# CoE 163

Computing Architectures and Algorithms

11b: Parallel Programming and Hardware

# BOTTLENECKS

You may have noticed that parallelizing a program does not always yield a better result. Worse, it is even slower than the sequential program equivalent!

*Why is that?*

# BOTTLENECKS

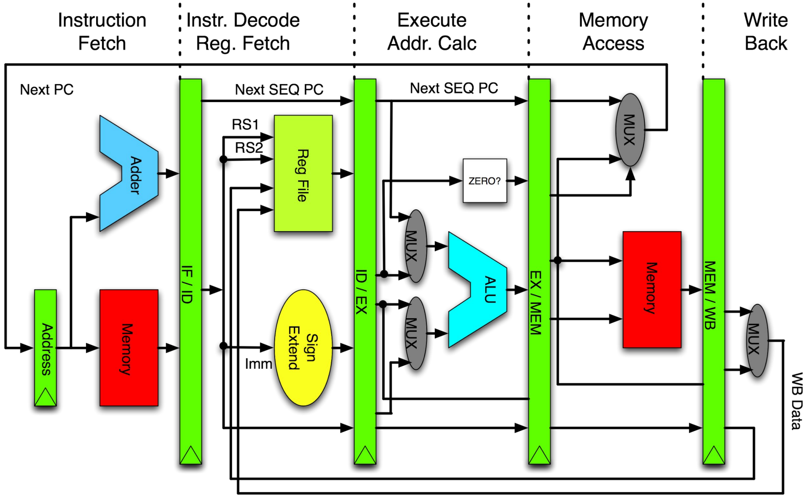
- Pipelined CPUs
- Memory references
- Atomic operations
- Memory barriers
- Cache misses
- I/O operations

# BOTTLENECKS: PIPELINED CPUS

- Modern CPUs have the capability to execute multiple parts of instructions in a single clock cycle - they have *instruction-level parallelism*
- Such CPUs use one or more of the following techniques to make the most out of each clock cycle:
  - Pipelining
  - Superscalar techniques
  - Out-of-order execution
  - Speculative execution
  - … and more!

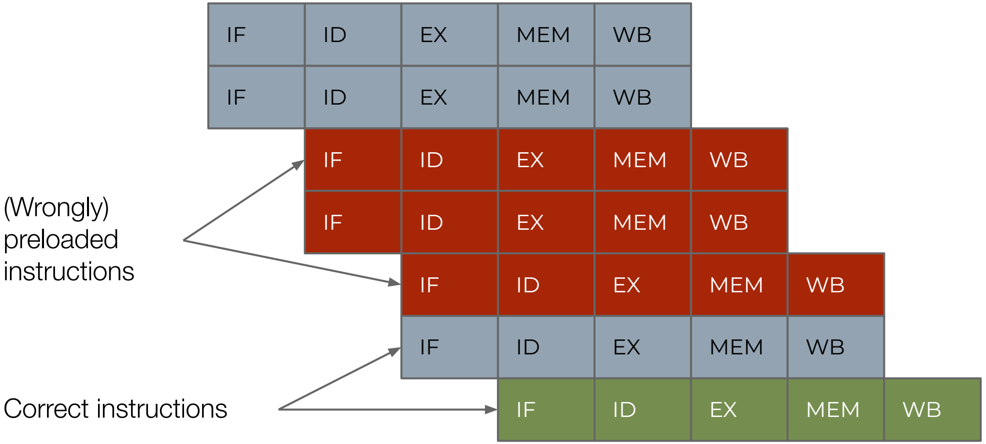
# BOTTLENECKS: PIPELINED CPUS

- Most pipelined CPUs will have an execution process similar to the one shown below - a five-stage pipeline.
- *Hazards* slow down the pipeline.



# BOTTLENECKS: PIPELINED CPUS

- A *superscalar* CPU can simultaneously process more than one instruction.
- Other techniques need to be implemented to prevent hazards, such as out-of-order execution and branch prediction.



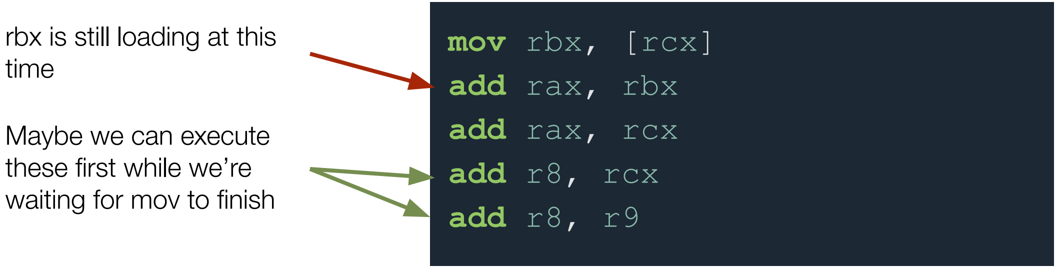
| | | | | | | | |
|---|---|---|---|---|---|---|---|
| IF | ID | EX | MEM | WB | | | |
| IF | ID | EX | MEM | WB | | | |
| | IF | ID | EX | MEM | WB | | |
| | IF | ID | EX | MEM | WB | | |
| | | IF | ID | EX | MEM | WB | |
| | | IF | ID | EX | MEM | WB | |
| | | | IF | ID | EX | MEM | WB |

(Wrongly) preloaded instructions

Correct instructions

# BOTTLENECKS: PIPELINED CPUS

- *Out-of-order execution* tries to maintain maximum usage of a CPU during hazard resolution by executing instructions independent of the instruction being resolved.
- What if we *need* the instructions to be executed in order?



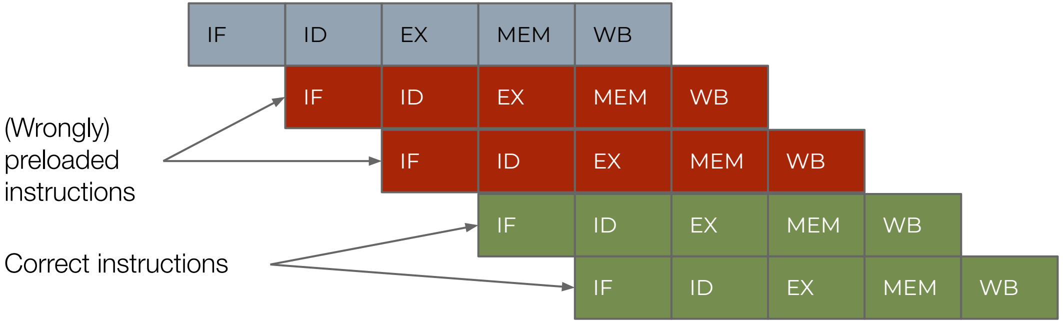
rbx is still loading at this time

Maybe we can execute these first while we're waiting for mov to finish

```
mov rbx, [rcx]
add rax, rbx
add rax, rcx
add r8, rcx
add r8, r9
```

# BOTTLENECKS: PIPELINED CPUS

- *Branch prediction* guesses whether a conditional instruction will be true and preloads the next instructions according to the predicted result.
- What if we guessed wrong? We have to *flush* the pipeline and load the correct set of instructions!



| | | | | | | | | |
|---|---|---|---|---|---|---|---|---|
| IF | ID | EX | MEM | WB | | | | |
| | IF | ID | EX | MEM | WB | | | |
| | | IF | ID | EX | MEM | WB | | |
| | | | IF | ID | EX | MEM | WB | |
| | | | | IF | ID | EX | MEM | WB |

(Wrongly) preloaded instructions

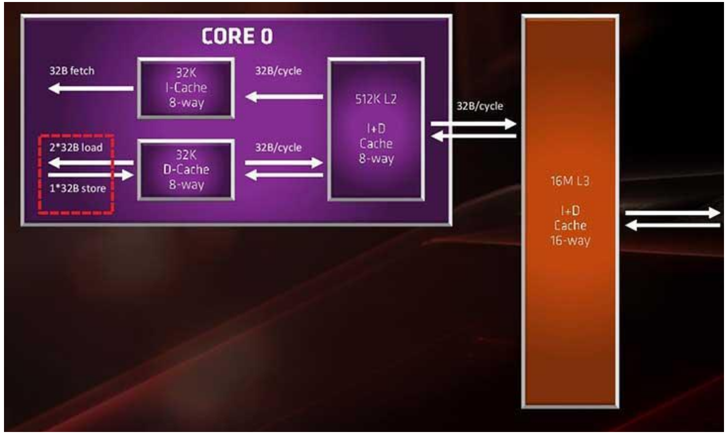
Correct instructions

# BOTTLENECKS: MEMORY REFERENCES

- Due to Moore’s Law, CPU speeds have dramatically increased over the past few decades.
- Unfortunately, the associated memory sizes and access speeds have not grown at the same speed.

# BOTTLENECKS: MEMORY REFERENCES

- Modern CPUs implement some sort of cache hierarchy.
- The lowest level closest to the CPU is fastest but has the smallest size.
- Random access (e.g. traversing a tree or linked list) leads to higher latency due to cache misses.



# BOTTLENECKS: ATOMIC OPERATIONS

- An *atomic operation* executes while barring any other process from reading or changing any data or state involved in the operation.
- It conflicts with the assembly line nature of pipelined CPUs because the instruction and data need to be held solely by the CPU.
- We may need to stall or flush the pipeline to execute the operation properly!

# BOTTLENECKS: ATOMIC OPERATIONS

- In some languages, we can force atomic operations by using some sort of *synchronization* to force programs to execute in order and one at a time.
- A code block or lock can be used to perform such synchronization.

```
// Java
synchronized(mythread) {
    a += 1;
    b -= 1;
}
```

# BOTTLENECKS: MEMORY BARRIERS

- A *lock* is a software mechanism that limits access to data between the time it is held or used.
- Such locks prevent out-of-order execution.

# BOTTLENECKS: MEMORY BARRIERS

- A *mutual exclusion (mutex) lock* can be used similar to the synchronized blocks in other languages.
- A *counting semaphore* can be used if it is acceptable for more than one process or thread to have access to the same resource.

```
// C++
void fun(int d) {
    mux_lock.lock();

    z += d;

    mux_lock.unlock();
}
```

# BOTTLENECKS: CACHE MISSES

- A *cache* holds frequently-accessed data for faster retrieval later.
- Cache accesses are fast, but if the data is not in the cache, the CPU needs to access the slower memory or data store, and then load the data into the cache.
- Frequent cache misses slow down execution time.

# BOTTLENECKS: I/O OPERATIONS

- Speed of communication of CPU to and from outside sources is highly variable.
- This includes network, storage, and human resources.
- A *distributed parallel program* requires such I/O access, which can drastically slow down its execution time compared to ones with shared memory.

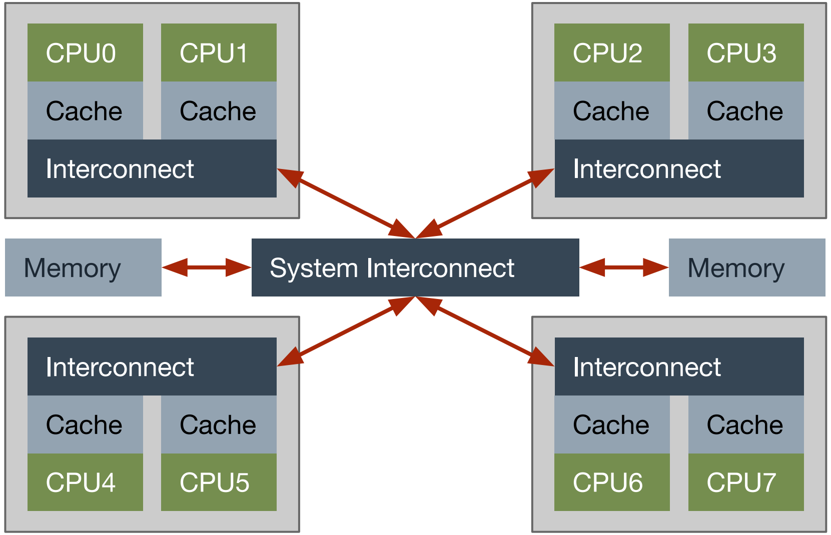
# OVERHEADS

The previous bottlenecks generate overheads in computer programs - not just parallel ones.

The CPU will have to go through all of these hurdles just to execute *one* instruction!

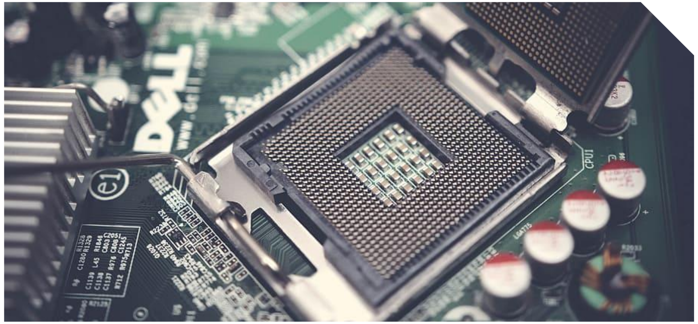
# OVERHEADS: SYSTEM HARDWARE ARCHITECTURE

- Shown below is an oversimplified diagram of an eight-core CPU
- Each core is grouped into two in a die, which can communicate with each other and to other cores via either a “local” or “global” interconnect



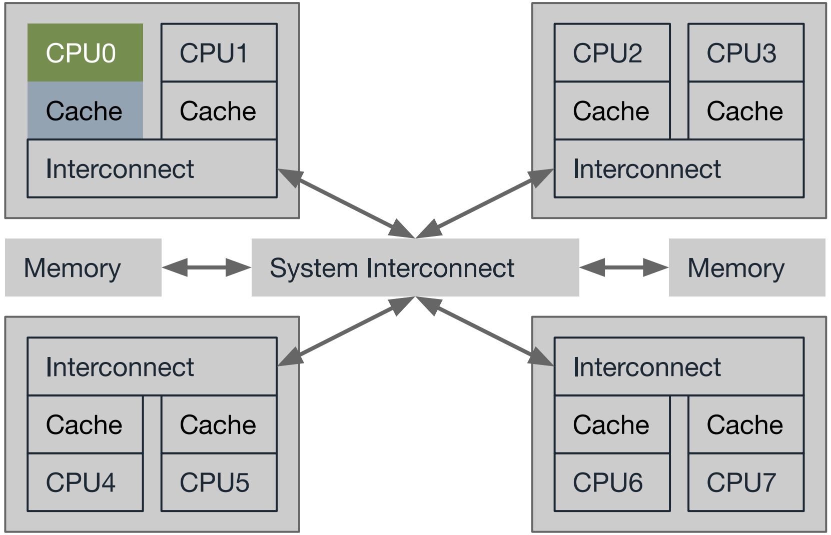
# CONSIDER...

CPU0 wants to write to a variable whose cache line is, apparently, stored in CPU7!



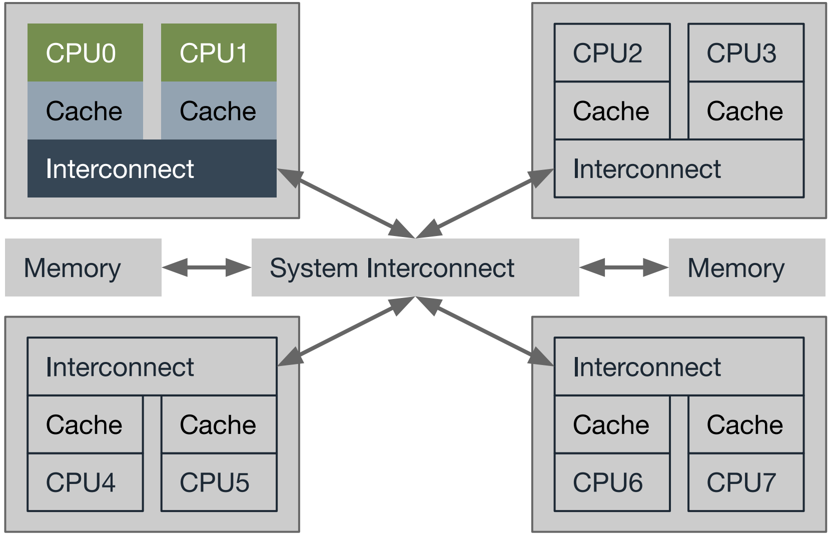
# OVERHEADS: VARIABLE WRITING

- CPU0 checks its cache and does not find the variable, so it sends a request to its local interconnect.



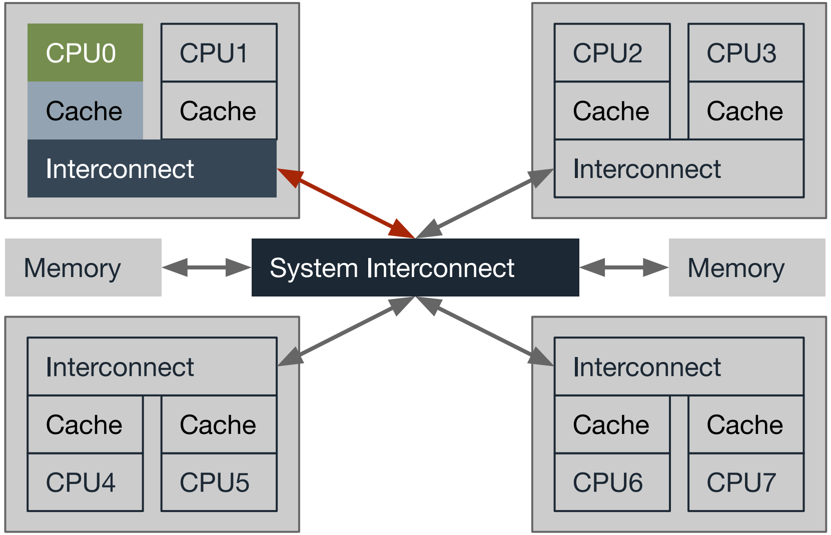
# OVERHEADS: VARIABLE WRITING

- The local interconnect checks CPU1 to find out that the variable is not there!
- The local interconnect sends the request to the global one.



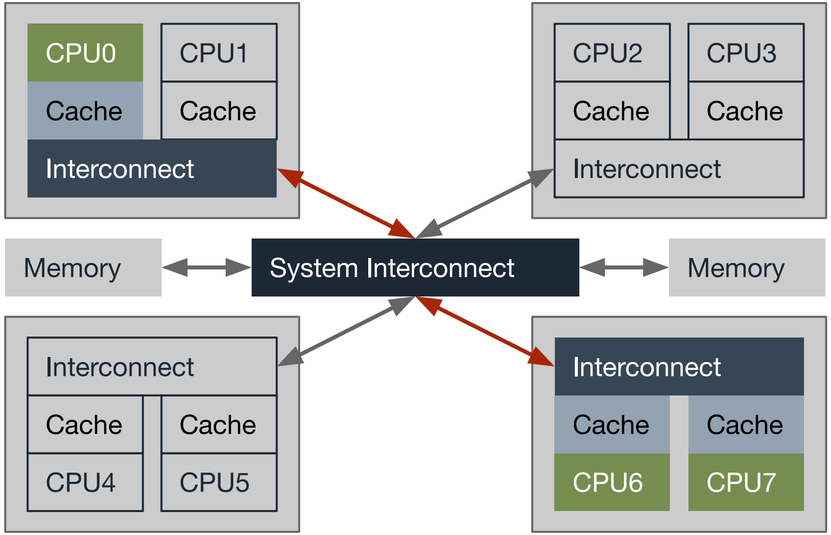
# OVERHEADS: VARIABLE WRITING

- The global interconnect finds out that the target variable is stored in the die containing CPU6 and CPU7.
- The global interconnect sends a request to the local one.



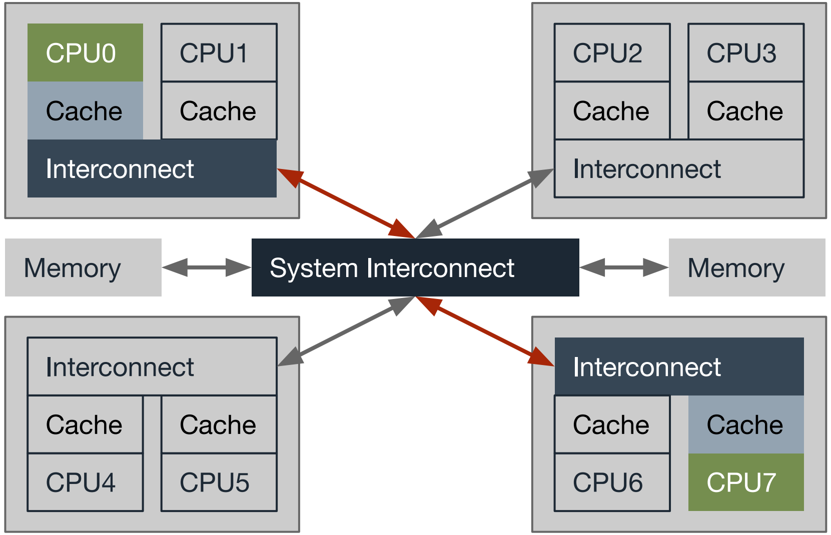
# OVERHEADS: VARIABLE WRITING

- The local interconnect checks both CPUs to find out that the variable is stored in CPU7.



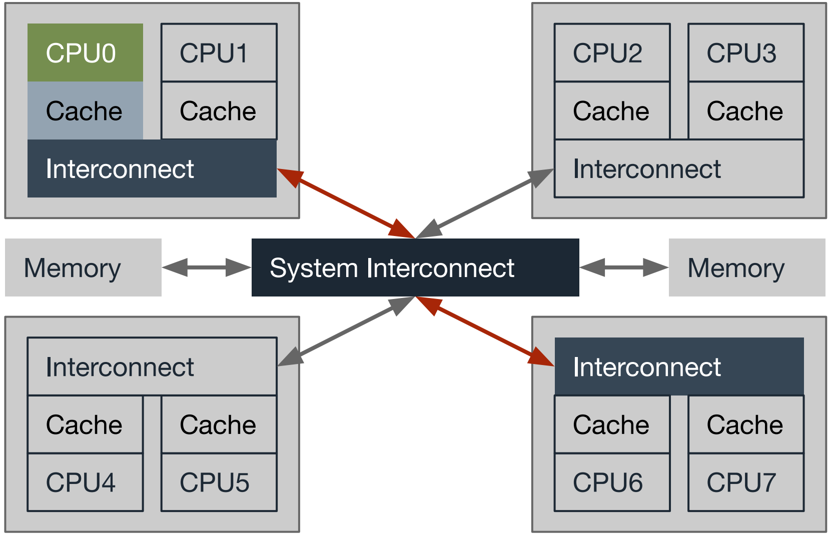
# OVERHEADS: VARIABLE WRITING

- CPU7 returns the relevant cache line and flushes it from its own cache.
- The returned cache line is sent to the local interconnect.



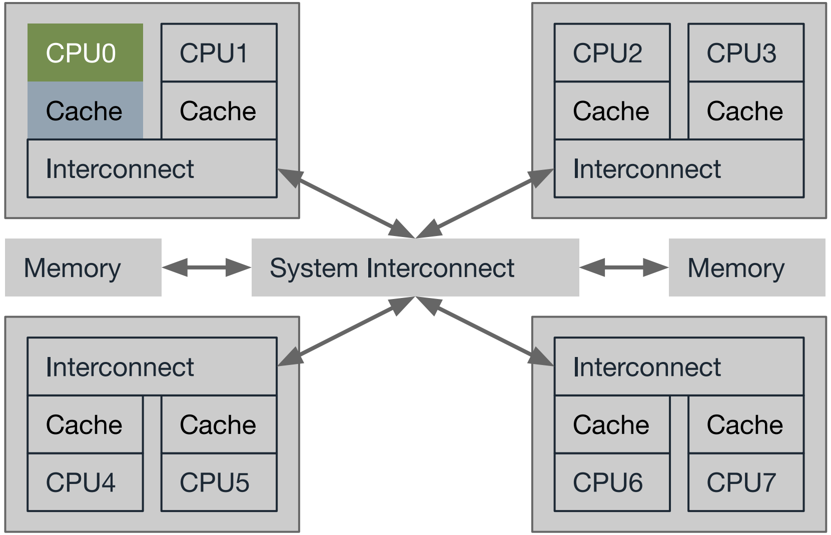
# OVERHEADS: VARIABLE WRITING

- The local interconnect sends the cache line to the global one.
- The global interconnect sends the cache line to the die containing CPU0.
- The local interconnect sends the cache line to the cache of CPU0.



# OVERHEADS: VARIABLE WRITING

- The cache line gets received and CPU0 can now write to the variable.
- The new value is then stored in its own cache.



# OVERHEADS:
# VARIABLE WHAT-IFS

- What if the variable does not exist in *any* of the caches?
- What if the cache line in CPU7 has already expired once the request arrives?
- What if CPU2 wanted to write to the same variable CPU0 wants to write to?
- What if there are *read-only* copies of the same cache line across the other CPUs?

# OVERHEADS: CACHE COHERENCY

Making sure that the cache of *all* of the cores reflect the truest state of a variable is a subject of *cache-coherency protocols*.

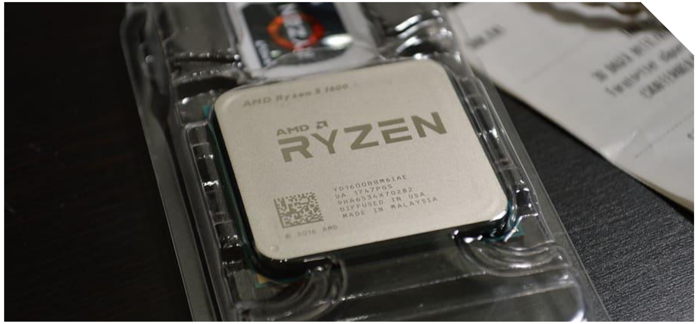
If the variable is only being read *most of the time*, then access will be very fast.

# OVERHEADS: OPERATION COST

We can get a feel for the effect of these overheads by profiling it against a CPU operation.

# CONSIDER...

A **compare-and-swap** (CAS) operation compares two registers and replaces the destination register with some value if their contents are equal. In x86_64, it is the `cmpxchg` instruction. We will make it atomic by using the `lock` prefix.



# OVERHEADS: OPERATION COST

- “Same CPU” CAS means that the current CPU has most recently accessed the variables used in the operation.
- “In-core” CAS means that there are hardware threads sharing a single core.

Fast → Slow

| | | | | | | | | |
|---|---|---|---|---|---|---|---|---|
| | CAS | CAS | Lock | CAS | CAS | CAS | CAS | CAS |
| | Same CPU | In Core | Same CPU | In Core | Off Core | Off Core | Off Socket | Off Socket |
| | | Blind | | | Blind | | Blind | |
| **Speed Ratio** | 14 | 15 | 32 | 38 | 100 | 214 | 312 | 930 |

* Speed ratio computed as execution time per clock period. Ratios obtained and estimated from a group of Intel Xeon Platinum 8176 processors.

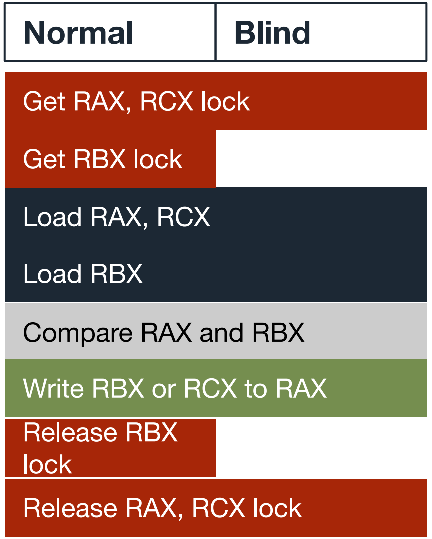
# OVERHEADS: OPERATION COST

In a *blind* CAS operation, the two registers to compare are readily available via software. Only one lock is required.

On the other hand, in a normal CAS operation, the other register where the "old" value is stored will have to be loaded from memory. This needs two locks!

```
lock

cmpxchg rbx, rcx
```



**Normal** | **Blind**

Get RAX, RCX lock

Get RBX lock

Load RAX, RCX

Load RBX

Compare RAX and RBX

Write RBX or RCX to RAX

Release RBX lock

Release RAX, RCX lock

# OPTIMIZATIONS

### Large cache lines

Enables fetching of a large chunk of the cache

Prone to *false sharing* resulting to high cache miss rate.

### Cache prefetching

Fetch subsequent cache lines during a fetch

Efficient only if the hardware is able to at least know when to prefetch

### Store buffers

Buffer to pre-empt writing during cache miss or when writes are on non-consecutive addresses

Prone to *memory misordering*.

# OPTIMIZATIONS

### Speculative execution

May lead to inefficiency if speculation goes wrong all the time, and is even prone to security attacks*!

### Large caches

Store a lot on the caches at the expense of higher latency during cache misses.

### Read-mostly replication

Store read-mostly data across *all* caches at the expense of a higher latency when the rare write operation happens.

* See Spectre and Meltdown vulnerabilities from 2018

# PARALLEL SOFTWARE DESIGN

Given all of the CPU bottlenecks and overheads, the most ideal way to formulate parallel algorithms and programs is to make sure that they are **embarrassingly parallel**.

# PARALLEL SOFTWARE DESIGN: OPTIMIZATIONS

**Independent threads**

Reduce communication between threads to reduce synchronization measures

**Read-mostly data sharing**

Take advantage of this CPU functionality for maximum speed

**Embarrassingly parallelizable**

Aside from reducing communication, threads should be able to run by themselves without any external intervention

# TIPS

- Consider the CPU architecture to gauge whether parallelizing a program is worth it
- Strive to make embarrassingly parallel programs and algorithms
- Practice decomposing problems into units

# RESOURCES

- Perfbook from the Linux Kernel Archives
- Blog post on a flaw on speculative execution from Google Project Zero

# CoE 163

## Computing Architectures and Algorithms

11b: Parallel Programming and Hardware