

# Design of Parallel and High-Performance Computing

Fall 2014

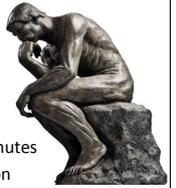
Lecture: Cache Coherence & Memory Models

Instructor: Torsten Hoefler & Markus Püschel

TAs: Timo Schneider, Arnamoy Bhattacharyya

**ETH**

Eidgenössische Technische Hochschule Zürich  
Swiss Federal Institute of Technology Zurich

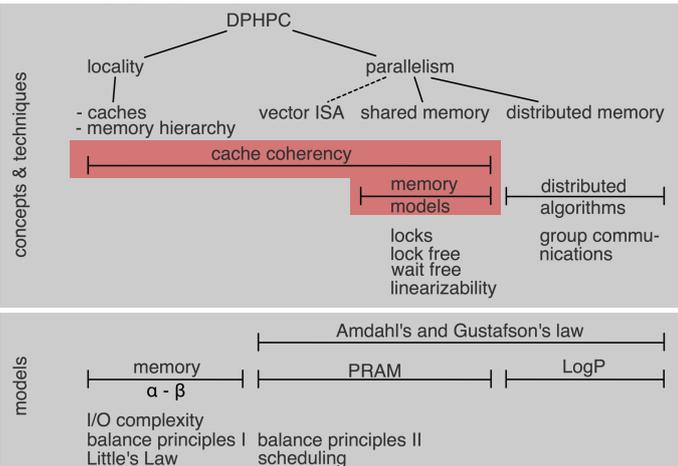


## Peer Quiz

- **Instructions:**
  - Pick some partners (locally) and discuss each question for 3 minutes
  - We then select a random student (team) to answer the question
- **What is the top500 list? Discuss its usefulness (pro/con)!**
  - What should we change?
- **What is the main limitation in single-core scaling today?**
  - i.e., why do cores not become much faster?
  - What will be the next big problem/limit?
- **What is the difference between UMA and NUMA architectures?**
  - Discuss which architecture is more scalable!
- **Describe the difference between shared memory, partitioned global address space, and distributed memory programming**
  - Name at least one practical example programming system for each
  - Why do all of these models co-exist?

2

## DPHPC Overview



3

## Goals of this lecture

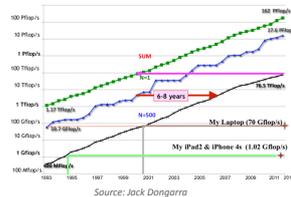
- **Memory Trends**
- **Cache Coherence**
- **Memory Consistency**

4

## Memory – CPU gap widens

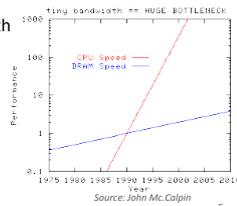
### ■ Measure processor speed as “throughput”

- FLOPS/s, IOPS/s, ...
- Moore's law - ~60% growth per year



### ■ Today's architectures

- POWER7: 256 GFLOPS/s – 128 GB/s memory bandwidth
- BG/Q: 205 GFLOPS/s – 42.6 GB/s memory bandwidth
- Trend: memory performance grows 10% per year



5

## Issues

### ■ How to measure bandwidth?

- Data sheet (often peak performance, may include overheads)
  - Frequency times bus width: 51 GiB/s
- Microbenchmark performance
  - Stride 1 access (32 MiB): 32 GiB/s
  - Random access (8 B out of 32 MiB): 241 MiB/s
  - Why?
- Application performance
  - As observed (performance counters)
  - Somewhere in between stride 1 and random access

### ■ How to measure Latency?

- Data sheet (often optimistic, or not provided)
  - <100ns
- Random pointer chase
  - 110 ns with one core, 258 ns with 32 cores!

6

## Conjecture: Buffering is a must!

- **Write Buffers**
  - Delayed write back saves memory bandwidth
  - Data is often overwritten or re-read
- **Caching**
  - Directory of recently used locations
  - Stored as blocks (cache lines)

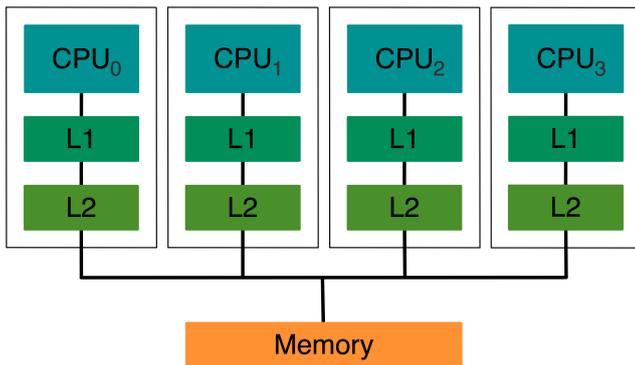
7

## Cache Coherence

- **Different caches may have a copy of the same memory location!**
- **Cache coherence**
  - Manages existence of multiple copies
- **Cache architectures**
  - Multi level caches
  - Multi-port vs. single port
  - Shared vs. private (partitioned)
  - Inclusive vs. exclusive
  - Write back vs. write through
  - Victim cache to reduce conflict misses
  - ...

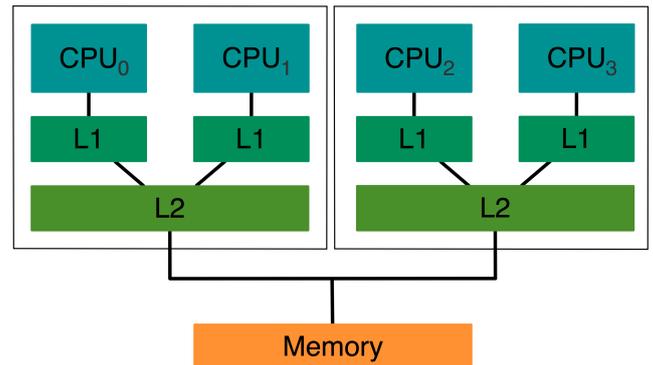
8

## Exclusive Hierarchical Caches



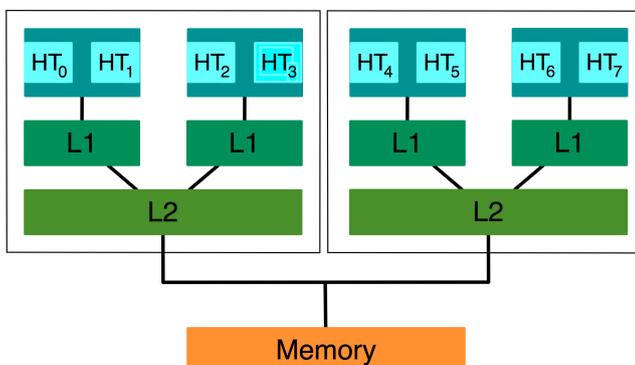
9

## Shared Hierarchical Caches



10

## Shared Hierarchical Caches with MT



11

## Caching Strategies (repeat)

- **Remember:**
  - Write Back?
  - Write Through?
- **Cache coherence requirements**

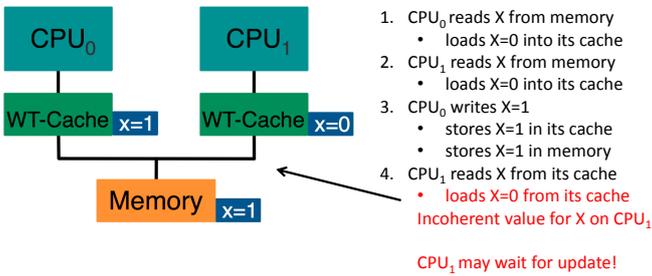
A memory system is coherent if it guarantees the following:

  - Write propagation (updates are eventually visible to all readers)
  - Write serialization (writes to the same location must be observed in order)

*Everything else: memory model issues (later)*

12

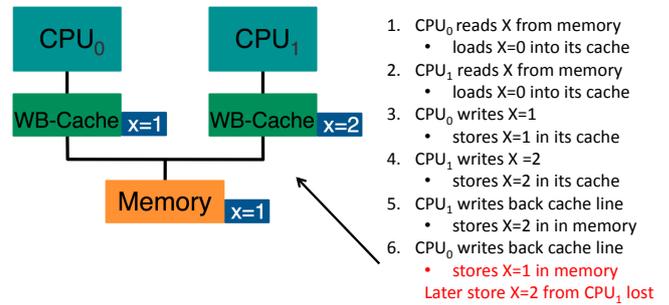
## Write Through Cache



Requires write propagation!

13

## Write Back Cache



Requires write serialization!

14

## A simple (?) example

### Assume C99:

```
struct twoint {
    int a;
    int b;
}
```

### Two threads:

- Initially: a=b=0
- Thread 0: write 1 to a
- Thread 1: write 1 to b

### Assume non-coherent write back cache

- What may end up in main memory?

15

## Cache Coherence Protocol

### Programmer can hardly deal with unpredictable behavior!

- Cache controller maintains data integrity
  - All writes to different locations are visible

### Fundamental Mechanisms

- Snooping**
  - Shared bus or (broadcast) network
  - Cache controller "snoops" all transactions
  - Monitors and changes the state of the cache's data
- Directory-based**
  - Record information necessary to maintain coherence
  - E.g., owner and state of a line etc.

16

## Cache Coherence Parameters

### Concerns/Goals

- Performance
- Implementation cost (chip space)
- Correctness
- (Memory model side effects)

### Issues

- Detection (when does a controller need to act)
- Enforcement (how does a controller guarantee coherence)
- Precision of block sharing (per block, per sub-block?)
- Block size (cache line size?)

17

## An Engineering Approach: Empirical start

### Problem 1: stale reads

- Cache 1 holds value that was already modified in cache 2
- Solution:
  - Disallow this state
  - Invalidate all remote copies before allowing a write to complete

### Problem 2: lost update

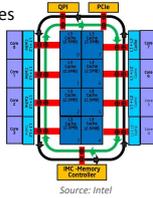
- Incorrect write back of modified line writes main memory in different order from the order of the write operations or overwrites neighboring data
- Solution:
  - Disallow more than one modified copy

18

## Cache Coherence Approaches

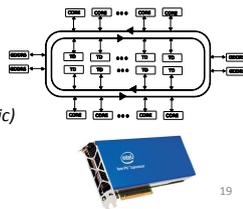
### Based on invalidation

- Broadcast all coherency traffic (writes to shared lines) to all caches
- Each cache snoops
  - Invalidate lines written by other CPUs
  - Signal sharing for cache lines in local cache to other caches
- Simple implementation for bus-based systems
- Works at small scale, challenging at large-scale
  - E.g., Intel Sandy Bridge



### Based on explicit updates

- Central directory for cache line ownership
- Local write updates copies in remote caches
  - Can update all CPUs at once
  - Multiple writes cause multiple updates (more traffic)
- Scalable but more complex/expensive
  - E.g., Intel Xeon Phi



19

## Invalidation vs. update

### Invalidation-based:

- Only write misses hit the bus (works with write-back caches)
- Subsequent writes to the same cache line are local
- Good for multiple writes to the same line (in the same cache)

### Update-based:

- All sharers continue to hit cache line after one core writes
  - Implicit assumption: shared lines are accessed often
- Supports producer-consumer pattern well
- Many (local) writes may waste bandwidth!

### Hybrid forms are possible!

20

## MESI Cache Coherence

- Most common hardware implementation of discussed requirements
  - aka. "Illinois protocol"

Each line has one of the following states (in a cache):

### Modified (M)

- Local copy has been modified, no copies in other caches
- Memory is stale

### Exclusive (E)

- No copies in other caches
- Memory is up to date

### Shared (S)

- Unmodified copies may exist in other caches
- Memory is up to date

### Invalid (I)

- Line is not in cache

21

## Terminology

### Clean line:

- Content of cache line and main memory is identical (also: memory is up to date)
- Can be evicted without write-back

### Dirty line:

- Content of cache line and main memory differ (also: memory is stale)
- Needs to be written back eventually
  - Time depends on protocol details

### Bus transaction:

- A signal on the bus that can be observed by all caches
- Usually blocking

### Local read/write:

- A load/store operation originating at a core connected to the cache

22

## Transitions in response to local reads

- State is M**
  - No bus transaction
- State is E**
  - No bus transaction
- State is S**
  - No bus transaction
- State is I**
  - Generate bus read request (BusRd)
  - May force other cache operations (see later)
  - Other cache(s) signal "sharing" if they hold a copy
  - If shared was signaled, go to state S
  - Otherwise, go to state E
- After update: return read value**

23

## Transitions in response to local writes

- State is M**
  - No bus transaction
- State is E**
  - No bus transaction
  - Go to state M
- State is S**
  - Line already local & clean
  - There may be other copies
  - Generate bus read request for upgrade to exclusive (BusRdX\*)
  - Go to state M
- State is I**
  - Generate bus read request for exclusive ownership (BusRdX)
  - Go to state M

24

## Transitions in response to snooped BusRd

- **State is M**
  - Write cache line back to main memory
  - Signal "shared"
  - Go to state S
- **State is E**
  - Signal "shared"
  - Go to state S and signal "shared"
- **State is S**
  - Signal "shared"
- **State is I**
  - Ignore

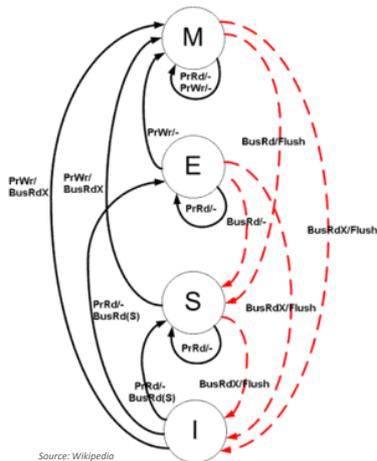
25

## Transitions in response to snooped BusRdX

- **State is M**
    - Write cache line back to memory
    - Discard line and go to I
  - **State is E**
    - Discard line and go to I
  - **State is S**
    - Discard line and go to I
  - **State is I**
    - Ignore
- **BusRdX\* is handled like BusRdX!**

26

## MESI State Diagram (FSM)



27

## Small Exercise

- **Initially: all in I state**

Action	P1 state	P2 state	P3 state	Bus action	Data from
P1 reads x					
P2 reads x					
P1 writes x					
P1 reads x					
P3 writes x					

28

## Small Exercise

- **Initially: all in I state**

Action	P1 state	P2 state	P3 state	Bus action	Data from
P1 reads x	E	I	I	BusRd	Memory
P2 reads x	S	S	I	BusRd	Memory
P1 writes x	M	I	I	BusRdX*	Cache
P1 reads x	M	I	I	-	Cache
P3 writes x	I	I	M	BusRdX	Memory

29

## Optimizations?

- **Class question: what could be optimized in the MESI protocol to make a system faster?**

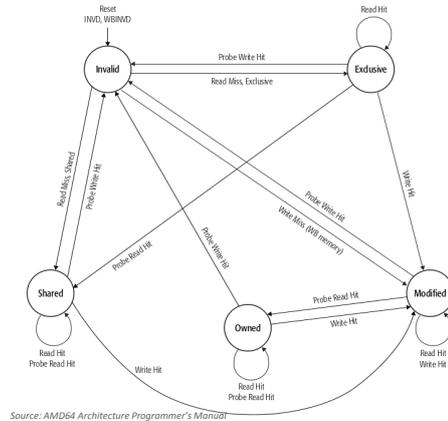
30

## Related Protocols: MOESI (AMD)

- **Extended MESI protocol**
- **Cache-to-cache transfer of modified cache lines**
  - Cache in M or O state always transfers cache line to requesting cache
  - No need to contact (slow) main memory
- **Avoids write back when another process accesses cache line**
  - Good when cache-to-cache performance is higher than cache-to-memory  
*E.g., shared last level cache!*
- **Broadcasts updates in O state**
  - Additional load on the bus

31

## MOESI State Diagram

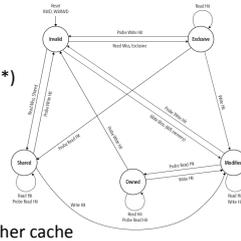


Source: AMD64 Architecture Programmer's Manual

32

## Related Protocols: MOESI (AMD)

- **Modified (M): Modified Exclusive**
  - No copies in other caches, local copy dirty
  - Memory is stale, cache supplies copy (reply to BusRd\*)
- **Owner (O): Modified Shared**
  - Exclusive right to make changes
  - Other S copies may exist ("dirty sharing")
  - Memory is stale, cache supplies copy (reply to BusRd\*)
- **Exclusive (E):**
  - Same as MESI (one local copy, up to date memory)
- **Shared (S):**
  - Unmodified copy may exist in other caches
  - Memory is up to date unless an O copy exists in another cache
- **Invalid (I):**
  - Same as MESI



33

## Related Protocols: MESIF (Intel)

- **Modified (M): Modified Exclusive**
  - No copies in other caches, local copy dirty
  - Memory is stale, cache supplies copy (reply to BusRd\*)
- **Exclusive (E):**
  - Same as MESI (one local copy, up to date memory)
- **Shared (S):**
  - Unmodified copy may exist in other caches
  - Memory is up to date unless an O copy exists in another cache
- **Invalid (I):**
  - Same as MESI
- **Forward (F):**
  - Special form of S state, other caches may have line in S
  - Most recent requester of line is in F state
  - Cache acts as responder for requests to this line

34

## Multi-level caches

- **Most systems have multi-level caches**
  - Problem: only "last level cache" is connected to bus or network
  - Snoop requests are relevant for inner-levels of cache (L1)
  - Modifications of L1 data may not be visible at L2 (and thus the bus)
- **L1/L2 modifications**
  - On BusRd check if line is in M state in L1  
*It may be in E or S in L2!*
  - On BusRdX(\*) send invalidations to L1
  - Everything else can be handled in L2
- **If L1 is write through, L2 could "remember" state of L1 cache line**
  - May increase traffic though

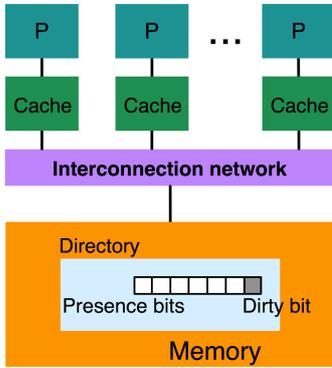
35

## Directory-based cache coherence

- **Snooping does not scale**
  - Bus transactions must be *globally* visible
  - Implies broadcast
- **Typical solution: tree-based (hierarchical) snooping**
  - Root becomes a bottleneck
- **Directory-based schemes are more scalable**
  - Directory (entry for each CL) keeps track of all owning caches
  - Point-to-point update to involved processors  
*No broadcast*  
*Can use specialized (high-bandwidth) network, e.g., HT, QPI ...*

36

## Basic Scheme



- System with  $N$  processors  $P_i$
- For each memory block (size: cache line) maintain a directory entry
  - $N$  presence bits
  - Set if block in cache of  $P_i$
  - 1 dirty bit
- For each cache block
  - 1 valid and 1 dirty bit
- First proposed by Censier and Feautrier (1978)

37

## Directory-based CC: Read miss

- $P_i$  intends to read, misses
- If dirty bit (in directory) is off
  - Read from main memory
  - Set presence[i]
  - Supply data to reader
- If dirty bit is on
  - Recall cache line from  $P_j$  (determine by presence[j])
  - Update memory
  - Unset dirty bit, block shared
  - Set presence[i]
  - Supply data to reader

38

## Directory-based CC: Write miss

- $P_i$  intends to write, misses
- If dirty bit (in directory) is off
  - Send invalidations to all processors  $P_j$  with presence[j] turned on
  - Unset presence bit for all processors
  - Set dirty bit
  - Set presence[i], owner  $P_i$
- If dirty bit is on
  - Recall cache line from owner  $P_j$
  - Update memory
  - Unset presence[j]
  - Set presence[i], dirty bit remains set
  - Supply data to writer

39

## Discussion

- Scaling of memory bandwidth
  - No centralized memory
- Directory-based approaches scale with restrictions
  - Require presence bit for each cache
  - Number of bits determined at design time
  - Directory requires memory (size scales linearly)
  - Shared vs. distributed directory
- Software-emulation
  - Distributed shared memory (DSM)
  - Emulate cache coherence in software (e.g., TreadMarks)
  - Often on a per-page basis, utilizes memory virtualization and paging

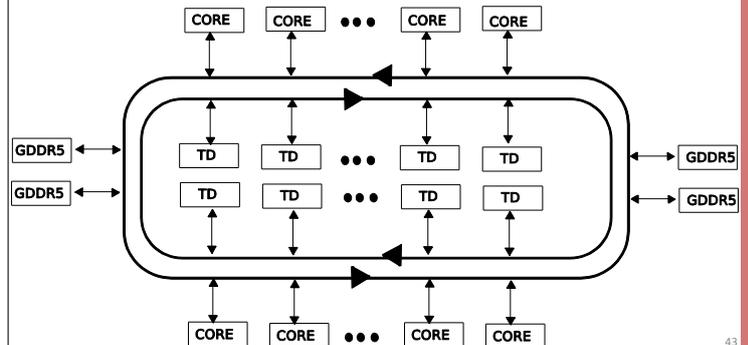
41

## Open Problems (for projects or theses)

- Tune algorithms to cache-coherence schemes
  - What is the optimal parallel algorithm for a given scheme?
  - Parameterize for an architecture
- Measure and classify hardware
  - Read Maranget et al. "A Tutorial Introduction to the ARM and POWER Relaxed Memory Models" and have fun!
  - RDMA consistency is barely understood!
  - GPU memories are not well understood!  
*Huge potential for new insights!*
- Can we program (easily) without cache coherence?
  - How to fix the problems with inconsistent values?
  - Compiler support (issues with arrays)?

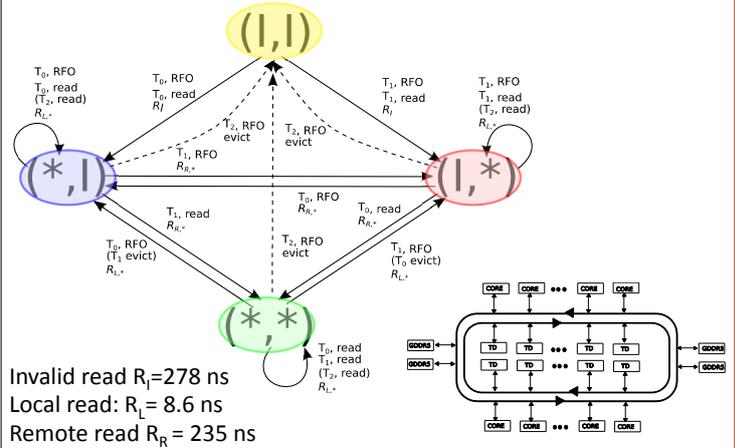
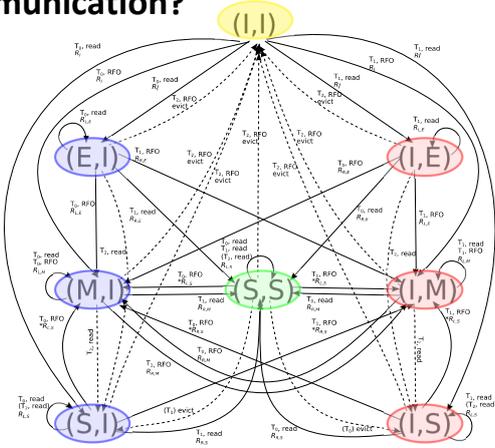
42

## Case Study: Intel Xeon Phi

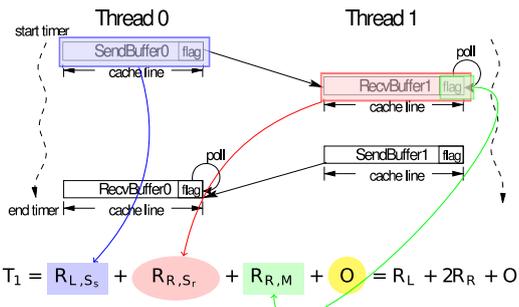


43

## Communication?



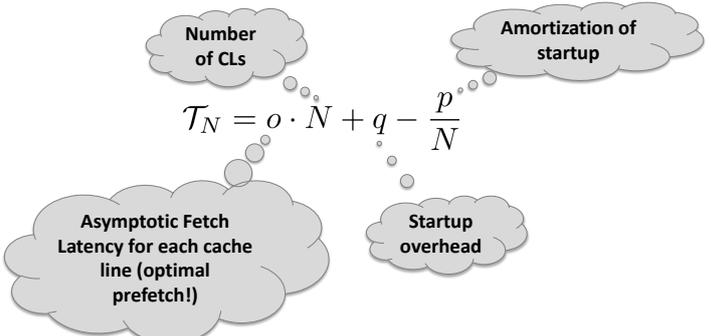
## Single-Line Ping Pong



- Prediction for both in E state: 479 ns
- Measurement: 497 ns (O=18)

## Multi-Line Ping Pong

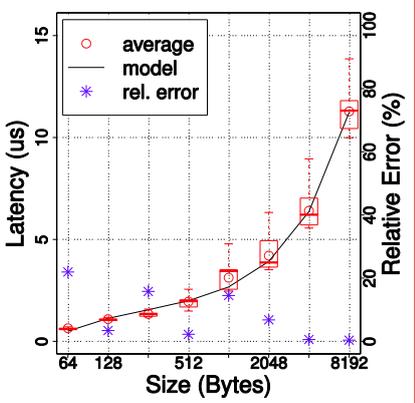
- More complex due to prefetch



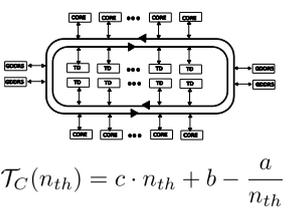
## Multi-Line Ping Pong

$$T_N = o \cdot N + q - \frac{p}{N}$$

- E state:
  - o=76 ns
  - q=1,521ns
  - p=1,096ns
- I state:
  - o=95ns
  - q=2,750ns
  - p=2,017ns



## DTD Contention



- E state:
  - a=0ns
  - b=320ns
  - c=56.2ns

