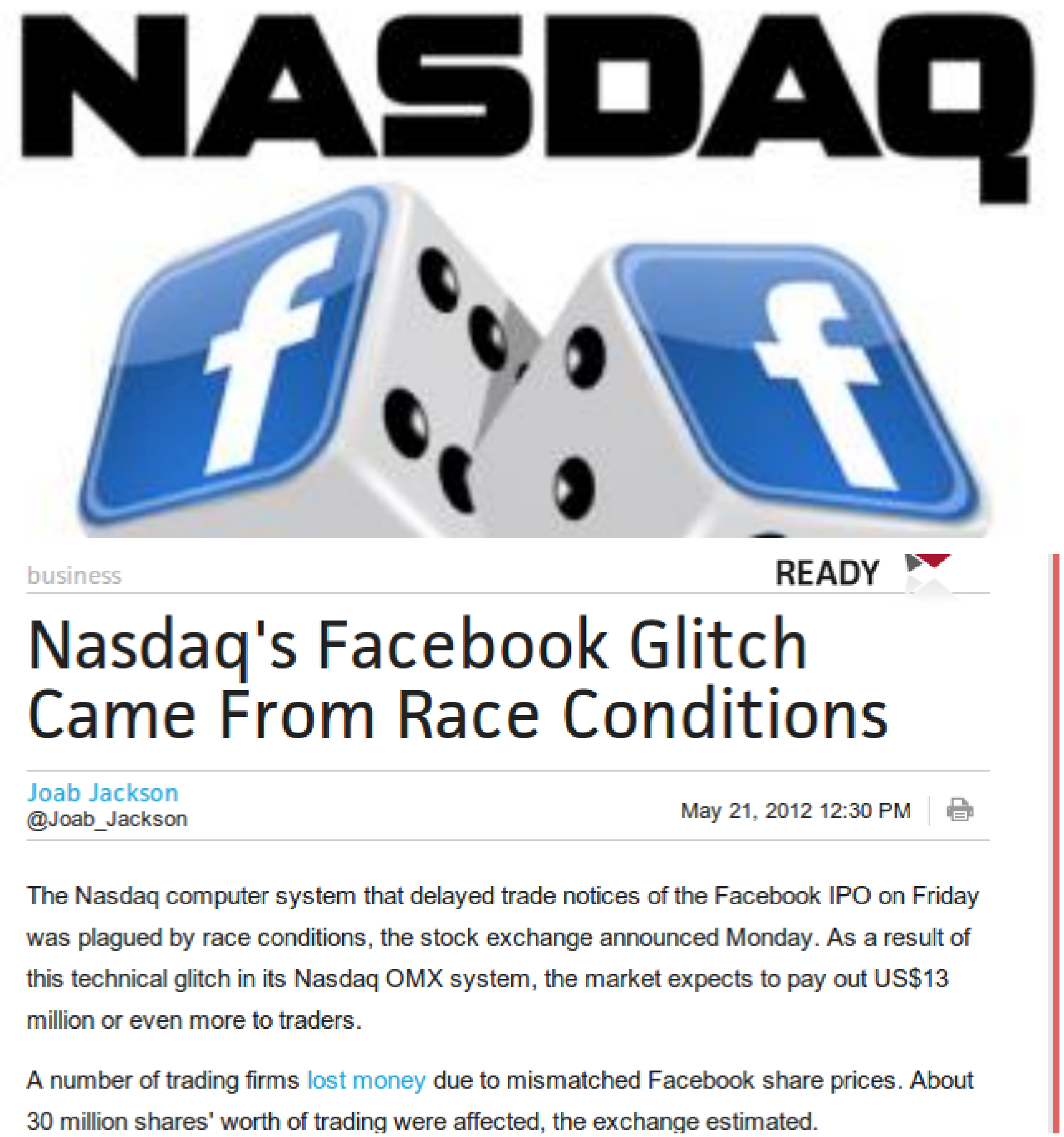
- **Starvation**
  - A thread is unable to get regular access to shared resources and so is unable to make progress

- **Livelock**
  - Threads are not blocked, their states change, but they are unable to make progress

Examples of Real-World Concurrency Bugs

003/45/7844
50 million people
ISAT GeoStar 45
23:15 EST 14 Aug. 2003





business

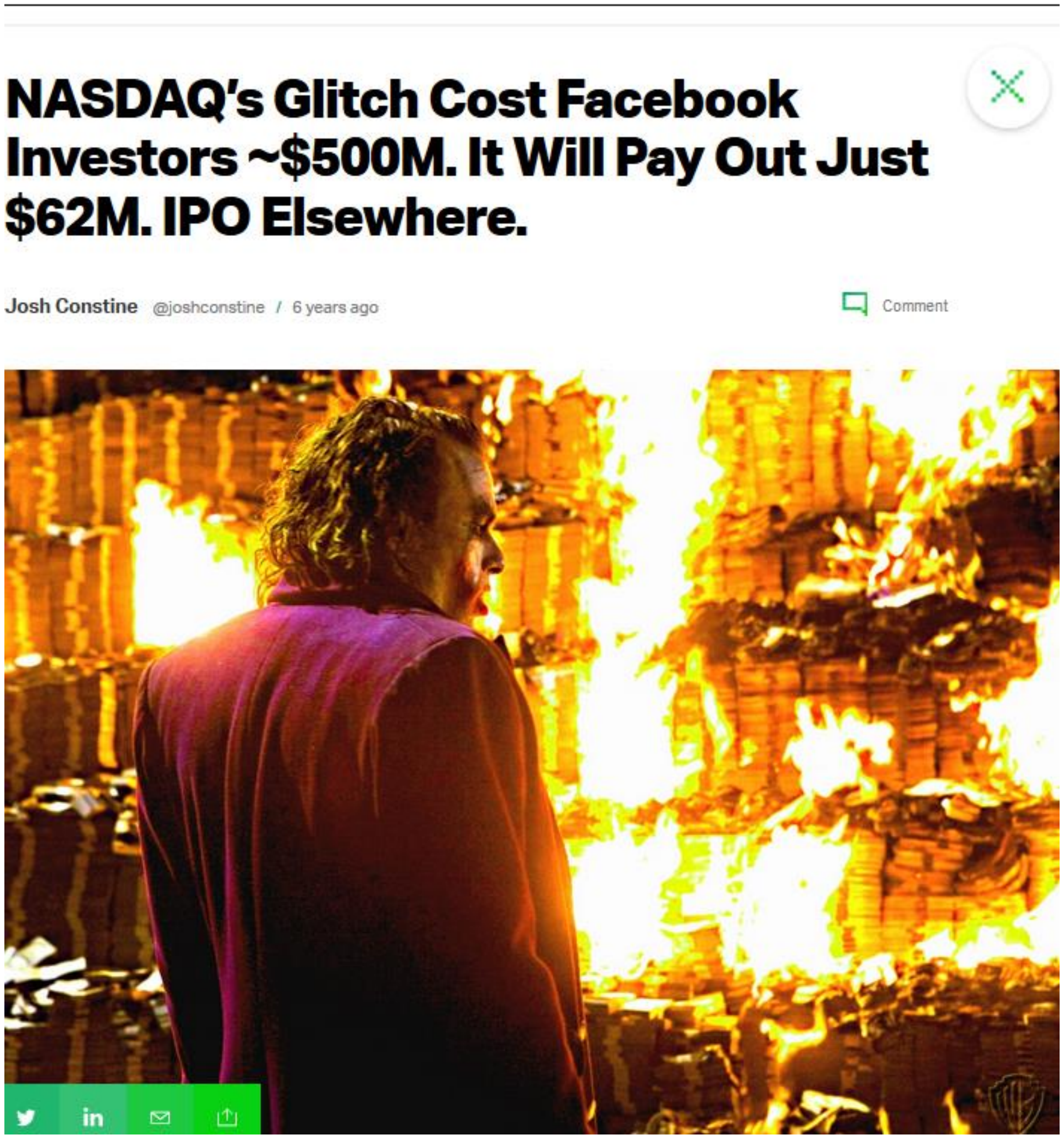
Nasdaq's Facebook Glitch Came From Race Conditions

Joab Jackson
@Joab_Jackson

May 21, 2012 12:30 PM

The Nasdaq computer system that delayed trade notices of the Facebook IPO on Friday was plagued by race conditions, the stock exchange announced Monday. As a result of this technical glitch in its Nasdaq OMX system, the market expects to pay out US$13 million or even more to traders.

A number of trading firms lost money due to mismatched Facebook share prices. About 30 million shares' worth of trading were affected, the exchange estimated.



NASDAQ's Glitch Cost Facebook Investors ~$500M. It Will Pay Out Just $62M. IPO Elsewhere.

Josh Constine @joshconstine / 6 years ago

KILLED BY A MACHINE: THE THERAC-25

by: Adam Fabio

139 Comments

October 26, 2015

The Therac-25 was not a device anyone was happy to see. It was a radiation therapy machine. In layman's terms it was a "cancer zapper"; a linear accelerator with a human as its target. Using X-rays or a beam of electrons,

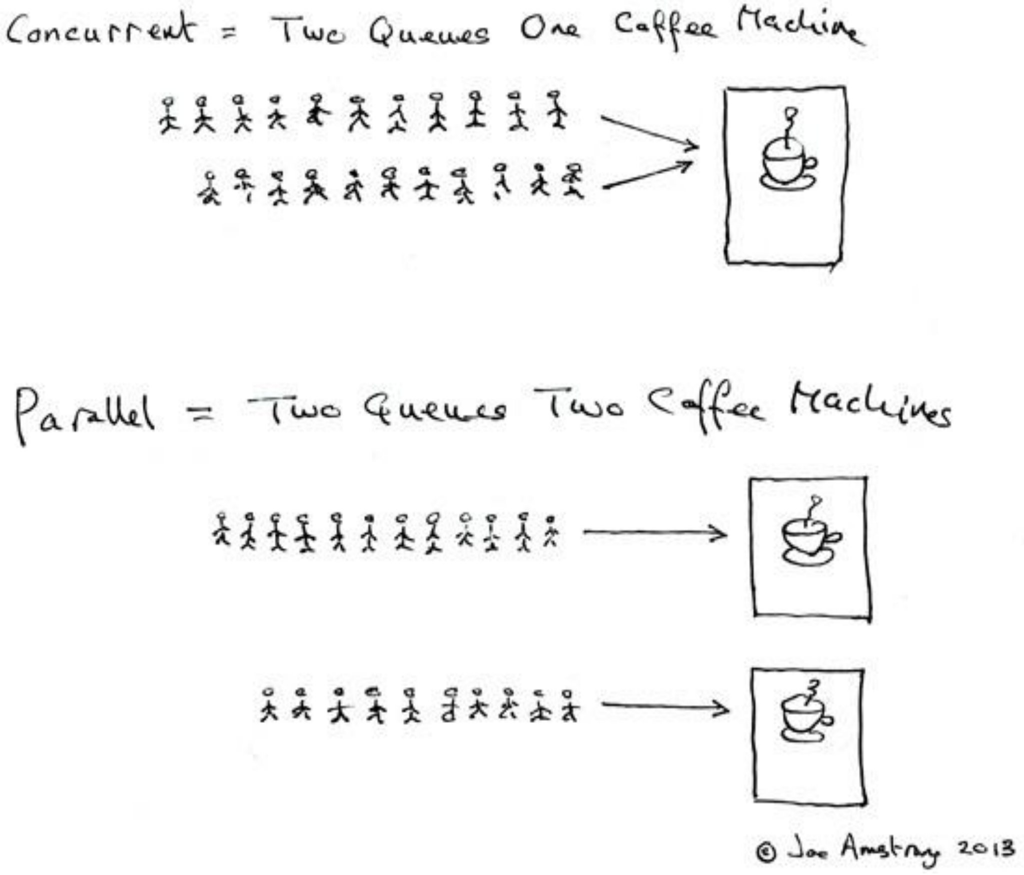
Therac-25 Accident

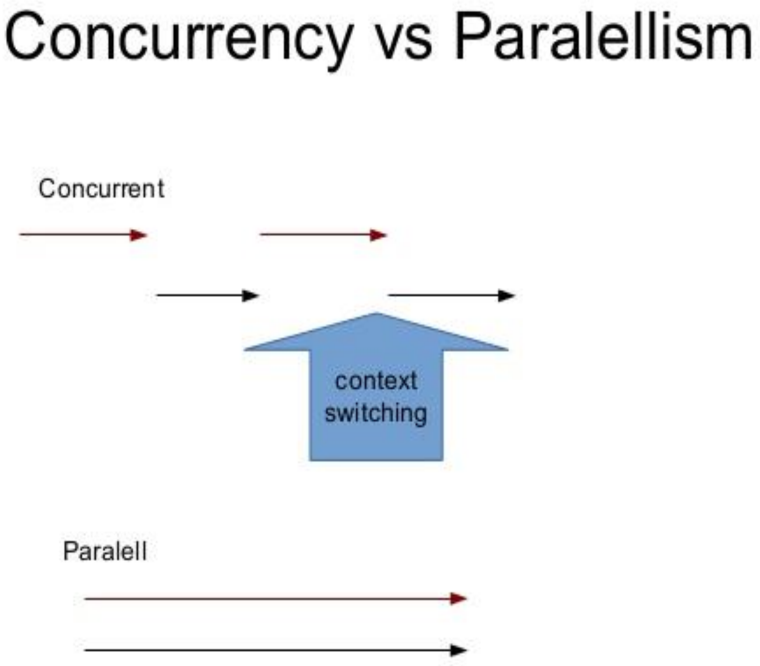
- Therac-25 was a computer-controlled radiation therapy machine

- It was involved in at least six accidents between 1985 and 1987, in which patients were given massive overdoses of radiation. Because of concurrent programming errors, it sometimes gave its patients radiation doses that were hundreds of times greater than normal, resulting in death or serious injury.

https://en.wikipedia.org/wiki/Therac-25

Parallelism vs Concurrency


Concurrent = Two Queues One Coffee Machine
Parallel = Two Queues Two Coffee Machines
© Joe Armstrong 2013



Parallelism vs Concurrency

Parallel programming

- Use additional resources to speed up computation
- Performance perspective

Concurrent programming

- Correct and efficient control of access to shared resources
- Correctness perspective

Distinction is not absolute

Challenges in Developing Parallel Programs

- Programmers tend to **think sequentially**
  - Correctness issues – concurrency bugs like data races and deadlocks
  - Performance issues – minimize communication across cores
- Amdahl's law
- Overheads of parallel execution
- Other challenges: load balancing

We will focus on performance aspects!

How to Write Efficient and Scalable Programs?

Good choice of algorithms and data structures

- Determines number of operations executed

Code that the compiler and architecture can effectively optimize

- Determines number of instructions executed

Proportion of parallelizable and concurrent code

- Amdahl's law

Sensitive to the architecture platform

- Efficiency and characteristics of the platform
- For e.g., memory hierarchy, cache sizes

Let us compare the performance!

```
for (i = 0; i < 100000000; i++) {
  W = 1.599999 * X;
  X = 0.999999 * W;
}
```

```
for (i = 0; i < 100000000; i++) {
  W = 1.599999 * W + 0.000001;
  X = 0.999999 * X;
  Y = 3.14159 * Y + 0.000001;
  Z = Z + 1.0001;
}
```

Adapted from CS 5441 by P. Sadayappan @ Ohio State University

Let us compare the performance!

```
for (i = 0; i < 100000000; i++) {
  W = 1.599999 * X;
  X = 0.999999 * W;
}
```

550-600 ms

```
for (i = 0; i < 100000000; i++) {
  W = 1.599999 * W + 0.000001;
  X = 0.999999 * X;
  Y = 3.14159 * Y + 0.000001;
  Z = Z + 1.0001;
}
```

??? ms

Let us compare the performance!

```
for (i = 0; i < 100000000; i++) {
  W = 1.599999 * X;
  X = 0.999999 * W;
}
```

550-600 ms

```
for (i = 0; i < 100000000; i++) {
  W = 1.599999 * W + 0.000001;
  X = 0.999999 * X;
  Y = 3.14159 * Y + 0.000001;
  Z = Z + 1.0001;
}
```

350-400 ms

Let us compare the performance!

```
#define N 32
#define T 1024 * 1024
double A[N][N];

for (it = 0; it < T; it++)
  for (j = 0; j < N; j++)
    for (i = 0; i < N; i++)
      A[i][j] += 1;
```

- #define N 32
- #define T 1024 * 1024

230 ms

Let us compare the performance!

```
#define N 32
#define T 1024 * 1024
double A[N][N];

for (it = 0; it < T; it++)
  for (j = 0; j < N; j++)
    for (i = 0; i < N; i++)
      A[i][j] += 1;
```

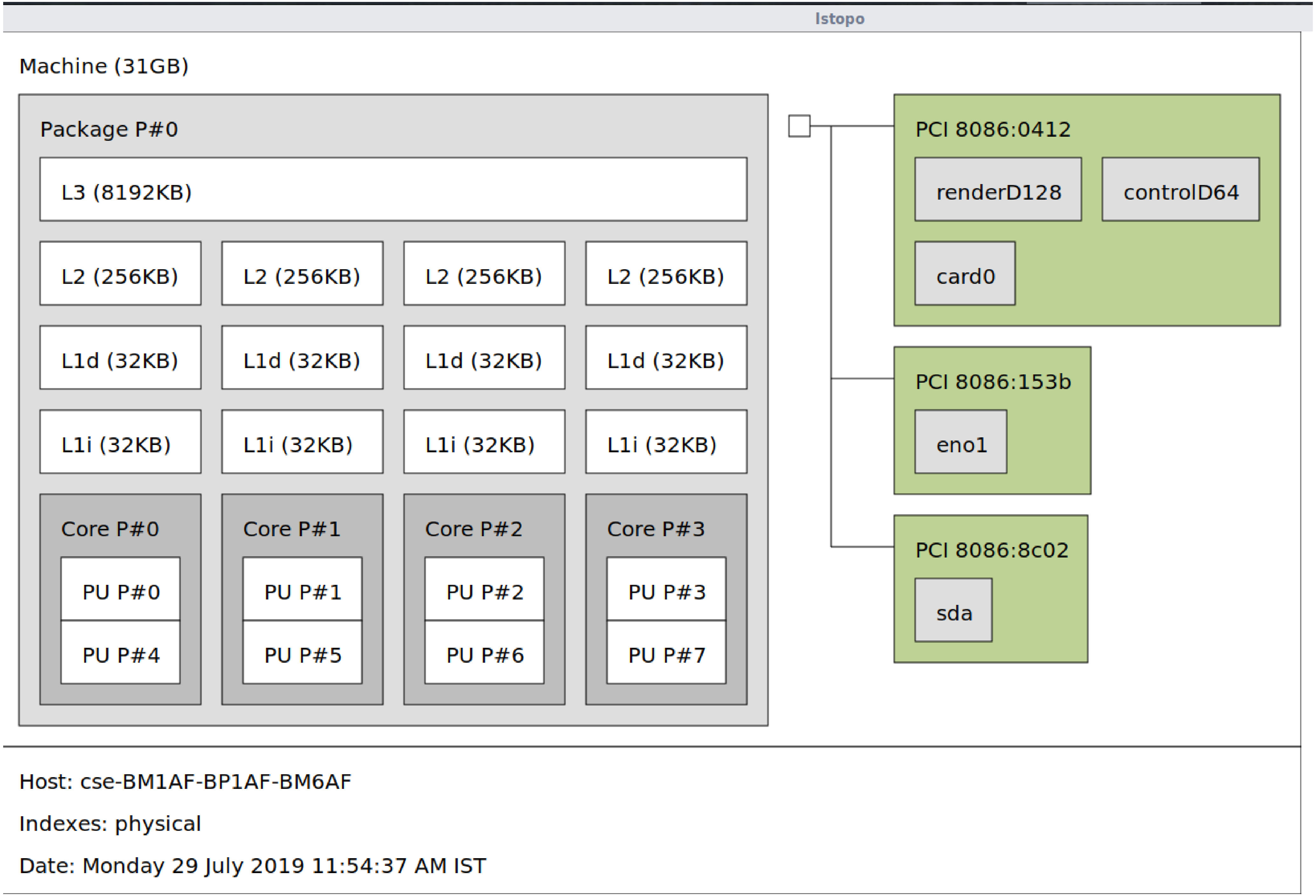
- `#define N 32`
- `#define T 1024 * 1024`

- `#define N 128`
- `#define T 1024 * 1024`

- `#define N 256`
- `#define T 1024 * 1024`

- `#define N 4096`
- `#define T 1024 * 1024`

??? ms



Host: cse-BM1AF-BP1AF-BM6AF

Indexes: physical

Date: Monday 29 July 2019 11:54:37 AM IST

Let us compare the performance!

```
#define N 32
#define T 1024 * 1024
double A[N][N];

for (it = 0; it < T; it++)
  for (j = 0; j < N; j++)
    for (i = 0; i < N; i++)
      A[i][j] += 1;
```

- #define N 32
- #define T 1024 * 1024

235 ms

- #define N 128
- #define T 1024 * 1024

240 ms

- #define N 256
- #define T 1024 * 1024

430 ms

- #define N 4096
- #define T 1024 * 1024

720 ms

Cache Memory: Quick Recap

Slides adapted from Bryant and O’Hallaron (CS 15-213 @ CMU)

Understanding the Memory Hierarchy

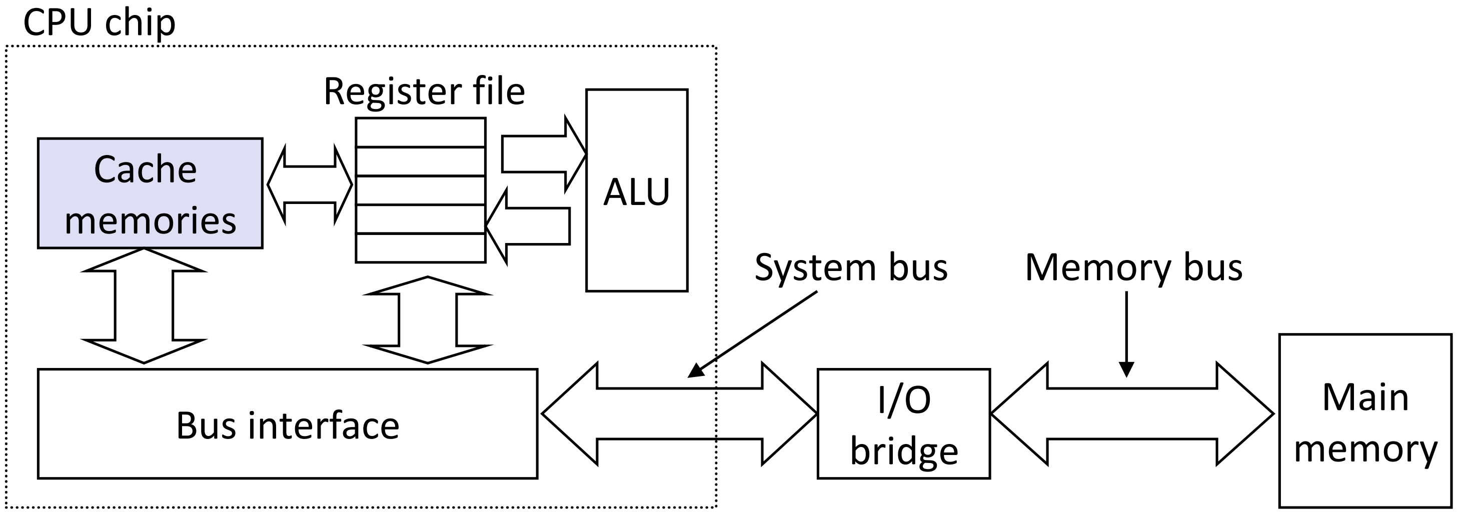
- Cache: A small, fast storage device that acts as a staging area for a subset of the data in a larger but slower device.

- **Key insight**
  - The memory hierarchy creates a large pool of storage that costs as much as the cheap storage near the bottom, but that serves data to programs at the rate of the fast storage near the top.

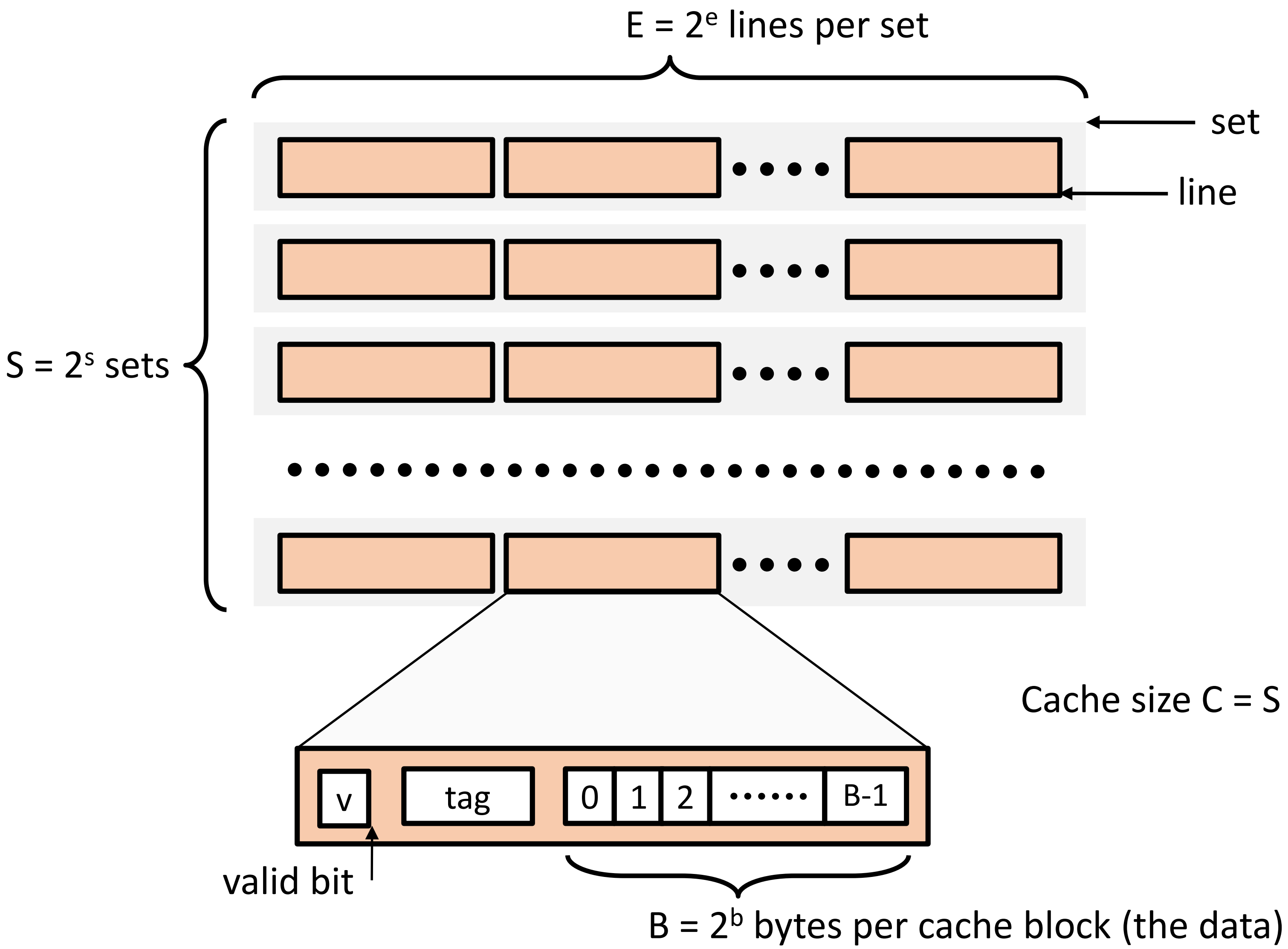
  - Because of locality, programs tend to access the data at level k more often than they access the data at level k+1.
  - For each k, the faster, smaller device at level k serves as a cache for the larger, slower device at level k+1.

Cache Memories

- 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:

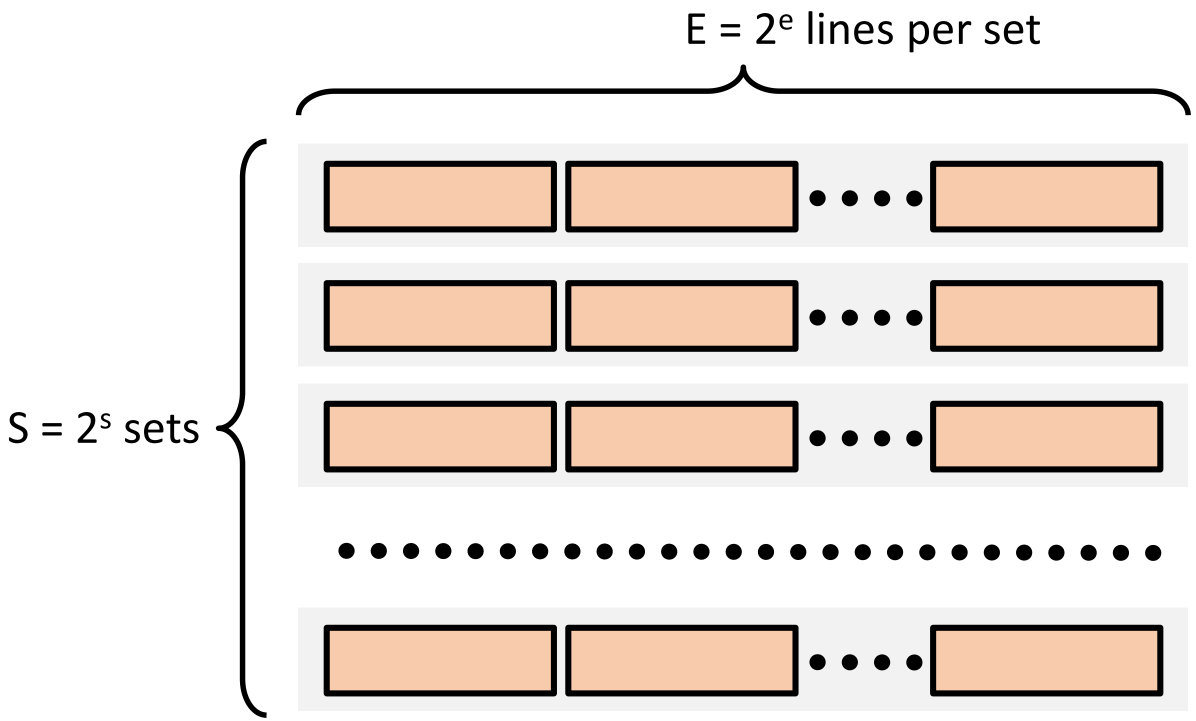


General Cache Organization (S, E, B)



Cache size C = S x E x B data bytes

Cache Read

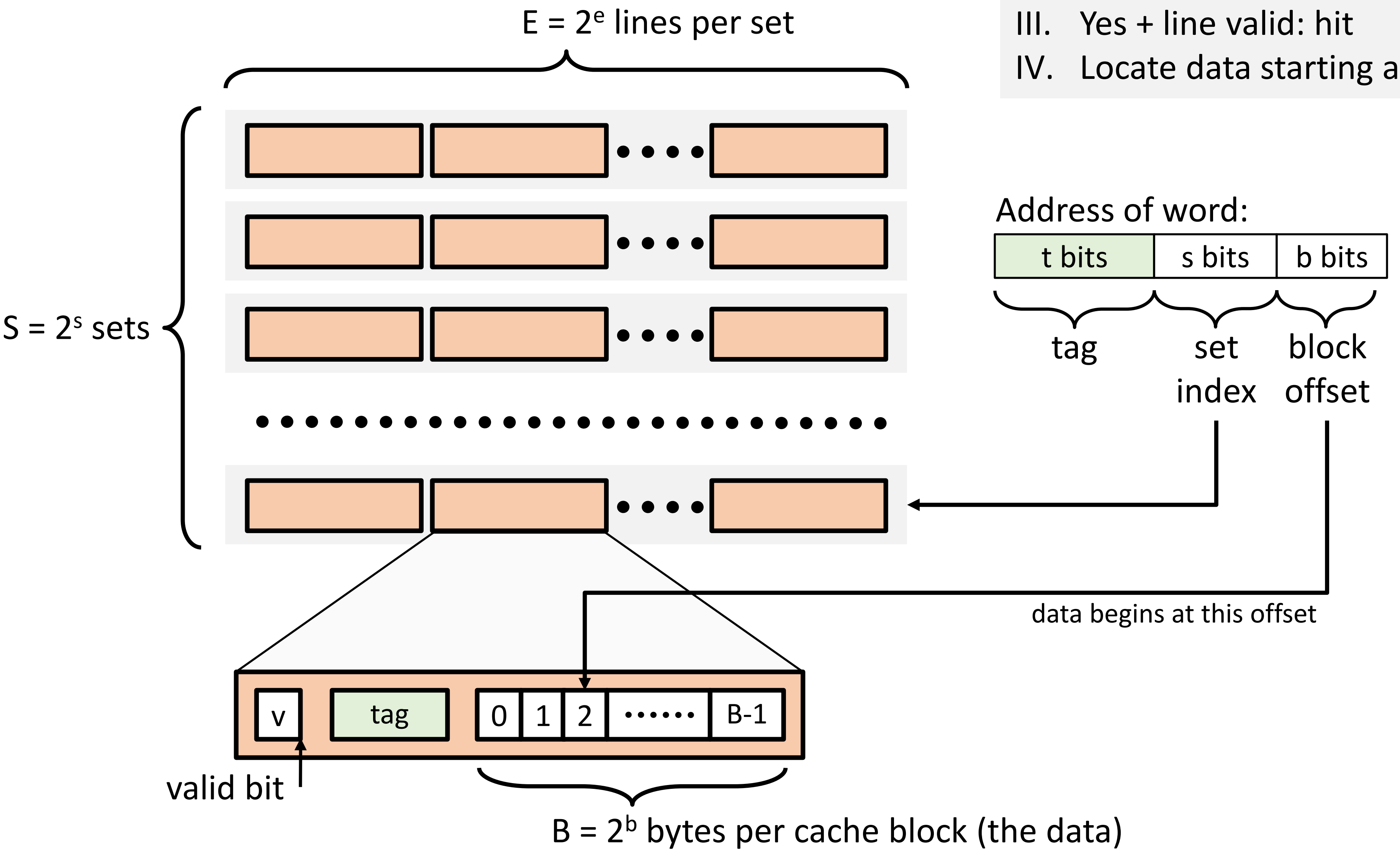


Address of word:

t bits	s bits	b bits
tag	set index	block offset

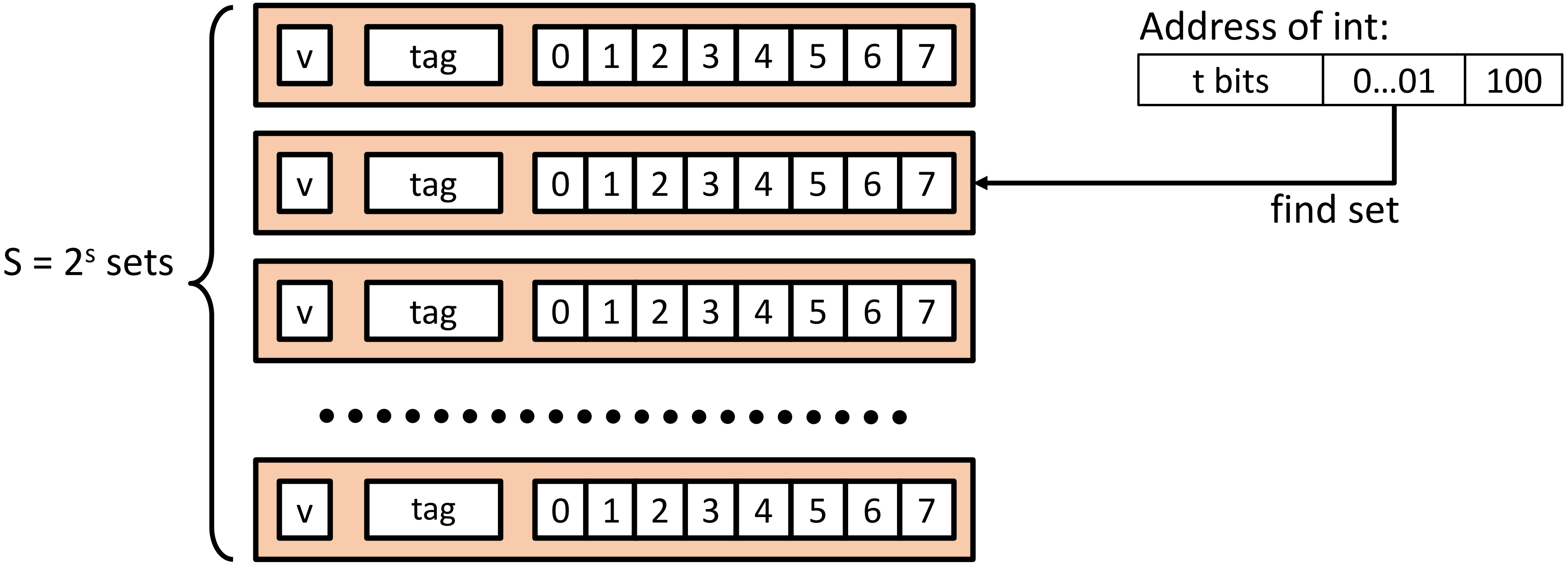
Cache Read

I. Locate set
II. Check if any line in set has matching tag
III. Yes + line valid: hit
IV. Locate data starting at offset



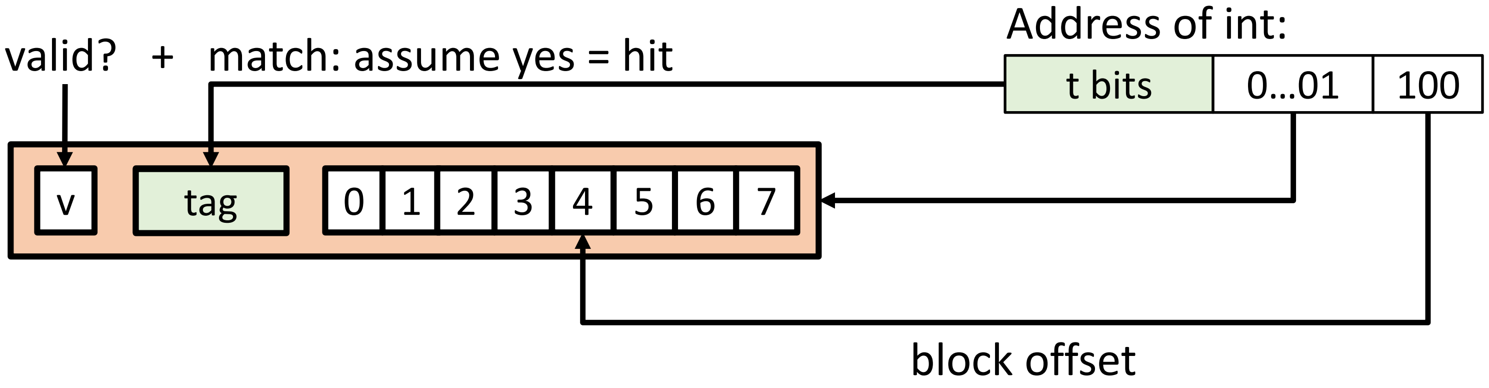
Example: Direct Mapped Cache (E = 1)

Direct mapped: One line per set
Assume: cache block size 8 bytes



Example: Direct Mapped Cache (E = 1)

Direct mapped: One line per set
Assume: cache block size 8 bytes



Direct-Mapped Cache Simulation

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

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

Address trace (reads, one byte per read):

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

	v	Tag	Block
Set 0			
Set 1			
Set 2			
Set 3			

Direct-Mapped Cache Simulation

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

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

Address trace (reads, one byte per read):

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

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

Direct-Mapped Cache Simulation

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

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

Address trace (reads, one byte per read):

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

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

Direct-Mapped Cache Simulation

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

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

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	$[1\underline{00}0_2]$,	
0	$[0\underline{00}0_2]$	

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

Direct-Mapped Cache Simulation

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

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

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	$[0\underline{00}0_2]$	

	v	Tag	Block
Set 0	1	1	M[8-9]
Set 1			
Set 2			
Set 3	1	0	M[6-7]

Direct-Mapped Cache Simulation

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

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

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

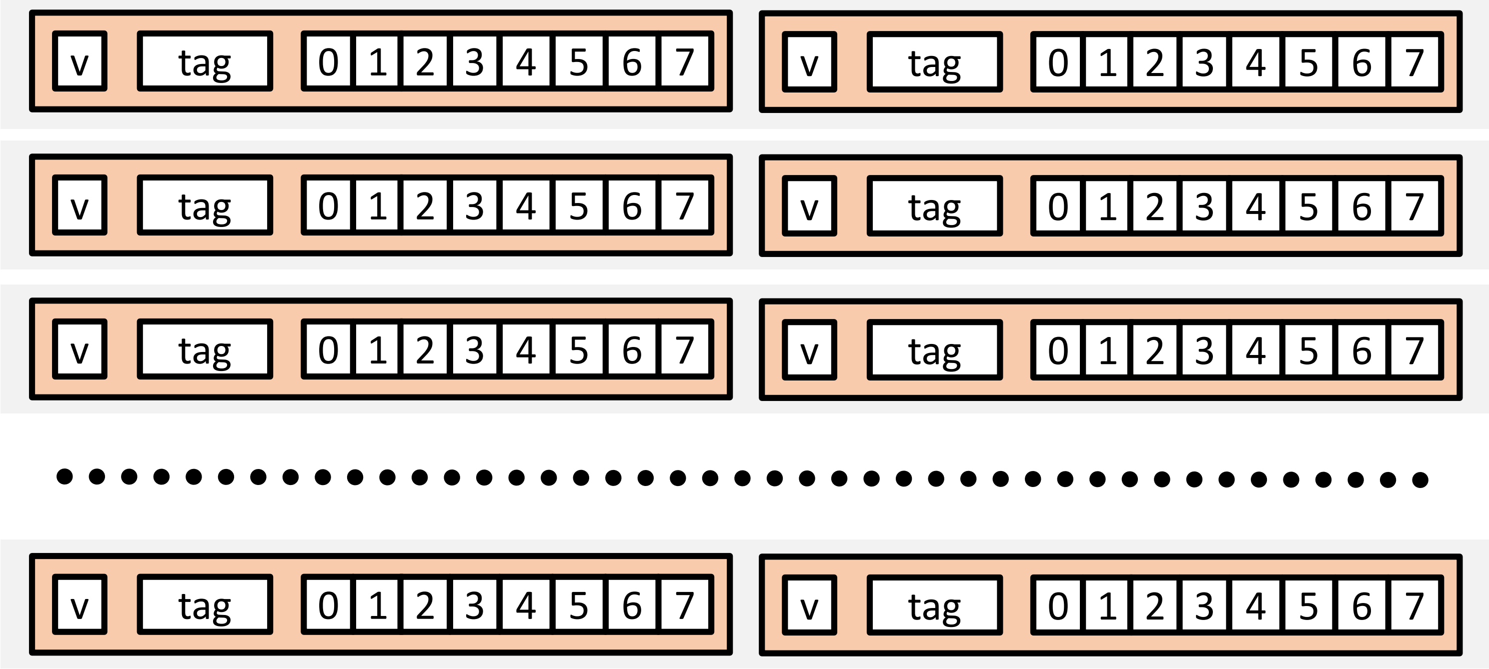
	v	Tag	Block
Set 0	1	0	M[0-1]
Set 1			
Set 2			
Set 3	1	0	M[6-7]

E-way Set Associative Cache (Here: E = 2)

E = 2: Two lines per set
Assume: cache block size 8 bytes

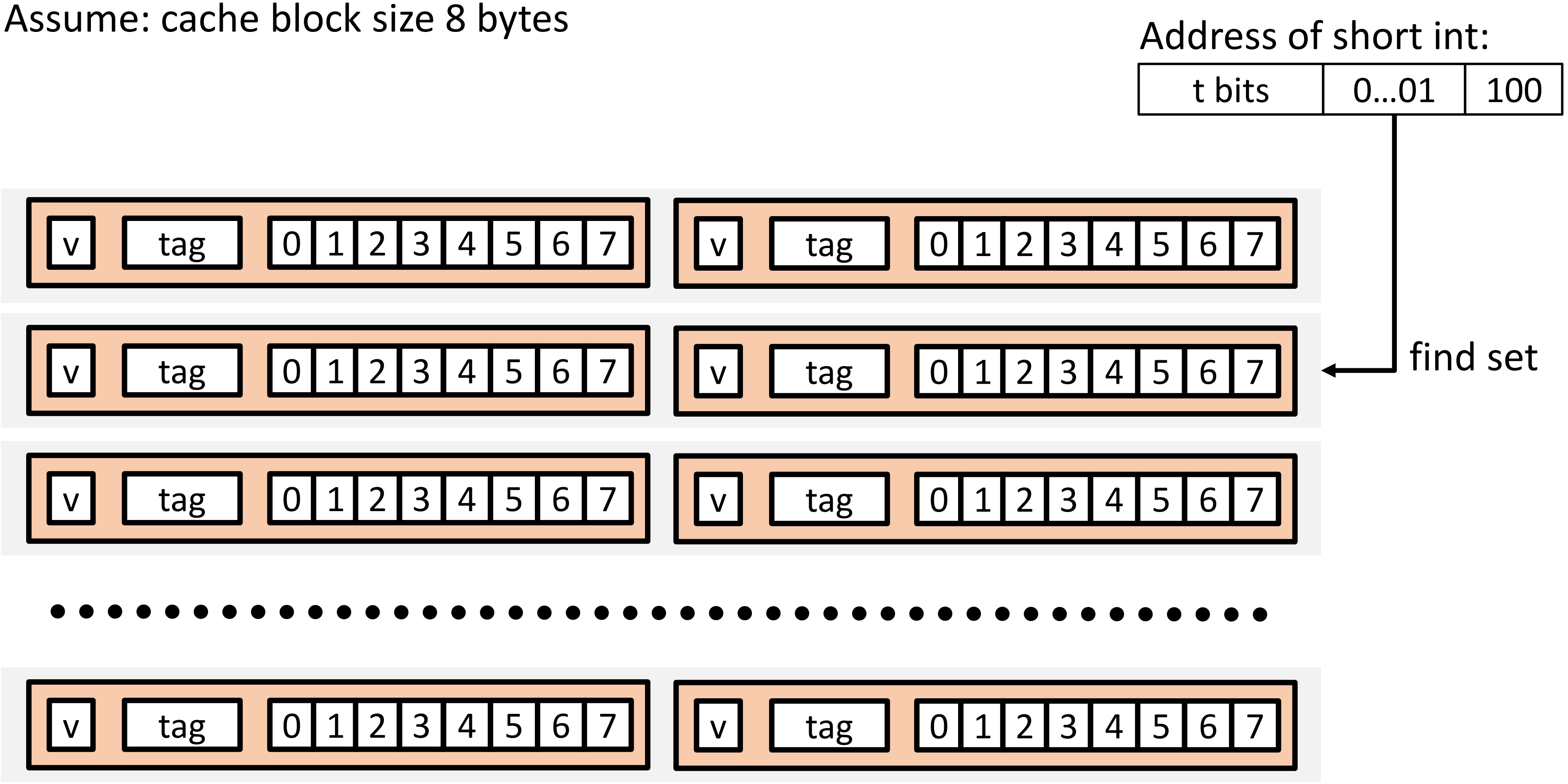
Address of short int:

t bits	0…01	100



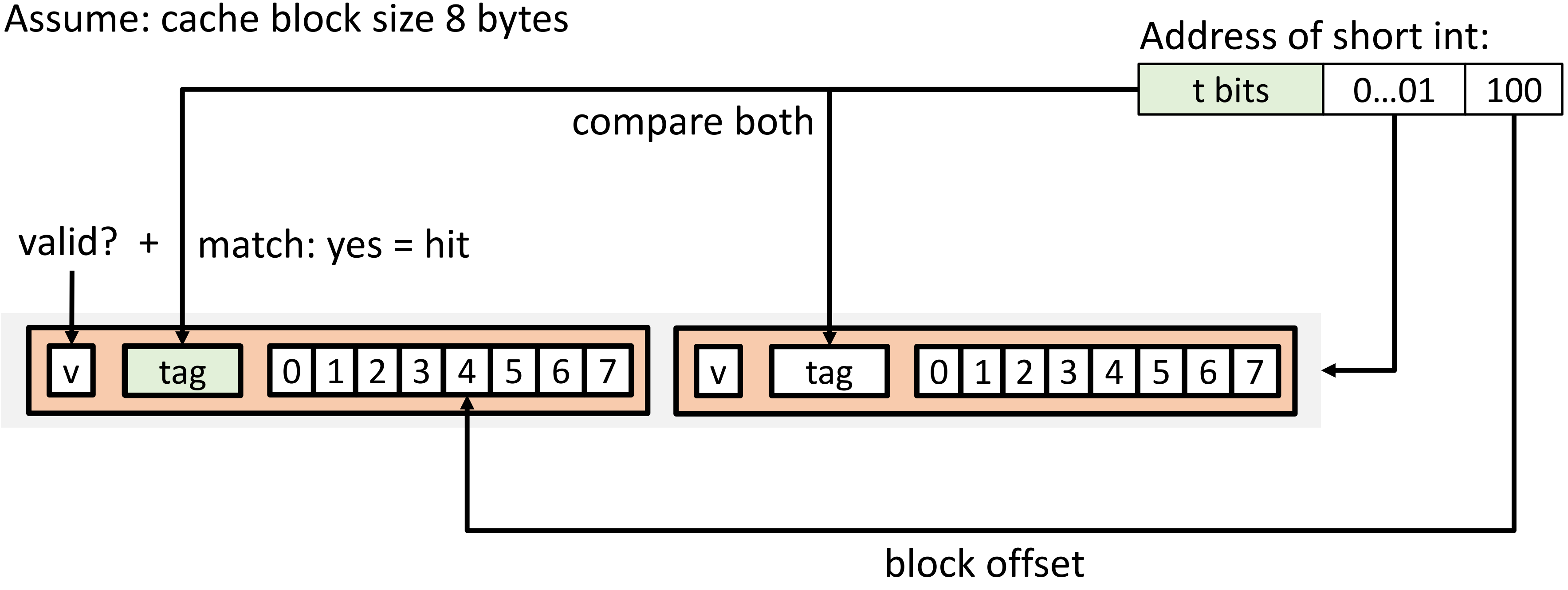
E-way Set Associative Cache (Here: E = 2)

E = 2: Two lines per set
Assume: cache block size 8 bytes



E-way Set Associative Cache (Here: E = 2)

E = 2: Two lines per set
Assume: cache block size 8 bytes



2-Way Set Associative Cache Simulation

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

M=16 byte addresses, B=2 bytes/block,
S=2 sets, E=2 blocks/set

Address trace (reads, one byte per read):

0	$[00\underline{0}0_2]$,
1	$[00\underline{0}1_2]$,
7	$[01\underline{1}1_2]$,
8	$[10\underline{0}0_2]$,
0	$[00\underline{0}0_2]$

	v	Tag	Block
Set 0	0		
	0		
Set 1	0		
	0		

2-Way Set Associative Cache Simulation

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

M=16 byte addresses, B=2 bytes/block,
S=2 sets, E=2 blocks/set

Address trace (reads, one byte per read):

0	$[00\underline{0}0_2]$,	miss
1	$[00\underline{0}1_2]$,	
7	$[01\underline{1}1_2]$,	
8	$[10\underline{0}0_2]$,	
0	$[00\underline{0}0_2]$	

	v	Tag	Block
Set 0	1	00	M[0-1]
	0		
Set 1	0		
	0		

2-Way Set Associative Cache Simulation

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

M=16 byte addresses, B=2 bytes/block,
S=2 sets, E=2 blocks/set

Address trace (reads, one byte per read):

0	$[00\underline{0}0_2]$,	miss
1	$[00\underline{0}1_2]$,	hit
7	$[01\underline{1}1_2]$,	
8	$[10\underline{0}0_2]$,	
0	$[00\underline{0}0_2]$	

	v	Tag	Block
Set 0	1	00	M[0-1]
	0		
Set 1	0		
	0		

2-Way Set Associative Cache Simulation

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

M=16 byte addresses, B=2 bytes/block,
S=2 sets, E=2 blocks/set

Address trace (reads, one byte per read):

0	$[00\underline{0}0_2]$,	miss
1	$[00\underline{0}1_2]$,	hit
7	$[01\underline{1}1_2]$,	miss
8	$[10\underline{0}0_2]$,	
0	$[00\underline{0}0_2]$	

	v	Tag	Block
Set 0	1	00	M[0-1]
	0		
Set 1	1	01	M[6-7]
	0		

2-Way Set Associative Cache Simulation

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

M=16 byte addresses, B=2 bytes/block,
S=2 sets, E=2 blocks/set

Address trace (reads, one byte per read):

0	$[00\underline{0}0_2]$,	miss
1	$[00\underline{0}1_2]$,	hit
7	$[01\underline{1}1_2]$,	miss
8	$[10\underline{0}0_2]$,	miss
0	$[00\underline{0}0_2]$	

	v	Tag	Block
Set 0	1	00	M[0-1]
	1	10	M[8-9]
Set 1	1	01	M[6-7]
	0		

2-Way Set Associative Cache Simulation

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

M=16 byte addresses, B=2 bytes/block,
S=2 sets, E=2 blocks/set

Address trace (reads, one byte per read):

0	$[00\underline{0}0_2]$,	miss
1	$[00\underline{0}1_2]$,	hit
7	$[01\underline{1}1_2]$,	miss
8	$[10\underline{0}0_2]$,	miss
0	$[00\underline{0}0_2]$	hit

	v	Tag	Block
Set 0	1	00	M[0-1]
	1	10	M[8-9]
Set 1	1	01	M[6-7]
	0		

Evaluating Cache Performance

Miss rate

- Fraction of memory references not found in cache (misses/access)

Hit time

- Time to deliver a line in the cache to the processor, including the time to determine whether the line is in the cache

Miss penalty

- Additional time required because of a miss

Average Memory Access Time

- AMAT = time_{hit} + $\text{prob}_{\text{miss}}$ * $\text{penalty}_{\text{miss}}$
- Let us compare performance of 99% and 97% hit rates
  - Consider cache hit time of 1 cycle
  - Miss penalty of 100 cycles
- $\text{AMAT}_{99\%}$ = ?
- $\text{AMAT}_{97\%}$ = ?

Average Memory Access Time

- AMAT = time_{hit} + $\text{prob}_{\text{miss}}$ * $\text{penalty}_{\text{miss}}$
- Let us compare performance of 99% hit rate with 97%
  - Consider cache hit time of 1 cycle
  - Miss penalty of 100 cycles
- $\text{AMAT}_{99\%}$ = 1 + 0.01*100 = 2 cycles
- $\text{AMAT}_{97\%}$ = 1 + 0.03*100 = 4 cycles
- For multilevel cache
  - AMAT_{i} (at level i) = $\text{time}_{\text{hit}_i}$ + $\text{prob}_{\text{miss}_i}$ * AMAT_{i-1}

Write Cache-Friendly Code

Slides adapted from Bryant and O’Hallaron (CS 15-213 @ CMU)

Is this function cache friendly?

```
int sumvec(int v[N]) {
  int sum=0;
  for (int i = 0; i < N; i++) {
    sum += v[i];
  }
  return sum;
}
```

Suppose v is block-aligned, words are 4 bytes, cache blocks are 4 words, and the cache is initially empty.

What can you say about locality of variables i, sum, and elements of v?

Is this function cache friendly?

```
int sumvec(int v[N]) {
  int sum=0;
  for (int i = 0; i < N; i++) {
    sum += v[i];
  }
  return sum;
}
```

ADDR	0	4	8	12	16	20
Contents	v_0	v_1	v_2	v_3	v_4	V_5
Iteration	0	1	2	3	4	5

Compare the two programs

```
for (int i = 0; i < n; i++) {
  z[i] = x[i] – y[i];
  z[i] = z[i] * z[i];
}
```

```
for (int i = 0; i < n; i++) {
  z[i] = x[i] – y[i];
}
for (int i = 0; i < n; i++) {
  z[i] = z[i] * z[i];
}
```

Which version is more efficient
if we have large arrays?

Layout of C Arrays in Memory

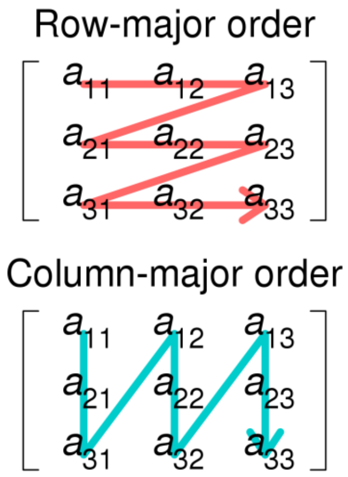
- C arrays allocated in row-major order
- Stepping through columns in one row
  - Exploits spatial locality if block size (B) > 4 bytes

```
int A[N][N];

for (i = 0; i < N; i++)
   sum += A[0][i];
```

- Stepping through rows in one column
  - Accesses distant elements, no spatial locality!

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


Row-major order
a_{11} a_{12} a_{13}
a_{21} a_{22} a_{23}
a_{31} a_{32} a_{33}
Column-major order
a_{11} a_{12} a_{13}
a_{21} a_{22} a_{23}
a_{31} a_{32} a_{33}

https://en.wikipedia.org/wiki/Row-_and_column-major_order

Zeroing an Array

```
for (int j = 0; j < n; j++)
  for (int i = 0; i < n; i++)
    Z[i][j] = 0;
```

```
for (int i = 0; i < n; i++)
  for (int j = 0; j < n; j++)
    Z[i][j] = 0;
```

Which version is more efficient if the dimensions are large?

Data Locality

Parallelism and data locality go hand-in-hand

- Repeated references to memory locations or variables are good – temporal locality
- Stride-1 reference patterns are good – spatial locality

Always focus on optimizing the common case

Compare Access Strides

```
int sumarrayrows(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 sumarraycols(int a[M][N]) {
  int i, j, sum=0;
  for (j = 0; j < M; j++)
    for (i = 0; i < N; i++)
      sum += A[i][j];
  return sum;
}
```

Compare Access Strides

```
int sumarrayrows(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 sumarraycols(int a[M][N]) {
  int i, j, sum=0;
  for (j = 0; j < M; j++)
    for (i = 0; i < N; i++)
      sum += A[i][j];
  return sum;
}
```

What are the miss rates per iteration if the array a (i) fits in cache and (ii) does not fit in cache?

Compare Access Strides

```
int sumarrayrows(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 sumarraycols(int a[M][N]) {
  int i, j, sum=0;
  for (j = 0; j < M; j++)
    for (i = 0; i < N; i++)
      sum += A[i][j];
  return sum;
}
```

4X slower

Miss Rate Analysis for Matrix Multiply

- Matrix-Vector Multiply and Matrix-Matrix Multiply are important kernels
  - Heavily used in computational science applications

```
/* 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;
  }
}
```

Miss Rate Analysis for Matrix Multiply

- Multiply NxN matrices with O(N^3) operations
- N reads per source element
- N values summed per destination
  - `sum` can be stored in a register
- $3N^2$ memory locations
- Algorithm is **computation-bound**
  - Memory accesses should not constitute a bottleneck

```
/* 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;
  }
}
```

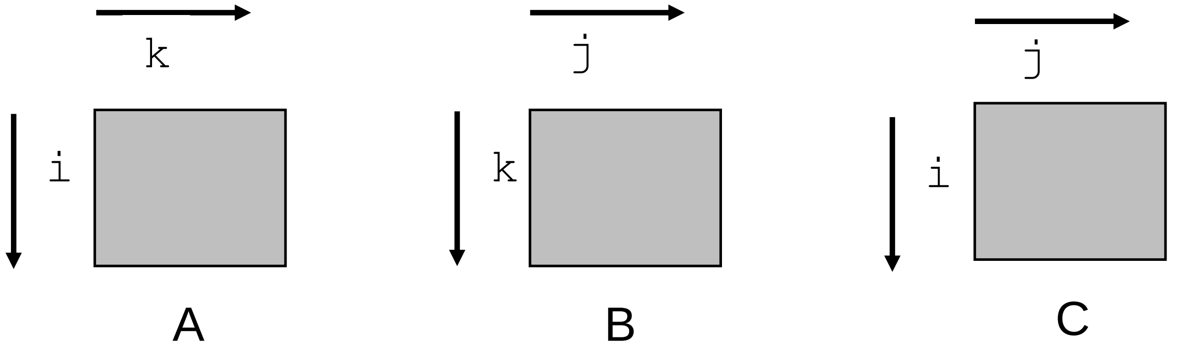
Cache Model

- Assumptions:
    - Only consider cold and capacity misses, ignore conflict misses
    - Large cache model: only cold misses
    - Small cache model: both cold and capacity misses

    - Line size = 32B (big enough for four 64-bit words)
    - Matrix dimension (N) is very large
        - Approximate $\frac{1}{N}$ as 0.0
    - Cache is not even big enough to hold multiple rows

Miss Rate Analysis for Matrix Multiply

- Analysis Method:
  - Look at access pattern of inner loop

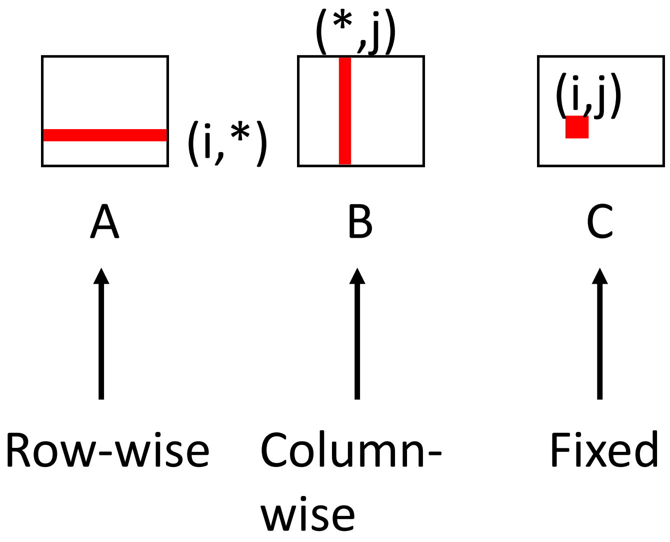


Matrix Multiplication (ijk)

```
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;
  }
}
```

two stores, zero loads

Inner loop:



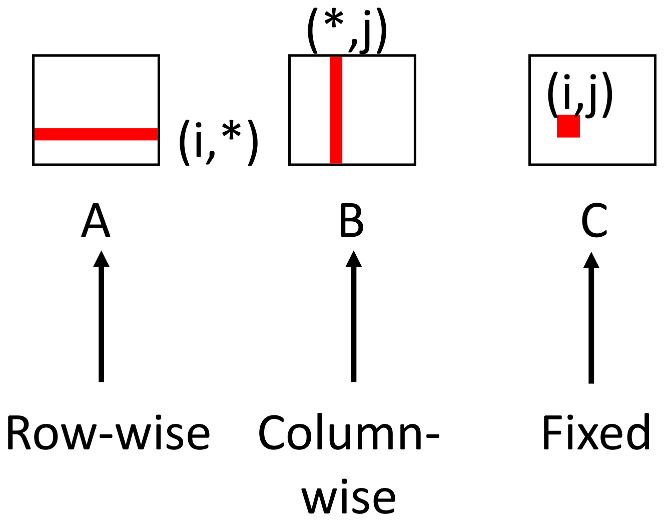
Misses per inner loop iteration:

A	B	C
0.25	1.0	0.0

Matrix Multiplication (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:

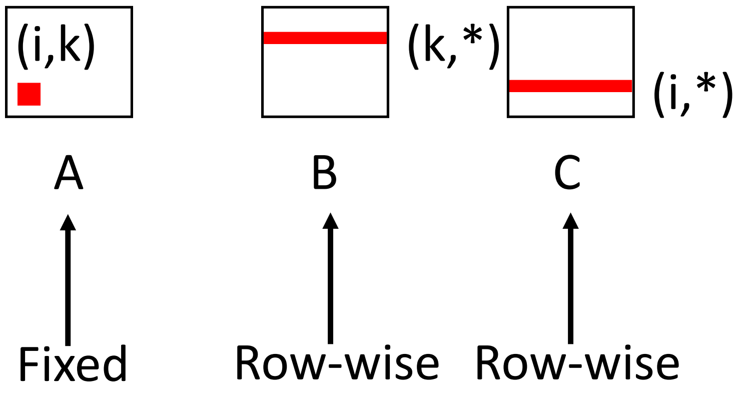
A	B	C
0.25	1.0	0.0

Matrix Multiplication (kij)

```
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];
  }
}
```

two stores,
one load

Inner loop:



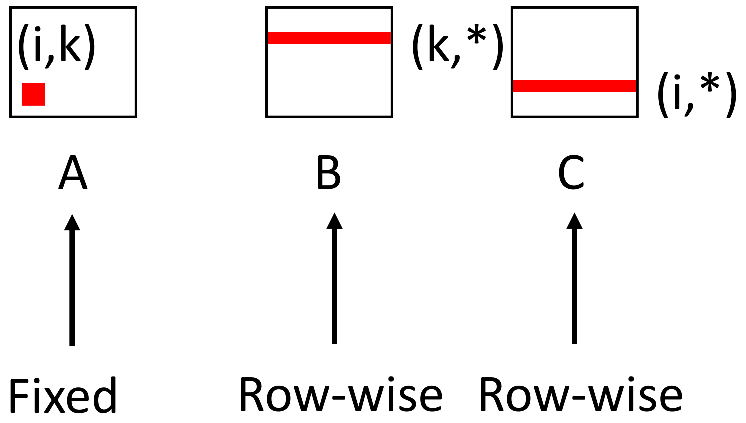
Misses per inner loop iteration:

A	B	C
0.0	0.25	0.25

Matrix Multiplication (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:



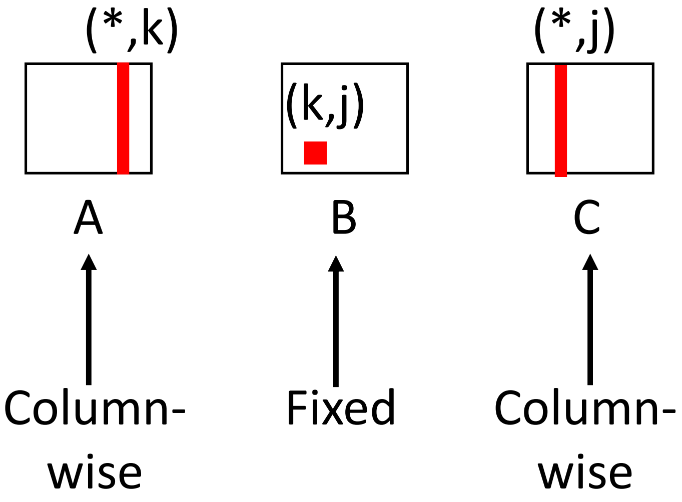
Misses per inner loop iteration:

A	B	C
0.0	0.25	0.25

Matrix Multiplication (jki)

```
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:



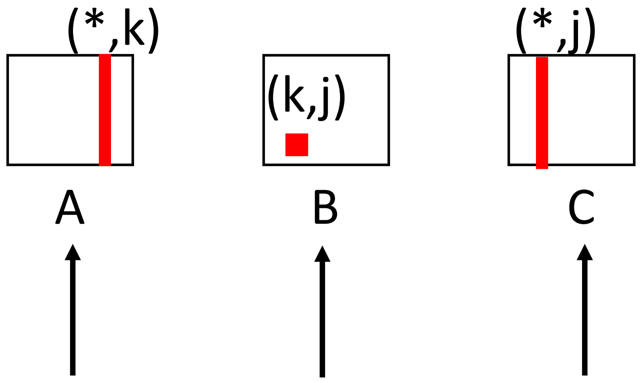
Misses per inner loop iteration:

A	B	C
1.0	0.0	1.0

Matrix Multiplication (kji)

```
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:



Column-wise

Fixed

Column-wise

Misses per inner loop iteration:

A	B	C
1.0	0.0	1.0

Summary of Matrix Multiplication

```
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

```
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):

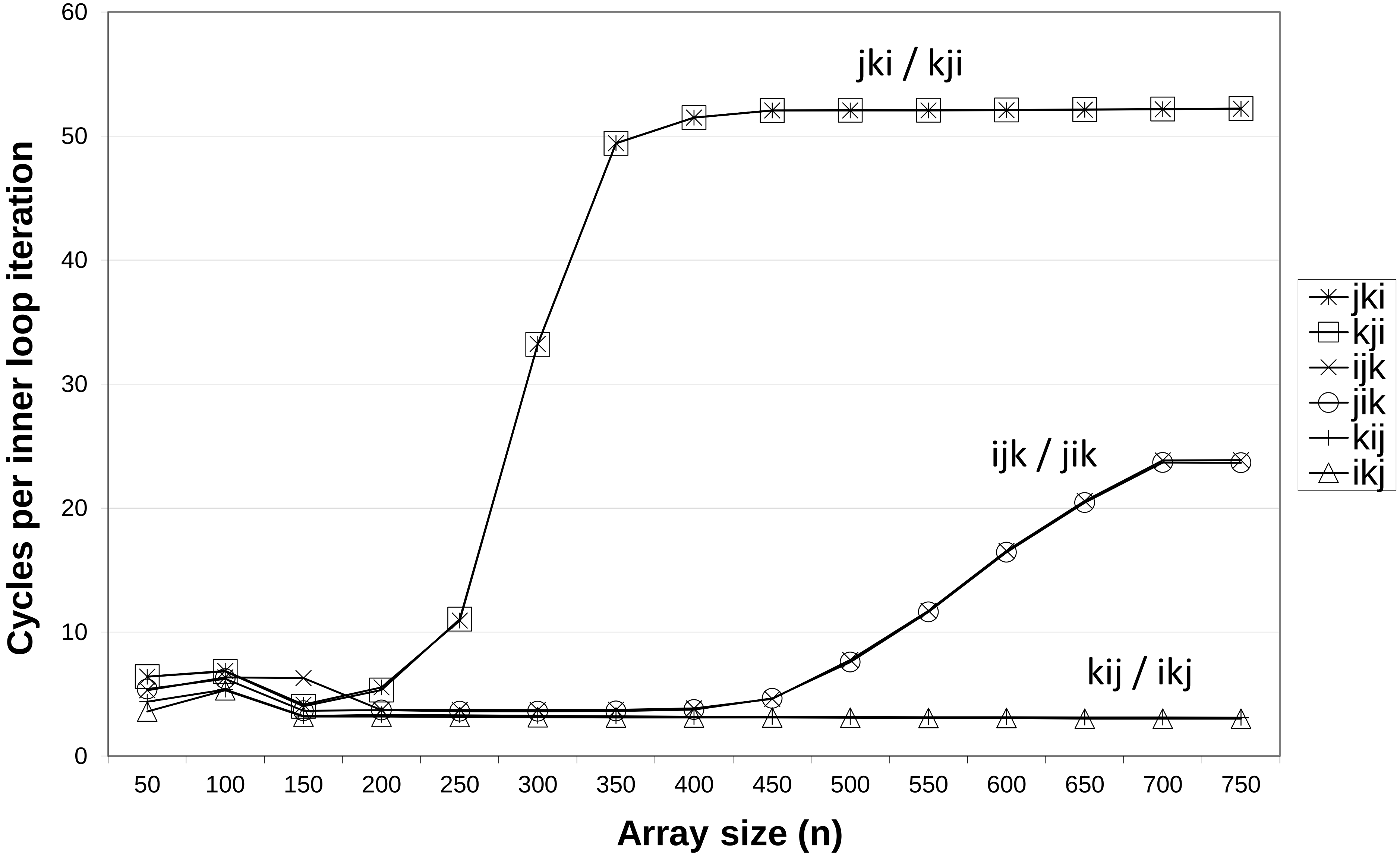
- 2 loads, 1 store
- misses/iter = 0.5

```
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



Total Cache Misses (ijk)

Matrices are very large compared to cache size

```
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;
  }
}
```

	A	**B**	**C**
I	n	n	n
J	n	n	n/BL
K	n/BL	n	1
	n^3/BL	**n^3**	**n^2/BL**

Total Cache Misses (jki)

Matrices are very large compared to cache size

```
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;
  }
}
```

	A	**B**	**C**
I	n	1	n
J	n	n	n
K	n	n	n
	n^3	**n^2**	**n^3**

Cache Miss Analysis for MVM

```
for (i = 0; i < M; i++)
    for (j = 0; j < N; j++)
      y[i] += A[i][j]*x[j];
  return sum;
}
```

$$\begin{bmatrix} 1 & 0 & 2 & 0 \\ 0 & 3 & 0 & 4 \\ 0 & 0 & 5 & 0 \\ 6 & 0 & 0 & 7 \end{bmatrix} \cdot \begin{bmatrix} 2 \\ 5 \\ 1 \\ 8 \end{bmatrix} = \begin{bmatrix} 4 \\ 47 \\ 5 \\ 68 \end{bmatrix}$$

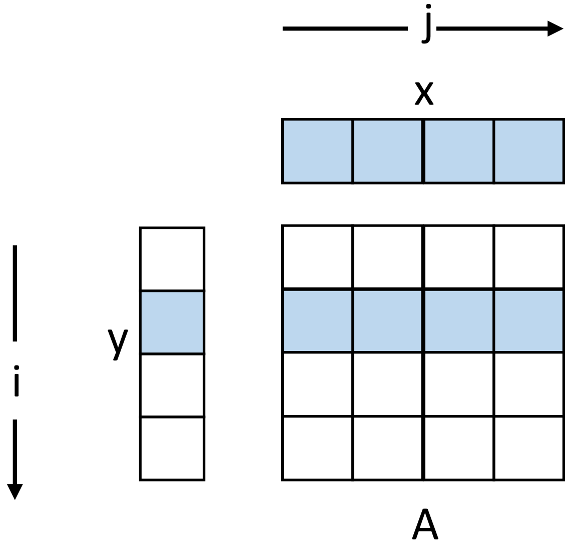
Cache Miss Analysis for MVM

- Number of memory locations: $N^2 + 2N$
- Number of operations: $O(N^2)$
- MVM is limited by memory bandwidth

```
for (i = 0; i < M; i++)
    for (j = 0; j < N; j++)
      y[i] += A[i][j]*x[j];
  return sum;
}
```

MVM (ij)

```
for (i = 0; i < M; i++)
    for (j = 0; j < N; j++)
      y[i] += A[i][j]*x[j];
  return sum;
}
```



Large Cache Model

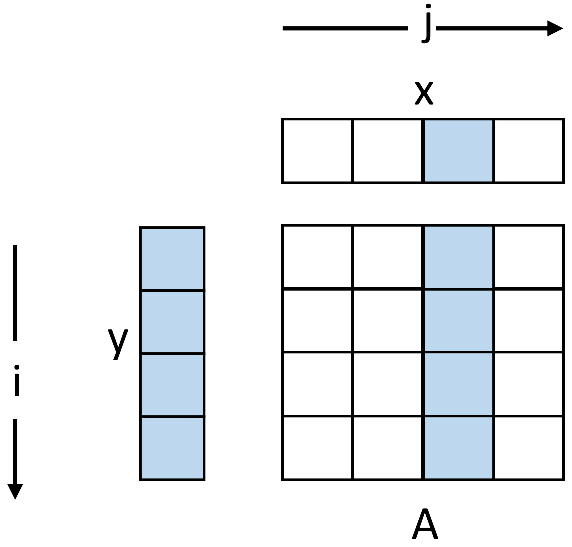
- Misses
  - A: N^2/B
  - X: N/B
  - Y: N/B
  - Total: $N^2/B + 2N/B$

Small Cache Model

- Misses
  - A: N^2/B
  - X: $N/B * N$
  - Y: N/B
  - Total: $2N^2/B + N/B$

MVM (ji)

```
for (j = 0; j < M; j++)
    for (i = 0; i < N; i++)
      y[i] += A[i][j]*x[j];
  return sum;
}
```



Large Cache Model

- Misses
  - A: N^2/B
  - X: N/B
  - Y: N/B
  - Total: $N^2/B + 2N/B$

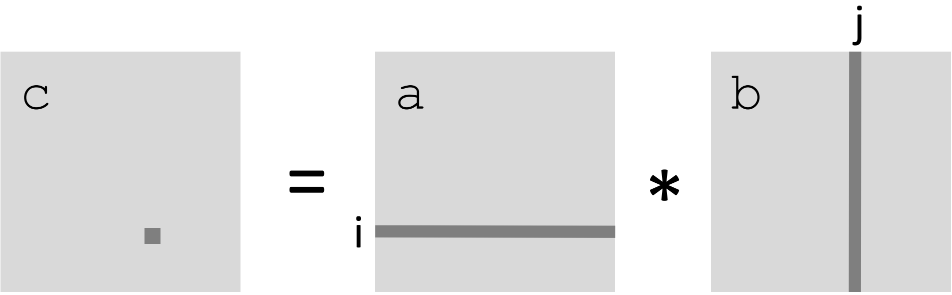
Small Cache Model

- Misses
  - A: N^2
  - X: N/B
  - Y: N^2/B
  - Total: $N^2 + N^2/B + N/B$

Using Blocking to Improve Temporal Locality

Example: Matrix Multiplication

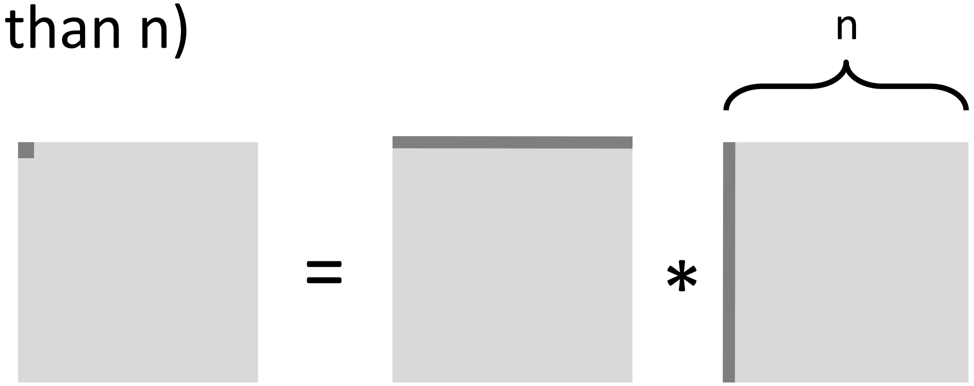
```
/* Multiply n x n matrices a and b  */
void mmm(double *a, double *b, double *c, int n) {
    int i, j, k;
    for (i = 0; i < n; i++)
        for (j = 0; j < n; j++)
            for (k = 0; k < n; k++)
                c[i*n+j] += a[i*n + k]*b[k*n + j];
}
```



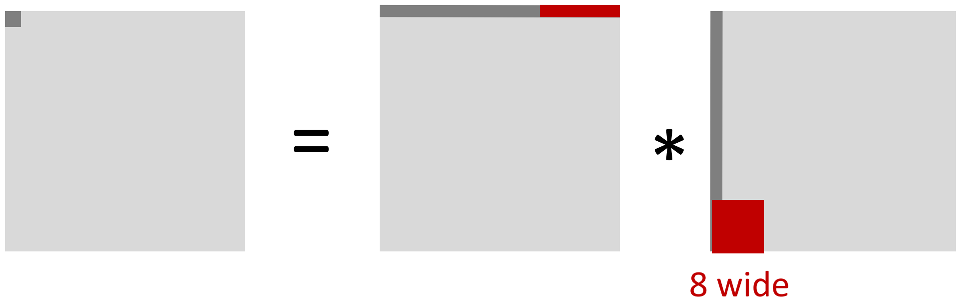
Cache Miss Analysis

- Assume:
  - Matrix elements are doubles
  - Cache block = 8 doubles
  - Cache size << n (much smaller than n)

- First iteration:
  - $\frac{n}{8} + n = \frac{9n}{8}$ misses



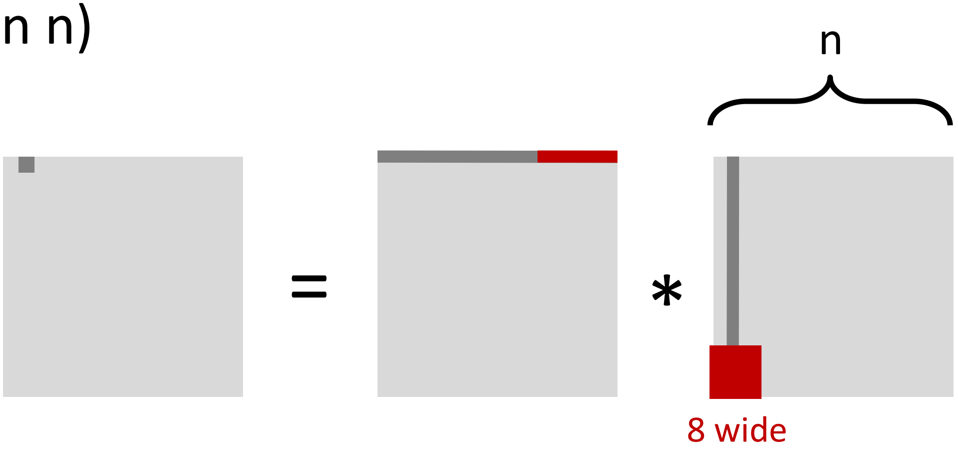
  - Afterwards in cache:
    (schematic)



Cache Miss Analysis

- Assume:
    - Matrix elements are doubles
    - Cache block = 8 doubles
    - Cache size << n (much smaller than n)

- Second iteration:
    - $\frac{n}{8} + n = \frac{9n}{8}$ misses



- Total misses:
    - $\frac{9n}{8} * n^2 = \frac{9}{8}n^3$

Cache Blocking

- Improve data reuse by chunking the data in to smaller blocks
  - The block is supposed to fit in the cache

```
for (i = 0; i < N; i++) {
  …
}
```

```
for (j = 0; j < N; j +=B) {
  for (i = j; i < min(N, j+B); j++) {
    …
  }
}
```

```
for (body1 = 0; body1 < NBODIES; body1 ++) {
   for (body2=0; body2 < NBODIES; body2++) {
     OUT[body1] += compute(body1, body2);
   }
}
```

```
for (body2 = 0; body2 < NBODIES; body2 += BLOCK) {
   for (body1=0; body1 < NBODIES; body1 ++) {
      for (body22=0; body22 < BLOCK; body22 ++) {
         OUT[body1] += compute(body1, body2 +
body22);
      }
   }
}
```

https://software.intel.com/en-us/articles/cache-blocking-techniques

MVM with 2x2 Blocking

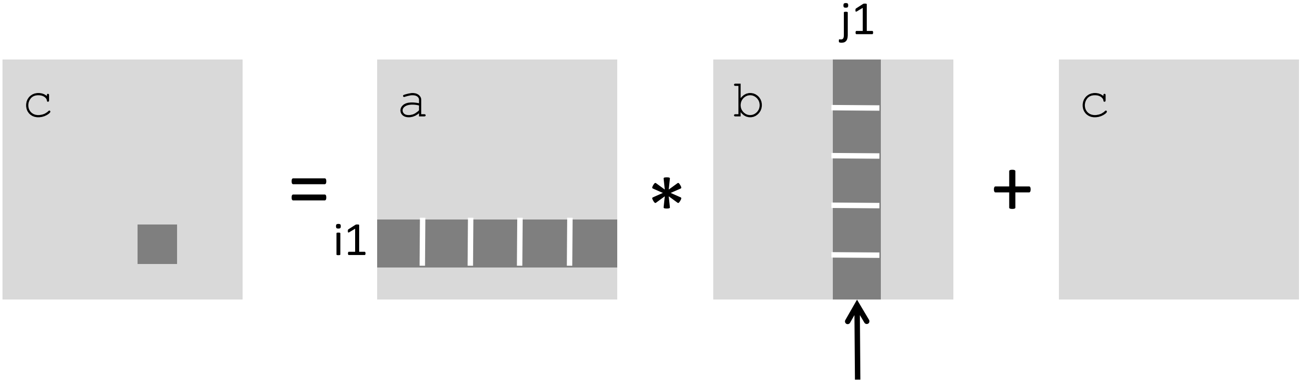
```
int i, j, a[100][100], b[100], c[100];
int n = 100;
for (i = 0; i < n; i++) {
  c[i] = 0;
  for (j = 0; j < n; j++) {
    c[i] = c[i] + a[i][j] * b[j];
  }
}
```

```
int i, j, x, y, a[100][100], b[100],
c[100];
int n = 100;
for (i = 0; i < n; i += 2) {
  c[i] = 0;
  c[i + 1] = 0;
  for (j = 0; j < n; j += 2) {
    for (x = i; x < min(i + 2, n); x++) {
      for (y = j; y < min(j + 2, n); y++) {
        c[x] = c[x] + a[x][y] * b[y];
      }
    }
  }
}
```

https://en.wikipedia.org/wiki/Loop_nest_optimization

Blocked Matrix Multiplication

```
/* Multiply n x n matrices a and b  */
void mmm(double *a, double *b, double *c, int n) {
    int i, j, k;
    for (i = 0; i < n; i+=B)
         for (j = 0; j < n; j+=B)
             for (k = 0; k < n; k+=B)
                  /* B x B mini matrix multiplications */
                  for (i1 = i; i1 < i+B; i++)
                      for (j1 = j; j1 < j+B; j++)
                          for (k1 = k; k1 < k+B; k++)
                              c[i1*n+j1] += a[i1*n + k1]*b[k1*n + j1];
}
```



Block size B x B

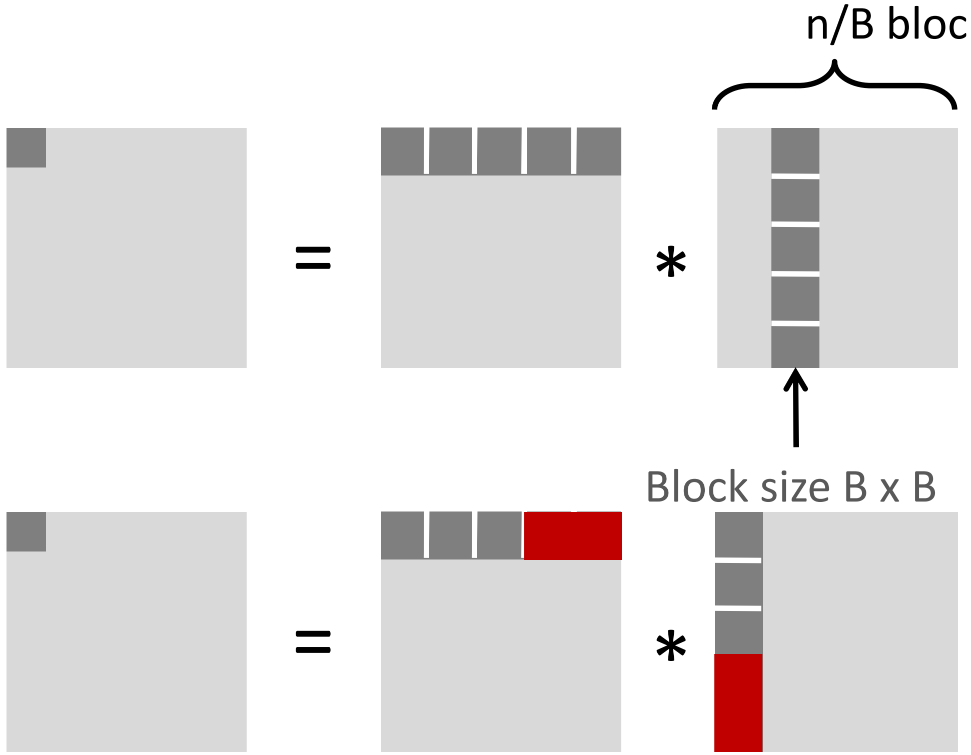
Cache Miss Analysis

- Assume:
  - Cache block = 8 doubles
  - Cache size << n (much smaller than n)
  - Three blocks ■ fit into cache: $3B^2 < C$

- First (block) iteration:
  - $\frac{B^2}{8}$ misses for each block
  - $2 * \frac{n}{B} * \frac{B^2}{8} = \frac{nB}{4}$

  (ignoring matrix C)

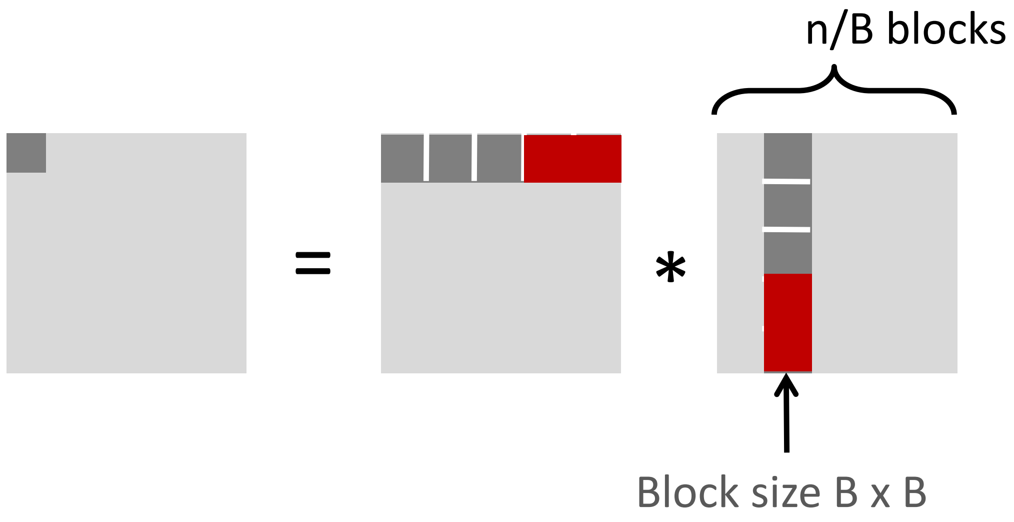
  - Afterwards in cache (schematic)



Cache Miss Analysis

- Assume:
  - Cache block = 8 doubles
  - Cache size << n (much smaller than n)
  - Three blocks ■ fit into cache: $3B^2 < C$

- Second (block) iteration:
  - Same as first iteration
  - $2 * \frac{n}{B} * \frac{B^2}{8} = \frac{nB}{4}$



- Total misses:
  - $\frac{nB}{4} * (\frac{n}{B})^2 = \frac{n^3}{4B}$

Summary

- No blocking: $\frac{9}{8} * n^3$
- Blocking: $\frac{1}{4B} * n^3$

- Find largest possible block size B, but limit $3B^2 < C$!

- Reason for dramatic difference:
  - Matrix multiplication has inherent temporal locality:
    - Input data: $3n^2$, computation $2n^3$
    - Every array elements used O(n) times!
  - But the program has to be written properly

Pointers to Exploit Locality in your Code

Focus on the more frequently executed parts of the code (i.e., common case)

- E.g., inner loops

Maximize spatial locality with low strides (preferably 1)

Maximize temporal locality by reusing the data as much as possible

References

- Keshav Pingali – CS 377P: Programming for Performance, UT Austin.
- P. Sadayappan and A. Sukumaran Rajam – CS 5441: Parallel Computing, Ohio State University.
- R. Bryant and D. O’Hallaron – Cache Memories, CS 15-213, Introduction to Computer Systems., CMU.
- R. Bryant and D. O’Hallaron – Computer Systems: A Programmer’s Perspective.
- A. Aho et al. – Compilers: Principles, Techniques and Tools.