Computation intensive tasks on contemporary hardware and memory

Pavel Píša

Based on A0B36APO course materials by Pavel Píša, Michal Štepanovski
Other used sources and materials: Henessy-Patterson books,
Ulrich Drepper, Paul McKenney, Austin T. Clements, Benny Akesson,
Michal Sojka

Czech Technical University in Prague, Faculty of Electrical Engineering

Lecture motivation

Quick Quiz 1.: Is the result of both code fragments a same?

Quick Quiz 2.: Which of the code fragments is processed faster and why?

A:

```
int matrix[M][N];
int i, j, sum = 0;
…
for(i=0; i<M; i++)
  for(j=0; j<N; j++)
    sum += matrix[i][j];
```

B:

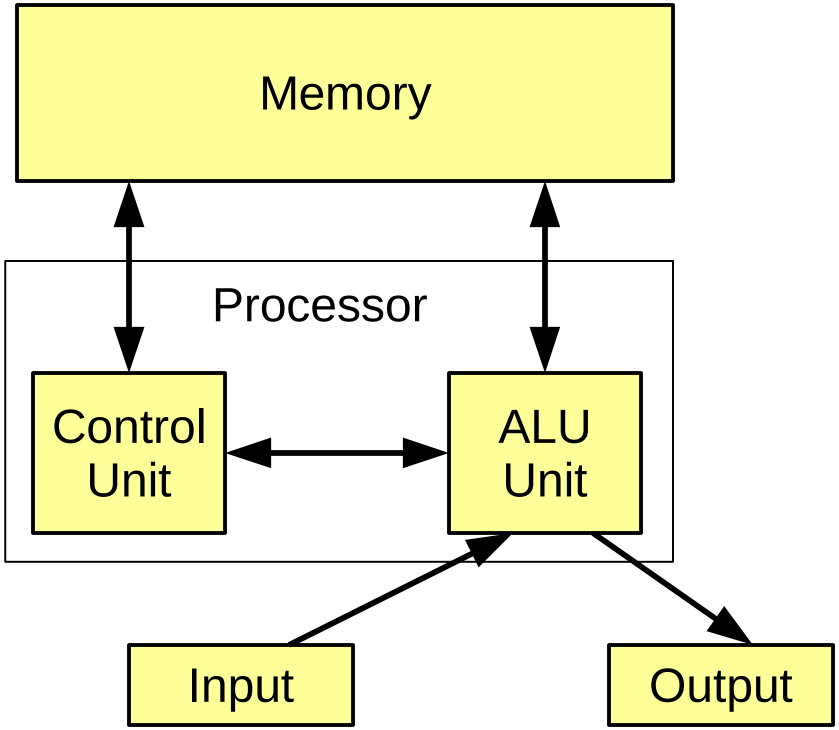
```
int matrix[M][N];
int i, j, sum = 0;
…
for(j=0; j<N; j++)
  for(i=0; i<M; i++)
    sum += matrix[i][j];
```

Is there a rule how to iterate over matrix element efficiently?

John von Neumann, Hungarian physicist

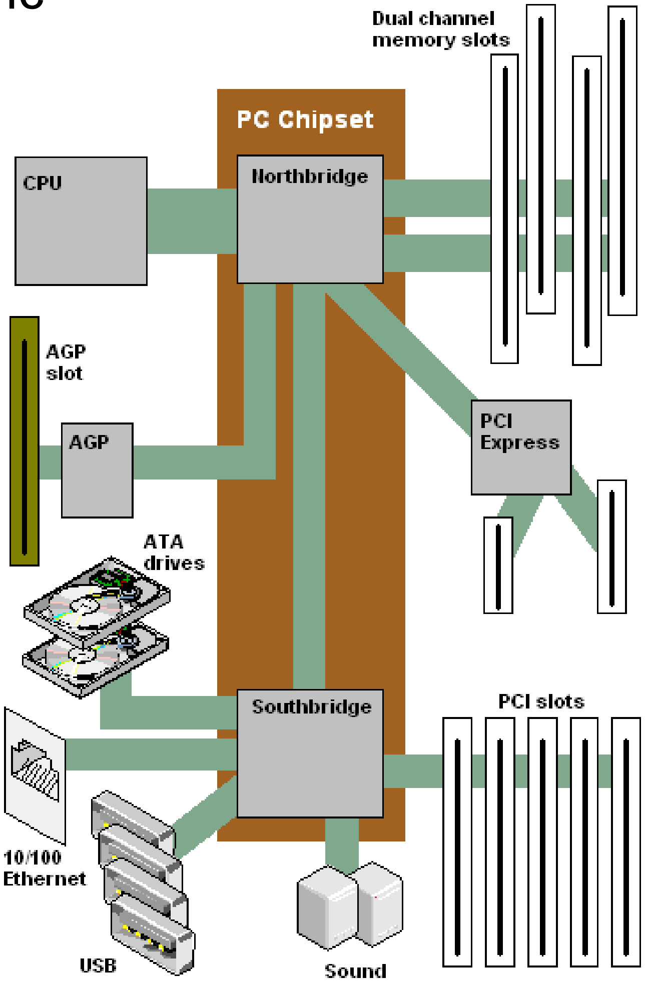
28. 12. 1903 - 8. 2. 1957

von Neumann's computer architecture



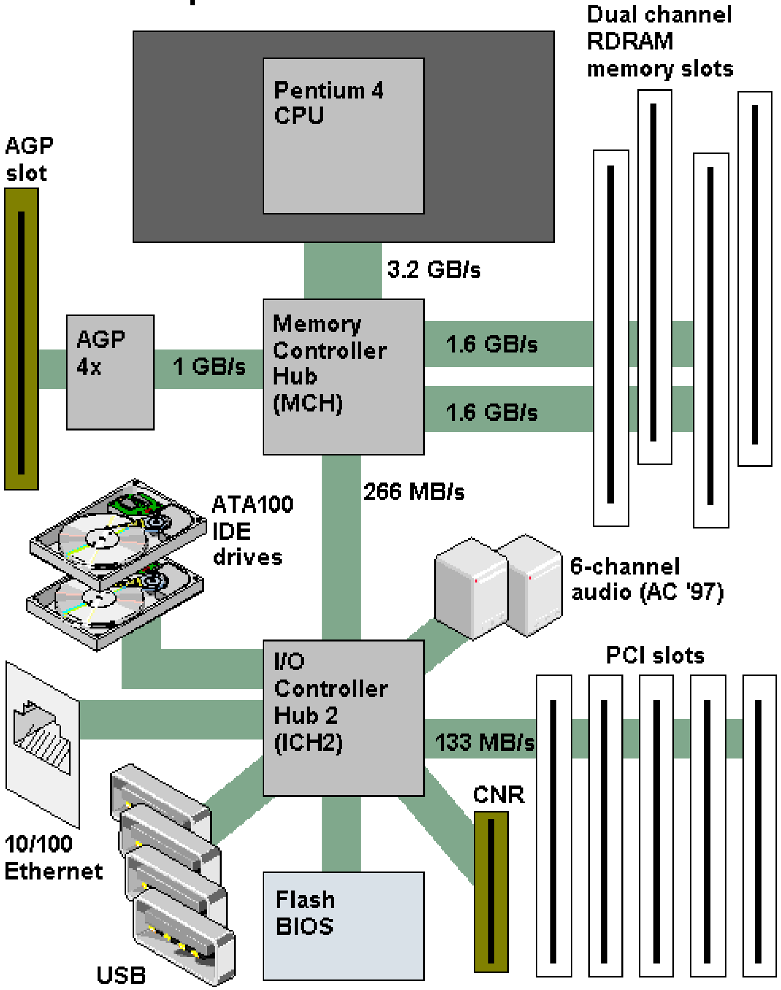
Computer architecture (desktop x86 PC)

generic



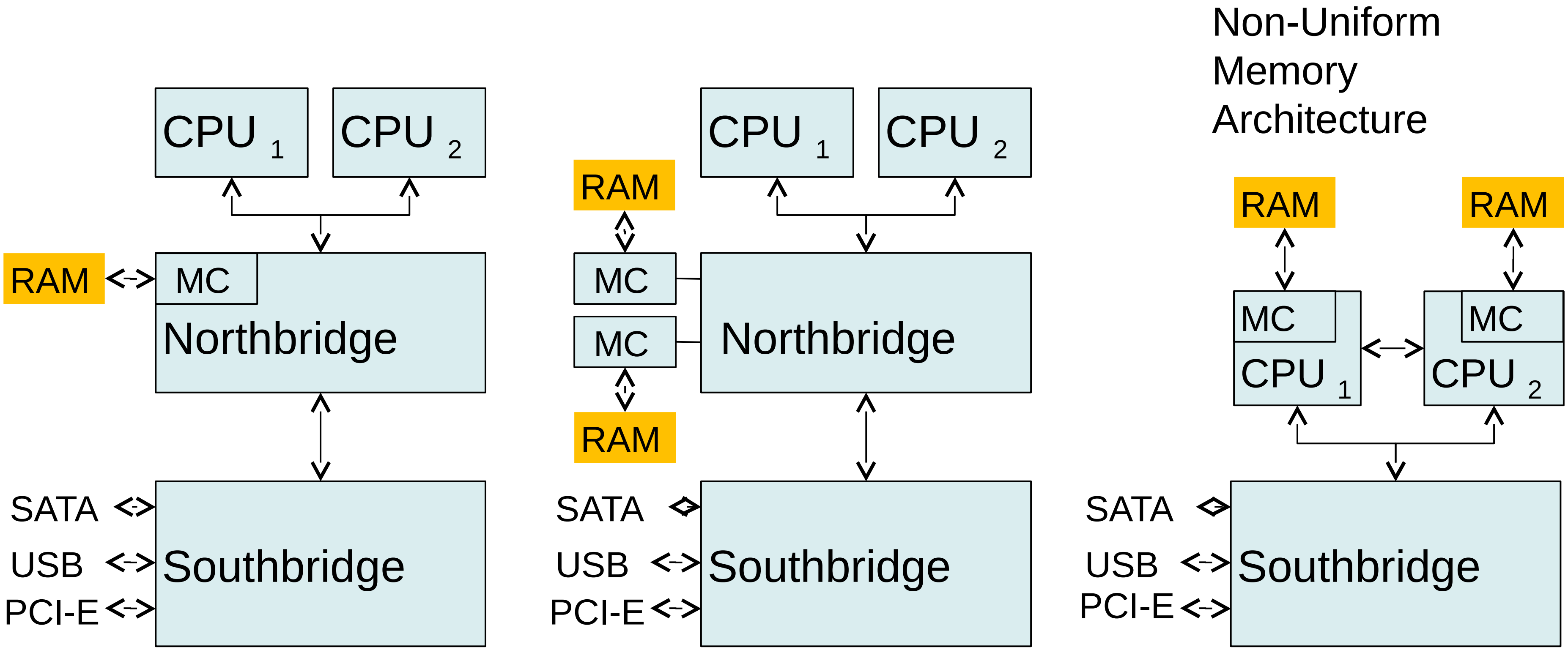
From Computer Desktop Encyclopedia
© 2005 The Computer Language Co. Inc.

example



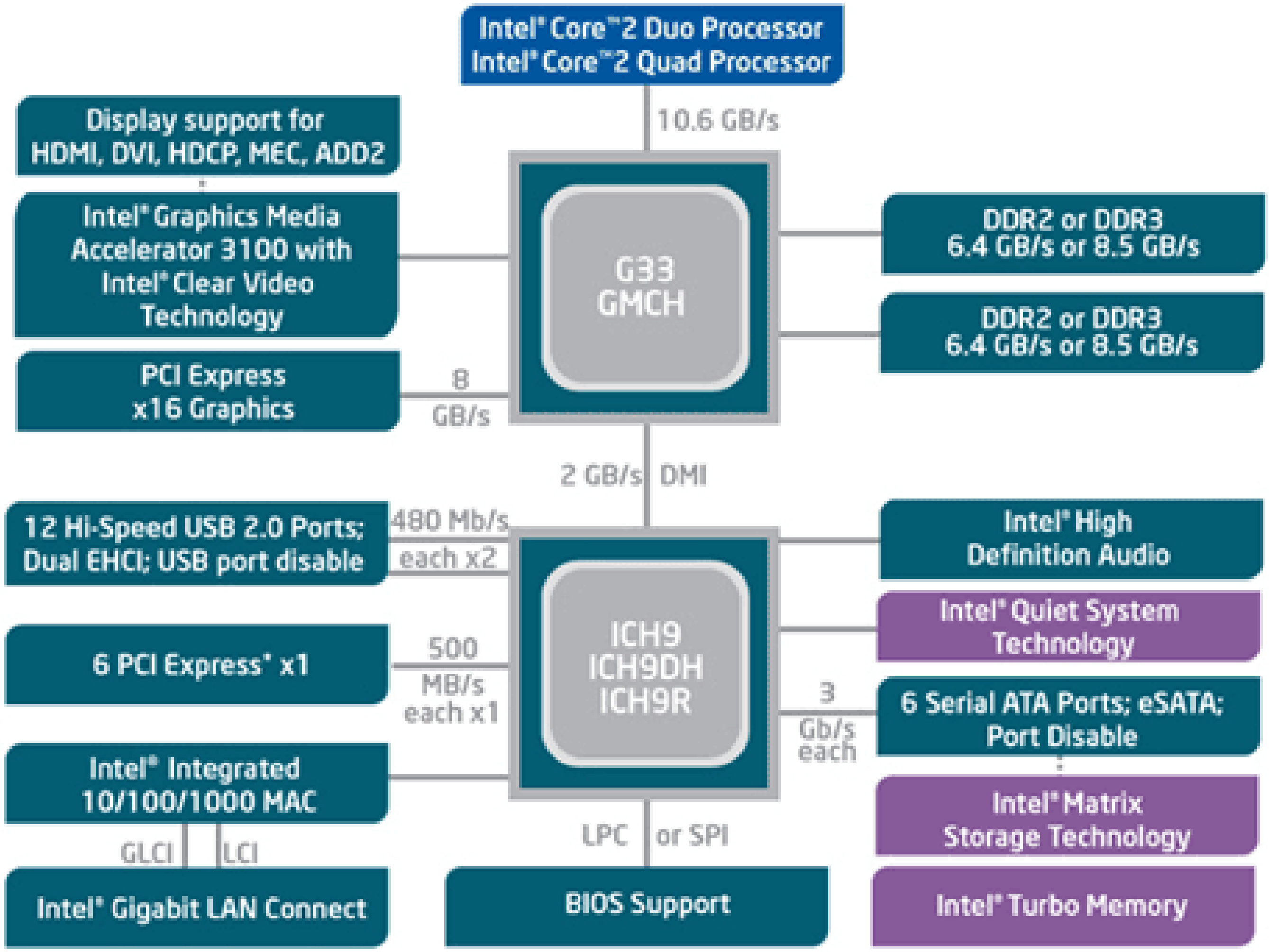
From Computer Desktop Encyclopedia
© 2001 The Computer Language Co. Inc.

From UMA to NUMA development (even in PC segment)



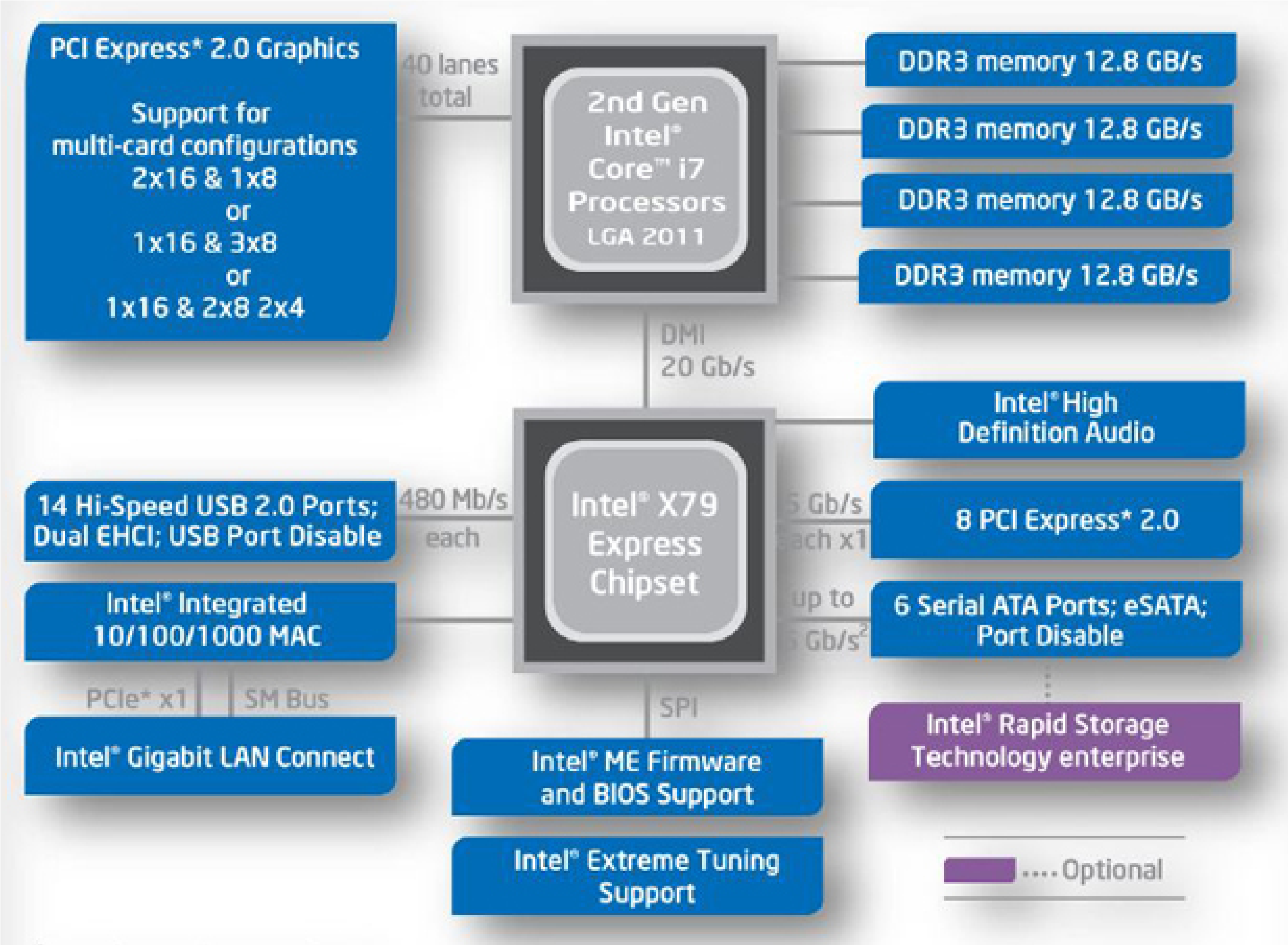
MC - Memory controller – contains circuitry responsible for SDRAM read and writes. It also takes care of refreshing each memory cell every 64 ms.

Intel Core 2 generation



Northbridge became Graphics and Memory Controller Hub (GMCH)

Intel i3/5/7 generation

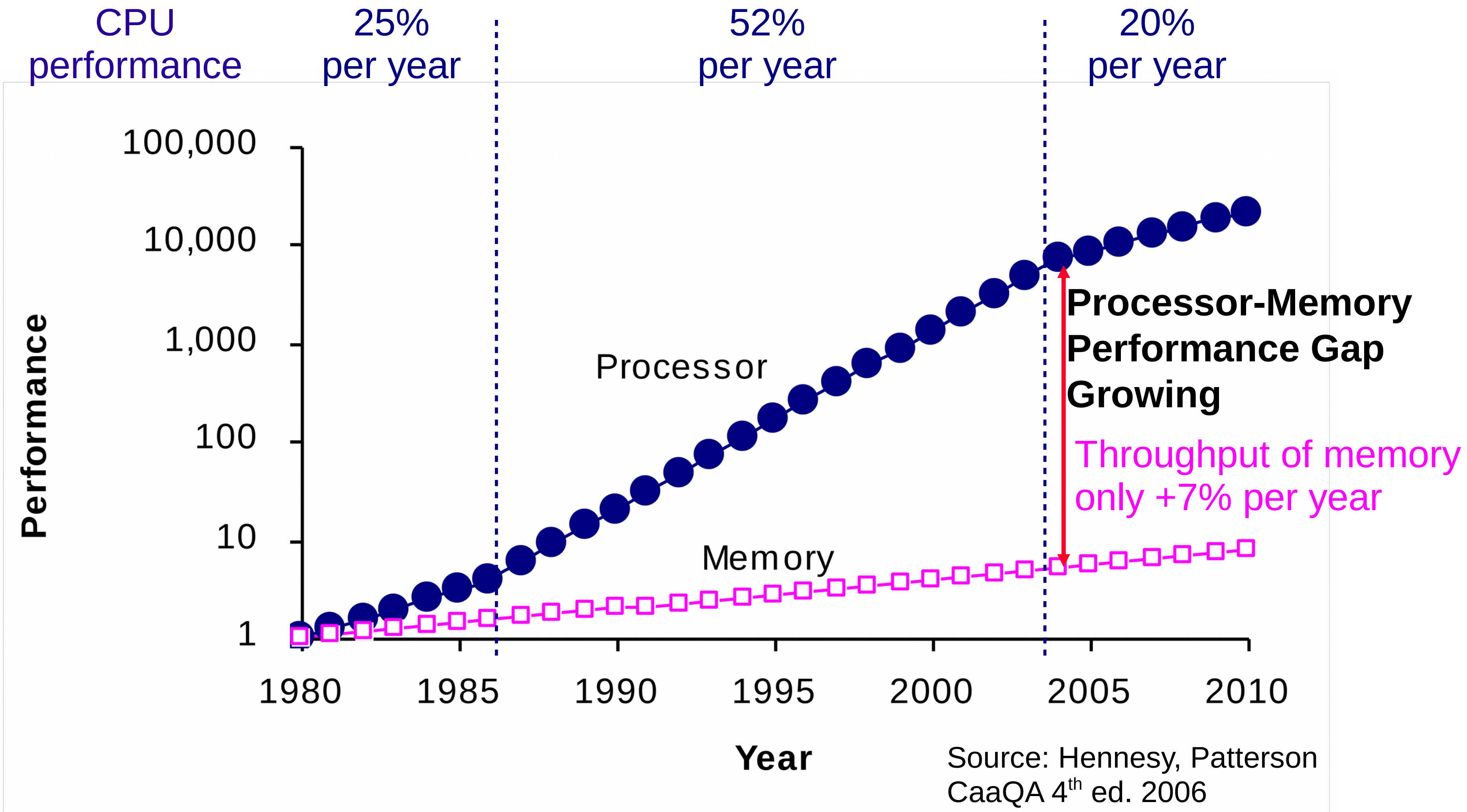

PCI Express* 2.0 Graphics
Support for multi-card configurations 2x16 & 1x8 or 1x16 & 3x8 or 1x16 & 2x8 2x4
40 lanes total
2nd Gen Intel® Core™ i7 Processors LGA 2011
DDR3 memory 12.8 GB/s
DDR3 memory 12.8 GB/s
DDR3 memory 12.8 GB/s
DDR3 memory 12.8 GB/s
DMI 20 Gb/s
Intel® High Definition Audio
14 Hi-Speed USB 2.0 Ports; Dual EHCI; USB Port Disable
480 Mb/s each
Intel® X79 Express Chipset
5 Gb/s each x1
8 PCI Express* 2.0
Intel® Integrated 10/100/1000 MAC
up to 6 Gb/s[2]
6 Serial ATA Ports; eSATA; Port Disable
PCIe* x1
SM Bus
SPI
Intel® Rapid Storage Technology enterprise
Intel® Gigabit LAN Connect
Intel® ME Firmware and BIOS Support
Intel® Extreme Tuning Support
Optional

[1] Theoretical maximum bandwidth

[2] All SATA ports capable of 3 Gb/s. 2 ports capable of 6 Gb/s.

Intel® X79 Express Chipset Block Diagram

Memory and CPU speed – Moore's law



Processor-Memory Performance Gap Growing

Throughput of memory only +7% per year

Source: Hennesy, Patterson
CaaQA 4th ed. 2006

Bubble sort – algorithm example from seminaries

```
int pole[5]={5,3,4,1,2};
int main()
{
    int N = 5,i,j,tmp;
    for(i=0; i<N; i++)
        for(j=0; j<N-1-i; j++)
            if(pole[j+1]<pole[j])
            {
                tmp = pole[j+1];
                pole[j+1] = pole[j];
                pole[j] = tmp;
            }
    return 0;
}
```

What we can consider and expect from our programs?

Think about some typical data access patterns and execution flow.

Memory hierarchy – principle of locality

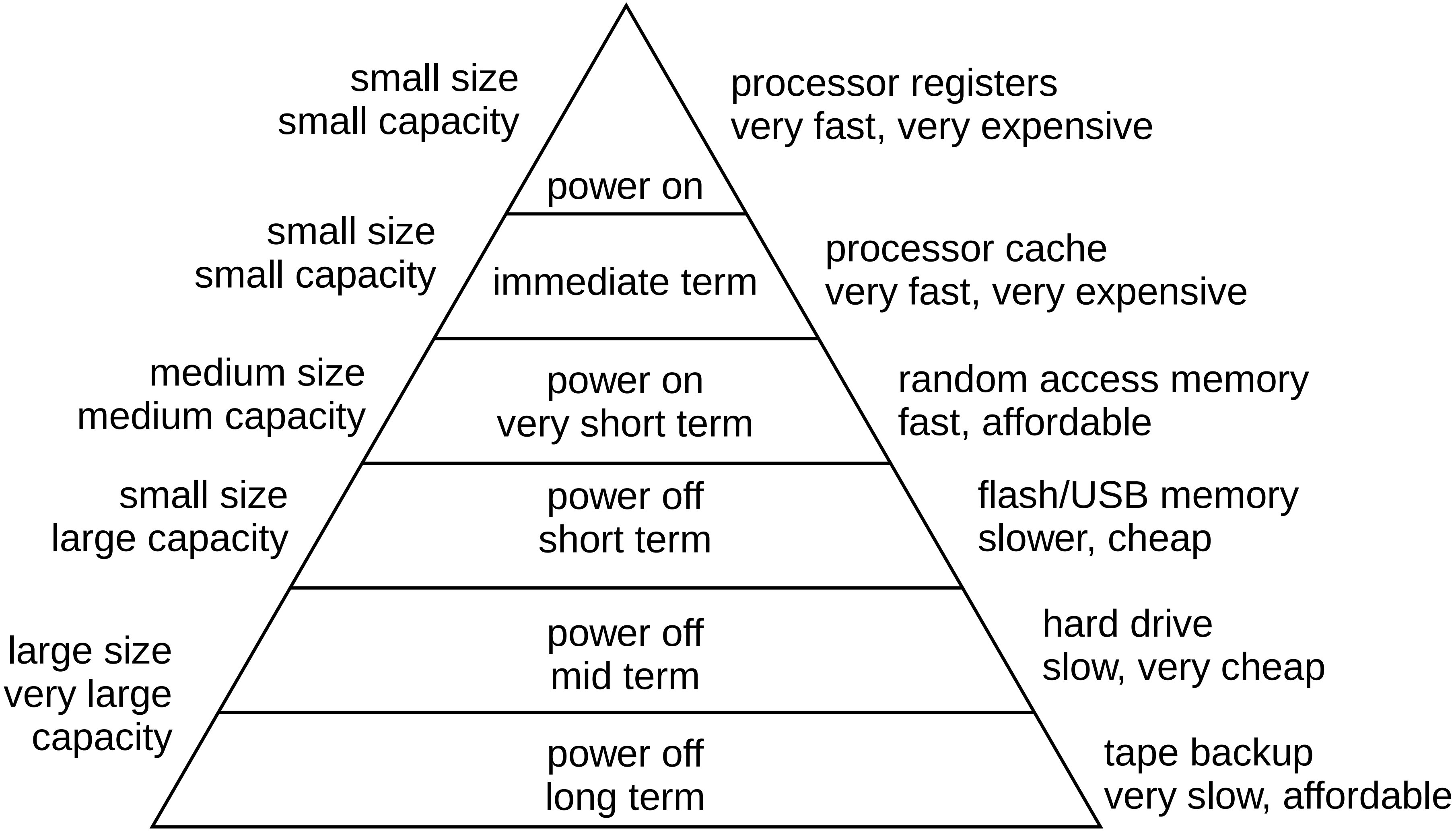
- Programs access a small proportion of their address space at any time
- Temporal locality
  - Items accessed recently are likely to be accessed again soon
  - e.g., instructions in a loop, induction variables
- Spatial locality
  - Items near those accessed recently are likely to be accessed soon
  - E.g., sequential instruction access, array data

Source: Hennesy, Patterson

Memory hierarchy introduced based on locality

- The solution to resolve capacity and speed requirements is to build address space (data storage in general) as hierarchy of different technologies.
- Store input/output data, program code and its runtime data on large and cheaper secondary storage (hard disk)
- Copy recently accessed (and nearby) items from disk to smaller DRAM based main memory (usually under operating system control)
- Copy more recently accessed (and nearby) items from DRAM to smaller SRAM memory (cache) attached to CPU (hidden memory, transactions under HW control), optionally, tightly coupled memory under program's control
- Move currently processed variables to CPU registers (under machine program/compiler control)

Memory hierarchy – speed, capacity, price



small size
small capacity

power on

processor registers
very fast, very expensive

small size
small capacity

immediate term

processor cache
very fast, very expensive

medium size
medium capacity

power on
very short term

random access memory
fast, affordable

small size
large capacity

power off
short term

flash/USB memory
slower, cheap

large size
very large
capacity

power off
mid term

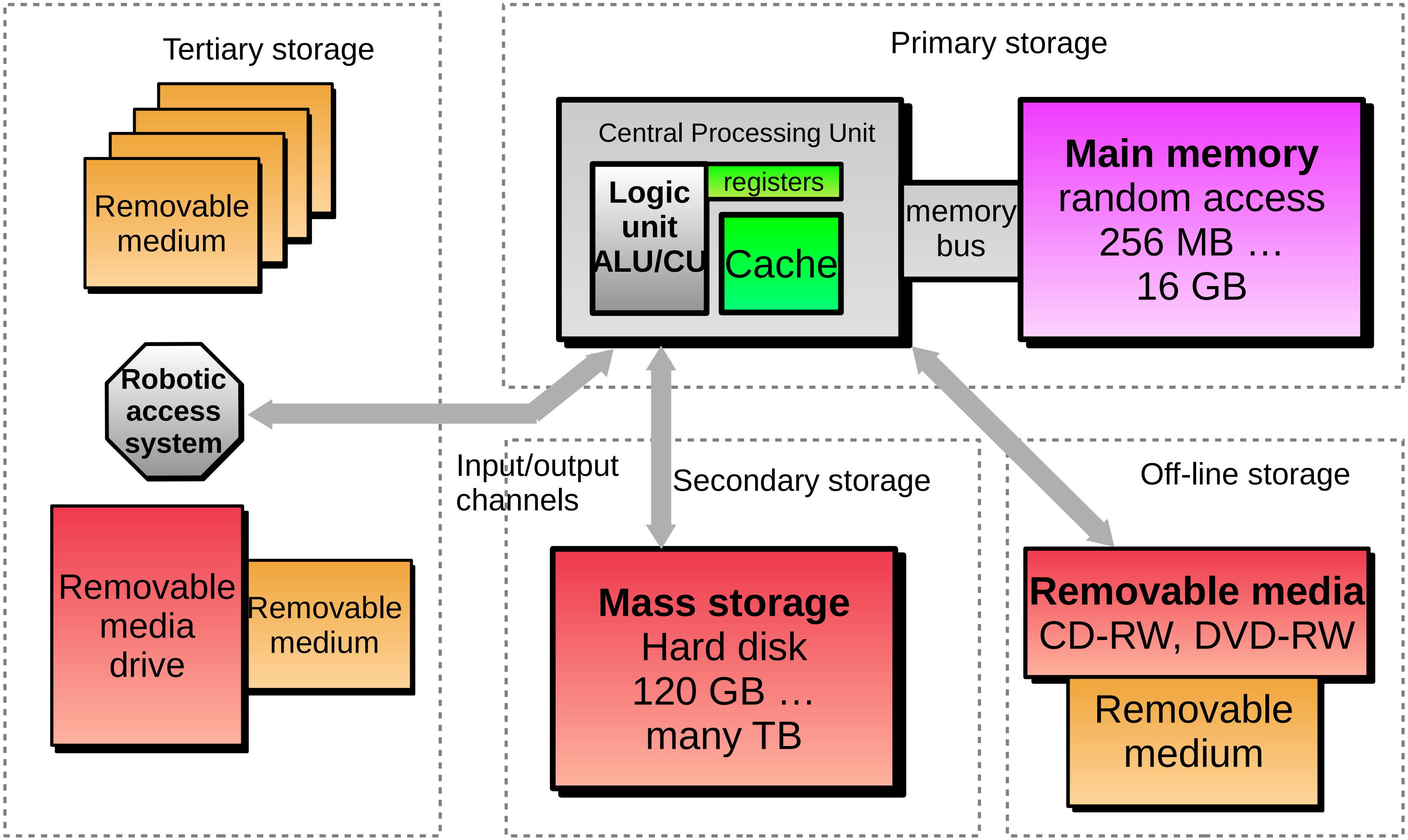
hard drive
slow, very cheap

power off
long term

tape backup
very slow, affordable

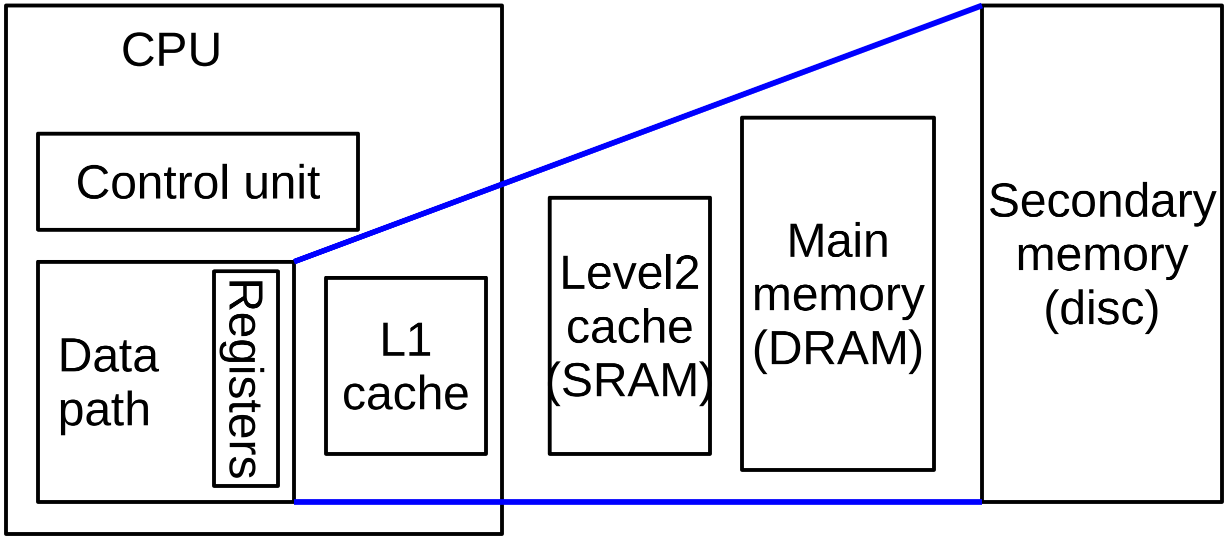
Source: Wikipedia.org

Memory/storage in computer system



Source: Wikipedia.org

Contemporary price/size examples



Type/ Size	L1 32kB	Sync SRAM	DDR3 16 GB	HDD 3TB
Price	10 kč/kB	300 kč/MB	123 kč/GB	1 kč/GB
Speed/ throughput	0.2...2ns	0.5...8 ns/word	15 GB/sec	100 MB/sec

Some data can be available in more copies (consider levels and/or SMP).
Mechanisms to keep consistency required if data are modified.

Mechanism to lookup demanded information?

- According to the address and other management information (data validity flags etc).
- The lookup starts at the most closely located memory level (local CPU L1 cache).
- Requirements:
  - Memory consistency/coherency.
- Used means:
  - Memory management unit to translate virtual address to physical and signal missing data on given level.
  - Mechanisms to free (swap) memory locations and migrate data between hierarchy levels
- Hit (data located in upper level – fast), miss (copy from lower level required)

Processor-memory performance gap solution – cache

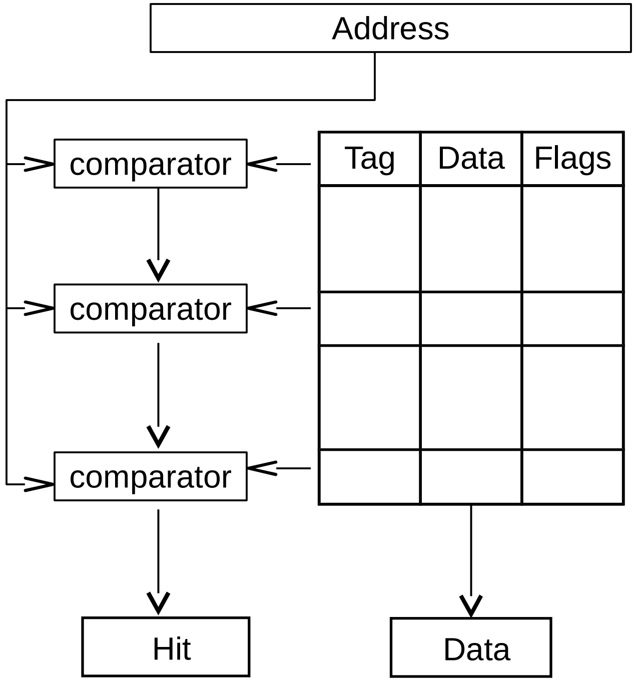
Performance gap between CPU and main memory

- Solution – **cache** memory
- Cache – component that (transparently) stores data so that future requests for that data can be served faster
- Transparent cache – hidden memory

- Placed between two subsystems with different data throughput. It speeds-up access to (recently) used data.
- This is achieved by maintaining copy of data on memory device faster than the original storage

Initial idea – fully associative cache

- **Tag** – the key to locate data (value) in the cache. The original address in the main memory for fully associative case. Size of this field is given by number of bits in an address – i.e. 32, 48 or 64
- **Data** – the stored information, basic unit – word – is usually 4 bytes
- **Flags** – additional bits to keep service information.

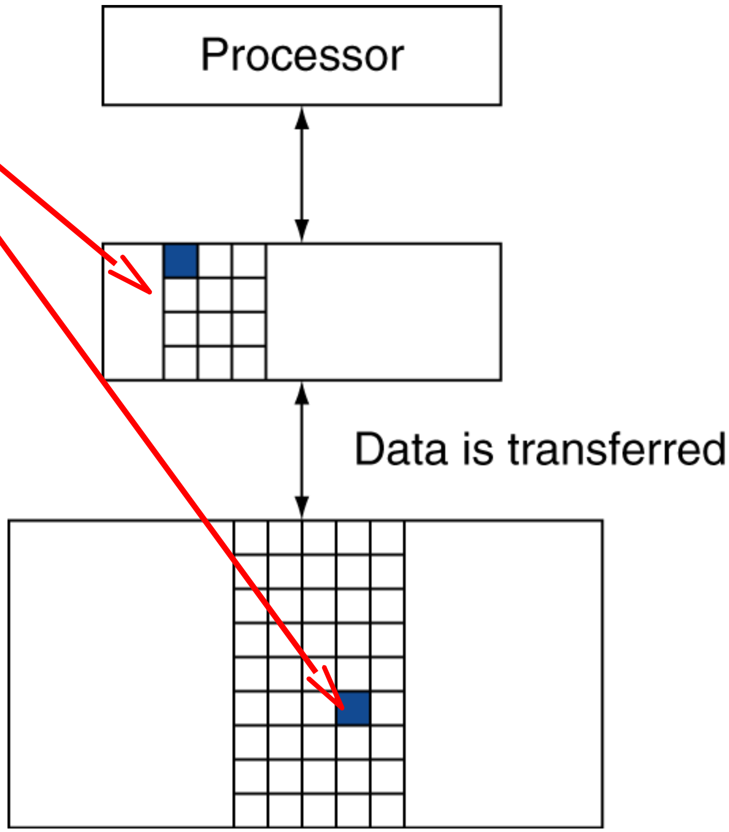


Cache line of fully associative cache

Tag	Data	Flags

Definitions for cache memory

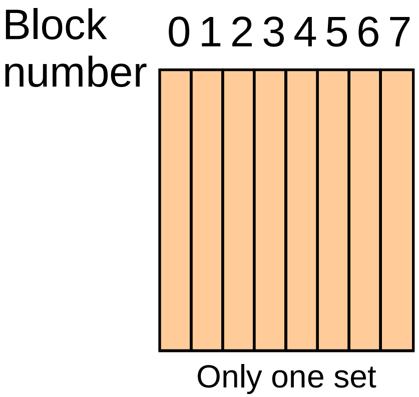
- **Cache line** or **cache block** – basic unit copied between levels
  - May be multiple words
  - Usual cache line size from 8B up to 1KB
- If accessed data is present in upper level
  - **Hit**: access satisfied by upper level
    - **Hit rate**: hits/accesses
- If accessed data is absent
  - **Miss**: block copied from lower level
    - Time taken: **miss penalty**
    - **Miss rate**: misses/accesses = 1 – hit rate
  - Then the accessed data is supplied from upper level



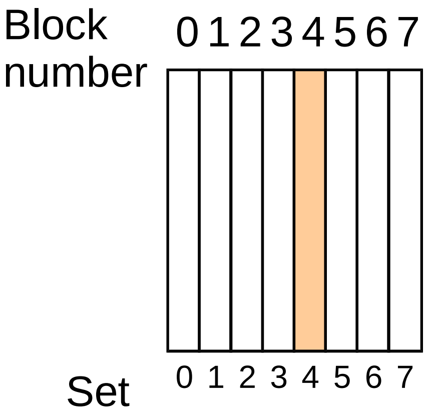
Example to illustrate base cache types

- The cache capacity 8 blocks. Where can be block/address 12 placed for
  - Fully associative
  - Direct mapped
  - N-way (set) associative – i.e. N=2 (2-way cache)

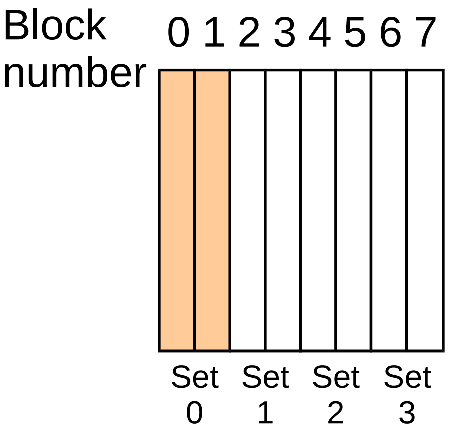
Fully associative:
Address 12 can be placed anywhere



Direct mapped:
Address 12 placed only to block 4 (12 mod 8)



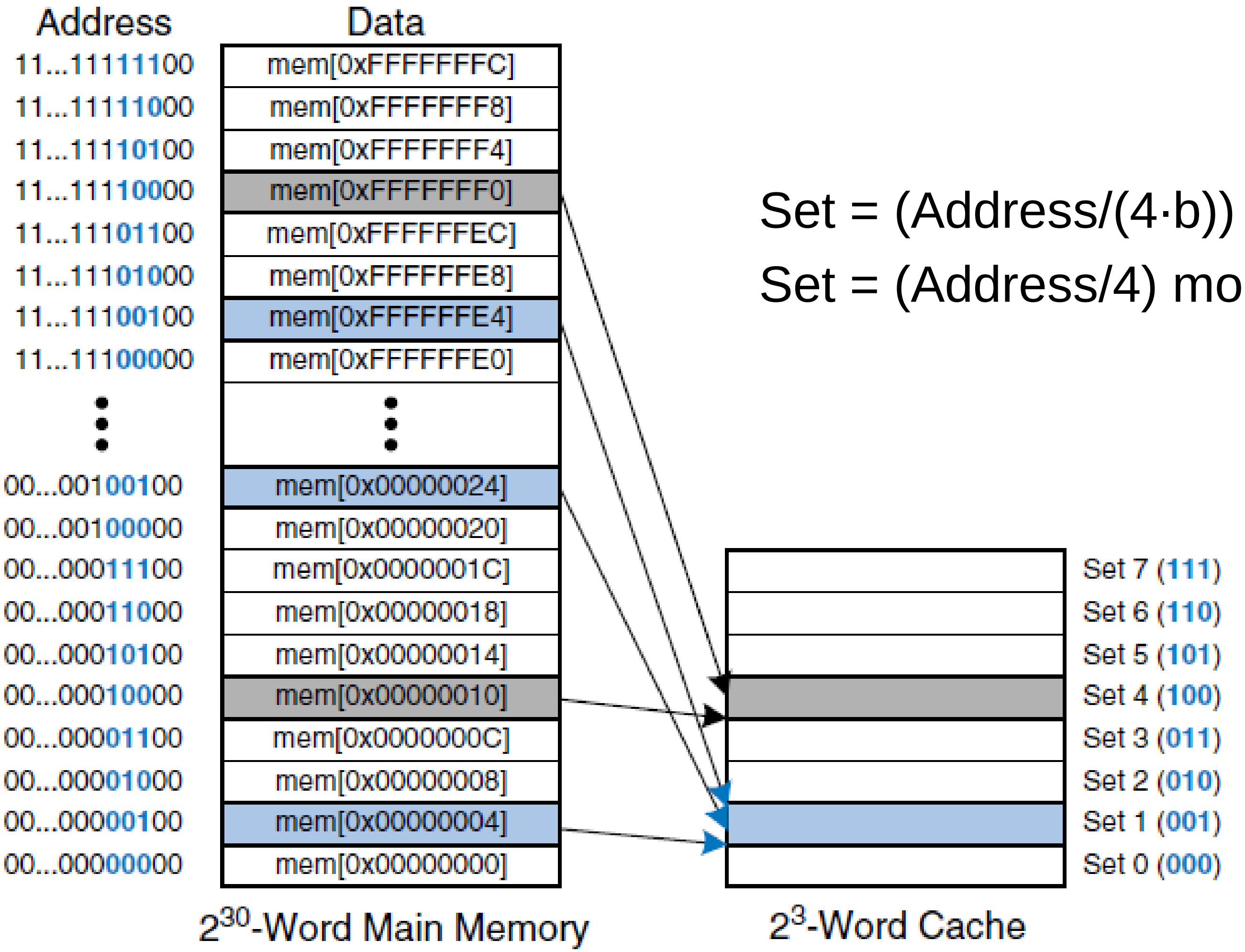
2-way associative:
Address 12 is placed into set 0 (12 mod 4)



Direct mapped cache

$$\text{Set} = (\text{Address}/(4 \cdot b)) \bmod S$$

$$\text{Set} = (\text{Address}/4) \bmod 8$$



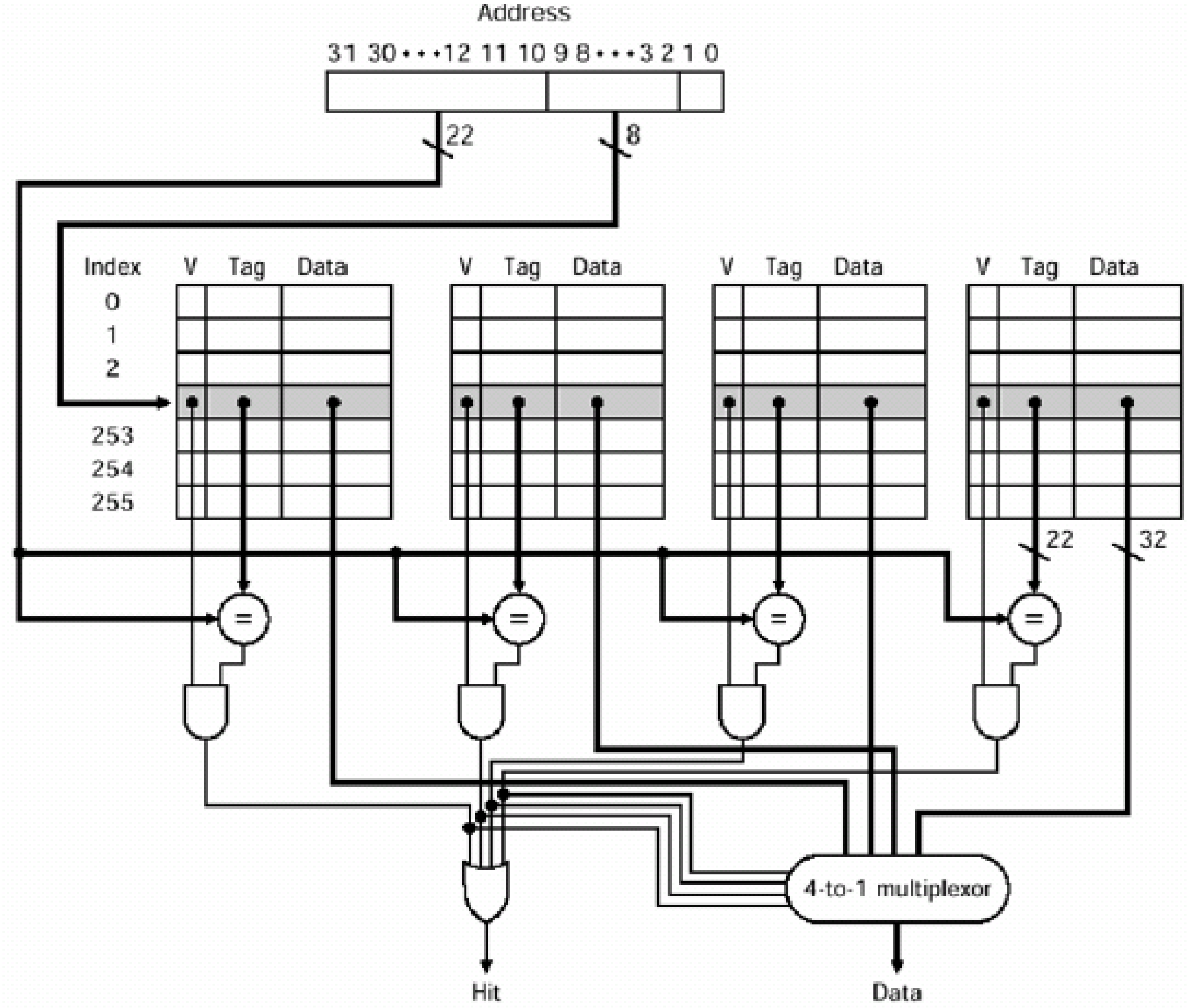
Address	Data
11...11111100	mem[0xFFFFFFFC]
11...11111000	mem[0xFFFFFFF8]
11...11110100	mem[0xFFFFFFF4]
11...11110000	mem[0xFFFFFFF0]
11...11101100	mem[0xFFFFFFEC]
11...11101000	mem[0xFFFFFFE8]
11...11100100	mem[0xFFFFFFE4]
11...11100000	mem[0xFFFFFFE0]
⋮	⋮
00...00100100	mem[0x00000024]
00...00100000	mem[0x00000020]
00...00011100	mem[0x0000001C]
00...00011000	mem[0x00000018]
00...00010100	mem[0x00000014]
00...00010000	mem[0x00000010]
00...00001100	mem[0x0000000C]
00...00001000	mem[0x00000008]
00...00000100	mem[0x00000004]
00...00000000	mem[0x00000000]

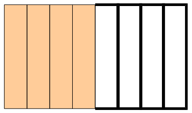
2^{30}-Word Main Memory

Set 7 (111)
Set 6 (110)
Set 5 (101)
Set 4 (100)
Set 3 (011)
Set 2 (010)
Set 1 (001)
Set 0 (000)

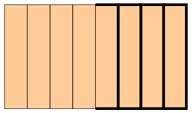
2^{3}-Word Cache

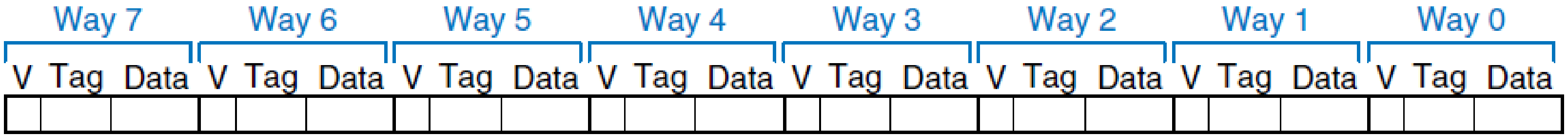
4-way set associative cache





Fully associative cache as special N-way case





- From the above, a fully associative cache can be considered as N-way with only one set. N=B=C/(b·4)
- The same way we can define direct mapped cache as a special case where the number of ways is one.

Important cache access statistical parameters

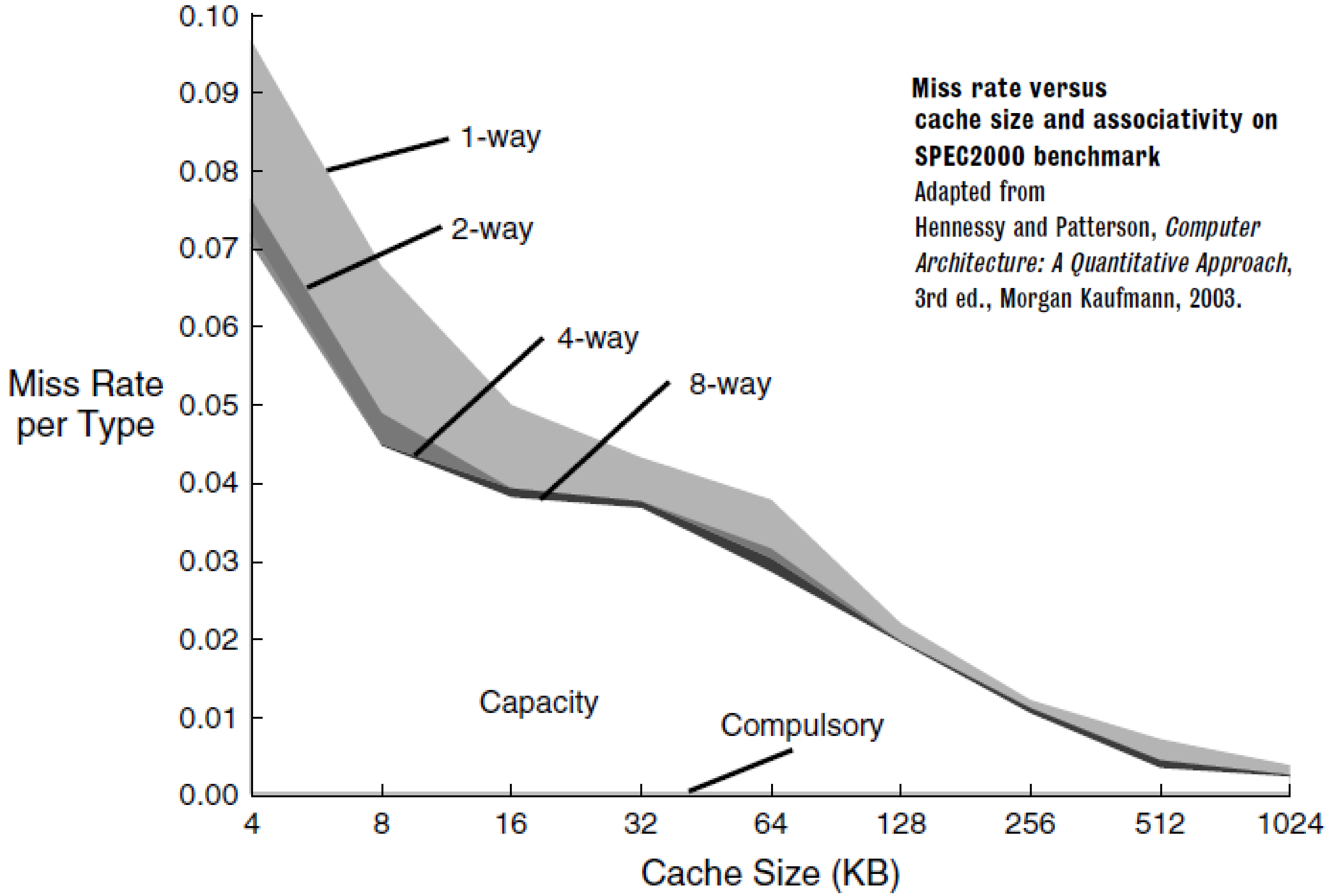
- **Hit Rate** – number of memory accesses satisfied by given level of cache divided by number of all memory accesses
- **Miss Rate** – same, but for requests resulting in

  access to slower memory = 1 – Hit Rate
- **Miss Penalty** – time required to transfer block (data) from lower/slower memory level
- Average Memory Access Time (AMAT)

$$\text{AMAT} = \text{Hit Time} + \text{Miss Rate} \times \text{Miss Penalty}$$

- Miss Penalty for multi-level cache can be computed by recursive application of AMAT formula

Comparison of different cache sizes and organizations



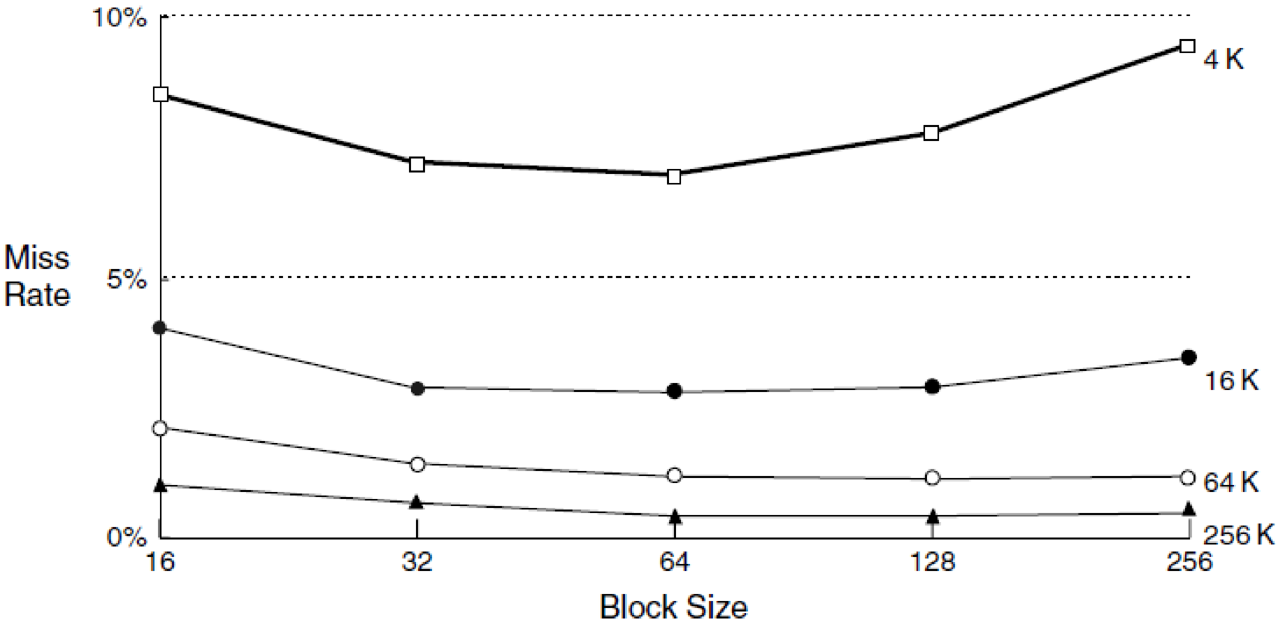
Miss rate versus cache size and associativity on SPEC2000 benchmark

Adapted from Hennessy and Patterson, *Computer Architecture: A Quantitative Approach*, 3rd ed., Morgan Kaufmann, 2003.

Remember: 1. miss rate is not cache parameter/feature!
2. miss rate is not parameter/feature of the program!

What can be gained from spatial locality?

Miss rate of consecutive accesses can be reduced by increasing block size. On the other hand, increased block size for same cache capacity results in smaller number of sets and higher probability of conflicts (set number aliases) and then to increase of miss rate.



Miss rate versus block size and cache size on SPEC92 benchmark Adapted from Hennessy and Patterson, *Computer Architecture: A Quantitative Approach*, 3rd ed., Morgan Kaufmann, 2003.

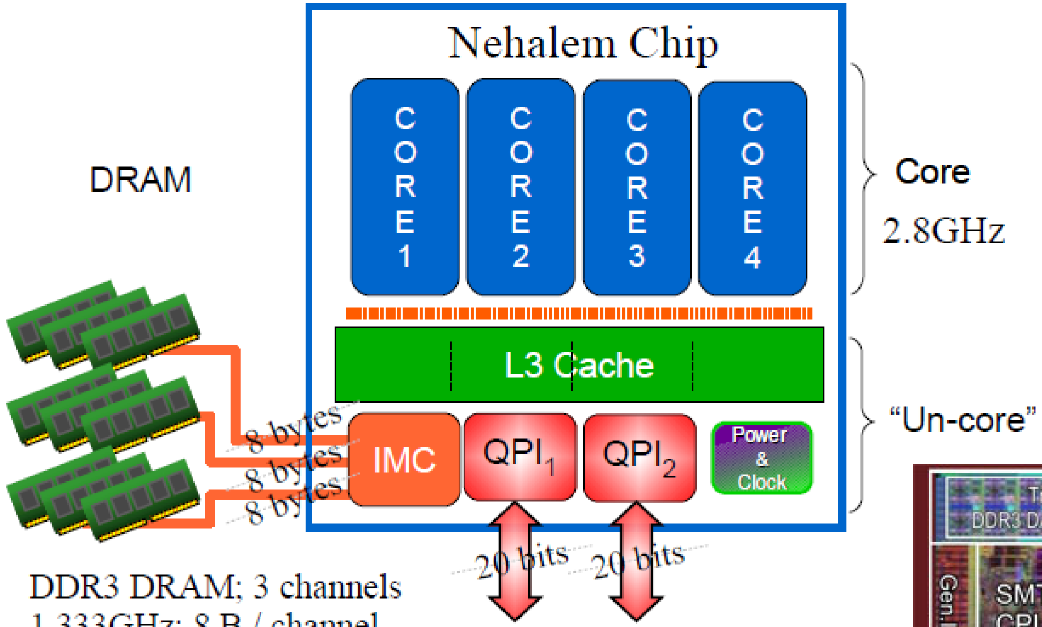
Multi-level cache organization

Multiple cache levels – development directions

- Primary/L1 cache – tightly coupled to the CPU
  - Fast but small. Main objective: minimal Hit Time/latency
  - Usually separated caches for instruction and for data
  - Size usually selected so that cache lines can be virtually tagged without aliasing. (set/way size is smaller than page size)
- L2 cache resolves cache misses of the primary cache
  - Much bigger and slower but still faster than main memory. Main goal: low Miss Rate
- L2 cache misses are resolved by main memory
- Trend to introduce L3 caches, inclusive versus exclusive cache

	Usual for L1	**Usual for L2**
Block count	250-2000	15 000-250 000
KB	16-64	2 000-3 000
Block size in bytes	16-64	64-128
Miss penalty (cycles)	10-25	100-1 000
Miss rates	2-5%	0,1-2%

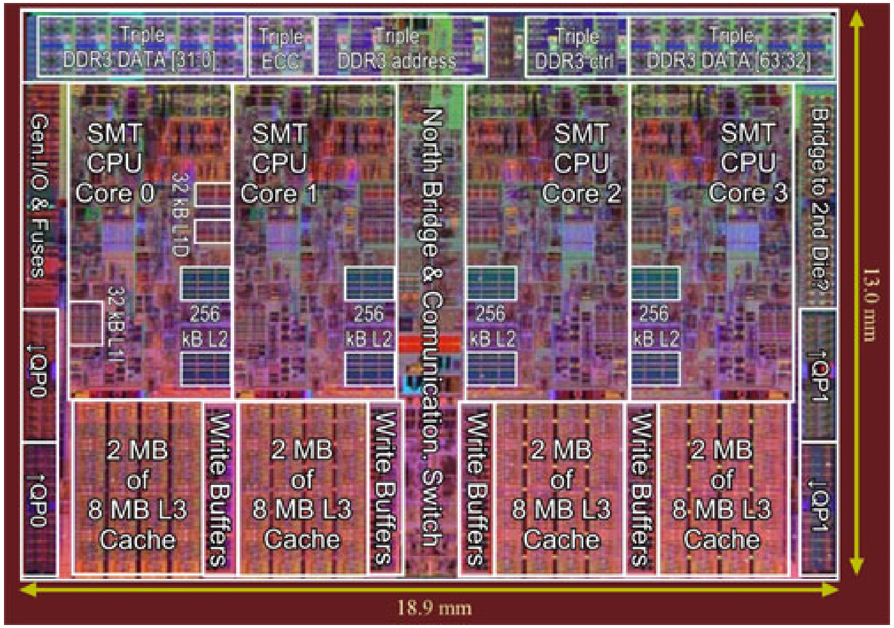
Intel Nehalem – example of Harvard three-level cache



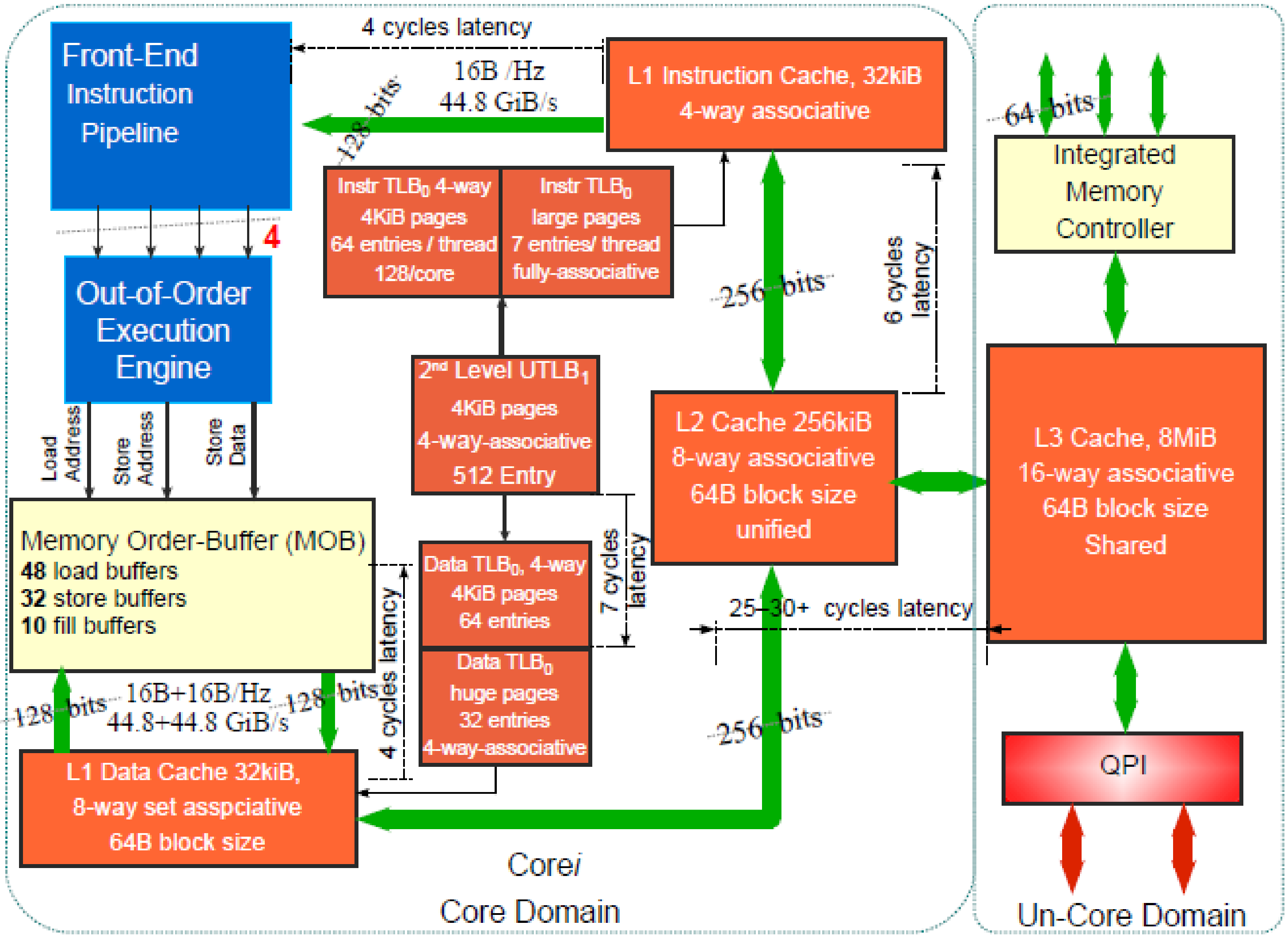
DDR3 DRAM; 3 channels
1.333GHz; 8 B / channel
31.992 GB/s aggregate
7.998 GB/s / core

Intel® QPI point-to-point link
6.4 GT/s; full-duplex;
12.8 GiB/s + **12.8**GiB/s

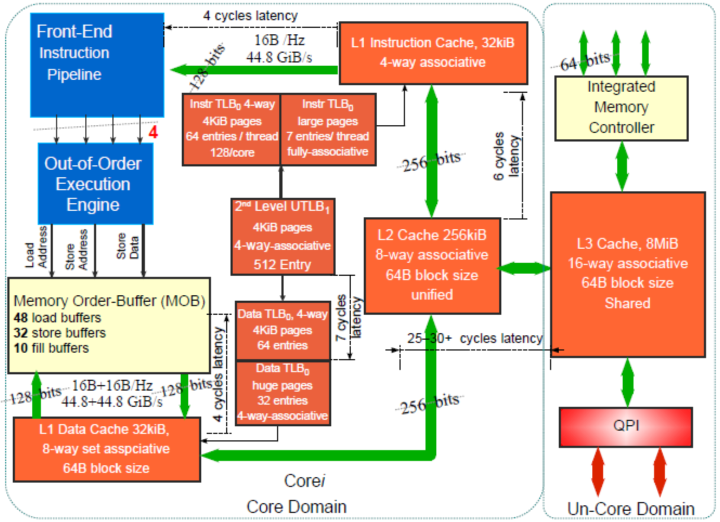
- IMC: integrated memory controller with 3 DDR3 memory channels,
- QPI: Quick-Path Interconnect ports



Intel Nehalem – memory subsystem structure


Front-End Instruction Pipeline
4 cycles latency
16B /Hz
44.8 GiB/s
128 bits
L1 Instruction Cache, 32kiB 4-way associative
Instr TLB$_0$ 4-way 4KiB pages 64 entries / thread 128/core
Instr TLB$_0$ large pages 7 entries/ thread fully-associative
4
Out-of-Order Execution Engine
256 bits
6 cycles latency
64 bits
Integrated Memory Controller
2nd Level UTLB$_1$ 4KiB pages 4-way-associative 512 Entry
L2 Cache 256kiB 8-way associative 64B block size unified
L3 Cache, 8MiB 16-way associative 64B block size Shared
Load Address
Store Address
Store Data
Memory Order-Buffer (MOB)
48 load buffers
32 store buffers
10 fill buffers
Data TLB$_0$, 4-way 4KiB pages 64 entries
7 cycles latency
25–30+ cycles latency
Data TLB$_0$ huge pages 32 entries 4-way-associative
4 cycles latency
128 bits
16B+16B/Hz
44.8+44.8 GiB/s
128 bits
256 bits
QPI
L1 Data Cache 32kiB, 8-way set asspciative 64B block size
Corei
Core Domain
Un-Core Domain

Notes for Intel Nehalem example


Front-End Instruction Pipeline
4 cycles latency
16B /Hz
44.8 GiB/s
128 bits
L1 Instruction Cache, 32kiB 4-way associative
4
Instr TLB$_0$ 4-way 4KiB pages 64 entries / thread 128/core
Instr TLB$_0$ large pages 7 entries/ thread fully-associative
256 bits
6 cycles latency
Out-of-Order Execution Engine
2nd Level UTLB$_1$ 4KiB pages 4-way-associative 512 Entry
L2 Cache 256kiB 8-way associative 64B block size unified
Load Address
Store Address
Store Data
Memory Order-Buffer (MOB)
48 load buffers
32 store buffers
10 fill buffers
Data TLB$_0$, 4-way 4KiB pages 64 entries
7 cycles latency
25–30+ cycles latency
Data TLB$_0$ huge pages 32 entries 4-way-associative
4 cycles latency
128 bits
16B+16B/Hz
44.8+44.8 GiB/s
128 bits
256 bits
L1 Data Cache 32kiB, 8-way set asspciative 64B block size
Core*i*
Core Domain
64 bits
Integrated Memory Controller
L3 Cache, 8MiB 16-way associative 64B block size Shared
QPI
Un-Core Domain

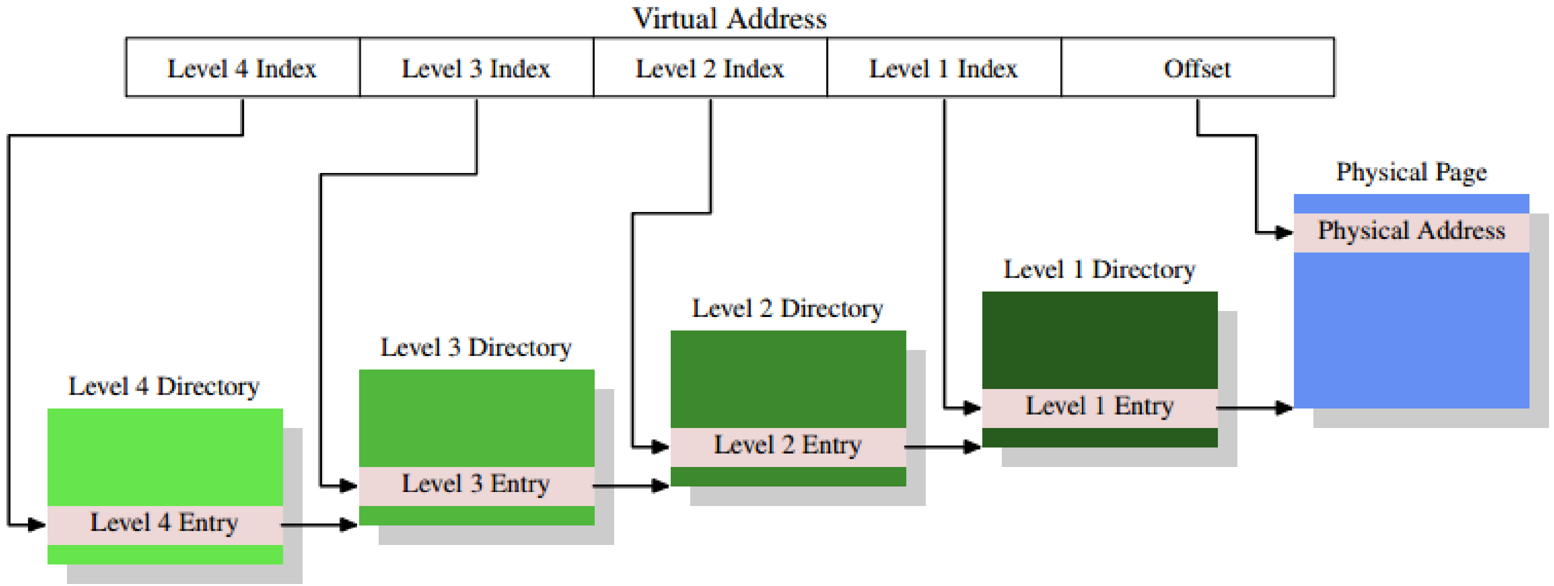
- Block size: 64B
- CPU reads whole cache line/block from main memory and each is 64B aligned
- (6 LS bits are zeros), partial line fills allowed
- L1 – Harvard. Shared by two (H)threads instruction – 4-way 32kB, data 8-way 32kB
- L2 – unified, 8-way, non-inclusive, WB
- L3 – unified, 16-way, inclusive (each line stored in L1 or L2 has copy in L3), WB
- Store Buffers – temporal data store for each write to eliminate wait for write to the cache or main memory. Ensure that final stores are in original order and solve “transaction” rollback or forced store for:

  - exceptions, interrupts, serialization/barrier instructions, lock prefix,..
- TLBs (Translation Lookaside Buffers) are separated for the first level

  Data L1 32kB/8-ways results in 4kB range (same as page) which allows to use 12 LSBs of virtual address to select L1 set in parallel with MMU/TLB

Virtual memory

Multi-level page table – translation overhead

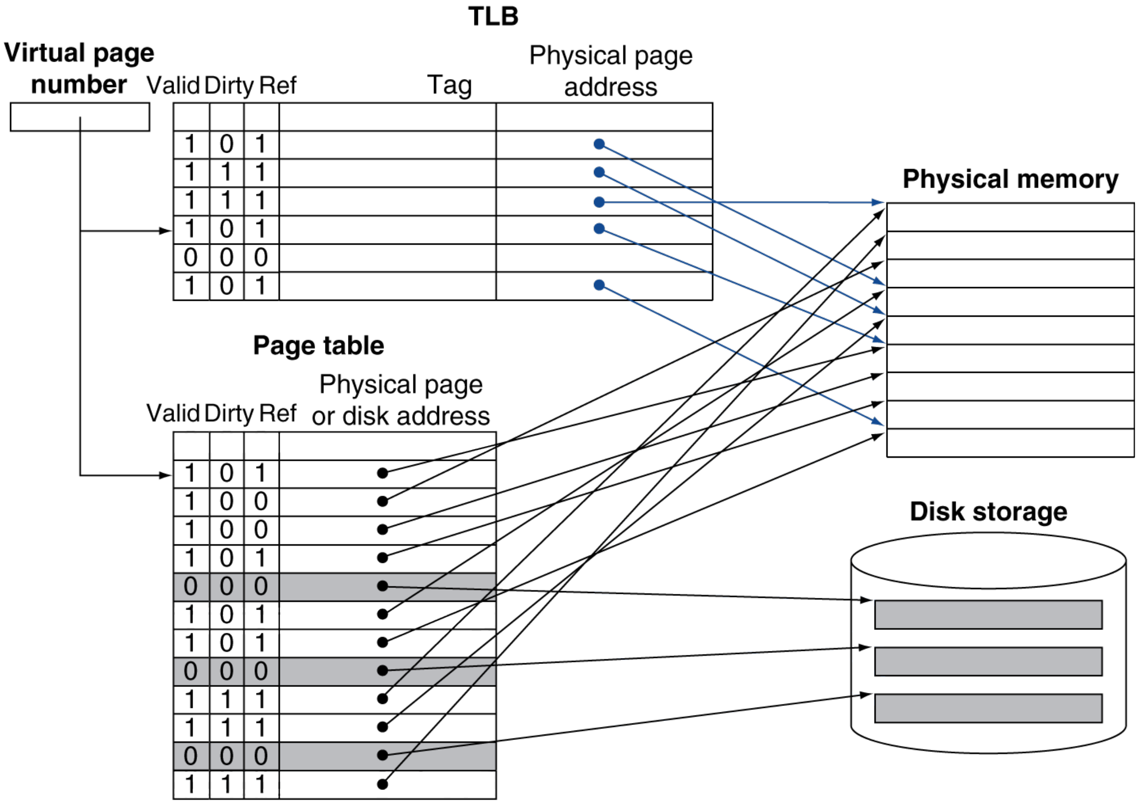


4-Level Address Translation

- Translation would take long time, even if entries for all levels were present in cache. (One access per level, they cannot be done in parallel.)
- The solution is to cache found/computed physical addresses
- Such cache is labeled as Translation Look-Aside Buffer
- Even multi-level translation caching are in use today

Fast MMU/address translation using TLB

- Translation-Lookaside Buffer, or may it be, more descriptive name – Translation-Cache
- Cache of frame numbers where key is page virtual addresses



TLB

Valid	Dirty	Ref	Tag	Physical page address
1	0	1		
1	1	1		
1	1	1		
1	0	1		
0	0	0		
1	0	1		

Page table

Valid	Dirty	Ref	Physical page or disk address
1	0	1	
1	0	0	
1	0	0	
1	0	1	
0	0	0	
1	0	1	
1	0	1	
0	0	0	
1	1	1	
1	1	1	
0	0	0	
1	1	1	

Typical sizes of today I/D and TLB caches comparison

	Typical paged memory parameters	Typical TLB
Size in blocks	16 000-250 000	40-1024
Size	500-1 000 MB	0,25-16 KB
Block sizes in B	4 000-64 000	4-32
Miss penalty (clock cycles)	10 000 000 – 100 000 000	10-1 000
Miss rates	0,00001-0,0001%	0,01-2
Backing store	Pages on the disk	Page table in the main memory
Fast access location	Main memory frames	TLB

Hierarchical memory caveats

Some problems to be aware of

- Memory coherence – definition on next slide
- Single processor (single core) systems
  - Solution: D-bit and Write-back based data transactions
  - Even in this case, consistency with DMA requited (SW or HW)
- Multiprocessing (symmetric) SMP with common and shared memory – more complicated. Solutions:
  - Common memory bus: Snooping, MESI, MOESI protocol
  - Broadcast
  - Directories
- More about these advanced topics in A4M36PAP

Coherency definition

- Memory coherence is an issue that affects the design of computer systems in which two or more processors, cores or bus master controllers share a common area of memory.
- Intuitive definition: The memory subsystem is coherent if the value returned by each read operation is always the same as the value written by the most recent write operation to the same address.
- More formal: P – set of CPU's. $\mathbf{x_m} \in \mathbf{X}$ locations. $\forall \mathbf{p_i}, \mathbf{p_k} \in \mathbf{P}$: $\mathbf{p_i} \neq \mathbf{p_k}$. Memory system is coherent if

  1. $\mathbf{p_i}$ read after $\mathbf{p_i}$ write value **a** to $\mathbf{x_m}$ returns **a** if there is no $\mathbf{p_i}$ or $\mathbf{p_k}$ write between these read and write operations
  2. if $\mathbf{p_i}$ reads $\mathbf{x_m}$ after $\mathbf{p_k}$ write **b** to $\mathbf{x_m}$ and there is no other $\mathbf{p_i}$ or $\mathbf{p_k}$ write to $\mathbf{x_m}$ then $\mathbf{p_i}$ reads **b** if operations are separated by enough time (in other case previous value of $\mathbf{x_m}$ can be read) or architecture specified operations are inserted after write and before read.
  3. writes by multiple CPU's to the given location are serialized such than no CPU reads older value when it already read recent one

Comparison of virtual memory and cache memory

Virtual memory	Cache memory
Page	Block/cache line
Page Fault	Read/Write Miss
Page size: 512 B – 8 KB	Block size: 8 – 128 B
Fully associative	DM, N-way set associative
Victim selection: LRU	LRU/Random
Write Back	Write Thru/Write Back

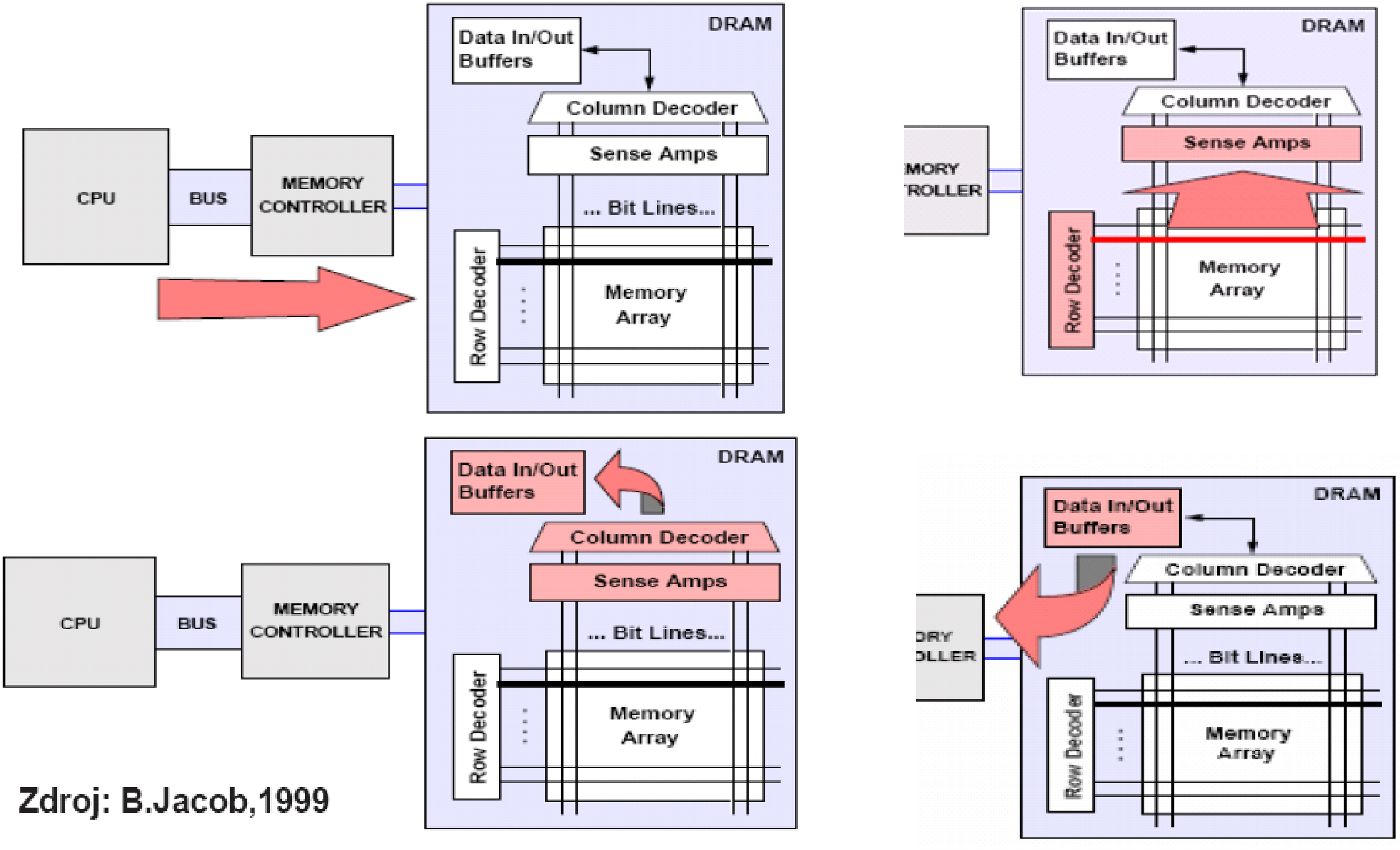
- Remarks.: TLB for address translation can be fully associative, but for bigger sizes is 4-way.
- Do you understand the terms?
  - What does victim represent?
- Important: adjectives cache and virtual mean different things.

Inclusive versus exclusive cache/data backing store

- Mapping of contents of the main memory to the cache memory is **inclusive**, i.e. main memory location cannot be reused for other data when corresponding or updated contents is held in the cache
- If there are more cache levels it can be waste of the space to keep stale/old data in the previous cache level. Snoop cycle is required anyway. The **exclusive** mechanism is sometimes used in such situation.
- **Inclusive** mapping is the rule for secondary storage files mapped into main memory.
- But for swapping of physical contents to swap device/file exclusive or mixed approach is quite common.

Memory realization – memory chips

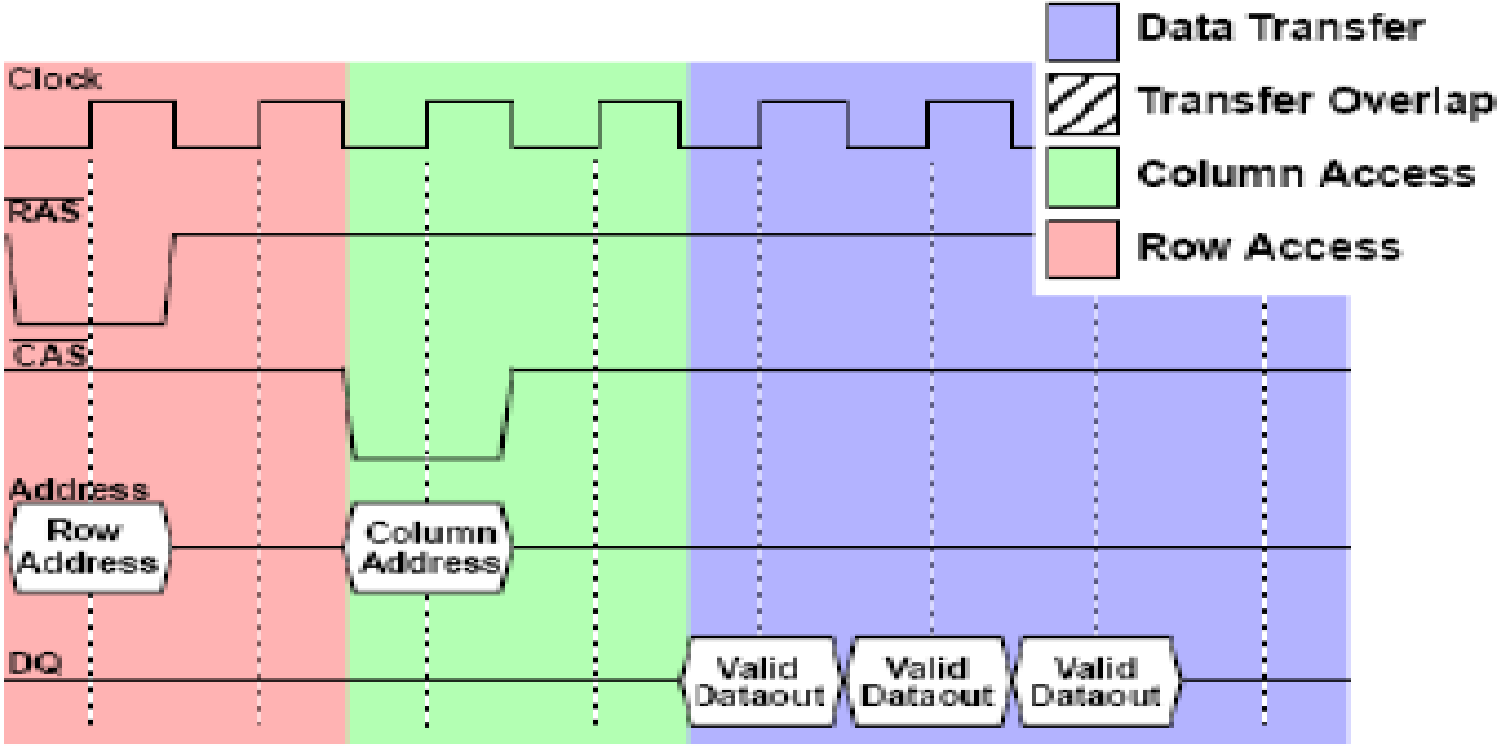
Phases of DRAM memory read



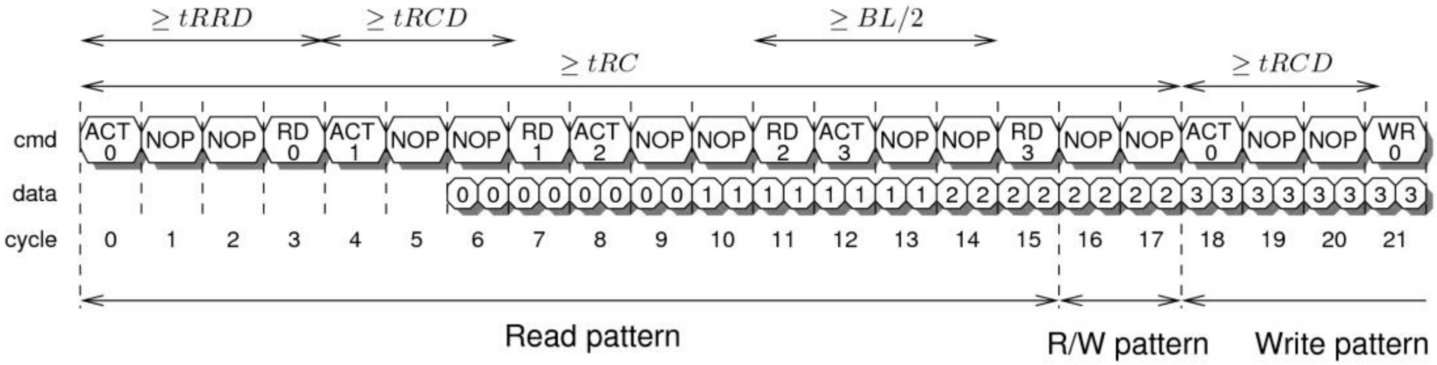
Zdroj: B.Jacob,1999

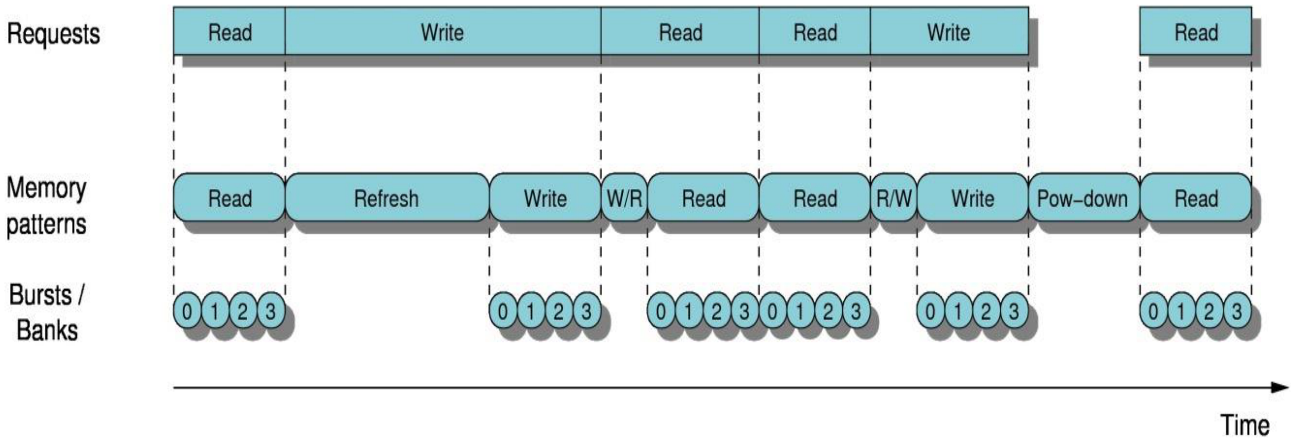
SDRAM – end of 90-ties – synchronous DRAM

- SDRAM chip is equipped by counter that can be used to define continuous block length (burst) which is read together



DDR2 Example and Timing



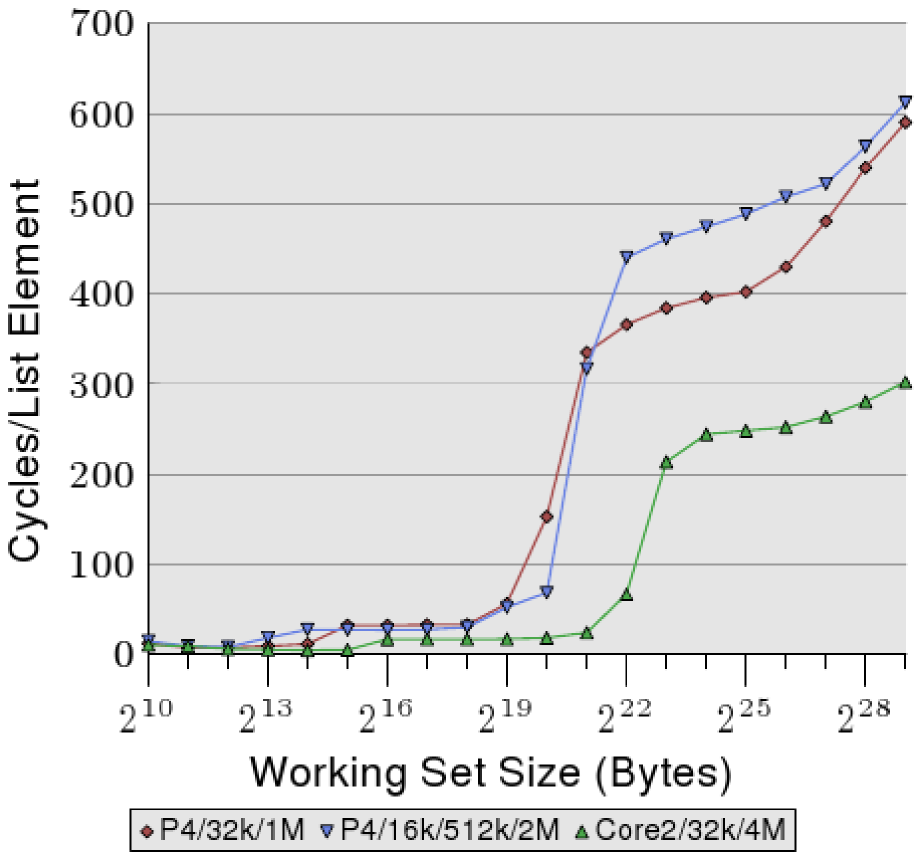


SDRAM – the most widely used main memory technology

- SDRAM – clock frequency up to 100 MHz, 2.5V.
- DDR SDRAM – data transfer at both CLK edges, 2.5V.
- DDR2 SDRAM – lower power consumption 1.8V, frequency up to 400 MHz.
- DDR3 SDRAM – even lower power consumption at 1.5V, frequency up to 800 MHz.
- DDR4 SDRAM …
- There are also other dynamic memory types, I.e. RAMBUS, that use entirely different concept
- All these innovations are focused mainly on throughput, not on the random access latency.

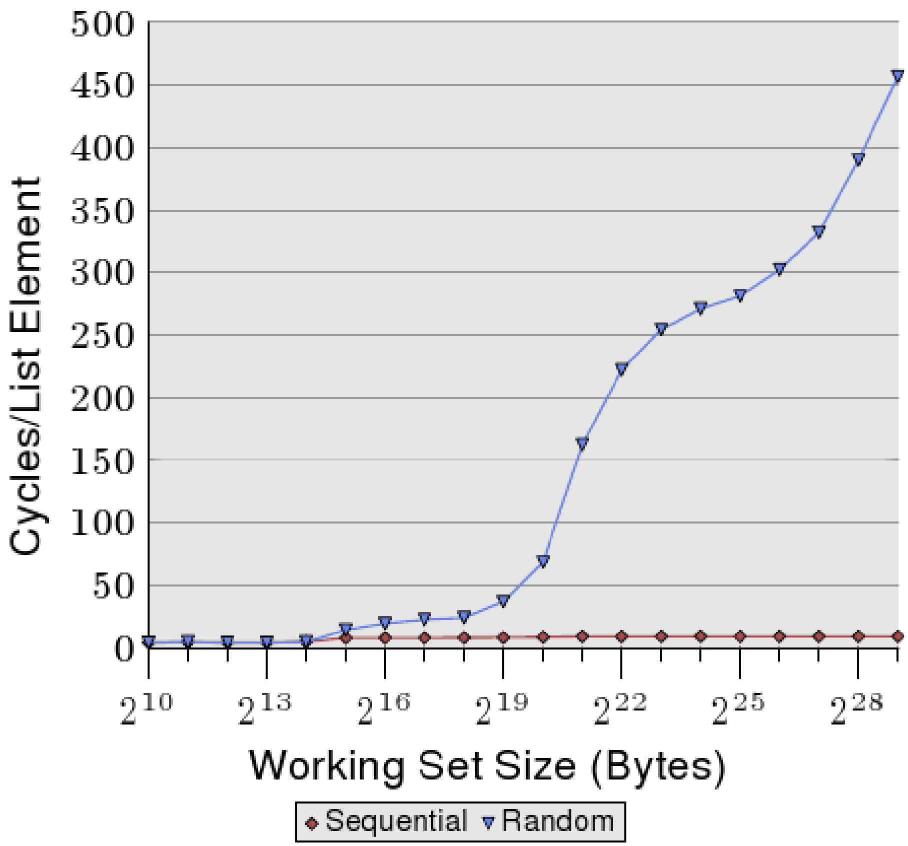
Real Memory Access Time Impact

Benchmark on Real System with L1+L2+L3 Cache



- Inc benchmark,
  128 bytes per element,
  sequential access
- 32kB L1d,
  1MB L2
- 16kB L1d, 512kBL2,
  2M L3
- 32kB L1d,
  4M L2

Single Thread Random Access



- Prefetching cannot help here
- We have seen that data can be accessed from main memory in 200 cycles. High numbers (400) are here because automatic prefetching if now working against us.
- The curve is not flattening at various plateaus: cache miss ratio increases

Example: Matrix multiplication

- Naive implementation

```
for (i = 0; i < N; ++i)
  for (j = 0; j < N; ++j)
    for (k = 0; k < N; ++k)
      res[i][j] += mul1[i][k] * mul2[k][j];
```

- With transposition

```
double tmp[N][N];
for (i = 0; i < N; ++i)
  for (j = 0; j < N; ++j)
    tmp[i][j] = mul2[j][i];
for (i = 0; i < N; ++i)
  for (j = 0; j < N; ++j)
    for (k = 0; k < N; ++k)
      res[i][j] += mul1[i][k] * tmp[j][k];
```

- Performance: naive: 100%, transposed: 23,4%

Single Pass Data Access Optimization

- If you know the data will be used only once, bypass the cache when writing. Hopefully, write-combining will be used.
  - Non-temporal write operations (gcc)

```
#include <emmintrin.h>
void _mm_stream_si32(int *p, int a);
void _mm_stream_si128(int *p, __m128i a);
void _mm_stream_pd(double *p, __m128d a);
#include <xmmintrin.h>
void _mm_stream_pi(__m64 *p, __m64 a);
void _mm_stream_ps(float *p, __m128 a);
#include <ammintrin.h>
void _mm_stream_sd(double *p, __m128d a);
void _mm_stream_ss(float *p, __m128 a);
```

Vectorized Operations with GCC

```
typedef int v4si __attribute__ ((vector_size (16)));
    v4si a, b, c;
    long l;

    c = a + b;
    a = b + 1;    /* a = b + {1,1,1,1}; */
    a = 2 * b;    /* a = {2,2,2,2} * b; */
    a = l + a;    /* Error, cannot convert long to int. */
```

The Basic Linear Algebra Subprograms (BLAS)

- Specifications for the computational kernels that form the basic operations of numerical linear algebra
- Building blocks for higher level linear algebra
- Implemented efficiently by vendors (and others) on most machines

The Three Levels of BLAS

- The Level 1 BLAS are concerned with scalar and vector operations, such as

$$y \leftarrow \alpha x + y,\ \ \alpha \leftarrow x^T y,\ \ \alpha \leftarrow \|x\|_2$$

- the Level 2 BLAS with matrix-vector operations such as

$$y \leftarrow \alpha Ax + \beta y,\ \ x \leftarrow T^{-1}x$$

- and the Level 3 BLAS with matrix-matrix operations such as

$$C \leftarrow \alpha AB + \beta C,\ X \leftarrow \alpha T^{-1}X$$

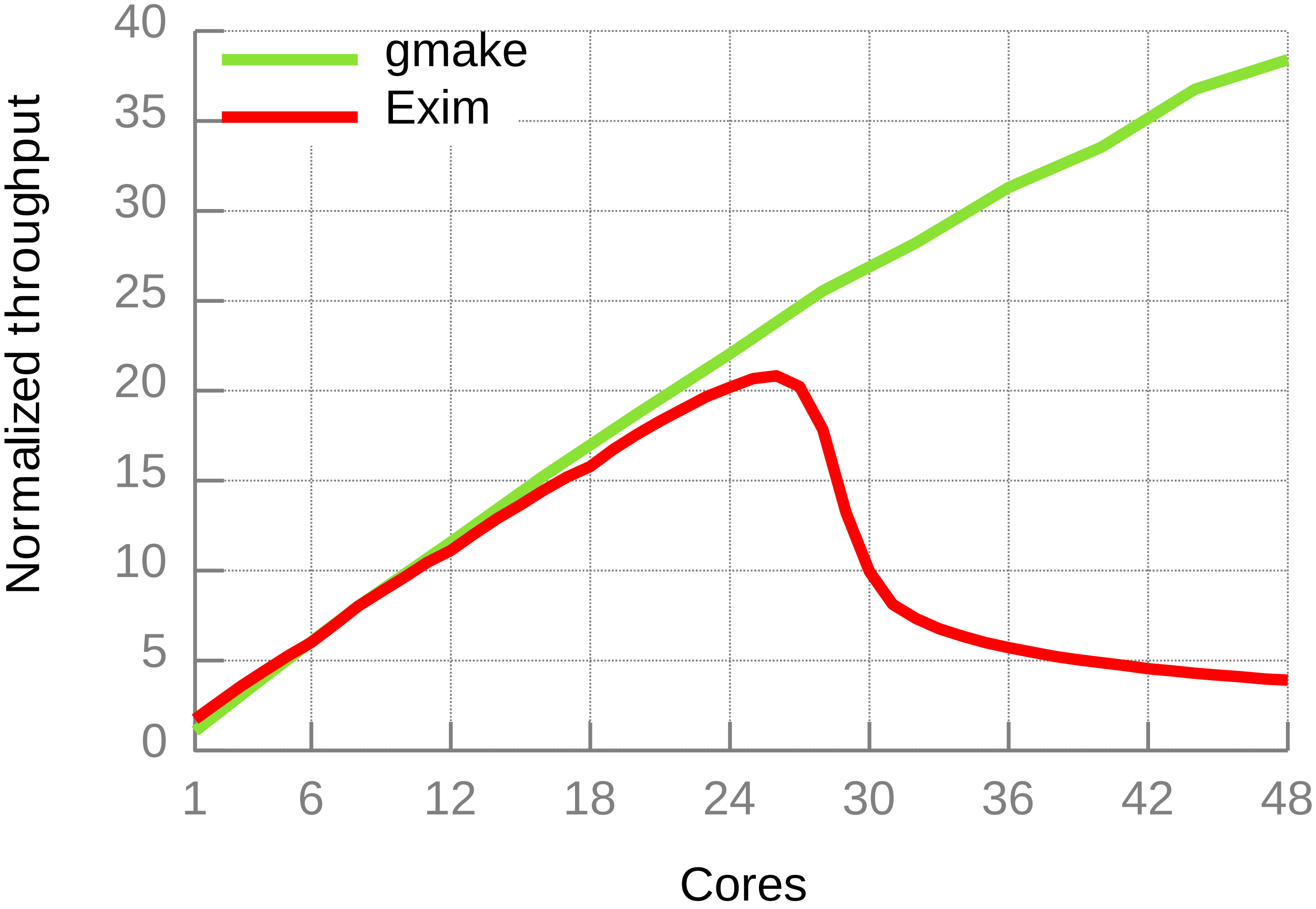
LAPACK

Linear Algebra PACKage for high-performance computers

- Systems of linear equations
- Linear least squares problems
- Eigenvalue and singular value problems, including generalized problems
- Matrix factorizations
- Condition and error estimates
- The BLAS as a portability layer

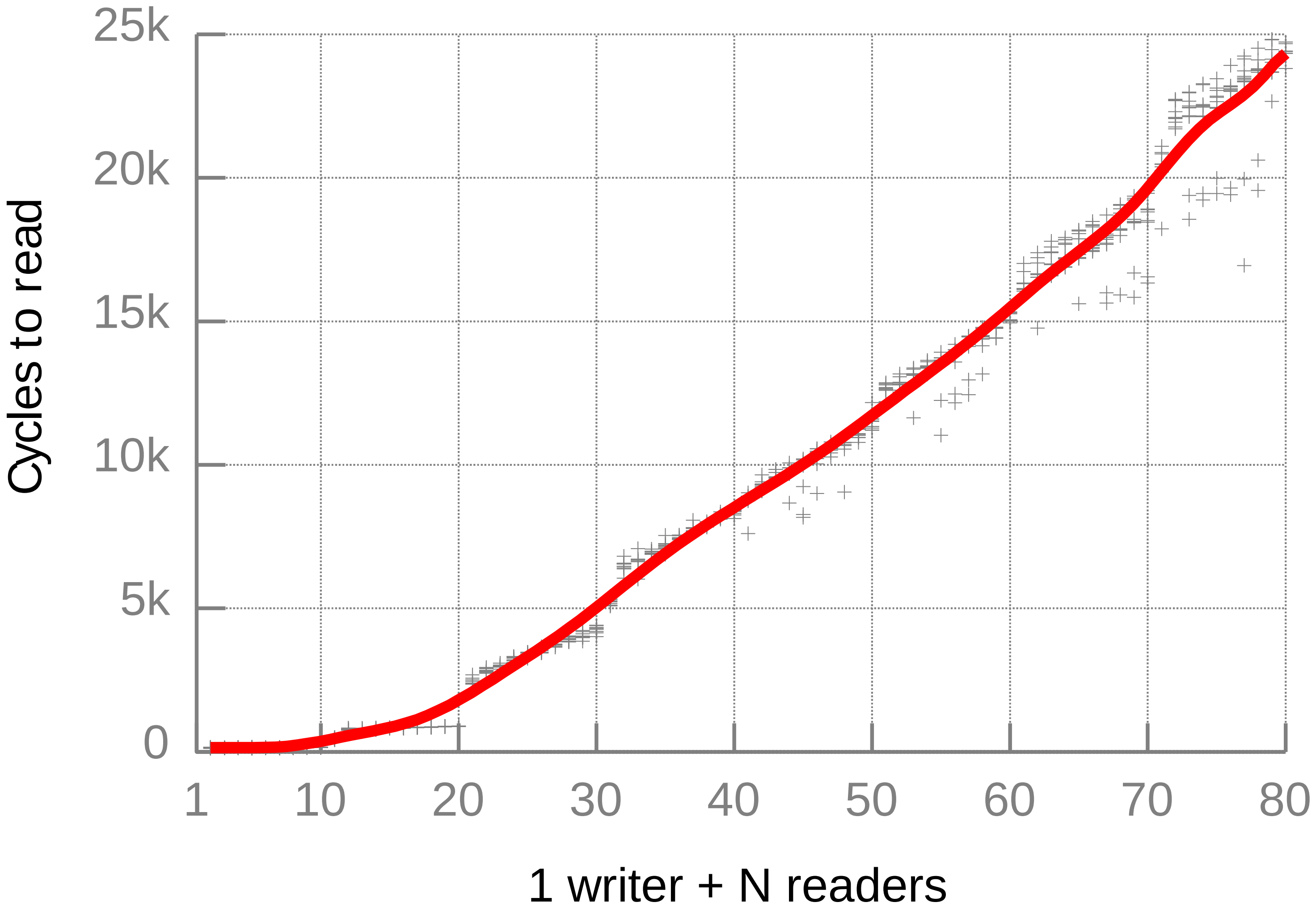
Dense and banded linear algebra for Shared Memory

A scalability bottleneck for Multicore Memory Access



A single contended cache line can wreck scalability

Cost of a contended cache line



What scales on today's multicores?

		Core X		
		W	R	-
Core Y	W	✗	✗	✓
	R	✗	✓	✓
	-	✓	✓	-

Source
The Scalable Commutativity Rule: Designing Scalable Software for Multicore Processors by Austin T. Clements

Multicore Scalable Patterns

- Layer scalability: use scalable data structures
  - Linear and radix arrays
  - Hash tables
  - **Not** balanced trees
- Defer work (reference tracking)
- Precede pessimism with optimism
  - Optimistic check stage followed by pessimistic update stage
- Don't read unless necessary
  - access(F_OK)

GCC Functions for Atomic Memory Access

- Legacy __sync – Intel Itanium Processor-specific Application Binary Interface
  - *type* __**sync_fetch_and**_*add* (*type* *ptr, *type* value, …)

    { tmp = *ptr; *ptr += value; return tmp; }
  - Operations: add sub or and and xor nand
  - *type* __**sync**_*add*_**and_fetch** (*type* *ptr, *type* value, …)
  - bool __sync_bool_compare_and_swap (type *ptr, type oldval, type newval, ...)
  - type __sync_val_compare_and_swap (type *ptr, type oldval, type newval, …)
  - __sync_synchronize (…)
  - type __sync_lock_test_and_set (type *ptr, type value, …)
  - void __sync_lock_release (type *ptr, ...)

C++11 Memory Model Based Atomic Operations

C++11 memory models

- __ATOMIC_RELAXED – No barriers or synchronization.
- __ATOMIC_CONSUME – Data dependency only for both barrier and synchronization with another thread.
- __ATOMIC_ACQUIRE – Barrier to hoisting of code and synchronizes with release (or stronger) semantic stores from another thread.
- __ATOMIC_RELEASE – Barrier to sinking of code and synchronizes with acquire (or stronger) semantic loads from another thread.
- __ATOMIC_ACQ_REL – Full barrier in both directions and synchronizes with acquire loads and release stores in another thread.
- __ATOMIC_SEQ_CST – Full barrier in both directions and synchronizes with acquire loads and release stores in all threads.

C++11 Atomic Operations

- type __atomic_load_n (type *ptr, int memmodel)

  RELAXED, SEQ_CST, ACQUIRE and CONSUME
- void __atomic_load (type *ptr, type *ret, int memmodel)
- __atomic_store_n (type *ptr, type val, int memmodel)

  RELAXED, SEQ_CST, RELEASE
- void __atomic_store (type *ptr, type *val, int memmodel)
- __atomic_exchange_n (type *ptr, type val, int memmodel)

  RELAXED, SEQ_CST, ACQUIRE, RELEASE and ACQ_REL
- void __atomic_exchange (type *ptr, type *val, type *ret, int memmodel)

C++11 Compare and Swap

- bool __atomic_compare_exchange_n (type *ptr, type *expected, type desired, bool weak, int success_memmodel, int failure_memmodel)
- bool __atomic_compare_exchange (type *ptr, type *expected, type *desired, bool weak, int success_memmodel, int failure_memmodel)

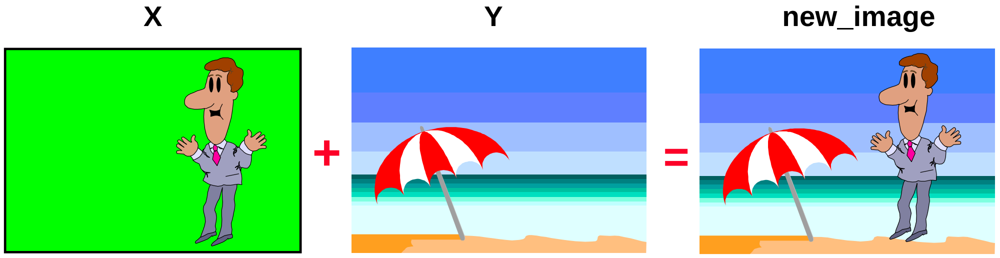
C++11 Arithmetic and Logic Operations

- type __atomic_add_fetch (type *ptr, type val, int memmodel)

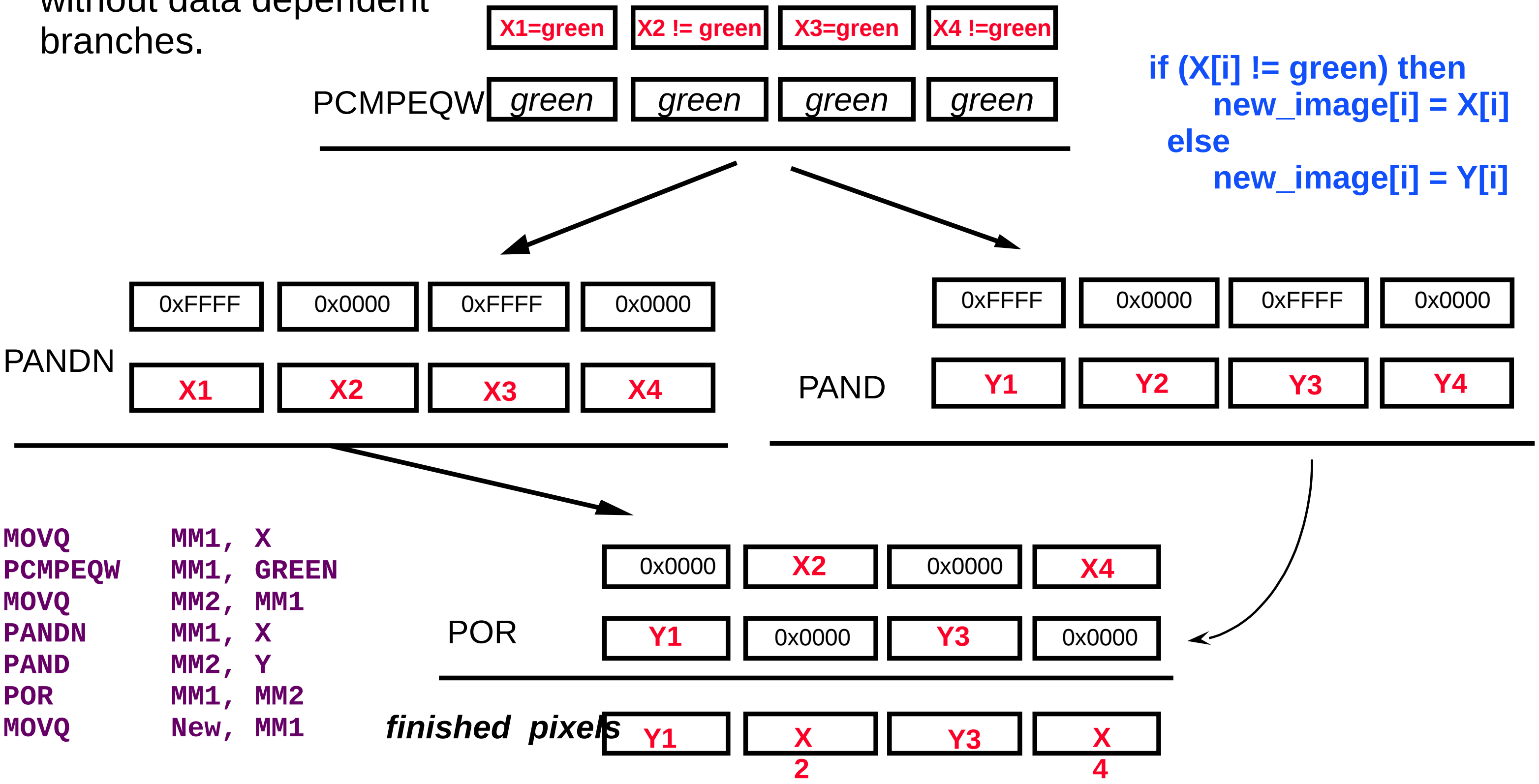
  add, sub, and, xor, or, nand
- type __atomic_fetch_add (type *ptr, type val, int memmodel)
- bool __atomic_test_and_set (void *ptr, int memmodel)
- void __atomic_clear (bool *ptr, int memmodel)
- void __atomic_thread_fence (int memmodel)
- void __atomic_signal_fence (int memmodel)
- bool __atomic_always_lock_free (size_t size, void *ptr)
- bool __atomic_is_lock_free (size_t size, void *ptr)

Other Source of Slowdown – Branch

Packed Comparison (PCMPCC) and the logical instructions enable conditional select operations in parallel and without data dependent branches.



```
if (X[i] != green) then
      new_image[i] = X[i]
  else
      new_image[i] = Y[i]
```



```
MOVQ      MM1, X
PCMPEQW   MM1, GREEN
MOVQ      MM2, MM1
PANDN     MM1, X
PAND      MM2, Y
POR       MM1, MM2
MOVQ      New, MM1
```

Literature to read

More materials:

- What Every Programmer Should Know About Memory by Ulrich Drepper, Red Hat, Inc.
  http://www.akkadia.org/drepper/cpumemory.pdf
  http://lwn.net/Articles/250967/
  http://people.redhat.com/drepper/cpumemory.pdf
- Ulrich Drepper: Parallel Programming with Transactional Memory.
  http://mags.acm.org/queue/200809/data/queue200809-dl.pdf
- Chapter 5 (Large and Fast: Exploiting memory hierarchy) from Hennesy, Patterson CaaQA
- Memory Ordering in Modern Microprocessors by Paul McKenney
  http://www.rdrop.com/users/paulmck/scalability/paper/ordering.2007.09.19a.pdf
- Is Parallel Programming Hard, And, If So, What Can You Do About It?
  https://www.kernel.org/pub/linux/kernel/people/paulmck/perfbook/perfbook.html
- The Scalable Commutativity Rule: Designing Scalable Software for Multicore Processors by Austin T. Clements
  http://dl.acm.org/citation.cfm?doid=2517349.2522712
- Memory Controllers for Real - Time Embedded Systems: Benny Akesson