

1. Why supercomputers?
2. **Modern processors**
 - Stored-program computer architecture
 - Von Neumann architecture
 - Instruction set architecture
 - General purpose cache-based microarchitecture
 - Pipelining
 - Superscalarity
 - SIMD
 - Memory hierarchies
 - Cache
 - Prefetching
 - Cache mapping
 - Cache coherence
3. Basic optimization techniques for serial code
4. Data access optimization
5. Parallel computers
6. Parallelization and optimization strategies
7. Parallel algorithms
8. Shared-memory programming with OpenMP
9. Distributed-memory programming with MPI
10. Hybrid programming (MPI + OpenMP)
11. Heterogeneous architectures (GPUs, Xeon Phi)
12. Energy efficiency

- **1975 – 1995: Specially designed systems for HPC**
 - High performance, high price
 - **Since 1980s: Single-chip general-purpose micro processors in HPC**
 - Invented in the early 1970s, now mature enough for HPC
 - **End of 1990s: Clusters of standard workstations or PC-based hardware become competitive in theoretical performance**
 - **Today: cost-effective, off-the-shelf systems with general-purpose processors in HPC**
 - Commodity clusters
- **Here, we look at the architecture of modern commodity processors**
- Useful to understand recent HPC architectures

1. Why supercomputers?
2. **Modern processors**
 - **Stored-program computer architecture**
 - **Von Neumann architecture**
 - Instruction set architecture
 - General purpose cache-based microarchitecture
 - Pipelining
 - Superscalarity
 - SIMD
 - Memory hierarchies
 - Cache
 - Prefetching
 - Cache mapping
 - Cache coherence
3. Basic optimization techniques for serial code
4. Data access optimization
5. Parallel computers
6. Parallelization and optimization strategies
7. Parallel algorithms
8. Shared-memory programming with OpenMP
9. Distributed-memory programming with MPI
10. Hybrid programming (MPI + OpenMP)
11. Heterogeneous architectures (GPUs, Xeon Phi)
12. Energy efficiency

■ Computers were not programmable

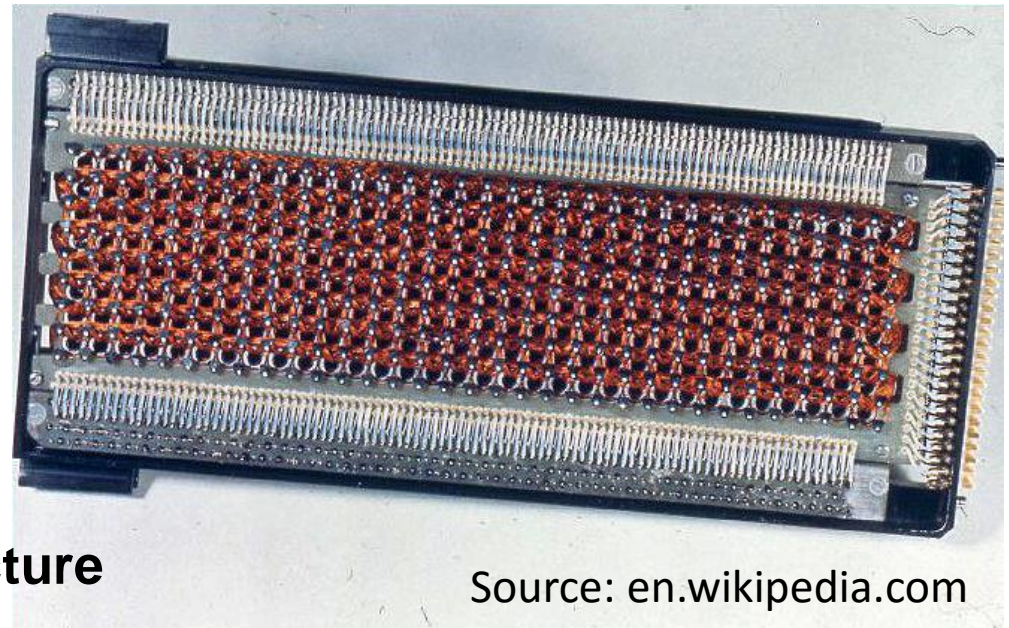
→ Computers executed just a single hard-wired program (fixed programs)

→ “Reprogramming” meant “re-wiring”

→ No program instructions

→ No program storage was necessary

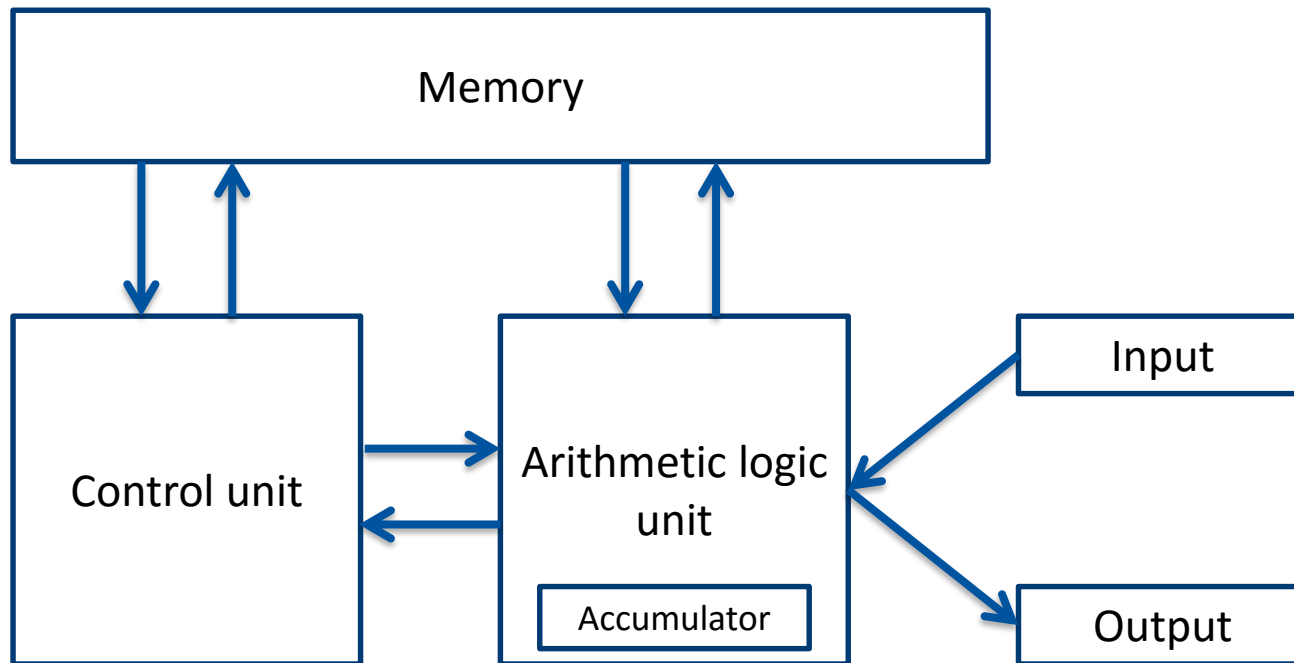
→ Core rope memory of the Apollo guidance computer



Source: en.wikipedia.com

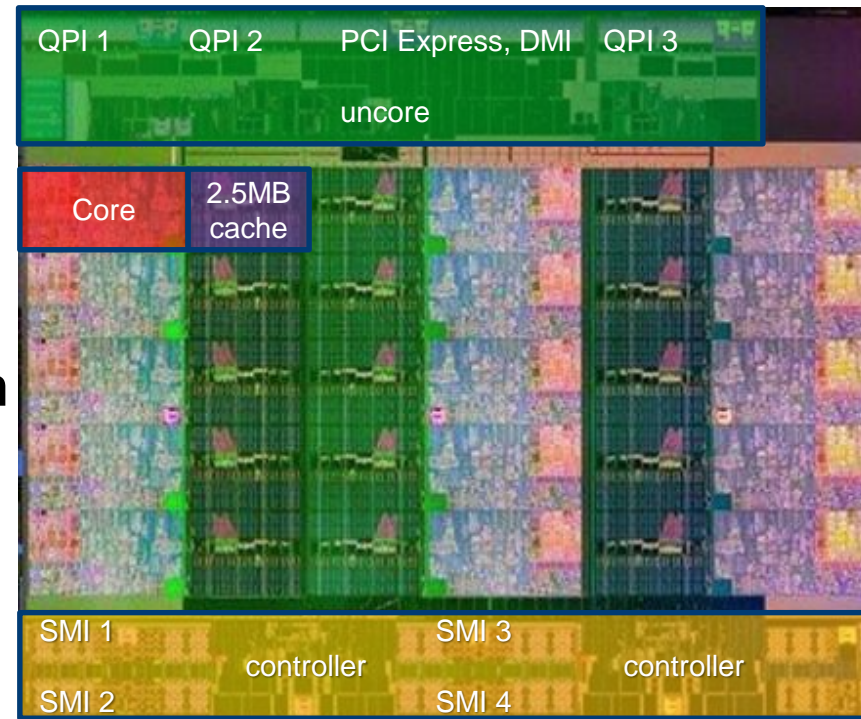
■ Then: Von Neumann architecture

- **Instruction decode: determine operation and operands**
- **Fetch operands from memory**
- **Perform some operation with them**
- **Write the results back into memory**
- **Continue with next instruction**

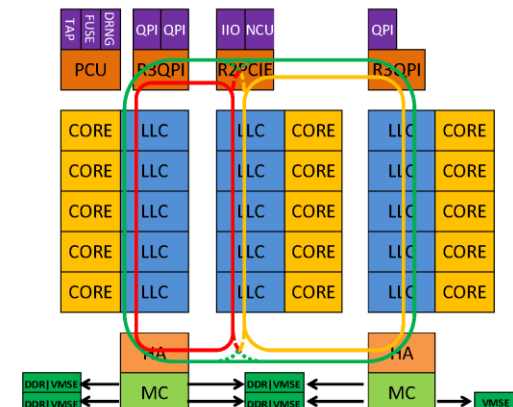


... Today: Intel Ivy Bridge EX

- Released 2013
- Up to 15 cores
- 4.31 billion transistors
- 22nm technology
- 16K L1Data, 16K L1Instruction
- 256 K L2
- 37.5 MB shared L3 cache
- Hyperthreading
- 2.8 GHz * 15 cores * 8 FLOPS /cycle = 672 Gflop/s peak
- Integrated memory controller
- QPI between processors
- Scales to 8 sockets per board



Processor Block Diagram



- 15 cores, 30 threads, 2 integrated memory controllers

- **Modern computers (still) implement the stored program architecture** (by Turing 1936, EDVAC 1949, widely known as Von Neumann architecture)

- Program instructions are numbers stored as data in computer memory

- **Control unit**

- Fetches instructions from memory and executes them

- **Arithmetical/Logic Unit (ALU)**

- Performs computations or manipulations of data stored in memory after being triggered by control unit

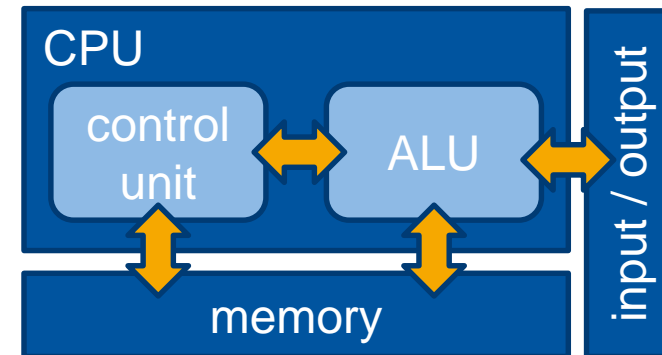
- **CPU consists of control unit + ALU**

- **Basis for all mainstream computers today (+ their problems)**

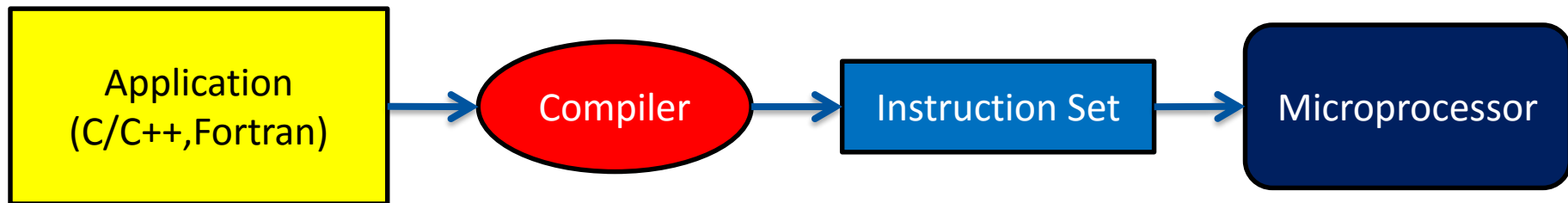
- **Bottlenecks**

- Von Neumann bottleneck: performance of computations is limited by memory access speed

- Inherently sequential architecture (SISD = Single Instruction Single Data)



- **Stored program/von Neumann concept still applies though**
 - Several memory levels (3-4)
 - Multiple arithmetic/logical units
 - Performance optimizations implemented in hardware
- **Applications are developed in high level programming languages**
 - C++, Fortran, ...
- **Compiler translates program to machine instructions specific to the underlying processor architecture**



1. Why supercomputers?
2. **Modern processors**
 - **Stored-program computer architecture**
 - Von Neumann architecture
 - **Instruction set architecture**
 - General purpose cache-based microarchitecture
 - Pipelining
 - Superscalarity
 - SIMD
 - Memory hierarchies
 - Cache
 - Prefetching
 - Cache mapping
 - Cache coherence
3. Basic optimization techniques for serial code
4. Data access optimization
5. Parallel computers
6. Parallelization and optimization strategies
7. Parallel algorithms
8. Shared-memory programming with OpenMP
9. Distributed-memory programming with MPI
10. Hybrid programming (MPI + OpenMP)
11. Heterogeneous architectures (GPUs, Xeon Phi)
12. Energy efficiency

■ Instruction set architecture (ISA)

- Portion of the computer visible to the programmer or compiler writer
- Contains e.g. opcode, operands, registers, addressing modes ...

■ Opcode (operation code)

- Specifies the operation of an instruction (usually binary or hexadecimal value)

■ Various types of operations

- | | |
|----------------------------|--|
| → Arithmetical and logical | Add, Subtract, AND, OR, XOR, Shift, ... |
| → Data transfer | Load, Store, Move |
| → Control flow | Jump, conditional Jump, Call, Return |
| → System | Interrupt calls |
| → Floating point | FP arithmetic: add, divide, multiply |
| → String operations | Move, Compare, Search |
| → Graphics: | Pixel and vertex operations, compression |
| → Decimal: | Decimal add, multiply |

■ 1960's: Complex Instruction Set Computers (**CISC**; IA-32,x86-64)

- Powerful, complex instructions
- Instructions are variable in length (1 - n bytes)

■ Mid 80's: Reduced Instruction Set computer (**RISC**; ARM, Sparc)

- Fixed instruction length
- Enables efficient pipelining & high clock frequencies
- Simple instructions e.g. $X=Y*Z$: —————→
- Nowadays: Superscalar RISC processors

```
LOAD Y→r0;  
LOAD Z→r1;  
MULT r0,r1 → r2;  
ST r2 → X;
```

■ In 2001: Explicitly Parallel Instruction Computing (EPIC)

- Potential to increase the number of parallel executed operations
- Only processor is the Intel Itanium

- **Typically variable instruction width and data sizes**
- **Processor specific multiples of one or two bytes**
- **Instructions can involve complex operations**
 - e.g. modifying and storing data in memory in one instruction
- **Instruction length depends on**
 - types of operands
 - number of operands
 - addressing mode (16bit, 32bit, 64bit)
- **One encoded instruction can vary in its total size and in case of misalignment requires more than one memory access**
 - Opcode can have 1 or more bytes

- Usually contains a limited number of instructions with fixed width
- RISC architecture tries to encode instructions, so that the position of opcode, operand and/or literals lies at the same bit positions
- Instructions can usually be handled in one clock cycle
- Simplify compiler design over large number of general purpose registers that can be used in varying context
- Clear distinction between data loading/storing and manipulation

■ X86 CPUs

→ X86-ISA is classical CISC

→ However

→ Since Pentium Pro: X86 codes get translated to internal RISC micro instructions before execution

→ “hybrid CISC/RISC CPUs”

■ Valid for recent Intel & AMD architectures

- **Byte ordering:** depending on the processor/ISA family a different byte ordering (Endianness) is implemented when values are stored in memory
- **Example: Storing the hexadecimal integer 0x12345678**

Memory Location:	00	01	02	03
Big Endian	12	34	56	78
Little Endian	78	56	34	12

- **Little Endian is quite common in x86/x64 type architectures**
- **Big Endian is for example used in MIPS, SPARC, PowerPC processors**
 - Some processors (ARM, MIPS, SPARC) can be switched explicitly from Big to Little Endian

1. Why supercomputers?
2. **Modern processors**
 - Stored-program computer architecture
 - Von Neumann architecture
 - Instruction set architecture
 - **General purpose cache-based microarchitecture**
 - **Pipelining**
 - Superscalarity
 - SIMD
 - Memory hierarchies
 - Cache
 - Prefetching
 - Cache mapping
 - Cache coherence
3. Basic optimization techniques for serial code
4. Data access optimization
5. Parallel computers
6. Parallelization and optimization strategies
7. Parallel algorithms
8. Shared-memory programming with OpenMP
9. Distributed-memory programming with MPI
10. Hybrid programming (MPI + OpenMP)
11. Heterogeneous architectures (GPUs, Xeon Phi)
12. Energy efficiency

- A single instruction usually takes several clock cycles to complete execution and data passes various stages

→ Example: multiplication of two FP arrays, e.g.



$$a = \vec{b}^T \vec{c} \quad \text{or} \quad a(k) = b(k)c(k), k \in \{1, \dots, n\}$$

Clock cycle	1	2	3	4	5	6	7	8
Seperate Mant/Exp	B(1) C(1)					B(2) C(2)		
Add Exponents		B(1) C(1)					B(2) C(2)	
Multiply mantissas			B(1) C(1)					B(2) C(2)
Normalize Result				A(1)				...
Insert sign					A(1)			

- **Problem: While instructions execute, most of the stages are idling and wait for input**
- **Pipelining overlaps multiple instructions in their execution**
 - Real world analogy: assembly line car manufacturing
 - Each station is doing something different
 - Each station working on a separate car
- **Pipelining increases the throughput but does not reduce an instructions execution time (latency)**

- **Idea: Subdivide complex operations (e.g. addition, multiplication) into several simpler fast suboperational stages**
 - Separate piece of hardware for each sub-stage
 - Instruction decode
 - Operand exponent alignment
 - Actual operation
 - Normalization
- **Makes short cycle time possible**
 - Floating point multiplication takes e.g. 5 cycles but
 - Processor can work on 5 multiplications simultaneously
 - One result is produced in each clock cycle

5-stage pipeline example

Clock cycle	1	2	3	4	5	...	N	N+1	N+2	N+3	N+4
Seperate Mant/Exp	B(1) C(1)	B(2) C(2)	B(3) C(3)	B(4) C(4)	B(5) C(5)	...	B(N) C(N)	 Wind-Down			
Add Exponents		B(1) C(1)	B(2) C(2)	B(3) C(3)	B(4) C(4)	...	B(N-1) C(N-1)	B(N) C(N)			
Multiply mantissas			B(1) C(1)	B(2) C(2)	B(3) C(3)	...	B(N-2) C(N-2)	B(N-1) C(N-1)	B(N) C(N)		
Normalize Result				A(1)	A(2)	...	A(N-3)	A(N-2)	A(N-2)	A(N)	
Insert sign	 Wind-Up				A(1)	A(2)	...	A(N-3)	A(N-2)	A(N-1)	A(N)

■ **First result available after 5 clock cycles**

→ latency or wind-up phase of pipeline

- **Pipeline must be filled with data → wind up-phase**
- **Efficient use of pipelines requires a large number of independent operations**
 - Instruction Level Parallelism (ILP): potential overlap among instructions
- **Requires some instruction scheduling by hardware or compiler**
 - Out-Of-Order Execution

- **Each stage in the pipeline should take the same time**
 - Expensive operations like the multiplication of mantissas can further be split into sub-stages
 - Cycle time is defined by the most complex operational stage

■ Besides arithmetical operations, the instruction execution itself is pipelined too

→ One instruction passes at least 3 steps

Clock cycle	1	2	3	4	5	...
Fetch Instruction	I(1)	I(2)	I(3)	I(4)	I(5)	...
Decode Instruction		I(1)	I(2)	I(3)	I(4)	...
Execute Instruction			I(1)	I(2)	I(3)	...

→ Conditional execution (branches/conditional jumps) can stall the pipeline

→ Speculative Execution

→ Each of the stages is further pipelined (e.g. Execute Inst. → Multiply pipeline)

Examples for pipeline lengths (estimates)



CPU	Integer Pipeline	Floating Point Pipeline
Intel Pentium III	12-17	25
Intel Pentium IV		31
R3010		6
R4400		8
R5000	5	5
R10000	5-6	7
IBM Power3		10
Intel Core2	12	
Intel Nehalem	14	
AMD K8	8	

Present-day micro processor
overall pipeline lengths:
~ 10 – 31 stages

Estimates of the potential speedup achieved by pipelining



- Assuming a type of operation with m stages
- Number of operations N

→ Execution without pipeline: $T_{seq} = mN$ cycles

Clock cycle	1	2	3	4	5	...	N	N+1	N+2	N+3	N+4
Seperate Mant/Exp	B(1) C(1)	B(2) C(2)	B(3) C(3)	B(4) C(4)	B(5) C(5)	...	B(N) C(N)				
Add Exponents		B(1) C(1)	B(2) C(2)	B(3) C(3)	B(4) C(4)	...	B(N-1) C(N-1)	B(N) C(N)			
Multiply mantissas			B(1) C(1)	B(2) C(2)	B(3) C(3)	...	B(N-2) C(N-2)	B(N-1) C(N-1)	B(N) C(N)		
Normalize Result				A(1)	A(2)	...	A(N-3)	A(N-2)	A(N-1)	A(N)	
Insert sign					A(1)	...	A(N-3)	A(N-3)	A(N-2)	A(N-1)	A(N)

m - 1
N

→ Pipelined Execution: $T_{pipe} = (N + m - 1)$ cycles

■ Speedup: $T_p = \frac{T_{seq}}{T_{pipe}}$

T_{seq} :	#cycles w/o pipeline
T_{pipe} :	#cycles w/ pipeline
T_p :	speedup
m :	#stages
N :	#operations

$$= \frac{mN}{(N + m - 1)} = \frac{m}{1 + \frac{m-1}{N}} \xrightarrow{N \rightarrow \infty} m$$

■ Throughput (=results per cycle):

$$\frac{N}{T_{pipe}} = \frac{N}{(N + m - 1)} = \frac{1}{1 + \frac{m-1}{N}} \xrightarrow{N \rightarrow \infty} 1$$

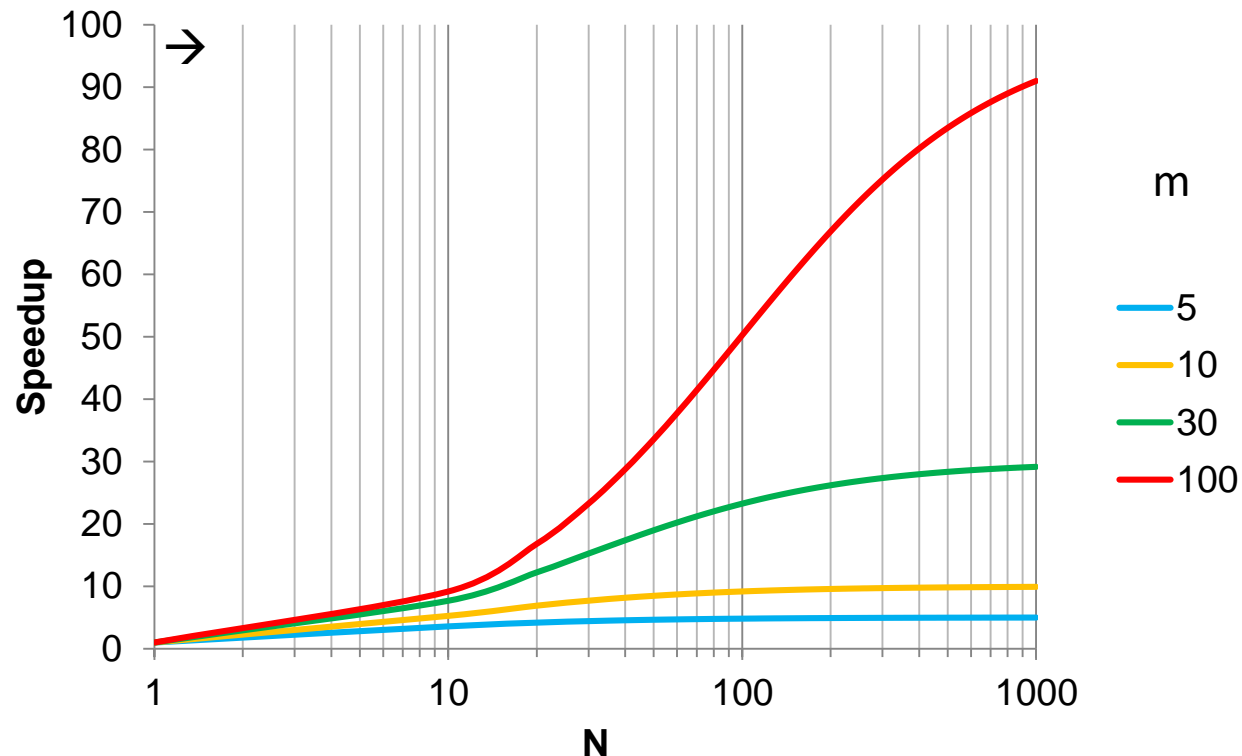
Examples of speedup

→ $N = 1$: $\frac{T_{seq}}{T_{pipe}} = \frac{m}{1 + \frac{m-1}{N}} = 1 \rightarrow$ No effect

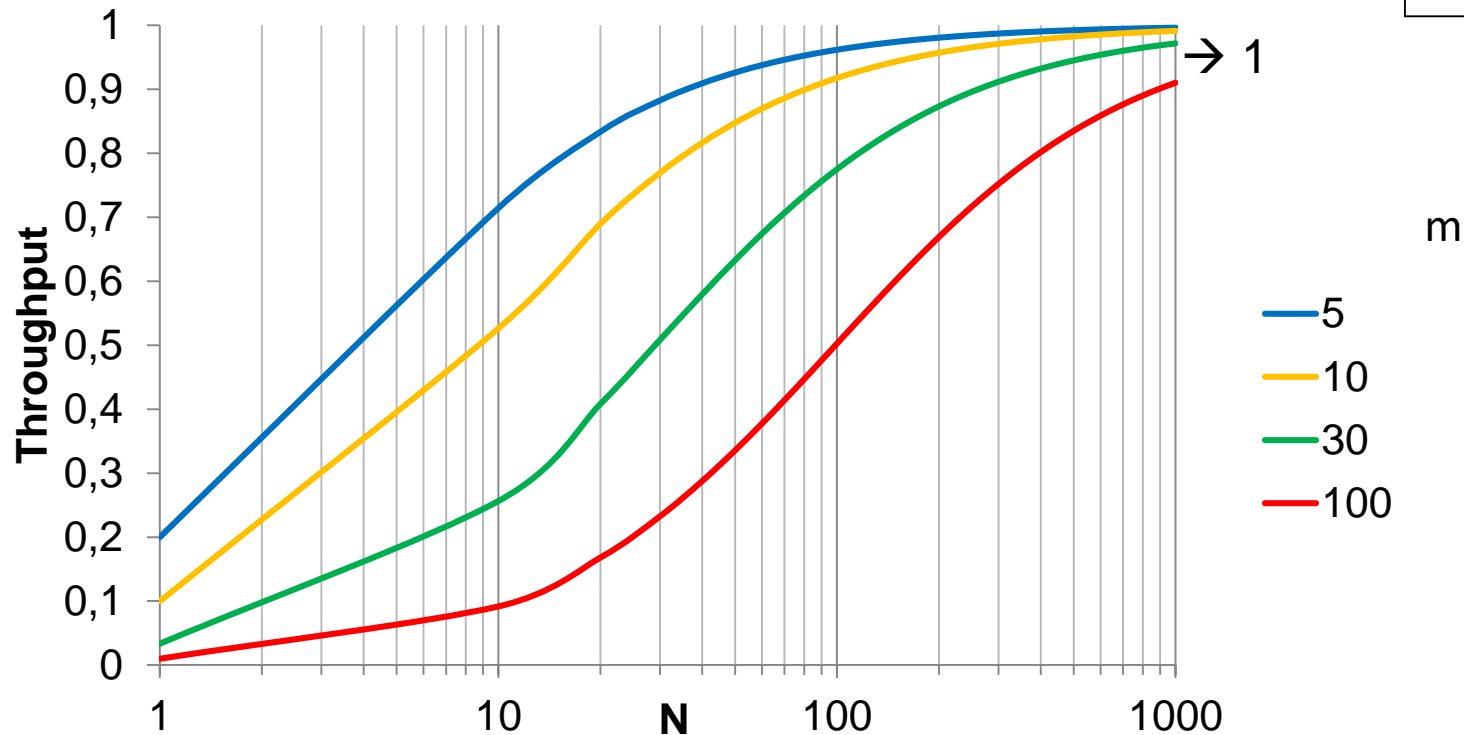
→ $N \gg m$: $\frac{T_{seq}}{T_{pipe}} \approx m$

→ The higher the number of pipelined operations, the nearer the speedup gets to m

T_{seq} : #cycles w/o pipeline
 T_{pipe} : #cycles w/ pipeline
 T_p : speedup
 m : #stages
 N : #operations



Throughput as function of pipeline stages



Peak performance is one instruction per cycle

→ CPUs with 1 instruction per cycle are called *scalar CPUs*

■ **Pipelining does not accelerate the execution time of an individual instruction**

- Overhead of controlling long pipelines even increases the individual execution time slightly
 - Pipeline register delay
 - Clock skew (irregular clocking, small deviances)

■ **Imbalance of different pipeline stages**

- Pipeline stall

■ **Instruction/ data needs to be independent**

- A situation in which the stream of instruction stalls the next instruction and thus prevents it from executing in the expected clock cycles is called a **hazard**


■ Structural hazards

- Limited resources are needed by two instructions at the same time, e.g. there may be only one memory interface that is occupied
- Resolve: Redundant hardware components

■ Data hazards

- An instruction depends on the result of a previous instruction in the pipeline
- Resolve: Out-of-Order Execution (OoOE, OOE)
 - Dependent instructions are skipped over to get future (independent) instructions
 - Correct order at the end must be satisfied

Example: Out of order Execution

<pre>c = a / b e = c + d g = d - f</pre>	<pre>// long latency DIV R2, R0, R1 // stalled for R2 ADD R4, R2, R3 // independent instr SUB R6, R3, R5</pre> 	<pre>DIV R2, R0, R1 SUB R6, R3, R5 ADD R4, R2, R3</pre>
--	--	---

- First and second instruction have a data dependency
- Division has a high latency due to the complexity of the operation
- Third instruction is data independent

■ Control hazards

→ A branch is encountered which depends on the result of a instruction in the pipeline

→ Resolve: Branch prediction:

essentially this is making educated guesses about the outcome of the depending operation. There are a lot of different approaches to this problem.

If the prediction proves wrong, a pipeline stall will occur and possibly the processor will have to revert the context (state of registers) to a previous step in time

■ CISC architectures

- Instructions are not of equal length
 - Fetch may take differing amounts of cycles depending on instruction
 - Decoding complexity differs
- Complexity of instructions differs
 - May include loads
 - May include stores
 - Instruction may include execution of processor microcode

- With the concept of pipelining in mind, programs have to be designed carefully or the increase in throughput may not be utilized

→ Assumption: Processor is able to detect data dependencies and stall execution until the required operands are available.

→ Assumption: Load (4 cycles), Mult (2 cycles), Store (2 cycles), Inc (1 cycle)

FORTRAN/C-Code:

```
//FORTRAN
DO I=1,N
  A(I)=A(I)*C
END DO
```

```
// C
for (i=0;i<N;++i)
  A[i]=A[i]*C;
```

Pseudo-Code:

```
Loop:
LOAD A[i]
MULT A[i], c, A[i]
STORE A[i]
INC i
JUMP Loop
```

Optimized Pseudo-Code:

```
Loop:
LOAD A[i+6]
MULT A[i+2], c, A[i+2]
STORE A[i]
INC i
JUMP Loop
```

Note: All these values are highly processor and architecture dependent and chosen arbitrarily, just for understanding purposes, in this example.

Pipeline utilization

96 cycles

19 cycles



CYCLE	NOT OPT	OPT (N=12)		
1	LOAD A[1]	LOAD A[1]		
2		LOAD A[2]		
3		LOAD A[3]		
4		LOAD A[4]		
5	MULT A[1], c, A[1]	LOAD A[5]	MULT A[1], c, A[1]	
6		LOAD A[6]	MULT A[2], c, A[2]	
7	STORE A[1]	LOAD A[7]	MULT A[3], c, A[3]	STORE A[1]
8		LOAD A[8]	MULT A[4], c, A[4]	STORE A[2]
9	LOAD A[2]	LOAD A[9]	MULT A[5], c, A[5]	STORE A[3]
10		LOAD A[10]	MULT A[6], c, A[6]	STORE A[4]
11		LOAD A[11]	MULT A[7], c, A[7]	STORE A[5]
12		LOAD A[12]	MULT A[8], c, A[8]	STORE A[6]
13	MULT A[2], c, A[2]		MULT A[9], c, A[9]	STORE A[7]
14			MULT A[10], c, A[10]	STORE A[8]
15	STORE A[2]		MULT A[11], c, A[11]	STORE A[9]
16			MULT A[12], c, A[12]	STORE A[10]
17	LOAD A[3]			STORE A[11]
18				STORE A[12]
19				

Load: 4 cy
Mult: 2 cy
Store: 2 cy

Pipeline
Prolog

Pipeline
Kernel

Pipeline
Epilog

pseudo code:
previous slide

- **Some processors require the compiler to consider that instructions are executed in a pipeline**
 - Instructions may not execute once at a time when there are dependencies
 - Operands that depend on results of nearby previous instructions are not available immediately after the calculating instruction → race conditions
 - The compiler may do software pipelining but...
 - These considerations require deep insight into the source code (data-dependencies) and architecture of the processor (latencies)
 - Nowadays most processors are able to determine such dependencies and stall instructions until their operands are ready or process other instructions in the meantime that are not dependent on current calculations
 - Out-of-Order Execution

■ Dependencies within loops may prevent efficient software pipelining

→ Software pipelining: interleaving of loop iterations to meet latency requirements (done by the compiler)

No Dependency

```
//FORTRAN
DO I=1,N
  A(I)=A(I)*c
END DO

//C
for(i=0;i<N;++i)
  A[i]=A[i]*c;
```

Real Dependency

```
//FORTRAN
DO I=2,N
  A(I)=A(I-1)*c
END DO

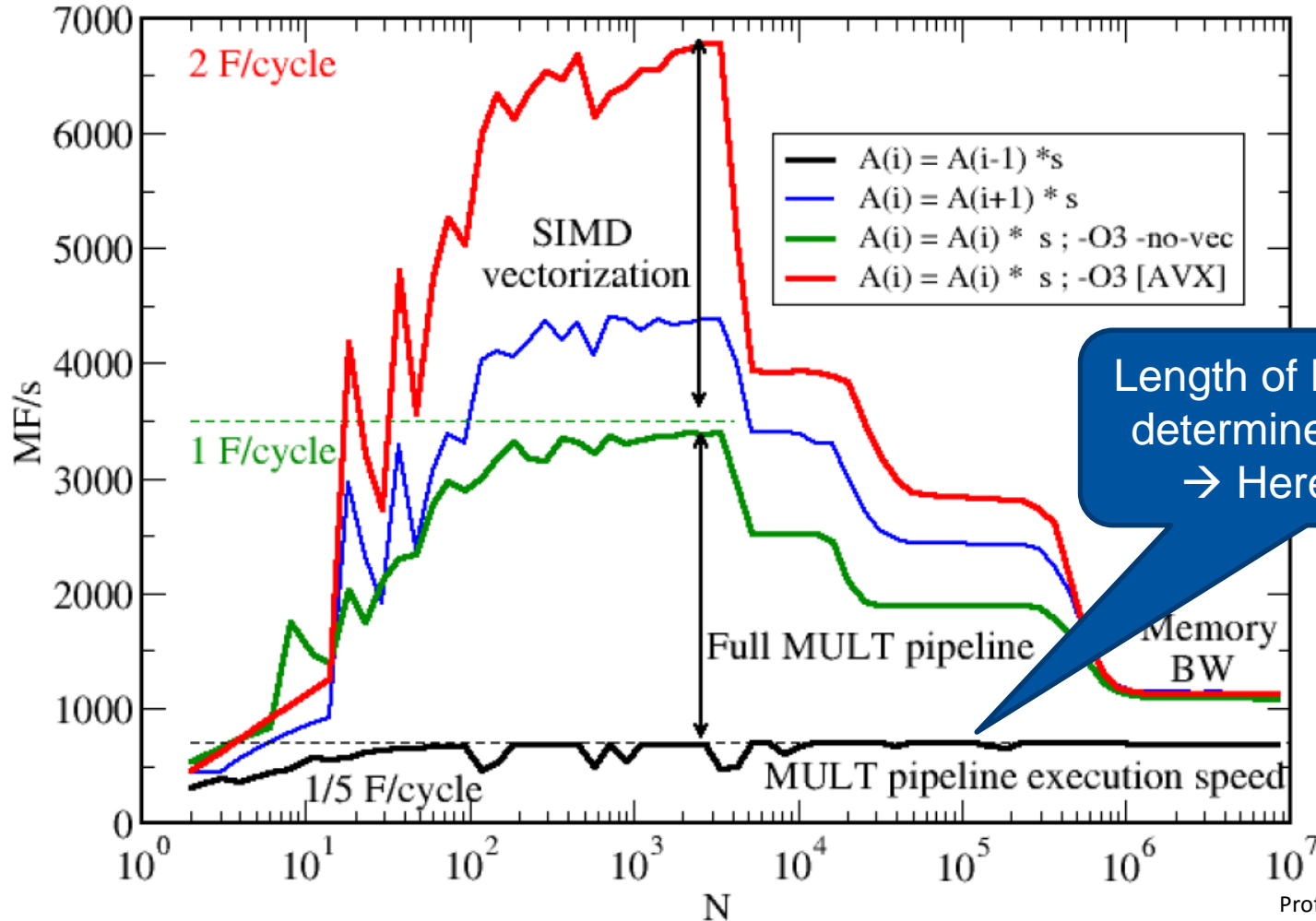
//C
for(i=1;i<N;++i)
  A[i]=A[i-1]*c;
```

Pseudo Dependency

```
//FORTRAN
DO I=1,N-1
  A(I)=A(I+1)*c
END DO

//C
for(i=0;i<N-1;++i)
  A[i]=A[i+1]*c;
```


Intel SandyBridge - 3.5 GHz fix - ifort 12.1.3 - UR8 Directive



Prof. Dr. G. Wellein, Dr. G. Hager,
M. Kreutzer Uni Erlangen-Nürnberg

- **Most ISA like the IA-32 family for example implement instructions for**
 - Integer and FP division
 - Square root (comes with sse)

- **Instructions for $\text{Log}(x)$, general exponentiation are not implemented!**
 - Exponentiation of 2 is though possible: Left shift a register n times

- **They are not pipelined!**
 - May require sequential iterative operations → number of cycles vary
 - Avoid complex operations like trigonometric functions as often as possible
 - Replace function with lookup-tables if you know the arguments range

■ Aliasing in C/C++

- Assuming the memory regions a and b point to do not overlap, the compiler could apply any software pipelining scheme it considers appropriate

```
void dosomething(double *a, double *b, int n)
{
    for(int i=0; i < n; ++i)
        a[i] = b[i-1]*42;
}
```

■ But: The C/C++ standard allows arbitrary aliasing of pointers

- The compiler must thus assume that a and b **do** overlap and generate code accordingly.
- **Certain optimization techniques are ruled out ab initio (e.g. SIMD Vectorization)**

- **The compiler can be told that there will be no aliasing in the code**

→ e.g. `-fno-nalias` (Intel C++)
e.g. `-fargument-noalias` (GCC)

→ Tells the compiler that no two arguments passed to any function will ever overlap

- **If you lie to the compiler your code may produce wrong results**
- **The assumption that aliasing may occur is one of the main reasons that C code generally tends to be slower than equivalent Fortran code (Aliasing is forbidden by the Fortran standard)**
 - Develop applications without aliasing and tell your compiler about it!

```
void dosomething(double *a, double
*b, int n)
{
  for(int i=0; i < n; ++i)
    a[i] = b[i-1]*42;
}
```

■ Inline functions and subroutines!

→ Obfuscating the compilers view on the code is generally counterproductive

```
double multiply(double b, double c)
{
    return b*c;
}

void calculation(...)
{
    for(int i=0; i<n; ++i)
        a[i]=multiply(b[i],c[i]);
}
```

→ When calling a function, arguments have to be passed on to the stack as well as a return address. Avoiding this overhead for fast-executing subroutines can improve performance

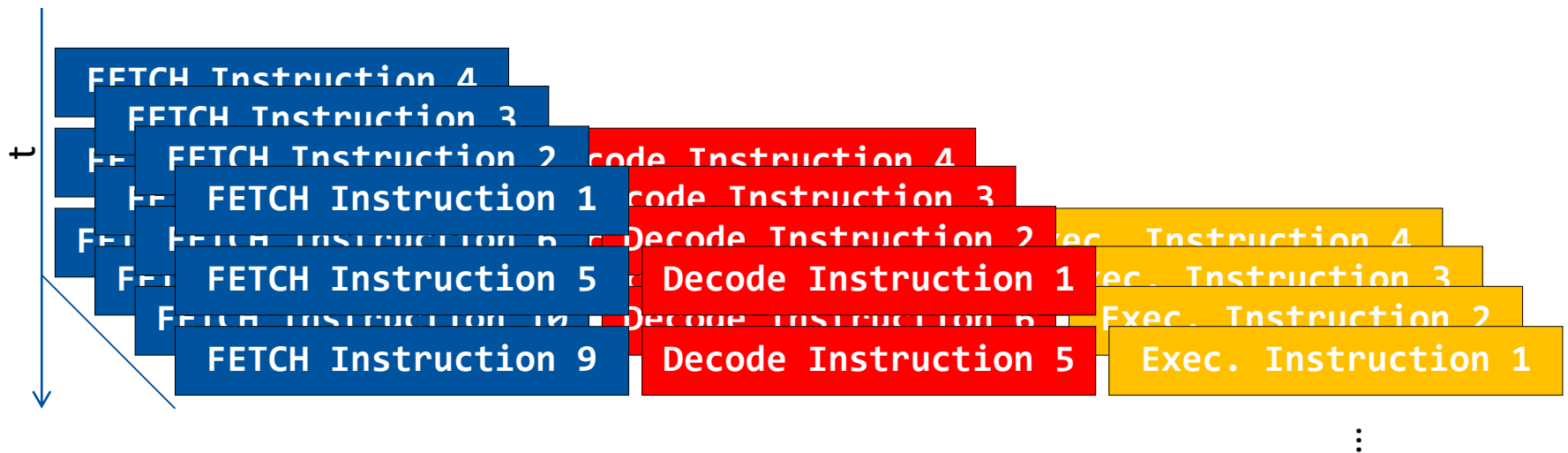
→ C/C++: keyword **inline**

→ Fortran: compiler options

1. Why supercomputers?
2. **Modern processors**
 - Stored-program computer architecture
 - Von Neumann architecture
 - Instruction set architecture
 - **General purpose cache-based microarchitecture**
 - Pipelining
 - **Superscalarity**
 - SIMD
 - Memory hierarchies
 - Cache
 - Prefetching
 - Cache mapping
 - Cache coherence
3. Basic optimization techniques for serial code
4. Data access optimization
5. Parallel computers
6. Parallelization and optimization strategies
7. Parallel algorithms
8. Shared-memory programming with OpenMP
9. Distributed-memory programming with MPI
10. Hybrid programming (MPI + OpenMP)
11. Heterogeneous architectures (GPUs, Xeon Phi)
12. Energy efficiency

- **A processor is said to be superscalar if it is designed to be capable of executing more than one instruction per cycle**
 - Multiple instructions can be fetched and decoded concurrently (3-6 up to now)
 - Multiple address/integer calculations in redundant hardware units
 - Multiple FP pipelines (typically 2 FP operations/cycle)
- **Superscalar processor provide additional hardware to execute multiple instructions per cycle**
- **Superscalarity is a variant of Instruction Level Parallelism (ILP)**
 - Out-Of-Order execution and compilers must cooperate to fully exploit the superscalar potential

- Multiple units achieve Instruction Level Parallelism
- Example: 4-way superscalar:



- Modern processors are 3-6 way superscalar

■ Example

→ e.g. 2 FMA (FP Add/Mul) Units cannot be busy with this task because there is a dependency on the summation variable t

Fortran	C/C++
<pre>DO i=1,n t = t + a(i) * b(i) END DO</pre>	<pre>for(i=0; i<n; ++i) t = t + a[i]*b[i];</pre>

■ Better

→ Modulo variable expansion

→ Process data in two independent streams; then summarize.

■ FP Math is not associative

→ Reordering changes result

That happens underneath (with optimizations enabled). Usually you don't have to write this code.

Fortran	C/C++
<pre>t1 = 0 t2 = 0 DO i=1,n,2 t1 = t1 + a(i) * b(i) t2 = t2 + a(i+1) * b(i+1) END DO t = t1 + t2</pre>	<pre>t1 = t2 = 0; for(i=0; i<n; i+=2) { t1 = t1 + a[i]*b[i]; t2 = t2 + a[i+1]*b[i+1]; } t = t1 + t2;</pre>

- 4 redundant decoding units
- Parallel units for Int./FP Arithmetic

→ 2x SSE ADD, MOVE

→ FP MUL & FP ADD in sep. units

→ 2x Address generation unit

→ Integer arithmetic

→ SSE MUL/DIV Unit

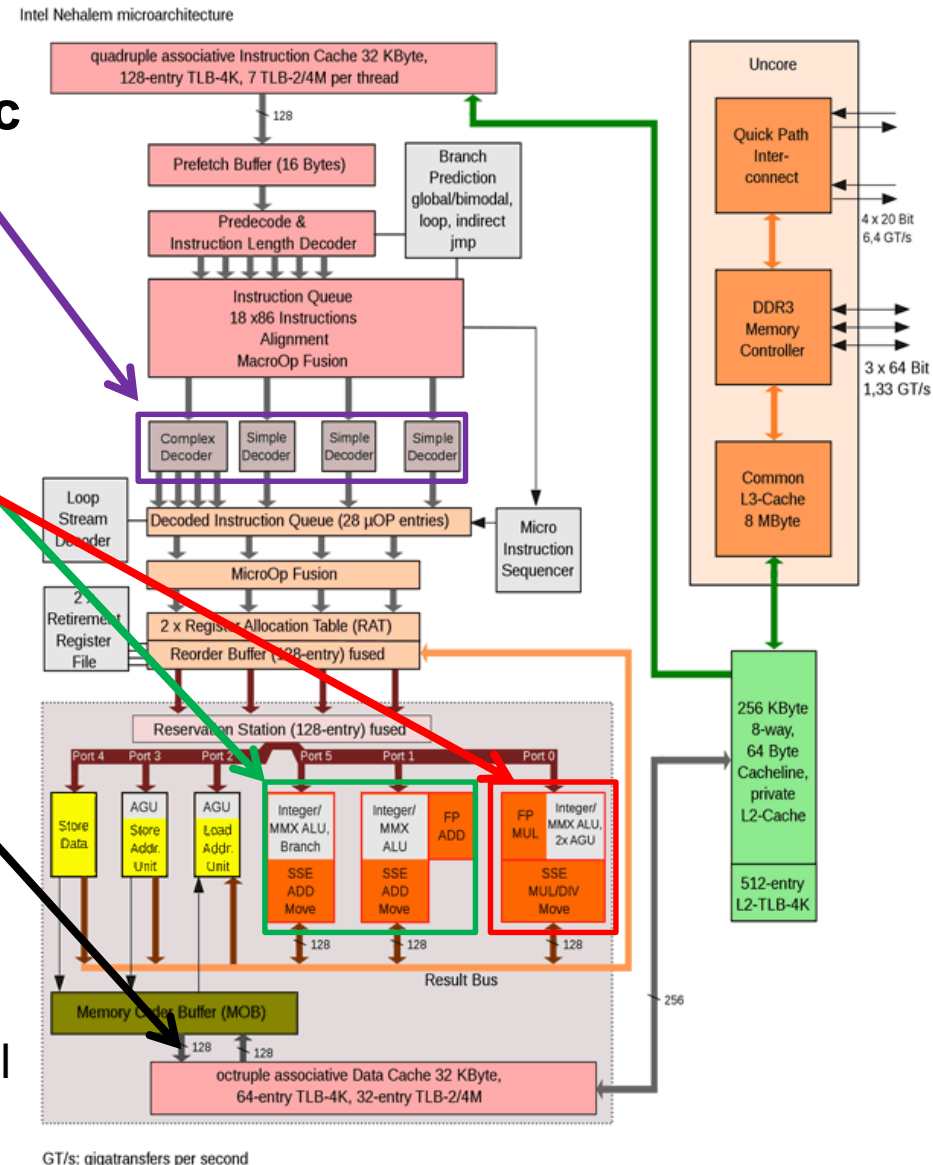
- Limitation: 128 bit per cycle load

→ Max perf. $A(i) = c1 + c2 \cdot B(i)$

→ 1/3 of max perf: $A(i) = A(i) + B(i) \cdot C(i)$

→ 6x 64bit load per pipeline step:

→ 3 cycles delay + 1 cycle Add/Mul



1. Why supercomputers?
2. **Modern processors**
 - Stored-program computer architecture
 - Von Neumann architecture
 - Instruction set architecture
 - **General purpose cache-based microarchitecture**
 - Pipelining
 - Superscalarity
 - **SIMD**
 - Memory hierarchies
 - Cache
 - Prefetching
 - Cache mapping
 - Cache coherence
3. Basic optimization techniques for serial code
4. Data access optimization
5. Parallel computers
6. Parallelization and optimization strategies
7. Parallel algorithms
8. Shared-memory programming with OpenMP
9. Distributed-memory programming with MPI
10. Hybrid programming (MPI + OpenMP)
11. Heterogeneous architectures (GPUs, Xeon Phi)
12. Energy efficiency

■ Idea: perform an identical operation on arrays/ vectors of data concurrently

→ A **single** instruction initiates **multiple** integer or floating point operations on “wide” registers (hardware-dependent)

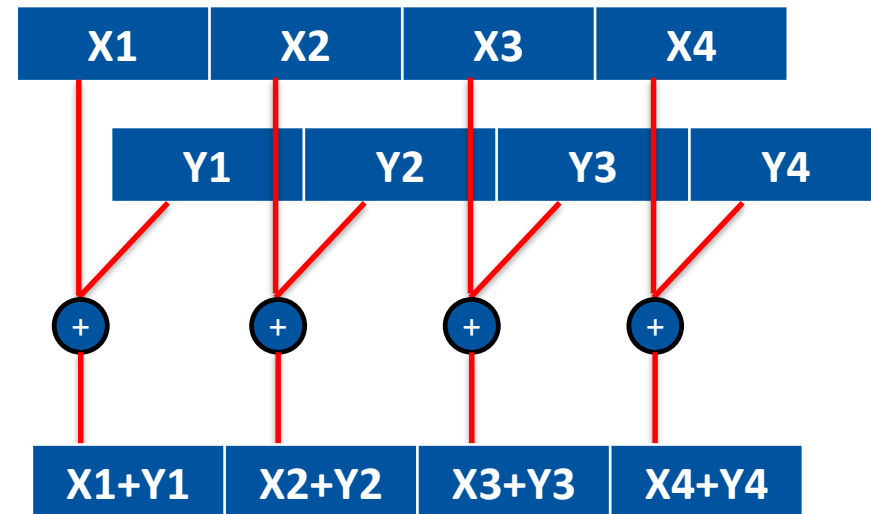
→ SSE: 128 bit registers

- 4 floats or
- 2 doubles

→ AVX: 256 bit registers

- 8 floats or
- 4 doubles

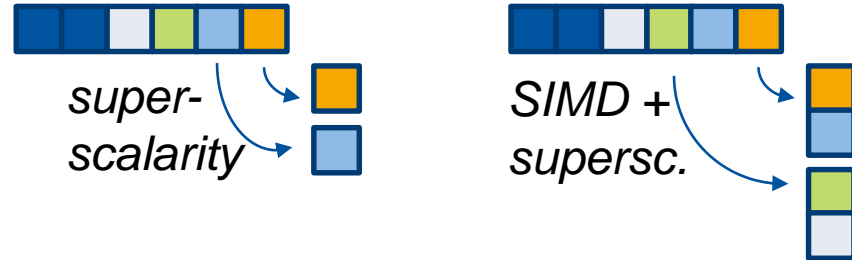
→ Special vector instructions
(that work on vector registers)



- **SIMD does not state if the operations are carried out truly parallel**

- SIMD over superscalarity

- SIMD with a single pipeline



- **Memory subsystem must be capable of keeping the pipelines busy**

- **SIMD requires vector-like, independent operations**

- **Compilers can produce “vectorized” code and furthermore utilize architectural resources capable of SIMD data processing**

■ Compiler optimization: Loop unrolling → enables SIMD processing

Fortran	C/C++
DO i=1,n C(i) = A(i) + B(i) END DO	for(i=0; i<n; ++i) C[i] = A[i]+B[i];

Loop unrolling

Fortran	C/C++
DO i=1,n,4 C(i) = A(i) + B(i) C(i+1) = A(i+1) + B(i+1) C(i+2) = A(i+2) + B(i+2) C(i+3) = A(i+3) + B(i+3) END DO	for(i=0; i<n; i+=4) { C[i] = A[i]+B[i]; C[i+1] = A[i+1]+B[i+1]; C[i+2] = A[i+2]+B[i+2]; C[i+3] = A[i+3]+B[i+3]; }

Machine-code

“Assembler”

```
VLOAD R0 ← &A[i]  
VLOAD R1 ← &B[i]  
VADD R0,R1  
VSTORE → &C[i]
```

Note: Pointer aliasing may prevent compiler from doing this kind of optimization

■ Use Structure of Arrays (SoA) instead of Array of Structures (AoS)

→ Color structure

```
struct Color{ //AoS
    float r;
    float g;
    float b;
}
Color* c;
```

```
struct Colors{ //SoA
    float* r;
    float* g;
    float* b;
}
```



Strided access usually prevents vectorization.



SoA enables contiguous data access and thereby vectorization.

■ Compiler options related to vectorization (here: Intel):

→ -O2

→ -[a]xSSE2, -[a]xSSE3, -[a]xSSE4.1, -[a]xSSE4.2, -[a]xAVX

→ -vec-report=<n>

■ In code (source code directives):

#pragma vector always !DEC\$ VECTOR ALWAYS

→ Vectorize no matter if it seems inefficient

#pragma novectorize !DEC\$ NOVECTORIZE

→ Do not vectorize even if it may be possible

#pragma vector aligned !DEC\$ VECTOR ALIGNED

→ Assume that memory is aligned in 16-byte boundaries (use with caution)

OpenMP 4.0
introduces
#pragma omp simd
(and aligned clause)
for vectorization.

■ If compiler fails to vectorize (or for possible improvement of automatic vectorization)

→ SIMD can be used directly by the programmer:

→ Directives supported by compiler/ programming model

→ High level C++ classes that map to SIMD functionality
(e.g. vec4 in OpenGL, float8 in OpenCL)

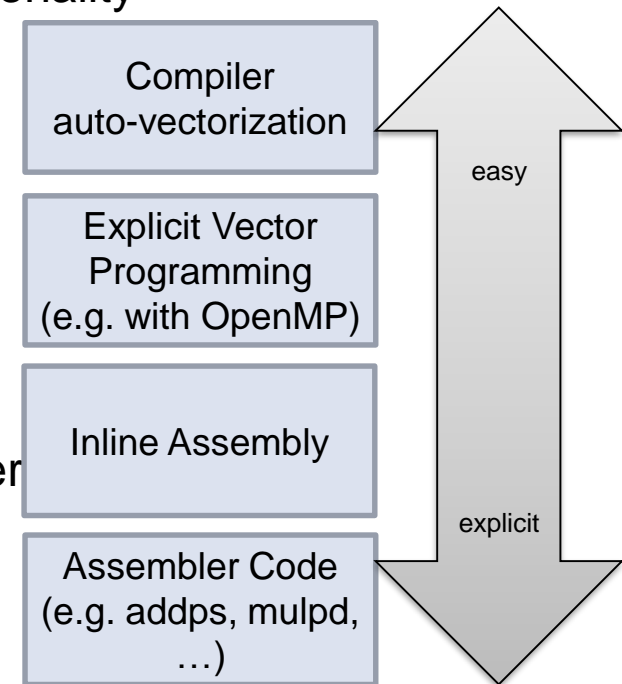
→ Compiler Intrinsics:

→ Map directly to assembler instructions
→ programmer can still work within the
programs context (local variables, ...)

→ Assembly language → experienced programmer

→ Highly machine specific

→ No optimization



■ Example: Intel/AMD x64 based processor's SIMD capabilities

- 1 FP multiply & 1 FP add pipeline can run in parallel → 1 FP ops/cycle each
 - SSE → operates on 128 bit registers → 2 double precision FP ops/cycle
 - AVX → operates on 256 bit register → 4 double precision FP ops/cycle
- **4 (or 8) FP ops/cycle (1 MULT & 1 ADD on 2 (4) operands)**

■ For n pipelines at a clockrate of f_c (double precision, one core):

- SSE: $2nf_c$ Flop/s
- AVX: $4nf_c$ Flop/s

■ E.g. a 3 GHz CPU with 2 FP pipelines (one core):

- SSE: 12 GFlop/s double precision, (24 GFlop/s single precision)
- AVX: 24 GFlop/s double precision, (48 GFlop/s single precision)

■ What is an ISA?

- What are differences in ISAs?
- What are the main differences between RISC & CISC?
- What do we have today?

■ What is the basis of today's processors?

- What are the corresponding bottlenecks?
- How can we overcome these?

■ What is ILP?

- What is its influence on today's computers (over time)?

■ Pipelining

- How does it work?
- How to compute possible speedup & throughput?
- Problems

■ Superscalarity

- What is a superscalar processor?

■ SIMD

- How does it work?
- What are recent vector register widths?
- What to think about if code vectorization is desired?

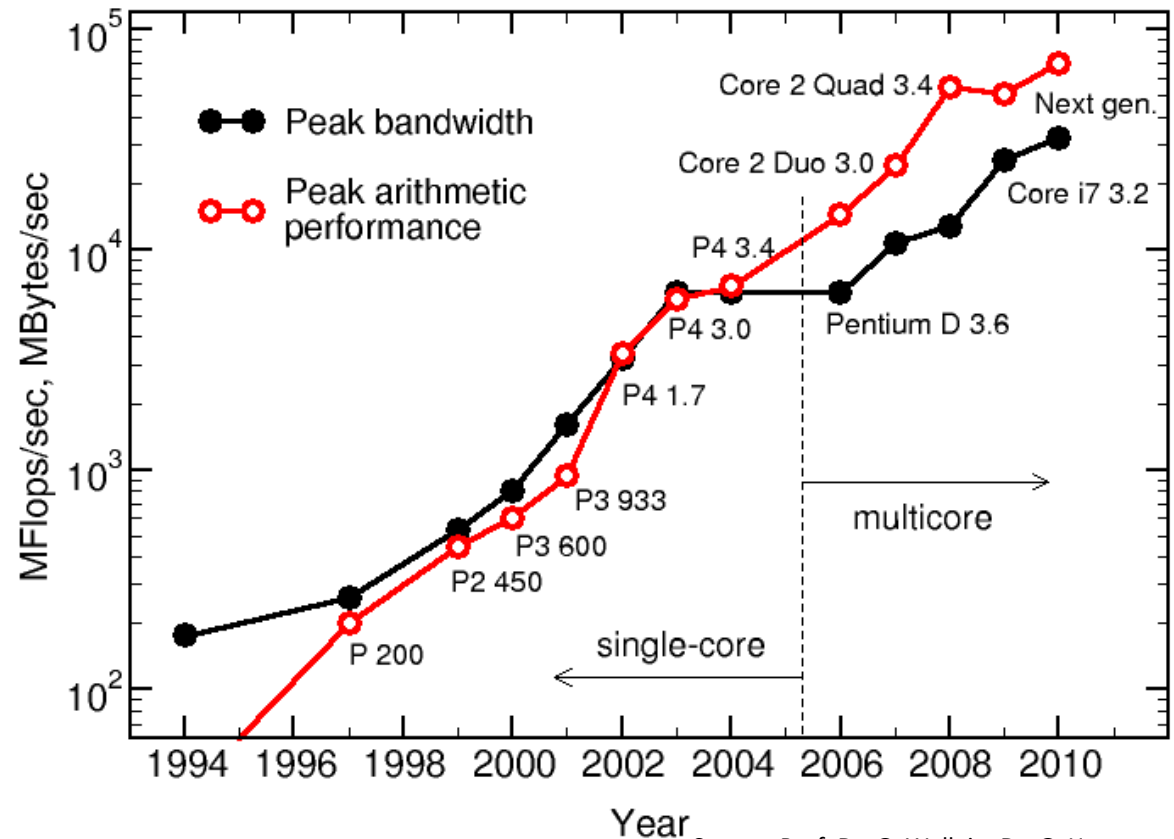
1. Why supercomputers?
2. **Modern processors**
 - Stored-program computer architecture
 - Von Neumann architecture
 - Instruction set architecture
 - General purpose cache-based microarchitecture
 - Pipelining
 - Superscalarity
 - SIMD
 - **Memory hierarchies**
 - **Cache**
 - Prefetching
 - Cache mapping
 - Cache coherence
3. Basic optimization techniques for serial code
4. Data access optimization
5. Parallel computers
6. Parallelization and optimization strategies
7. Parallel algorithms
8. Shared-memory programming with OpenMP
9. Distributed-memory programming with MPI
10. Hybrid programming (MPI + OpenMP)
11. Heterogeneous architectures (GPUs, Xeon Phi)
12. Energy efficiency

■ CPU can process data very fast (~100 GFlop/s, Core 2 quad)

→ Access registers without delay

→ But main memory is (still) very slow (~13 GByte/s per DDR3 channel) in comparison

■ Data bandwidth does usually not speed up at the same rate as processor development



Source: Prof. Dr. G. Wellein, Dr. G. Hager, M. Kreutzer Uni Erlangen-Nürnberg

■ Bandwidth

→ Maximum data rate that can be held during the transfer from memory to the CPU. Unit is MB/s, GB/s, etc.

■ Latency

→ Time that memory needs to prepare for a data transfer to the CPU. Also described as the time a zero-byte message delivery takes.

■ Memory latency hits performance (100-1000 cycles latency)

→ Under this conditions the CPU cannot be kept busy continuously

→ At considerable cost faster memory is available

→ Caches are a small amount of that fast memory

- **Requires fast access to operands**

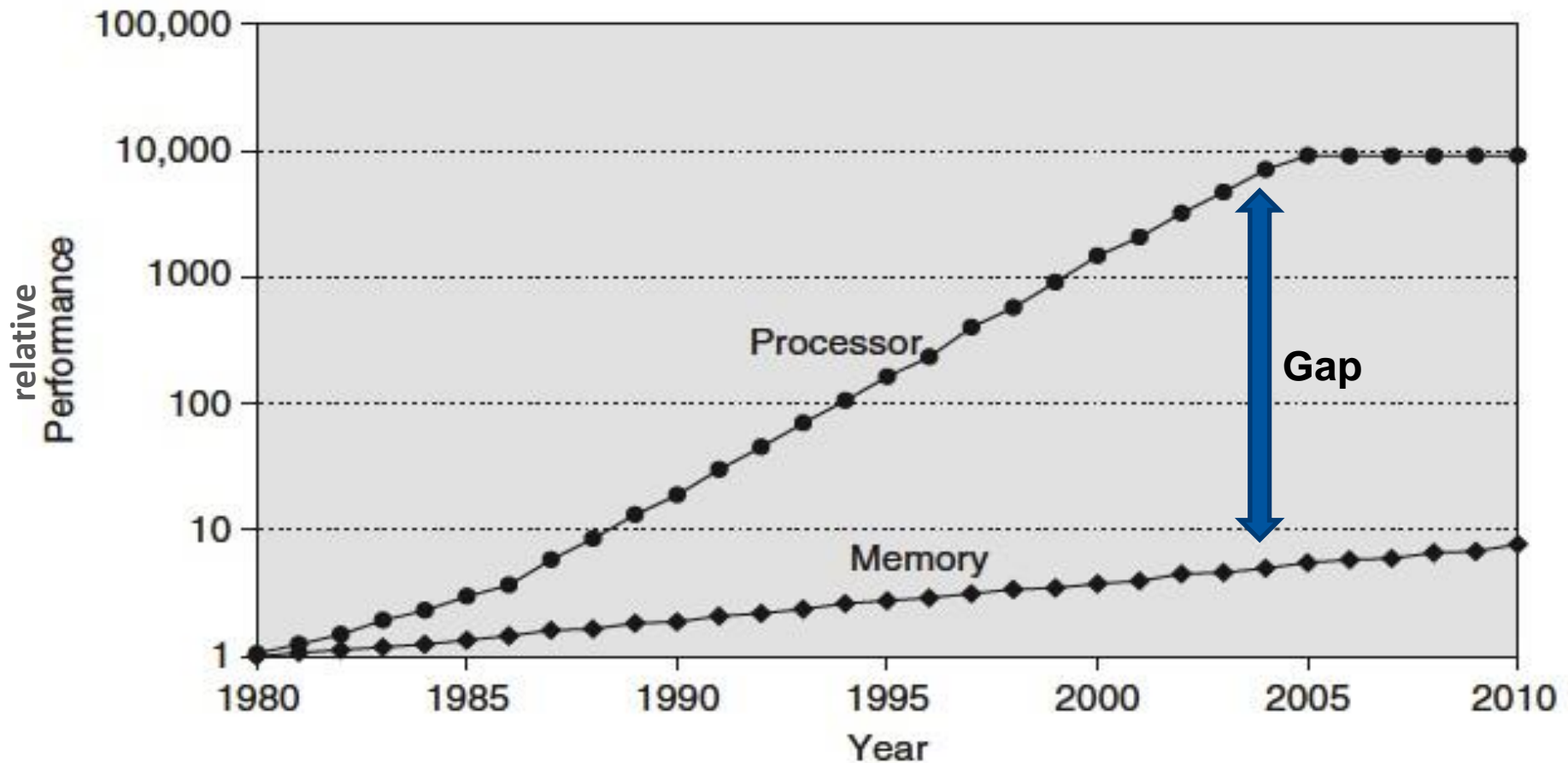
- Executing out-of-order can only hide small latencies

- **Might need new data each cycle**

- Double precision addition and store requires 24 Bytes/s per FLOP/s
so at e.g. 1 GFLOP/s, 24 GByte/s would be needed

- **But: Processor often works on small datasets and has repeating access patterns**

- Gap between processor performance (single core) and memory access speed



■ Memory

- DIMM bandwidth is increasing with DDR, DDR2 and DDR3 but cannot keep up
- Increasing number of memory channels to compensate
- Capacity grows according to Moore's law

■ Cache

- Low capacity (L1: 16-64 Kbyte, L3 up to 32 MB)
- High speed (multiple transfers per cycle)
- Usually organized in multiple levels (L1, L2,...)
- Commonly integrated into the CPU die
- Can alleviate the effects of the “DRAM gap” by holding “copies” of recently requested data

■ Usually caches are distinguished in

- Unified Caches → Store data and instructions
- Data cache → Stores only data
- Instruction cache → Stores only instructions
- Instruction Trace Cache / Micro-OP cache → Stores decoded instructions

■ Additionally there are multiple levels of cache of varying size in modern processors

- L1 usually separated into Instruction and Data Cache and one per core
- L2, L3 caches (unified) with increasing size and may or may not be shared by cores
- Speed-capacity trade-off

■ In modern computer design, memory is divided into different levels:

→ Registers

→ Caches

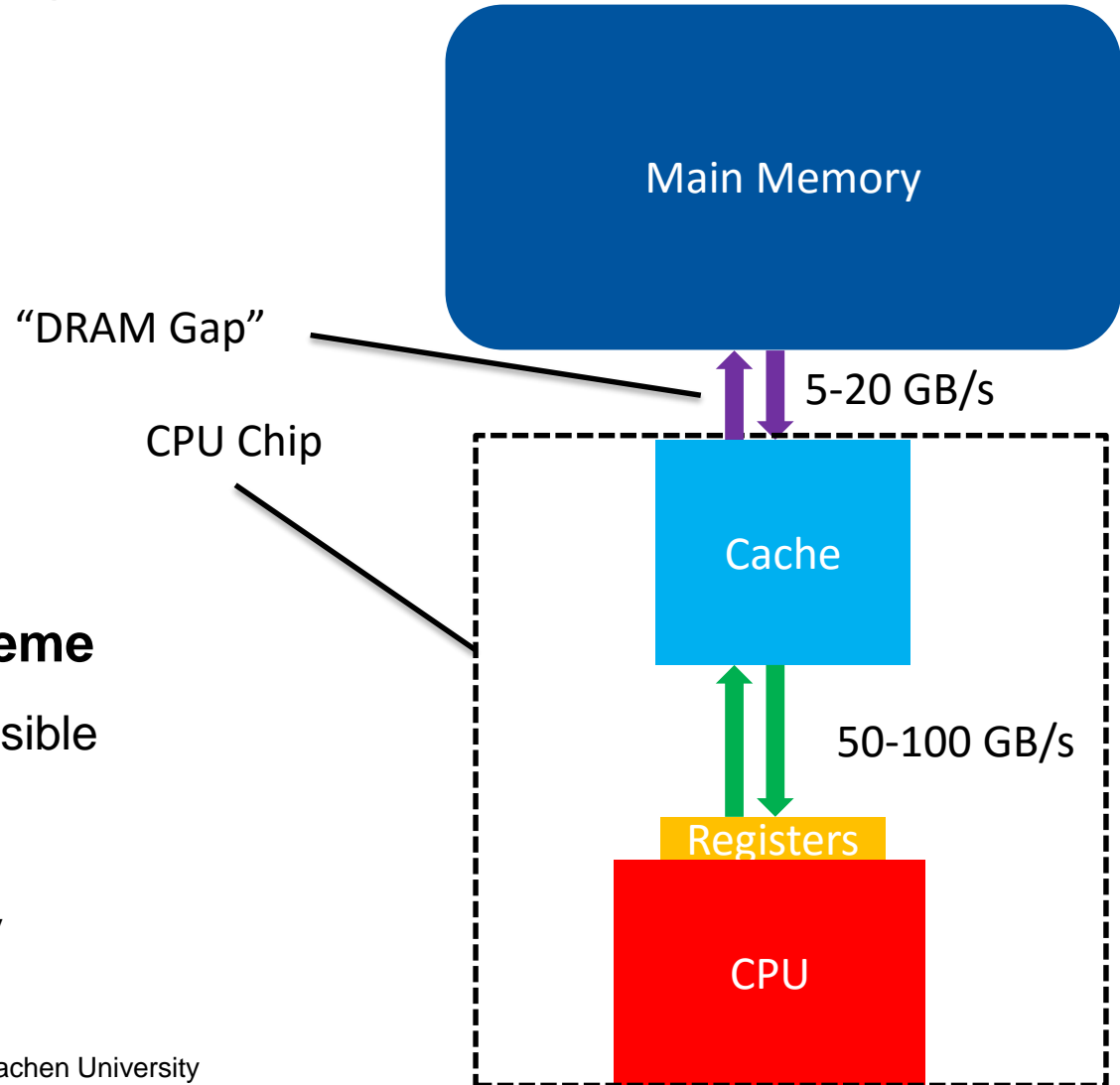
→ Main Memory

■ Access follows the scheme

→ Registers whenever possible

→ Then the cache

→ At last the main memory



1. CPU issues load request to transfer data item into a register
2. Cache logic checks if data is already in a cache
3. If Data item is in cache → **Cache-Hit**
4. If Data Item is not in cache → **Cache Miss**
 - Data item is loaded from main memory and cache holds copy of it
 - If the cache is full, another item needs to be evicted in favor of the currently loaded

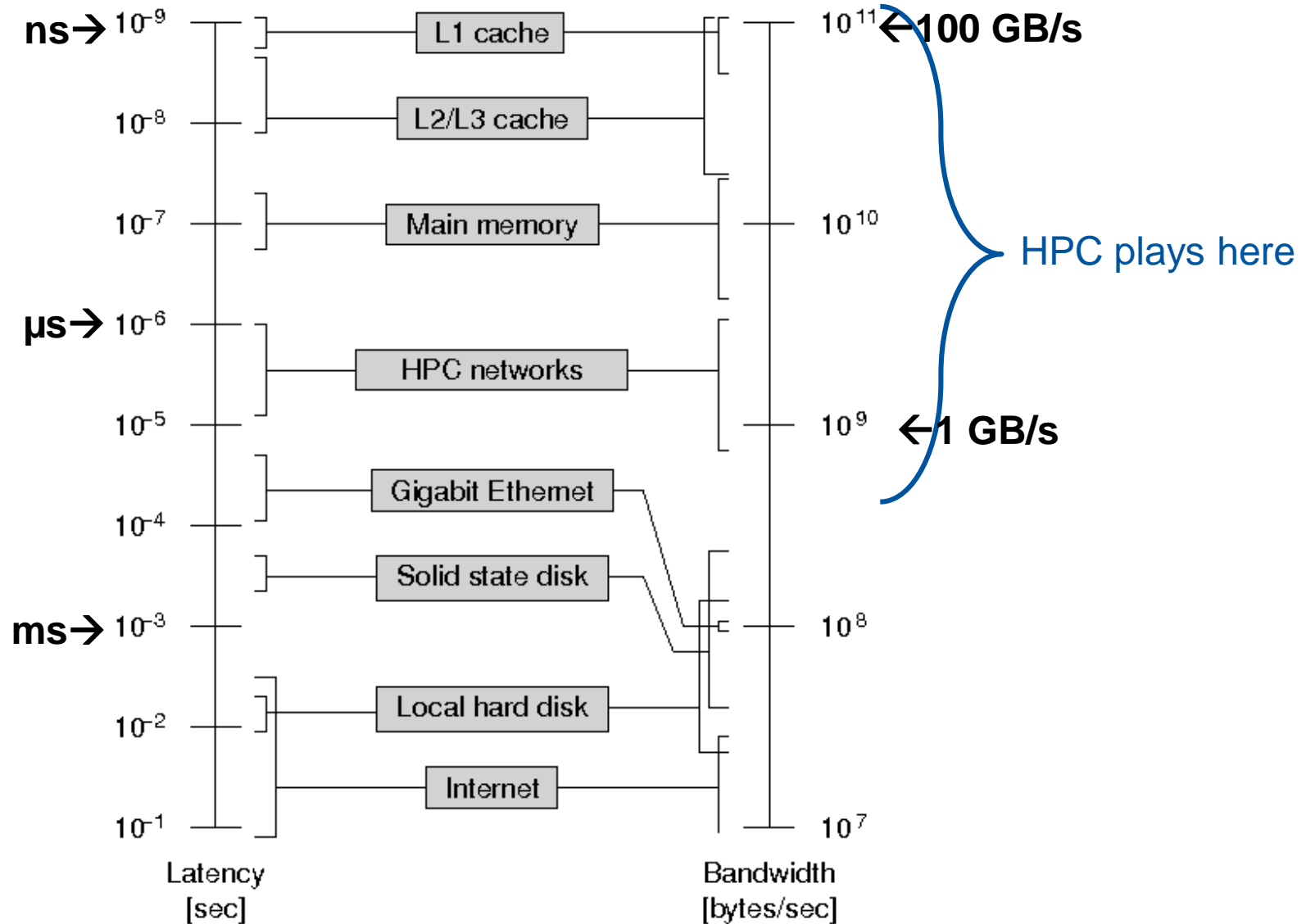
■ **Recap: Two most important quantities that are related to performance for the memory subsystem:**

- **Latency** (T_{lat}): The time memory needs to prepare for a data transfer to the CPU. Also described as the time a zero-byte message delivery takes.
- **Bandwidth** (B): Maximum data rate that can be held during the transfer from memory to the CPU. Unit is MB/s, GB/s, etc.

■ **Overall transfer time: $T = T_{lat} + \frac{N}{B}$,**

where N is the amount of data to be transferred from main memory

Latency & bandwidth comparison of different mediums



Main memory bandwidth measured by Stream benchmark



				1 thread
Function	Best Rate MB/s	Avg time	Min time	Max time
Copy:	9197.3	0.696029	0.695860	0.696443
Scale:	8314.0	0.772881	0.769786	0.797328
Add:	10029.1	0.959716	0.957212	0.968983
Triad:	10003.7	0.959811	0.959647	0.960055

- **Benchmarks can measure the sustainable cache bandwidth**

→ Stream

- **Stream measures 4 operations**

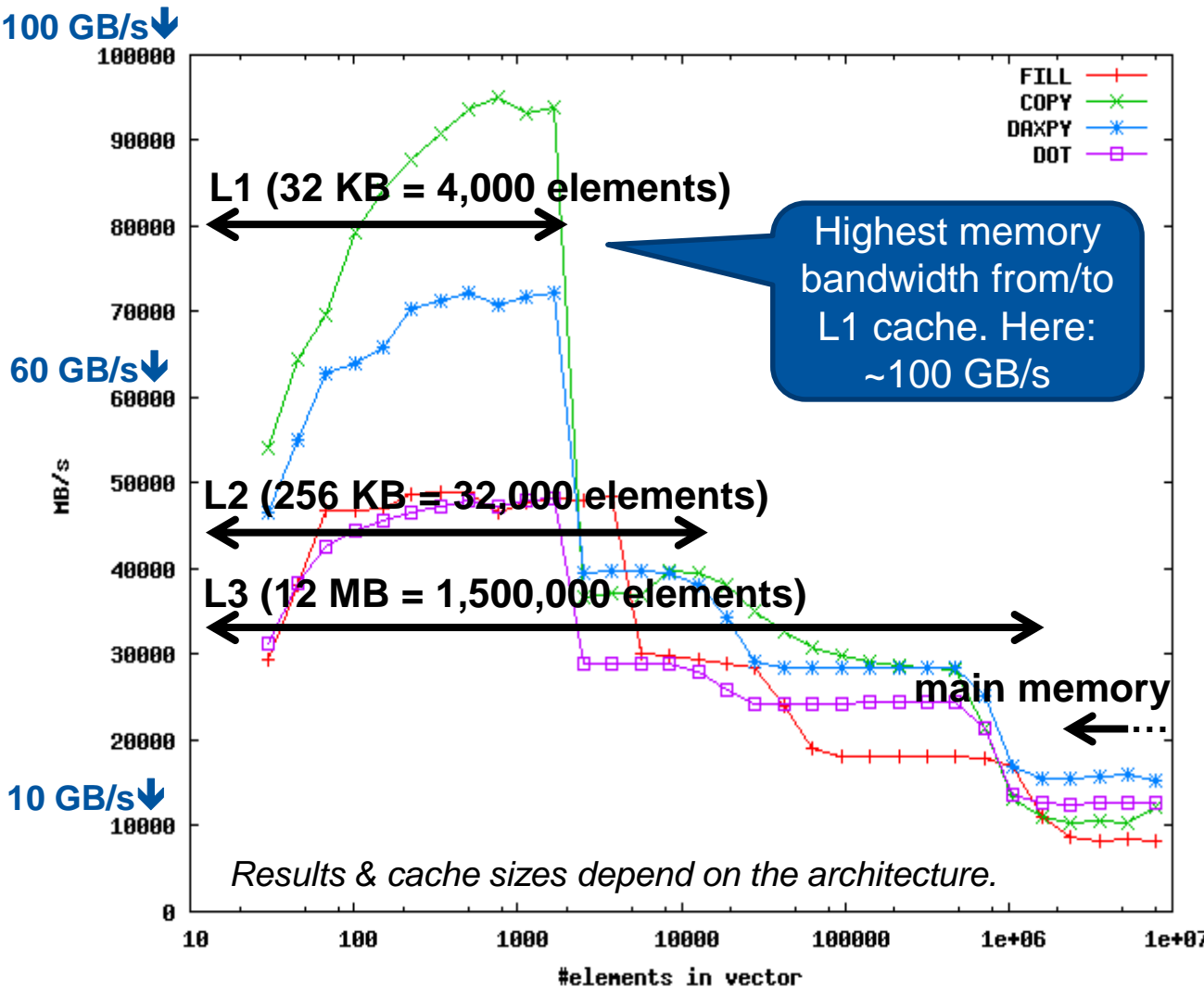
copy: $\vec{a} = \vec{b}$
scale: $\vec{a} = s \cdot \vec{b}$
add: $\vec{a} = \vec{b} + \vec{c}$
triad: $\vec{a} = \vec{b} + s \cdot \vec{c}$

				6 threads (balanced)
Function	Best Rate MB/s	Avg time	Min time	Max time
Copy:	35691.8	0.182867	0.179313	0.190822
Scale:	34616.7	0.188637	0.184882	0.197780
Add:	41735.5	0.234548	0.230020	0.241814
Triad:	40153.1	0.242858	0.239085	0.250663

				12 threads
Function	Best Rate MB/s	Avg time	Min time	Max time
Copy:	41439.0	0.154888	0.154444	0.155224
Scale:	39805.2	0.161369	0.160783	0.161781
Add:	41120.2	0.233850	0.233462	0.234490
Triad:	40187.6	0.239753	0.238880	0.240569

6 threads can saturate the memory bandwidth of this 2-socket server: 40 GB/s

Cache levels measured with the Stream2 benchmark *(compare exercise 1)*



- Benchmarks can measure the sustainable cache bandwidth
→ Stream2
- Stream2 measures 4 operations w/ different read/write properties

fill: $\vec{a} = q$
 copy: $\vec{a} = \vec{b}$
 daxpy: $\vec{a} = \vec{b} + q \cdot \vec{c}$
 dot: $sum = sum + \vec{a} \cdot \vec{b}$

■ Typical values for modern microprocessors (main memory):

→ $T_{lat} = 100\text{ns}$; $BW = 4\text{GB/s}$

→ e.g. transfer of 8 bytes (one double precision FP value):

$$\rightarrow 100\text{ ns} + \frac{8\text{ byte}}{4 \cdot 1024^3 \text{ byte/s}} = 100\text{ ns} + 1.86 \cdot 10^{-9} \approx 102\text{ ns}$$

$$\rightarrow \text{Transfer rate: } 8\text{ Byte in } 102\text{ns: } \frac{8\text{ byte}}{102\text{ ns}} = 0.078 \frac{\text{byte}}{\text{ns}} = \mathbf{0.073\text{ GB/s}}$$

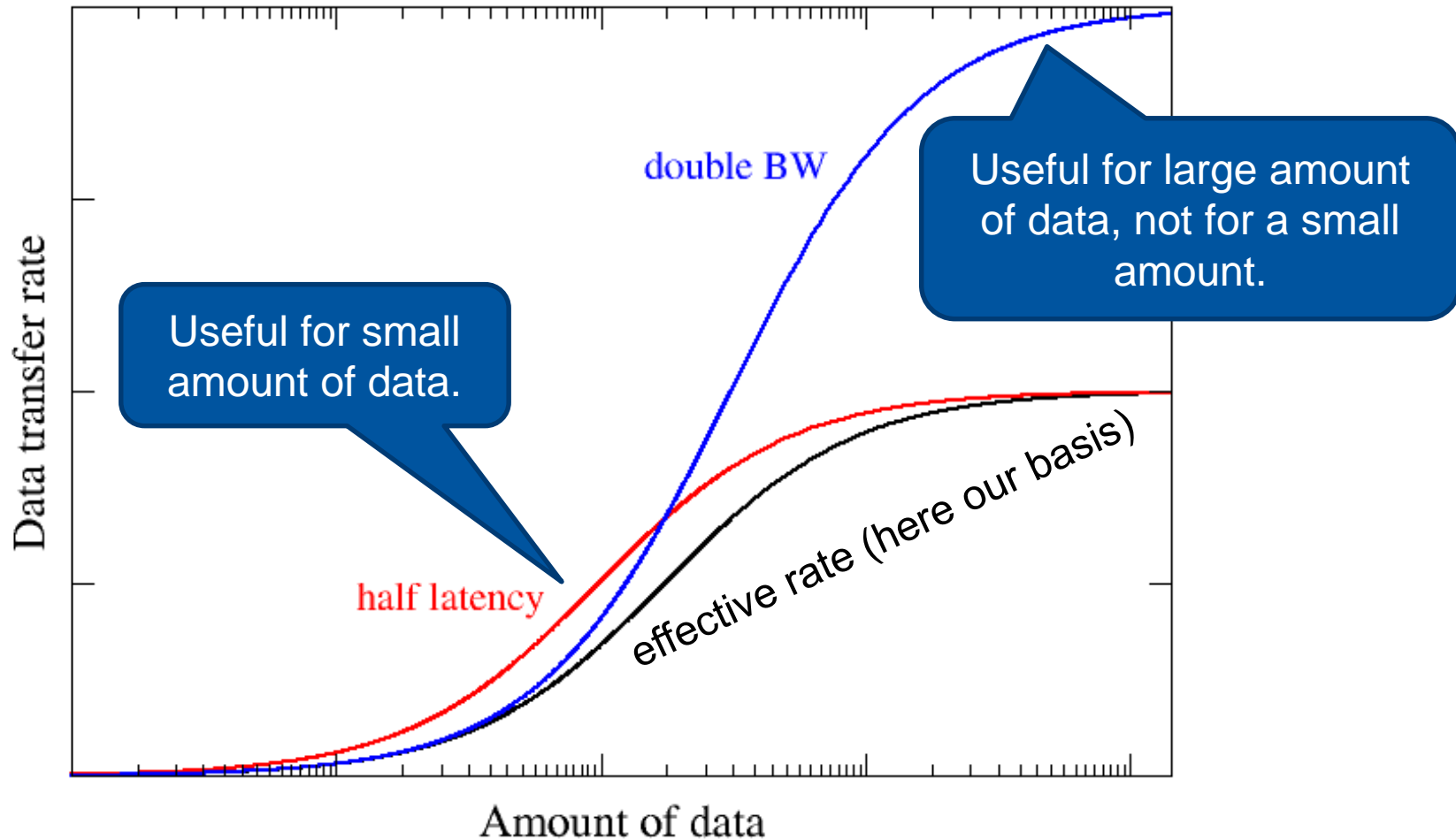
■ Caches are divided into cache lines of length C_L

→ A memory request always transfers a whole cache line from main memory if it can not be satisfied from cache. Typical value for C_L is 64 byte:

→ Time for cache line transfer: 114ns

→ Transfer rate: $\frac{64\text{ byte}}{114\text{ ns}} = \mathbf{0.63\text{ GB/s}} \rightarrow$ Cache lines are insufficient in hiding memory latency

■ Example for transfer rate with varying latency and bandwidth:



■ Principle of locality

→ Program accesses only a relatively small portion of the address space over time

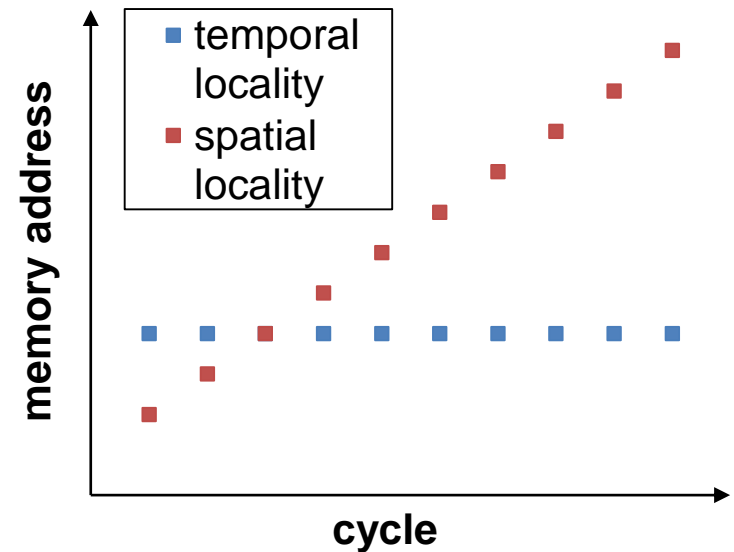
■ Two types of locality

→ Temporal locality

→ An item that was referenced will soon be referenced again

→ Spatial locality

→ Items that are close to a recently referenced item are likely to be referenced in near future



■ Example

→ Consider the following program (vector norm)

C/C++	Fortran
<pre>for(i=0; i<N; ++i) s += a[i]*a[i];</pre>	<pre>DO i=1,N s = s + a(i)*a(i); END DO</pre>

■ Its streaming form is clearly spatial locality

→ Cache lines are optimal for contiguous access (stride 1) → Streaming

→ Non contiguous access reduces the performance (worst case, stride= C_L)

- **Hit rate β :**

- Percentage of memory accesses that can be resolved from cache because there was a recent load to the same address

- **Miss rate: $(1 - \beta)$**

- **Access time: Memory (T_m), Cache (T_c)**

- **Average access time:**

- $T_{av} = \beta T_c + (1 - \beta) T_m$

- **Let $\tau = \frac{T_m}{T_c}$ denote the speedup due to using caches then the average access performance gain can be expressed as:**

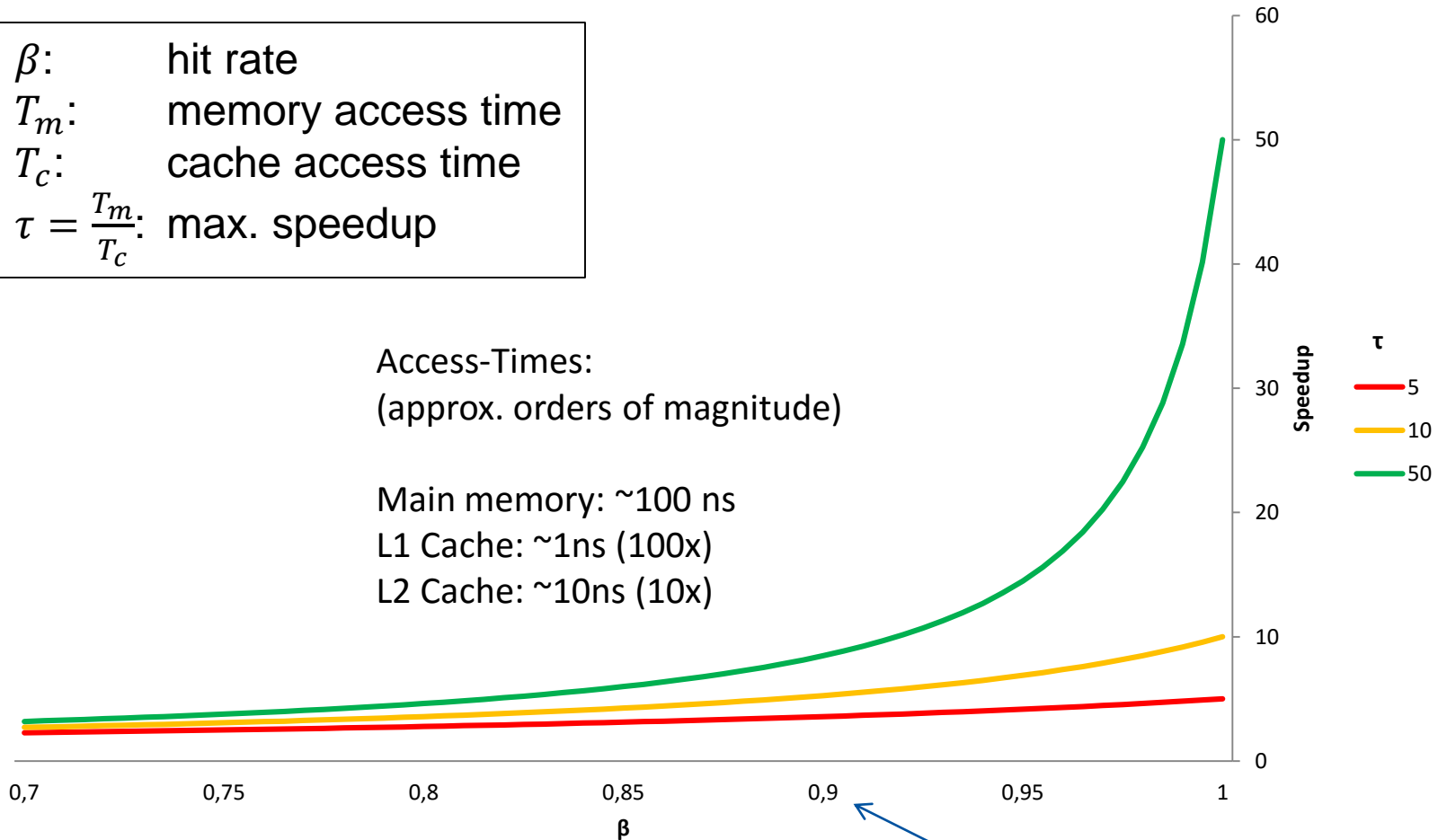
$$G(\tau, \beta) = \frac{T_m}{T_{av}} = \frac{\tau T_c}{\beta T_c + (1 - \beta) \tau T_c} = \frac{\tau}{\beta + \tau(1 - \beta)}$$

■ Potential gain in access time as function of cache speedup

β : hit rate
 T_m : memory access time
 T_c : cache access time
 $\tau = \frac{T_m}{T_c}$: max. speedup

Access-Times:
(approx. orders of magnitude)

Main memory: ~100 ns
L1 Cache: ~1ns (100x)
L2 Cache: ~10ns (10x)



e.g. one load and 9 consecutive accesses that can be satisfied from cache

1. Why supercomputers?
2. **Modern processors**
 - Stored-program computer architecture
 - Von Neumann architecture
 - Instruction set architecture
 - General purpose cache-based microarchitecture
 - Pipelining
 - Superscalarity
 - SIMD
 - **Memory hierarchies**
 - Cache
 - **Prefetching**
 - Cache mapping
 - Cache coherence
3. Basic optimization techniques for serial code
4. Data access optimization
5. Parallel computers
6. Parallelization and optimization strategies
7. Parallel algorithms
8. Shared-memory programming with OpenMP
9. Distributed-memory programming with MPI
10. Hybrid programming (MPI + OpenMP)
11. Heterogeneous architectures (GPUs, Xeon Phi)
12. Energy efficiency

- The program executions stalls until the memory request is finished



- **Cache concept is adequate for reducing latency on items that are already in the cache**

- But not sufficient to hide the main memory's latency

- **Prefetching**

- Mechanism to issue consecutive loads on the memory bus s.th. it is kept busy continuously

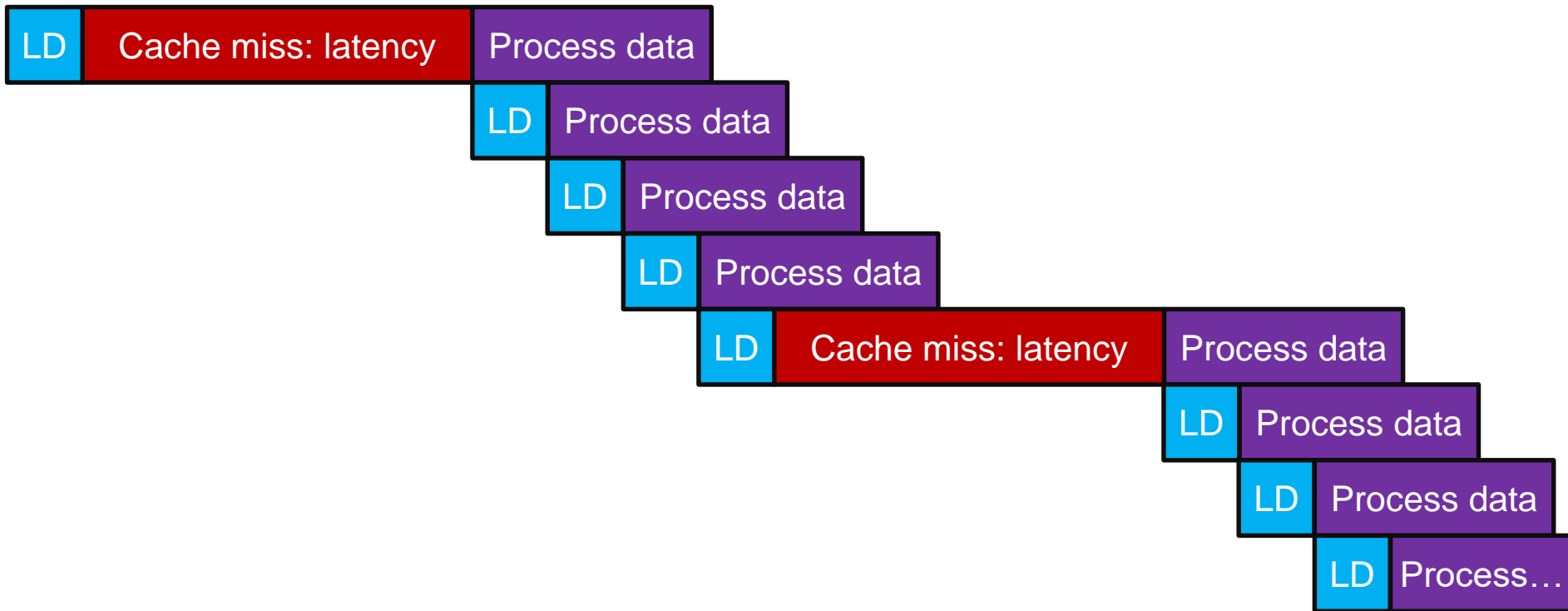
- Software prefetching:

- Possible due to special operations

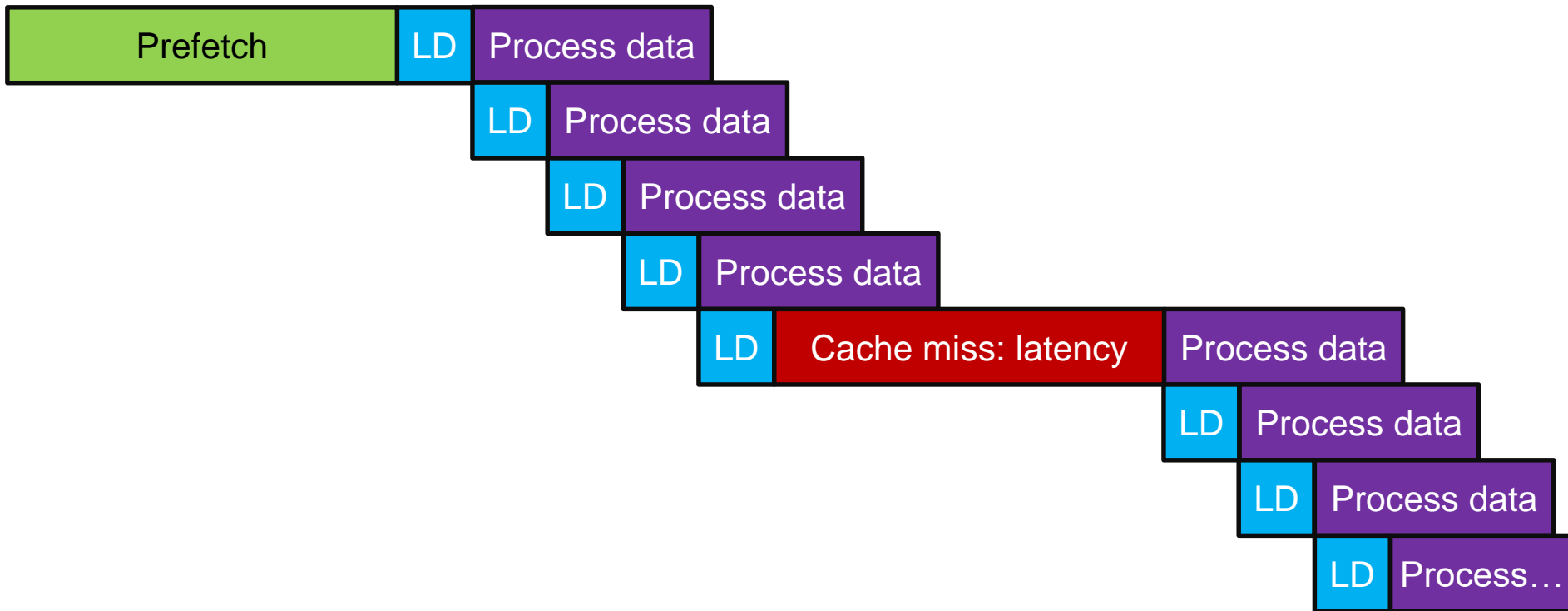
- Limited use

- ...

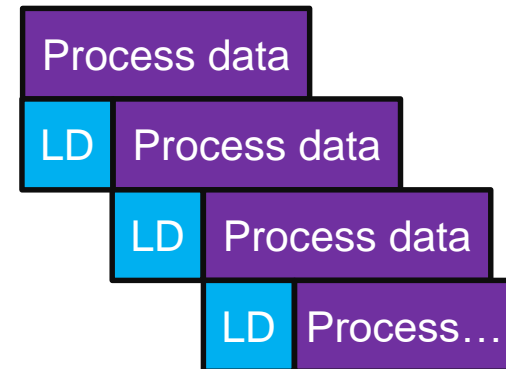
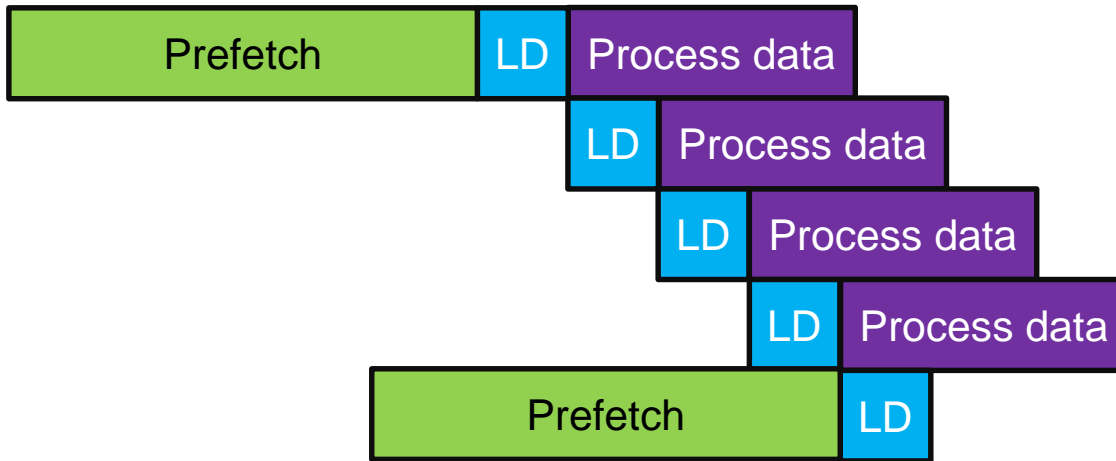
Example with software prefetching



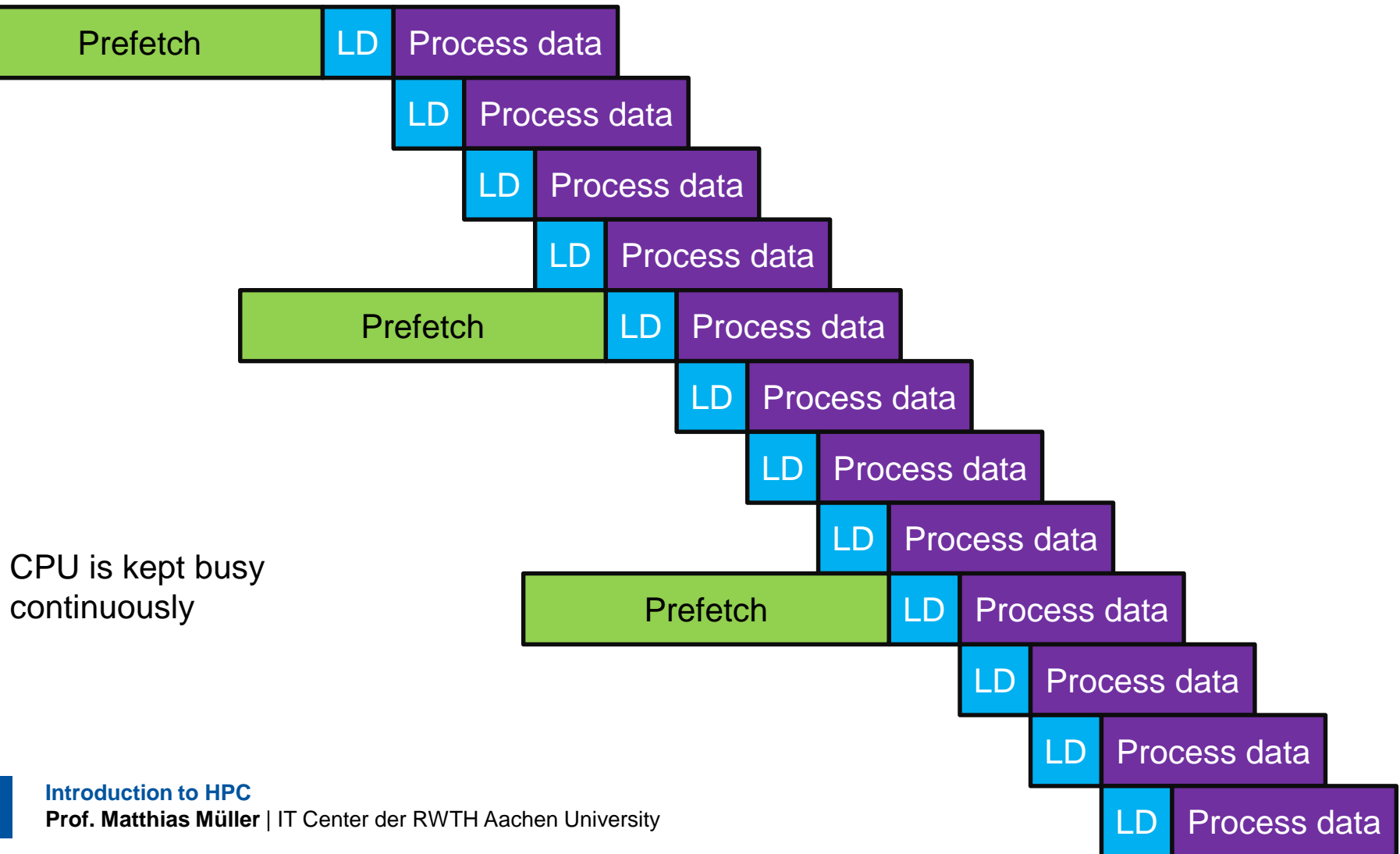
Example with software prefetching



Example with software prefetching



Example with software prefetching



- **Most architectures like e.g. the IA-32 family (Intel/AMD x86/x64, IBM Power) use hardware-based automatic prefetching mechanisms**
 - Hardware detect certain predefined patterns of memory accesses and prefetches at special conditions:
 - Intel x86: “Adjacent cache line prefetch” loads 2 (64-byte) cache lines on L3 miss → Effectively doubles line length on loads
 - Intel x86: Hardware prefetcher: Prefetches complete page (4 KByte) if 2 successive Cachelines in a page are accessed

- If cache is full the “old item” that occupies the cache line has to be removed if new data is to be loaded
- There exist different strategies for that task:
 - Least recently used (LRU)
 - Replaced block is the one that has been unused for the longest time
 - First in, first out (FIFO)
 - If LRU is too complicated to calculate: *oldest* block is replaced
 - Random
 - Candidate blocks are randomly selected to spread allocation uniformly
 - Most recently used (MRU), Not recently used (NRU),...
- If the cache line to be evicted was modified it might need a write-back to memory prior to being overwritten

1. Why supercomputers?
2. **Modern processors**
 - Stored-program computer architecture
 - Von Neumann architecture
 - Instruction set architecture
 - General purpose cache-based microarchitecture
 - Pipelining
 - Superscalarity
 - SIMD
 - **Memory hierarchies**
 - Cache
 - Prefetching
 - **Cache mapping**
 - Cache coherence
3. Basic optimization techniques for serial code
4. Data access optimization
5. Parallel computers
6. Parallelization and optimization strategies
7. Parallel algorithms
8. Shared-memory programming with OpenMP
9. Distributed-memory programming with MPI
10. Hybrid programming (MPI + OpenMP)
11. Heterogeneous architectures (GPUs, Xeon Phi)
12. Energy efficiency

■ How is a cache organized internally?

- Data RAM: contains copies of data that was loaded from main memory (cache lines)
- Tag RAM: contains the main memory addresses that these lines were loaded from

■ When a load request is issued..

- The Tag RAM is checked whether the cache already contains the requested address
 - Yes: Cache-Hit → Resolve the request from cache
 - No: Cache-Miss → Resolve the request from another cache or memory

■ Direct mapped cache

→ Part of the address selects one entry in the tag RAM for comparison

■ N-way set associative cache

→ Part of the address selects n entries in the tag RAM for comparison

■ Fully associative cache

→ Address is compared to all tag RAM entries

■ Requested memory address is decomposed into certain parts

→ Valid for direct mapped and N-way associative caches

■ Assume

→ Cache line length C_L of 2^k

→ Total size of cache: 2^s (in case of N-way associative: $N \times 2^s$)

→ **#cache lines** = $\frac{2^s}{2^k} = 2^{s-k} =: N_{CL}$

→ The least-significant k bits of the memory address denote the byte that is requested.

→ The next $s - k$ bits denote the cache line Index

→ For a wordsize of $N_{word} = 2^W$, the remaining $W - s - k$ bits denote the Tag

Example: Direct mapped cache

■ Cache has

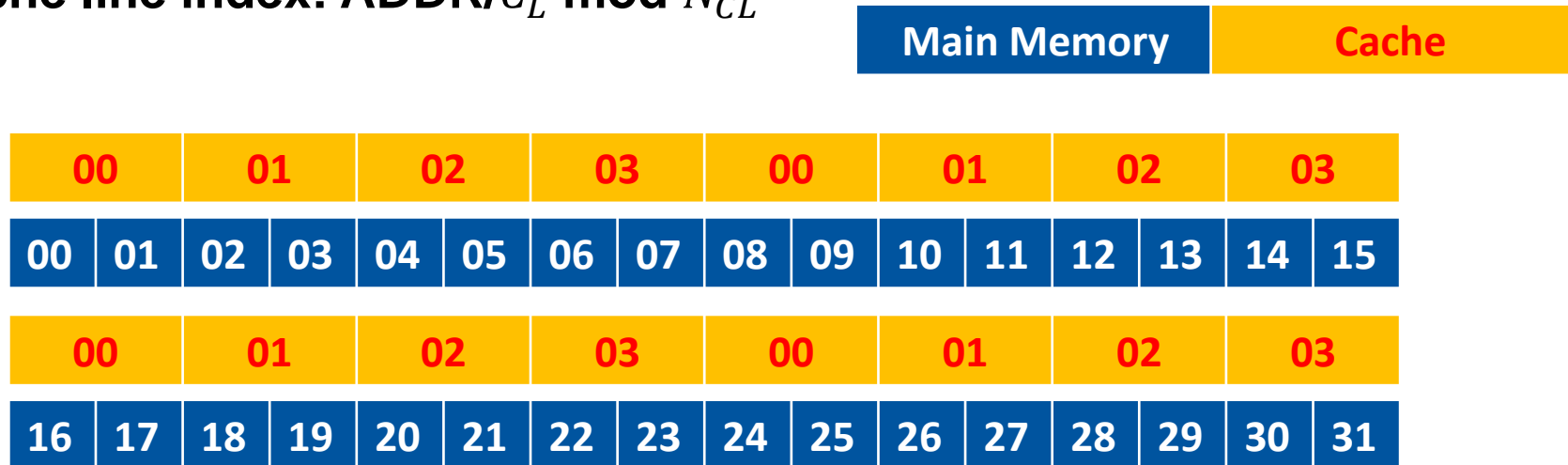
- certain cache line Length C_L and
- certain number of cache lines N_{CL}

■ Total cache size

- $N_{CL}C_L$ is mapped over the entire address space repeatedly so that multiple address ranges map to the same cache line

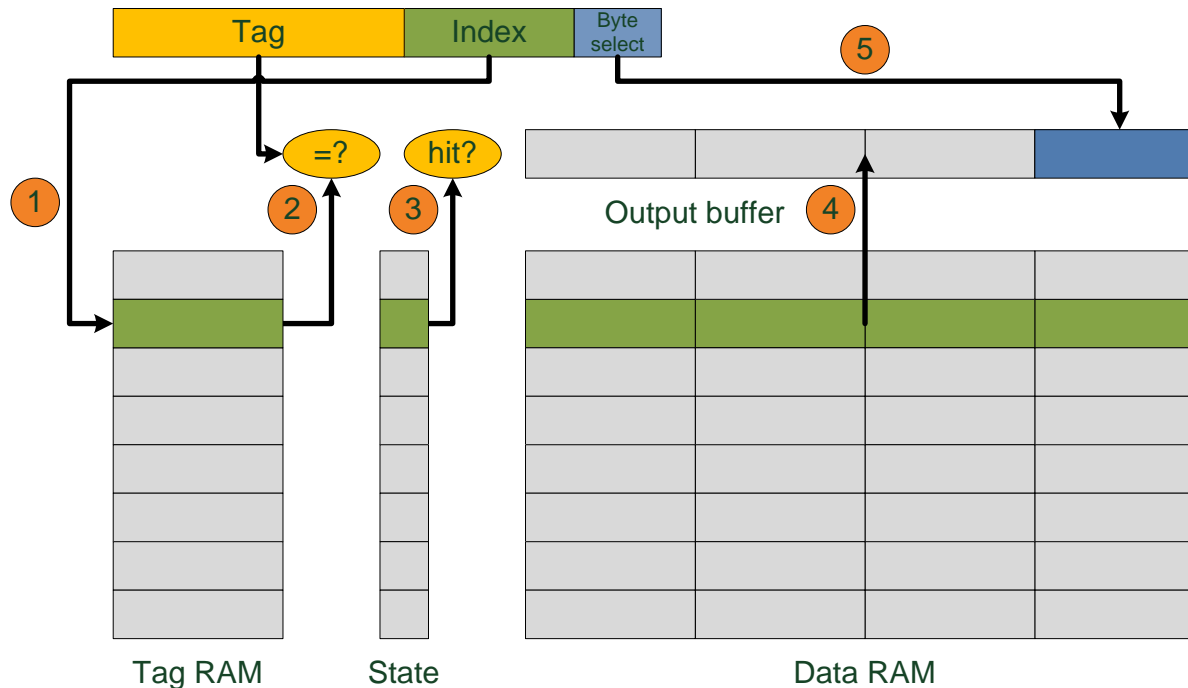
■ Cache line index: $\text{ADDR}/C_L \bmod N_{CL}$

Example: $C_L = 2, N_{CL} = 4$



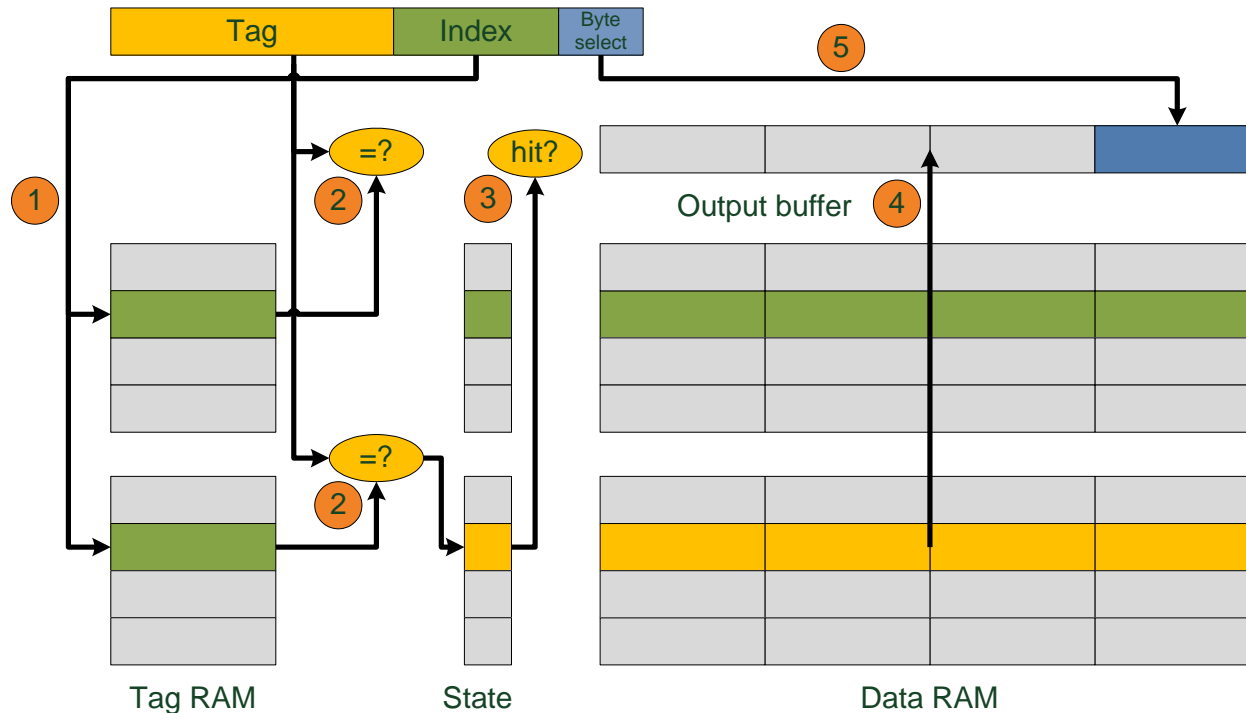
- Fully associative caches map a certain addresses into any cache lines.
- The tag in this case consists of $W - k$ bits.
- The remaining k bits denote the byte that is requested.

Direct Mapped Cache:



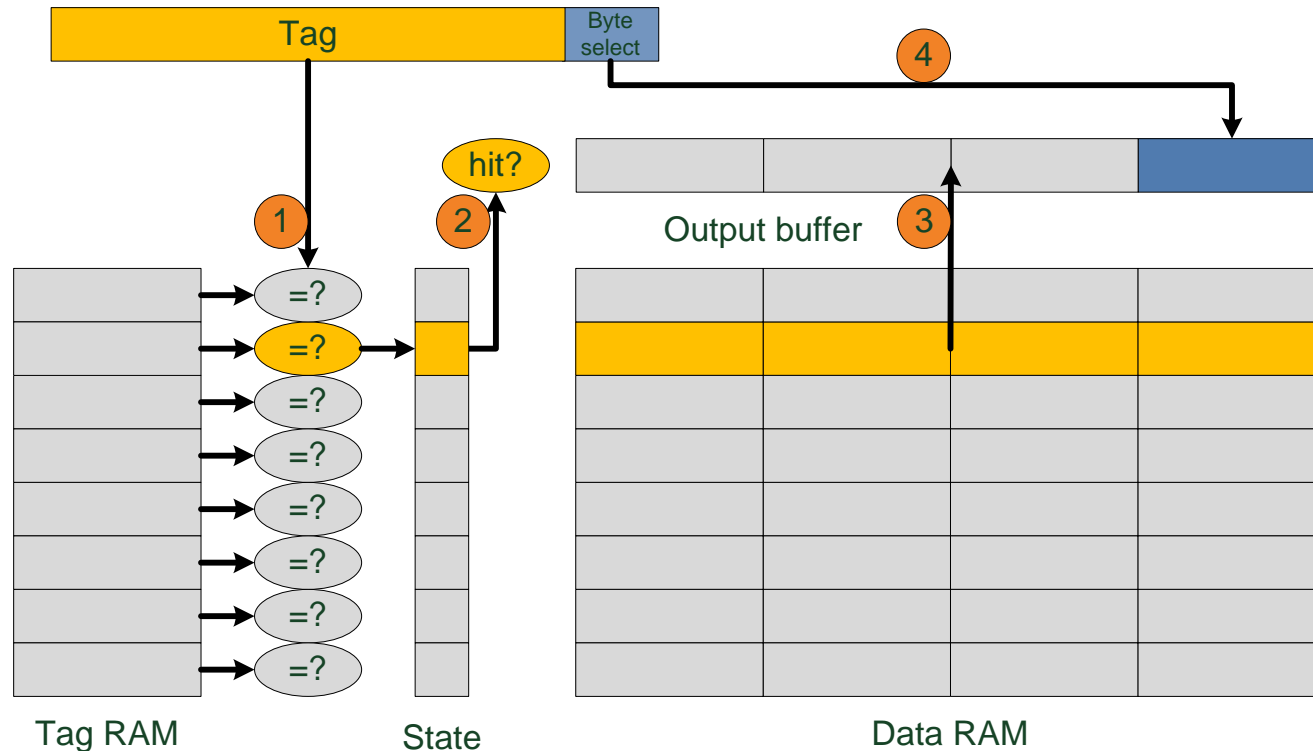
1. Index selects one cache line
2. Check if selected tags are equal (no => miss)
3. Check if cache line is valid (no => miss)
4. Copy data to output buffer
5. Select required part from cacheline (if data is valid)

N-way set associative cache



1. Index selects n cache lines (one in each way)
2. Check if any of selected tags equal requested address (no \Rightarrow miss)
3. Check if cache line is valid (no \Rightarrow miss)
4. Copy data to output buffer
5. Select required part from cacheline (if data is valid)

Full-associative cache



1. Check if any tag equals requested address (no=>miss)
2. Check if cache line is valid (no=>miss)
3. Copy data to output buffer
4. Select required part from cacheline

Comparison of cache mappings

Fully associative:

Block 12 can go anywhere

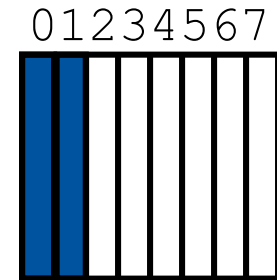
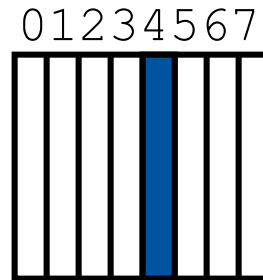
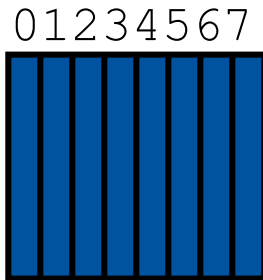
Direct mapped:

Block 12 can go only into block 1
($12 \bmod 8$)

Set associative:

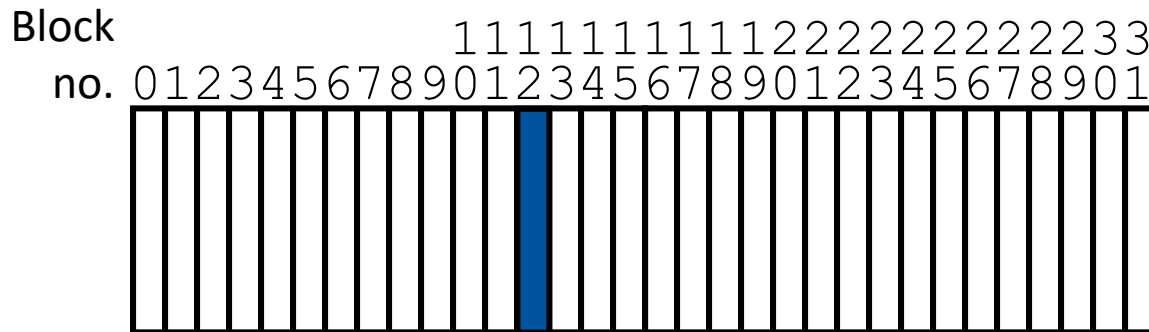
Block 12 can go anywhere in set 0
($12 \bmod 4$)

Cache



Set0Set1Set2Set3

Memory



Assumption

- Cache: 8 block frames
- Memory: 32 blocks

■ Fully associative Cache

- Memory addresses may be mapped to every entry in the cache (theor. ideal)
- Impractical: Large bookkeeping overhead: For each memory access every cache line needs to be checked for its TAG RAM.
- Best utilization of cache space

■ Direct Mapped Cache

- Every memory address maps to a certain cache line:
 - Fast determination of hit or miss
 - Bad utilization of cache space
 - May be disposed because of cache trashing

Example: Cache trashing (Direct mapped cache)

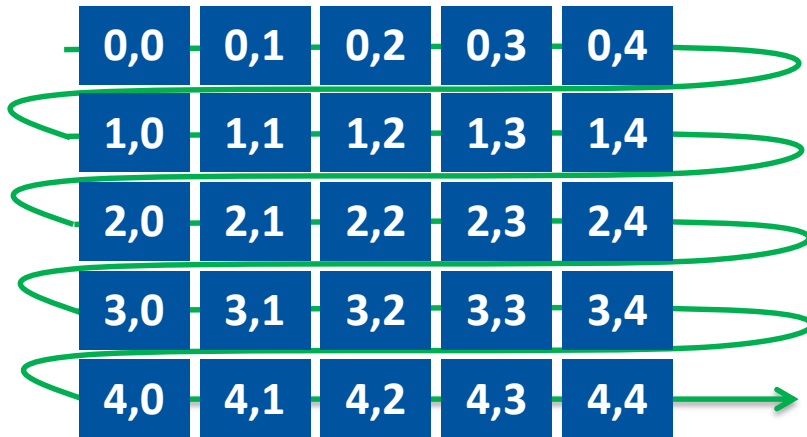


- Lines are loaded and evicted rapidly in succession because of the memory access pattern of the application:

Strided vector addition:

C/C++	Fortran
<pre>for(i=0; i<N; i+=CACHE_SIZE/8) A[i] = B[i] + C[i];</pre>	<pre>DO i=1, N, CACHE_SIZE/8 A(i) = B(i) + C(i) END DO</pre>

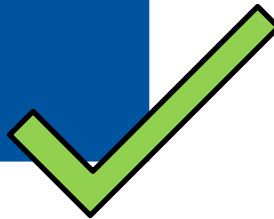
- Cache reuse factor: 0
- Situation is called *conflict miss*
- N-way associative caches address this problem:
 - Compromise between access time and cache space utilization
 - Alleviate the impact of cache trashing
 - Bookkeeping overhead is manageable



Row by Row Ordering or
row major ordering

Rows are stored consecutively
in memory.

```
for(i=0; i<N; ++i)
  for(j=0; j<N; ++j)
    a[i][j] = i*j;
```



```
for(j=0; j<N; ++j)
  for(i=0; i<N; ++i)
    a[i][j] = i*j;
```



Memory layout for Fortran




0,0	0,1	0,2	0,3	0,4
1,0	1,1	1,2	1,3	1,4
2,0	2,1	2,2	2,3	2,4
3,0	3,1	3,2	3,3	3,4
4,0	4,1	4,2	4,3	4,4


Column by Column Ordering or
column major ordering

Columns are stored consecutively
in memory

```
DO i=1,N  
  DO j=1,N  
    a(j,i) = i*j;  
  END DO  
END DO
```



```
DO j=1,N  
  DO i=1,N  
    a(j,i) = i*j;  
  END DO  
END DO
```



■ Write through

- The information is written to the cache line as well as the corresponding location in main memory

■ Write back

- The information is only written to the cache line
- The information is written to main memory only when the cache line is evicted by a replacement strategy

■ Comparison WT & WB

- WT does not result in write back when cache lines need to be evicted
- WB prevents repeating writes to have an impact on performance

■ Compulsory misses

- First access to a certain memory location which results into a load from main memory → Prefetching mechanism can alleviate these effects

■ Capacity misses

- Working set does not fit into the cache; Cache lines need to be evicted and loaded from main memory repeatedly
 - Multiple levels of cache with increasing size

→ Conflict misses

- Strided accesses to memory location that map to the same cache lines

■ Caches are transparent

- Pro: No need to care about → implemented in hardware
- Con: Optimizing a program requires detailed knowledge of the hardware

■ Writing to a memory location results in loading it into the cache

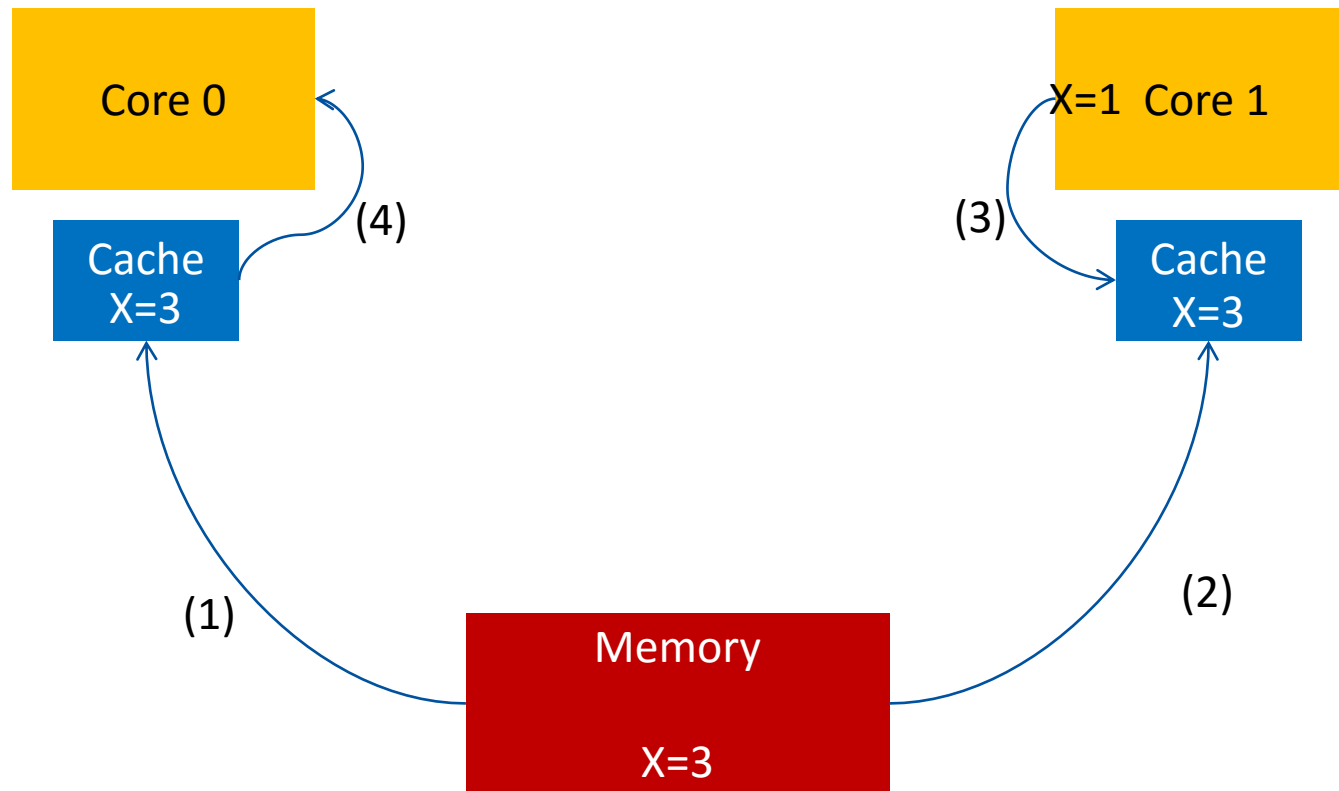
■ Recent processors introduce instructions to control caches

- Prefetch instructions
- Flush instructions
- Nontemporal writes that directly write to main memory
 - For handling data that is only to be accessed once

1. Why supercomputers?
2. **Modern processors**
 - Stored-program computer architecture
 - Von Neumann architecture
 - Instruction set architecture
 - General purpose cache-based microarchitecture
 - Pipelining
 - Superscalarity
 - SIMD
 - **Memory hierarchies**
 - Cache
 - Prefetching
 - Cache mapping
 - **Cache coherence**
3. Basic optimization techniques for serial code
4. Data access optimization
5. Parallel computers
6. Parallelization and optimization strategies
7. Parallel algorithms
8. Shared-memory programming with OpenMP
9. Distributed-memory programming with MPI
10. Hybrid programming (MPI + OpenMP)
11. Heterogeneous architectures (GPUs, Xeon Phi)
12. Energy efficiency

■ When switching to multicore processors, the concept of caching needs some considerations:

- Caches store
 - Private data only used by a single core
 - Data that is shared by multiple cores
- Sharing data inside a common Cache
 - Reduces memory latency on more than one core
 - Increases the effective bandwidth of shared data
- Cache coherence problem
 - Modifications of cached data must be visible to all cores



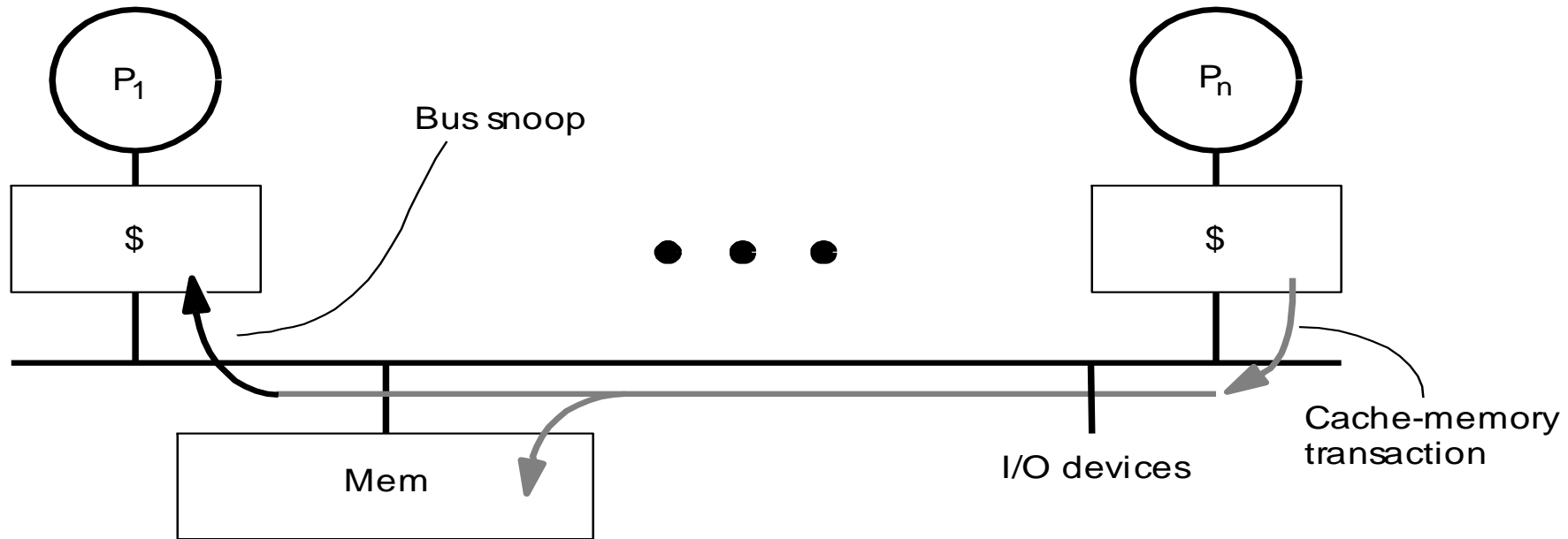
- After time step 3, cores see different values for X
- Depending on which cache writes back X , value might be stale

■ Directory based

→ Sharing status of a block of physical memory is kept in only one place, the directory

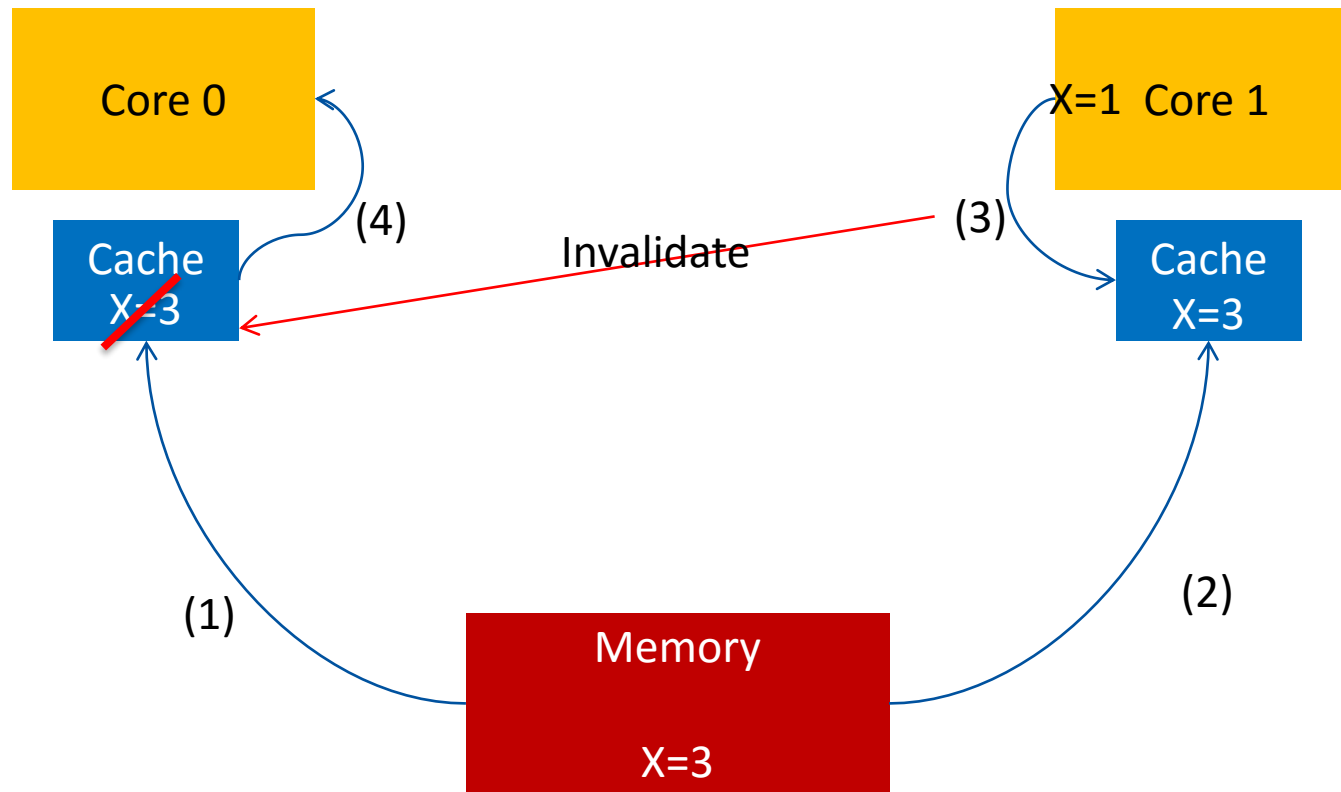
■ Snooping

→ Caches are connected via a broadcast medium and snoop(monitor) on that medium for memory blocks that they currently store



■ **Cache controller snoops on the medium and take relevant action to ensure cache coherence:**

- Invalidate
- Update
- Supply value



■ **All recent MP systems use write-invalidate**

■ Write-through

→ Get most recent value from main memory

■ Write-back

→ All cores check their caches for addresses placed on the memory bus

→ If core has most recent copy of requested cache block it provides it in response to a read request

■ Write-back vs. Write-through

→ Write-back needs lower memory bandwidth

→ Most recent multiprocessors use write-back

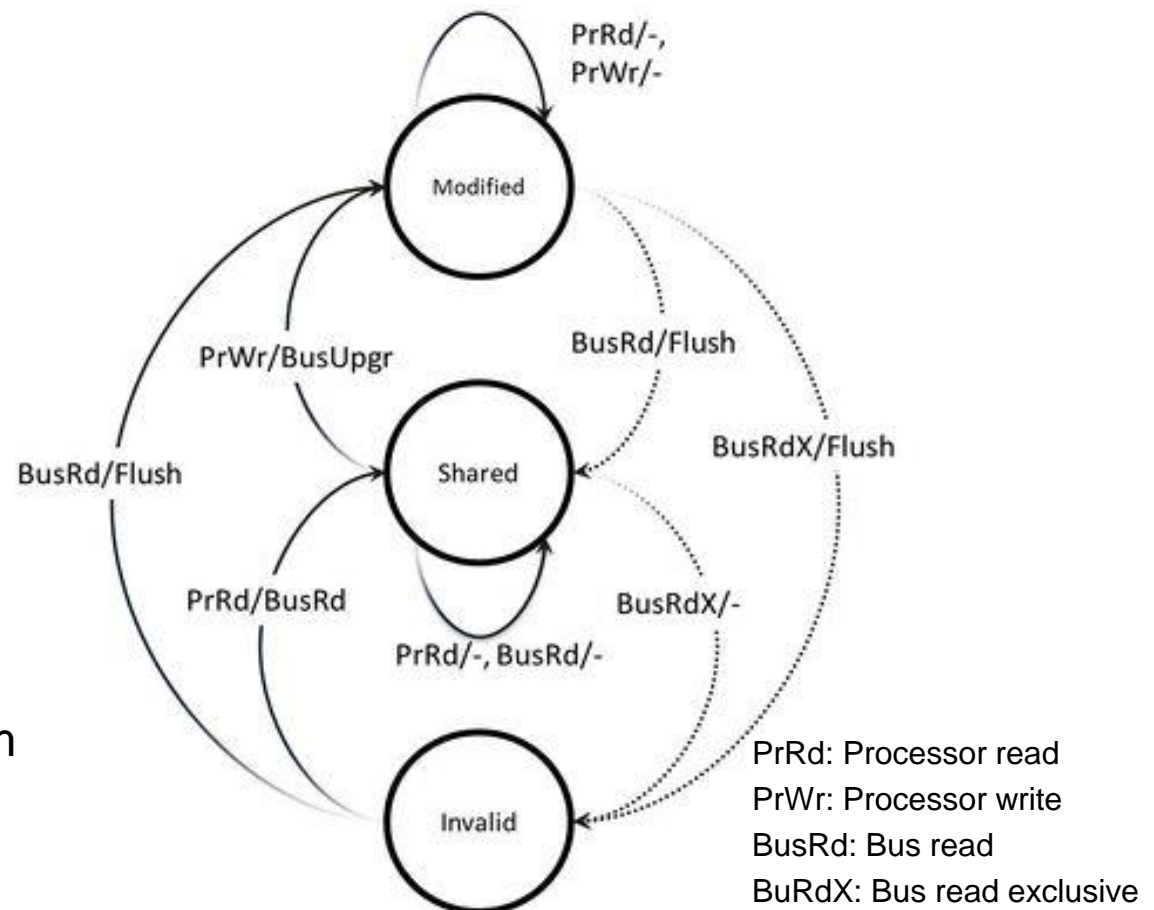
Cache coherence protocols

Example: MSI invalidate protocol



- Write-back snooping protocol based on block invalidation
- Each block of memory has one of the following states

- Modified: exactly one cache (owner) holds a valid copy, memory is stale
- Shared: more than one cache hold valid copy
- Invalid: block contains invalid data, needs the most updated copy from owner



■ See slide 117

→ Von Neumann architecture/ bottlenecks, ISA, CISC, RISC, ILP, pipelining, superscalarity, SIMD

■ Why were caches introduced?

- What is a cache?
- What performance characteristics does it have?
- When does it make sense to use a cache?
- What is a cache line?
- What are cache hits & cache misses?
- What is the principle of locality?

■ Which cache types do exist & what are their differences?

■ Why is cache coherence important?

→ How is the coherence achieved?

■ When is prefetching used?