IMM

Application Tuning

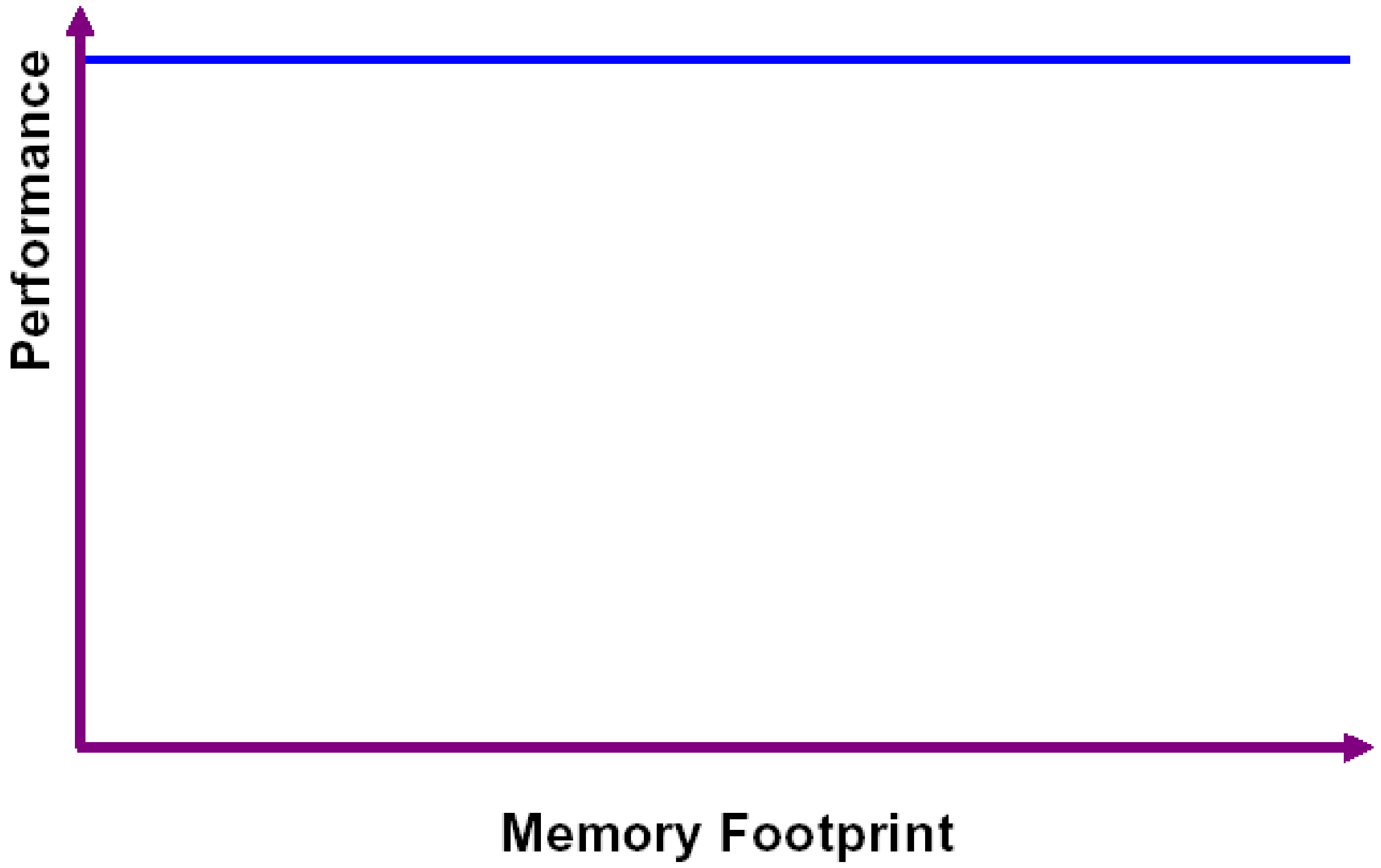
Memory and caches in modern computers

DTU

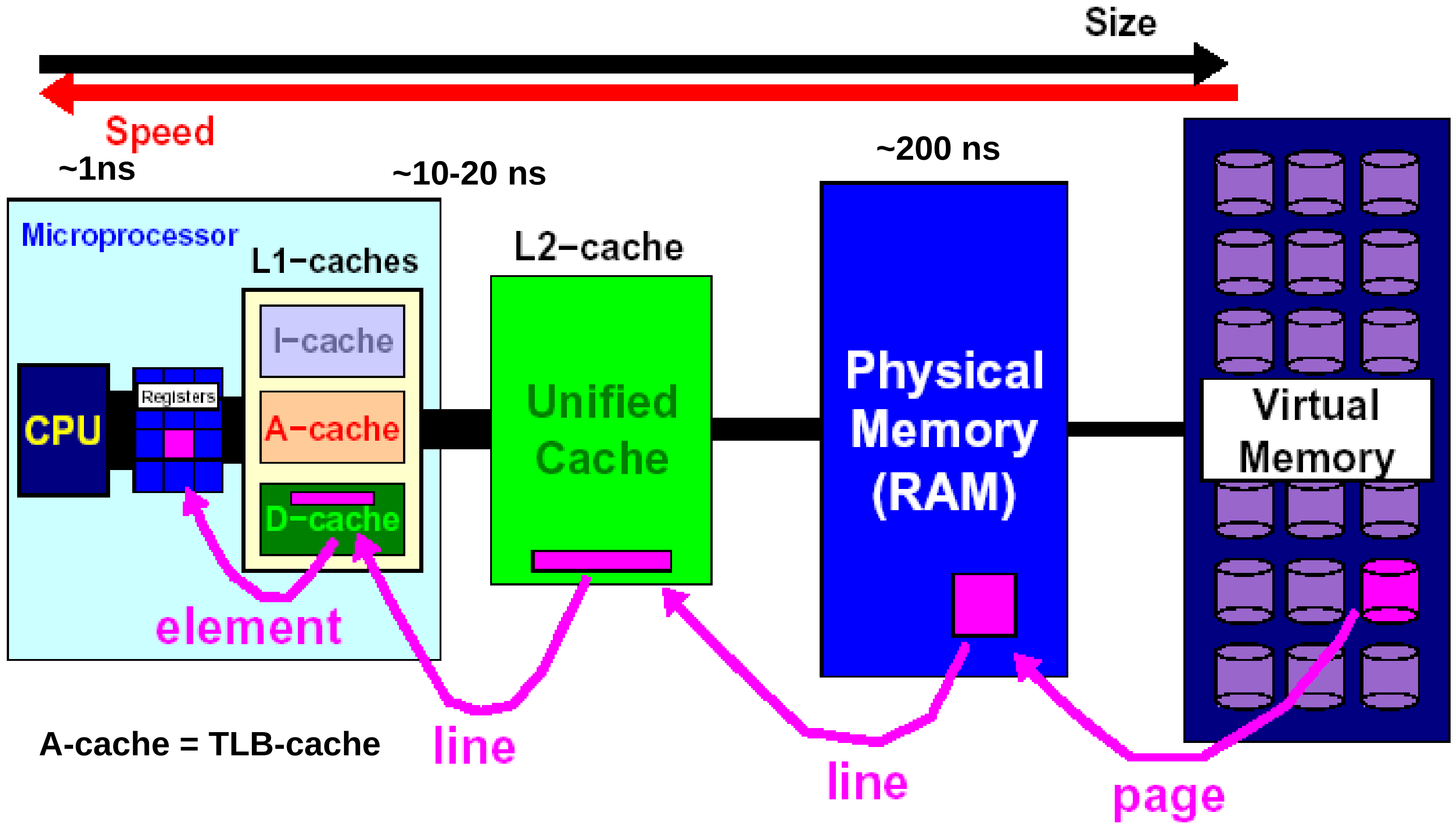
The Memory Hierarchy

The Memory Hierarchy

Intuitive Performance Graph:



The Memory Hierarchy

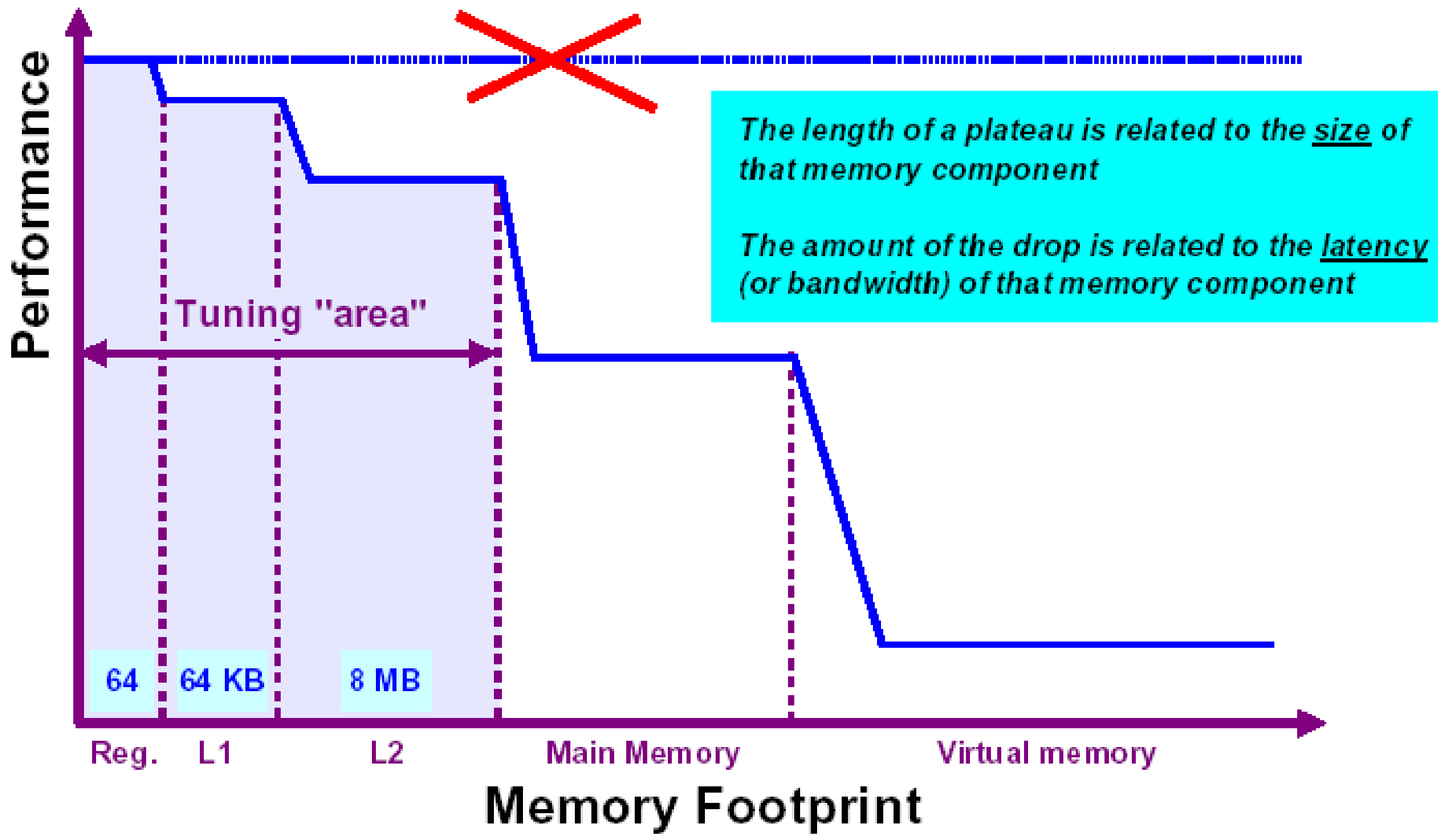


A-cache = TLB-cache

Memory Optimization:
Keep frequently used data close to the processor

The Memory Hierarchy

Performance is not uniform:



The length of a plateau is related to the __size__ of that memory component

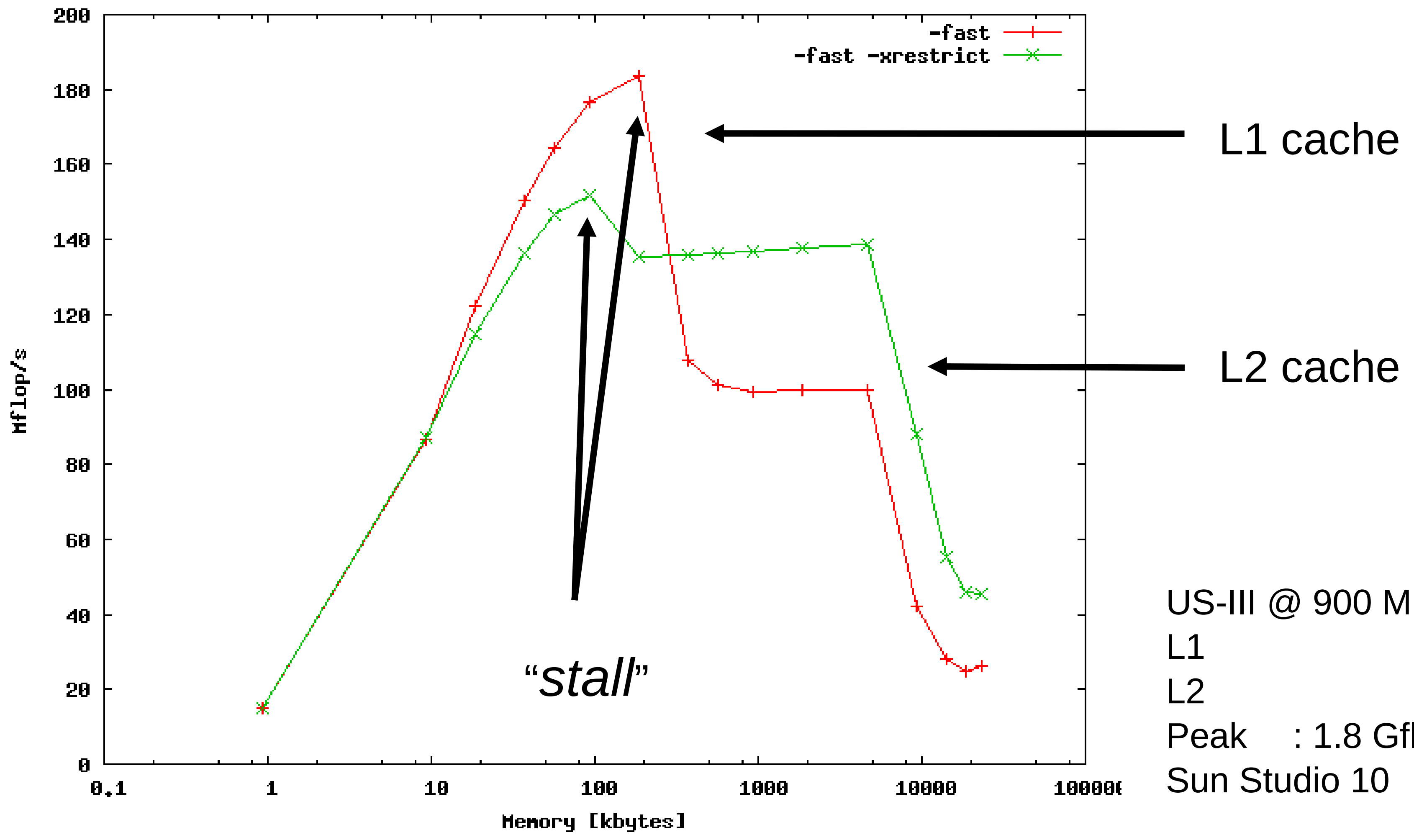
The amount of the drop is related to the __latency__ (or bandwidth) of that memory component

The Memory Hierarchy

- Memory plays a crucial role in performance
- Not accessing memory in the right way will degrade performance on **all** computer systems
- The extent of degradation will depend on the system
- Knowledge about the relevant memory characteristics helps to write code that minimizes those problems

Cache effects in real applications

Vector addition example: performance drops visible



US-III @ 900 MHz

L1	: 64 kB
L2	: 8 MB
Peak	: 1.8 Gflop/s

Sun Studio 10

Caches – and all that ...

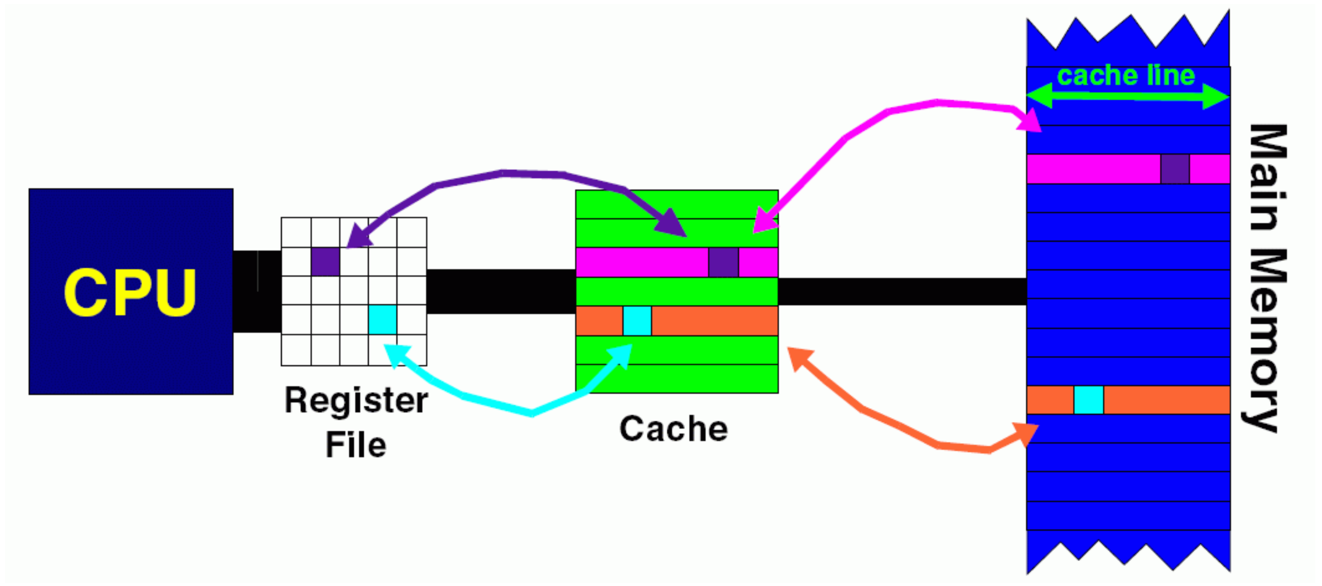
How do those caches work?

Caches

- Cache memory or cache for short (from French: cacher – to hide): fast buffers that help to hide the memory latency
- One distinguishes between
  - data cache
  - instruction cache
  - address cache (also called TLB – Translation Lookaside Buffer) – mapping between virtual and physical adresses

Cache Lines

- To get good performance, optimal use of the caches is crucial
- The unit of transfer is a "*cache line*":
  - linear structure of fixed length (bytes)
  - fixed starting address in memory



Cache Organisation

Direct Mapped:

- Each memory address maps onto exactly one line in cache
- simple and efficient
- built-in replacement policy
- easy to scale to larger sizes
- downside: no control by usage – danger of replacing data that will be needed again soon

Cache Organisation

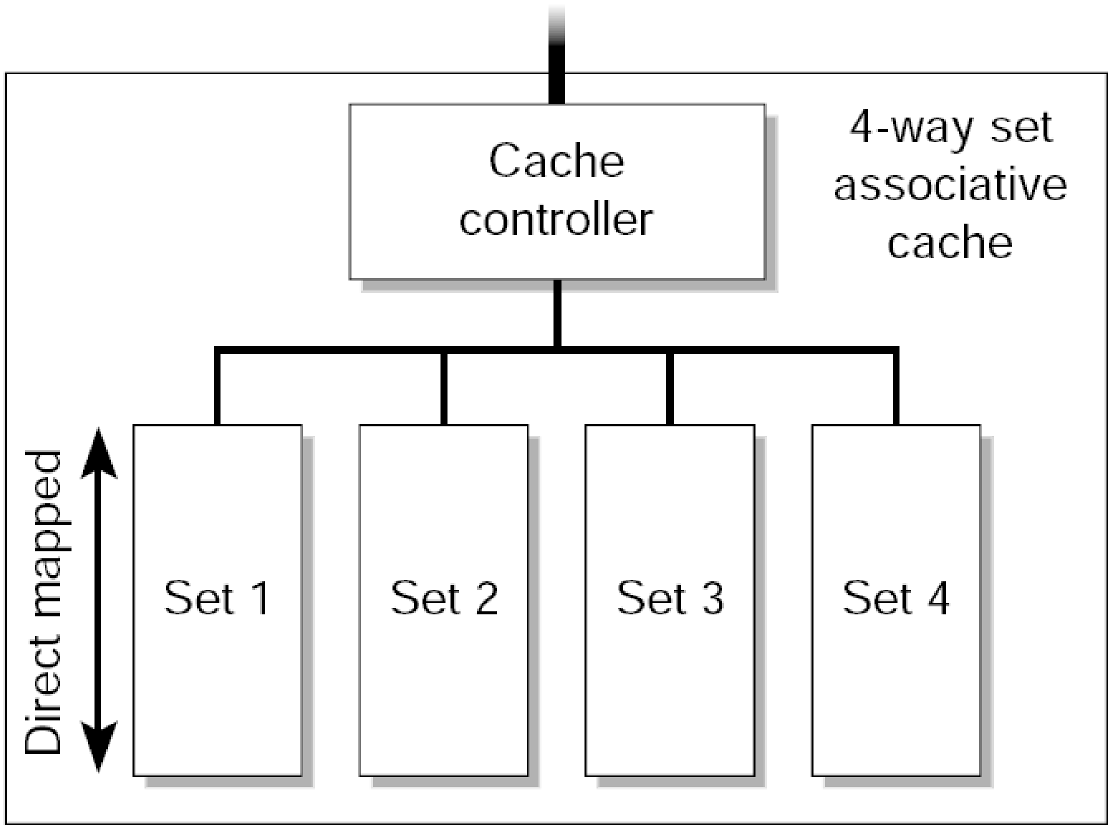
Fully Associative:

- Every memory address can be mapped anywhere in cache
- Need to track usage of cache lines
- Requires a replacement policy, e.g. *least recent used* (LRU), *least frequent used* (LFU), random, etc
- Doesn't scale well to large sizes
- Costly design

Cache Organisation

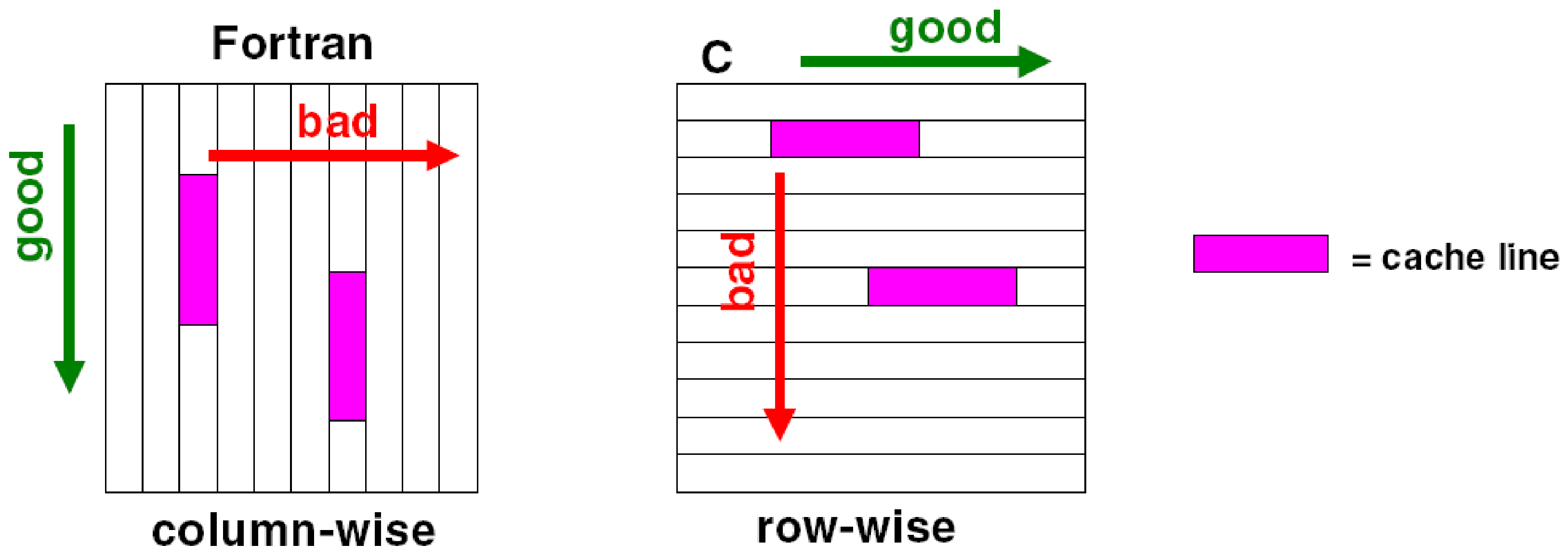
N-way Set Associative:

- Sets of direct mapped caches:



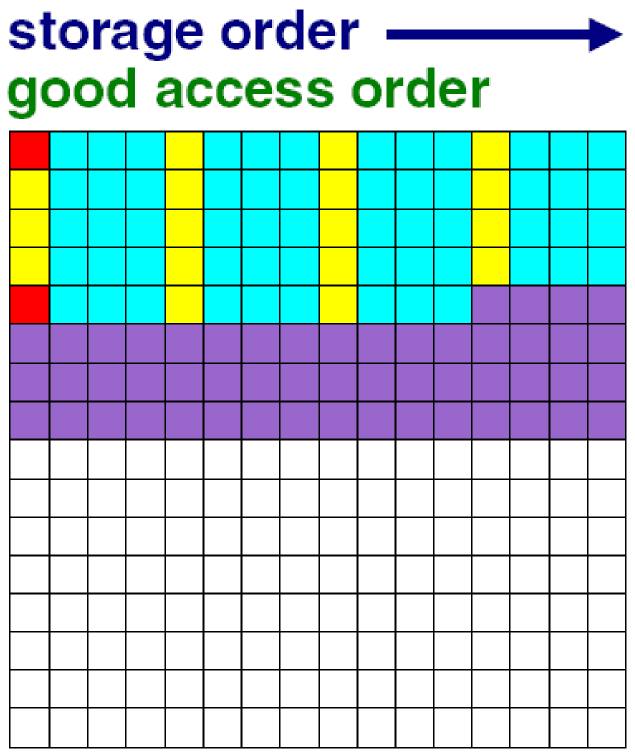
Memory access

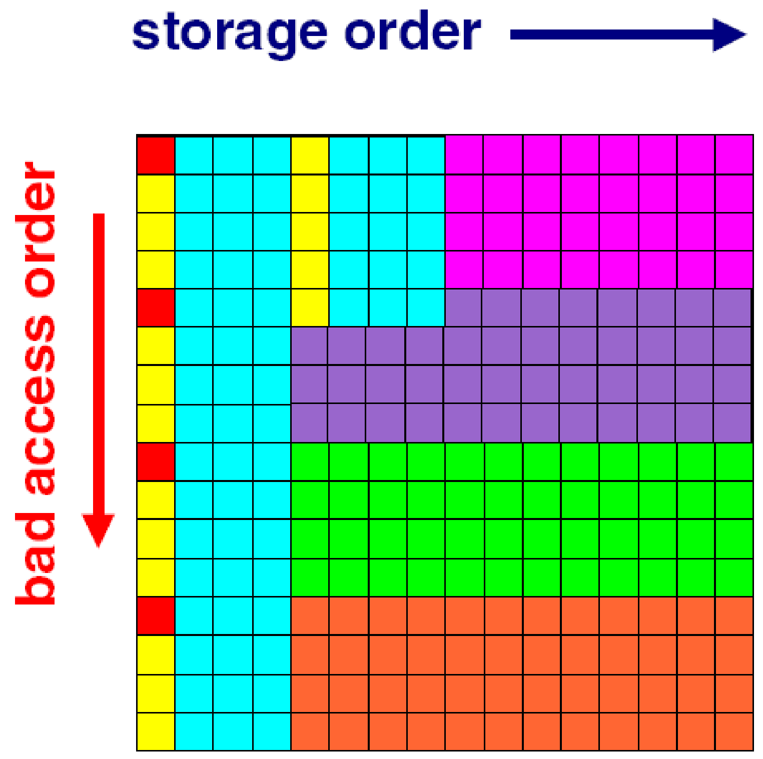
- Memory has a 1-dimensional linear structure
- Access to multi-dimensional arrays depends on how data is stored



Bad memory access has a huge impact on performance!!!

Bad memory access – C example





🟥 = TLB miss
🟨 = D-cache miss
🟦 = Cached elements
🟪 ▨ = Virtual memory page

- If the entire matrix fits in the cache, the access pattern *hardly* matters.
- For large (out-of-cache) matrices, the access pattern **does** matter – both data cache and TLB misses

The TLB cache

- the Translation Lookaside Buffer (TLB) translates virtual memory addresses (in your application) to physical addresses
- unit: page – typical size 4kB
- creation of lookup table is an expensive operation
- cost: 10 – 100 clock cycles/miss
- modern CPUs are having more advanced TLBs
- support for variable page sizes

About cache misses

Some simple rules:

- ❑ You cannot avoid cache misses – there are part of the nature of cache-based systems
- ❑ But you should try to minimize them to get good performance

Cache Line Utilization

Two key rules: Maximize ...

- Spatial locality ⇒ Use all data in one cache line
  - depends on storage layout
  - depends on access patterns
    - stride = 1 is good
    - random access is really bad
- Temporal locality ⇒ Re-use data in a cache line
  - depends on algorithm used