C++ now

Unlocking Modern CPU Power

Next-Gen C++ Optimization Techniques

Fedor G Pikus

2024

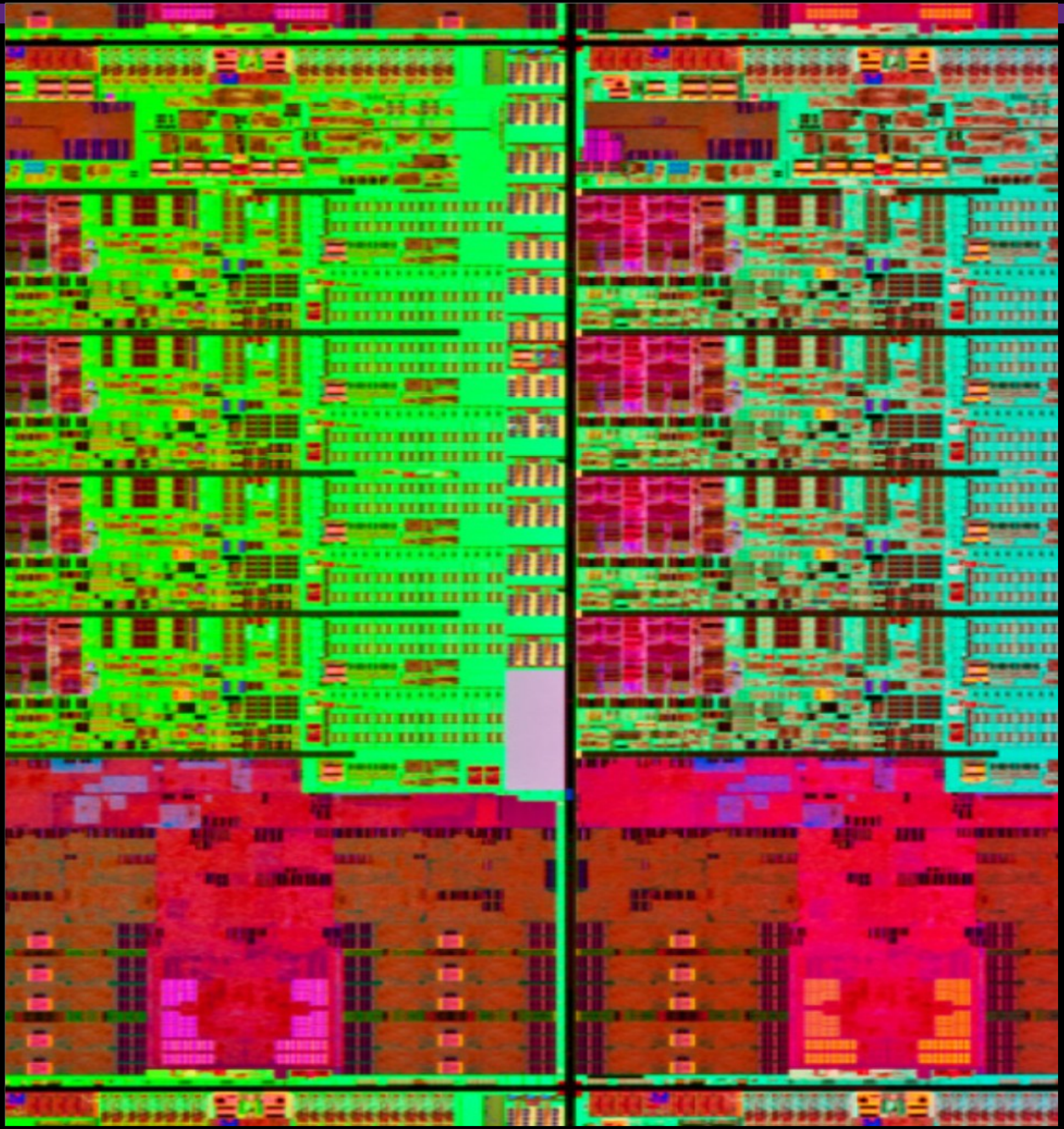
Unlocking Modern CPU Power

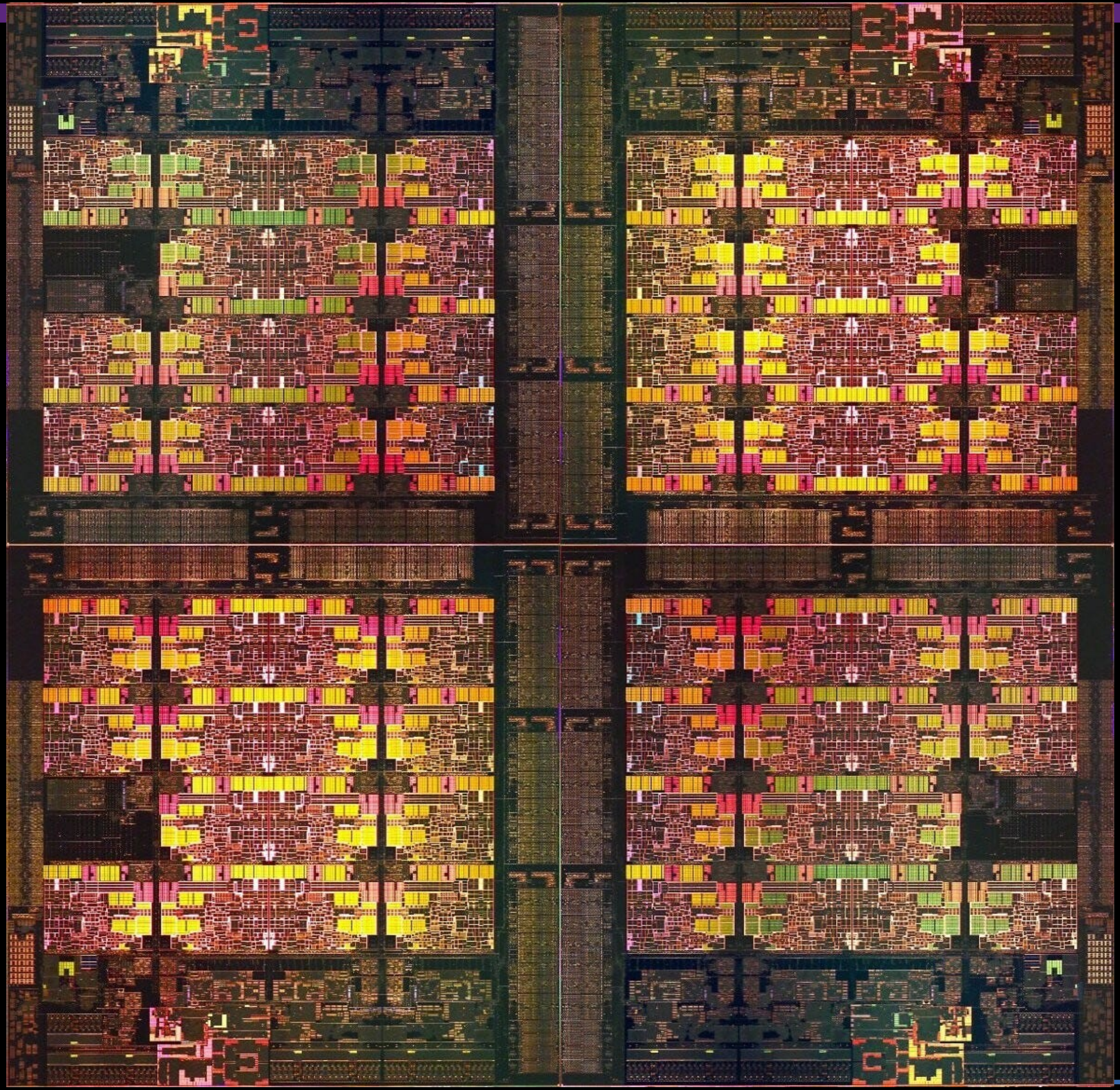
How to get the most out of modern CPUs

Fedor G Pikus
Siemens Fellow

SIEMENS

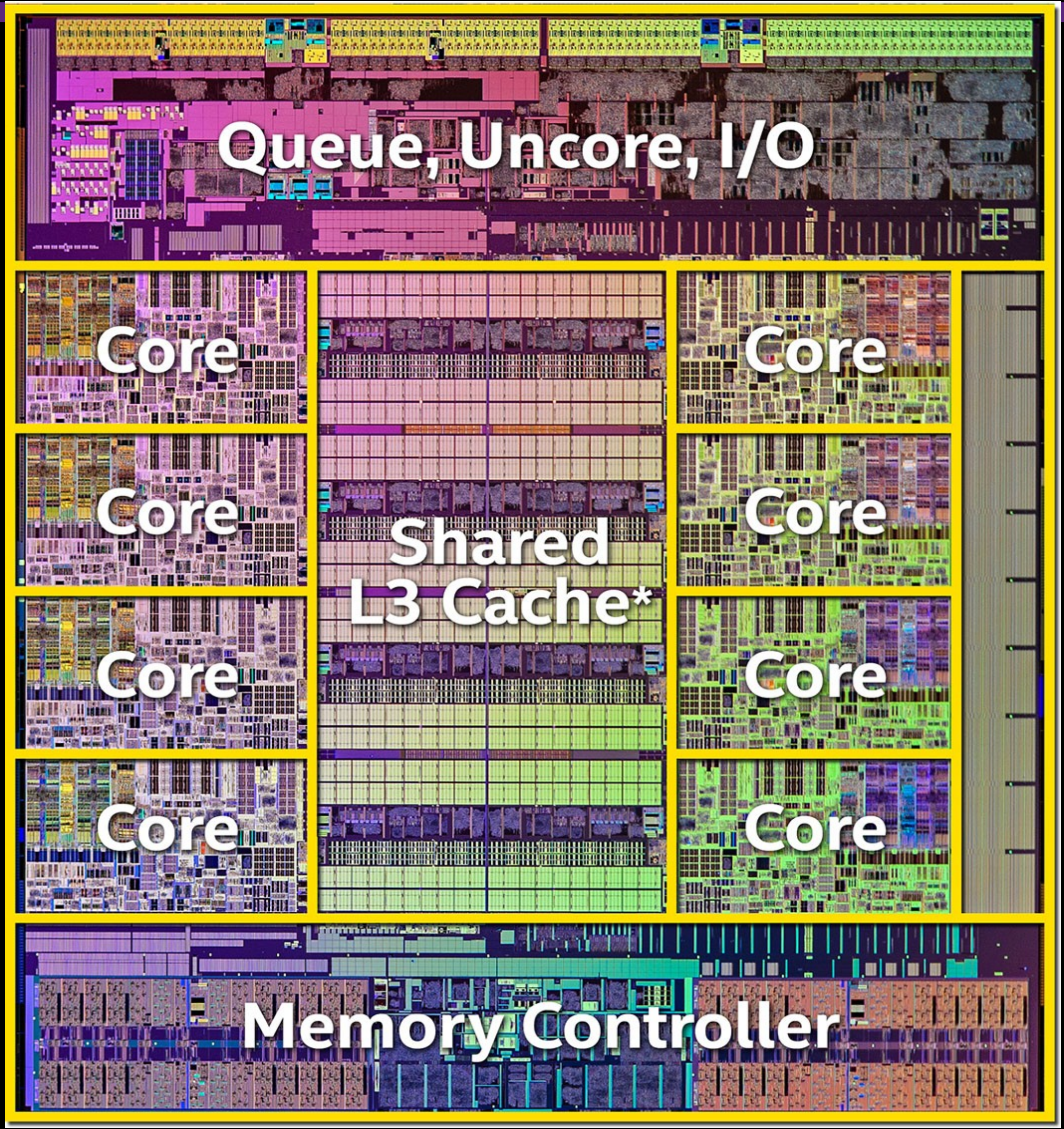
Ten years of CPU evolution in pictures

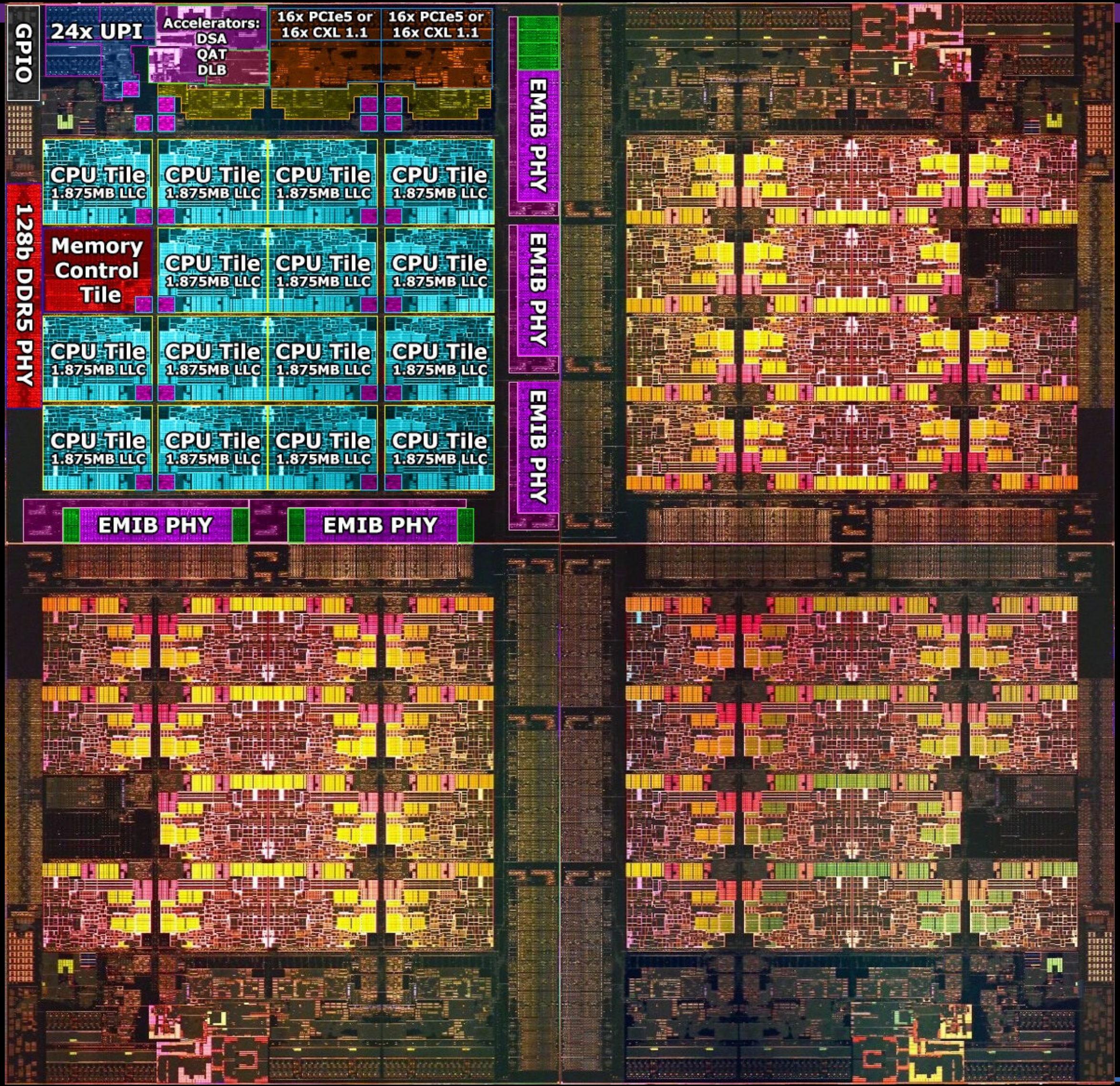




SIEMENS

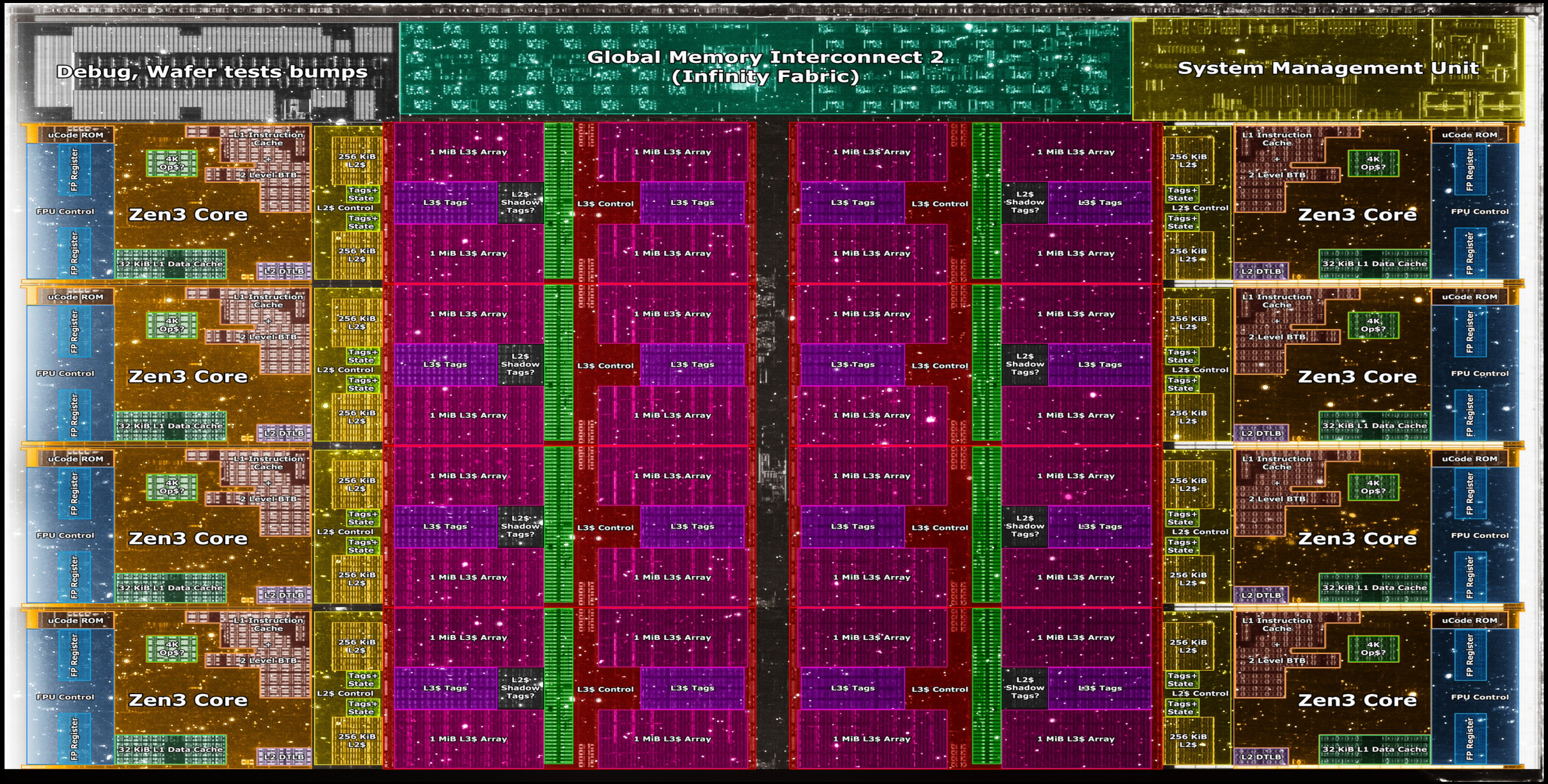
Ten years of CPU evolution in pictures





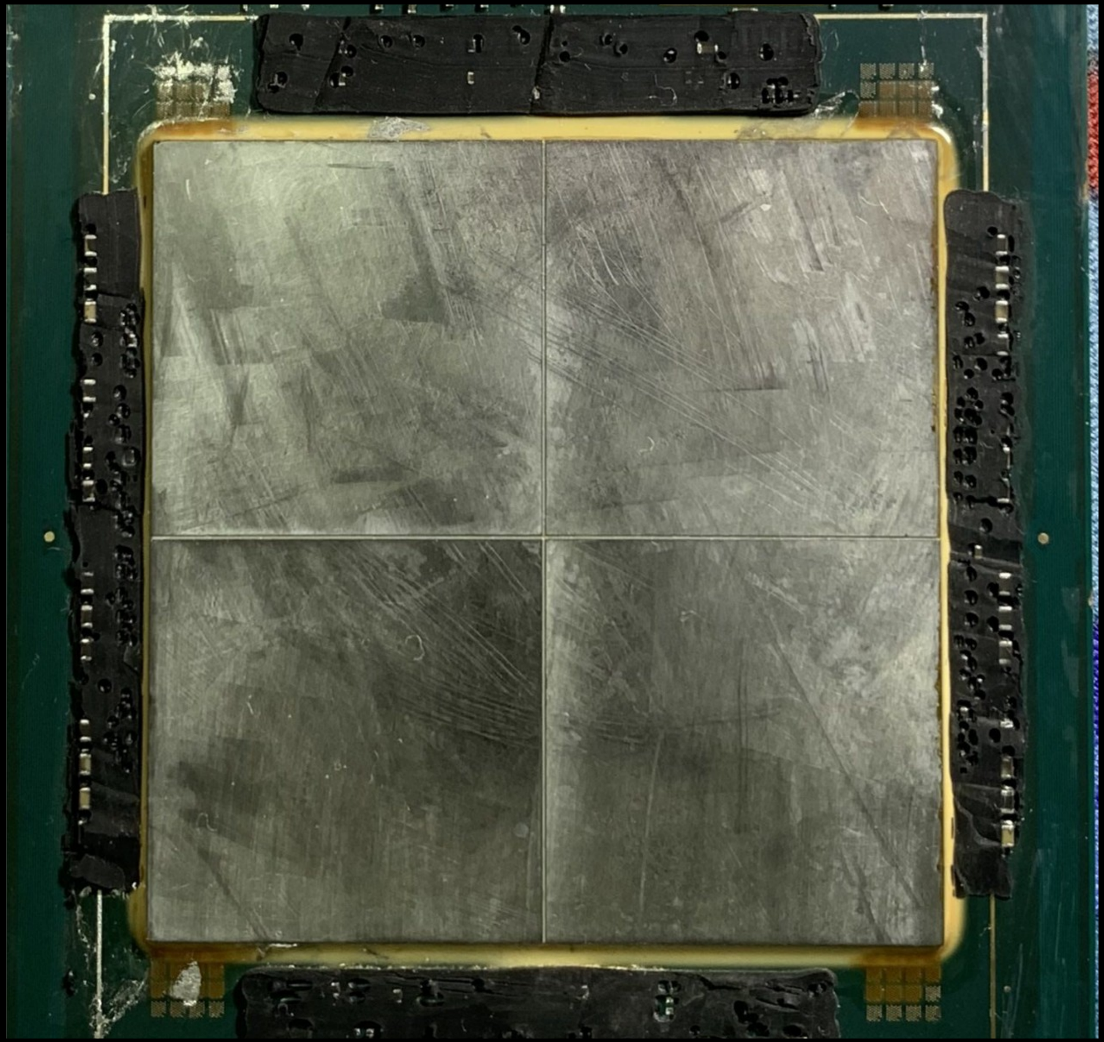
SIEMENS

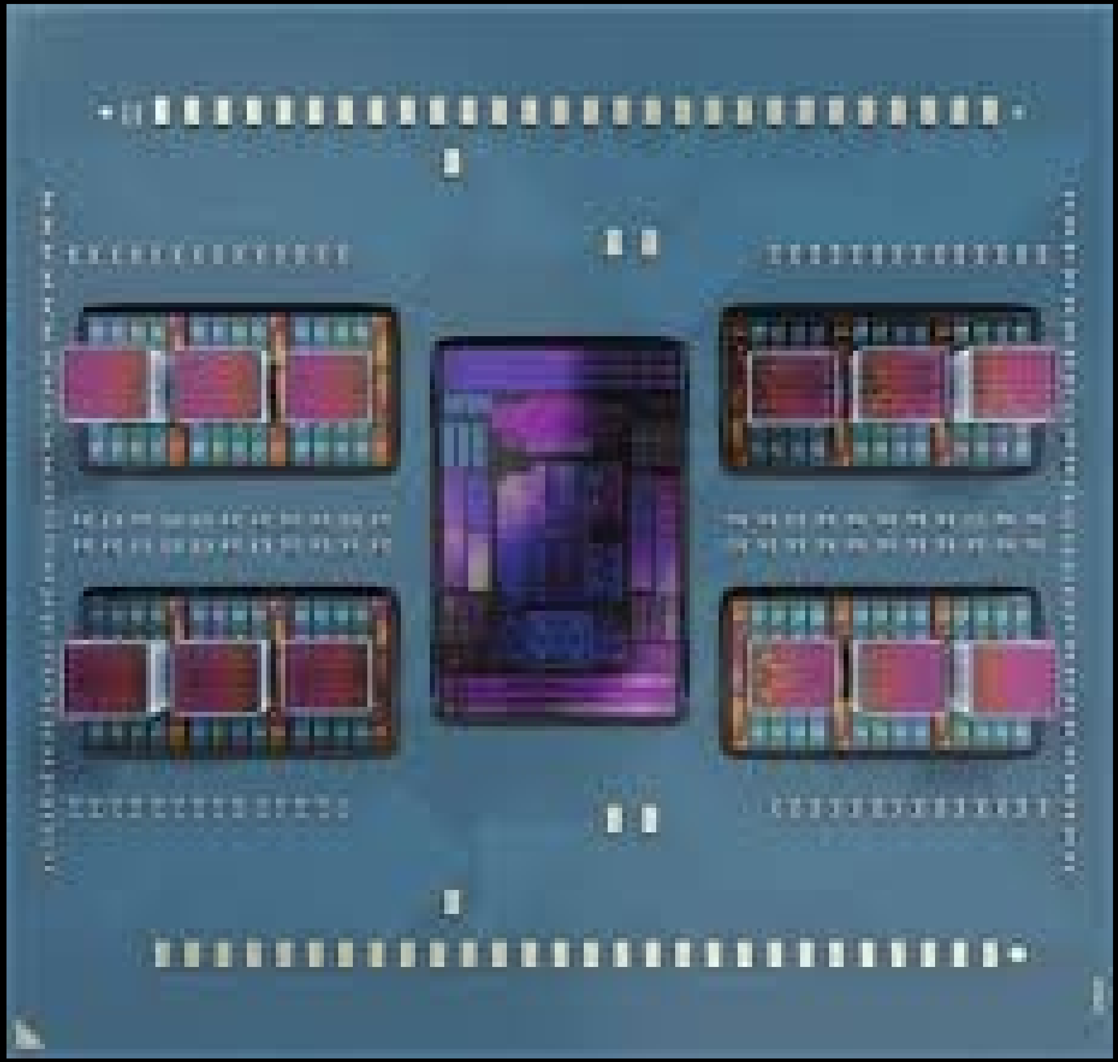
Ten years of CPU evolution in pictures



SIEMENS

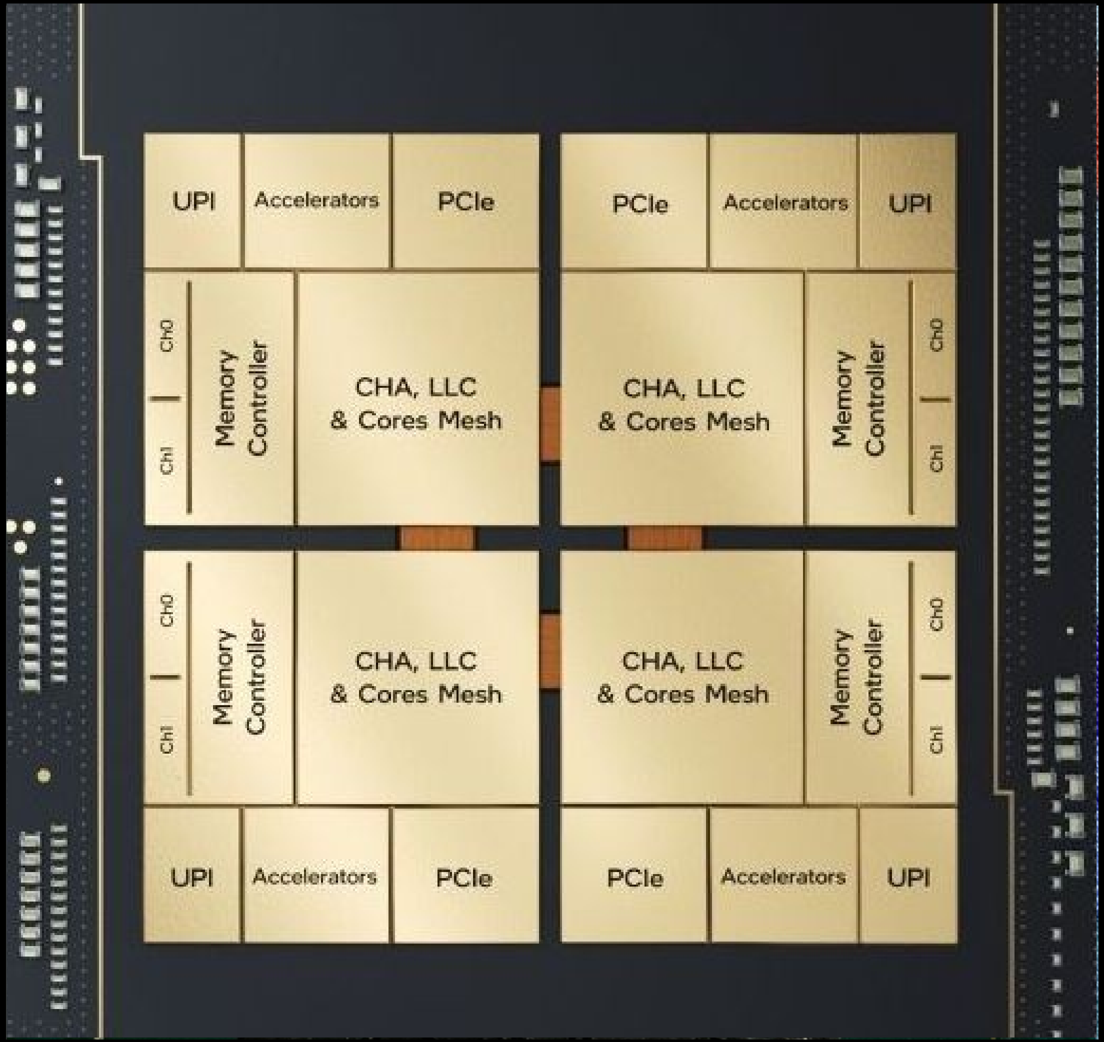
Ten years of CPU evolution in pictures

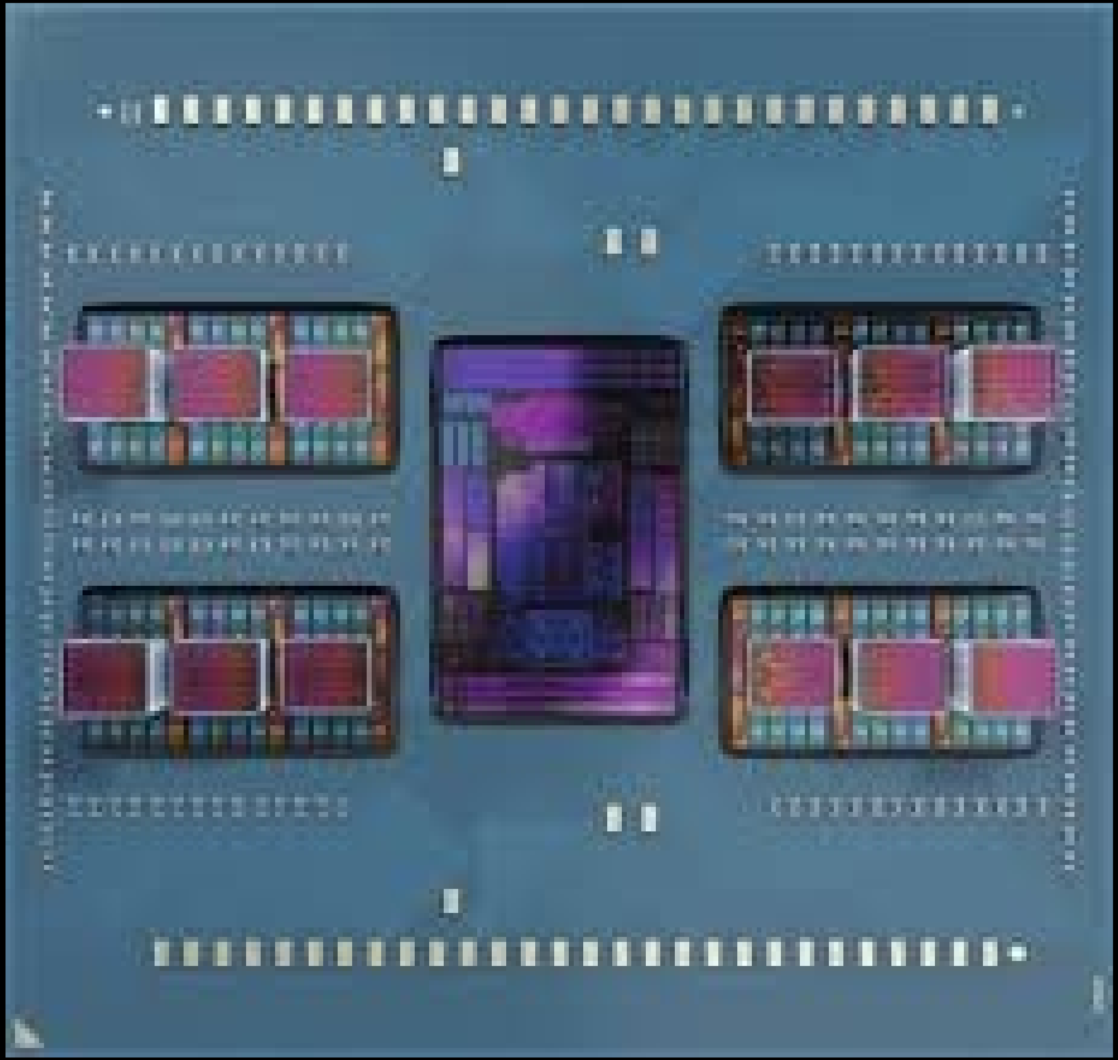




SIEMENS

Ten years of CPU evolution in pictures





SIEMENS

Ten years of CPU evolution in numbers

- Similar die size – around 400mm^2
  - 2014: one CPU – one die
  - 2024: several dies in a package
- Much higher transistor count:
  - 2014: 2 to 3 Billion
  - 2024: ~40 Billion (~100 Billion with GPUs)
- Similar clock frequencies: 2 to 4 GHz
- Significantly more cores
- Much larger caches:
  - 2014: L1 – 32K/core, L2 – 256K/core, L3 ~ 40M shared
  - 2024: L1 – 3M/core, L2 – 96M/core, L3 ~ 1G shared

How to not fail at using the new hardware

- CPUs are much more complex
- Much greater computing power!
  - Unless you’re not using it...
- Much larger caches – why?
  - Memory is relatively slow, accessing data is the bottleneck
  - Huge caches help (unless you render them useless…)
- Much more complex internal state (e.g., longer pipelines)
  - Works well if the computations are predictable
  - Expensive to discard and recreate

Outline

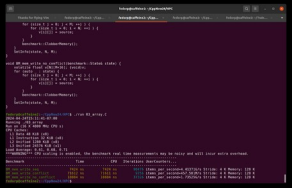
- Of course the CPU does that!
- Makes sense that the CPU would do that…
- The CPU does WHAT?!

SIEMENS

Part I

- Of course the CPU does that!
- Makes sense that the CPU would do that…
- The CPU does WHAT?!

CPUs can do a lot of work at once



- Multiple operations on the data in the general registers
  - Integer, floating-point, logical, and address math
- Memory accesses
- SIMD vector instructions
- Other special-purpose instructions

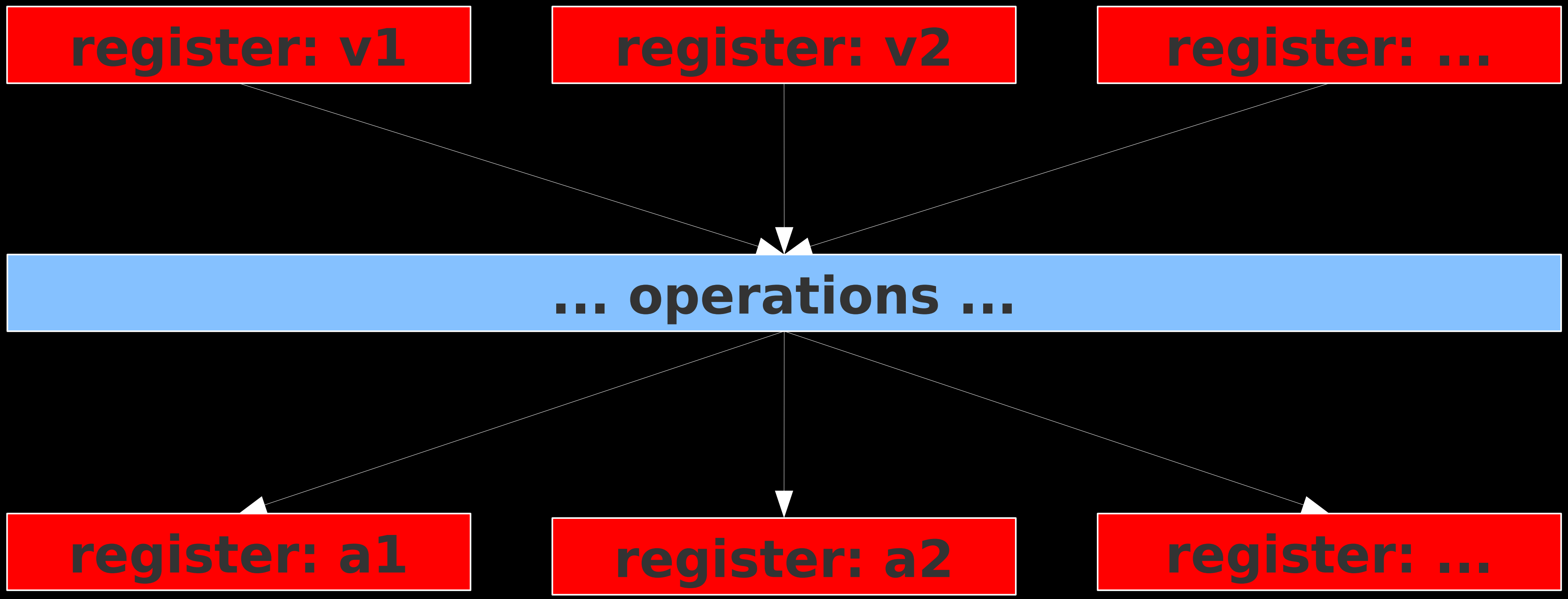
```
for (size_t i = 0; i != N; ++i) {
  … a1[i] + a2[i] …
  … a1[i] - a2[i] …
  … a1[i] += 7 …
  … a2[i] << 3 …
}
```

Everything at once!

Up to a point...

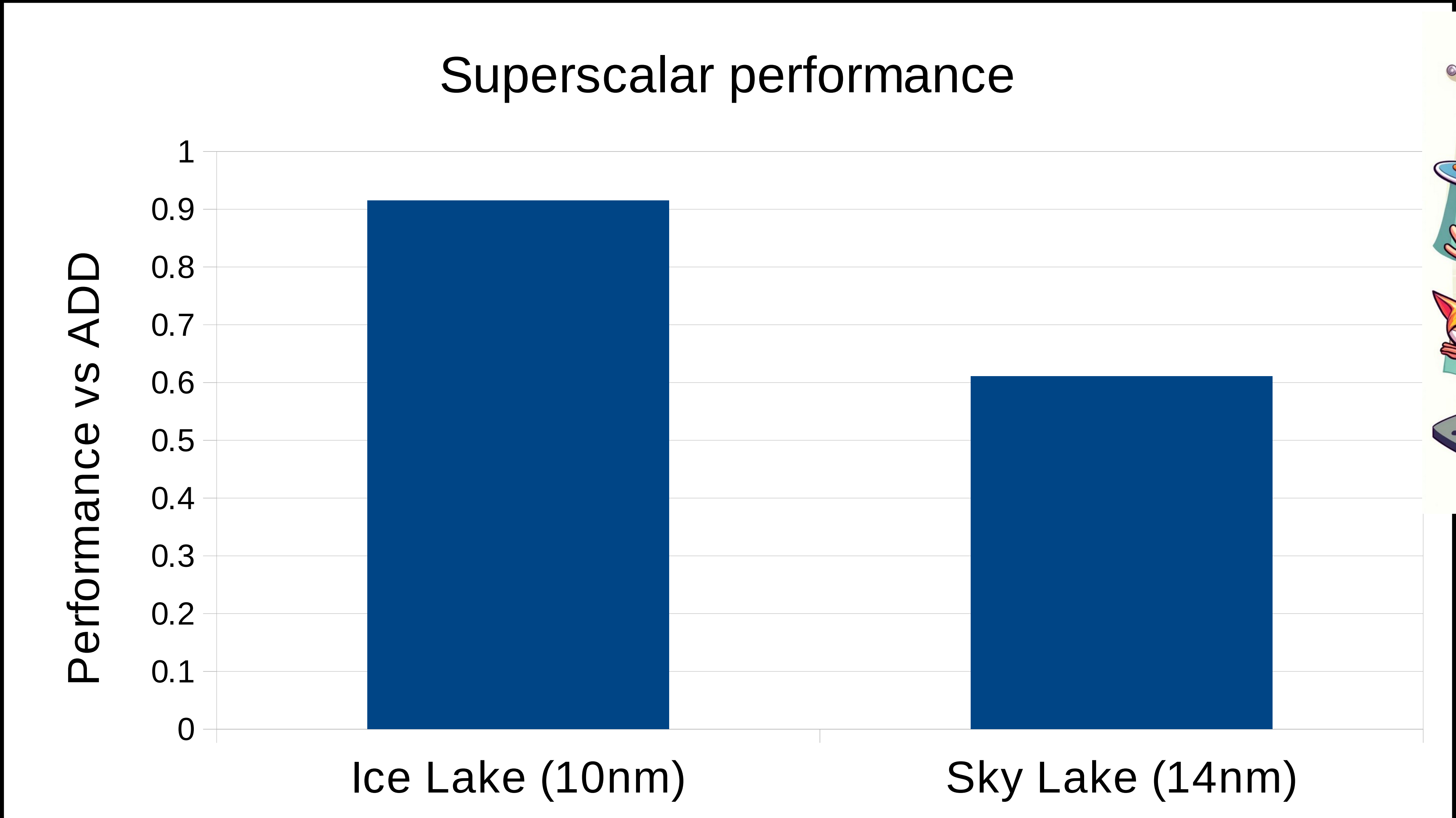
SIEMENS

Modern CPUs can do a lot more work at once



SIEMENS

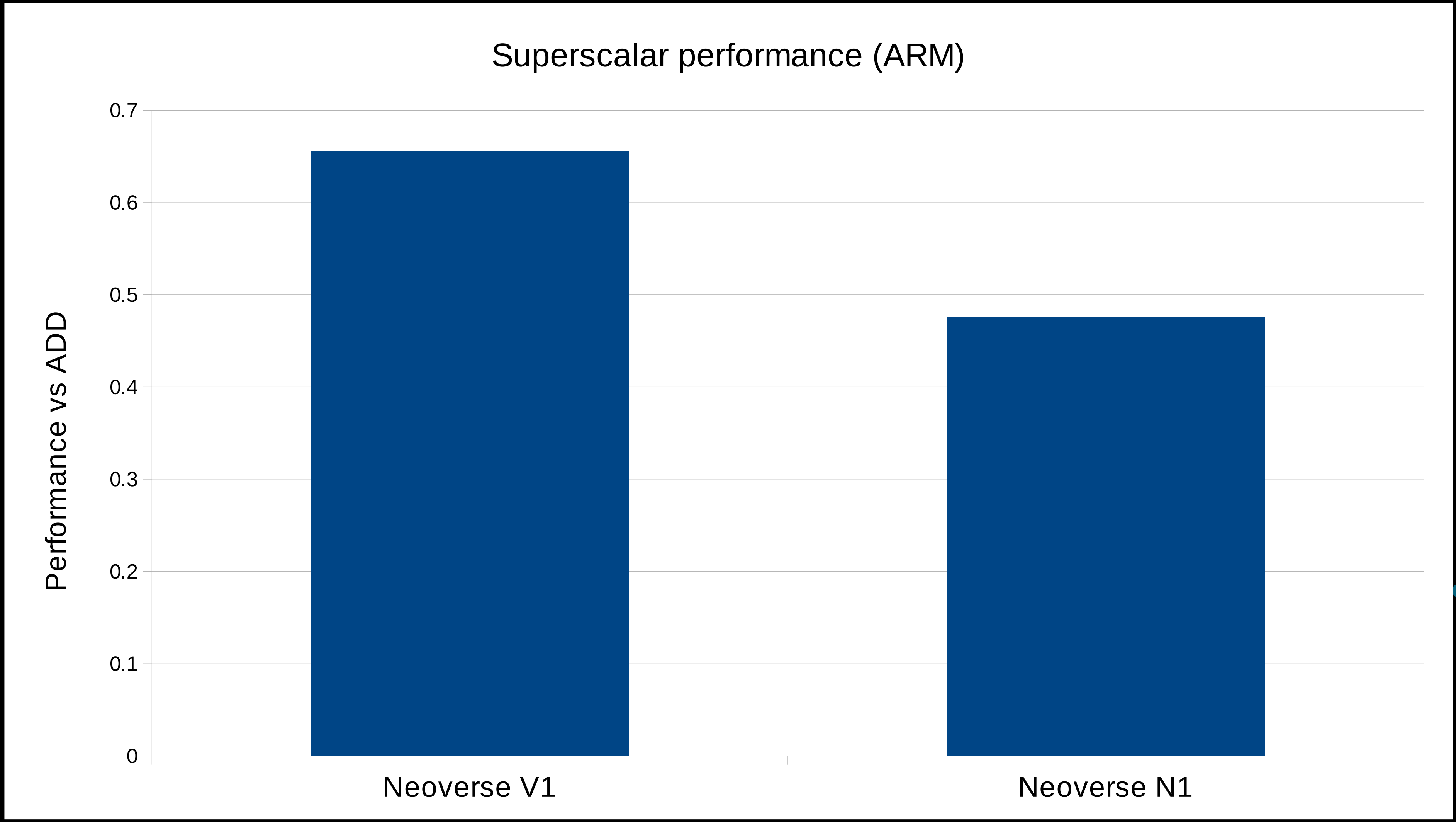
Modern CPUs can do a lot more work at once



- Test: is it slower to do many operations at once?

SIEMENS

Modern CPUs can do a lot more work at once



- Not just X86 CPUs are getting better at doing multiple operations

SIEMENS

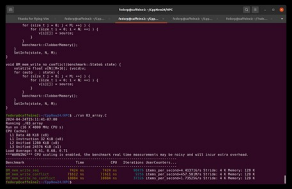
Taking advantage of the superscalar CPU

- Practical benefits of superscalar architectures are limited mostly by memory accesses and data dependencies
- Reducing memory and data dependencies often requires significant code reorganization, maybe a redesign
  - Memory is a “great equalizer:” CPUs get faster, memory – not so much
- It is more efficient to access memory in large words (_m256i)
- Large caches and prefetch help to hide slow memory
- The solution to data dependencies is pipelining
- Modern CPUs rely on caches and pipelining to a much greater degree to achieve their peformance
  - The penalty for not using caches and for disrupting pipelines is far greater!

How do completely disable prefetch?

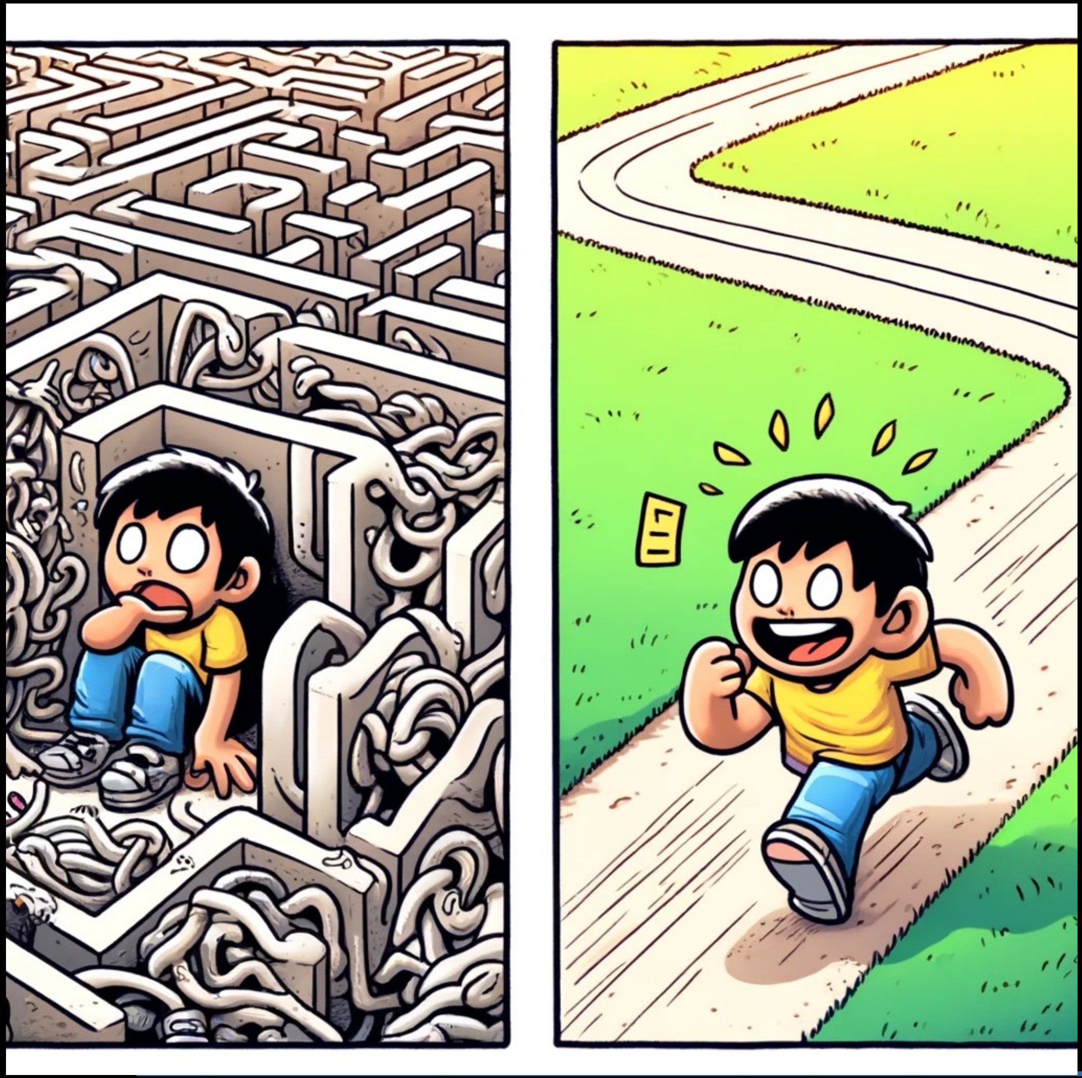
- Memory access is characterized by bandwidth and latency
  - Bandwidth is much higher than “latency per word”
  - Random access speed is limited by latency
  - Sequential access speed is limited by bandwidth
- Prefetch attempts to predict future memory accesses and transfers memory content into cache in advance
  - Random access defeats prediction

How do completely disable prefetch?



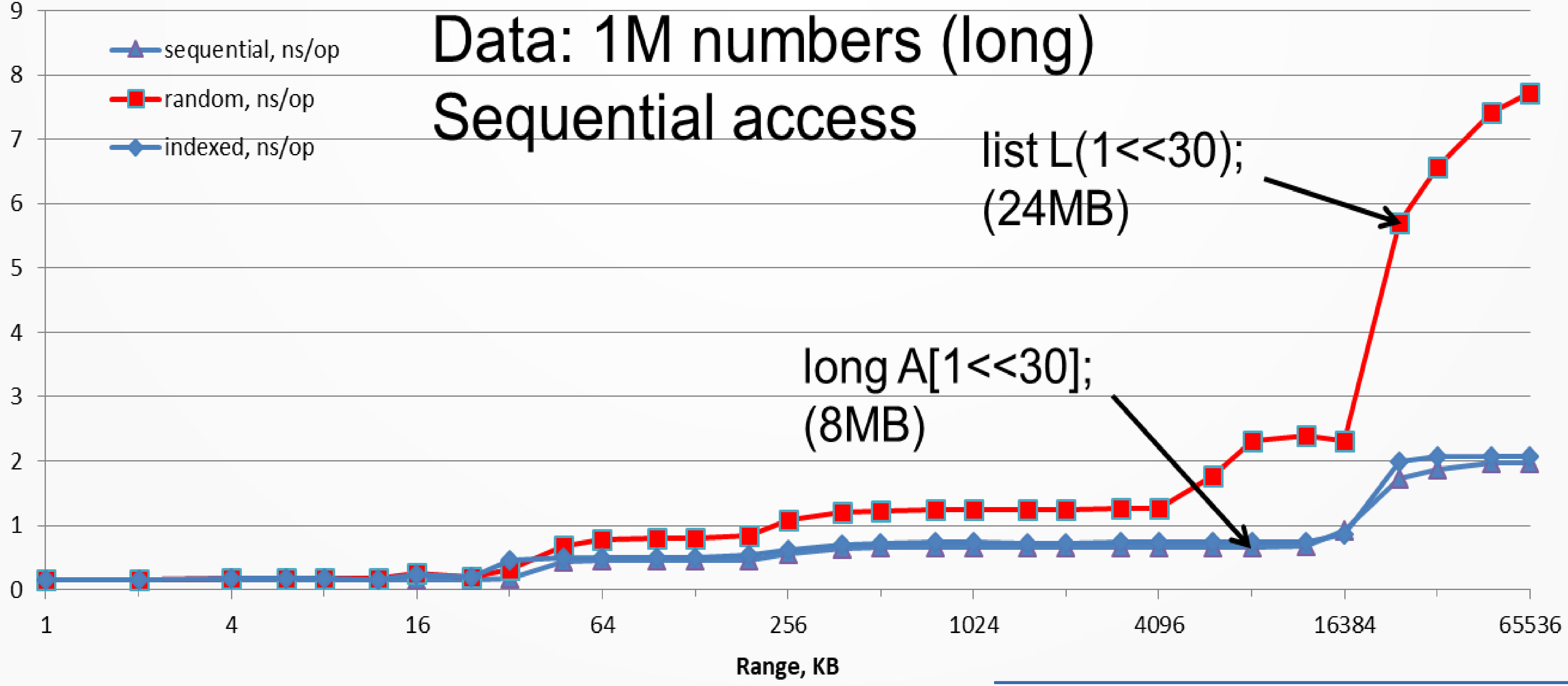
```
Container<T> data(N);
for (T& x : data) ++x;
```

- Container can provide sequential access (vector) or random access (list)



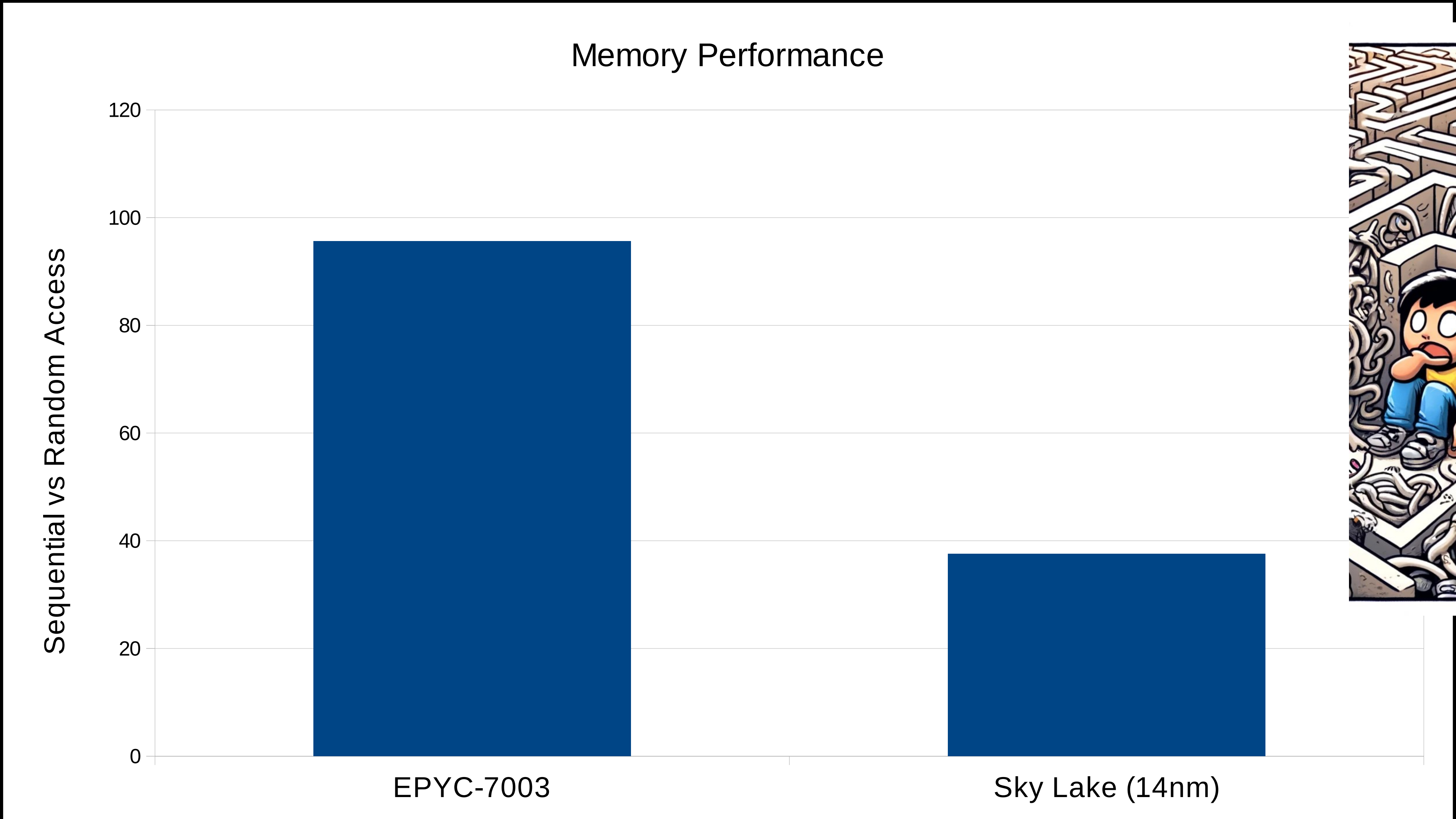
SIEMENS

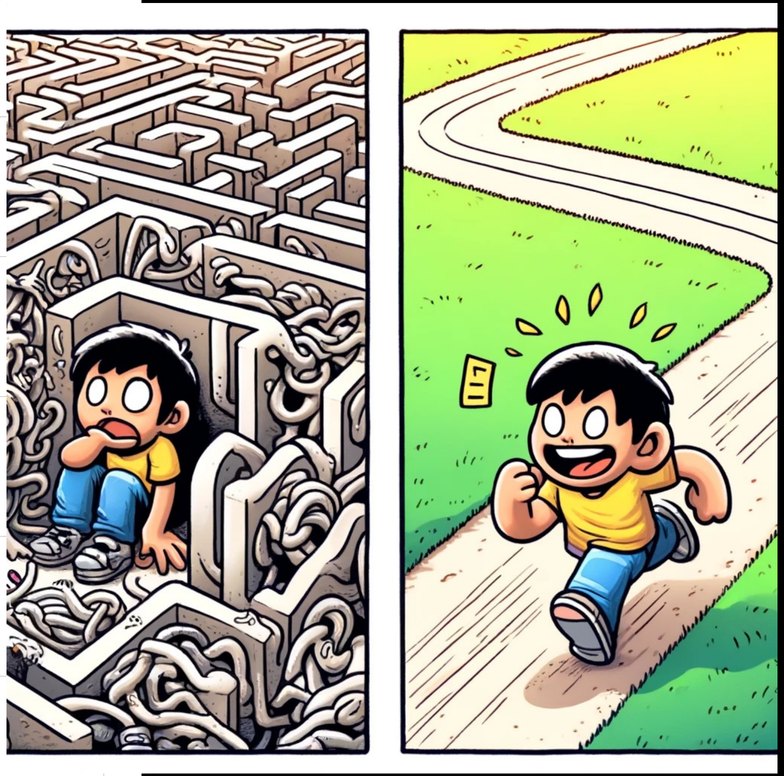
Sequential vs Random access – Expectations



SIEMENS

How large is the random access penalty?





- Random access is bad
  - And gets worse

SIEMENS

How to [almost] completely disable caches?

```
float v[N][M];
```

```
for (size_t i=0; j<N; ++i) {
  for (size_t j=0; j<M; ++j) {
    v[i][j] = 1;
  }
}
```

```
for (size_t j=0; j<M; ++j) {
  for (size_t i=0; i<N; ++i) {
    v[i][j] = 1;
  }
}
```

- What is the performance difference between these loops?

How to [almost] completely disable caches?

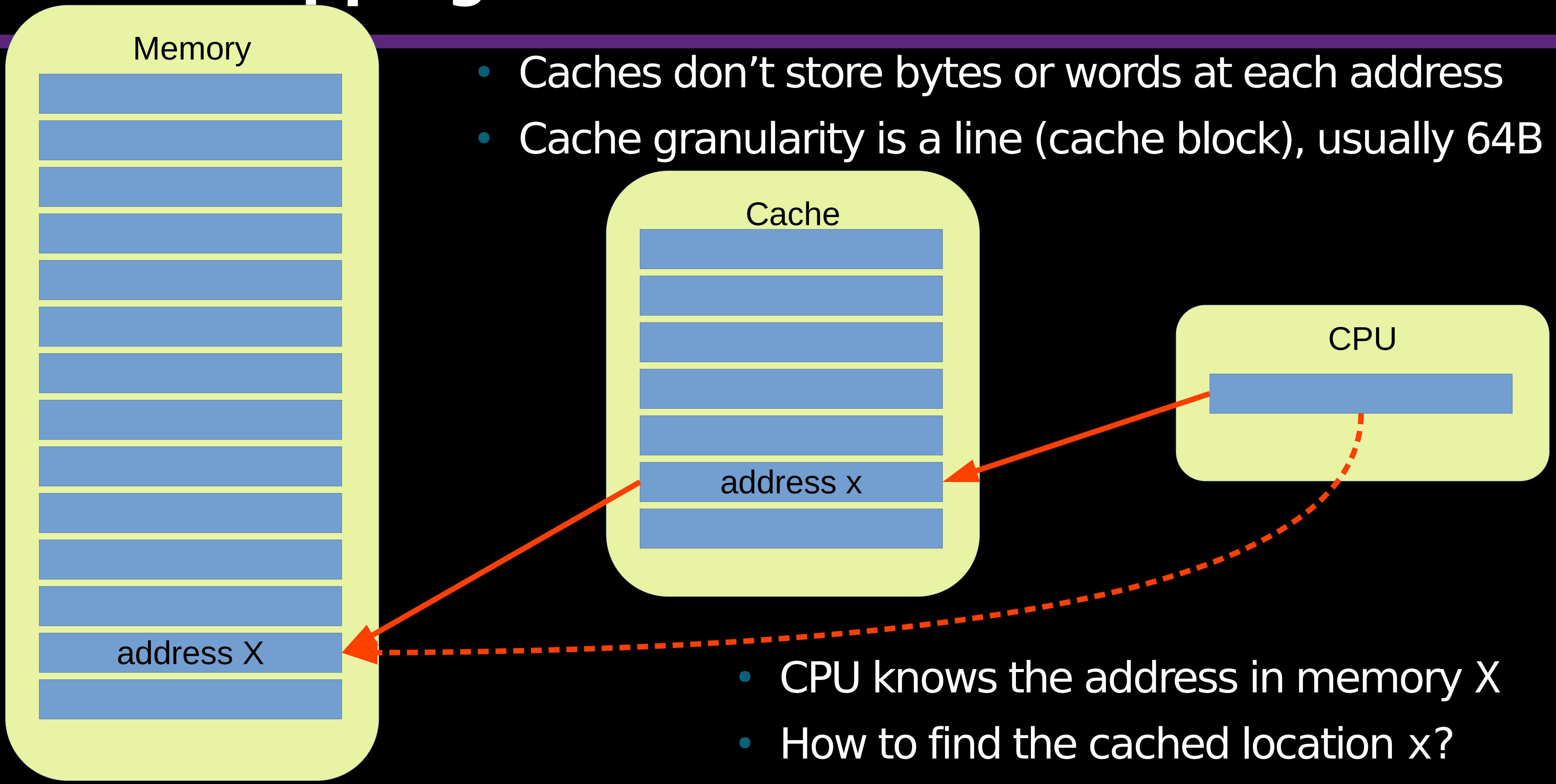
```
float v[N][M];
```

```
for (size_t i=0; j<N; ++i) {
  for (size_t j=0; j<M; ++j) {
    v[i][j] = 1;
  }
}
```

```
for (size_t j=0; j<M; ++j) {
  for (size_t i=0; i<N; ++i) {
    v[i][j] = 1;
  }
}
```

- What is the performance difference between row-major and column-major matrix accesses?

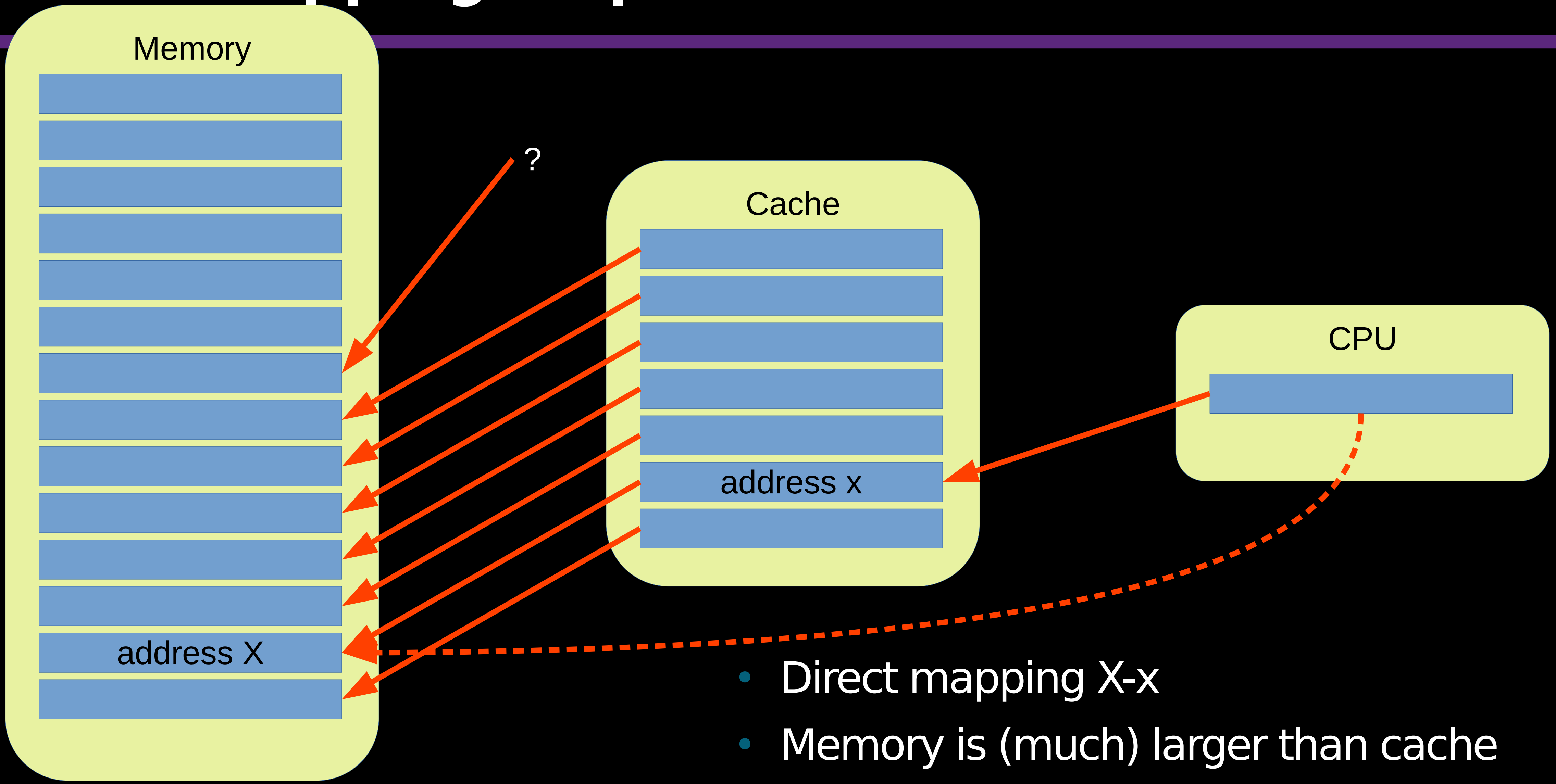
Cache mapping



- Caches don’t store bytes or words at each address
- Cache granularity is a line (cache block), usually 64B

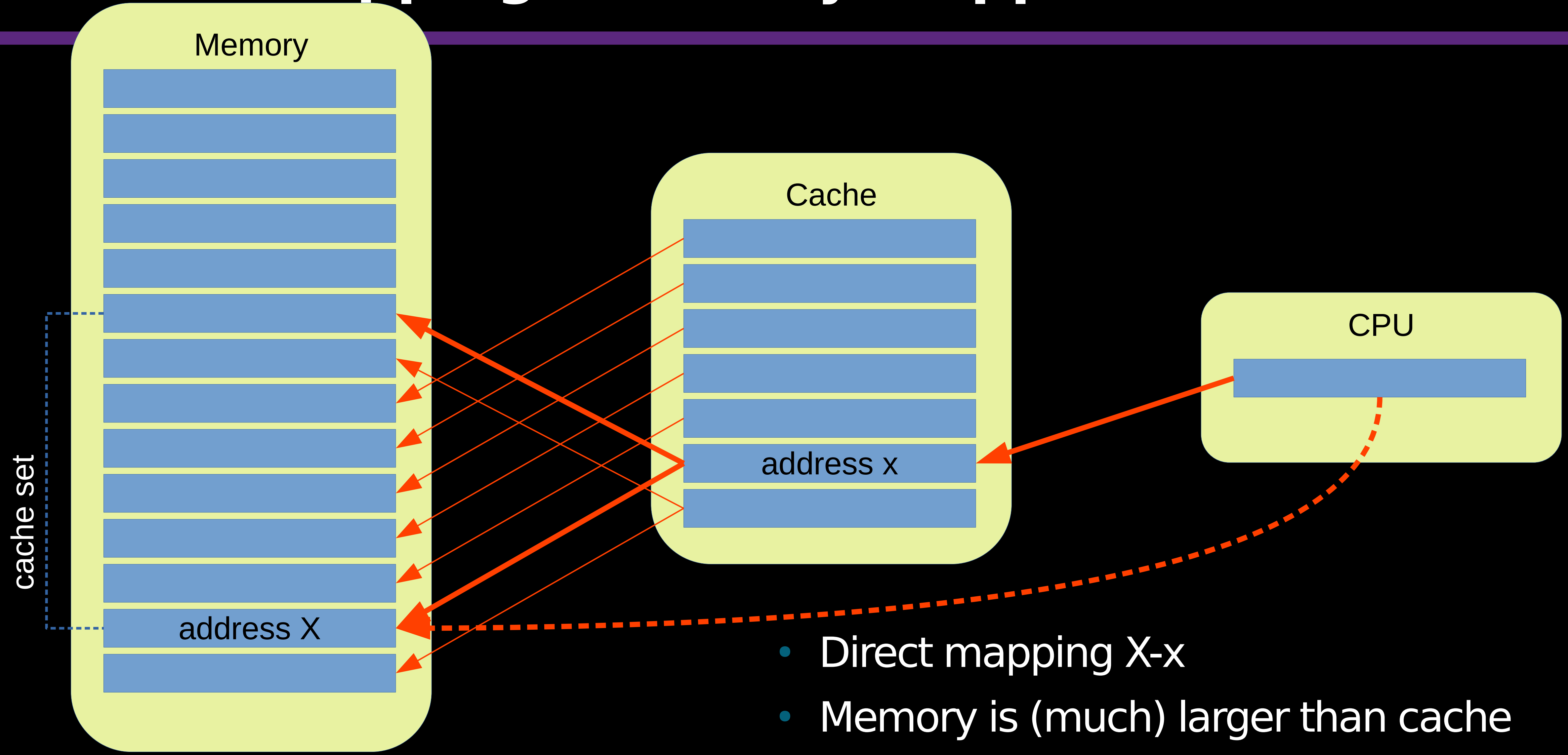
- CPU knows the address in memory X
- How to find the cached location x?

Cache mapping – Option 1



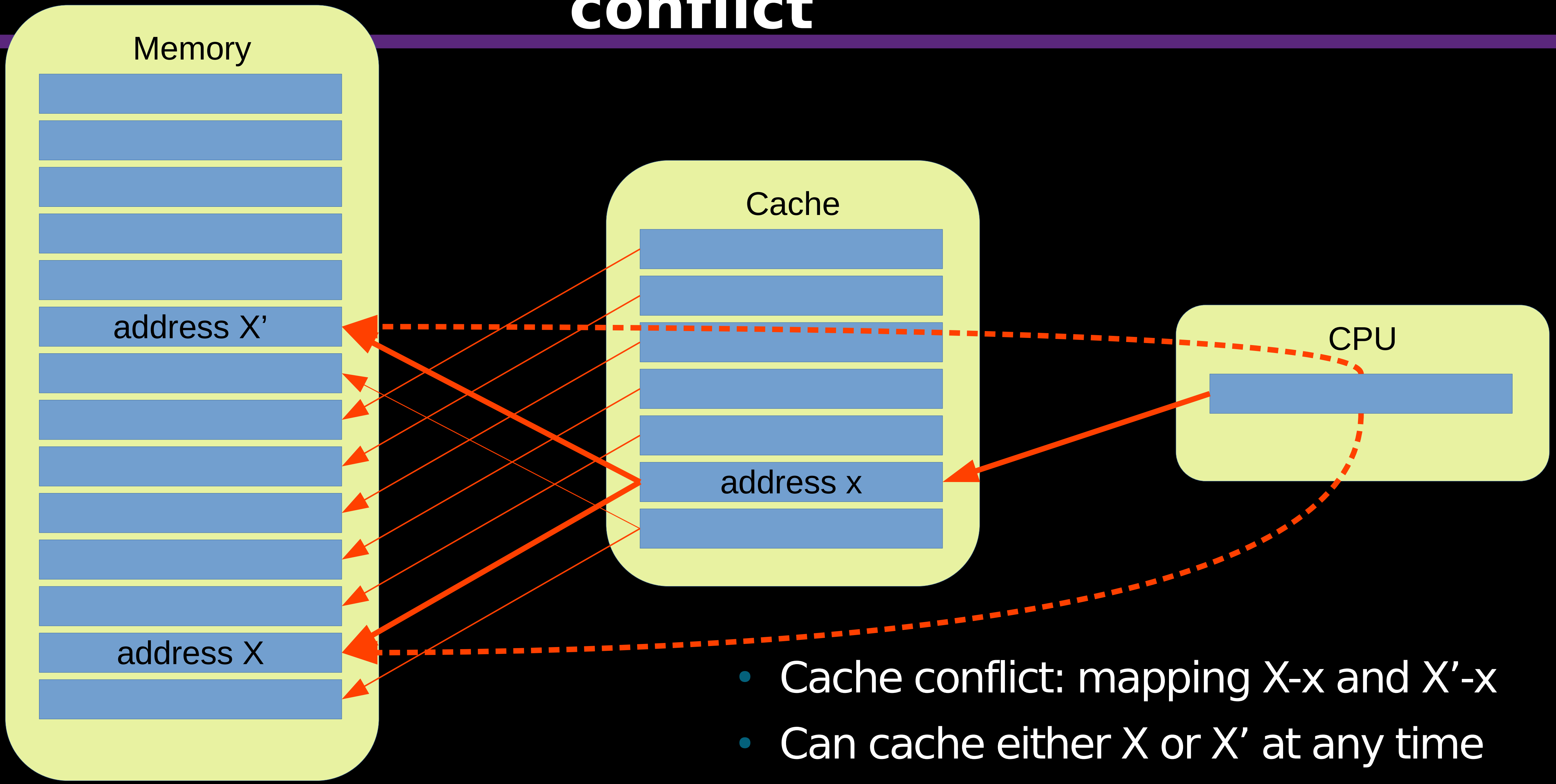
- Direct mapping X-x
- Memory is (much) larger than cache

Cache mapping – Directly mapped cache



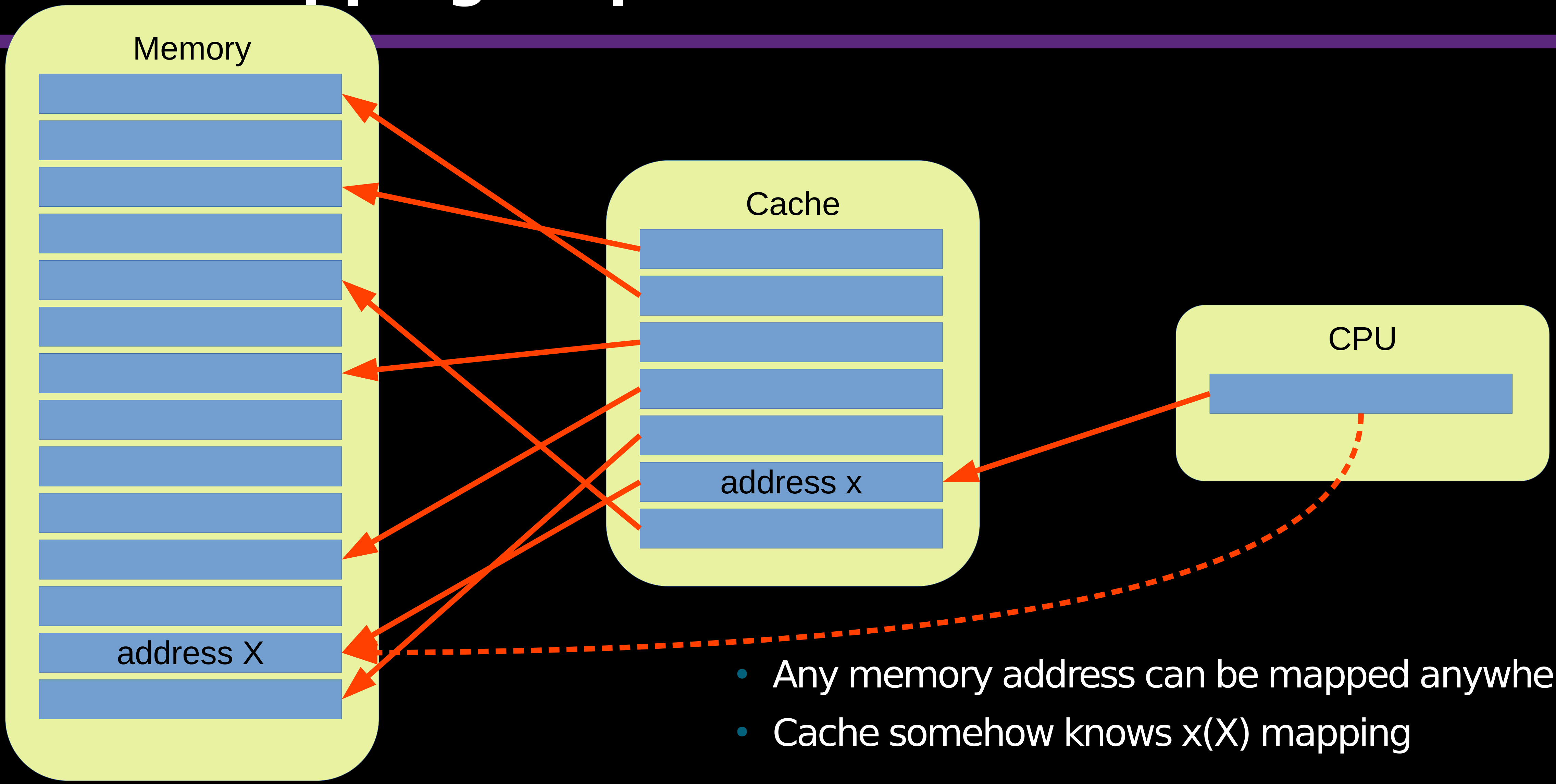
- Direct mapping X-x
- Memory is (much) larger than cache

Cache mapping – Directly mapped cache conflict



- Cache conflict: mapping X-x and X'-x
- Can cache either X or X' at any time

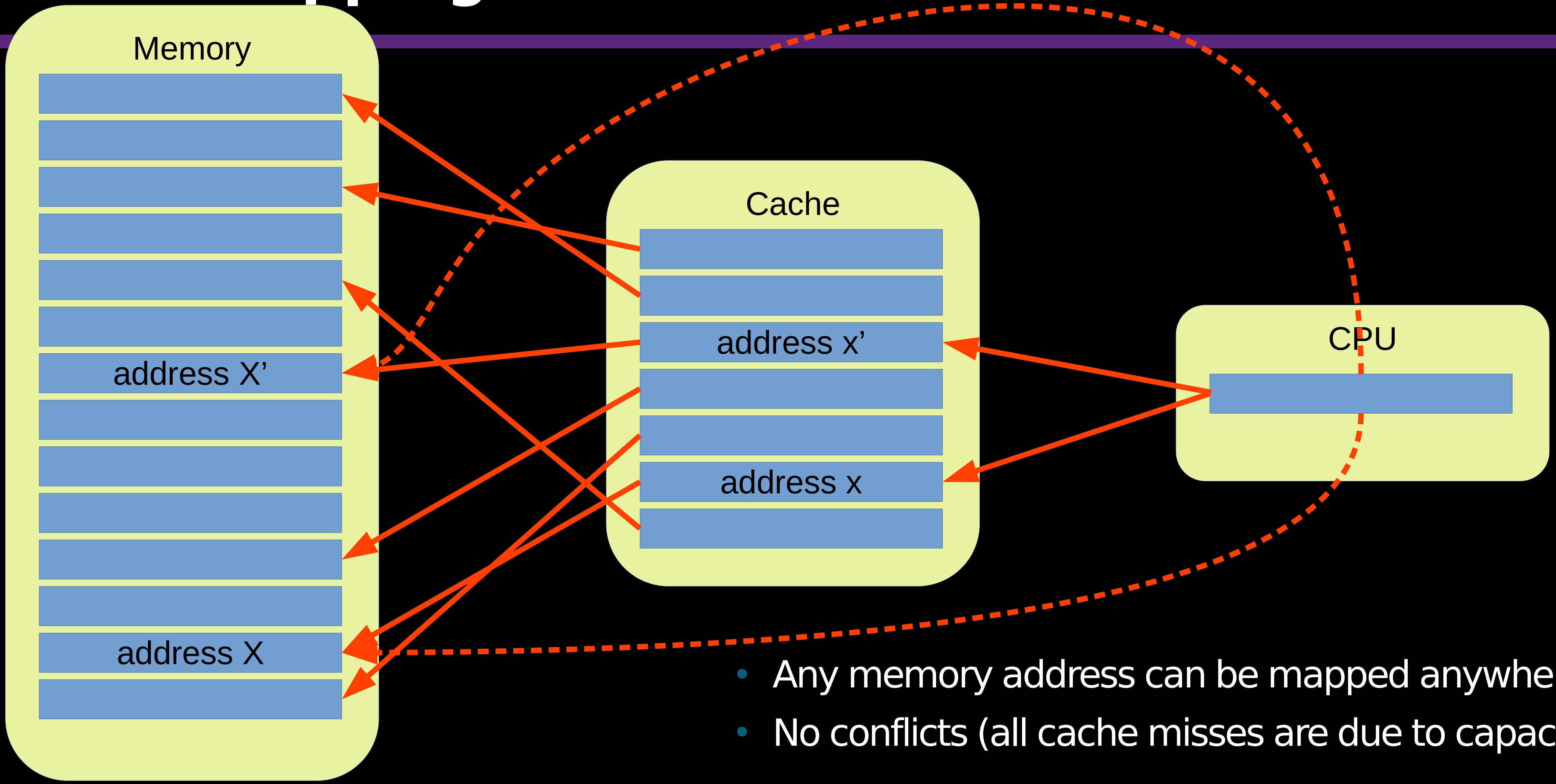
Cache mapping – Option 2



- Any memory address can be mapped anywhere
- Cache somehow knows x(X) mapping

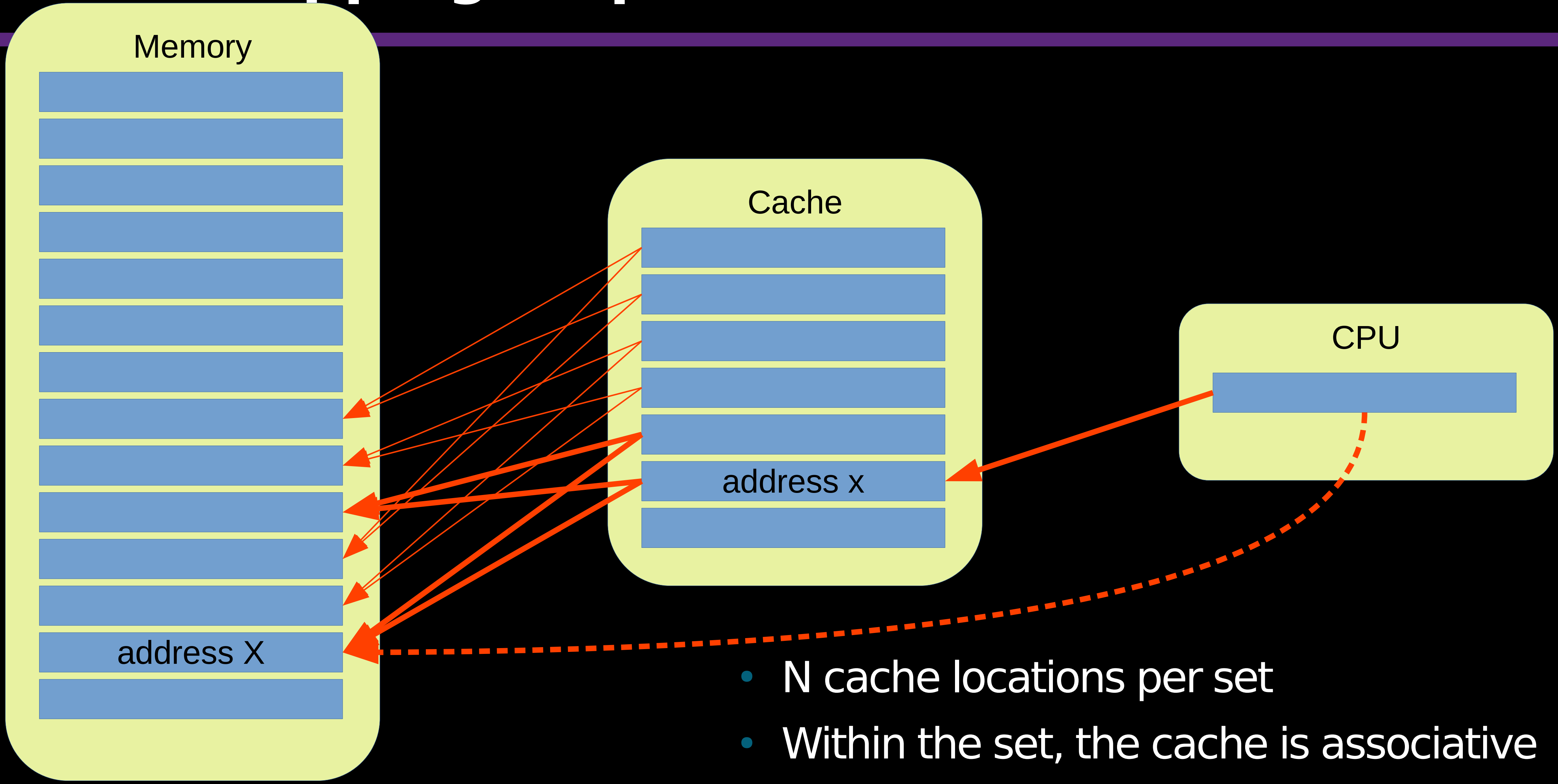
SIEMENS

Cache mapping – Associative cache



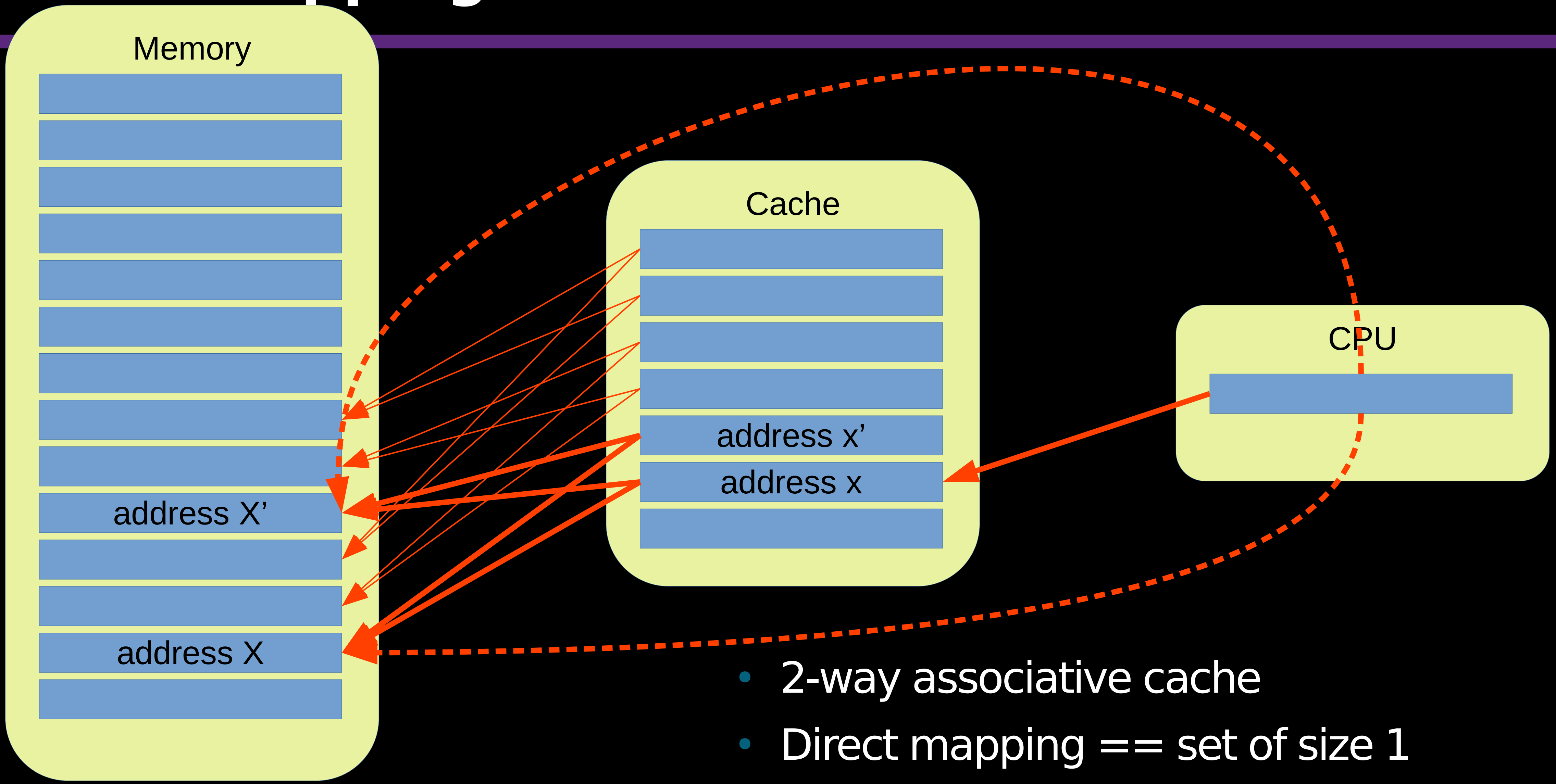
- Any memory address can be mapped anywhere
- No conflicts (all cache misses are due to capacity)

Cache mapping – Option 3



- N cache locations per set
- Within the set, the cache is associative

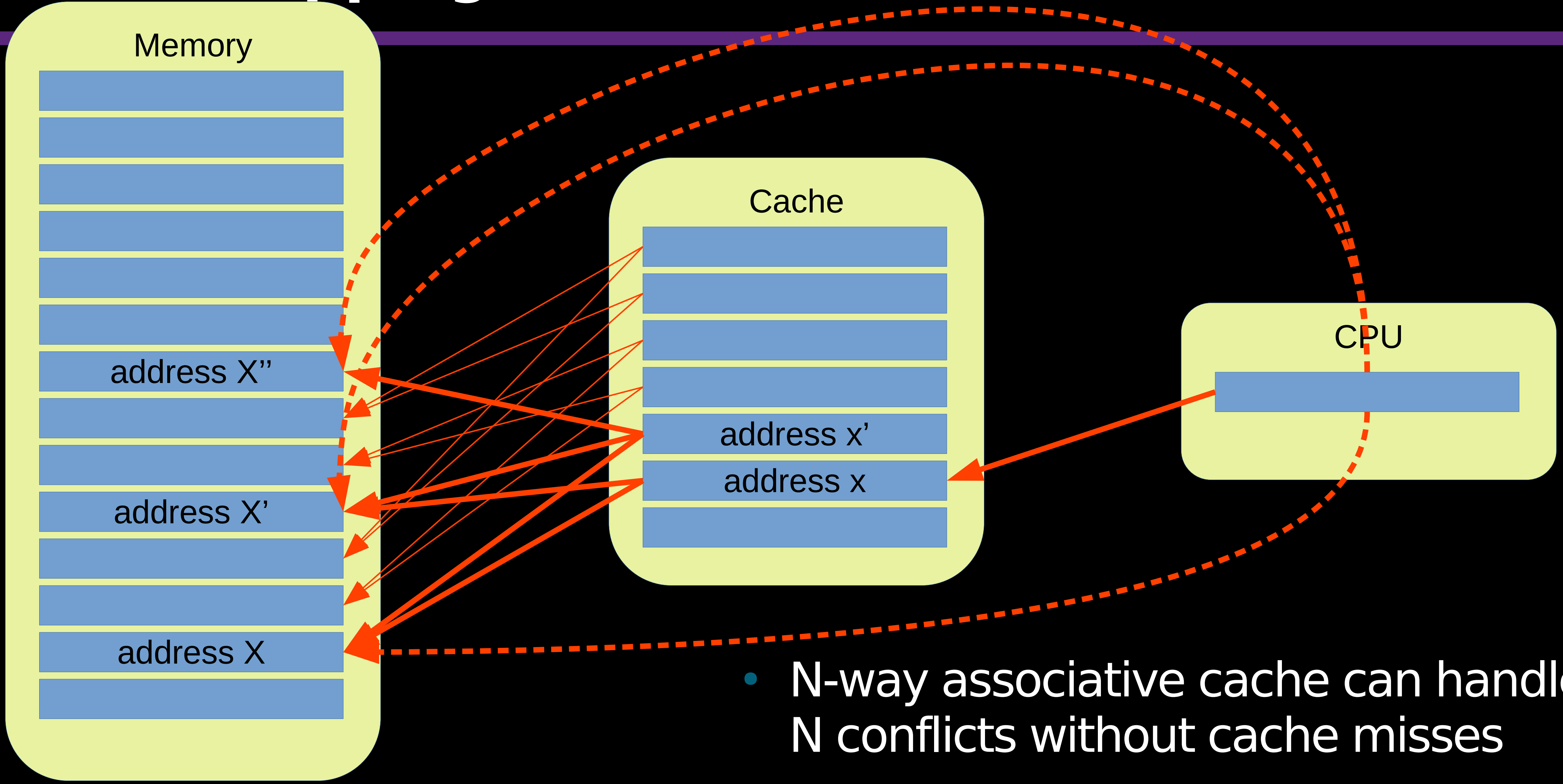
Cache mapping – Set-associative cache



- 2-way associative cache
- Direct mapping == set of size 1

SIEMENS

Cache mapping – Set-associative cache



- N-way associative cache can handle N conflicts without cache misses

Caches and memory addresses

- 32KB 8-way associative cache
  - 64B cache lines (cache blocks)
  - 32KB/64B = 512 cache lines
  - 512/8 = 64 sets (2^6 – 6 bits for set index)
  - Byte address is 64-byte offset in the line (2^6 – 6 bits for byte offset)
  - 64 bits in the address – 64-6-6=52 bit “tag” (index of one of memory locations in this set)

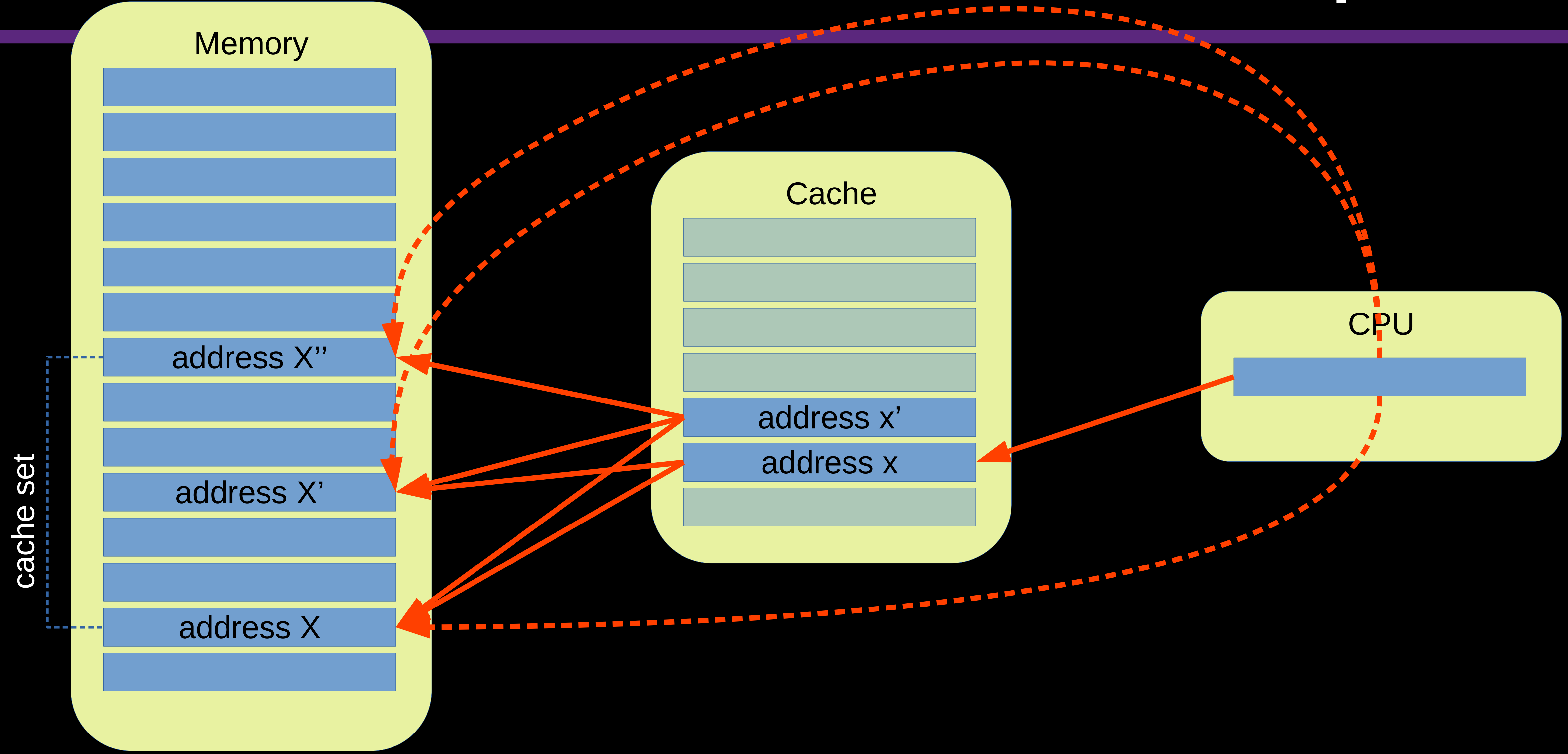
- All addresses with the same set bits map to the same set of cache blocks
  - In a 8-way cache, 8 lines with different tag bits and same set bits can be cached

Caches and memory addresses

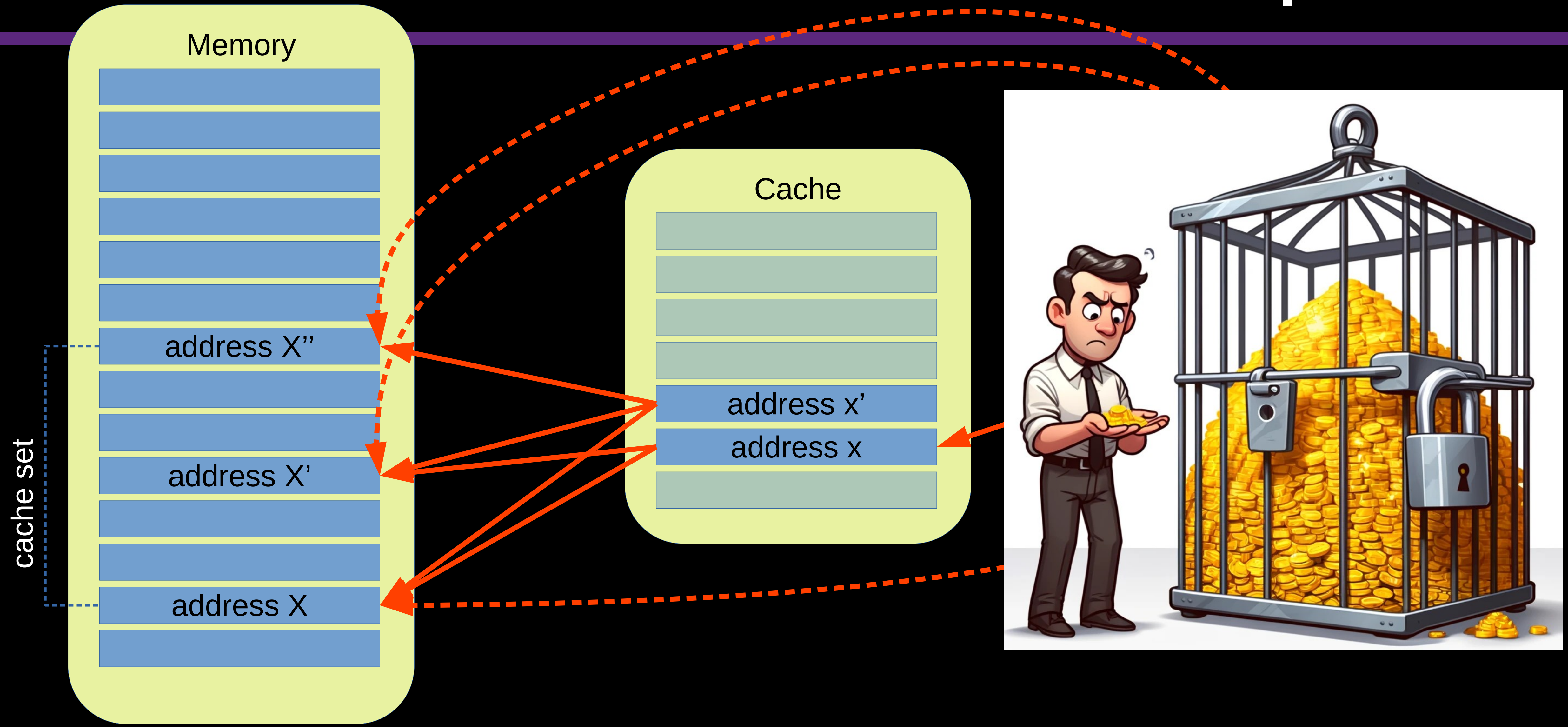
- 48KB 12-way associative cache
  - 64B cache lines (cache blocks)
  - 48KB/64B = 768 cache lines
  - 768/12 = 64 sets (2^6 – 6 bits for set index)
  - Byte address is 64-byte offset in the line (2^6 – 6 bits for byte offset)
  - 64 bits in the address – 64-6-6=52 bit “tag” (index of one of memory locations in this set)

- All addresses with the same set bits map to the same set of cache blocks
  - In a 8-way cache, 8 lines with different tag bits and same set bits can be cached

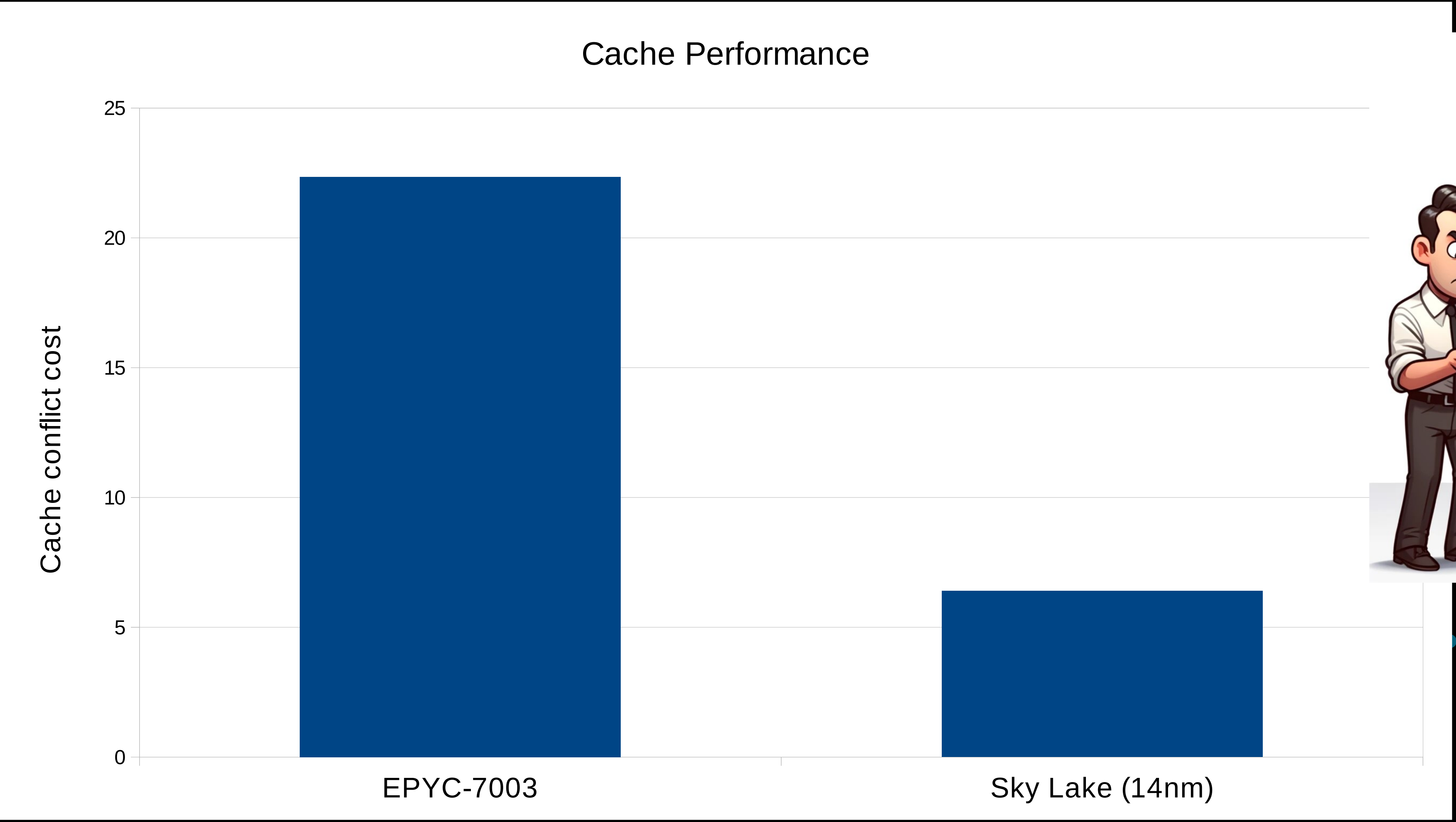
Set-associative cache and stride access pattern



Set-associative cache and stride access pattern

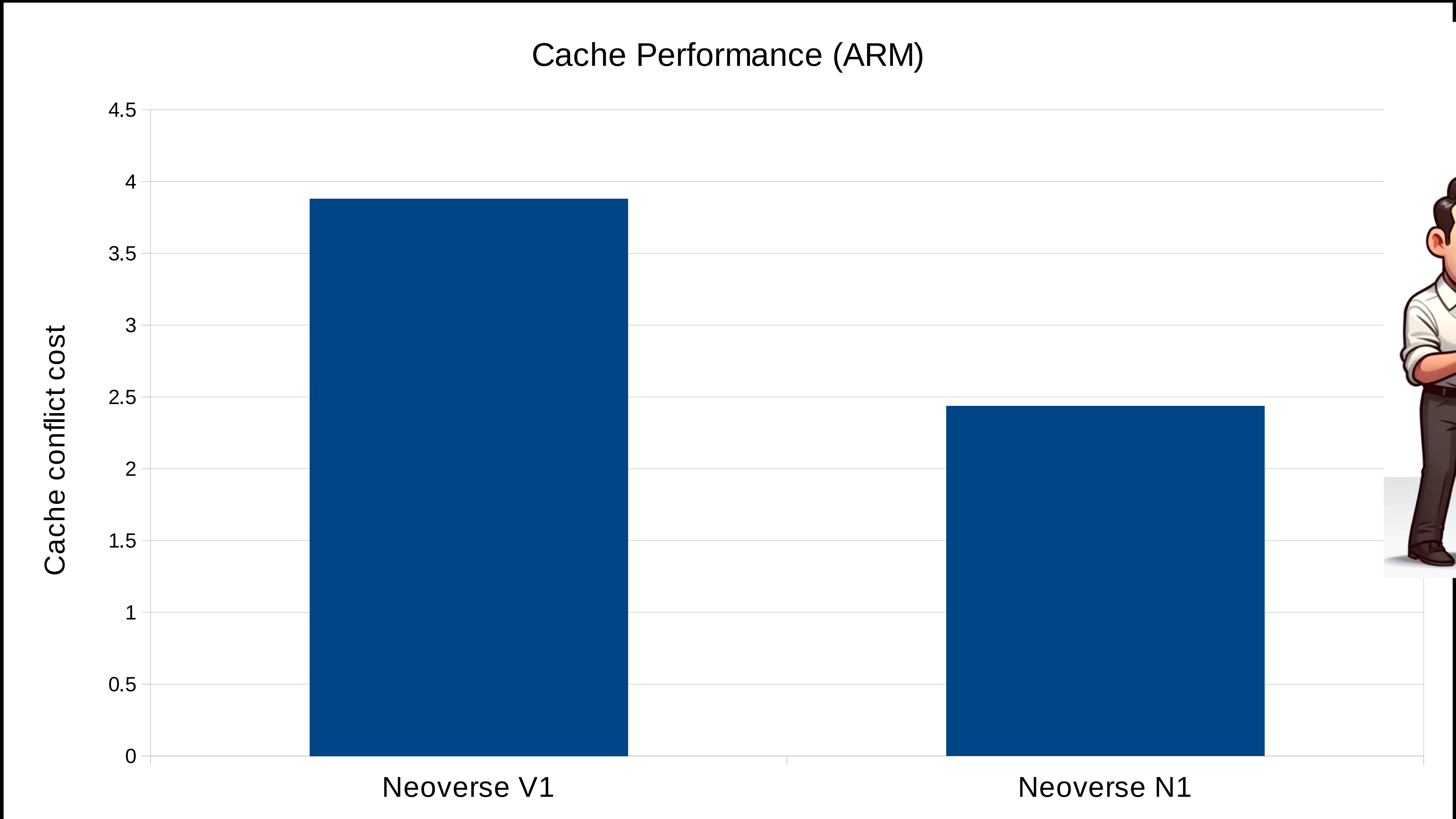


How important is the cache?



- Using 1/64th of the cache hurts
  - Hurts more now

How important is the cache?



- True for ARM CPUs as well

Why pipelines?

- Pipelining is critical for CPUs to execute many operations at once:

```
a1 += (v1[i] + v2[i])*(v1[i] - v2[i])
```

s[i]:v1[i]+v2[i] **d[i]:v1[i]-v2[i]**

s1[i]*d2[i]

Data dependency

SIEMENS

Why pipelines?

- Pipelining is critical for CPUs to execute many operations at once:

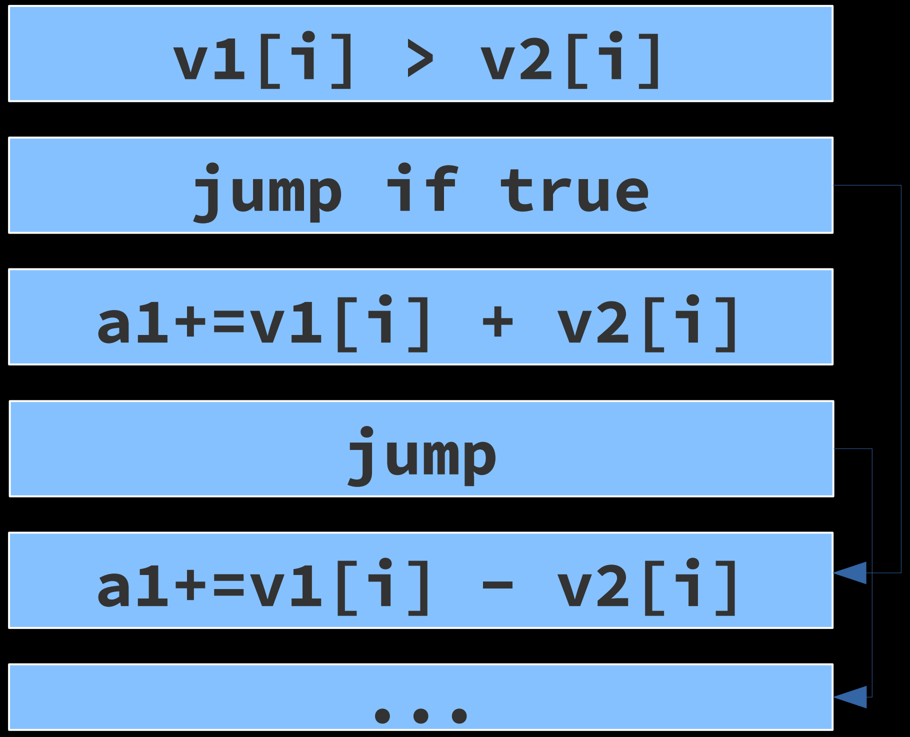
```
a1 += (v1[i] + v2[i])*(v1[i] - v2[i])
```

s[i-1]:v1[i-1]+v2[i-1]	d[i-1]:v1[i-1]-v2[i-1]	s1[i-2]*d2[i-2]
s[i]:v1[i]+v2[i]	d[i]:v1[i]-v2[i]	s1[i-1]*d2[i-1]
s[i+1]:v1[i+1]+v2[i+1]	d[i+1]:v1[i+1]-v2[i+1]	s1[i]*d2[i]

What could go wrong?

- Hard to pipeline code:

```
a1 +=(v1[i] > v2[i]) ? (v1[i] - v2[i]) : (v1[i] + v2[i])
```

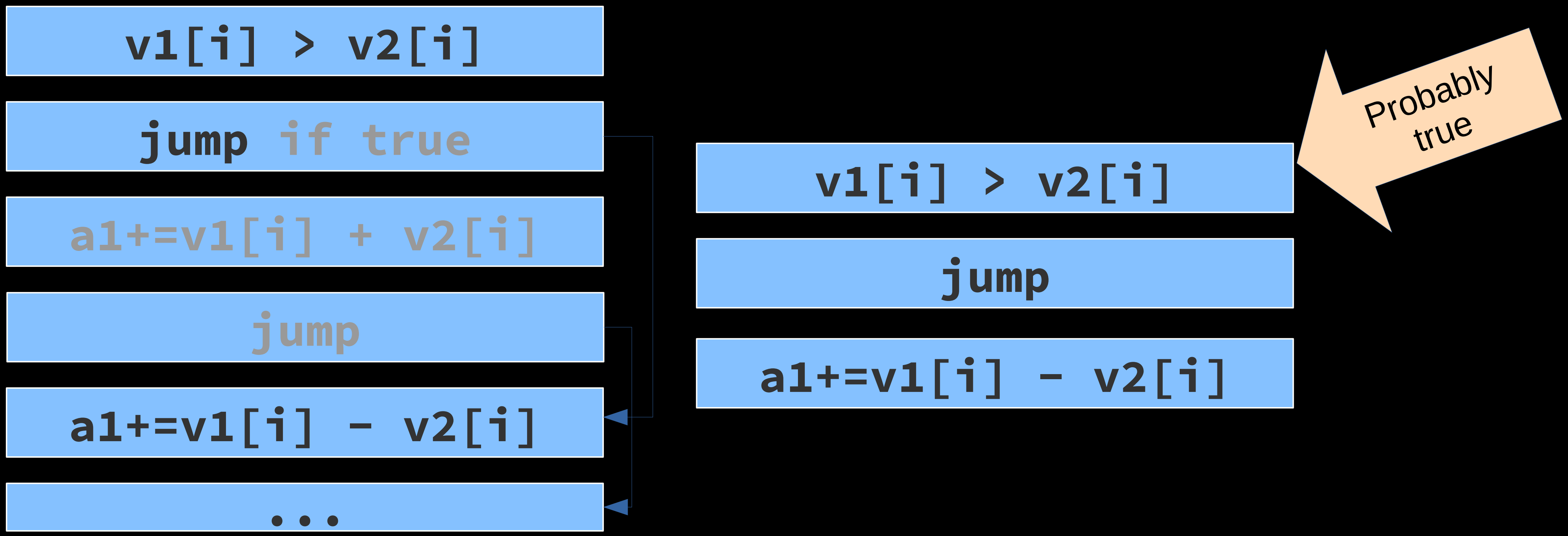


- Pipelining relies on a continuous stream of instructions
- Instructions are fetched, decoded, and executed
- Conditional jumps (branches) disrupt that order
- CPU must wait until it knows which instruction to fetch next
  - Very different from conditional data!

Branch prediction to the rescue!

- Speculatively pipelined code:

```
a1 +=(v1[i] > v2[i]) ? (v1[i] - v2[i]) : (v1[i] + v2[i])
```



Predictable is good

- As long as predictions hold, CPU executes a static instruction sequence
  - Instructions are read, decoded, scheduled, executed, and retired
  - Predictions are confirmed long after the conditional jump was executed
  - Pipelining is near-perfect (conditional instructions still must be executed)

SIEMENS

What could possibly go wrong?

- If a prediction is not confirmed, everything executed on the wrongly guessed branch must be discarded as if it never happened (work is wasted)
  - Decoded instructions are discarded – pipeline flush
  - Currently executing instructions are aborted
  - Memory loads are ignored
- Anything that cannot be discarded must be undone

SIEMENS

Undo this

- Anything that cannot be discarded must be undone…

```
x += condition ? a[i] : b[i];
```

  - Out-of-bounds memory reads?

```
(condition ? a : b) = 0;
```

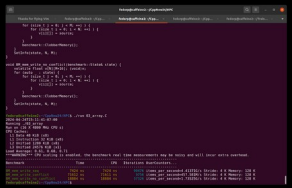
  - Memory writes?

```
if (i < N) a[i] = 0;
```

  - Out-of-bounds writes? Into protected memory?

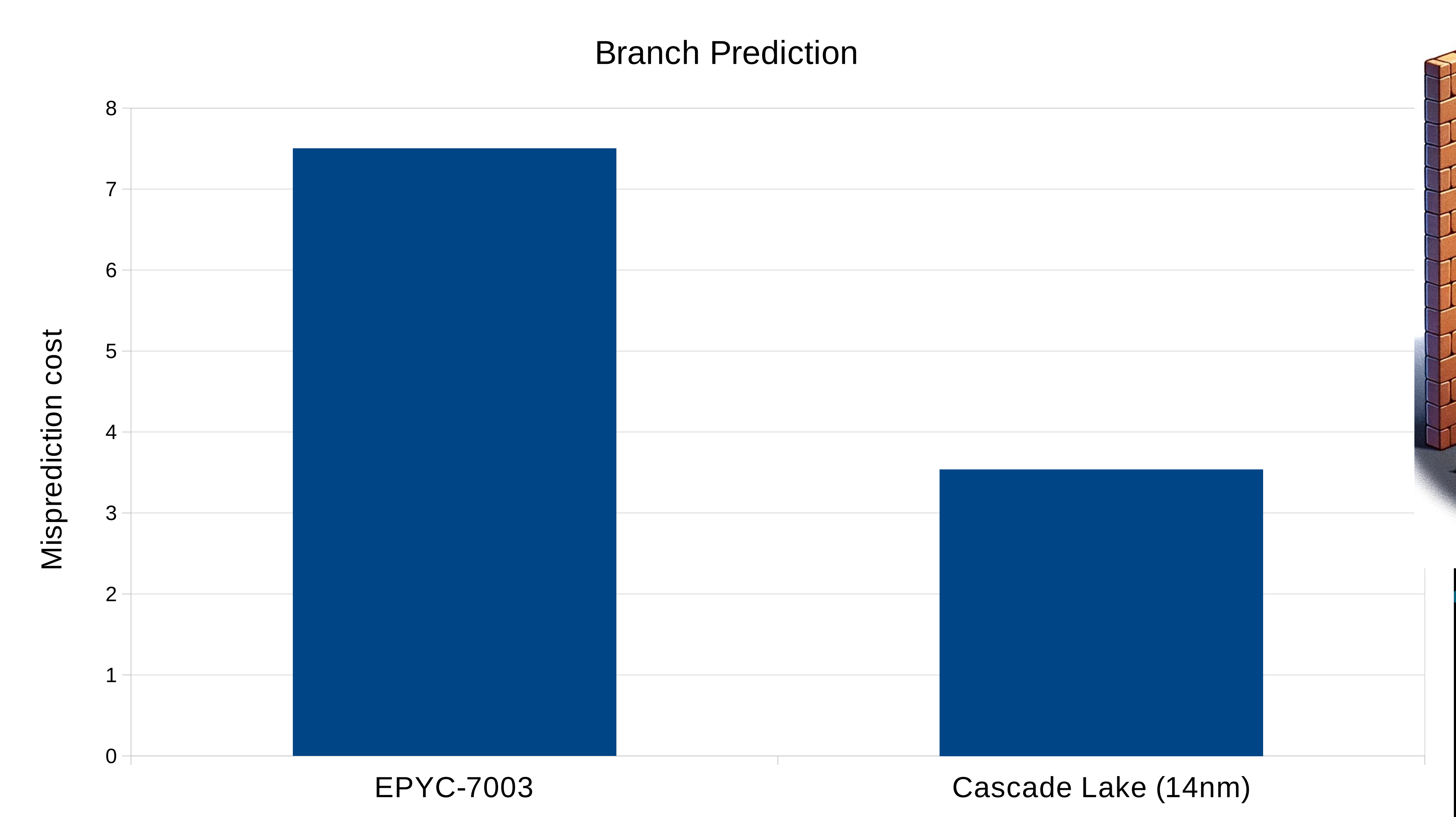
```
if (p) *p = 0;
```

  - Undefined behavior?



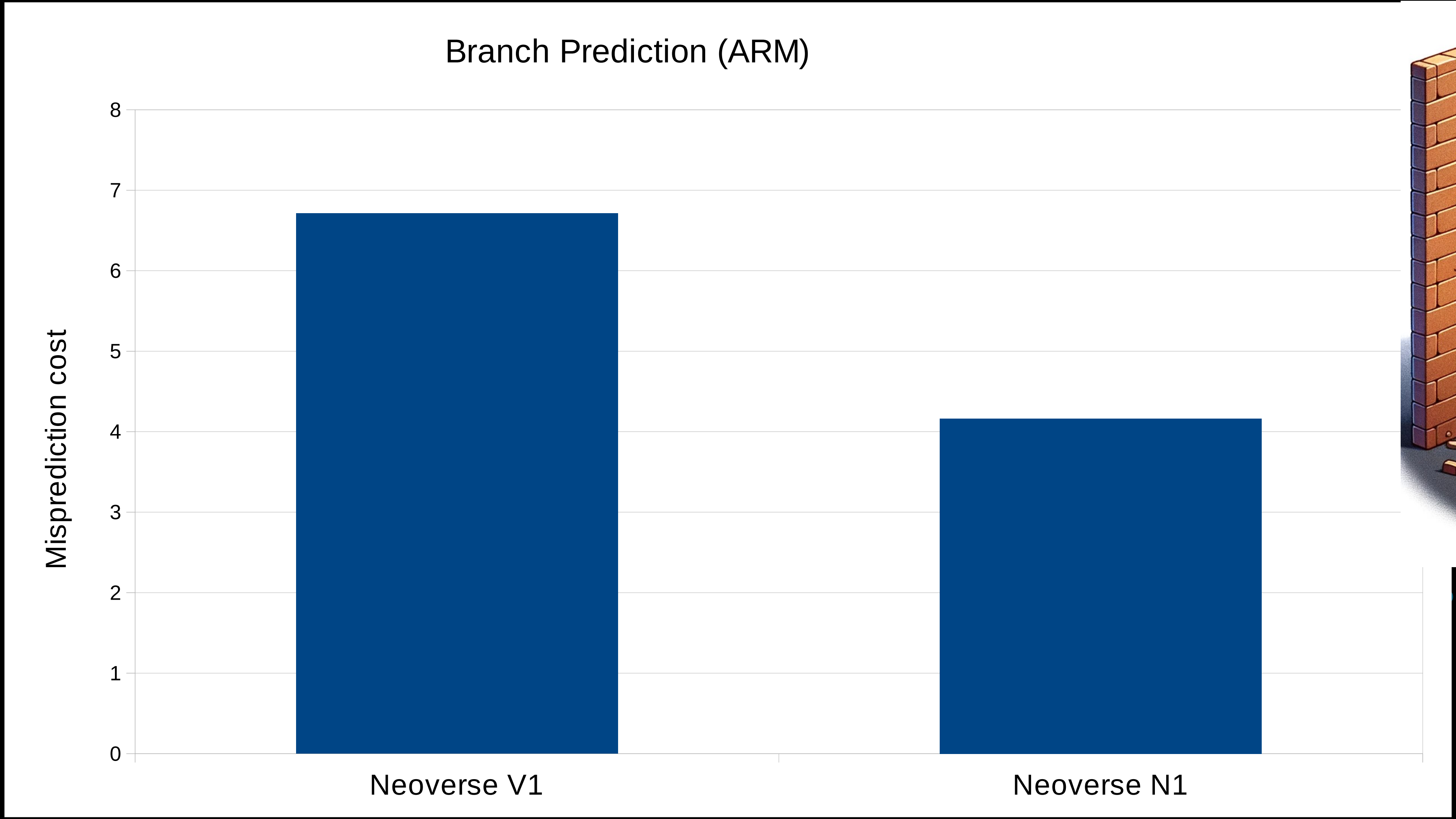
SIEMENS

Cost of branch misprediction



- Mispredictions are expensive
  - And getting more so

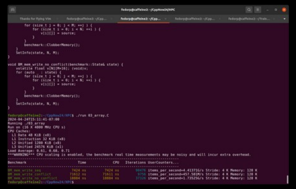
Cost of branch misprediction



- Not just on X86

SIEMENS

Branchless alternative



- If branches are poorly predictable, can we rewrite the code to avoid conditional execution?

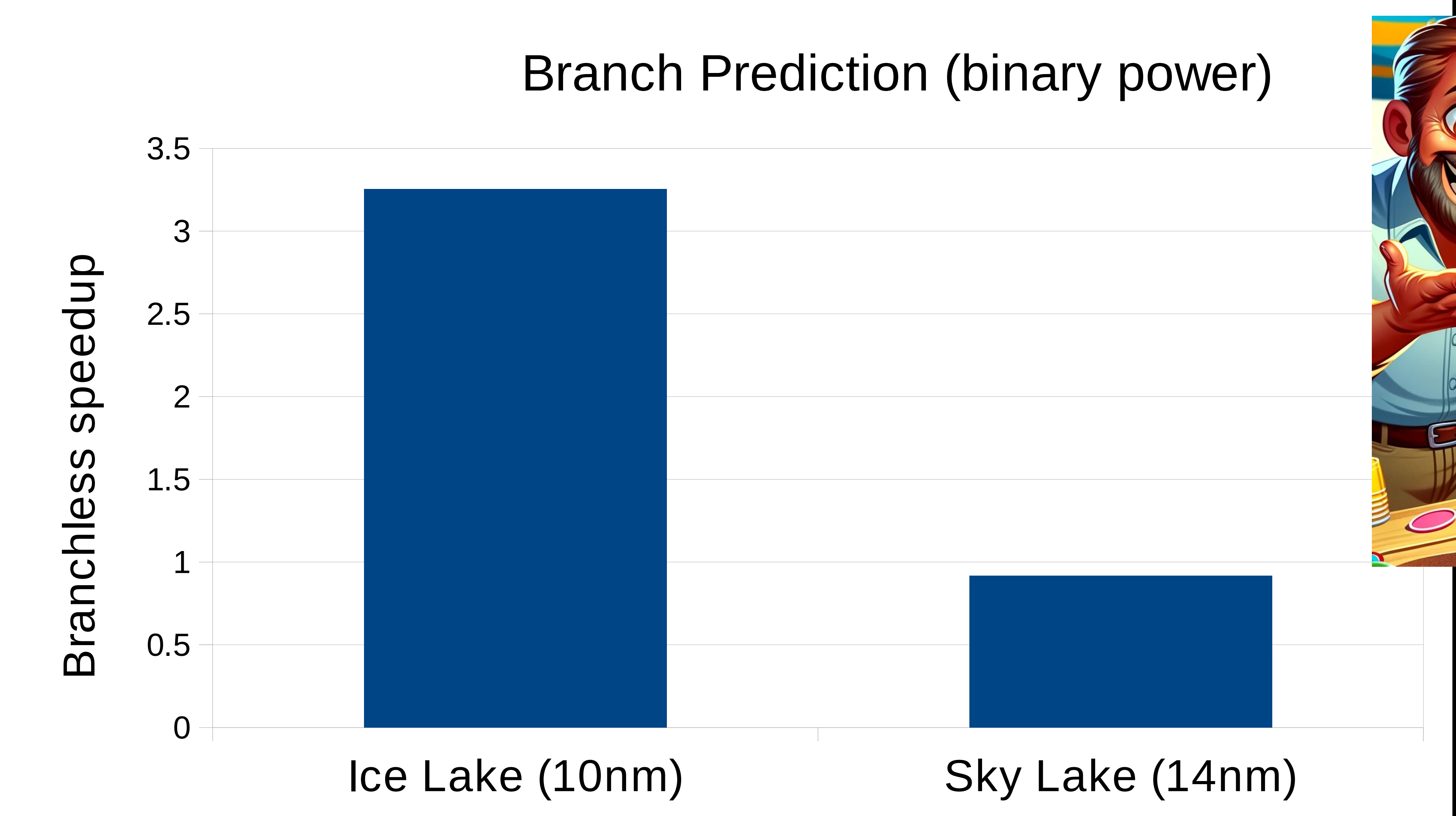
```
inline long binpow(long base, long exponent, long modulus)
{ // (base ^ exponent) % modulus, modulus=2^k
    --modulus;
    long res = 1;
    base = base & modulus;
    while (exponent != 0) {
        if (exponent & 1) {
            res = (res * base) & modulus;
        }
        base = (base * base) & modulus;
        exponent >>= 1;
    }
    return res;
}
```

Random branch

```
long x[2] = {
    res, (res * base) & modulus };
res = x[exponent & 1];
```

Extra work

Branch vs Branchless



- Mispredictions are expensive
  - And getting more so

SIEMENS

Conclusions

- Recent CPUs are (in general) more powerful
- They can execute more operations per cycle
  - Provided there is work to execute
- They have much larger caches
  - and are much more sensitive to poor cache utilization
- They like programs to be predictable
  - and don't respond well to chaos
- Note on ARM processors:
  - They are superscalar (but relative instruction costs are different)
  - They suffer from cache misses and branch mispredictions
    - Not as much as X86 (relatively)

Part II

- Of course the CPU does that!
- Makes sense that the CPU would do that…
- The CPU does WHAT?!

Aside: What is concurrency?

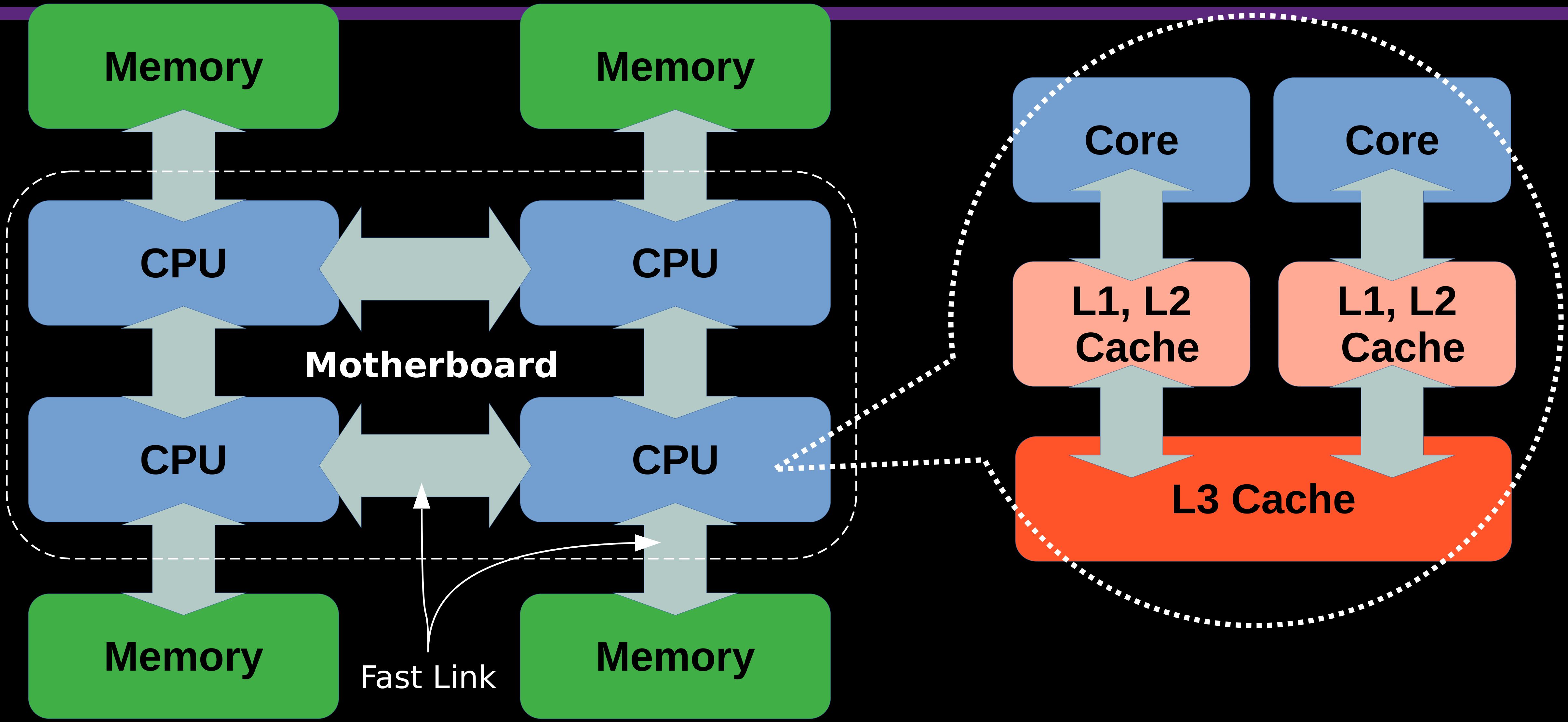
- For the purposes of THIS talk:
- Many threads run on multi-processor machine
- Shared state
  - Memory shared between threads
  - Locking or lock-free synchronization
- High CPU load

SIEMENS

Aside: What is concurrency?

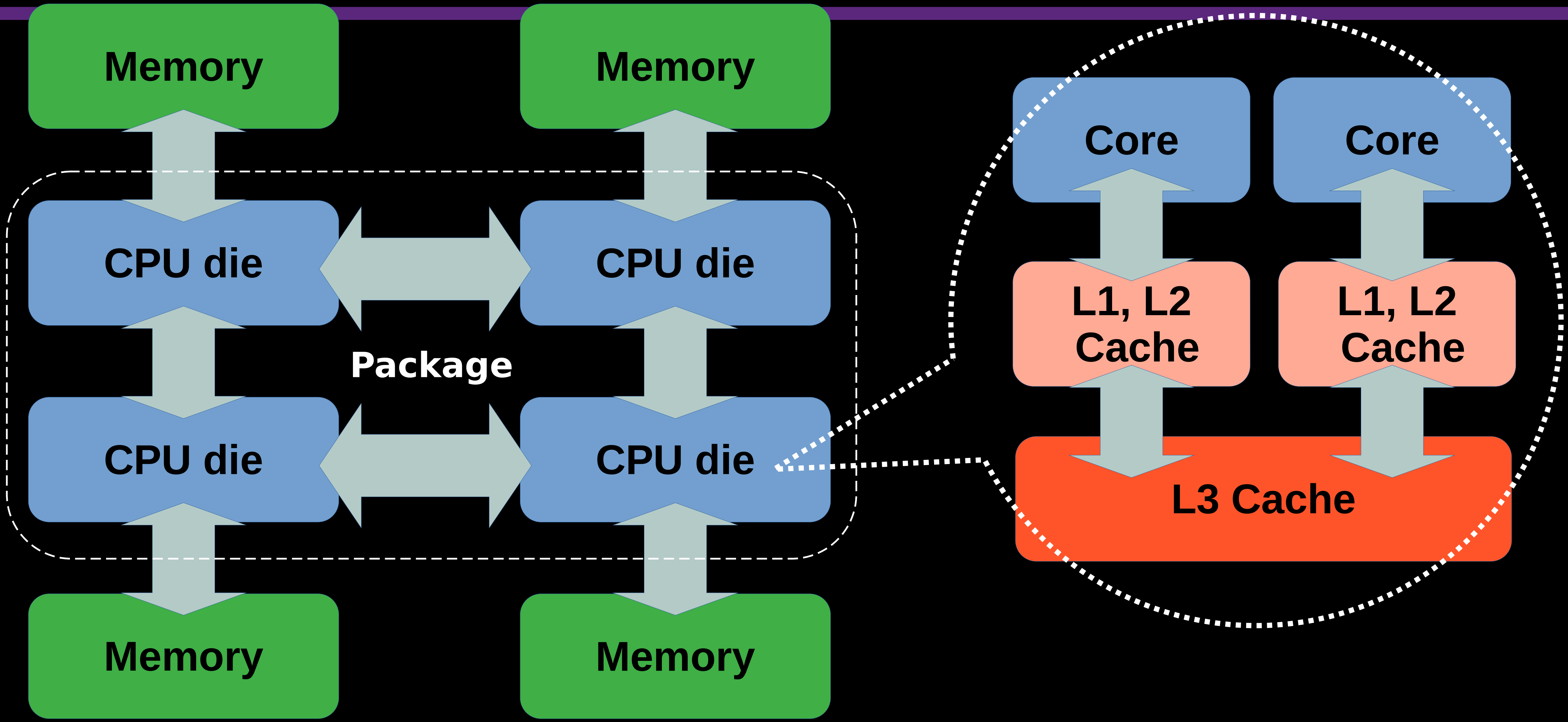
- What is not considered:
  - Entirely separate threads or processes
  - Often pinned to specific cores
- For applications limited by latency not CPU, talk is only partially relevant

Modern server CPUs are designed for NUMA



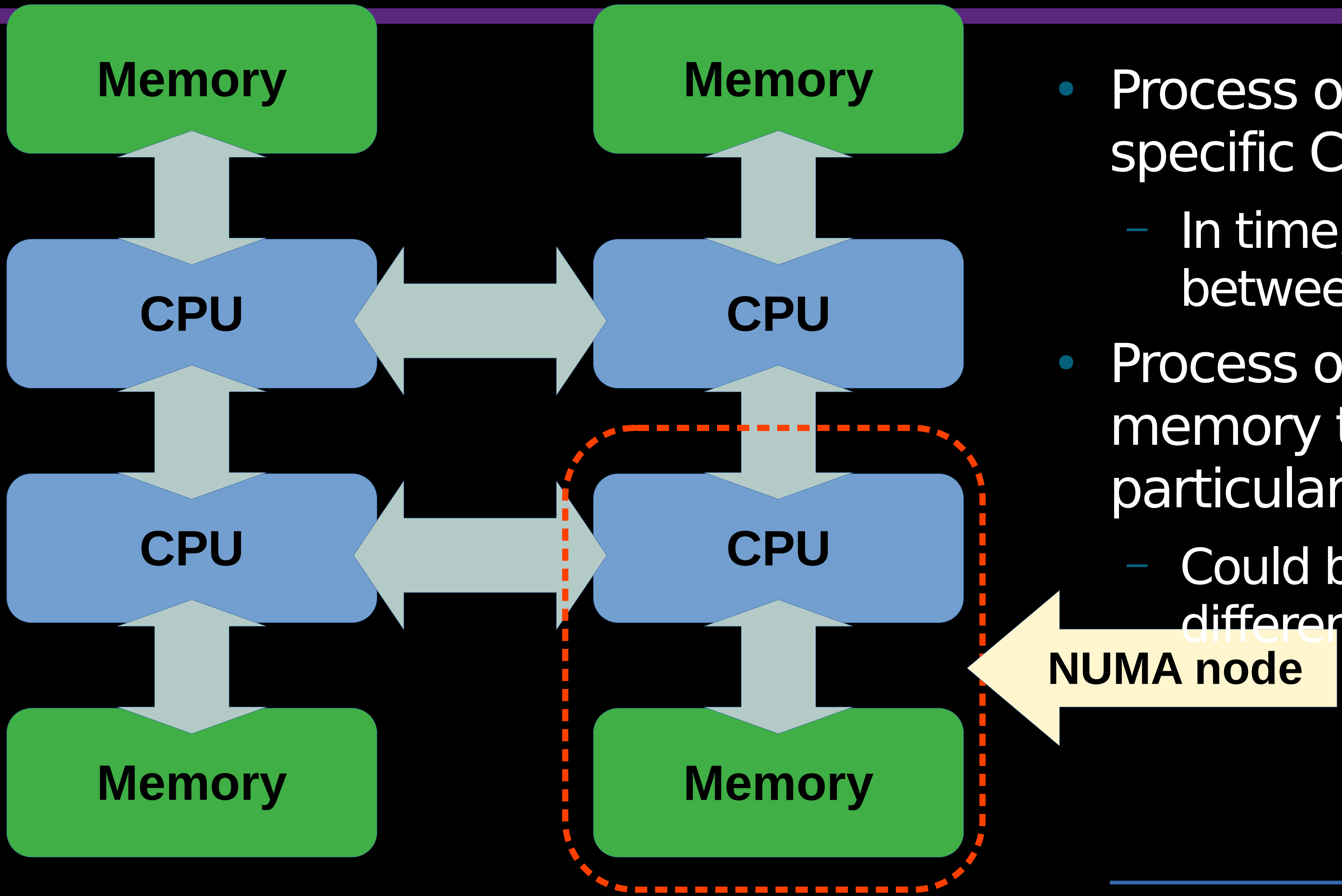
SIEMENS

Latest generation CPUs are NUMA in a package



SIEMENS

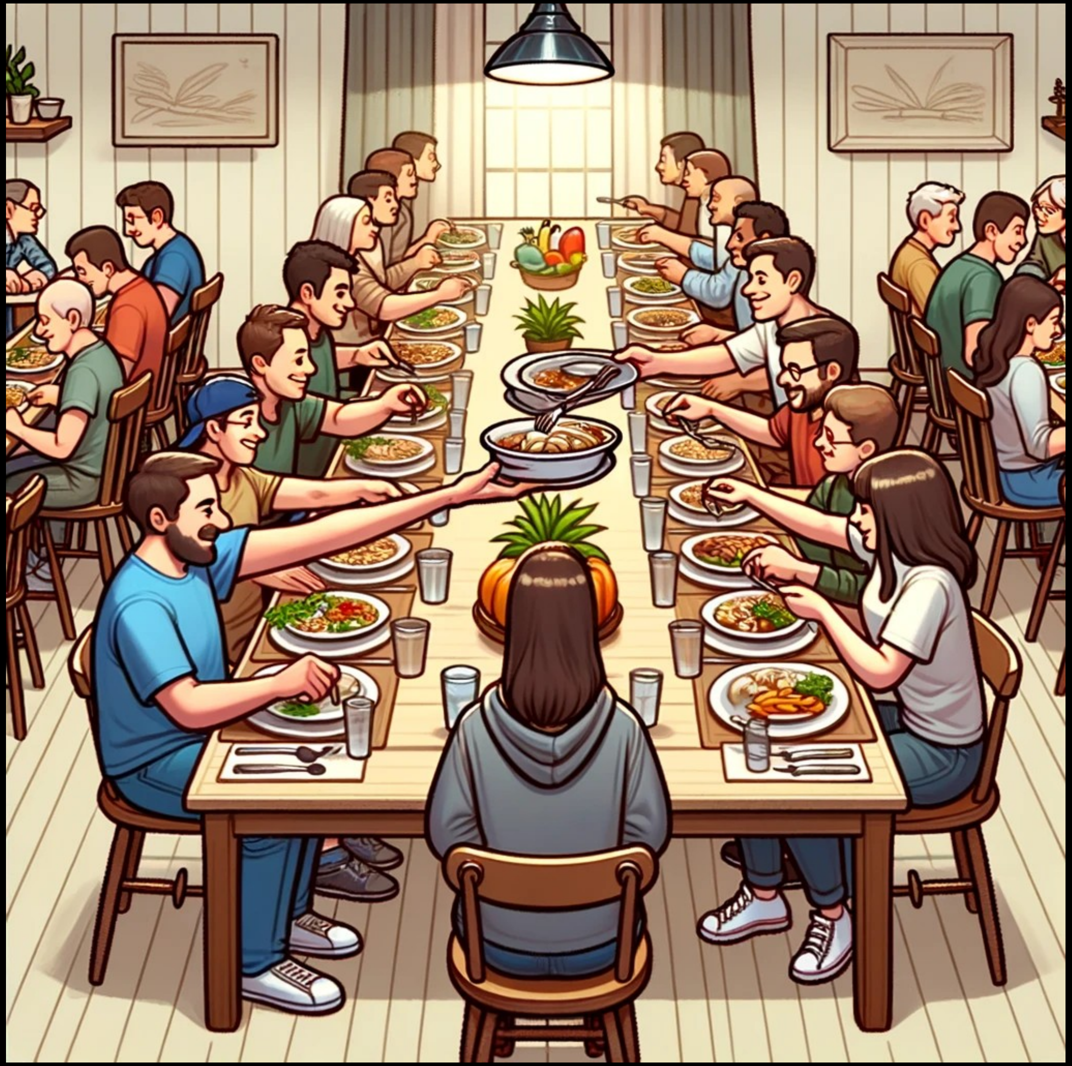
Non-Uniform Memory Architecture



- Process or thread runs on a specific CPU at any time
  - In time, processes can move between CPUs and nodes
- Process or thread accesses memory that resides on a particular node
  - Could be the same node or different

Performance implications of NUMA

- Accessing local memory is different from accessing non-local memory
- To access memory from another node, that node's CPU must be involved

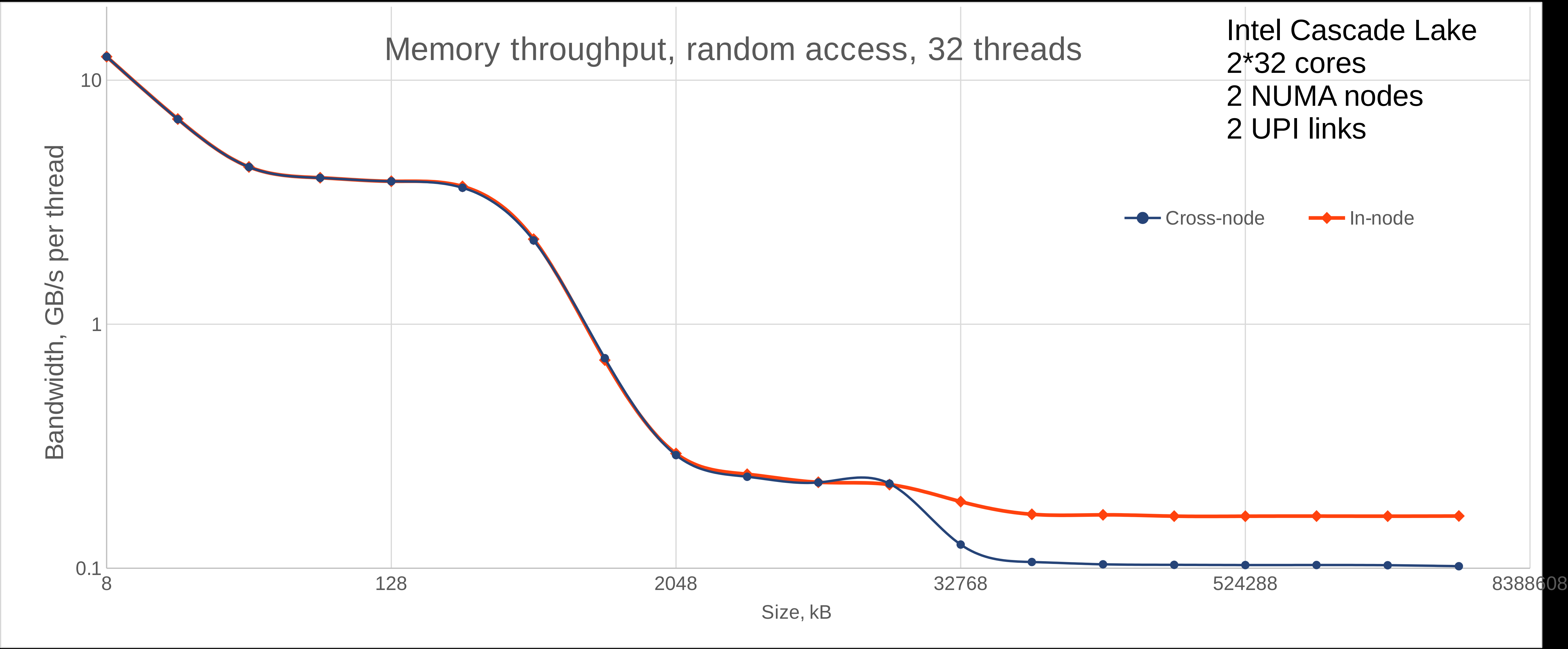


SIEMENS

Performance implications of NUMA

- Accessing local memory is different from accessing non-local memory
- To access memory from another node, that node's CPU must be involved
- Peak bandwidth is reduced by 40% to 60% for cross-node access
  - This is remarkably universal for x86 hardware from 2014 to 2022
  - Some systems show degradation for even 1 thread, some do not
  - Newer systems have higher bandwidth but larger penalty for accessing cross-node memory
- Cache bandwidth is not affected (without data sharing)
- For multi-die processors, the hierarchy is 3 levels: in-node, cross-node within the socket, cross-node/cross-socket (the slowest)

Memory Throughput in NUMA systems



SIEMENS

Performance implications of NUMA

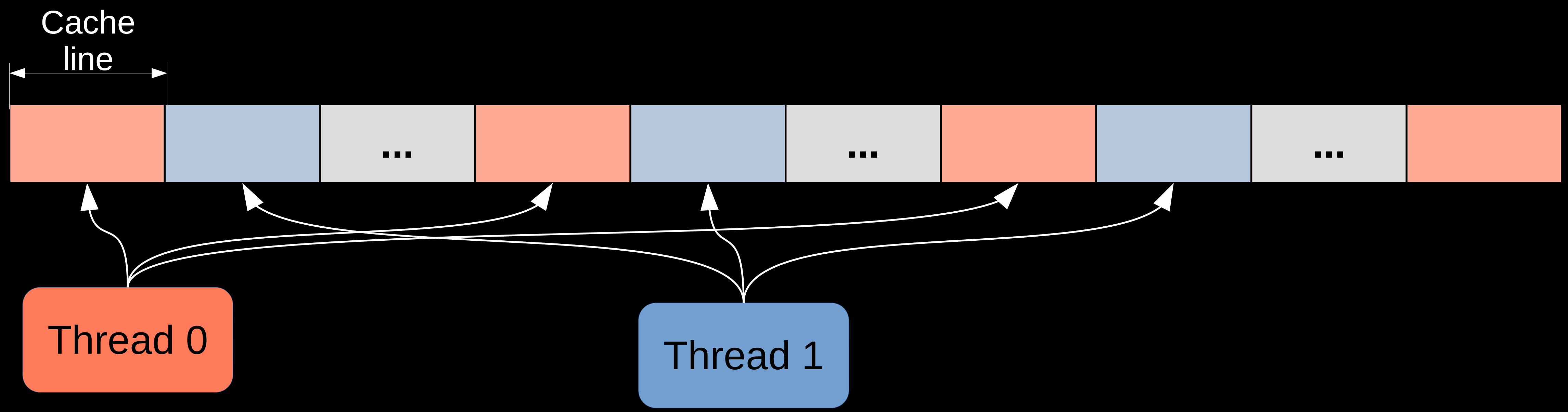
- Accessing local memory is different from accessing non-local memory
- To access memory from another node, that node's CPU must be involved
- Memory latency is significantly larger for cross-node accesses
  - Random memory access is largely limited by memory latency
- More recent CPUs have longer latencies (price of higher bandwidth)
  - QPI cross-node latency: 120-140 ns (40% over in-node)
  - UPI cross-node latency: 140-160 ns (80% over in-node)
- Cross-node accesses involve LL (last-level) cache miss
- Cached data accesses are unaffected by NUMA (without data sharing)

NUMA for concurrent programs

- Memory-bound programs are affected directly by bandwidth and latency
  - This is without data sharing (independent threads)
- Most concurrent program have shared data accessed concurrently
- There is sharing and then there is sharing
- Accessing the same address (true sharing) requires locks or atomic access
  - This is slow even between CPU cores within the same node
- Accessing different addresses within the same cache line (false sharing)
  - All the pain of sharing without the benefits (just don’t do it)
- Accessing different cache lines
  - This is usually as fast as any other memory accesses
  - Except on NUMA systems

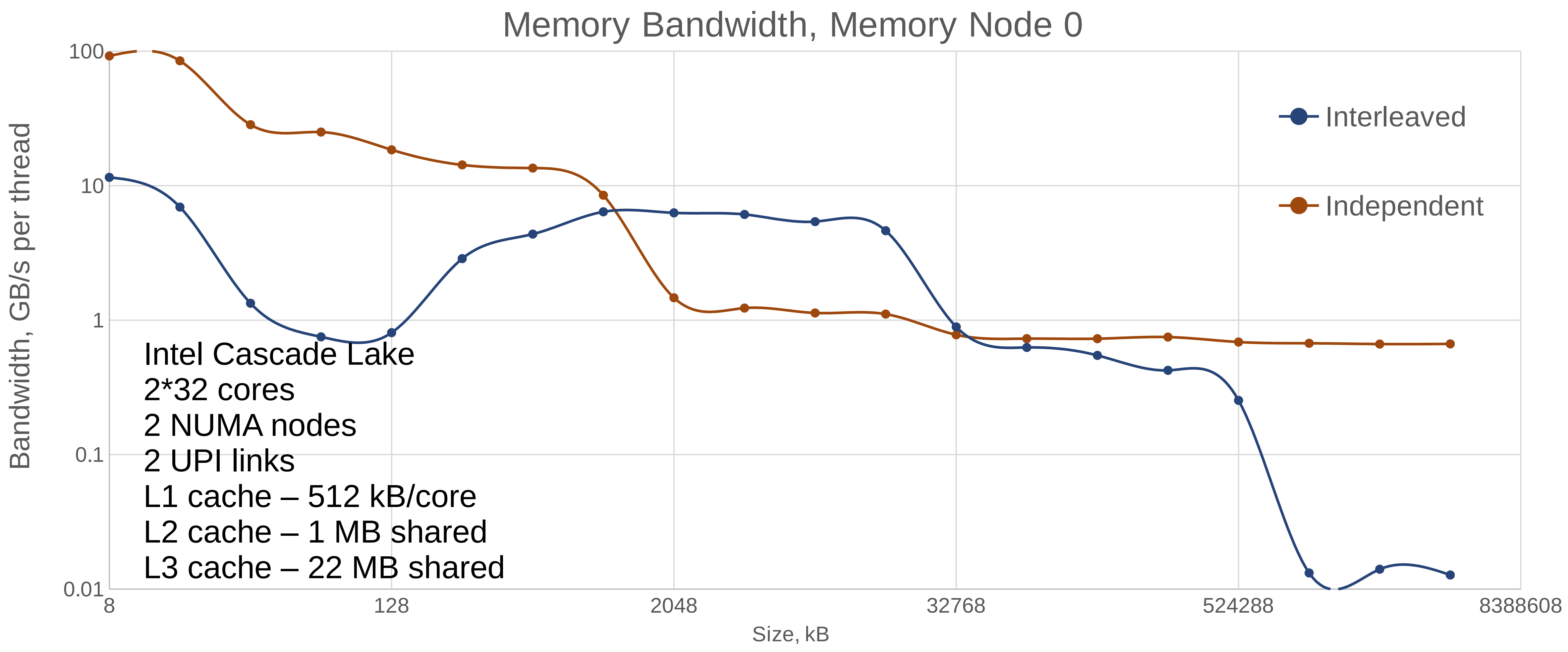
What is “concurrent access” on NUMA?

- Test: all data resides on memory node 0
- 64 threads run on 64 cores (1 thread per core, all nodes)
- No cache lines shared between threads
- Thread data is interleaved and accessed sequentially (with stride)



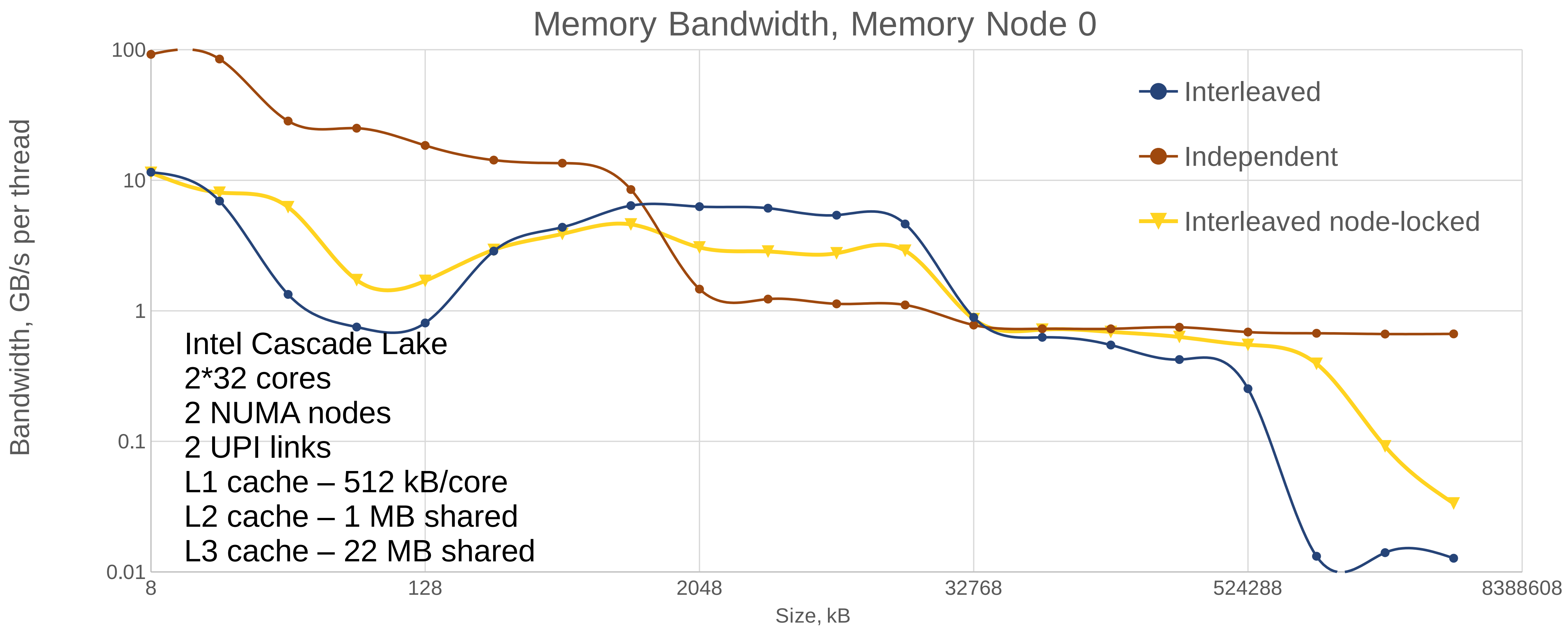
SIEMENS

NUMA – another kind of false sharing



SIEMENS

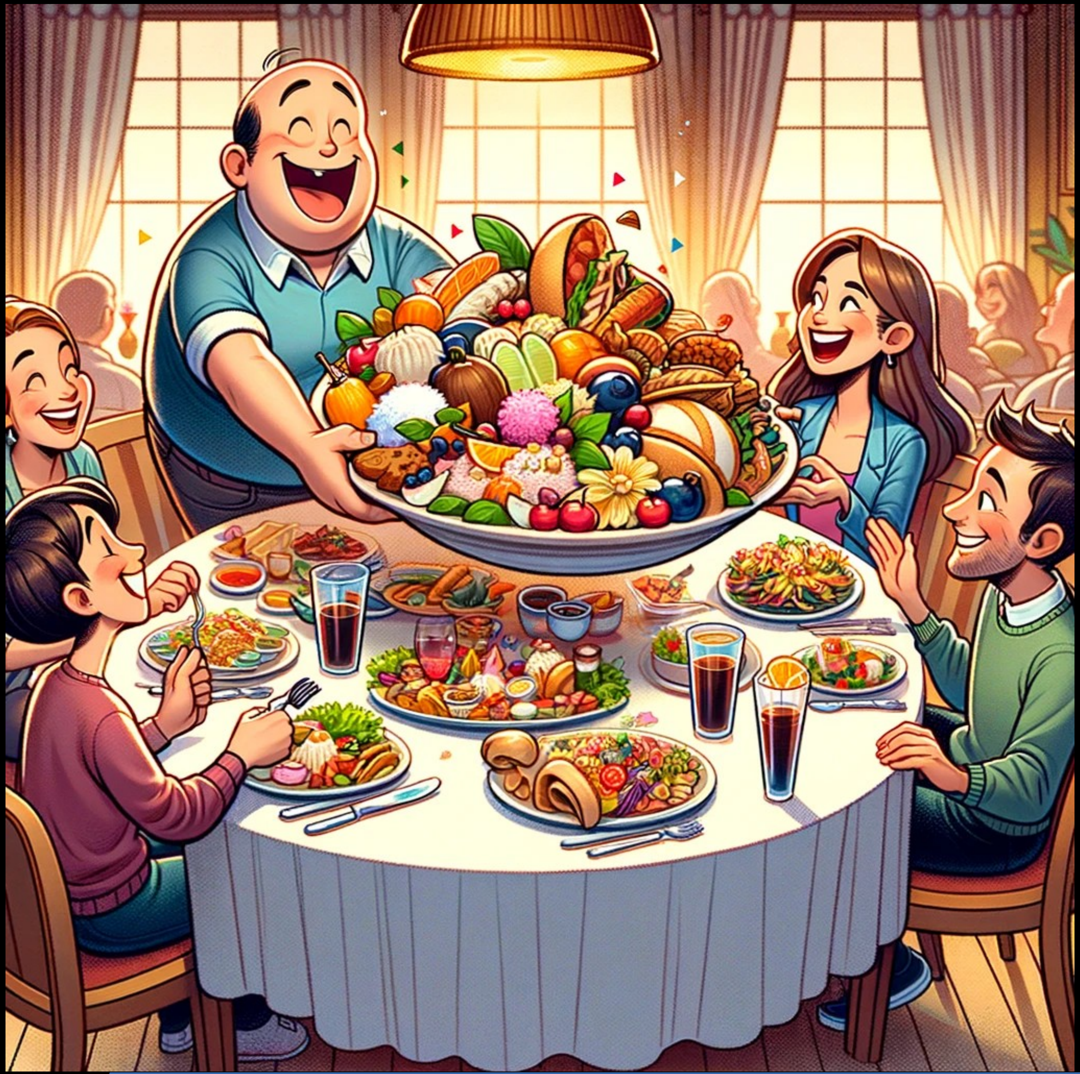
NUMA – another kind of false sharing



SIEMENS

NUMA for concurrent programs

- Accessing different cache lines
  - This is usually as fast as any other memory accesses
  - Except on NUMA systems
  - CPUs on a NUMA machine exchange larger blocks of memory



SIEMENS

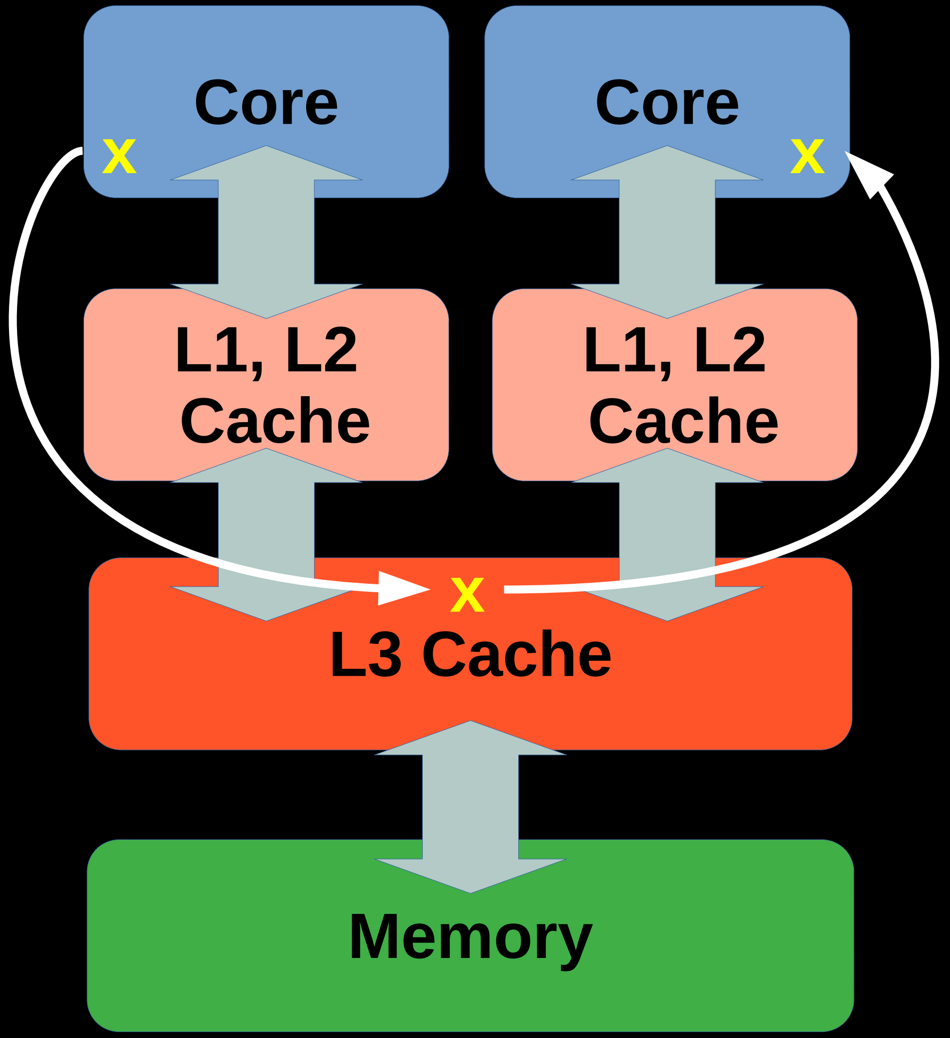
Performance implications of NUMA

- Accessing local memory is faster than accessing non-local memory
- Per-thead memory and cache locality are more important with NUMA
- Concurrent processing of data in close proximity is slower
  - So much slower that using half the cores may be faster
- Cores exchange memory in cache lines (64B)
- NUMA nodes exchange memory in pages (4K on X86, up to 32K on ARM)

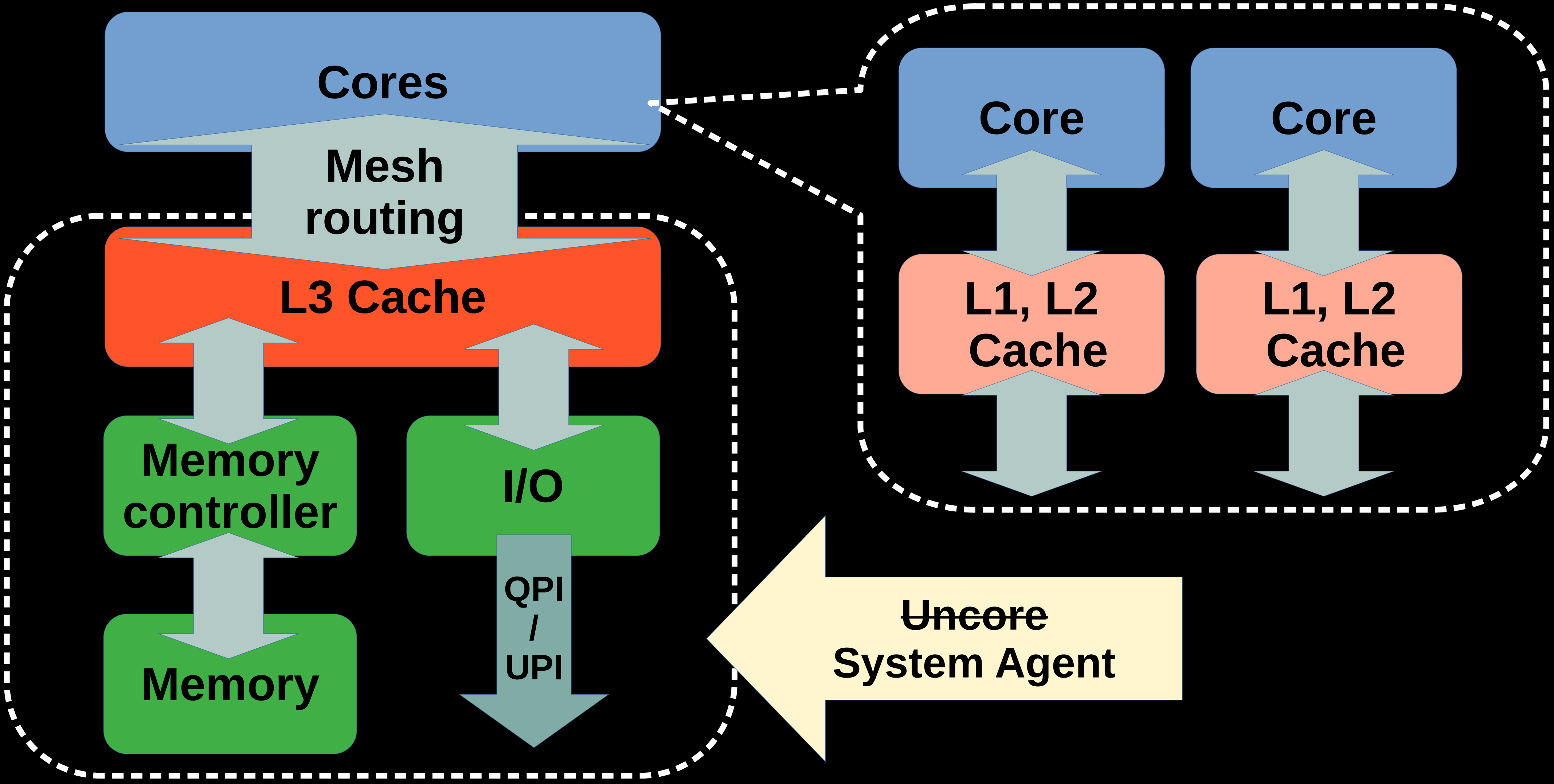
Data sharing on NUMA systems

- Concurrent access of the same data – true data sharing
  - Throughput is limited by the latency of atomic accesses
- Most concurrent programs have share data
- Some algorithms are perfectly parallel, no data sharing at all, but...
- Any thread scheduler is going to have some shared data
  - Task queue (queue lock or atomics)
  - Task count or whatever you wait on
  - Other data, depending on the thread scheduler/pool implementation

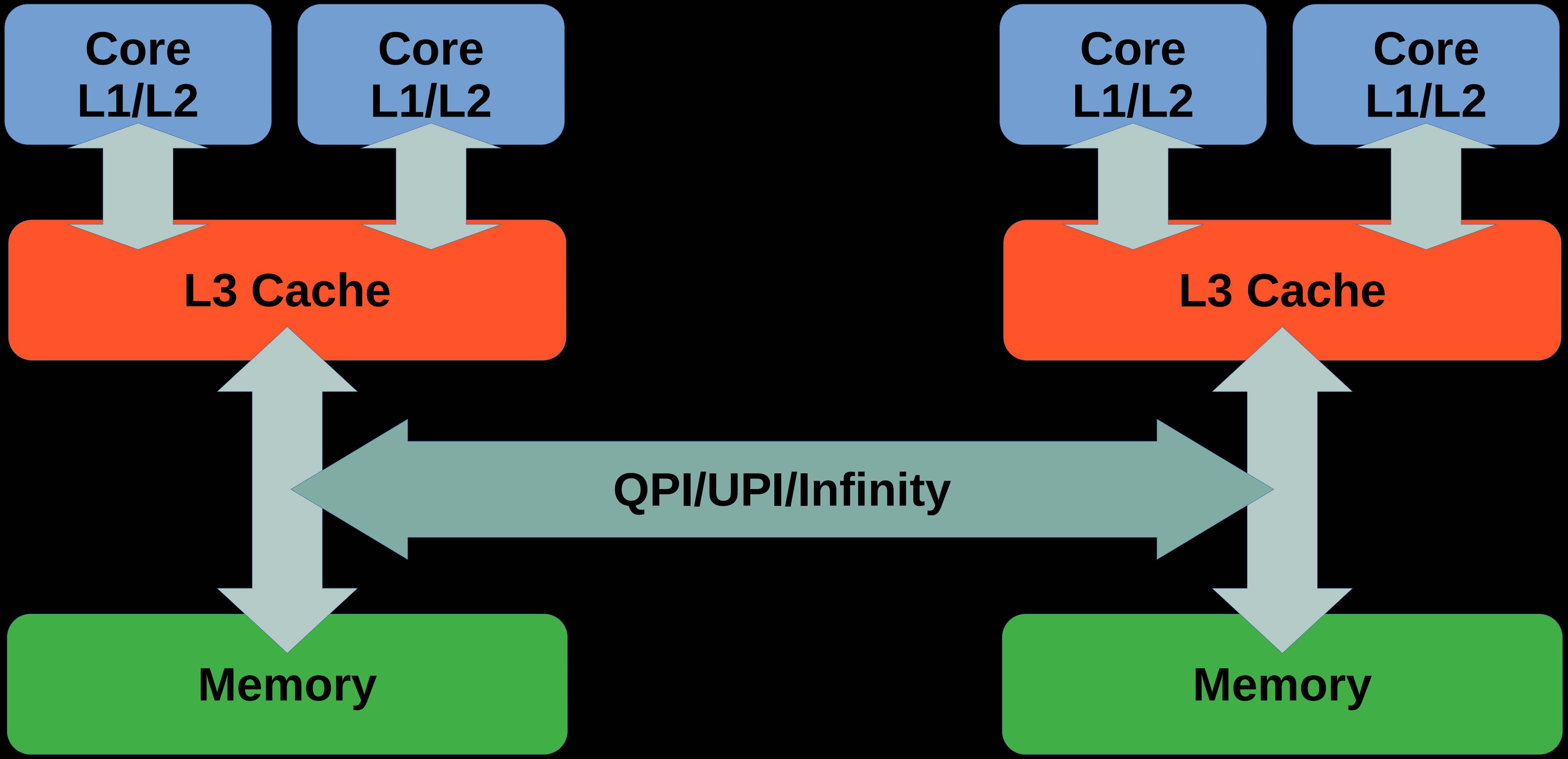
This is your data sharing



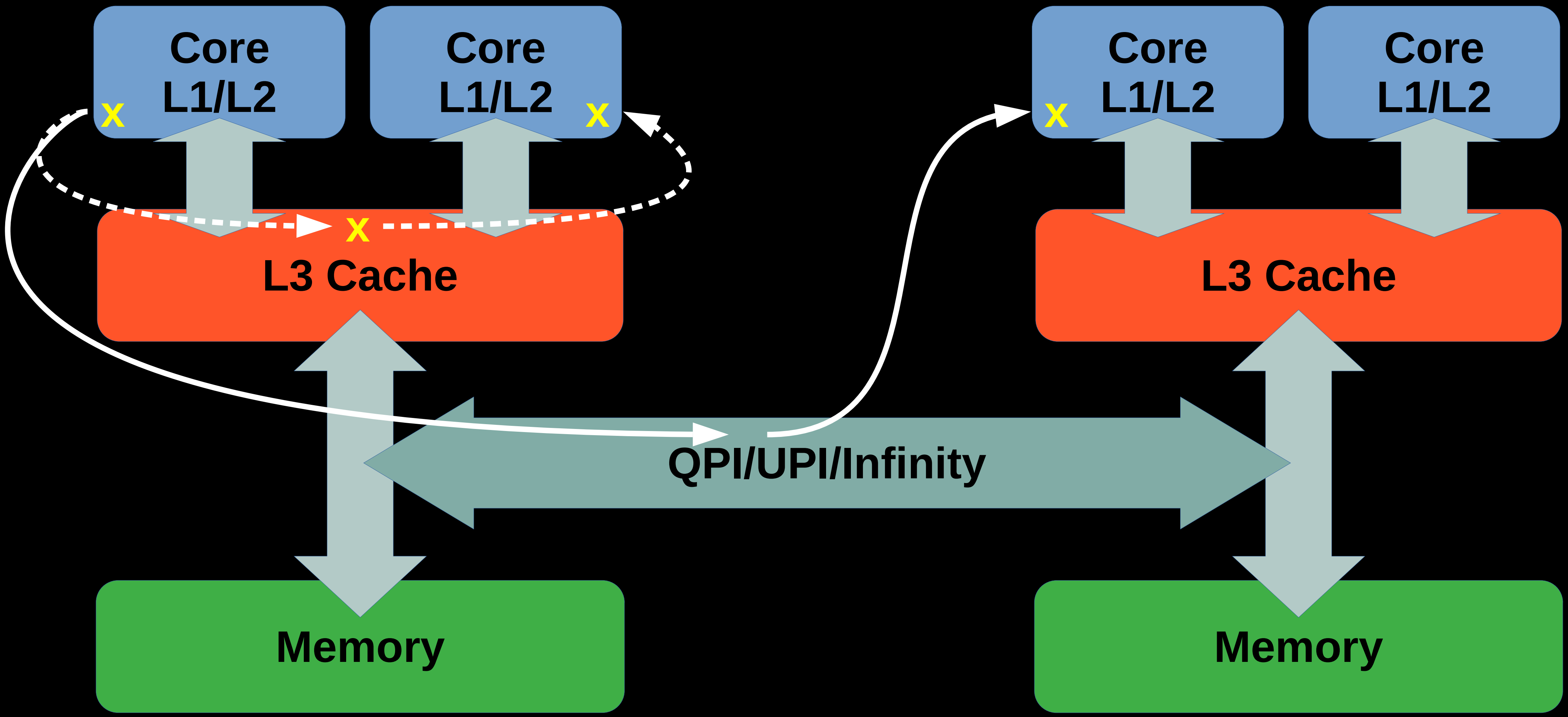
Multi-threading and latency



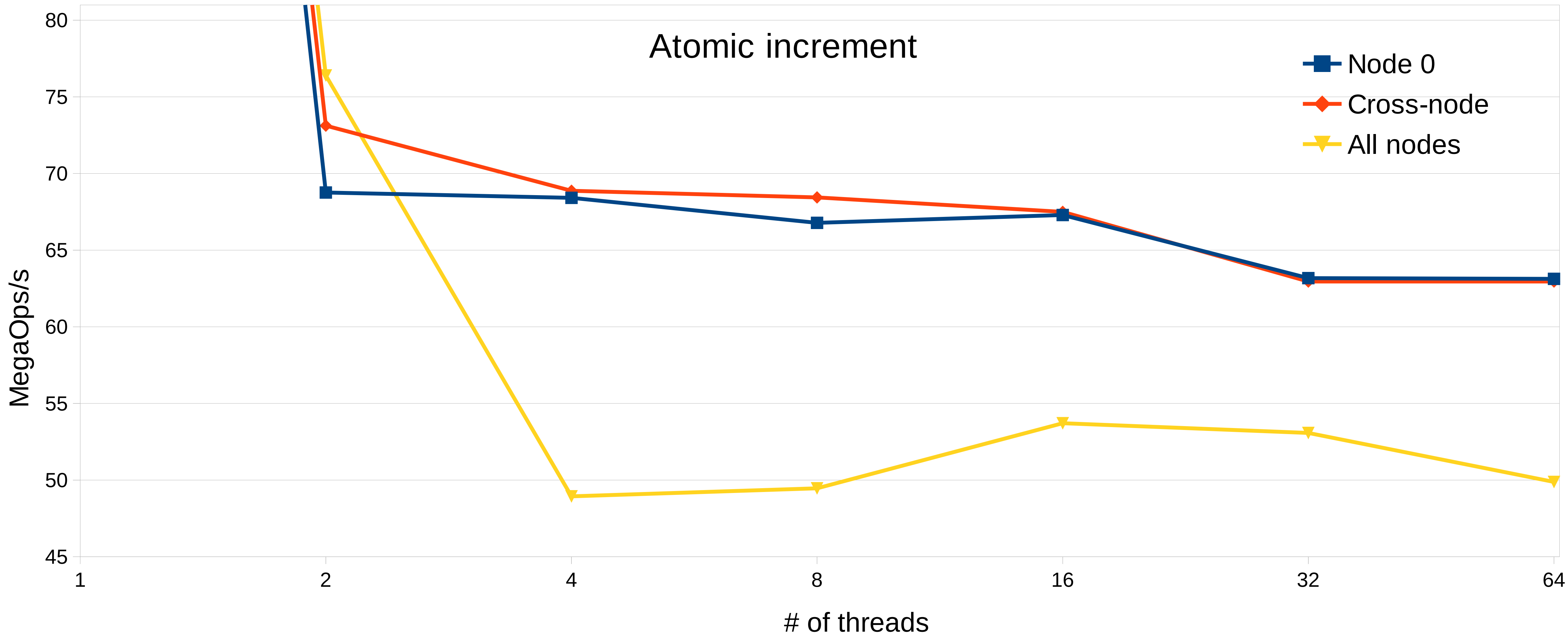
Multi-threading and latency



This is your data sharing on NUMA



Is NUMA performance hurt by data sharing?



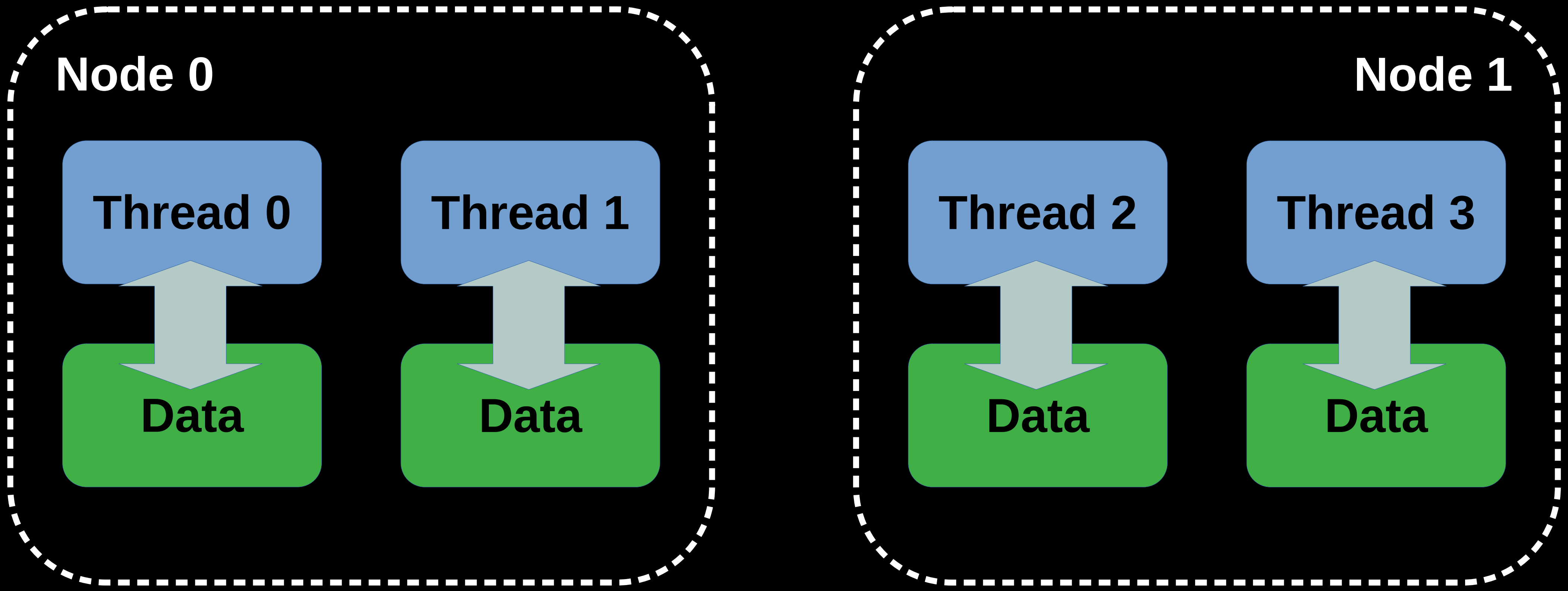
SIEMENS

Is NUMA performance hurt by data sharing?

- A single atomic operation often shows some slowdown on NUMA systems
  - Especially for Read-Modify-Write operations
  - Usually not for atomic load or store (so RCU algorithms may have advantage)
  - Older Intel QPI systems may not show any slowdown at all
  - AMD EPYC systems show less NUMA penalty than recent Intel systems
- Most concurrent data structures and executors use shared variables and atomic operations
  - Locks too (usually atomic exchange)
- Maintaining consistent global shared state on NUMA systems is expensive

Memory-bound programs

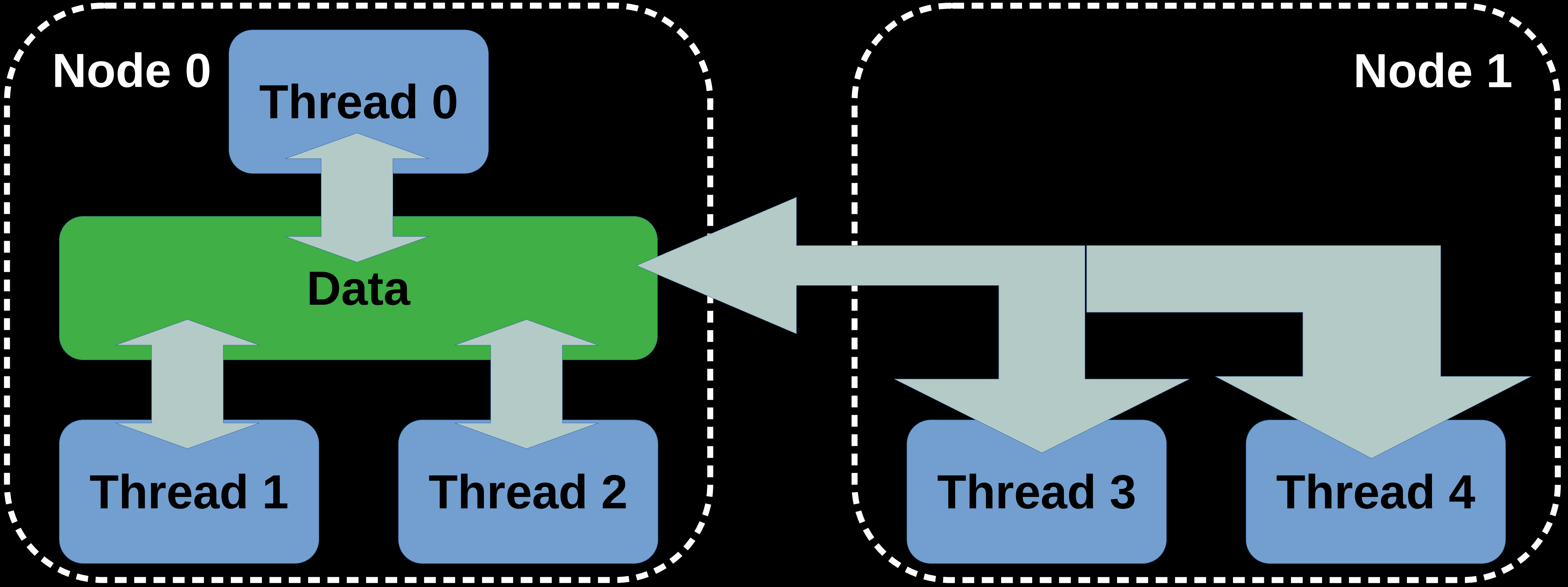
- If each thread mostly uses its own memory, you can bind threads to NUMA nodes (but be mindful of load balancing)
  - Verify that it is needed – OS scheduler usually does good enough job



SIEMENS

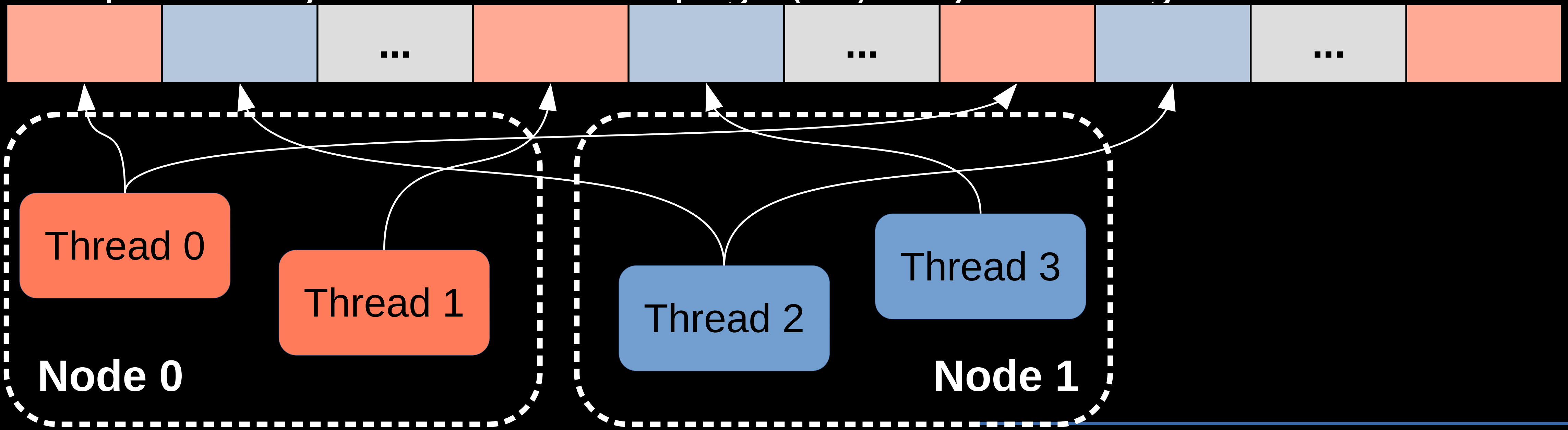
Memory-bound programs

- Memory is usually allocated on the calling thread's node by default
  - But if the parent thread allocates memory, you may want to explicitly move it
  - Recent Linux kernels will eventually do it for you (with some caveats)



Memory-bound programs

- If multiple threads update the same data structure (but not the same memory locations) the working sets of these threads must be separated
  - Read-only data is OK
- Algorithm changes may be required to increase per-thread work size
  - Separated by how much? One page (4K) may be enough



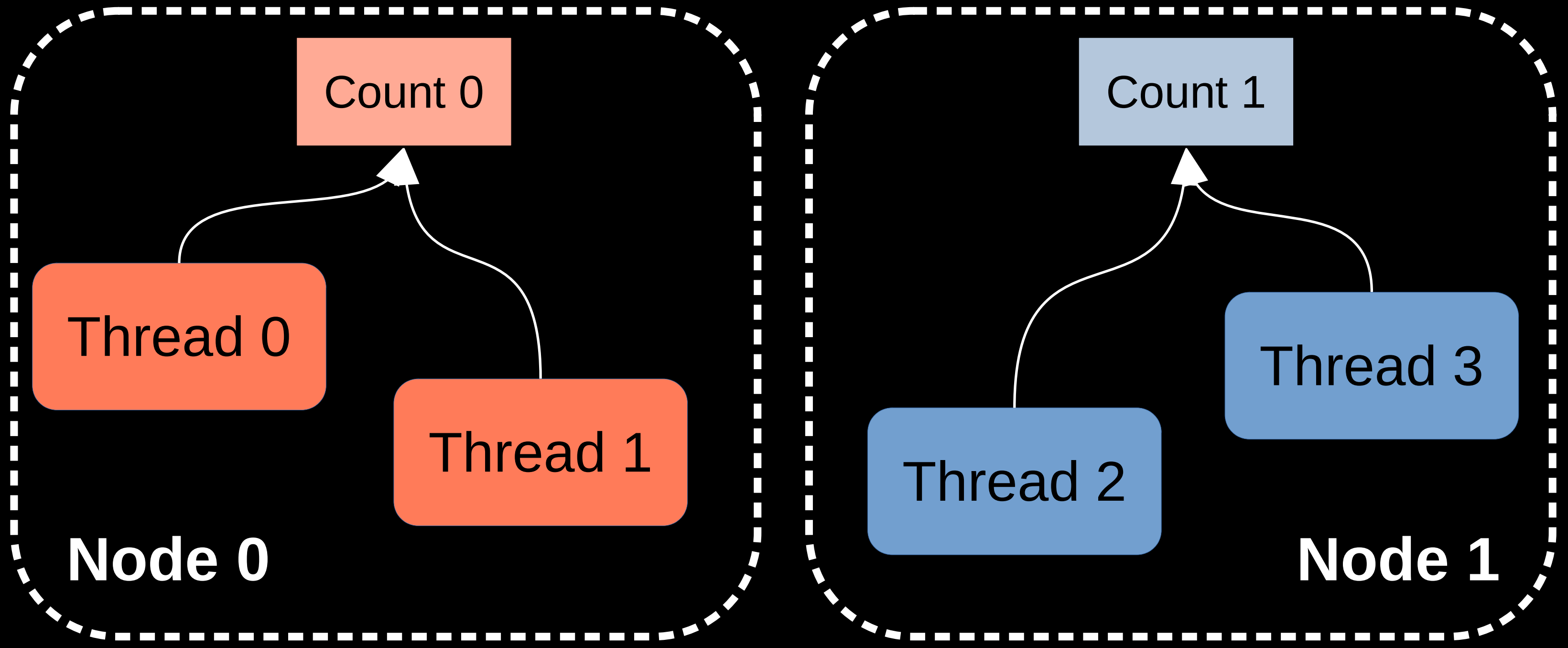
SIEMENS

Memory-bound programs

- If each thread mostly uses its own memory, you can bind threads to NUMA nodes (but be mindful of load balancing)
  - Verify that it is needed – OS scheduler usually does good enough job
  - Memory is usually allocated on the same node by default
  - But if the parent thread allocates memory, you may want to explicitly move it
  - Recent Linux kernels will eventually do it for you (with some caveats…)
- If multiple threads update the same data structure (but not the same memory locations) the working sets of these threads must be separated
  - Read-only data is OK
- Algorithm changes may be required to increase per-thread work size
  - Separated by how much? One page (4K) may be enough (but there are complications…)

Latency-bound programs

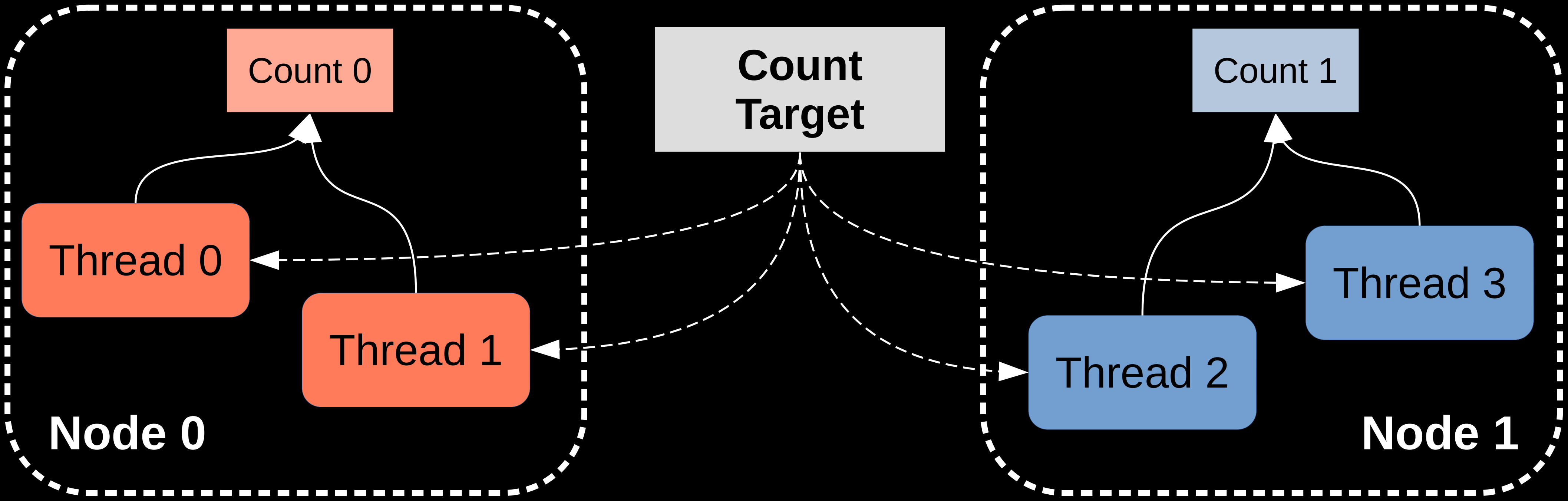
- If cross-node traffic caused by some shared data is significant, the implementation has to be redesigned to use per-node state



- But we still need a global state…

NUMA-aware counter

- If cross-node traffic caused by some shared data is significant, the implementation has to be redesigned to use per-node state
- Synchronization is likely to be more complex



SIEMENS

NUMA-aware counter

```
class NUMA_task_count {
  static constexpr size_t max_node_count = 64;
  task_count_t done_task_counts_[max_node_count];
  std::atomic<size_t> task_count_;
  public:
  size_t Count() const {  // This is what we wait on
    size_t res = task_count_.Acquire_Load();
    for (size_t i = 0; i != num_nodes; ++i)
      res -= done_task_counts_[i].value.Acquire_Load();
    return res;
  }
}; // NUMA_task_count
```

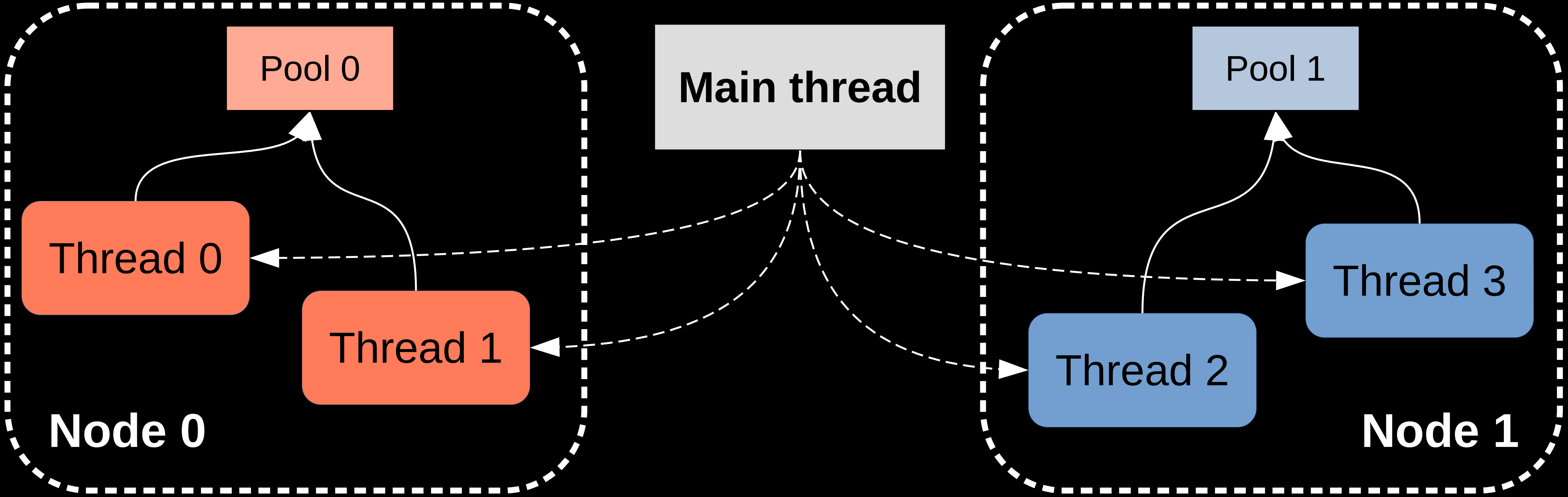
size_t
alignas(64)

read-only during wait

updated during wait

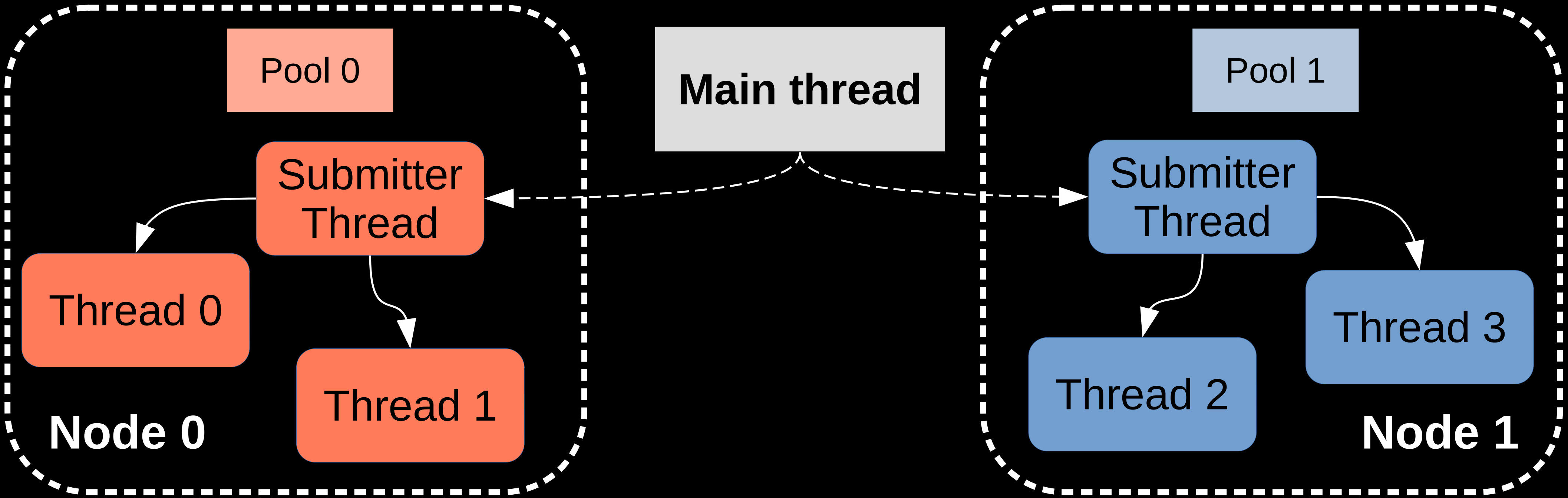
NUMA-aware thread pool

- Per-node pools, cross-node work stealing (only when one node is empty)
- Task submission from the main thread is the bottleneck



NUMA-aware thread pool

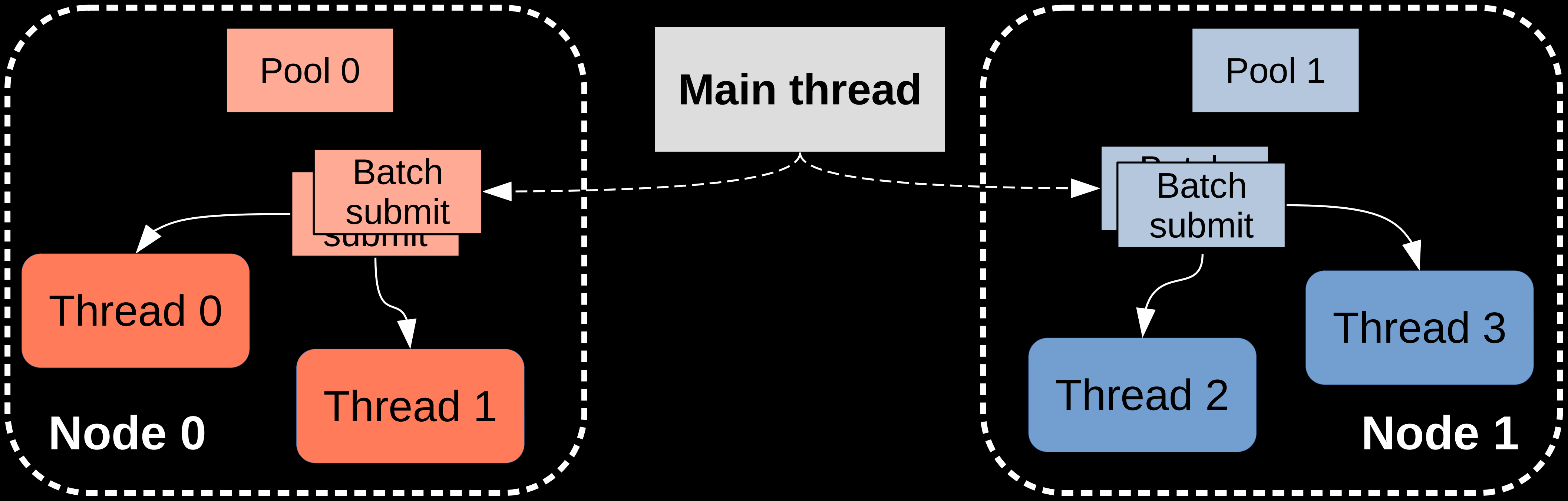
- Per-node pools, cross-node work stealing (only when one node is empty)
- Task submission from the main thread is the bottleneck
- Can be solved with a two-stage task submitter



SIEMENS

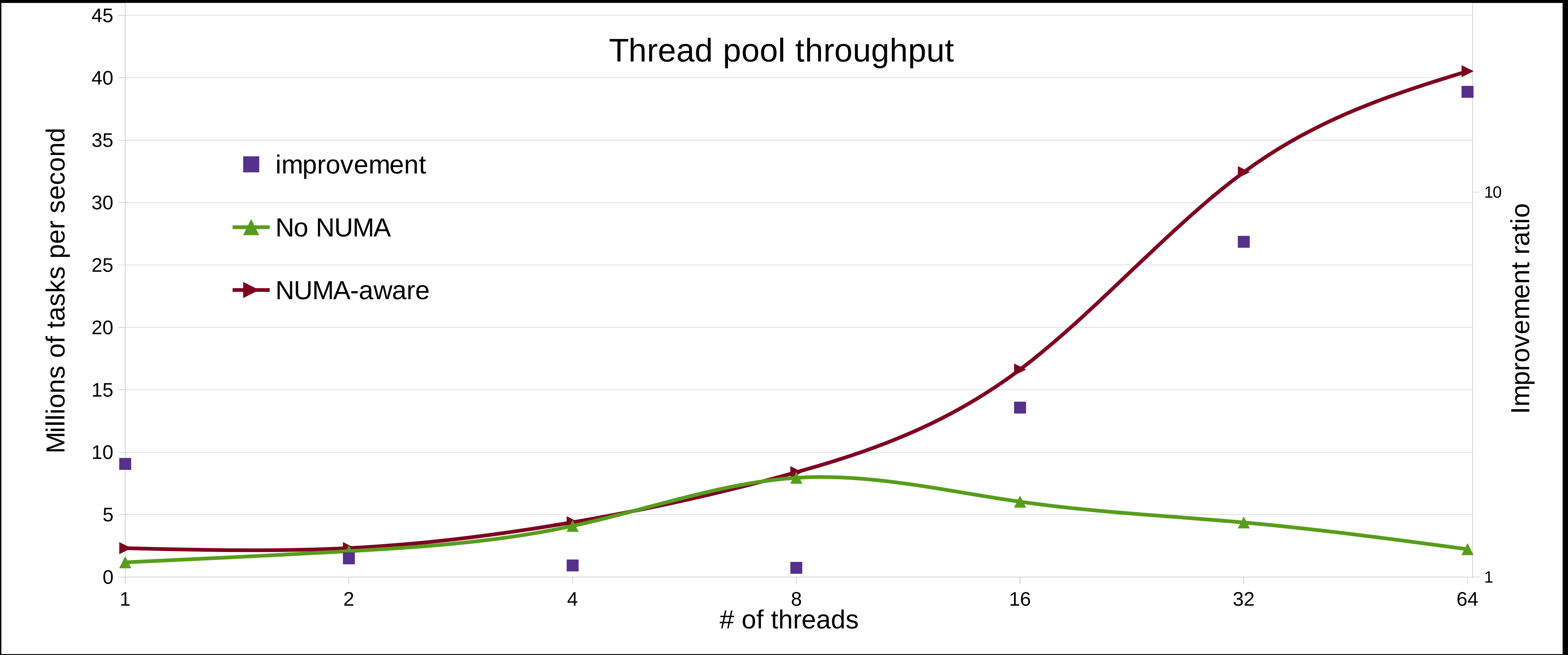
NUMA-aware thread pool

- Per-node pools, cross-node work stealing (only when one node is empty)
- Reduce cross-node data contention – fewer interacting threads or less frequent interaction



SIEMENS

NUMA-aware thread pool



SIEMENS

Performance implications of NUMA

- All modern large systems use NUMA
  - The latest CPUs are NUMA-in-a-package
- Accessing local memory is faster than accessing non-local memory
- Per-thead memory, data sharing, and cache locality are more important
- True data sharing can be significantly more expensive
  - Simple benchmarks with different node binding often point to problems
  - Hardware-assisted profiling is useful if you know what to look for
- Use good practices for memory access and node binding only if needed
- [Some] Concurrent data structures need to be redesigned for NUMA

SIEMENS

NUMA concerns for distributed applications

- Distributed systems often send a lot of traffic across the network
- Which NUMA node is the network controller connected to?

```
# /sbin/lspci
04:00.0 Ethernet controller: Intel Corporation 82599ES
# cat /sys/bus/pci/devices/0000\:04\:00.0/numa_node
0
```

- Does it matter? Sometimes.
- Case study: distributed application (1000s of remote CPUs)
  - Use numactl to bind to node 0 vs node 1 – 10% run time difference
  - Built-in throughput tester: 20% difference between nodes 0 and 1

NUMA concerns for GPUs and accelerators

- GPUs are a special case of I/O
  - GPU interfaces are usually faster than any other device
  - Data transfer is often the bottleneck of GPU acceleration
- CUDA Bandwidth test for Tesla V100: 10GB/s (node 0) vs 7GB/s (node 1)
- BUT…
- If GPU-CPU transfer is a concern, pinned memory should be used
- Pinned memory is on the GPU node
  - At least by default?
  - Transfer rate 13GB/s

SIEMENS

NUMA concerns for I/O-bound applications

- High-speed PCI-E devices are attached to a specific NUMA node
- NUMA may be important if the device is very fast
  - Simple test: bind the program to node 0 or node 1 and compare
    - Comparing with unbound run mostly tests the OS scheduler
- There may be other downsides that fall into Part III
- For I/O-bound programs, use the right node for the data

SIEMENS

Part III

- Of course the CPU does that!
- Makes sense that the CPU would do that…
- The CPU does WHAT?!

SIEMENS

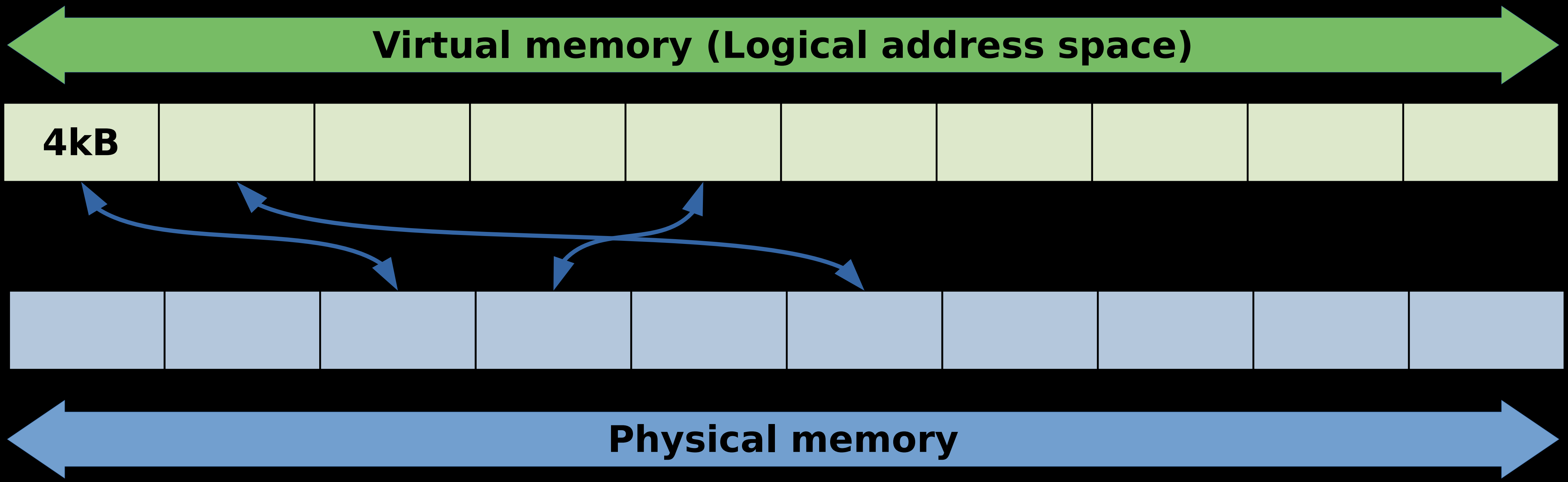
Real-life case

- Program runs slower on (faster) NUMA hardware than on old machines
- The slowdown is limited to certain places in the code
- For these functions, slowdown is large (2x to 10x)
- The code is not limited by bandwidth or latency but there is one common trait: they touch large amounts of memory
  - Like updating links in a list of large elements
- The slowdown is OS-dependent:
  - Not present on old versions of Linux

Debugging of the slowdown

- Debug setup: modify the real program to run the problem step repeatedly or on larger data volume
- Detailed time reporting: mostly system CPU time, not user time
- Profiling shows that the time is spent in the Linux kernel
- Profiling with kernel symbols shows that the time is spent in two sections:
  - NUMA management (memory migration, etc)
  - TLB updates
- What is TLB?

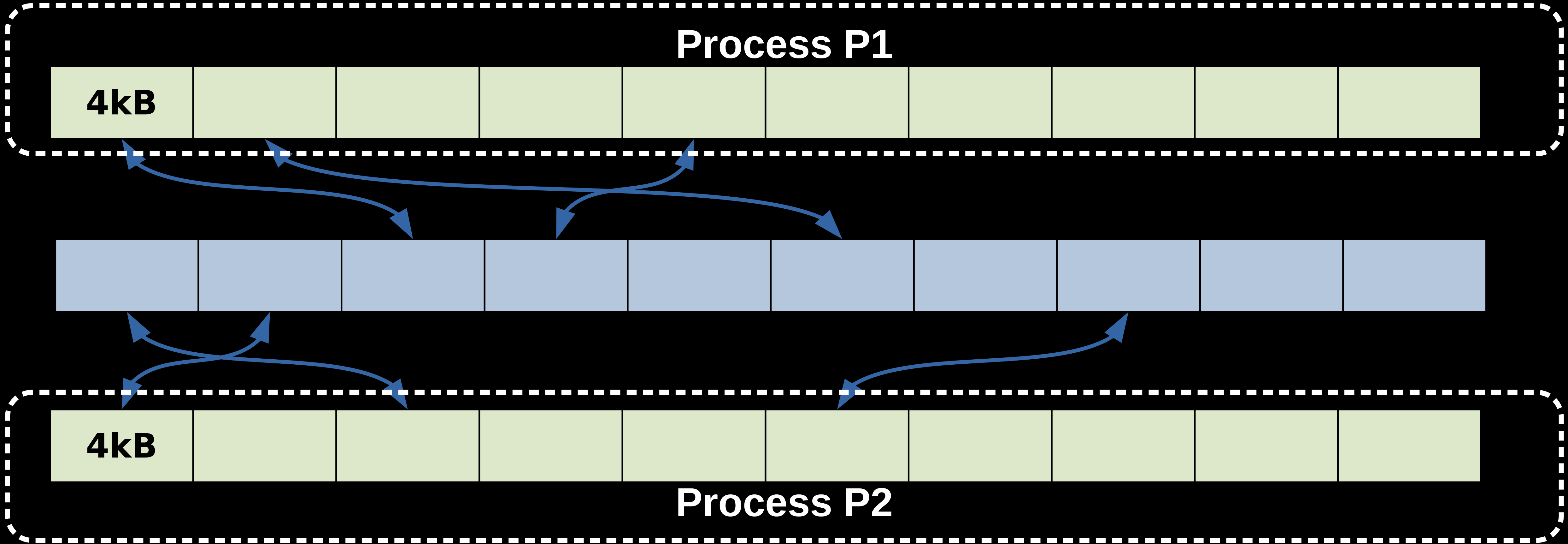
Virtual memory management



- Only pages that are used are mapped to physical pages
- Physical addresses are arbitrary and not visible to the program

SIEMENS

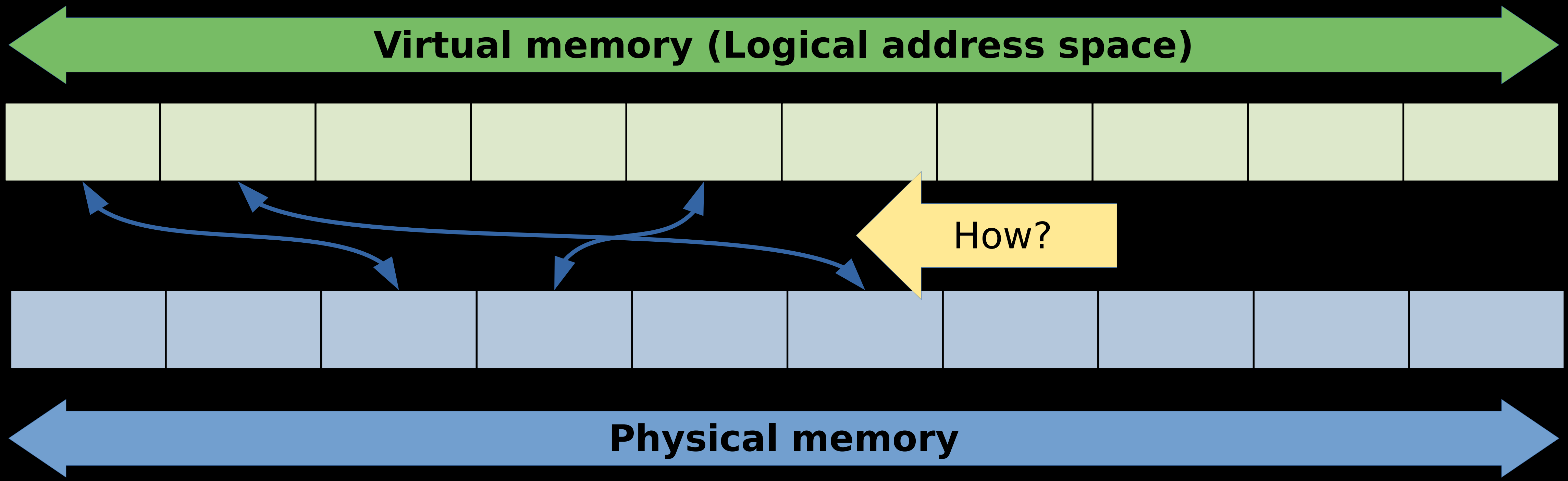
Virtual memory management



- Each process has its own virtual address space
- All virtual addresses are mapped to physical memory

SIEMENS

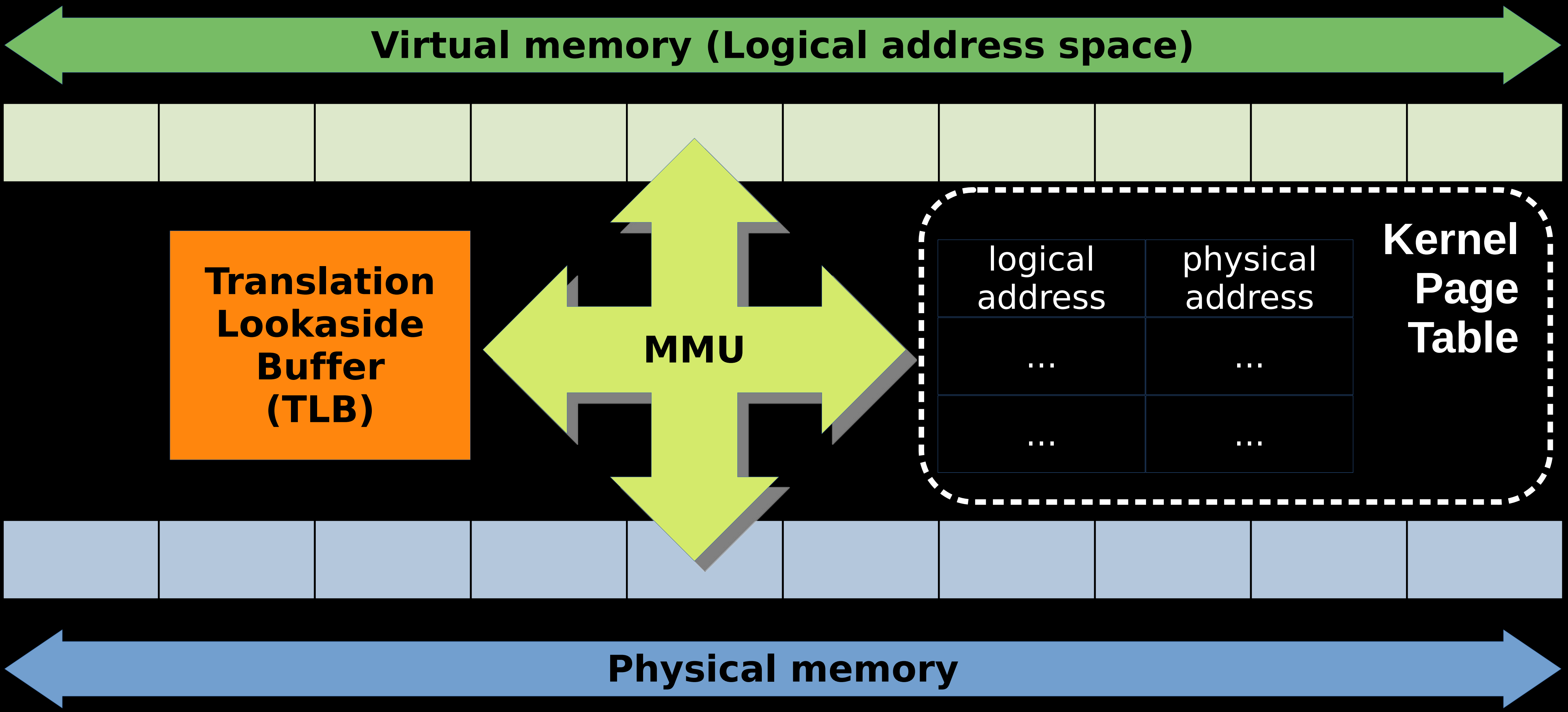
Virtual memory management



- Mapping of logical to physical addresses is handled by the Memory Management Unit (MMU, part of the processor)
- MMU looks up the address map in the kernel page table

SIEMENS

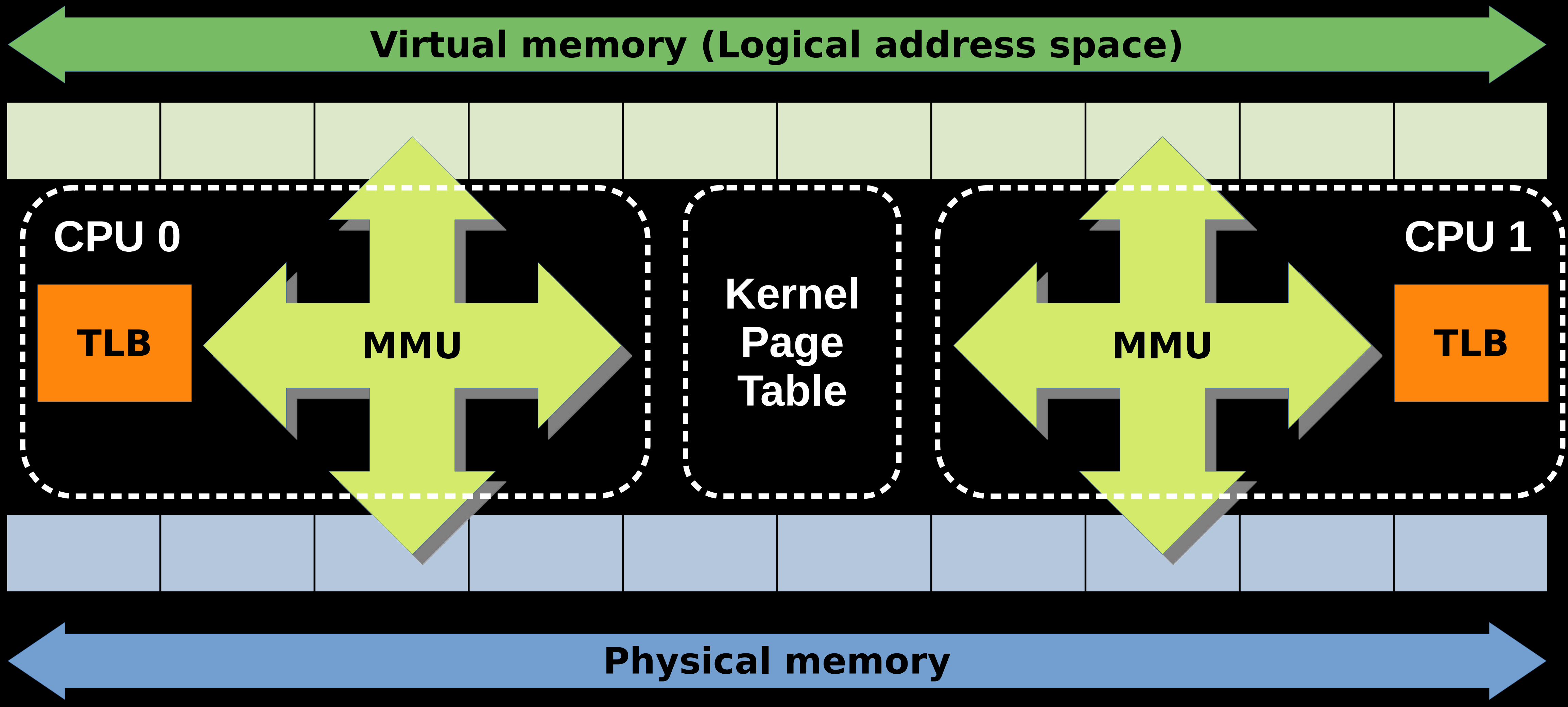
Virtual memory management



Virtual memory management

- The OS maintains the complete map of logical to physical addresses for all processes in the page table (per 4K page)
- The CPU has a hardware cache for recently used addresses
  - Translation lookaside buffer (TLB)
  - [Set-]Associative cache, similar to the regular CPU caches
    - Also known as Content-Addressable Memory
- If the address is found in the TLB (TLB hit), the MMU uses it
- If the address is not found in the TLB (TLB miss), the kernel has to walk the page table to find the map
- If the address is not found in the page table, page fault is triggered

Virtual memory management in NUMA systems

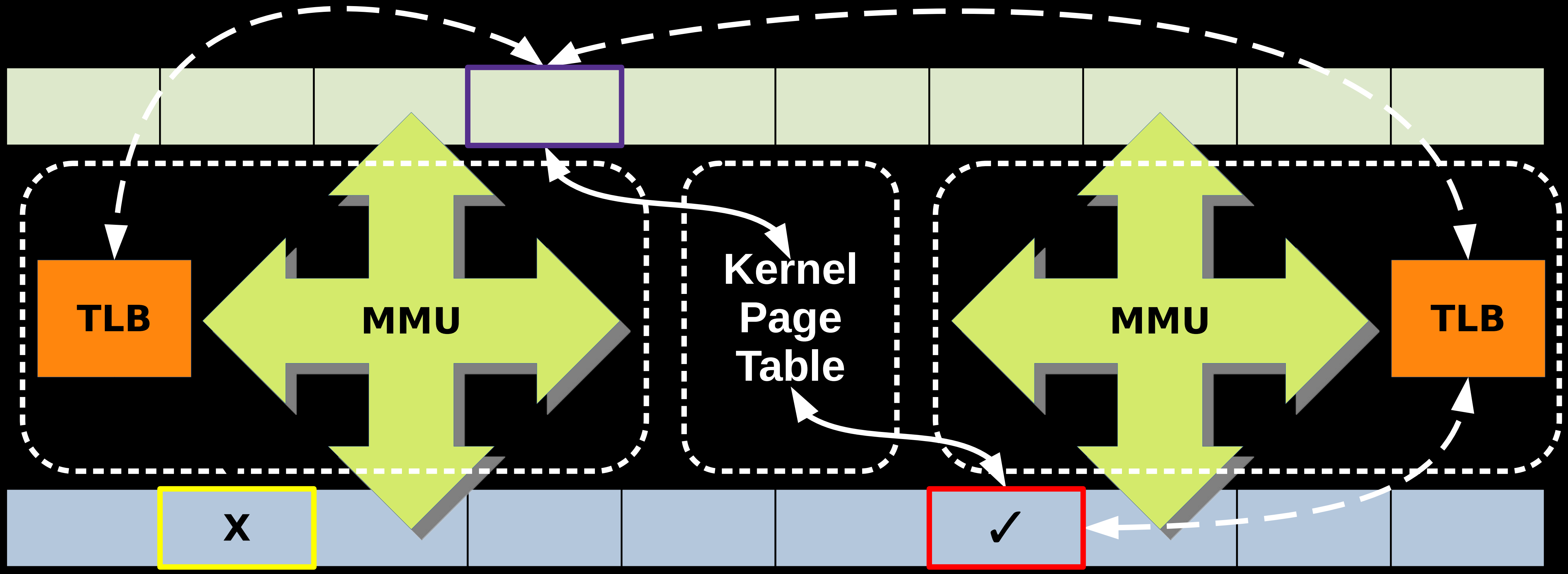


SIEMENS

Virtual memory management in NUMA systems

- Logical address space is per process
  - Spans all NUMA nodes
- Physical address space is global
  - Each address belongs to a particular node but accessible by any CPU
- Logical to physical address map (page table) is global
- MMU and TLB are CPU circuits, generally one per node
  - Some architectures have several TLBs in one CPU – not important now
- TLB is similar to other caches: local cache of the global state
- Unlike CPU caches, there is no hardware TLB coherency
- TLB content can differ between CPUs

Virtual memory management in NUMA systems



- If TLB content becomes outdated, the kernel must update it
  - There is no hardware assist for TLB updates

Virtual memory management in NUMA systems

- TLB on different processors can get out of date
  - Unlike caches, there is no hardware TLB coherency
- The kernel updates the TLB using a “TLB shootdown”
  - *“Translation Lookaside Buffer Consistency: A Software Approach,” David L. Black, R. Rashid, D. Golub, C. R. Hill, R. Baron, ASPLOS 1989*
  - Update may affect just one address or many (possibly all)
  - TLB shootdown is an inter-processor interrupt
  - Kernel code runs on the CPU when a shootdown occurs
- TLB shootdown handling can be seen in the profiler (kernel calls) and is accounted as “system time”
- Why do shootdowns happen when that particular code is running?

TLB shootdowns and their causes

- A TLB shootdown is processed by the kernel and interrupts the CPU
- TLB shootdown happens when a TLB is out of date, usually caused by
  - ~~Changes in memory mapping~~
  - ~~Memory access modes and protection~~
  - ~~Explicit changes to page table~~
  - NUMA migrations
- The profiler shows that the problem section spends its time in the kernel calls for TLB management and NUMA load balancing
- Migrating pages from one NUMA node to another changes the physical address and requires TLB updates

Debugging NUMA and TLB shootdowns

- TLB shootdowns cannot be disabled but the profiler can show the time spent in the kernel processing them
  - Recent Intel CPUs have hardware counts for TLB flushes
- NUMA migration can be disabled
  - Bind the program to a single NUMA node
  - Turn off automatic NUMA balancing

    ```
    echo 0 > /proc/sys/kernel/numa_balancing
    ```
- Disabling NUMA migrations “solves” the performance slowdown
  - The overall program generally becomes significantly slower
  - Auto balancing is a global state (affects everyone, needs root)

Reducing the impact of TLB shootdowns

- TLB shootdowns cannot be disabled
- NUMA balancing shouldn't be disabled
- Why this particular code?
  - Happens in code that allocates and deallocates a lot of memory rapidly
    - The solution is to pre-allocate and reuse memory – we already do that
  - Happens in multi-threaded code that quickly walks over large memory
    - Changing this behavior may be possible but may incur costs elsewhere
- Can we make TLB shootdowns cheaper?
  - It's mainly the cost of the page table walk (one entry per 4kB)
  - Does it have to be 4kB?

Page table optimization

- Page table problems are not new to NUMA
  - But can be worse in NUMA systems because of TLB inconsistency
- Often the solution is to enable "huge pages" – 2MB page
  - Must be enabled in the OS (usually on by default)

    ```
    # cat /sys/kernel/mm/transparent_hugepage/enabled
    always [madvise] never
    ```
  - ```
    echo always > /sys/kernel/mm/transparent_hugepage/enabled
    echo madvise > /sys/kernel/mm/transparent_hugepage/enabled
    ```
- "Madvise" means requested by the program
  - "Always" means the kernel can allocate huge pages as it sees fit
  - In practice, you cannot rely on that and must request huge pages
    ```

Using huge pages in your program

- Converting a virtual address range to huge pages:
  `madvise(address, size, MADV_HUGEPAGE);`
  - address and size must be 4kB-aligned (size must be n*2M to be useful)
- The memory must come from `mmap()` private anonymous map
  - `mmap()` is used to reserve large chunks of virtial address space
    - No physical memory is reserved immediately
    - Page table may be optimized for large unmapped regions
    - Physical pages are reserved as page faults are processed
- Using huge pages reduces the number of page table/TLB entries
- Switching to huge pages resolved the observed slowdown

Performance implications of NUMA

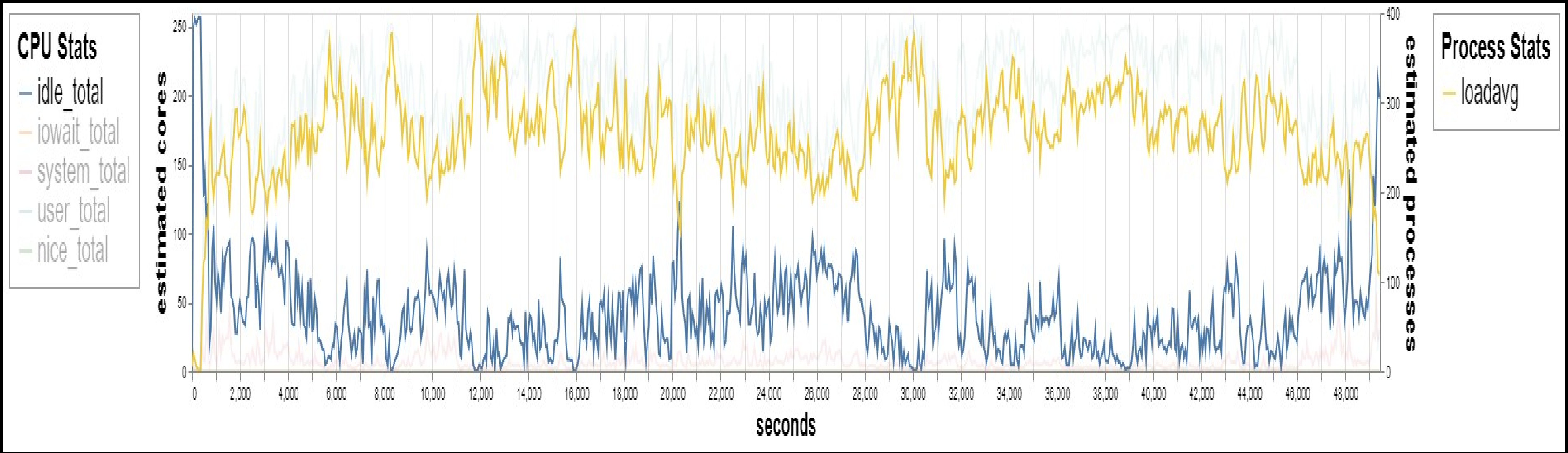
- What is NUMA?
- Accessing local memory is faster than accessing non-local memory
- Per-thead memory, data sharing, and cache locality are more important
- True data sharing can be significantly more expensive
- Simple benchmarks with different node binding often point to problems
- Hardware-assisted profiling is useful if you know what to look for
- Use good practices for memory access and node binding only if needed
- [Some] Concurrent data structures need to be redesigned for NUMA
- For I/O-bound programs, use the right node for the data
- There’s going to be something you don’t know about…

Real-life case

- Program runs slower on (faster) NUMA hardware than on old machines
- The slowdown is not associated with any specific code locations
- The slowdown manifests as low CPU utilization by the program
  - Embedded timers report ~70% of CPU time
  - OS-level measurements show significant idle time
- The slowdown is somewhat OS-dependent

Real-life case

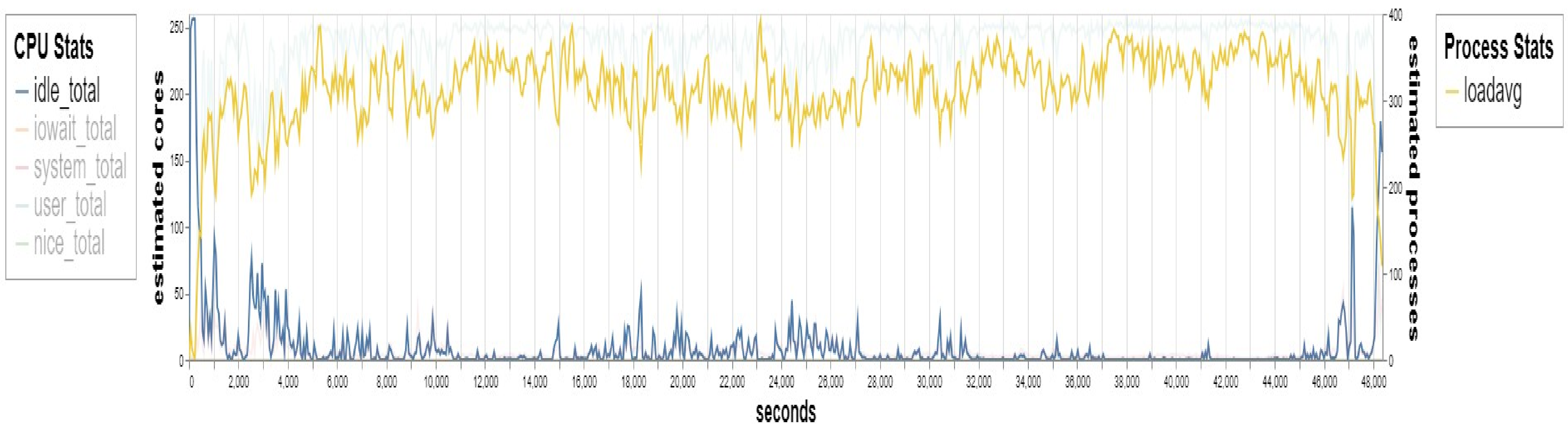
- The slowdown manifests as low CPU utilization by the program
  - Embedded timers report ~70% of CPU time
  - OS-level measurements show significant idle time



SIEMENS

Kernel tuning

- The CPU under-utilization is related to NUMA:
  - It does not happen on single-socket system with the same CPU
  - Not straightforward: EPYC processors are NUMA in one package
- Lowering kernel.sched_migration_cost_ns reduces the idle time



SIEMENS

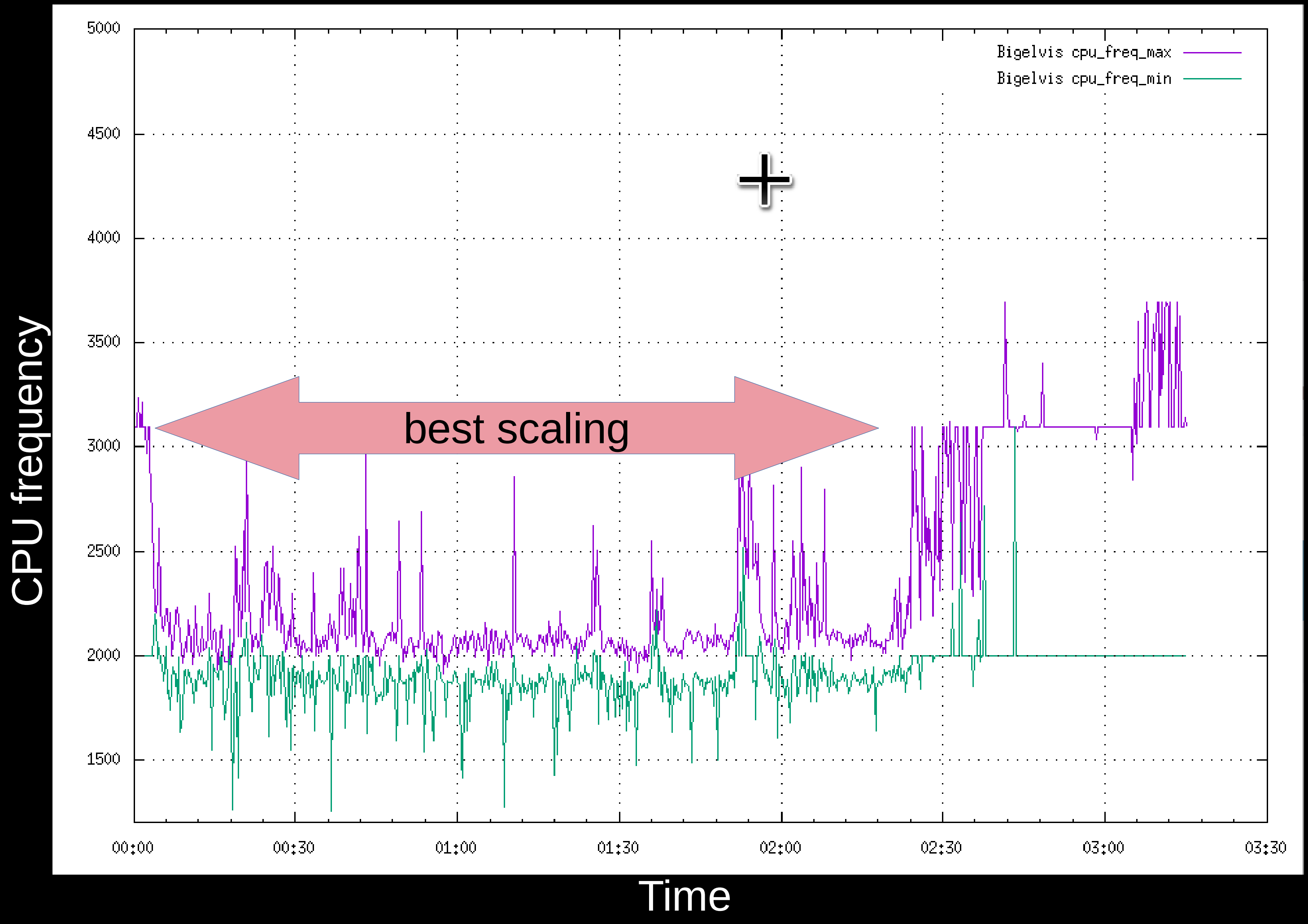
Kernel tuning

- The CPU under-utilization is related to NUMA:
  - It does not happen on single-socket system with the same CPU
  - Not straightforward: EPYC processors are NUMA in one package
- Lowering kernel.sched_migration_cost_ns reduces the idle time
  - Amount of time after the last execution that a task is considered to be “cache hot” in migration decisions – lower values allow easier migration
- Two factors favor easier migration of processes:
  - Process stay closer to the memory they access
  - Processes that need to access the network move to node 0
    - NIC is connected to PCIE lanes of node 0
  - Processes that are done with I/O move to the other node

Real-life case

- Program runs slower on (faster) NUMA hardware than on old machines
- The slowdown is not associated with any specific code locations but occurs in the most parallel (best scaling) sections
- CPU utilization is close to 100% all the time
- Some CPU models are strongly affected, others not at all

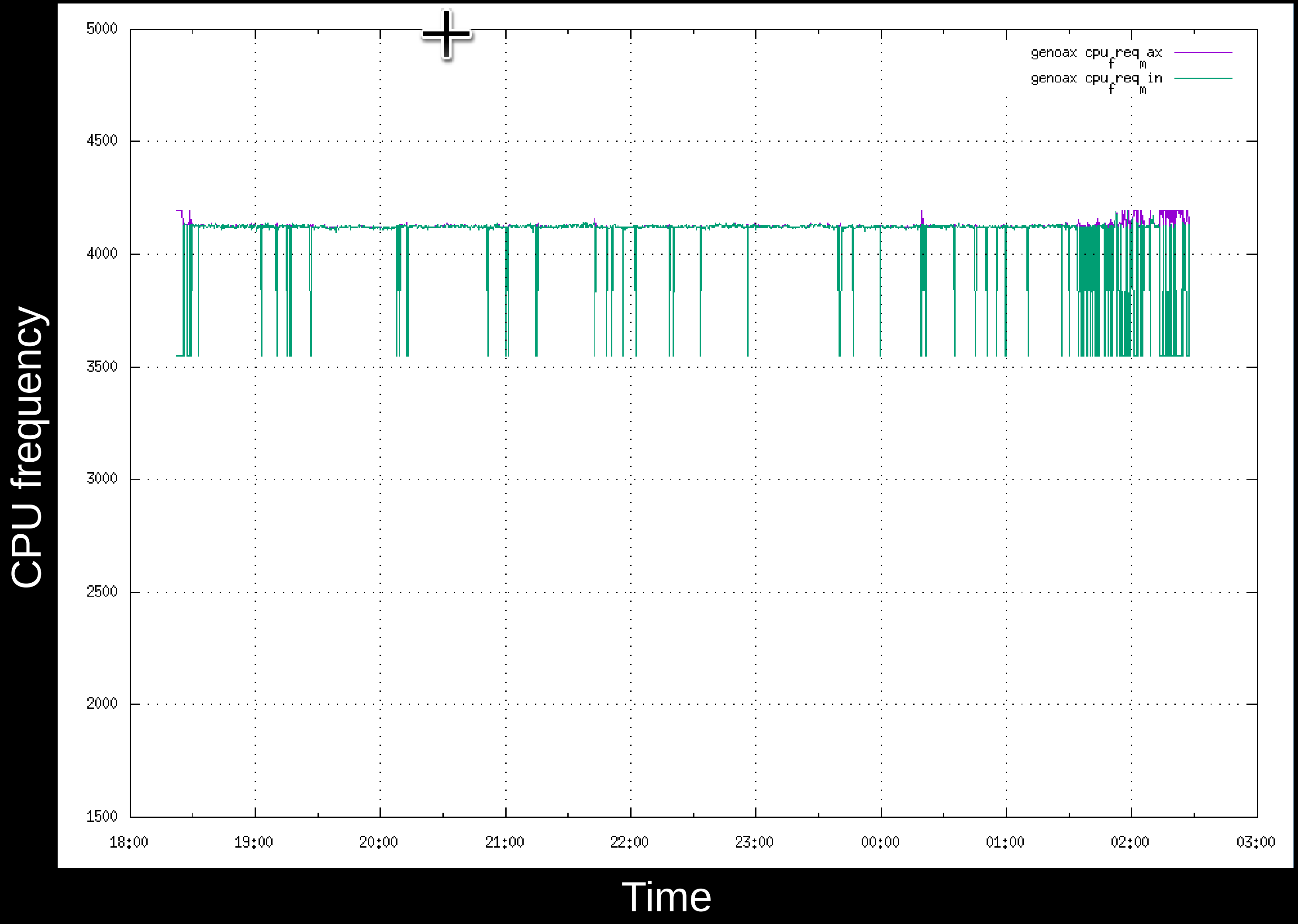
Something strange is going on



- Thorough monitoring helps
  - Collect more data
- CPU clock frequency drops when the load is highest
- 240W power is not sufficient to power all cores at max speed!

SIEMENS

Pay attention to CPU specs



- 320W CPU can power all cores consistently

SIEMENS

Another peculiarity of modern CPUs

- High-end X86 CPUs need about 400W to power everything constantly
  - Requires the program to use everything constantly
  - for some values of “everything”
- Lower-power variants are intended for low-latency applications with bursts of high load, not for sustained high throughput
- Under-powered CPUs are not useful for performance optimizations:
  - A more efficient program may trigger clock speed reduction
  - The CPU fights back against programmer’s attempt to optimize their code

How to write efficient code for modern CPUs?

- CPU’s like predictable execution patterns:
  - Memory access, branches, etc
- CPUs can do a many operations at once:
  - Superscalar architecture
  - SIMD and other special operations
- Predictable code is better at using CPU resources
- CPU resources can be traded for more predictable code
- Be aware of memory sharing effects when writing concurrent programs
- NUMA effects are complex and varied
  - Include memory allocations, process migration, and peripherals
- Selecting the right processor for the job is not as easy as it used to be

How to write efficient code for modern CPUs?

- You know what to worry about. What should you do?
- Profiling and monitoring are very important
  - Profiling helps to isolate problems, explain them, and find solutions
    - Especially profiling with hardware counters
  - Monitoring helps to discover problems
    - Also collects data that is valuable for analysis
- Know your hardware
  - Have a library of small benchmarks for hardware features that are relevant to your programs and might affect how you write the code

How to get the most out of modern CPUs?

Questions?

Possibly answers too...

SIEMENS