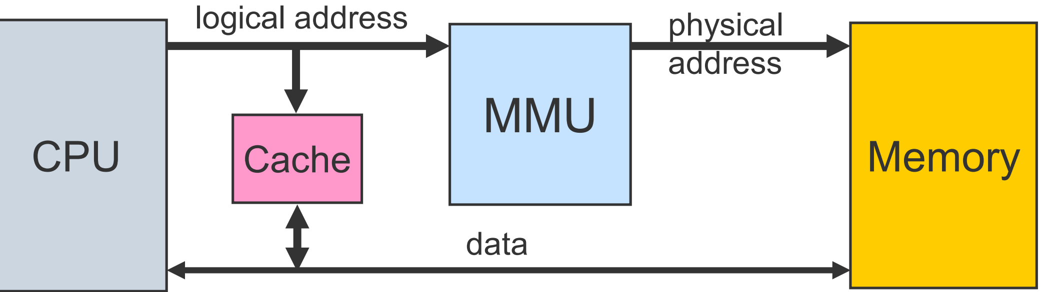
Prof. Dr.-Ing Jochen H. Schiller
Inst. of Computer Science
Freie Universität Berlin
Germany



TI II: Computer Architecture Memories

Hierarchy
Types
Physical & Virtual Memory
Segmentation & Paging
Caches

Content

1. Introduction
   - Single Processor Systems
   - Historical overview
   - Six-level computer architecture

2. Data representation and Computer arithmetic
   - Data and number representation
   - Basic arithmetic

3. Microarchitecture
   - Microprocessor architecture
   - Microprogramming
   - Pipelining

4. Instruction Set Architecture
   - CISC vs. RISC
   - Data types, Addressing, Instructions
   - Assembler

5. **Memories**
   - **Hierarchy, Types**
   - **Physical & Virtual Memory**
   - **Segmentation & Paging**
   - **Caches**

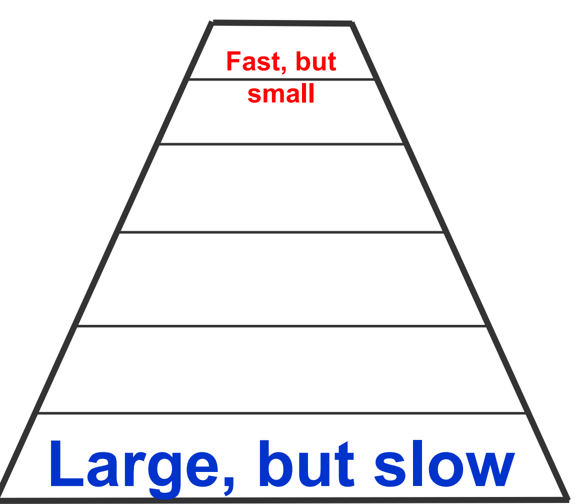
MEMORY HIERARCHY

Motivation for a memory hierarchy

A single memory with low latency AND large capacity is technologically feasible, but very expensive

Solutions

- Layered architecture of different types of memory and transfer of data between these different layers
- Cache memory: Relatively low access times ➔ speed-up of load/store instructions
- Virtual memory: Increase of real memory size to support, e.g., simultaneous execution of multiple processes



Memory hierarchy

Desktop	Register
Desktop environment	On-Chip-Cache
Shelves	Secondary level Cache (SRAM)
	Main memory (DRAM)
Archive	Secondary memory (Hard disks)
Remote archive	Archive memory (Tapes, disks)

Memory hierarchy

Register

On-Chip-Cache

Secondary level Cache
(SRAM)

Main memory
(DRAM)

Secondary memory
(Hard disks)

Archive memory
(Tapes, disks)

Increasing cost per byte

Decreasing capacity

Decreasing access time

Memory hierarchy

Behavior: like a single, large and ultra-fast memory, if

- Locality of program execution
- Transfer of data happens fast/early enough (replacement and prefetching strategies)
- Non-homogeneous layers of memory are "invisible" to the user (virtual memory)

Performance of the whole hierarchy very much depends on the

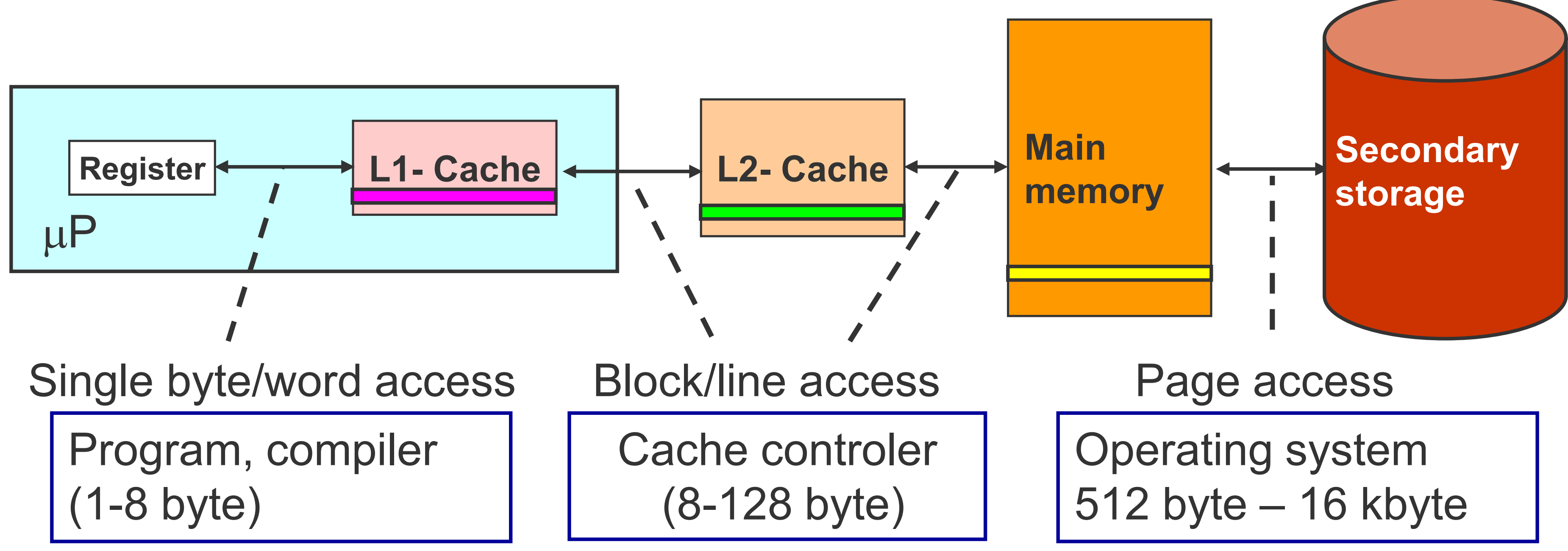
- characteristics of the memory technology (access time),
- addressing of the memory cells (random, sequential) and
- organization by the operating system/virtual memory management

Memory hierarchy - parameters

Capacity	Level	Access time
256-1024 byte	Register	ps
16-512 kbyte	1st level Cache	ps
1-16 Mbyte	Second/Third Level Cache (SRAM)	ns
256 Mbyte - 64 Gbyte	Main Memory (DRAM)	ns
128-1024 Gbyte	**SolidStateDisk**	**<ms**
200-8000 Gbyte	Secondary Memory (harddisk, virtual disk, ...)	ms
100-100000 Tbyte	Archive (disks, tapes, optical storage)	min

Memory hierarchy

The system transfers data only between neighboring layers of the hierarchy.



MAIN MEMORY

Overview

Two different basic types of memory:

- Permanent storage of data
  ➔ long term memory
- Read Only Memory (ROM), used for e.g. firmware, operating system of controllers, system tables, boot loaders…

- Temporary storage of data
  ➔ short term memory
- Random Access Memory (RAM), volatile, used for user programs, temporary data, …

Terminology

Memory cell

- Typically 1 bit – the fundamental building block of memory
- The bit is stored e.g. in a DRAM cell (one transistor/capacitor), SRAM cell (4 or 6 transistors) or flip-flop
- https://en.wikipedia.org/wiki/Memory_cell_(computing)

Byte

- Fixed number of memory cells accessible via a single address, typically 8 bit (smallest addressable unit)

Memory word

- Maximum number of memory elements transferred in a single bus cycle between processor and memory (e.g., 32 bit, 64 bit, depending on the width of the data bus)

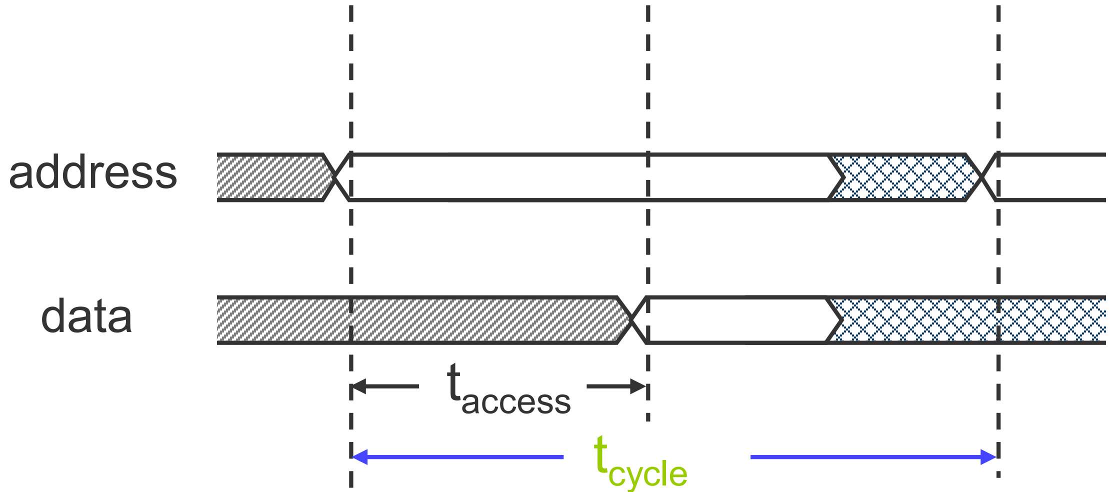
Random access

- Direct and individual access of each memory cell possible (without the need to address other cells before)
- Selection of a cell is based on an address decoder (address -> signals to address a cell).

Terminology

Characterization of the performance of memories

- Access time
  - Maximum latency between the availability of an address at the address bus of the memory until it outputs the desired data on the data bus
- Cycle time
  - Minimum waiting time between two consecutive memory accesses



Access time vs. cycle time

Cycle time can be much larger than access time!

Reasons

- Memory cells must “recover” after access
- Reading out a memory cell destroys the stored data in some technologies and, thus, this data has to be written back into the cell

Ideal case: cycle time = access time

Reality: cycle time > access time (up to 80%)

Classification of semi-conductor memory

Semi-conductor memory

- Read only memory
  - Non-reversible
    - ROM
    - PROM
  - Reversible
    - EPROM
    - EEPROM
- Random access memory
  - Static RAM
  - Dynamic RAM
  - pseudo-static RAM (Static RAM, Dynamic RAM)
- Non-volatile RAM (EEPROM, Static RAM) ← "Flash RAM"

RAM	Random Access Memory
ROM	Read-Only Memory
PROM	Programmable ROM
EPROM	Erasable PROM
EEPROM	Electrically EPROM

Types of DRAM

No real revolutionary designs of new memories up to now. Integration density increases, DRAM stays as no. 1 type of memory.

August 12th, 1981

- First IBM PC (Model 5150 with Intel 8086 processor),
  16 kbyte RAM on 8 single chips with 16 kbit capacity each.

Today

- The size of this historical RAM easily fits into a L1-cache of a wristwatch…

https://en.wikipedia.org/wiki/Dynamic_random-access_memory

Synchronous DRAM (SDRAM)

First generation RAM operated in an asynchronous fashion

- Output data not in sync with clock, delay depends on technology

SDRAM

- All I/O signals synchronous to the rising edge of a clock signal
- Today the dominating technology – comes in many different versions / enhancements
- CPU, chips-set on motherboard and memory communicate via a bus system in a synchronous fashion determined by a clock

Typically, SDRAMs use 2 or more internal memory banks to overlap access

- This hides the access latency (keeps the data bus continuously busy)
- The internal memory controller automatically switches banks for subsequent addresses

Double Data Rate (DDR) SDRAM and more …

Next step in the SDRAM development early 2000: use both edges of the clock signal to transfer data

- System uses rising and falling edge of the clock to transfer data
- Requires more internal banks to keep the bus busy
- https://en.wikipedia.org/wiki/DDR_SDRAM

Performance

- Current dominating DDR4 standard: up to 25.6 Gbyte/s per 64 bit module
- New (2020) DDR5 standard: up to 67.2 Gbyte/s per 64 bit module (2 channels with up to 33.6 Gbyte/s) including error correction
- But that pretty much depends on the workload and the overall system performance…

Versions

- Low Power (LPDDR) for e.g. smart phones
- Graphics (GDDR) for GPUs
    - Typically faster due to shorter lines, optimized GPU/RAM configuration, no interconnects but soldered
        - GDDR6 offers up to 64 Gbyte/s, GDDR6X about 84 Gbyte/s per chip (about 1 Tbyte/s for a system…)

Questions & Tasks

- What are disadvantages of a memory hierarchy? So, why do we do it at all?
- Why are SSDs that successful?
- Why is the cycle time of RAM typically higher than the access time?
- ROM is not flexible – are there still reasons for using it?
- What determines the speed of RAM?

Main memory

ORGANIZATION OF MAIN MEMORY

Organization of memory

Memory

- Linear list of memory words
- Consists of one ore more memory chips
- Access time depends on technology (latencies in the range of 5-15 ns)
- Memory width typically the same as width of data bus (e.g. 8, 16, 32, 64 bit, but also 384 bit in e.g. graphics adapters). This typically equals the amount of bits transferred in a bus cycle.
  - Depends on coding of memory bus etc.

Memory										
Address	0	1	2							2^n-1

History: What is a byte?

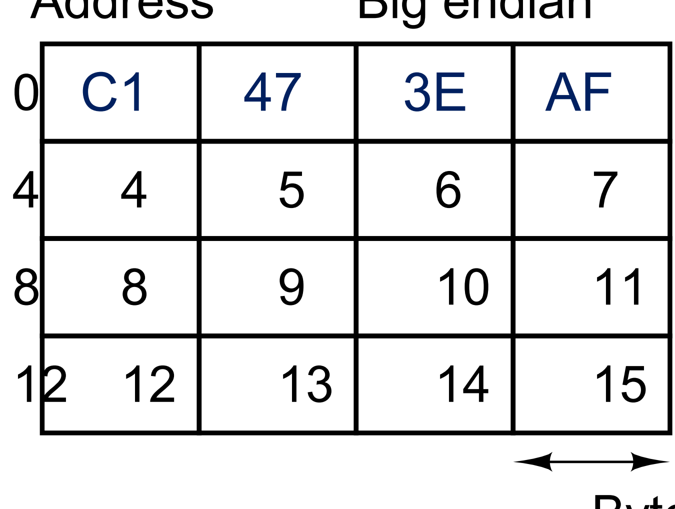
… or why a byte/smallest addressable unit has not always equaled 8 bits

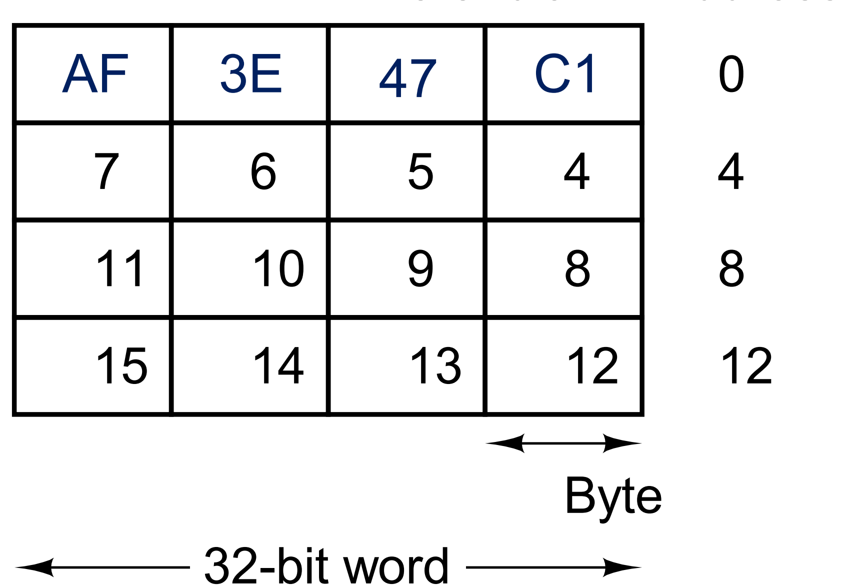
Computer	bit/cell
Burroughs B1700	1
IBM PC	8
DEC PDP-8	12
IBM 1130	16
DEC PDP-15	18
XDS 940	24
Electrologica X8	27
XDS Sigma 9	32
Honeywell 6180	36
CDC 3600	48
CDC Cyber	60

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

Byte Ordering – Endianness: Big endian vs. little endian

0xAF3E47C1





SPARC, IBM mainframes, Internet, ARM, RISC-V

Intel (thus the PC world), ARM, RISC-V

Bi-endianness: Switchable endianness

Address		
0	AF	C1
1	3E	47
2	47	3E
3	C1	AF

Organization of Memory

Processors with data bus width > 8 bit can quite often still access or manipulate single bytes

- Often data fetched as word, then byte selected
- Memory capacity still measured in bytes

The width *n* of the address bus typically determines the maximum capacity of the memory 2^n

Typical maximum memory capacities:

- 8-bit processor with 16-bit address bus: 64 KiB
  - note: this equals 2^{16} = 65536 byte, thus *kibibyte* according to the IEC standard instead of the former KB or kbyte or kByte or Kbyte as kilo means 1000, https://en.wikipedia.org/wiki/Kilobyte
  - In the networking world *kilo* is used (e.g. kbit/s) meaning 10^3 bit
- 16-bit processor with 24-bit address bus: 16 MiB (*mebibyte*)
- 32-bit processor with 32-bit address bus: 4 GiB (*gibibyte*)
- 64-bit processor with 64-bit address bus: 16 EiB (*exbibyte*)

Memory Map

Indicates which type of memory is used for which address range, where I/Os are, or which component is responsible for a certain address range

bit 31 0 address 2^n-1

ROM

I/O

DRAM

SRAM

0

R/$\overline{W}$

…

address

n-2 bits

$a_n \ldots a_2$

1-of-(n-2) decoder

…

…

Data bus

8, 16, or 32 bit

2 address bits a_1, a_0

1-of-4 decoder

Address	Region
0xFFFF – 0xFE00	Interrupts (FFE0)
0xFE00 – 0x1100	Flash ROM (60 K) — PROG1 Text, PROG2 Text, Firmware Text
0x1100 – 0x1000	Infomem
0x1000 – 0x09FF	Bootloader
0x09FF – 0x0200	RAM (2 K) — Stack, PROG1 Data, PROG2 Data, Firmware Data
0x0200 – 0x0000	I/O

Example Memory Map

Here

- Higher addresses
    - ROM, e.g. as FLASH for the non-volatile parts of the operating system (bootstrap, BIOS)
- Followed by
    - I/O assuming a processor with memory-mapped I/O
- Lower addresses
    - RAM, typically DRAM due to the low cost and large capacity
    - Disadvantage: DRAMs are relatively slow (require e.g. refresh)
    - Thus, parts of the addresses could be mapped onto SRAM that does not require wait cycles
        - Or: hide delay via caches – covered later

Questions & Tasks

- How long does it take to transfer the content of a 16 Gbyte RAM via a 1 Gbit/s connection? Why is Gbyte misleading? (And what should we know about communication overhead etc. …)
- When is the endianness of importance?
- What is a memory map needed for?

CACHE MEMORY

Cache Memory

Problem

- Clock cycles of fast processors are much shorter than cycle times of large and cheap DRAMs
- Additionally, DRAMs require refresh cycles (e.g. every row in DRAM every 64 ms)
  - This requires wait cycles
- SRAM does not require wait cycles, but requires more space per bit (4 or 6 transistors instead of 1) and, thus is more expensive and has a higher power dissipation
  - Therefore, SRAMs typically have less capacity

Solution

- Insert a fast, but smaller memory consisting of SRAM between the registers of the processor the relative slow, but larger DRAM
- This is called a cache memory

In general, a cache is a small, but fast buffer in front of a larger, but slower memory to enhance access performance (read and/or write).

Cache Application

Application

- Hiding the access time of the main memory (DRAM) of a computer by avoiding wait cycles
    - Between CPU registers and DRAM, can be separated as data and instruction cache (or unified)
    - This is common today as level 1, level 2, level 3 cache on-chip
- Lowering the access time of disk drives by integrating RAM (disk cache)

In this lecture we focus on the first: How to avoid wait cycles when accessing the main memory?

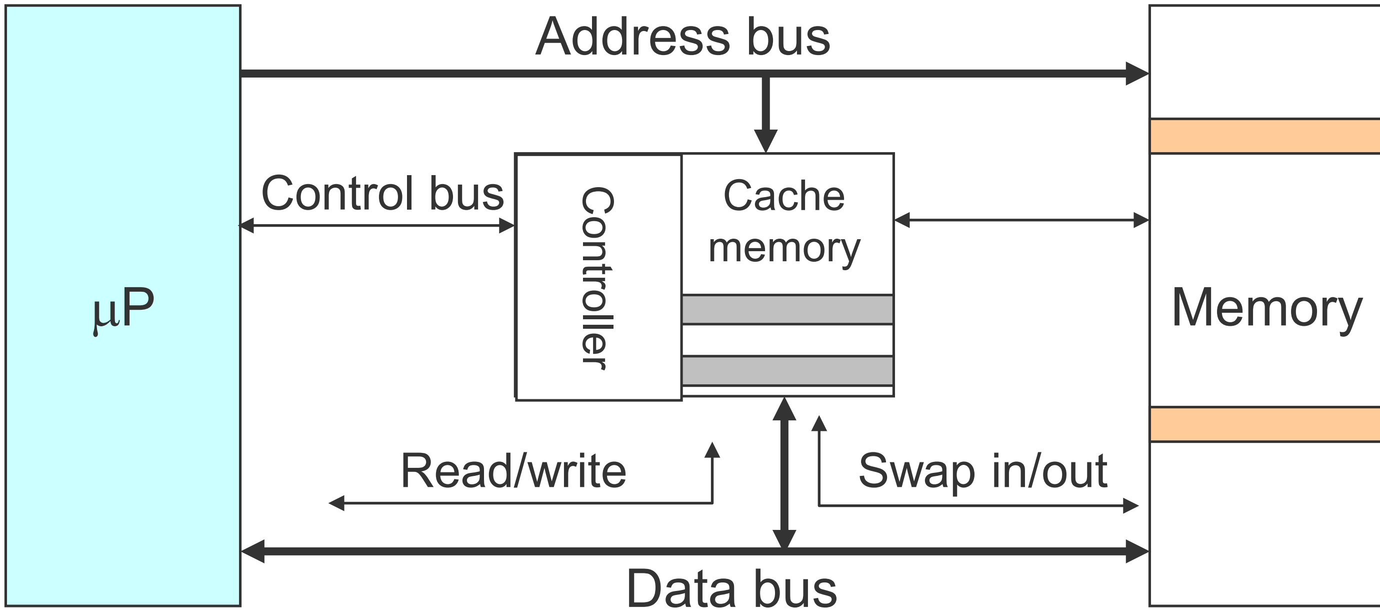
https://en.wikipedia.org/wiki/CPU_cache

Cache Architecture (high level)

The cache comprises a (relatively) small but fast memory plus a controller.

It stores with a high probability copies of those parts of the main memory the CPU will access in the near future.

In the ideal case the controller swaps these copies into the cache *before* the CPU wants to read them to avoid wait cycles.



Basic questions for cache design

Where to place a block of data (cache line)?
- Cache placement policies
- Fully associative, set associative, direct-mapped

How to find a block of data?
- Block identification
- Tags per block

Which block to replace in case of a cache miss?
- Cache replacement policies
- Random, FIFO, LRU

What happens in case of a cache write?
- Write policies
- Write back or write through (w/o write buffer)

Cache Location I

The cache memory is located between CPU (registers) and main memory.

The CPU should be able to access the cache memory (almost) as fast as the registers.

The cache memory can be on-chip (i.e. integrated on the die of the processor) or off-chip using fast SRAM technology.

Example: Today's processor comprise level 1, 2 and 3 cache on-chip

- E.g. each core has its own L1 as separated data and instruction cache (Harvard), a unified L2 (v. Neumann) and a single L3 for all cores of the CPU
- Sometimes even level 4 on-chip…

Specialized caches exist, e.g., branch target cache, trace cache…

Cache Location II

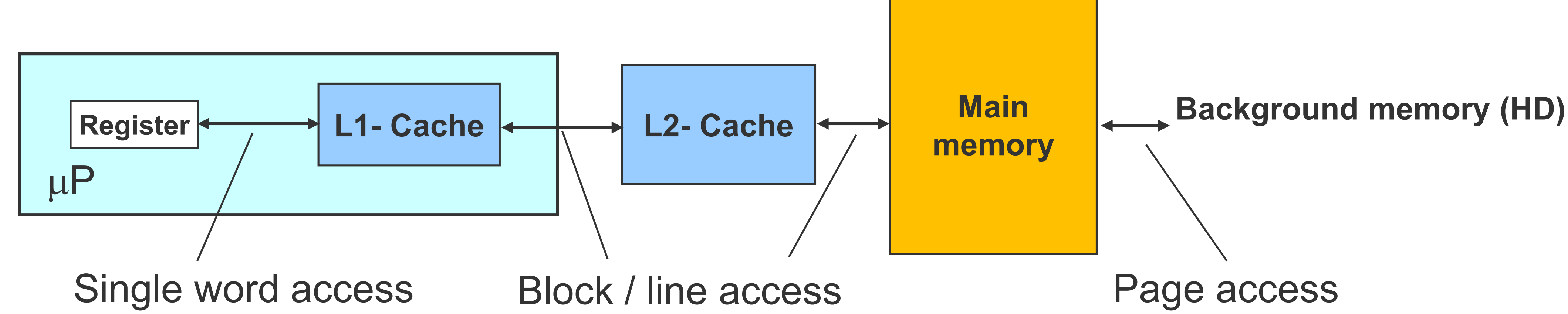
On-Chip-Cache

- Integrated on the processor die
- Very low access times (similar to registers)
- Depending on the die size quite limited capacity

Off-Chip-Cache

- External to the CPU
- Higher access times due to e.g. power level conversion, distance

Example (be aware: L2, L3, … can be on-chip as well!)



Cache Controller

The controller of the cache has to make sure that the data needed by the CPU is most likely stored in the cache

➡ With a high probability, the CPU can fetch required data from the fast cache and does not have to access the much slower main memory.

How to achieve this behavior?

- The cache controller has to copy all required data into the cache from main memory
- In an ideal operation this happens *before* the CPU accesses the data
- Otherwise, this happens at the time of accessing the data

In case of size restrictions (the cache is much smaller than the main memory) the cache controller has to replace cache content.

- In an ideal setting this is content the processor does not need anymore.
- Otherwise, the content the processor does not need in the near future.

Cache – Why does it work?

The efficiency of a cache mainly depends on the **locality** of the cached data (locality of reference). This means that the CPU repeatedly accesses the same/neighboring data (e.g. in loops, sequential access)

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

Temporal locality

- Data that will be accessed in the near future has already been accessed with high probability (e.g. in loops).

Spatial locality

- A future access to data happens within close storage locations with a high probability (e.g. sequential access).

Operation of a cache: read

Read access

- Before accessing the main memory the cache controller checks if the data is available in the cache memory.

If available: **Cache Hit**

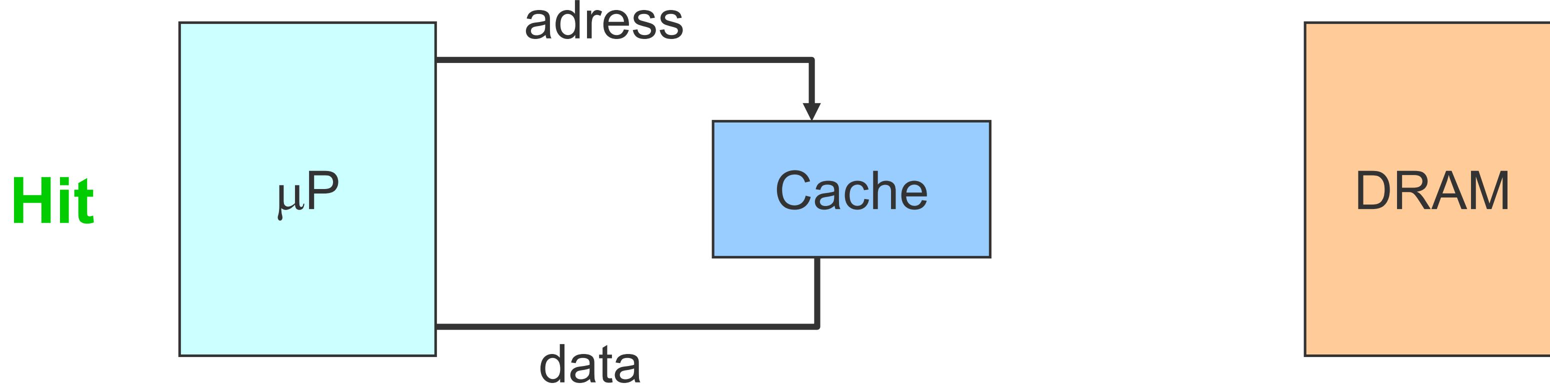
- The CPU can read the data without wait cycles directly from cache memory.

If not available: **Cache Miss**

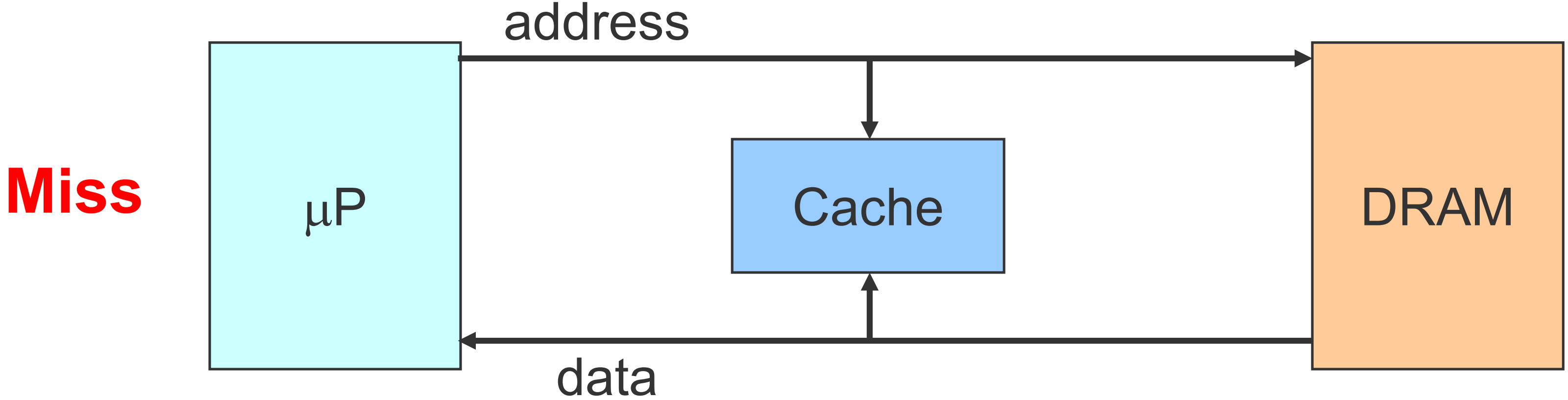
- The CPU must read the data from main memory with wait cycles plus the cache controller inserts the data into cache memory.

Operation of a cache: read

Hit



Miss



Definitions

Cache Hit Ratio: Ratio of successful cache accesses

$$Cache\ hit\ ratio = \frac{Number\ of\ cache\ hits}{Number\ of\ accesses} = \frac{Number\ of\ cache\ hits}{(Number\ of\ cache\ hits + Number\ of\ cache\ misses)}$$

The **average access time** is calculated as follows:

$$t_{Access} = (\text{hit ratio}) \times t_{Hit} + (1 - \text{hit ratio}) \times t_{Miss}$$

- t_{Hit} : cache access time
- t_{Miss}: memory access time

Operation of a cache: write

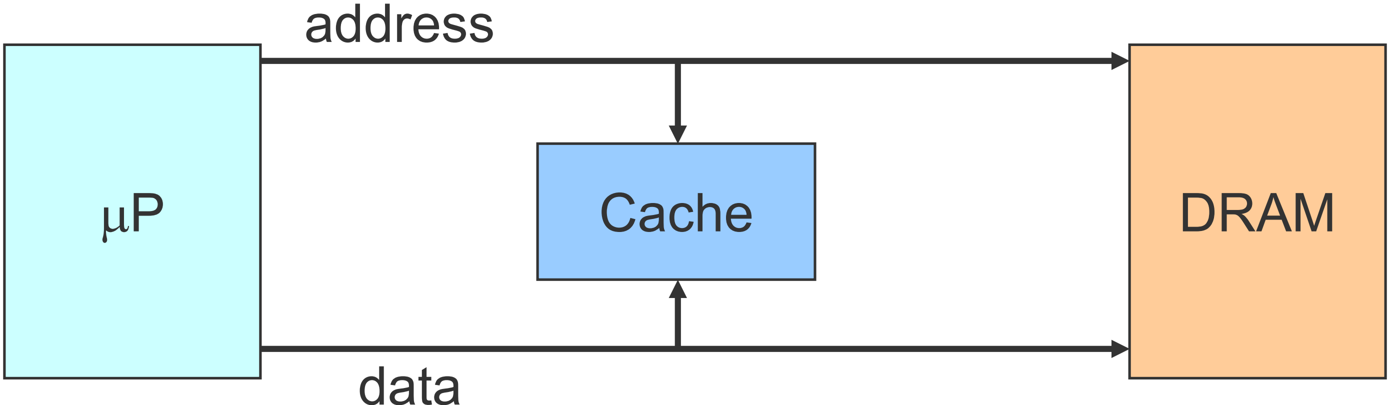
Write access

- In case of a cache miss the CPU writes the data into the cache memory and the main memory.
  - This may require a cache replacement strategy.

- In case of a cache hit, i.e., writing into the cache may change the stored data, several policies exist.

Write through policy

- The CPU always writes data into cache and main memory.



Advantage

- Guaranteed consistency between cache memory and main memory.

Disadvantage

- A write access always requires wait cycles and blocks the data bus.

Buffered write through policy

- Modification of the write through policy.
- A small write buffer for temporary storage of data mitigates the disadvantage of the write through policy.
- The cache controller transfers the data from this small buffer to main memory while the processor continues with its operation.

Write back policy

- The CPU writes data in the cache memory only and marks it with a special bit (altered / modified / dirty bit).
- The cache controller writes back data into main memory only if altered data has to be replaced.

CPU write

μP

address

Cache

data

DRAM

Write back

μP

address

Cache

DRAM

data

Write back policy – pros and cons

Advantage

- Write access can happen without wait cycles at cache memory speed

Disadvantage

- Consistency issues between cache and main memory
- Examples
  - Other components of the computer system (e.g. DMA controller) may read outdated data from the main memory, i.e., data the CPU has changed in the cache but not yet transferred to the main memory
  - Other components of the computer system may have changed data in main memory, while the CPU still operates on old data stored in the cache
- Relatively complex mechanisms needed to avoid inconsistencies, e.g., by informing the cache controller about changes in main memory – see end of this chapter!

Questions & Tasks

- Where can you find caches? Also outside of computers?
- What is an ideal cache?
- Why using Harvard or v. Neumann for caches?
- Why having caches on-chip?
- What is the role of a cache controller?
- What makes caches efficient?
- Write back seems to be more complex – why using this scheme?

Cache

ARCHITECTURE OF CACHE MEMORY

General architecture of cache memory

Address bus

Comparator

Data memory

Address

Data

Address memory

Data bus

Components of cache memory 1

A cache memory consists of (at least) two memory units

- Data memory
  - Contains all data stored in the cache
- Address memory
  - Contains the addresses in main memory of the stored data in the cache

Typically,

- each entry in the cache memory consists of a block of data, the so-called **cache line** (e.g. 64 byte)
- with each word the CPU accesses, the cache controller loads a larger part of the main memory comprising this word to benefit from spatial locality
- the address stored in the address memory is the **base address** of this larger copy of the main memory

Components of cache memory 2

A comparator checks, if the data belonging to an address on the address bus is stored in the cache memory

➡ this requires a comparison of the address on the address bus with the base address plus the range of the memory it represents

This comparison must be very fast (within a cycle) – otherwise the cache would not be efficient.

Associative memory (also known as content addressable memory) plays an important role (for smaller caches).

Blocks, lines, sets, frames …

Block **frame**

- Memory in the cache for data plus **address** tag and **status** bits.
- The address tag contains the memory address (physical or virtual) of the currently stored copy of data from the main memory.
- The status bits indicate if the entry is valid.

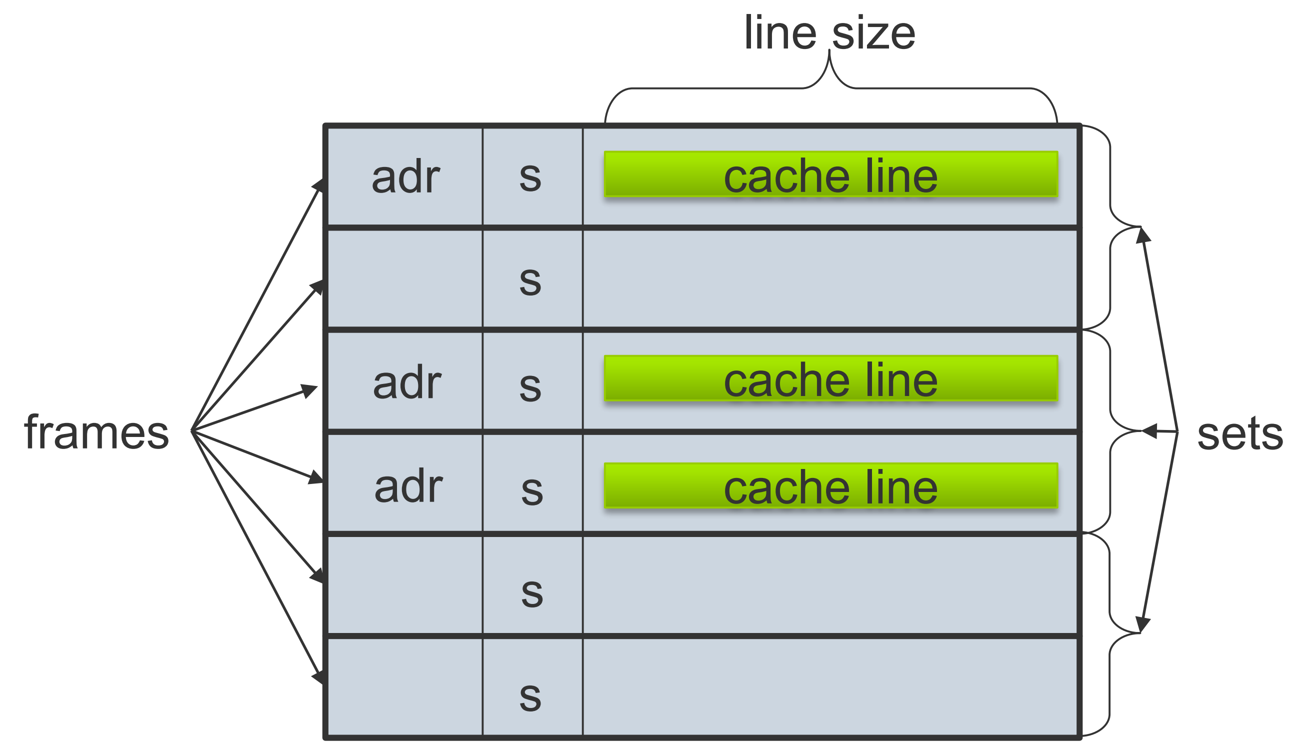
Block size or **line size**

- Number of bytes within a block frame

Block or **cache line**

- The data that fits into a block frame, e.g., 32, 64 or 128 byte

All the block frames can be subdivided into **sets**.



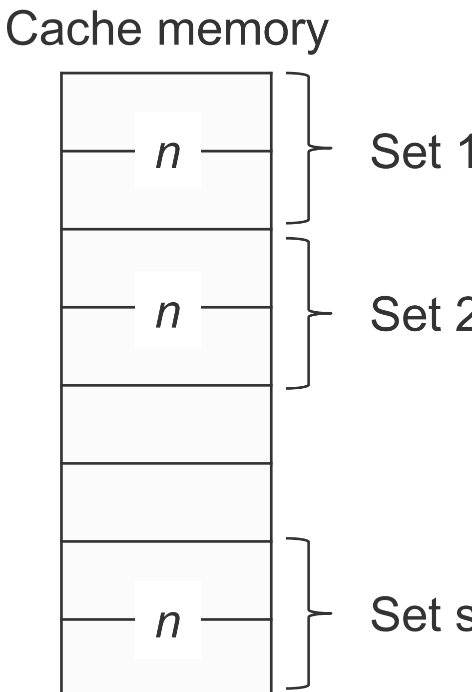
Associativity

Number of block frames per set

The total number c of block frames in a cache is the product of the number s of sets and the associativity n, thus $c = s * n$.

A cache is called

- **fully associative**, if it consists of a single set only ($s = 1$, $n = c$)
- **direct mapped**, if each set consists of a single frame only ($n = 1$, $s = c$)
- **n-way set associative**, otherwise ($s = c / n$)



Replacement policies

In case of a cache miss the controller may have to evict a cache line to make room for the new entry.

- May require a write back into main memory if changed.

The replacement policy determines which entry to evict.

Only fully or n-way set associative caches need a replacement policy.

Typically, caches use simple policies

- Least Recently Used (LRU)
- Least Frequently Used (LFU)
- Optimum: evict the entry that will not be needed for the longest time *in the future*
- https://en.wikipedia.org/wiki/Cache_replacement_policies

Fully associative cache

Main memory

Cache

Tag	Block 63
≈	≈
Tag	Block 1
Tag	Block 0

Associative mapping

Main memory
Block 1023
≈
Block 0

Fully associative cache

Capacity: 256 byte

Block = 4 words = 16 byte

31 **Address** 0

Tag

28

2

2

Byte selection

Word selection

Tag **V** **D**

0

1

2

15

Frame

Valid hit

AND

Tag hit

32 32 32 32

32

Data bus

Fully associative cache

Comparison with all addresses in the address memory of the cache in parallel in a single cycle

Advantage

- A cache line can be placed in an arbitrary frame
- Optimal usage of the cache, choice of replacement policies

Disadvantage

- Requires more hardware (one comparator per frame)
  ➔ feasible only for small caches
- The large flexibility for the mapping requires additional hardware for the replacement policy (which entry to evict if the cache if full)

Direct mapped cache

Each block of the main memory is directly mapped to a certain frame in the cache.

Simple mapping using, e.g., *n* out of *m* bits to determine the frame (*m* bit addresses).

Direct mapped cache

Capacity: 256 byte

Block = 4 words = 16 byte

Address
31
0
Tag
Index
24
4
2
2
Byte selection
Frame selection
Word selection
0
1
2
15
Tag
V
D
Frame
Valid hit
AND
=
Tag hit
32
32
32
32
32
Data bus

Characteristics of direct mapped caches

Advantages

- Very simple hardware implementation
  - A single comparator plus a tag memory

- No replacement policy needed as the mapping is fully determined by a selection of address bits.

Disadvantages

- Eviction of cache entries may be needed even if the cache is not full!

- **Thrashing**
  - If the CPU alternately reads parts of the memory with identical index, those parts evict each other
    - E.g. calling subroutines at C0**0**0, D6**0**0, A2**0**0, C0**0**0, D7**0**0 etc. using bits 4 to 7 as index

N-way set associative cache

N frames together for a set.

Compromise between direct mapped and fully associative cache.

2-way set associative cache

Capacity: 256 byte

Block = 4 words = 16 byte

Characteristics of n-way set associative caches

Increased hit ratio due to the possibility to select an entry

- A replacement policy can selected the entry to evict out of n entries

Replacement policies

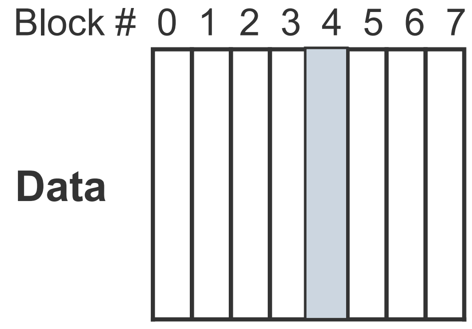
- Similar to fully associative caches
- Could also be FIFO or random instead of LRU

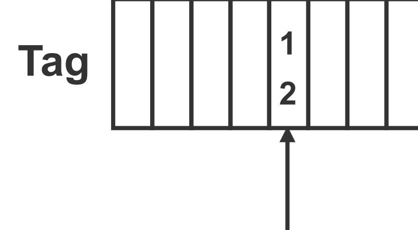
To find an entry a check of all *n* tags in parallel having the same index is necessary

- The effort increases with *n*, for larger *n* the behavior is similar to an fully associative cache
- Thus, this is a compromise between direct mapped and fully associative caches

Example: Organization of a cache with 8 cache lines capacity

Direct mapped

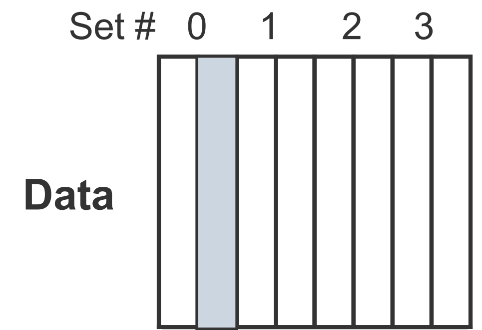


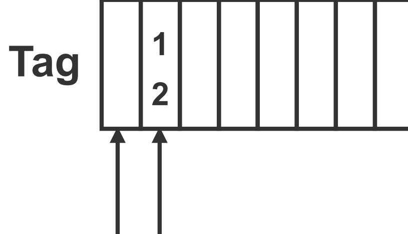


search

only one place for
cache line 12

2-way set associative

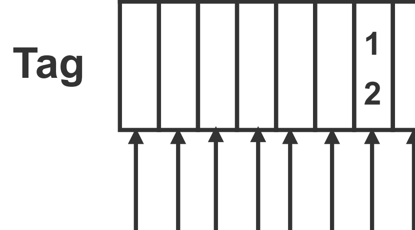




search

two possible places
for cache line 12

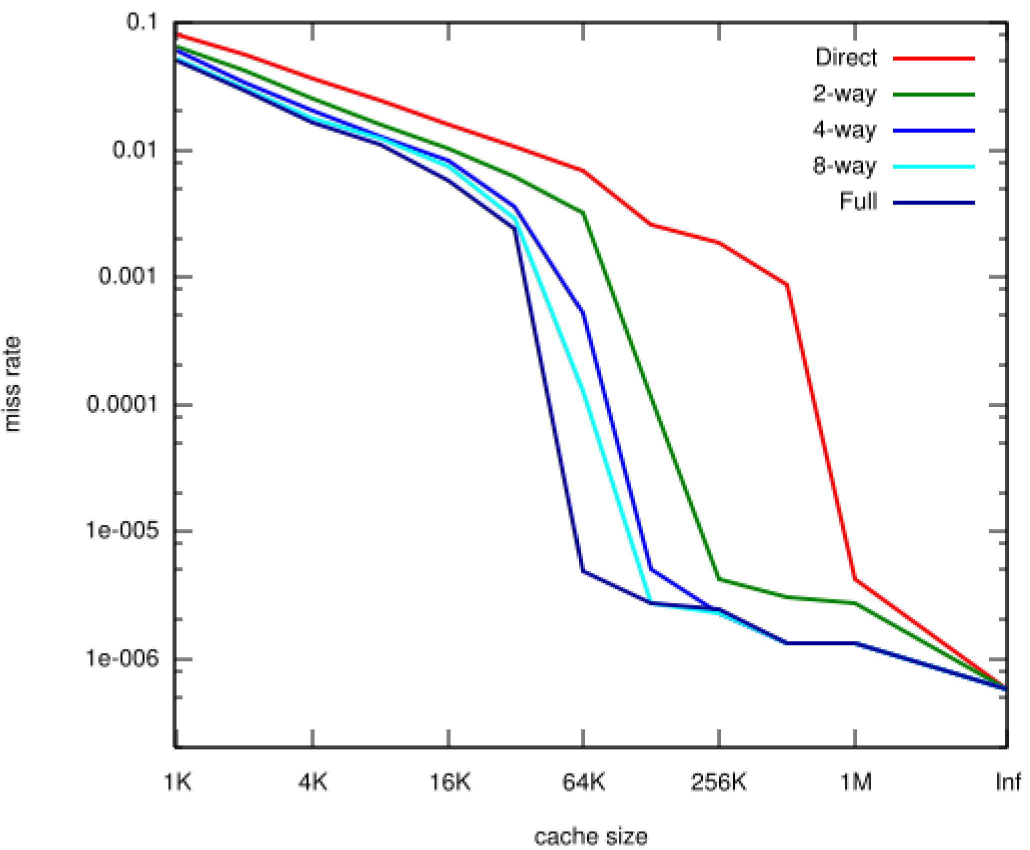
Fully associative



search

cache line 12
can be anywhere

Example: cache hit ratios



Miss rate versus cache size on the Integer portion of SPEC CPU2000
https://www.cs.mcgill.ca/

Some more examples

According to Agarwal, Hennessy and Horowitz:

- A cache hit ratio of 94% can already be achieved with a 64 KiB cache (however, the larger the cache the higher the hit ratio)
- Harvard architecture is helpful for small caches, however do not show big benefits for caches 8 KiB or larger
- For caches 64 KiB or larger direct mapped caches perform similar to 2- or 4-way set associative caches

➡ Only relatively small caches with 32-128 entries are fully associative. Larger caches use direct mapped or 2-/4-way set associative organization.

Be aware: it very much depends on the working set!

Questions & Tasks

- What makes fully associative caches more complex? Why using them at all?
- How could systems avoid thrashing of direct mapped caches?
- What is a typical replacement policy of direct mapped caches?
- What would be the optimal replacement policy?
- What is the trade-off between different n-way set associative caches?

VIRTUAL MEMORY

Virtual Memory: Motivation

Applications typically need more memory than available as physical RAM

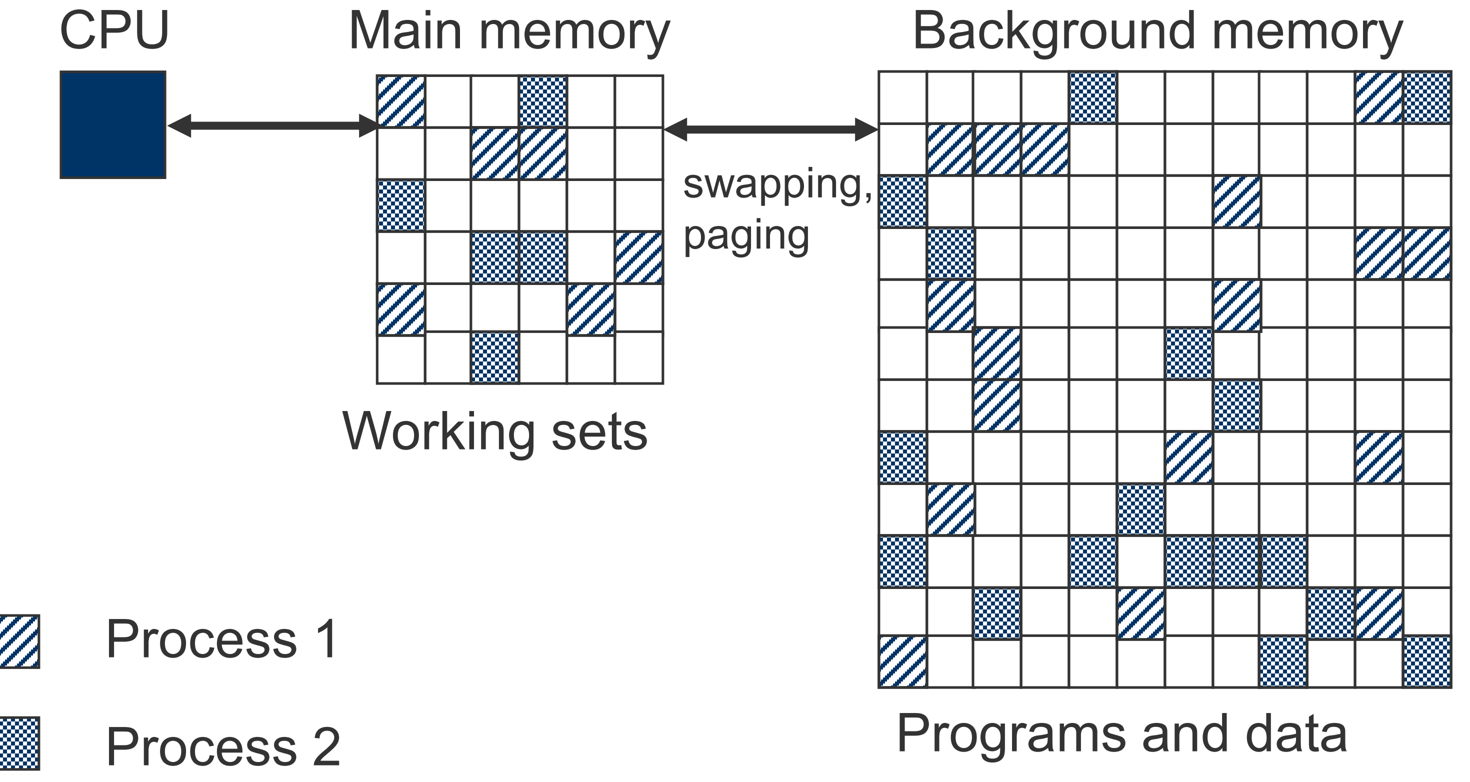
- Especially multi-user, multi-tasking operating systems running many processes in parallel

The operating system has to be able to relocate processes in memory, i.e., fixed physical base addresses are not practical.

Therefore, memory references in instructions of the object code use so-called **virtual** or **logical addresses**. Typically, a memory management unit (**MMU**) translates these addresses into **physical addresses** during program execution.

- A single process sees a single large address space independent of the other processes.
- This supports memory protection among the processes (one process is not allowed to directly access the memory space of another process; simpler systems come with such a MPU, Memory Protection Unit, only).
- https://en.wikipedia.org/wiki/Memory_management_unit

Basic idea of virtual memory management



Virtual memory management 1

Operating system

- Manages all free and used fragments of the memory
- Swaps changed fragments of the main memory to the background memory in case of scarce free memory, deletes unchanged fragments to free memory otherwise
- Swaps required fragments of the background memory into main memory on demand.

Translation tables

- Store all the required address mappings

The whole process is transparent to users

- i.e., the working memory appears to be much larger to a user (process) than the RAM is in reality (thus virtual memory)

MMU (Memory Management Unit)

- Supports the OS management of virtual memory by dedicated hardware
- Fast translation of virtual (logical) addresses to physical addresses

Virtual memory management 2

Efficiency is based on the principle of locality of programs (instructions) and data (operands)

- Programs typically used only a small part of the address space during a short time interval.
- This locality during program execution ensures with high probability that data requested by the CPU is stored in the main memory.

Temporal locality

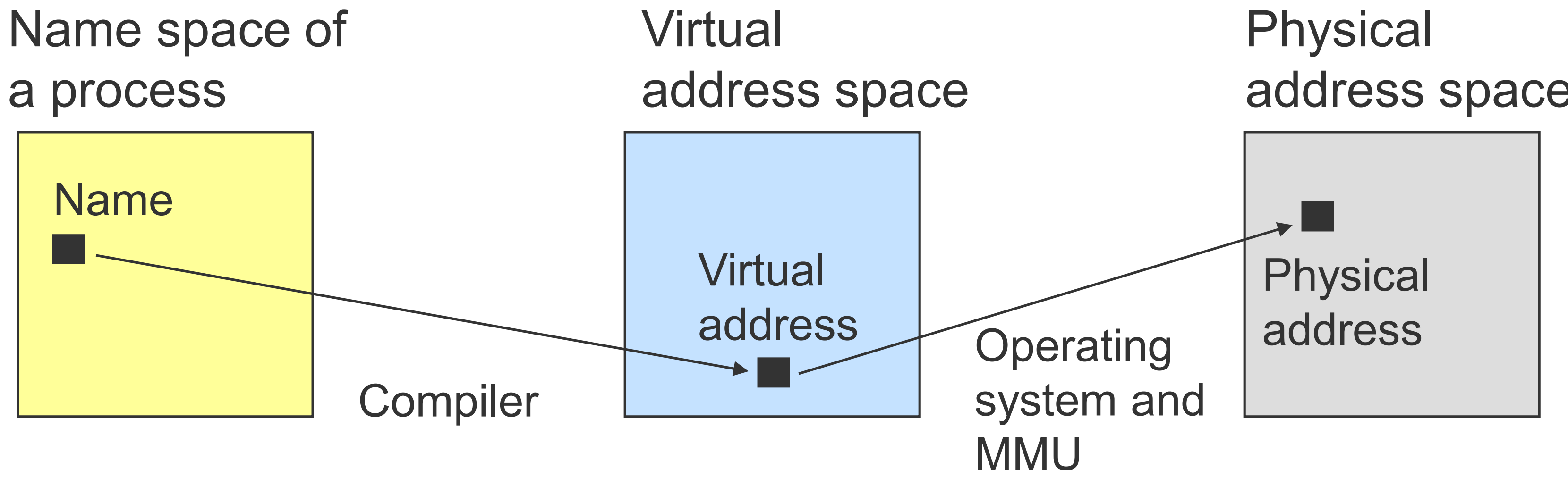
- Data that will be accessed in the near future has already been accessed with high probability (e.g. in loops).

Spatial locality

- A future access to data happens within close storage locations with a high probability (e.g. sequential access).

That's basically the same already discussed in the context of caches - but now with a larger granularity!

Mapping of addresses



User: identifies all objects programs, subroutines, variables, … by names

Compiler: translates all names into virtual (logical) addresses

Virtual memory management: translates virtual addresses into physical addresses during runtime depending on the current memory mapping (current real location of the object in the physical memory)

Mapping of virtual addresses

16 bit addresses, e.g. 2^{16} addressable memory places.

Real memory only 2^{12} byte = 4KiB, 12 bit addresses are enough!

Address space

Address

8191

4096

0

Mapping

4K Main memory

4095

0

Example

Name	JMP somewhere
	↓ Compiler/Assembler
	JMP -128
	↓ Dynamic address calculation ((PC) - 128)
Logical address	4583
	↓ Virtual memory management (MMU)
Physical address	23112

Questions & Tasks

- Considering the memory hierarchy – what does a virtual memory hide?
- What is the role of the OS and the MMU in the context of virtual memory? What else is the MMU good for?
- Compare cache and virtual memory – what do they have in common?

Virtual memory

PAGING

Segmentation and paging

There are two basic mechanisms for virtual memory management:

- **Paging**
- Segmentation – considered legacy (https://en.wikipedia.org/wiki/Memory_segmentation)

Subdivision of the memory into pages

- Subdivide the logical (virtual) and physical address space into segments of fixed length, the so-called pages.
- The pages are relatively small (e.g. 4 KiB, 64 KiB, …) or sometimes large (1 GiB, 16 GiB, …)
- The OS splits a process across several pages without any special context (i.e., a page contains data, no matter what this data represents)
- For more details see OS course!

Architecture	Smallest page size	Larger page sizes
x86 (classical 32 bit)	4 kbyte	2 Mbyte, 4 Mbyte
x86-64 (64 bit)	4 kbyte	2 Mbyte, 1 Gbyte
IA-64 (Itanium, VLIW)	4 kbyte	8 / 64 / 256 kbyte, 1 / 4 / 16 / 256 Mbyte
SPARC v8	4 kbyte	256 kbyte, 16 Mbyte
UltraSPARC	8 kbyte	64 / 512 kbyte, 4 / 32 / 256 Mbyte, 2 / 16 Gbyte
ARMv7	4 kbyte	64 kbyte, 1 / 16 Mbyte
Power	4 kbyte	64 kbyte, 16 Mbyte, 1 Gbyte

Pages in virtual and real memory

Page	Virtual addresses
≈	≈
15	61440 - 65535
14	57344 - 61439
13	53248 - 57343
12	49152 - 53247
11	45056 - 49151
10	40960 - 45055
9	36864 - 40959
8	32768 - 36863
7	28672 - 32767
6	24576 - 28671
5	20480 - 24575
4	16384 - 20479
3	12288 - 16383
2	8192 - 12287
1	4096 - 8191
0	0 - 4095

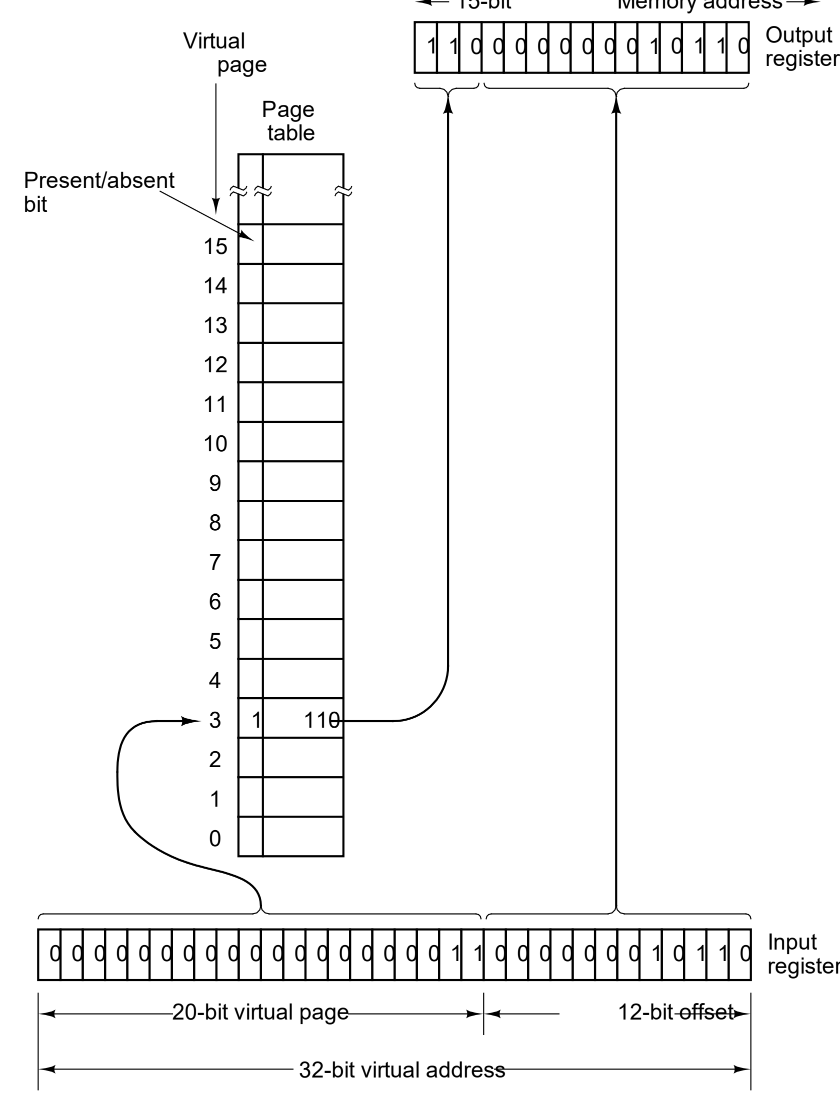
(a)

Mapping of the virtual addresses onto physical addresses!

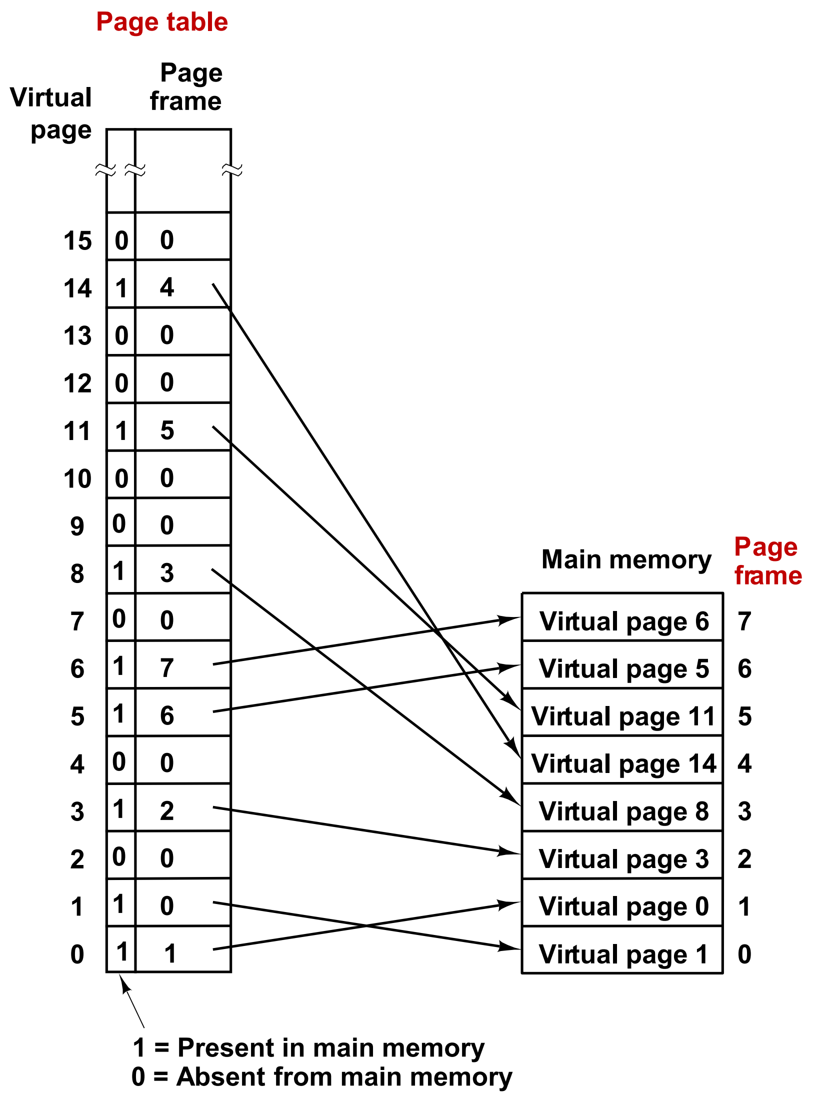
Page frame	Bottom 32K of main memory Physical addresses
7	28672 - 32767
6	24576 - 28671
5	20480 - 24575
4	16384 - 20479
3	12288 - 16383
2	8192 - 12287
1	4096 - 8191
0	0 - 4095

(b)

Translation of virtual to real addresses



Possible mapping of virtual pages



Paging

Advantages

- Small pages allow for a better use of memory (the OS swaps only those parts of a program that are really needed into RAM)
- Less management overhead compared to segmentation

Disadvantages

- More frequent data transfers compared to segmentation

Virtual Memory

CHALLENGES

Challenges for the virtual memory management

Two main challenges for the swapping of data between main memory and background memory

1. When to swap a page?

- When is the best point in time to swap a page into main memory?

- Common approach
  - On demand paging
    - Swap a page into main memory as soon as a process tries to access data stored on this page
    - The access to data located on a page that is currently not stored in the main memory is called a **page fault**.
    - Page faults cause an interrupt that triggers the OS to suspend the faulting process and to swap the required page into main memory.

Challenges for the virtual memory management

2. Which page to replace?

- In case of a full main memory, which page should the OS replace to free some space for a new page?

- The most common approaches:
  - FIFO (first-in-first-out): Replace the oldest page
  - LIFO (last-in-first-out): Replace the newest page
  - LRU (least recently used): Replace the page that has not been referenced for the longest time
  - LFU (least frequently used): Replace the page with the lowest number of references
  - LRD (least reference density): Mixture of LRU and LFU – replace the page with the lowest ratio of references / age of page in main memory
  - Random: As the name indicates – replace an arbitrary page
  - https://en.wikipedia.org/wiki/Page_replacement_algorithm

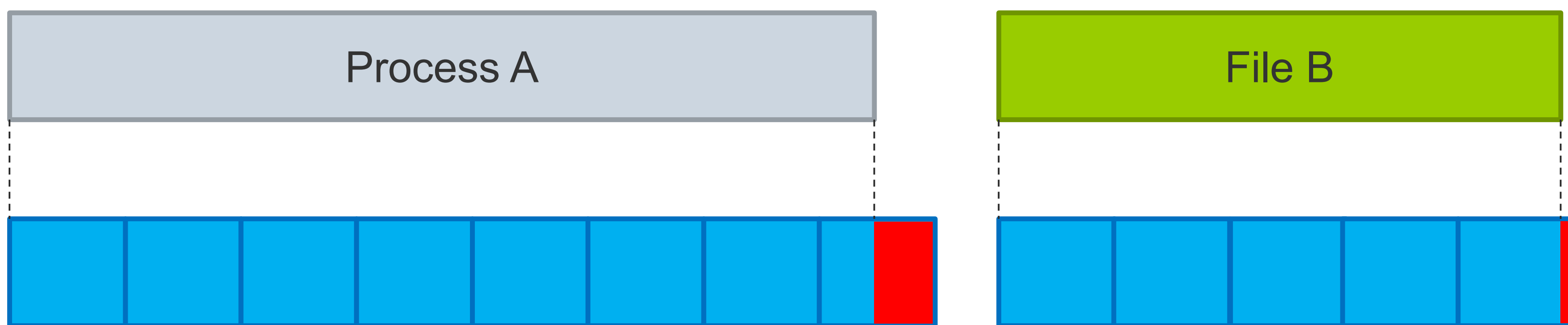
- Additionally, favor unchanged pages
  - No writing back required!

Internal fragmentation

Pages always fit into main memory if there is free space (assuming a fixed page size)

However, depending on the page size and typical size of a process or block of data **internal fragmentation** may happen.

- The OS distributes a process or data in general over several pages
- Assuming a fixed page size the last page of a process will most likely contain unused space



Using different page sizes can give more flexibility but comes with higher complexity.

Questions & Tasks

- What are the pros and cons of small vs. large page sizes?
- What are the two main challenges of paging?
- Who detects a page fault and what happens in case of a page fault?
- In which situations could random replacement be better than LRU?
- Pages can be “pinned”, i.e. never swapped to secondary storage. Which type of content could be a candidate for such non-swappable (or locked/fixed/wired) pages?

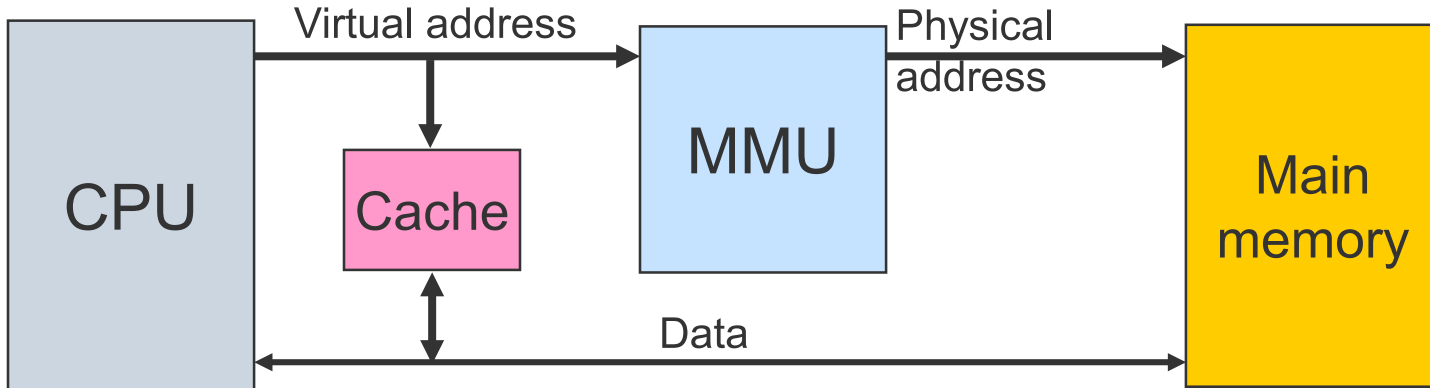
VIRTUAL MEMORY AND CACHES

Location of cache and memory management unit

Two possible ways for the integration of a cache using virtual memory management:

1. Virtual cache

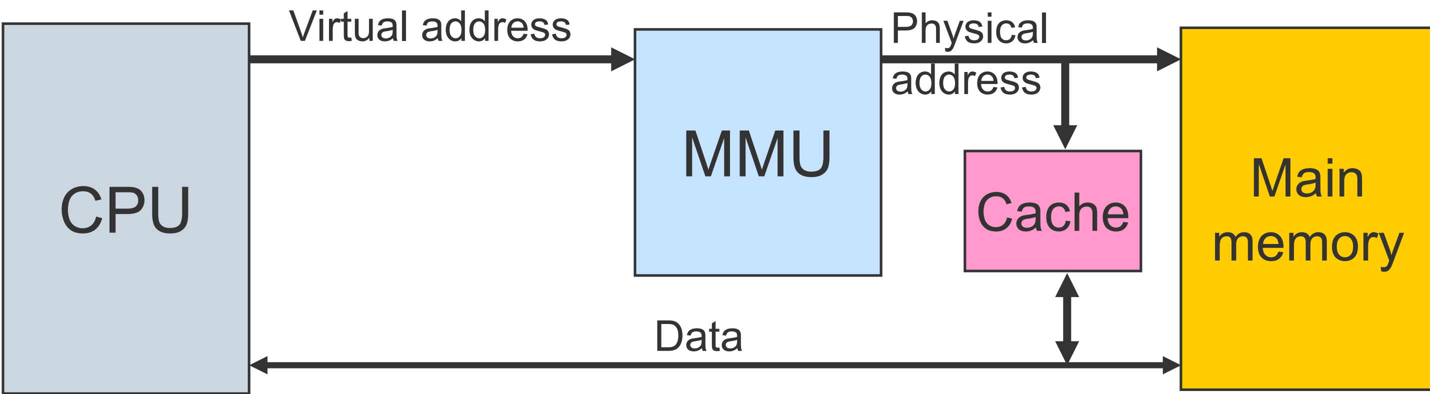
- The cache is located between the CPU and the MMU and operates on virtual addresses, i.e., the cache uses the more significand bits of the virtual address as tags.



Location of cache and memory management unit

2. Physical cache

- The cache is located between MMU and memory and operates on physical addresses, i.e., the cache uses the more significand bits of the physical address as tags



Virtual vs. physical cache

Advantages of a virtual cache

- Cache hits do not require the MMU and, thus, are faster

Advantages of a physical cache

- Often, the physical address uses less bits compared to a virtual address. Therefore, the cache needs to store fewer bits as tag.
- If the MMU is integrated on the CPU chip, only physical cache can be extended.

Memory protection

Many processors offer protection mechanisms to block illegal memory accesses caused by processes during runtime

- integrated features of an MMU or a simplified MPU (Memory Protection Unit)
- The MMU supports the OS for all time-critical functions (fast look-up and translation of addresses, checking legal/illegal memory accesses)

How to protect the memory?

- The MMU knows which process is allowed to access which addresses
- Separation of system software (OS, I/O-system etc.) from user processes
- Separation of different user processes with clear interfaces and controlled methods for data exchange (e.g. via the OS)
- Several protection layers with dedicated access rights.
- More information given in the OS course!

Questions & Tasks

- Many different processes can operate in their respective virtual address spaces in parallel. The OS and MMU separate these address spaces. But what happens with a virtual cache in case of a process (context) switch?
- What is the idea of an MPU and why should even the simplest “thing” in the Internet of Things have one?

MEMORY IN MULTI PROCESSOR SYSTEMS

Basics

Multiprocessor systems or multiprocessor computers belonging to the MIMD class according to Flynn’s classification.

Shared memory multiprocessor systems

- Shared address space between all processors
- Communication and synchronization via shared variables
- Symmetric multiprocessor system
    - Common address space, single global memory
- Distributed shared memory system
    - Common address space, physically distributed memory

Message passing multiprocessor systems

- Physically distributed memory only plus local address spaces for all processors
- Communication and synchronization done via message passing between the processors

N.B.: “memory” here references to the main memory of a system, not caches

Configurations

	Global memory	Physically distributed memory
Shared address space	Processor, Processor, Interconnect, Shared memory Symmetric multiprocessor	Processor, Processor, Local memory, Local memory, Interconnect Distributed shared memory multiprocessor
Distributed address space	Empty	Prozessor, Prozessor, Local memory, Local memory, send, receive, Interconnect Message passing (shared nothing) multiprocessor

Shared memory multiprocessor systems

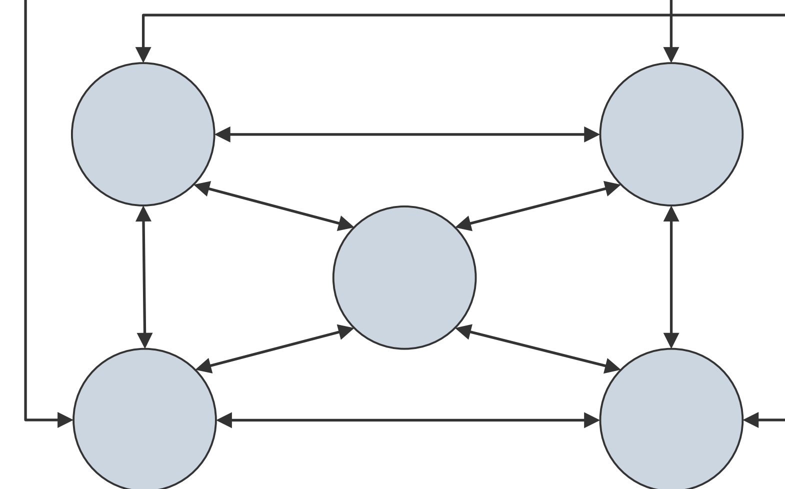
Uniform memory access (UMA)

- All processors access the same common memory.
- Particularly, the **access time** to the common memory is the same for all processors
- Each processor may additionally have a local cache.
- Typical example: symmetric multiprocessing (SMP, https://en.wikipedia.org/wiki/Symmetric_multiprocessing)
- https://en.wikipedia.org/wiki/Uniform_memory_access

Non-uniform memory access (NUMA)

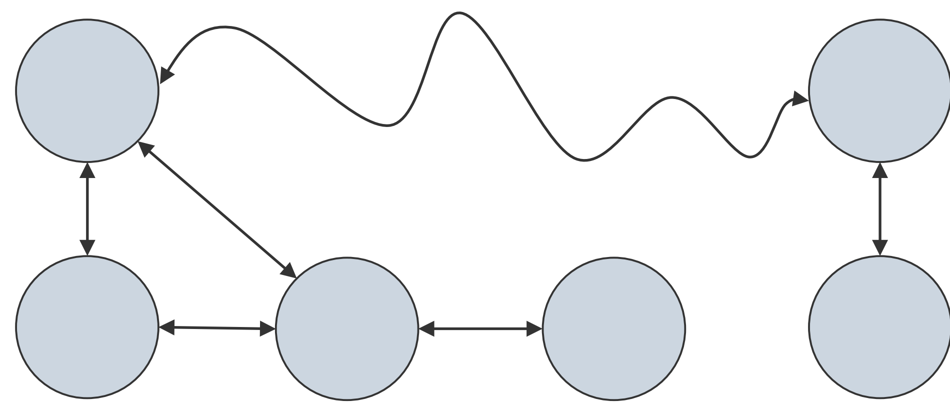
- The memory **access time** depends on the memory location relative to the processor.
    - Access time to local memory lower compared to remote memory access.
- The different memory modules are physically distributed over the computer system.
- Typical example: distributed shared memory systems (DSM)
- https://en.wikipedia.org/wiki/Non-uniform_memory_access

Message passing multiprocessor systems



Similar to shared memory one can distinguish between

- **Uniform communication architecture / uniform message passing**
  - The transmit time of equal sized messages is the same between all processors



- **Non-uniform communication architecture / non-uniform message passing**
  - The transmit time of equal sized messages depends on the sending and receiving processor

Performance metrics for parallel systems

- The overall **execution time *T*** of a parallel program is the time between the start of the program on one of the processors P_i until the final execution of the program on the last processor running that program.
- During program execution all processors are in one of the three states: computing, communicating or idle.

P_1

P_2

P_3

P_4

T

time

Execution time T

The execution time T of a parallel program on a dedicated computer (i.e., we do not consider the OS, multi user, interrupts etc. here) comprises:

- Computation time T_{comp}
  - Time for the “real” operations needed for the problem
- Communication time T_{comm}
  - Time for sending and receiving messages including overheads
- Idle time T_{idle}
  - Waiting time (waiting for sending, receiving, synchronization)

Overall execution time: $T = T_{comp} + T_{comm} + T_{idle}$

Transmission time of a message T_{msg}

The time required for sending a message of a certain size between two (or more) processors.

Each message is comprised of several words or data packets of typically equal size.

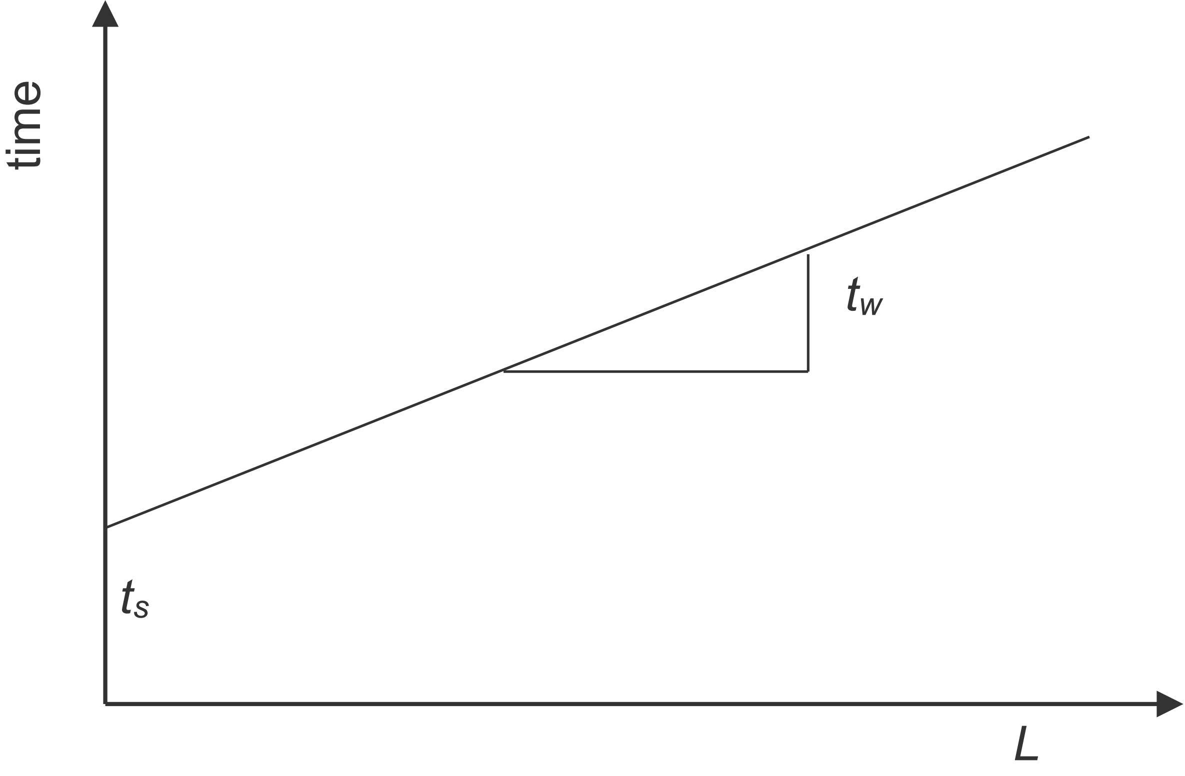
The communication time of a message comprises

- **Message startup time t_s**
    - Initialization of the communication

- **Transfer time t_w** for each transmitted word or data packet
    - Depends on the data rate of the communication medium

Transmission time of a message T_{msg}

The overall transmission time of a message consisting of *L* words: $T_{msg} = t_s + t_w L$



Some definitions

$P(1)$: Number of executed operations of a program on a single processor system.

$P(n)$: Number of executed operations of a program on a multiprocessor system with n processors.

- Both programs, the one on the single processor system and the one on the multiprocessor system have the identical functionality.

- Furthermore, for simplification we assume operations of equal duration.

$T(1)$: Execution time on a single processor system in cycles or time units.

$T(n)$: Execution time on a multi processor system with n processors in cycles or time units.

Basic findings

$T(1) = P(1)$

- Assuming a simple single processor system (not super scalar etc.) a single operation requires on cycle.

$T(\text{n}) \leq P(\text{n})$

- A multi processor system with n processors ($n \geq 2$) can execute in a single cycle more than one operation.

This is all simplified: assuming an average operation, no super scalar processors etc.

Speed-up and efficiency

Speed-up

$$S(n) = \frac{T(1)}{T(n)}$$

Normal case: $1 \leq S(n) \leq n$
Degradation: $S(n) < 1$
Synergy: $S(n) > n$

Efficiency

$$E(n) = \frac{S(n)}{n}$$

Typically: $1/n \leq E(n) \leq 1$

$T(1)$: Execution time on a single processor system in cycles or time units.

$T(n)$: Execution time on a multi processor system with n processors in cycles or time units.

The speed-up depends on the algorithm

The definition of the terms **speed-up** and **efficiency** can be independent of the algorithm or dependent of the algorithm.

Absolute speed-up / absolute efficiency

- Compare the best known sequential algorithm on a single processor system with the best known parallel algorithm on a multiprocessor system

Relative speed-up / relative efficiency

- Use the best known parallel algorithm in a sequential fashion and measure the run-time on a single processor system.
- This includes the required overhead for parallelization such as communication and synchronization (although not needed on a single processor system).

Scalability of parallel computer systems

Adding more computing resources (cores, processors) leads to reduced execution time without changing the program.

Ideal: Linear increase of the speed-up with an efficiency close to 1.

- Example: double the number of processors to half the execution time

Important for the scalability is the problem size.

- If the problem is too small it will not scale.

If the number of processors increases and the problem size is fixed, the system will go into **saturation** at a certain number of processors.

- Scalability is limited – even worse, after saturation **performance degradation** may take place.
- If the problem can be scaled with the number of processors this effect should not happen

Definitions

Overhead for the parallelization:

$$R(n) = \frac{P(n)}{P(1)}$$

Always some overhead for parallel programming:

$$1 \leq R(n)$$

Parallel index *I*(*n*) (How “parallel” is my program?)

$$I(n) = \frac{P(n)}{T(n)}$$

Definitions

Utilization

- Normalized parallel index
- Describes how many operations each Processor executed on average per time unit

$$U(n) = \frac{I(n)}{n}$$

$$= R(n) \cdot E(n) = \frac{P(n)}{n \cdot T(n)}$$

- R(n): Overhead for parallelization
- E(n): Efficiency
- P(n): Number of executed operations of a program on a multiprocessor system with n processors
- T(n): Execution time on a multi processor system with n processors in cycles or time units.

Conclusions

All expressions equal 1 for a single processor (n = 1).

The parallel index gives an upper limit for the speed-up

$$1 \leq S(n) \leq I(n) \leq n$$

The utilization gives an upper limit for the efficiency:

$$\frac{1}{n} \leq E(n) \leq U(n) \leq 1$$

Example I

A (simplified) single processor system requires for the execution of 1000 operations 1000 cycles.

A multiprocessor system with 4 processors may require 1200 operations, but needs only 400 cycles to execute.

Therefore

- $P(1) = T(1) = 1000$, $P(4) = 1200$ and $T(4) = 400$

This results in

- $S(4) = 2.5$ und $E(4) = 0.625$

This means that each processor contributes with 62.5% to the speed-up.

- Again, assuming a problem and algorithm that distributed equally over all processors.

Example II

$I(4) = 3$ und $U(4) = 0.75$

- That means, that on average only 3 processors compute at the same time or each processor is active only 75% of the time.

$R(4) = 1.2$

- The overhead using the parallel computer is 20%, i.e., the computation on the parallel computer requires 20% more operations compared to the computation on the single processor system.

Amdahl's Law (1967, Gene Amdahl)

$$T(n) = T(1) \cdot \frac{1-a}{n} + T(1) \cdot a$$

a = Portion of the program that can only be executed sequentially

$$S(n) = \frac{T(1)}{T(n)} = \frac{T(1)}{T(1) \cdot \frac{1-a}{n} + T(1) \cdot a}$$

$$= \frac{1}{\frac{1-a}{n} + a} = \frac{n}{(1-a) + n \cdot a}$$

$$S(n) \leq \frac{1}{a}$$

Remark: Synchronization and communication ignored!
https://en.wikipedia.org/wiki/Amdahl%27s_law

Discussion of Amdahl’s Law

Following Amdahl’s Law a small portion of sequential operations can limit the speed-up of a parallel computer significantly.

Example:

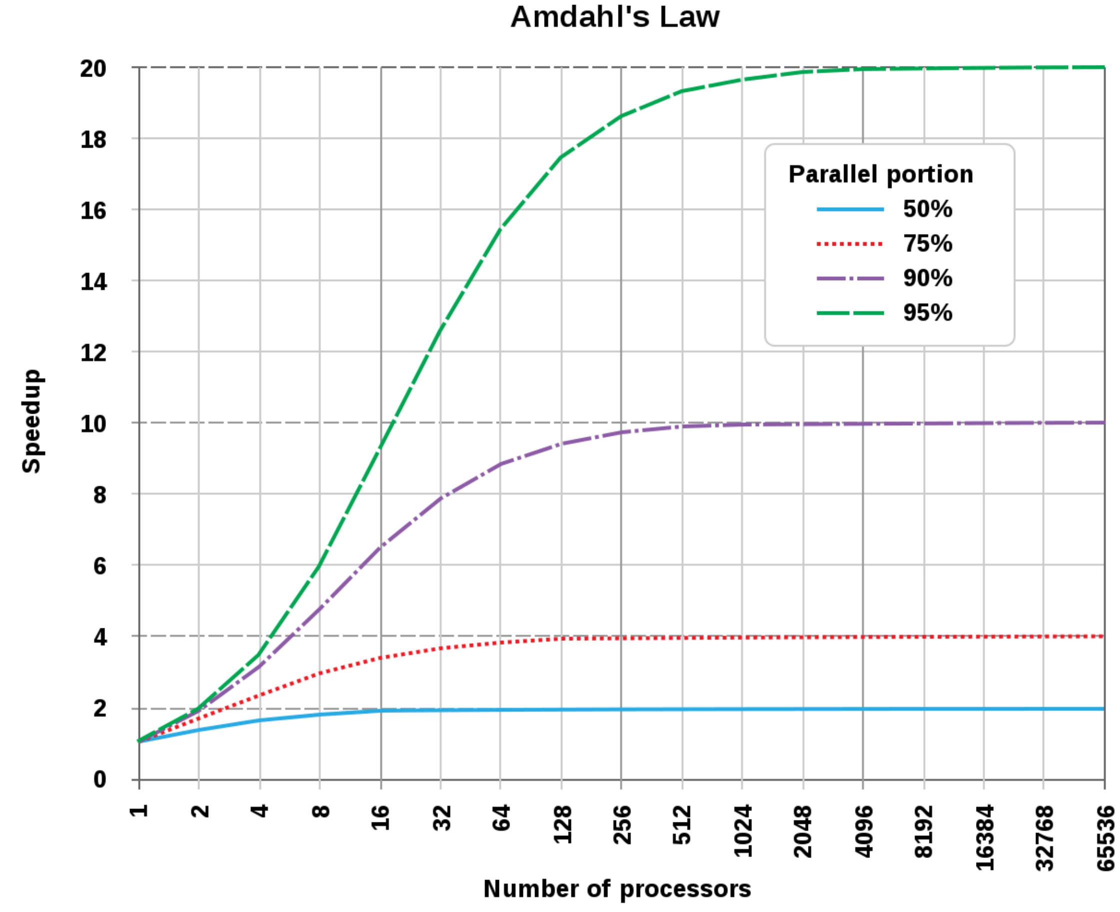
- a = 1/10, i.e. the computer can execute 10% of the program only in a sequential fashion

  ➔ The maximum speed-up of the program can be 10 compared to a purely sequential program.

This can bee seen as an argument against multiprocessor systems

However:

- Many parallel programs have only a very small sequential portion
- Caches not considered



https://en.wikipedia.org/wiki/Amdahl%27s_law#/media/File:AmdahlsLaw.svg

Synergetic effects or super linear speed-up (S>n, E>1)

Theory: a super linear speed-up is impossible

- A single processor system can simulate every parallel algorithm by emulating all the operations of the parallel processors in a loop.

However, a super linear speed-up can be observed in real systems, e.g.,

- A single processor computer can not keep all data in the main memory thus requiring frequent paging.
- Using a multiprocessor system the data is distributed over different main memories and may even fit into the caches thus paging is not necessary and sometimes even fewer main memory accesses.

Consequences

It is difficult to directly compare single and multi processor systems as the latter quite often come with n times the memory for n processors.

If the problem scales, more processors can compute the same problem in less time plus due to higher memory capacity larger problems can be computed.

Quite often theory assumes an infinite memory, but real systems may show a super linear behavior!

Potential problems of multiprocessor systems

Management overhead with an increasing number of processors

Deadlocks due to resource conflicts

Bottlenecks, effects of saturation due to limited resources

Questions & Tasks

- From a programmer’s perspective, what are the main differences between shared memory and message passing multiprocessor systems?
- What are the pros and cons of a single global memory compared to a distributed memory?
- What about scalability when you compare shared memory vs. message passing systems?
- How does the “ideal” problem look like when executed via a highly distributed parallel computer system? Which parameters of the execution time should be optimized?
- Why is there a saturation effect – or sometimes even performance degradation in parallel systems?
- What is a scalable problem?

Memory in multi processor systems

INTERCONNECTS

Interconnects

Connection between processors and/or memories

Static networks

- Hard-wired interconnects between different pairs of nodes (processor, memory) of a network

Dynamic networks

- Contain some switching component that interconnects all nodes
- There is no hard-wired connection between two components

Classification of interconnects

Interconnect

static

dynamic

One dimensional

Two dimensional

Three dimensional

N-dimensional

…

Bus

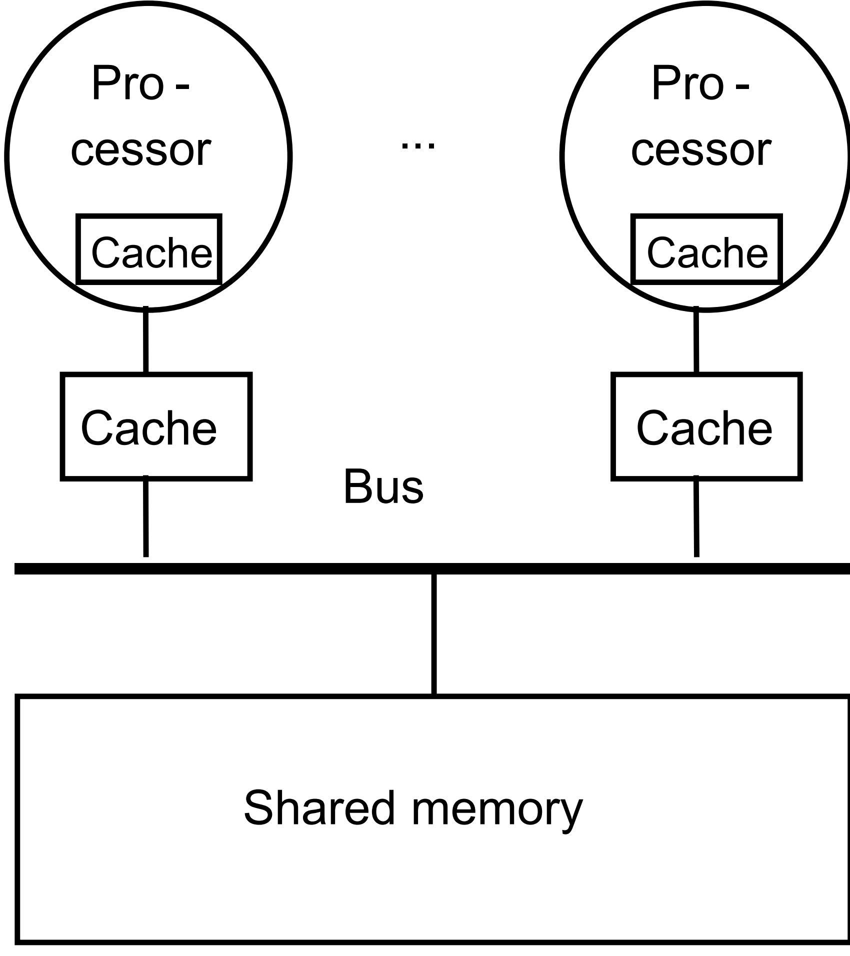
Crossbar switch

Banyan network

…

A classical dynamic interconnect: bus

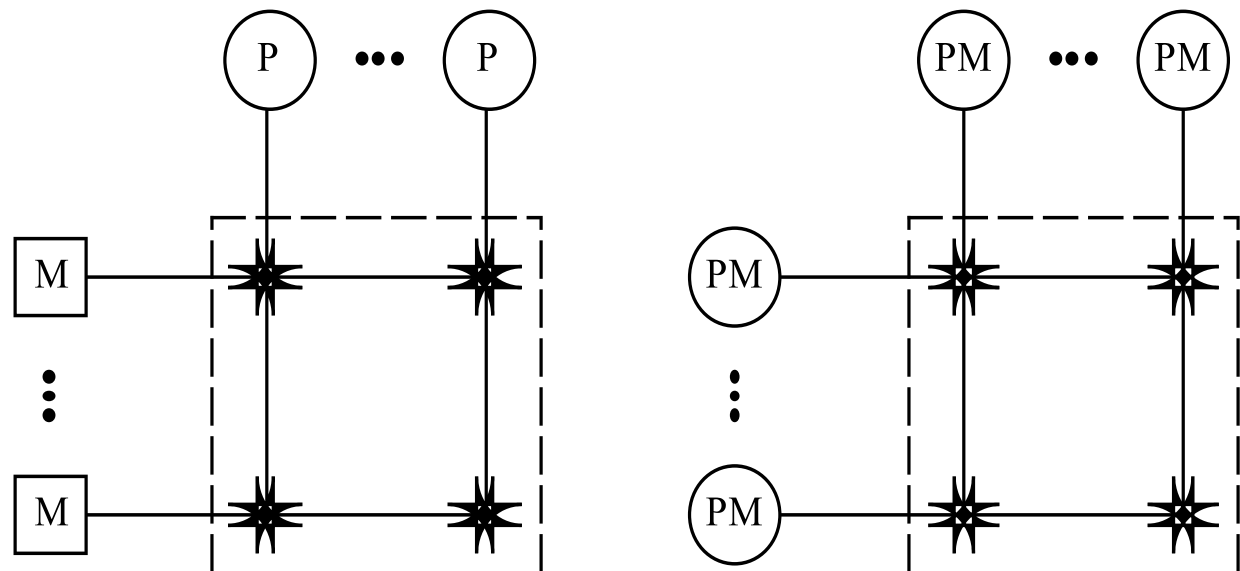
Shared memory multiprocessor with a single bus and local caches



Dynamic high performance interconnect: crossbar switch

A crossbar switch

- Hardware that enables the communication between disjoint pairs of components (processors and/or memories) to communicate simultaneously without blocking.
- A single switch in the switching matrix connects dynamically two components for a certain time.
- An *n* x *n* crossbar switch allows for the simultaneous, non-blocking communication of *n* disjoint sender/receiver pairs.



Memory in multi processor systems

CACHE COHERENCE

The cache coherence problem

What happens if several components read and write the content of the same addresses in main memory and/or caches?

- Which write is valid? What to read after several writes? Does this depend on the memory or cache?

Coherence

- Defines the behavior of reads and writes to a single address (word, variable, memory location)
- The computer system has to proceed in a deterministic way by guaranteeing that each read operation always fetches the most current update of the content at a single address location.
- Outdated content, eventually stored in a cache or main memory, must not be used!
- However, the content of a single address location stored at several caches and the main memory may be inconsistent (as long as no processor reads this data…)
- https://en.wikipedia.org/wiki/Cache_coherence

Consistency

- A memory system is consistent if all copies of the content of a single address are identical at all locations (i.e. all caches and the main memory)
- This guarantees coherence, but at a high effort …

Why differentiating between consistency and coherence?

As soon as a processor updates a variable in the cache only and not in main memory a data inconsistency occurs between main memory (old data) and cache (updated data).

- This is caused by the more efficient write-back cache policy that tries to minimize main memory access.
- The alternative write through policy avoids this problem, but puts a heavy load on the data bus and main memory.

In order to keep all copies of a variable/memory word consistent (in all caches and main memory) the system would require many (potentially useless) update operations!

Trick

- Allow inconsistencies in a limited fashion.
- Use a dedicated so-called cache coherence protocol to ensure cache coherence.
- This protocol has to ensure that the system always reads the newest content of a variable and never outdated content.

Cache coherence protocols

Write update protocol

- As soon as one processor updates a copy of the content of single address location (e.g. a variable) in a cache the system has to update all copies in all other memories
- The system may delay this updating up to an access to the content of this address
- This requires sending the data over an interconnect to the other memory location

Write invalidate protocol

- Before changing the copy of data in a cache memory the system declares all other copies in caches as invalid
- No direct updates, but … see MESI!

Symmetric multiprocessor systems quite often use a write invalidate cache coherence protocol with write back caches. This puts only minimal load on the system interconnect.

The MESI protocol: basics

Snooping

- All caches monitor all read and write accesses
  on a common bus (or other type of broadcasting interconnect)

The snooping logic monitors all addresses other processors put on the common bus

If the snooped address matches with an address stored in the cache

- In case of a snooped write and the cache entry is unmodified (only read so far)
  - Declare cache entry as invalid
- In case of a snooped read or write and the cache entry is modified
  - Snoop logic takes over bus control, writes back the cache entry to main memory and then allows the snooped transaction to proceed

MESI – the foundation of the most prominent write invalidate protocols (comes in many flavors)

- https://en.wikipedia.org/wiki/MESI_protocol

The MESI protocol: states

The following states of a cache line form the acronym MESI:

- (Exclusive) **M**odified
  - A write modified the cache line from the content in main memory (i.e. the line is *dirty*)
  - The cache line is present in this cache only
  - A write back to main memory is required before any other cache reads the cache line from main memory
- **E**xclusive unmodified
  - The cache line is present in this cache only and matches the content in main memory (i.e. it is *clean*)
  - A read access transferred the cache line from main memory
- **S**hared (unmodified)
  - Copies of the cache line are in more than one cache for read access
  - But all copies of the cache line still match the content in main memory (i.e. they are all *clean*)
- **I**nvalid
  - The cache line is invalid or unused

The MESI state diagram – two bits per cache line in each cache

RH	Read hit
RMS	Read miss, shared
RME	Read miss, exclusive
WH	Write hit
WM	Write miss
SHR	Snoop hit on a read
SHW	Snoop hit on a write or read-with-intent-to modify

Dirty line copyback

Invalidate transaction

Read-with-intent-to-modify

Cache line fill

Invalid

Shared unmodified

Exclusive modified

Exclusive unmodified

SHR

RH

SHW

RMS

RME

WH

WM

SHW

SHR

SHW

SHR

RH

WH

WH

RH

MESI example

```
ld 4711
st 4711
```

```
ld 4711
ld 4711
```

RH	Read hit
RMS	Read miss, shared
RME	Read miss, exclusive
WH	Write hit
WM	Write miss
SHR	Snoop hit on a read
SHW	Snoop hit on a write or read-with-intent-to modify

P1

P2

1

1

Shared unmodified

Shared unmodified

SHR

RMS

1

4711

Main memory

Yet another MESI example I

I
CPU1 CPU2 CPU3 CPU4 CPU5
A
Exclusive

CPU 1 **reads** block A into cache

II
CPU1 CPU2 CPU3 CPU4 CPU5
A A
Shared Shared

CPU 2 **reads** block A
⇒ CPU 1 broadcasts on bus that it holds a copy of A

III
CPU1 CPU2 CPU3 CPU4 CPU5
A
Modified

CPU 2 **writes** block A (only in cache, not in main memory!)
⇒ CPU 2 broadcasts an invalid signal

Yet another MESI example II

IV

CPU1 CPU2 CPU3 CPU4 CPU5

A Shared A Shared

CPU 3 **reads** block A
⇒ CPU2 snoops write access
⇒ CPU2 stops CPU3 via a signal
⇒ CPU2 writes back to main memory,
CPU3 reads from main memory

V

CPU1 CPU2 CPU3 CPU4 CPU5

A Modified

CPU 2 **writes** block A

VI

CPU1 CPU2 CPU3 CPU4 CPU5

Modified A Invalid

CPU 1 **writes** block A
⇒ CPU2 stops CPU1 via a signal
⇒ CPU2 writes back to main memory
⇒ CPU1 reads from main memory
with intend to modify

Questions & Tasks

- What are the pros and cons of static vs. dynamic interconnects?
- What would cache consistency mean for the system?
- Why is a write invalidate protocol so efficient?
- What is a prerequisite for snooping to be effective?
- Why is there no *shared modified* state in MESI? What would this mean?
- Why to distinguish between *shared* and *exclusive unmodified*? Why not simply a state *unmodified*?
- MESI has no problems if two consecutive writes take place on the same cache line done by two different processors. What can happen? Who has to solve potential problems?

Configurations

	Global memory	Physically distributed memory
Shared address space	Processor, Processor — Interconnect — Shared memory Symmetric multiprocessor	Processor, Processor — Local memory, Local memory — Interconnect Distributed shared memory multiprocessor
Distributed address space	Empty	Prozessor, Prozessor — Local memory, Local memory — send, receive — Interconnect Message passing (shared nothing) multiprocessor

Distributed shared memory multiprocessor systems

DSM (Distributed shared memory)

- All processors share a single address space, i.e. the same physical address on two different processors belong to the same location in memory
- However, the systems distributes the memory modules over all processors
- https://en.wikipedia.org/wiki/Distributed_shared_memory

Consequence

- The access to local memory is typically much faster than the access to remote memory (i.e. the local memory of another processor) $\Rightarrow$ Belongs to the NUMA class
  - https://en.wikipedia.org/wiki/Non-uniform_memory_access

Typically, the processors have one or more levels of cache memory (plus sometimes a non-shared private memory).

- It is quite common to have cache coherent NUMA systems (ccNUMA)
  - Cache coherence via explicit communication between cache controllers or versions of MESI
- Non cache coherent NUMA (nccNUMA) is simpler to build, but more difficult to program

ccNUMA, COMA, nccNUMA (I)

ccNUMA (cache coherent Non-Uniform Memory Access):

- All caches of the system are coherent
- If the access to remote data causes a local cache miss the system transfers the remote data to the local cache
- MESI-type cache coherence protocols / directory-based protocols required
- See, e.g., SCI (scalable coherent interface) and several processor architectures (often historical)

COMA (Cache-Only Memory Architecture):

- Special case of ccNUMA: no more (distributed) main memory with copies of data in caches and address space allocated to main memory
- If needed, the system migrates data to the cache of the processor requesting the data (i.e. the initial placement of data changes over time)
- See experimental computers like Data Diffusion Machine, KSR-2 in the 80s/90s

ccNUMA, COMA, nccNUMA (II)

nccNUMA (non cache coherent Non-Uniform Memory Access)

- No system-wide cache coherence, no cache coherence protocol
  - I.e. no cache coherence protocol between different processors
  - However, local cache coherence between local cache and main memory
- Much simpler hardware, no snooping or directory-based protocols
- Different commands for access to local data (in the cache) and remote data (in other caches or main memory)
- Requests for remote data bypass the local cache
- Coherence of a program must be ensured by software (e.g. mutexes)!

Two types of distributed shared memory multiprocessor systems

Access to memory is transparent from a program's point of view

- Common for ccNUMA systems
- Specialized hardware must distinguish between local memory access and remote memory access

Remote access to memory requires special commands

- Common for nccNUMA systems
- Requires the extension of the instruction set of a processor

Configurations

Message passing multiprocessor systems

There are no shared memories or common address spaces in message passing systems.

Communication happens via message exchange over an interconnect and/or other processors.

All processors have a local memory only.

Processor nodes typically connect to other nodes via point-to-point connections.

Scalability is (in theory) unlimited.

Efficient support of parallel computing is on program/process level.

Parallel computing on thread or instruction level is not efficient due to the large overheads.

Distributed Systems / distributed computing

One step further away from the concrete hardware system and network topology.

A distributed system is a set of networked computers which communicate and coordinate via message passing to achieve a common goal.

- Distributed program / distributed programming

Much looser coupled system compared to parallel computers.

Often Internet technology used for communication.

- TCP/IP protocol family

Different architectures possible

- Client-server, peer-to-peer, n-tier

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

Advantages of distributed systems

Reuse of unused/underutilized computers (huge potential!)

Better use of existing resources (large, distributed memory)

Use of heterogeneous hardware possible (even specialized components possible)

- Sub-programs can be placed on dedicated, specialized computers (e.g. graphics, data bases etc.)

It is relatively easy to extend a **virtual parallel computer** based on several servers/workstations/computers in general by more and more other computers.

Standard computers can be used for program development.

Standardized interfaces for message passing exist for distributed systems and parallel computers providing source code compatibility

- Started with PVM: parallel virtual machine, https://en.wikipedia.org/wiki/Parallel_Virtual_Machine
- then MPI: message-passing interface, https://en.wikipedia.org/wiki/Message_Passing_Interface

Fault tolerance: faulting machines will/should not bring down the whole distributed system

Disadvantages of distributed systems

More difficult to ensure security

- Difficult to operate on distributed, encrypted data using different OS, hardware, operators, …

Difficult to give certain QoS (Quality of Service) guarantees (delays, computation time etc.) due to heterogeneous networks and computing hardware

Use of (relatively) high-level interfaces (e.g. socket interfaces using UDP/TCP)

Relatively slow communication over the network

- Supports only coarse grained parallelism with little communication
- Specialized communication hardware exists, but often replaced by Ethernet

Newer technologies/interfaces exist (sometimes for dedicated purposes)

- E.g. Hadoop (https://en.wikipedia.org/wiki/Apache_Hadoop),
  Spark (https://en.wikipedia.org/wiki/Apache_Spark), Flink (https://en.wikipedia.org/wiki/Apache_Flink)

Grid computing

Widely distributed computers used for a common task.

- Computing resources as easy to access as electrical power (“power grid”)
- (Autonomous) resources loosely coupled via the Internet (or dedicated high performance connections)
- Different (international, public/private) organizations (or individuals) form a virtual organization for the common task
- Typically consists of many super-computer centers interconnected
- Examples: grand challenge problems such as earthquake predictions, protein folding, search for extraterrestrials… but also CERN, Bitcoin
- https://en.wikipedia.org/wiki/Grid_computing

Cloud computing

- Also widely distributed data centers, but they typically belong to one organization
- Used for storage and computation
- https://en.wikipedia.org/wiki/Cloud_computing

Edge computing

- The cloud comes closer to the customer to reduce latency
- https://en.wikipedia.org/wiki/Edge_computing

…and some more computer systems

Embedded Systems

- Hardware and Software are components of a bigger system, e.g. plant and process control, robotics, dish washer, ventilator ...

ICT (Information and Communications Technology)

- Hardware and Software are main components of communication networks, multimedia equipment, mobile phones, ...
- https://en.wikipedia.org/wiki/Information_and_communications_technology

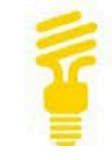
Operational Technology

- Use of Hardware and Software for process and asset monitoring in industry
- Industrial control systems like PLC (Programmable Logic Controller), SCADA (Supervisory control and data acquisition), Building Automation Systems…
- https://en.wikipedia.org/wiki/Operational_technology

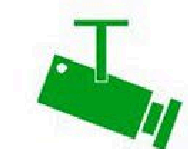
The Internet of Things

Cars, animals, people, monitoring

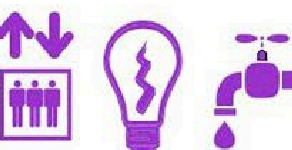
Agriculture



Energy



Security, surveillance



Building automation

Embedded systems

Internet of things

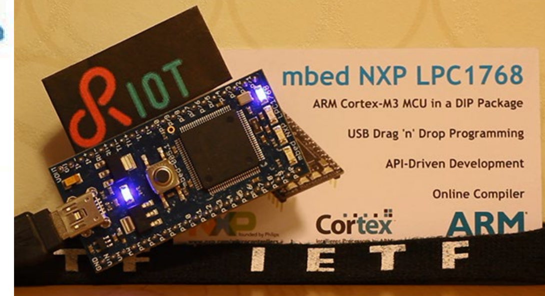
Machine-to-machine com., Sensor networks

Everyday things

Smart Home / City

Health care

Source: The Telecare Blog, thetelecareblog.blogspot.de, 24.10.14



Source: RIOT OS, www.riot-os.org
1,5 kByte RAM, 5 kByte ROM, real-time, multi-threaded

➡ **One common technology for everything!**

Questions & Tasks

- What is the difference between a symmetric multiprocessor and a distributed shared multiprocessor? What do they have in common? Pros and cons?
- What are the advantages and disadvantages of cache coherence?
- How to really scale computer systems?
- What type of problems are well suited for distributed systems and when to use a super computer?
- What are pros and cons of distributed systems?
- The terms are somewhat fuzzy, but what distinguishes grid from cloud and edge computing?
- What is specific about embedded systems?

Summary

Memory hierarchy

Main memory

Cache memory

Virtual memory

Multiprocessor systems

Distributed systems

Content

1. Introduction
   - Single Processor Systems
   - Historical overview
   - Six-level computer architecture

2. Data representation and Computer arithmetic
   - Data and number representation
   - Basic arithmetic

3. Microarchitecture
   - Microprocessor architecture
   - Microprogramming
   - Pipelining

4. Instruction Set Architecture
   - CISC vs. RISC
   - Data types, Addressing, Instructions
   - Assembler

5. **Memories**
   - **Hierarchy, Types**
   - **Physical & Virtual Memory**
   - **Segmentation & Paging**
   - **Caches**