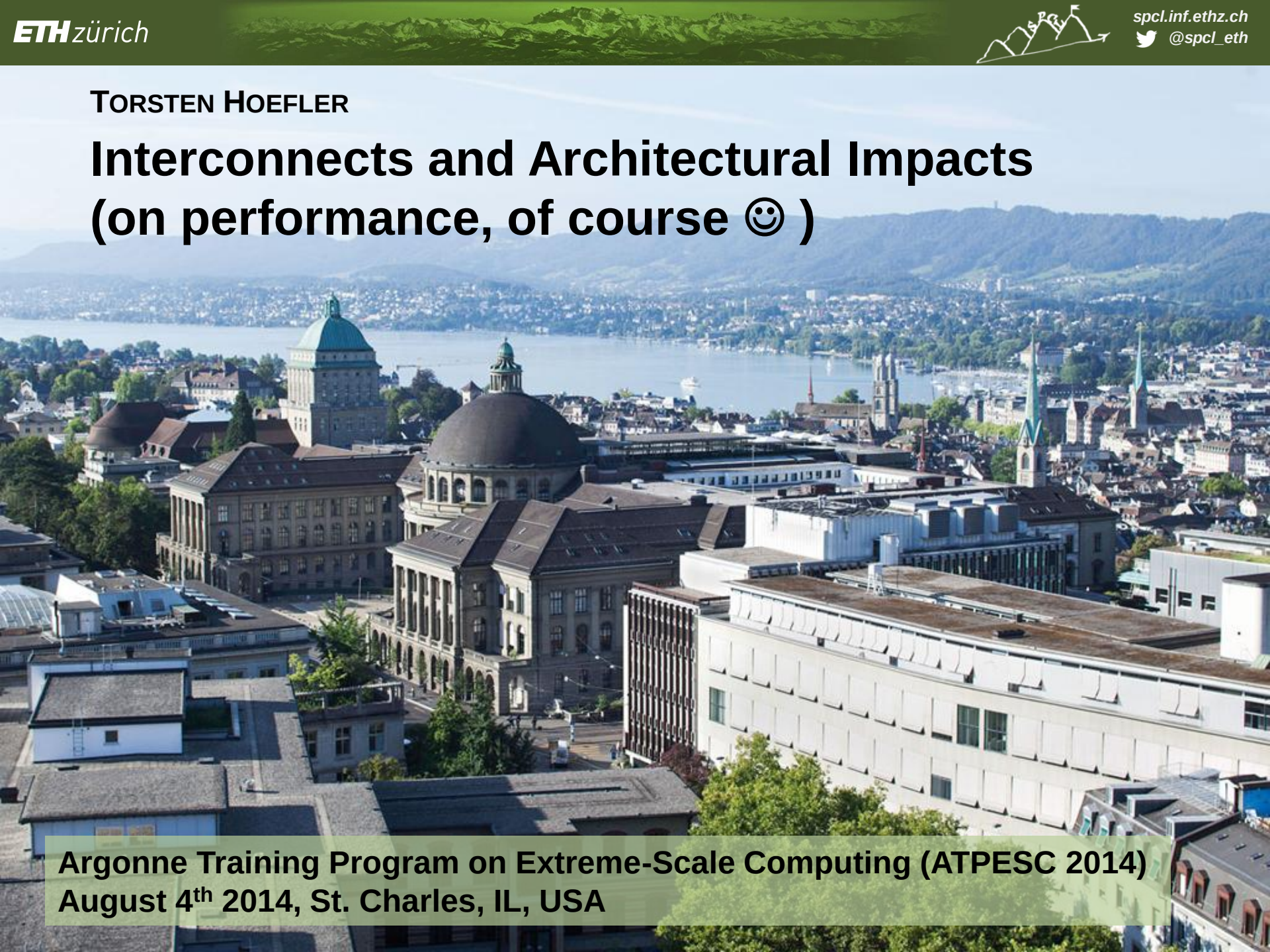
TORSTEN HOEFLER

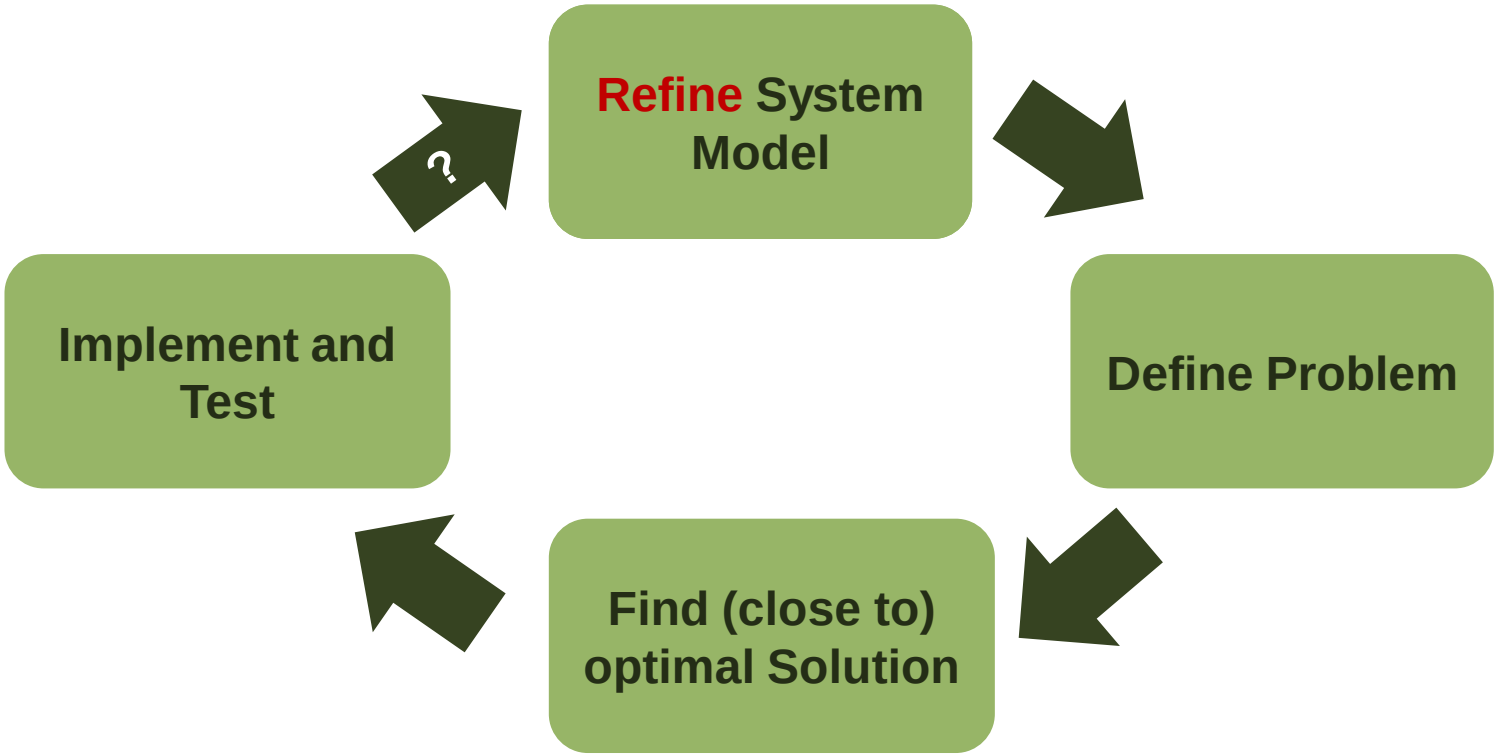
Interconnects and Architectural Impacts (on performance, of course ☺)



Argonne Training Program on Extreme-Scale Computing (ATPESC 2014)
August 4th 2014, St. Charles, IL, USA

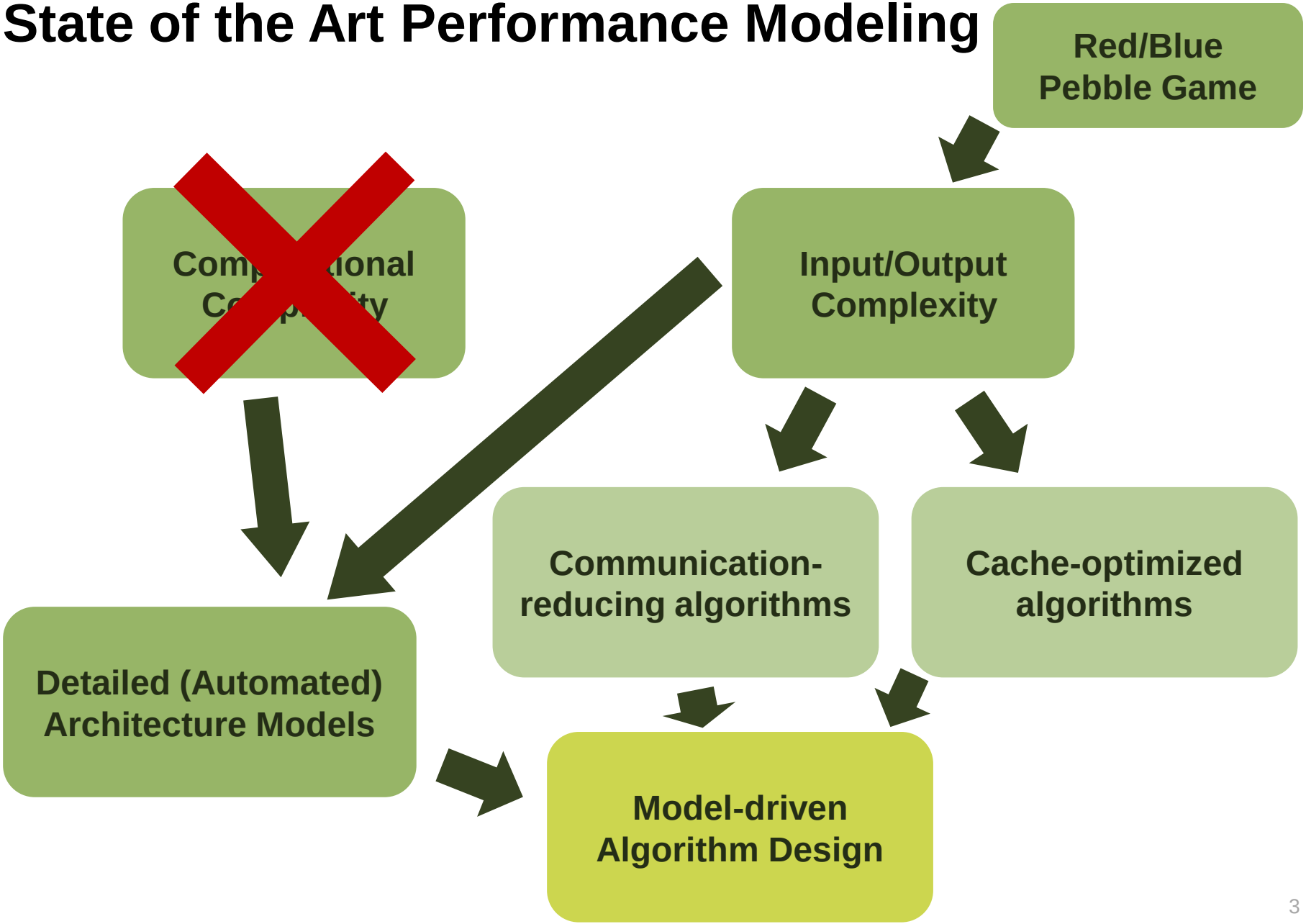
Model-based Performance Engineering

- **My dream: provably optimal performance (time and energy)**
  - From problem to machine code



- **Will demonstrate techniques & insights**
  - And obstacles ☺

State of the Art Performance Modeling



Example: Message Passing, Log(G)P

A new parallel machine model reflects the critical technology trends underlying parallel computers

A Practical Model of Parallel Computation

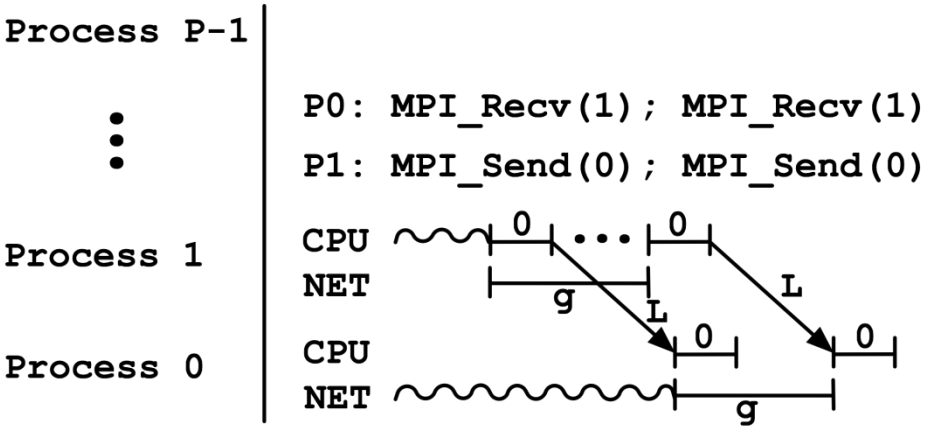
LogP

David E. Culler, Richard M. Karp, David Patterson, Abhijit Sahay, Eunice E. Santos, Klaus Erik Schauser, Ramesh Subramonian, and Thorsten von Eicken

UR GOAL IS TO DEVELOP A MODEL OF PARALLEL COMPUTATION THAT WILL serve as a basis for the design and analysis of fast, portable parallel algorithms, such as algorithms that can be implemented effectively on a wide variety of current and future parallel machines. If we look at the body of parallel algorithms developed under current parallel models, many are impractical because they exploit artificial factors not present in any rea-

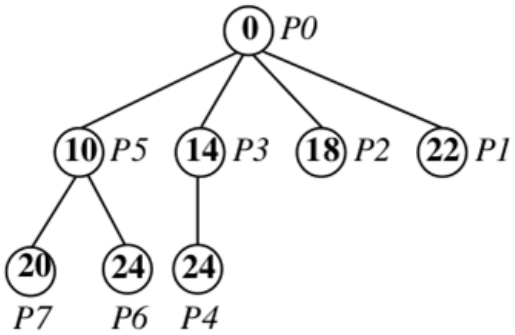
PRAM consists of a collection of processors which compute synchronously in parallel and communicate with a global random access

CACM 1996



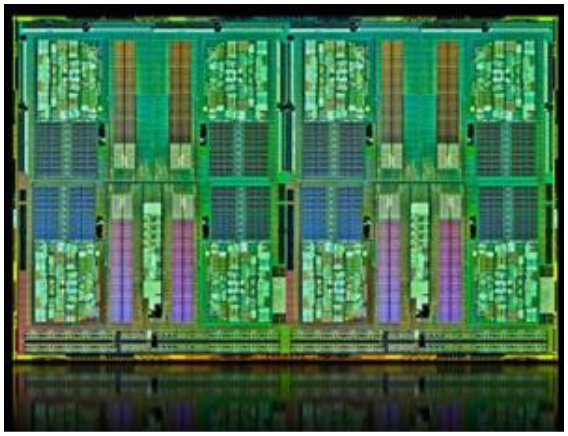
Broadcast Problem

Optimal Solution [1]

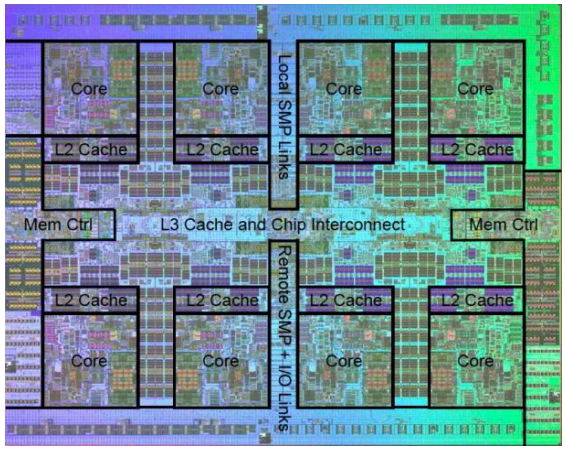


[1]: Karp et al.: "Optimal broadcast and summation in the LogP model", SPAA 1993

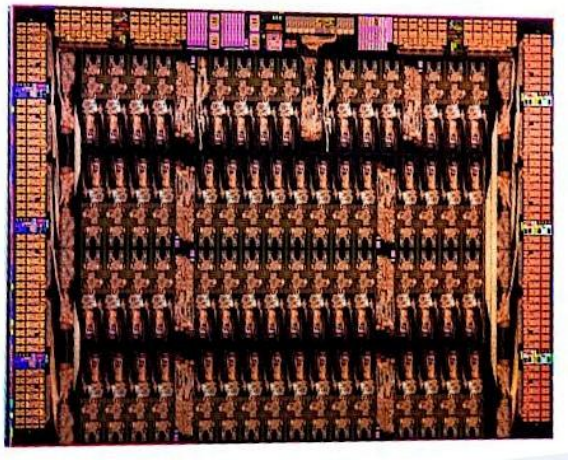
Hardware Reality



Interlagos, 8/16 cores, source: AMD

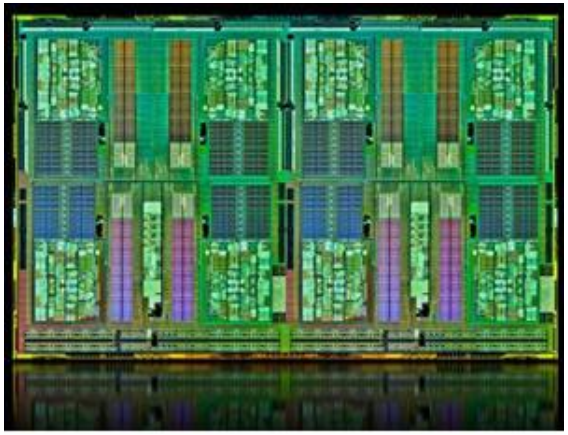


POWER 7, 8 cores, source: IBM

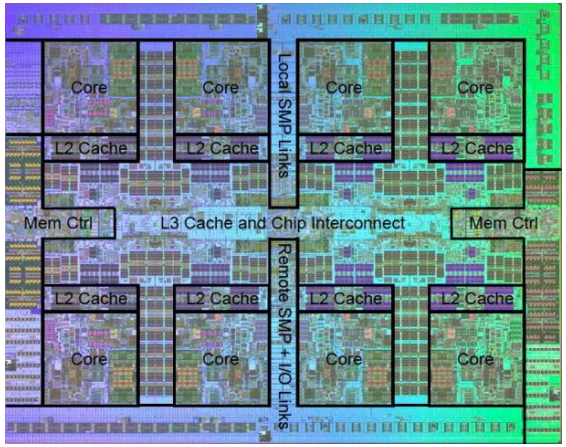


Xeon Phi, 64 cores, source: Intel

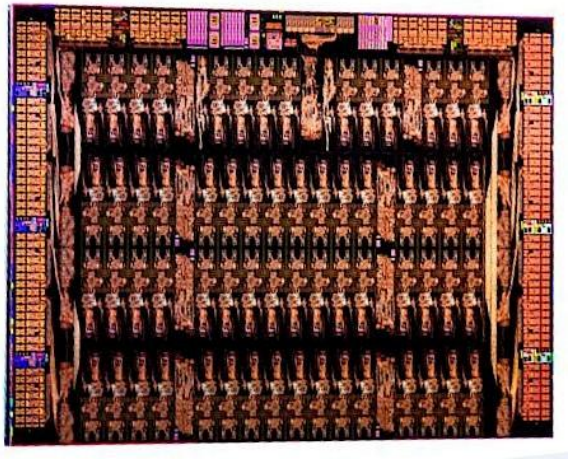
Hardware Reality



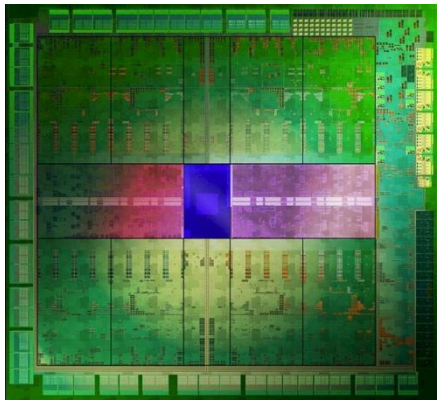
Interlagos, 8/16 cores, source: AMD



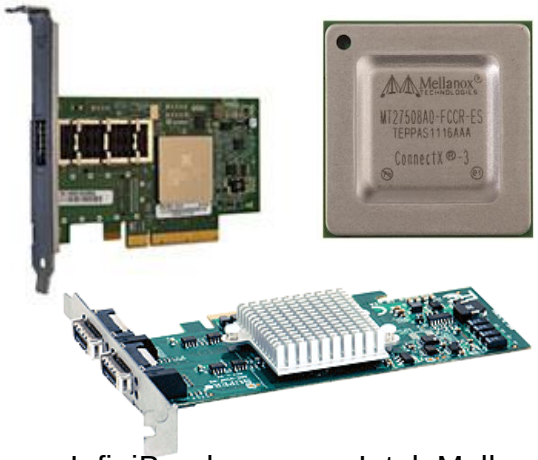
POWER 7, 8 cores, source: IBM



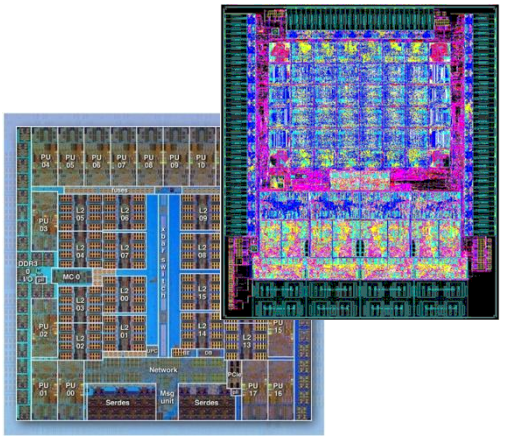
Xeon Phi, 64 cores, source: Intel



Kepler GPU, source: NVIDIA

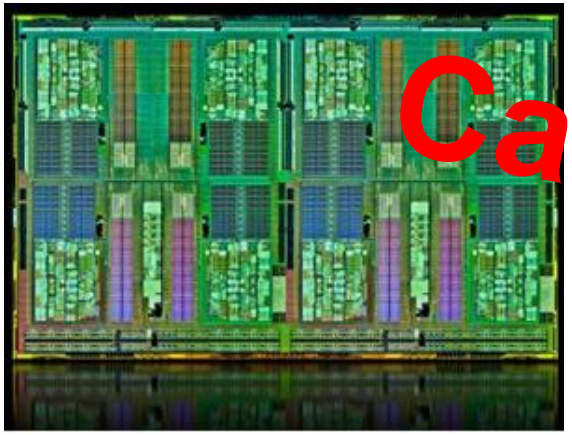


InfiniBand, sources: Intel, Mellanox

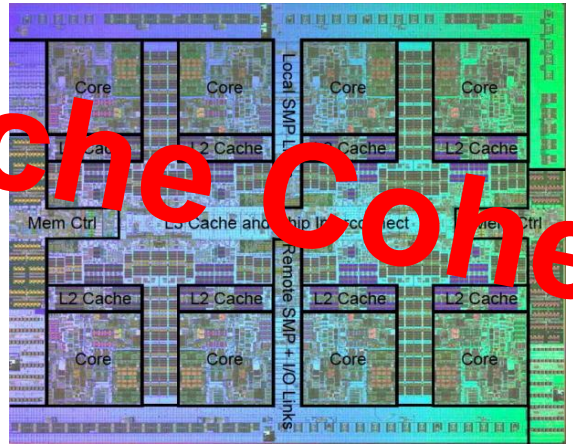


BG/Q, Cray Aries, sources: IBM, Cray

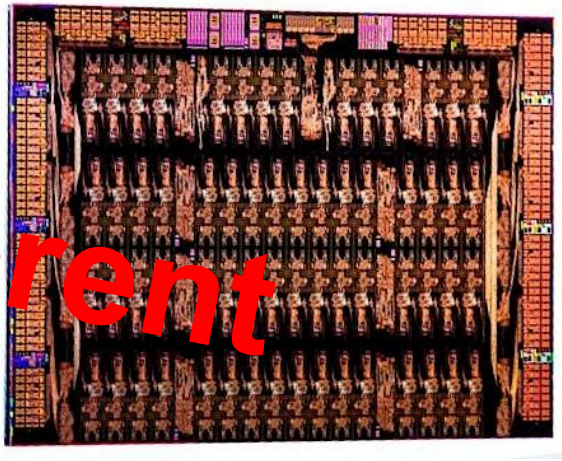
Hardware Reality



Interlagos, 8/16 cores, source: AMD

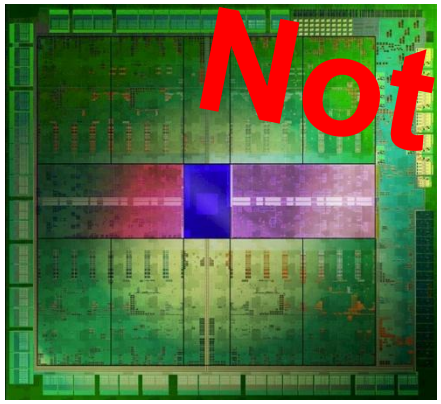


POWER 7, 8 cores, source: IBM



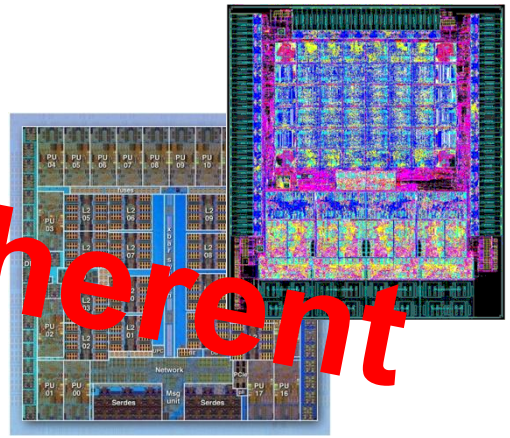
Xeon Phi, 64 cores, source: Intel

Cache Coherent



Kepler GPU, source: NVIDIA

InfiniBand, sources: Intel, Mellanox



BG/Q, Cray Aries, sources: IBM, Cray

Not Cache Coherent

Caching Strategies (repeat)

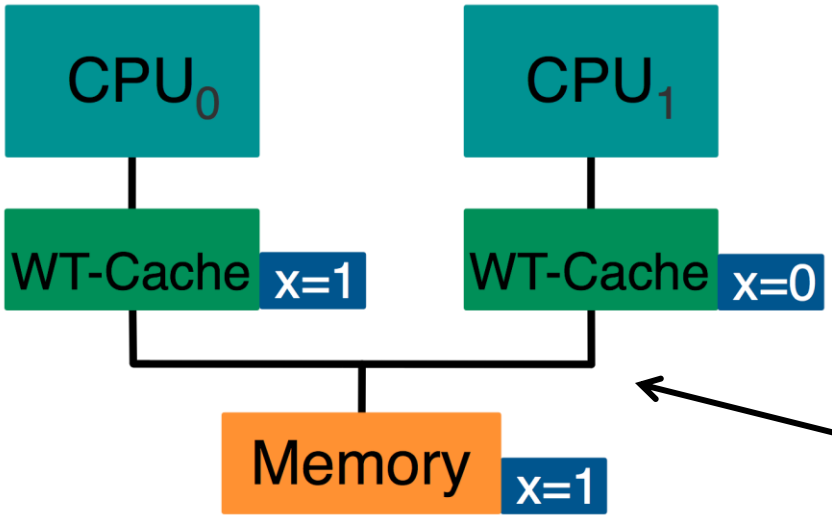
- **Remember:**
  - Write Back?
  - Write Through?

- **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 (not in this talk, very complex)

Write Through Cache



(initially X=0 in memory)

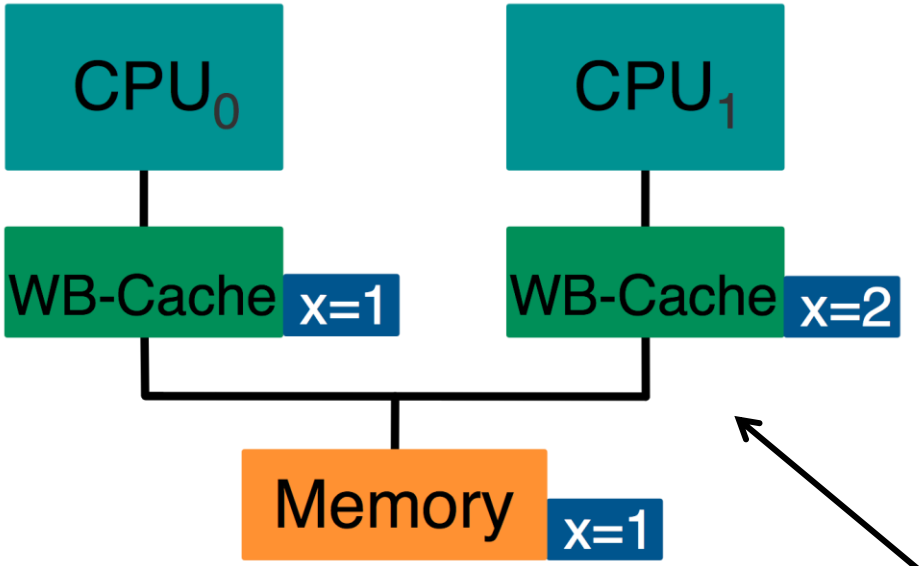
1. CPU_0 reads X from memory
   - loads X=0 into its cache
2. CPU_1 reads X from memory
   - loads X=0 into its cache
3. CPU_0 writes X=1
   - stores X=1 in its cache
   - stores X=1 in memory
4. CPU_1 reads X from its cache
   - loads X=0 from its cache

Incoherent value for X on CPU_1

CPU_1 may wait for update!

Requires write propagation!

Write Back Cache



(initially X=0 in memory)

1. CPU_0 reads X from memory
   - loads X=0 into its cache
2. CPU_1 reads X from memory
   - loads X=0 into its cache
3. CPU_0 writes X=1
   - stores X=1 in its cache
4. CPU_1 writes X =2
   - stores X=2 in its cache
5. CPU_1 writes back cache line
   - stores X=2 in in memory
6. CPU_0 writes back cache line
   - stores X=1 in memory

   Later store X=2 from CPU_1 lost

Requires write serialization!

A simple example

- **Assume C99:**

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

- **Two threads:**
  - a=b=0 and struct twoint aligned at a 64-Bytes cacheline boundary
  - Thread 0: write a=1
  - Thread 1: write b=1

- **Assume non-coherent write back cache**
  - What may end up in main memory?

Cache Coherence Protocol

- **Programmer cannot 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.

An Engineering Approach: Empirical start

- **Problem 1: stale reads**
  - Cache 1 holds value that was already modified in cache 2
  - Solution:

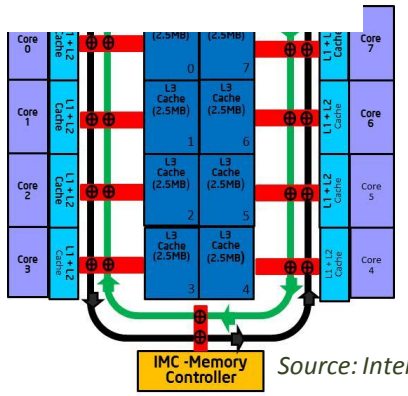
    Disallow this state

    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

Cache Coherence Approaches



Source: Intel

- **Based on invalidation**
  - Broadcast all coherency traffic (writes to shared lines) to all caches
  - Each cache snoops

    Invalidate lines written by other CPUs

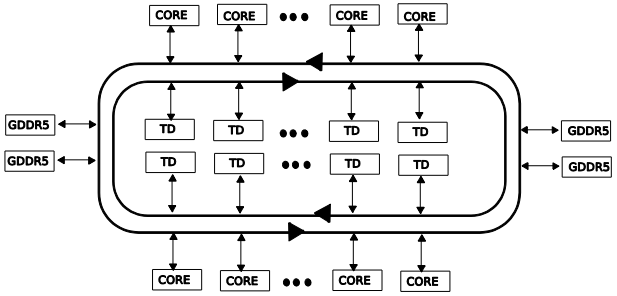
    Signal sharing for cache lines in local cache to other caches
  - Simple implementation for bus-based systems
  - Works at small scale, challenging at large-scale

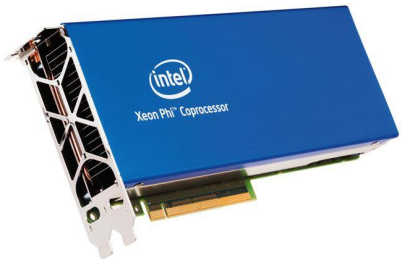
    E.g., Intel Sandy Bridge
- **Based on explicit updates**
  - Central directory for cache line ownership
  - Local write updates copies in remote caches

    Can update all CPUs at once

    Multiple writes cause multiple updates (more traffic)
  - Scalable but more complex/expensive

    E.g., Intel Xeon Phi





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!**

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

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

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**

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

Transitions in response to snooped BusRd

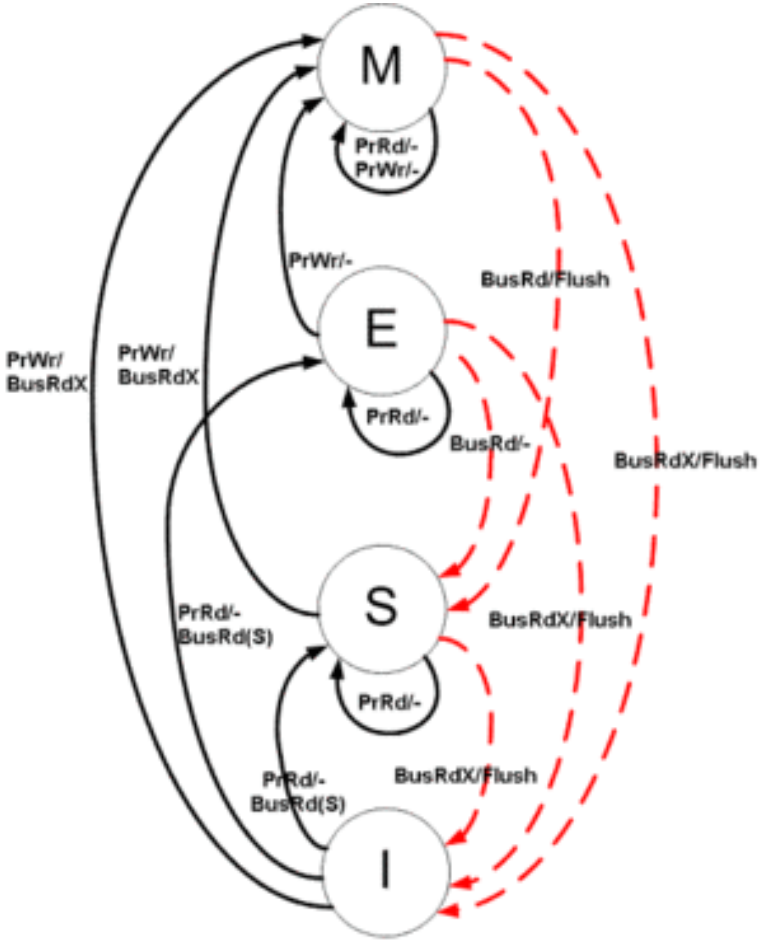
- **State is M**
  - Write cache line back to main memory
  - Signal “shared”
  - Go to state S
- **State is E**
  - Signal “shared”
  - Go to state S and signal “shared”
- **State is S**
  - Signal “shared”
- **State is I**
  - Ignore

Transitions in response to snooped BusRdX

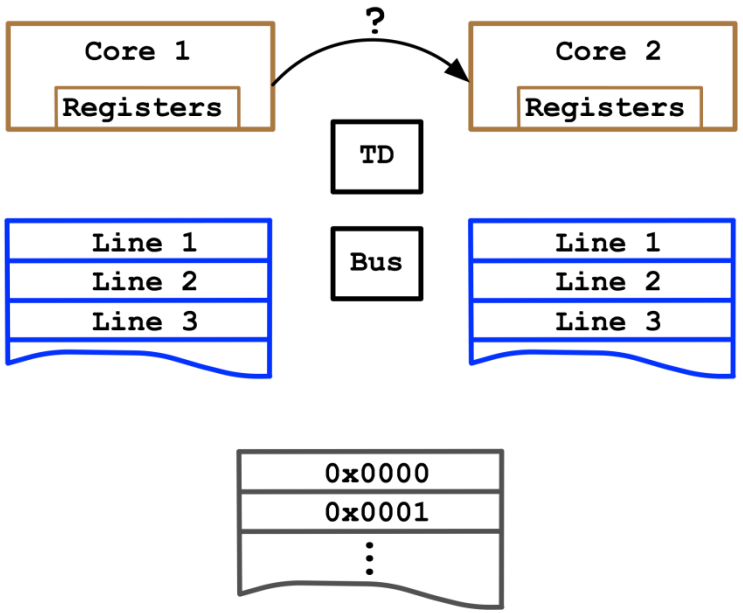
- **State is M**
  - Write cache line back to memory
  - Discard line and go to I
- **State is E**
  - Discard line and go to I
- **State is S**
  - Discard line and go to I
- **State is I**
  - Ignore

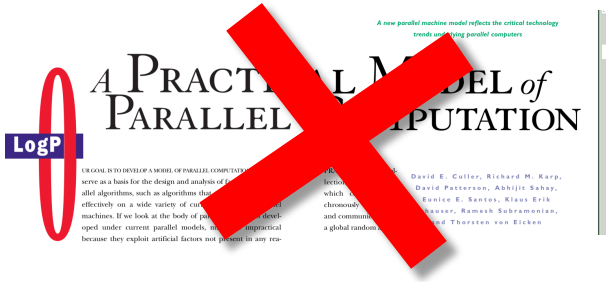
- **BusRdX* is handled like BusRdX!**

Topic: Cache-Coherent Communication

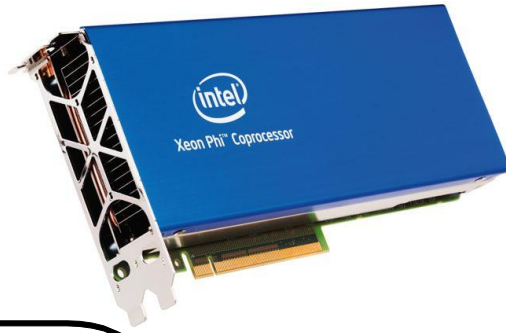


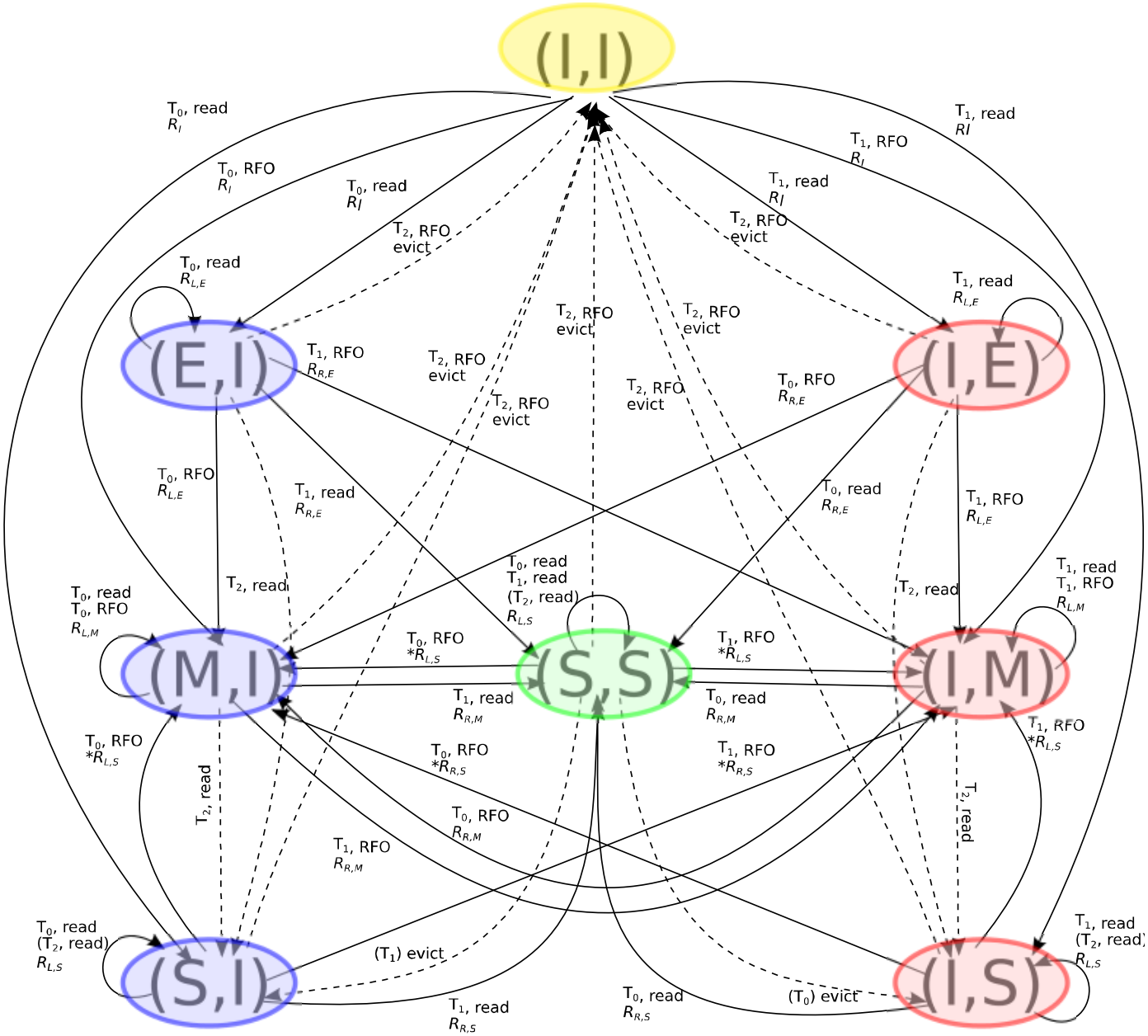
Source: Wikipedia



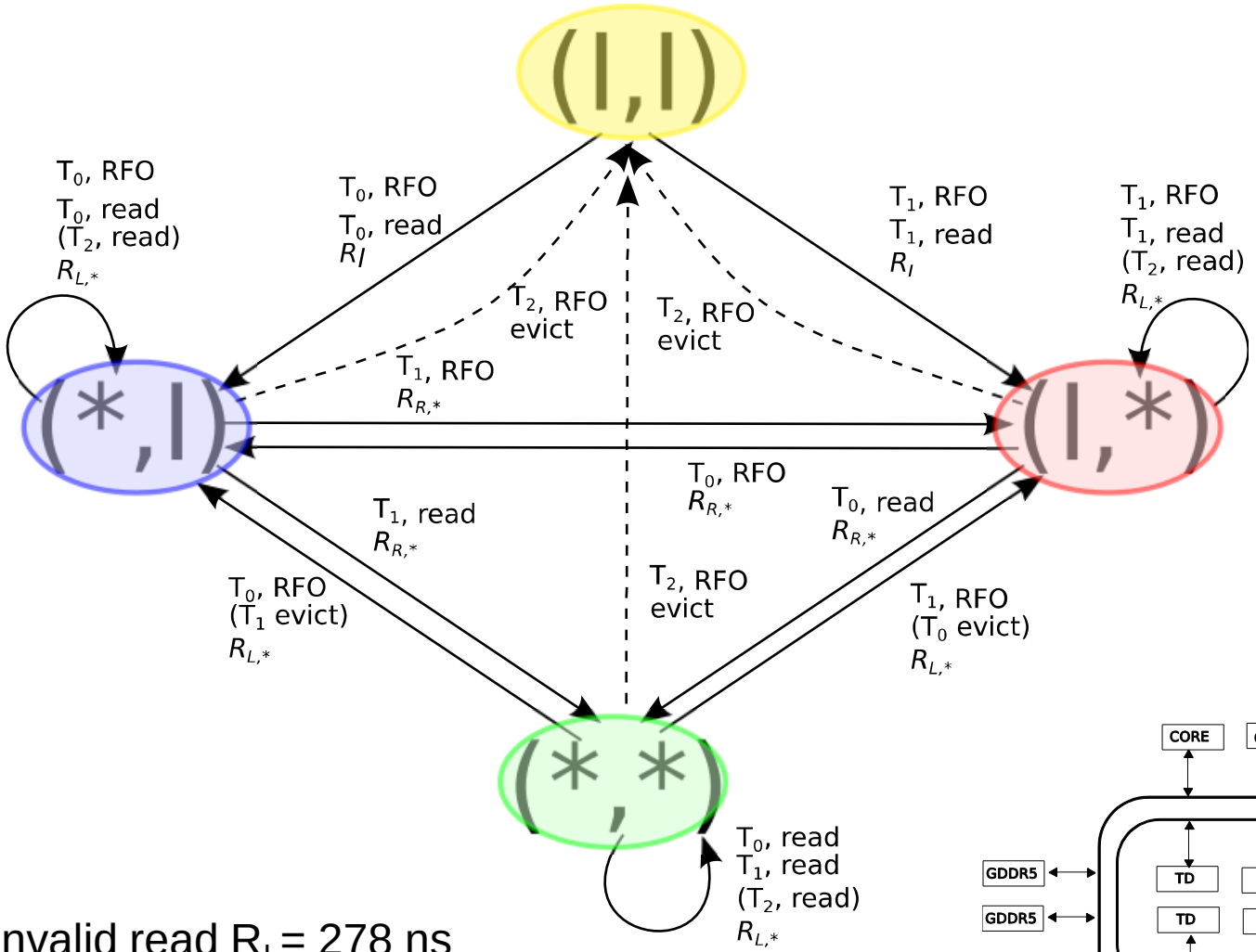


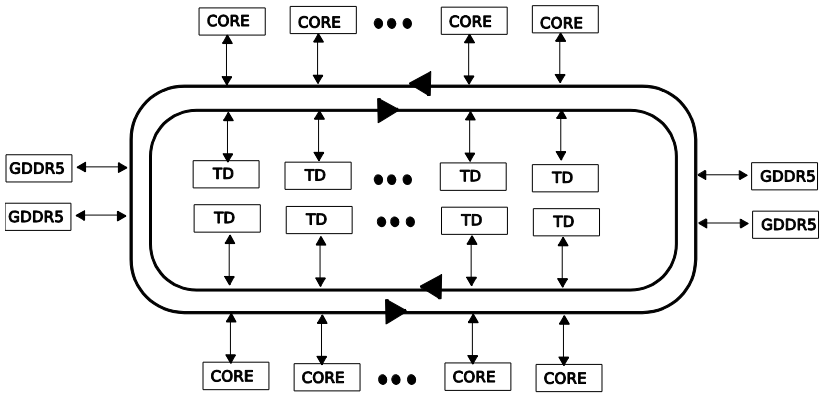
Xeon Phi (Rough) Architecture





Ramos, Hoefler: "Modeling Communication in Cache-Coherent SMP Systems - A Case-Study with Xeon Phi ", HPDC'13





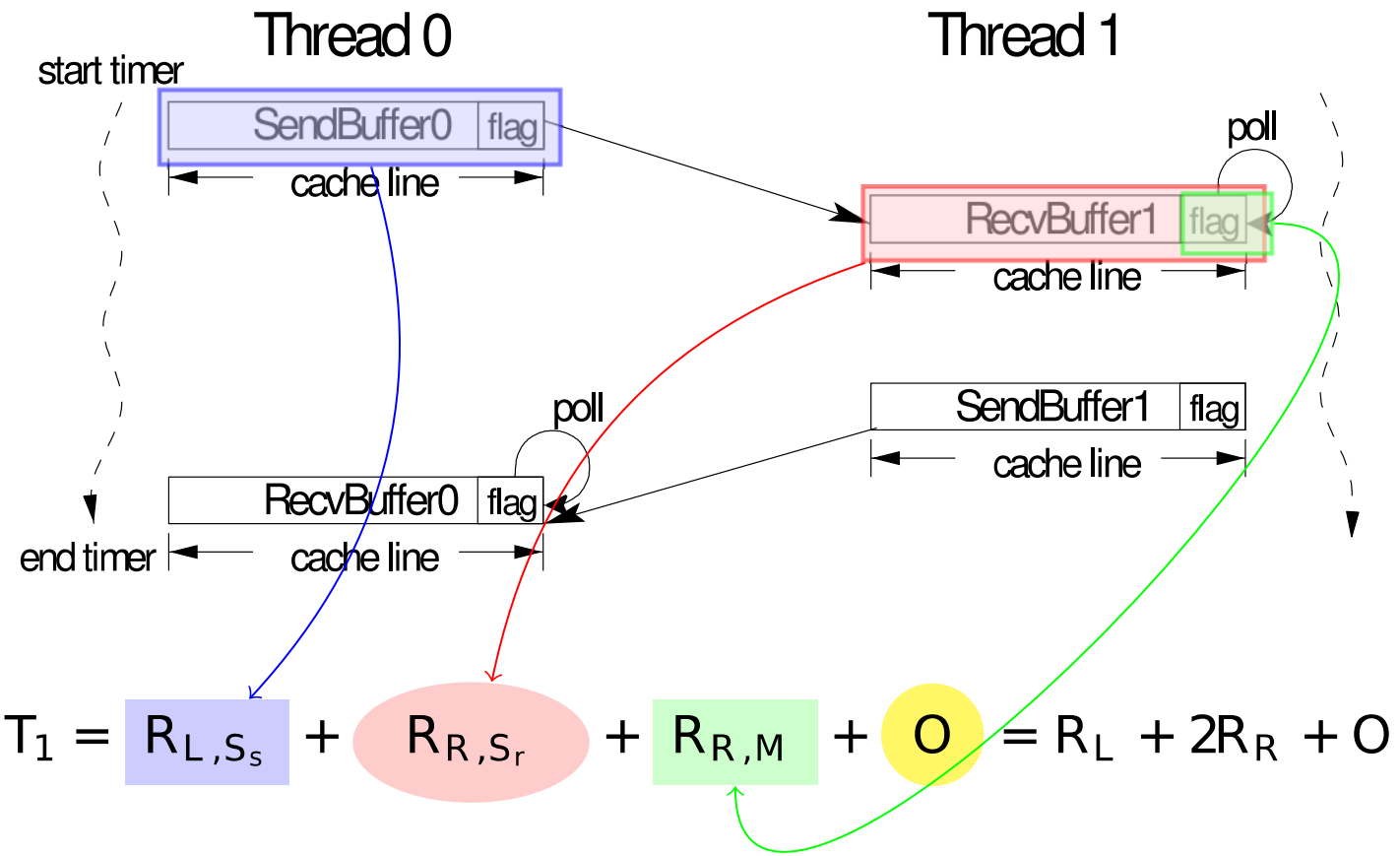
Invalid read R_I = 278 ns

Local read: R_L = 8.6 ns

Remote read R_R = 235 ns

Inspired by Molka et al.: "Memory performance and cache coherency effects on an Intel Nehalem multiprocessor system"

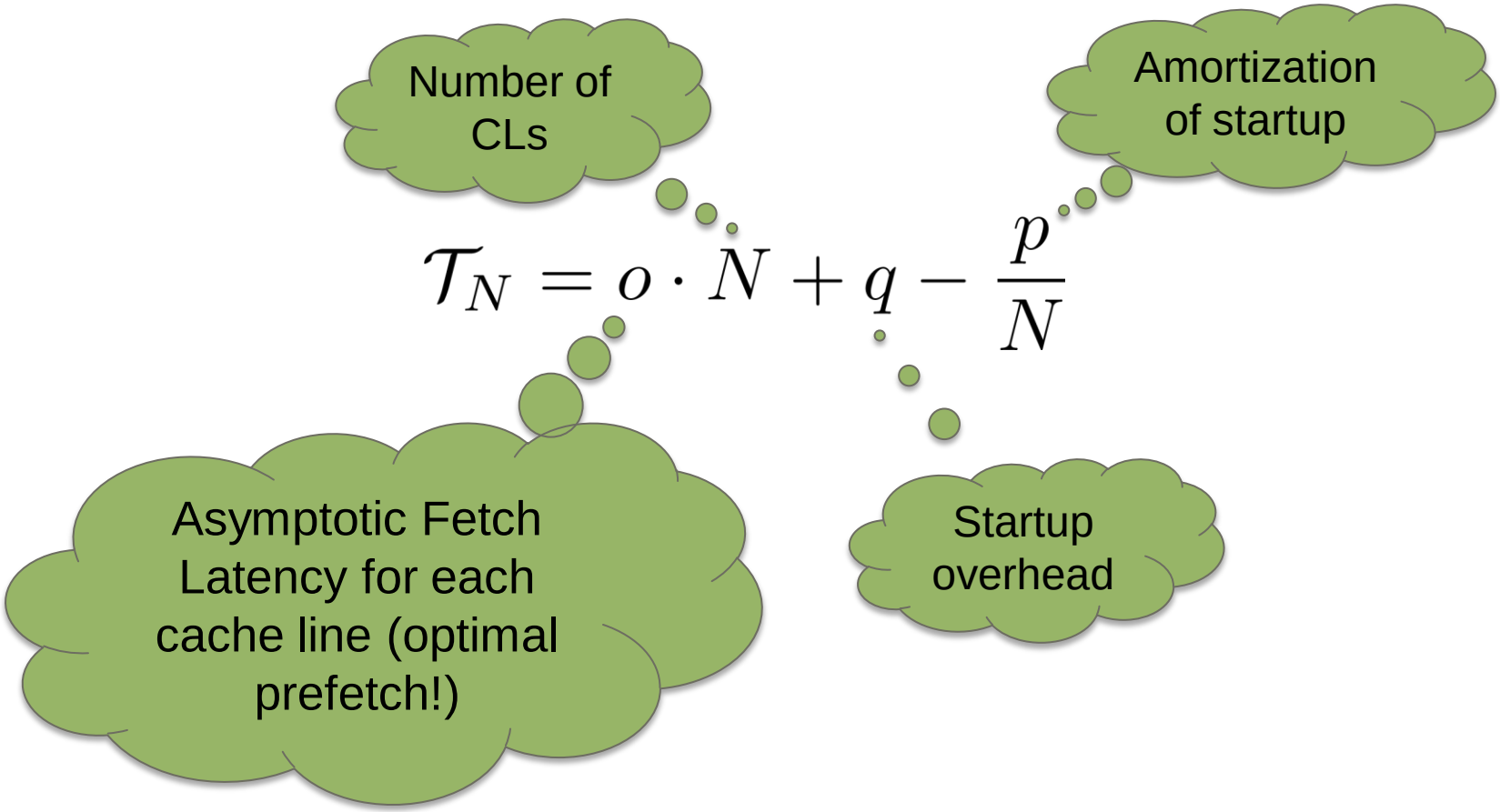
Single-Line Ping Pong



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

Multi-Line Ping Pong

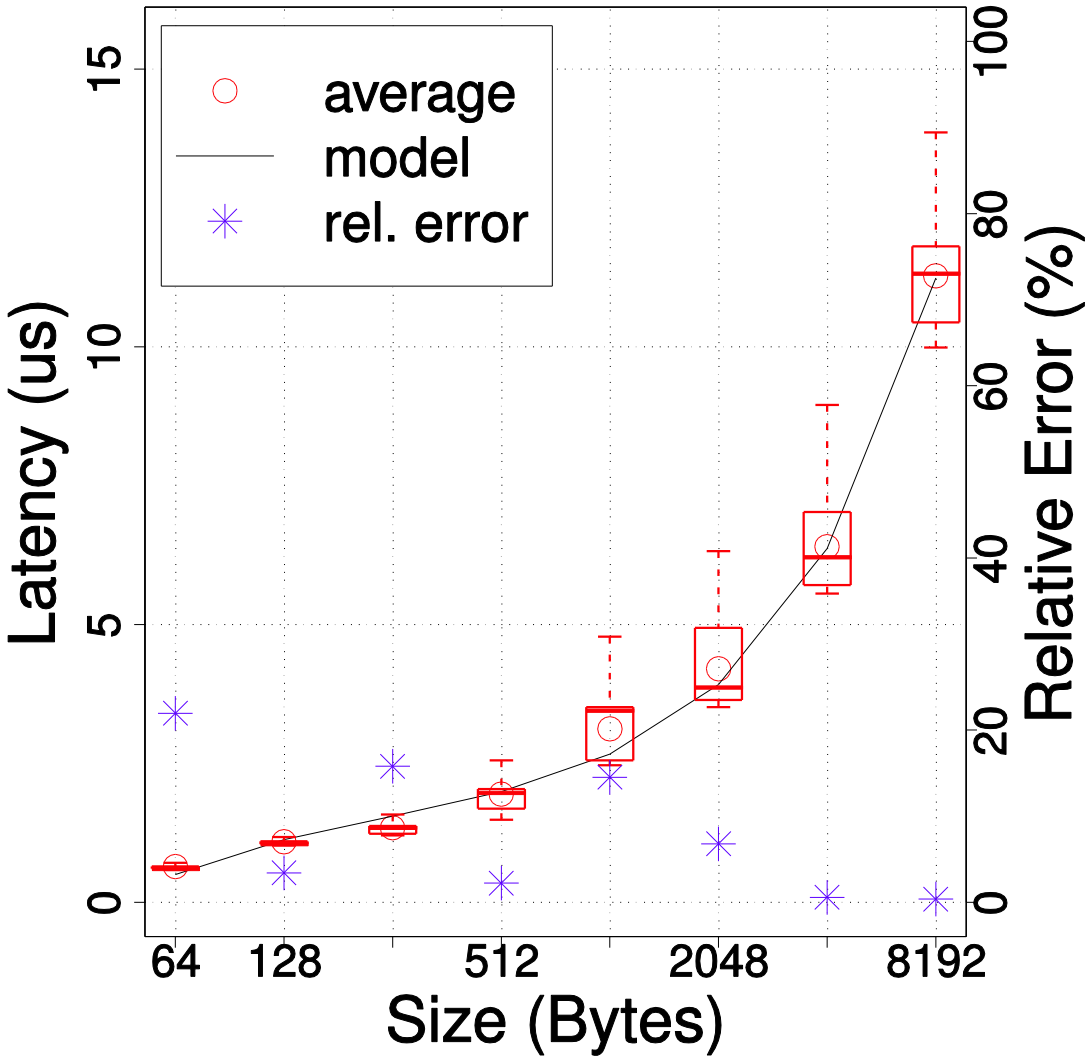
- **More complex due to prefetch/pipelining**



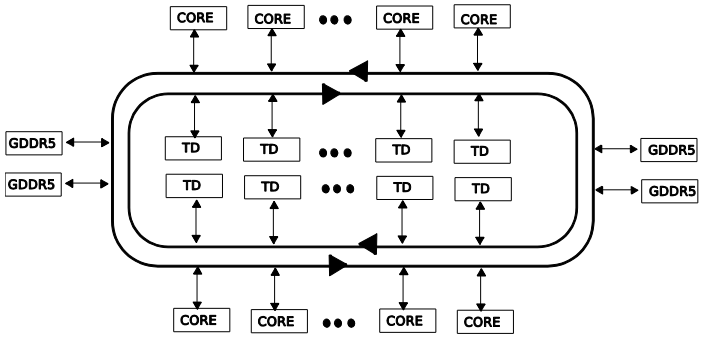
Multi-Line Ping Pong

$$\mathcal{T}_N = o \cdot N + q - \frac{p}{N}$$

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

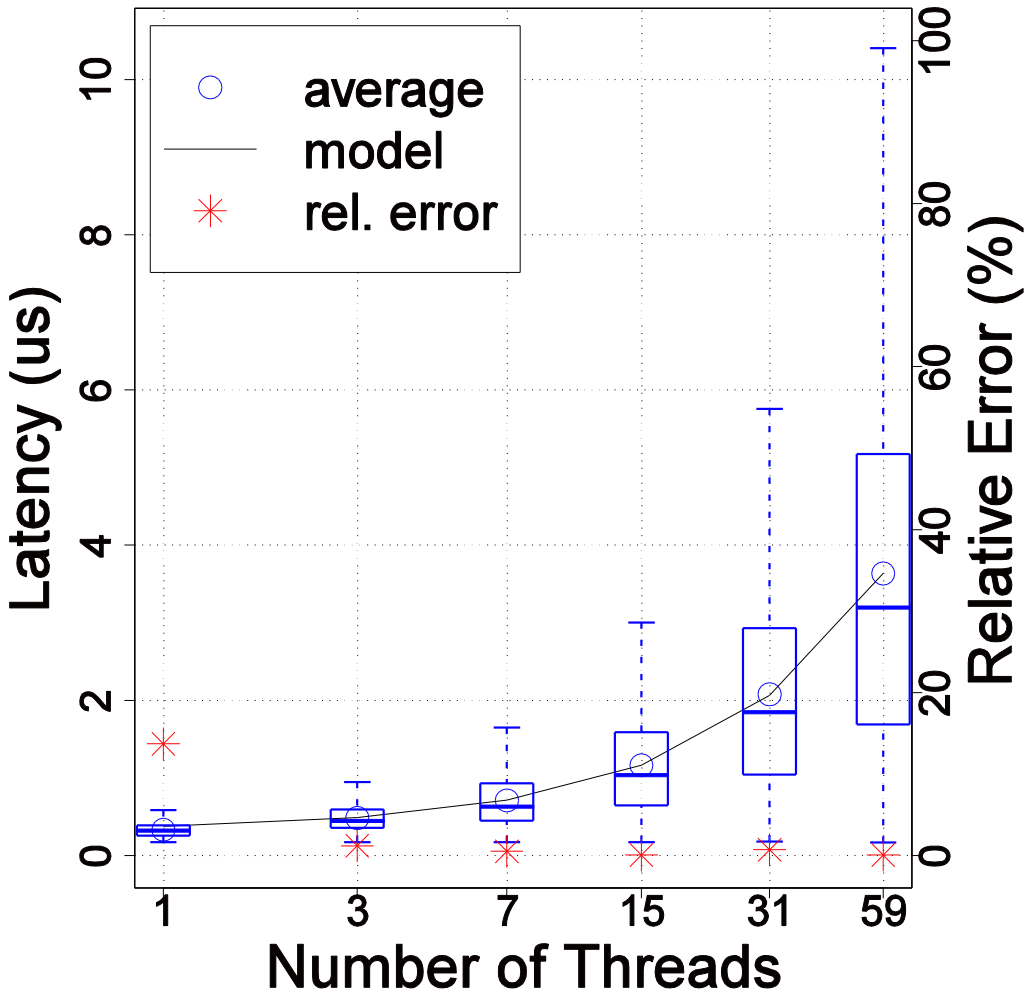


Contention ~~and/or Congestion~~?



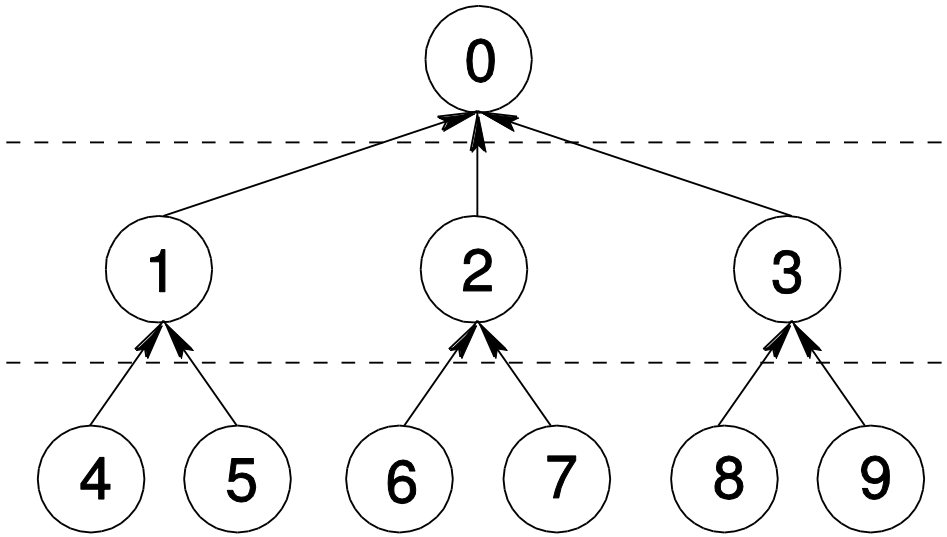
$$\mathcal{T}_C(n_{th}) = c \cdot n_{th} + b - \frac{a}{n_{th}}$$

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



Designing a Broadcast

- **Assume single cache line → forms a Tree**
  - We choose d levels and k_j children in level j
  - Reachable threads: $n_{th} \leq 1 + \sum_{i=1}^{d} \prod_{j=1}^{i} k_j$
  - Example: d=2, k_1=3, k_2=2:



$t_0 = 0$

$t_1 = k_1^*c + b$

$t_2 = t_1 + k_2^*c + b$

$$\mathcal{T}_{tree} = \sum_{i=1}^{d} \mathcal{T}_C(k_i) = \sum_{i=1}^{d} (c \cdot k_i + b)$$
$$= \sum_{i=1}^{d} (R_R + R_L + c \cdot (k_i - 1))$$

c = DTD contention
b = transmit latency

Designing Optimal Algorithms

- **Broadcast example:**

Bcast cost

Number of levels

Reached threads

$$\mathcal{T}_{tree} = \sum_{i=1}^{d} \mathcal{T}_C(k_i) = \sum_{i=1}^{d} (c \cdot k_i + b)$$

$$= \sum_{i=1}^{d} (R_R + R_L + c \cdot (k_i - 1))$$

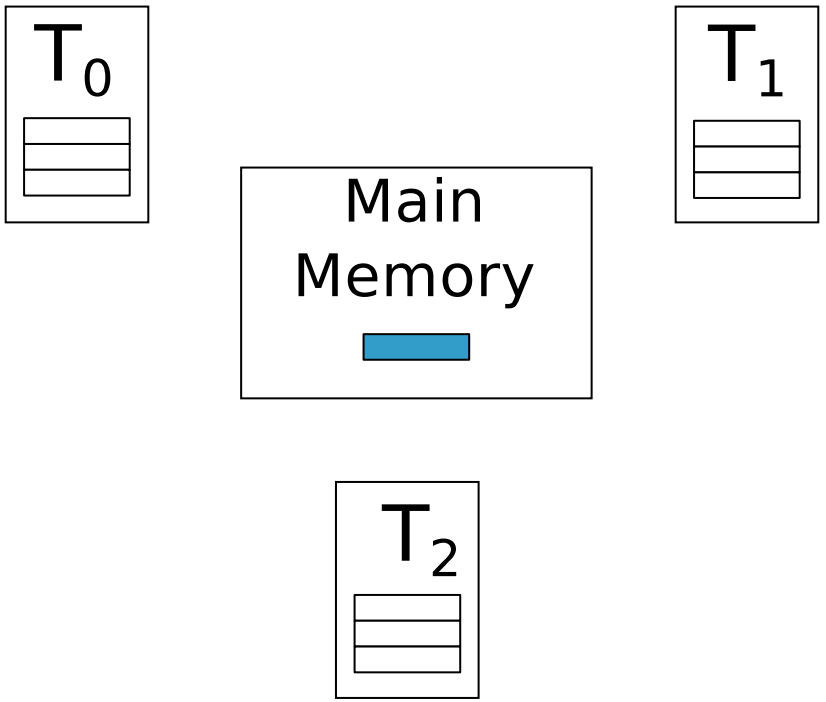
$$n_{th} \leq 1 + \sum_{i=1}^{d} \prod_{j=1}^{i} k_j$$

$$\mathcal{T}_{sbcast} = \min_{d,k_i} \left(\mathcal{T}_{fw} + \sum_{i=1}^{d} (c \cdot k_i + b) + \sum_{i=1}^{d} \mathcal{T}_{nb}(k_i + 1) \right)$$

$$N \leq 1 + \sum_{i=1}^{d} \prod_{j=1}^{i} k_j, \quad \forall i < j, k_i \leq k_j$$

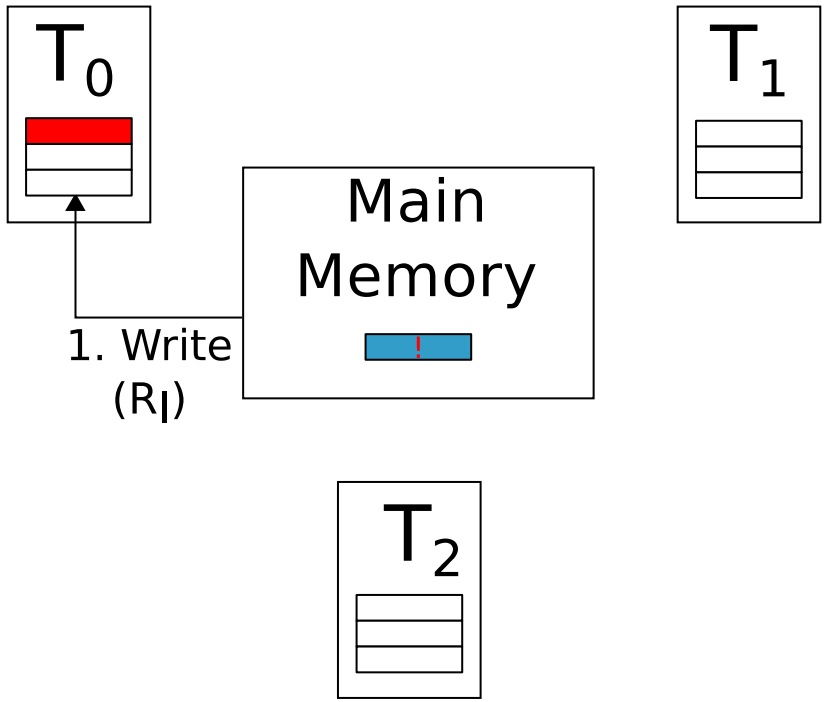
Min-Max Modeling

- **Example:**
  - T_0 + T_1 write CL
  - T_2 polls for completion



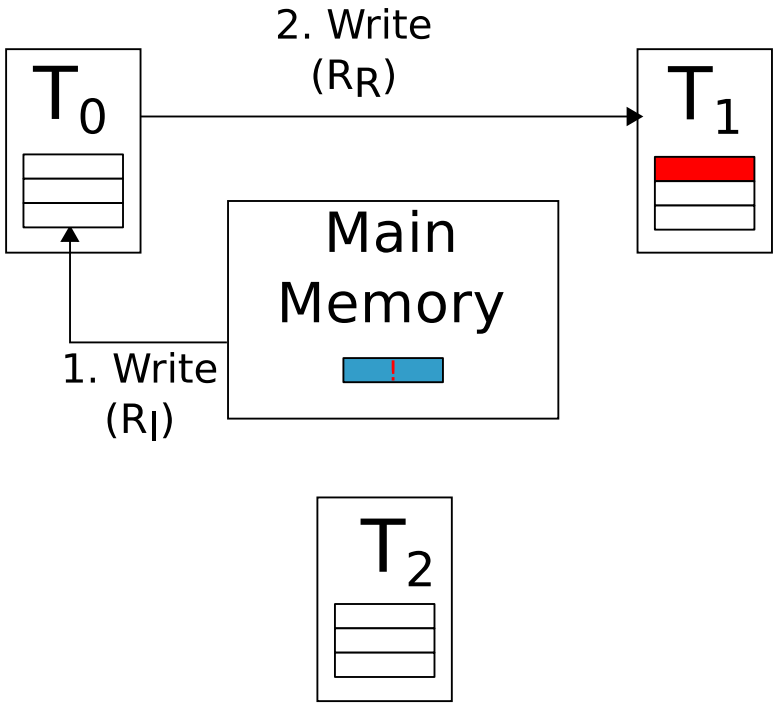
Min-Max Modeling

- **Example:**
  - T_0 + T_1 write CL
  - T_2 polls for completion



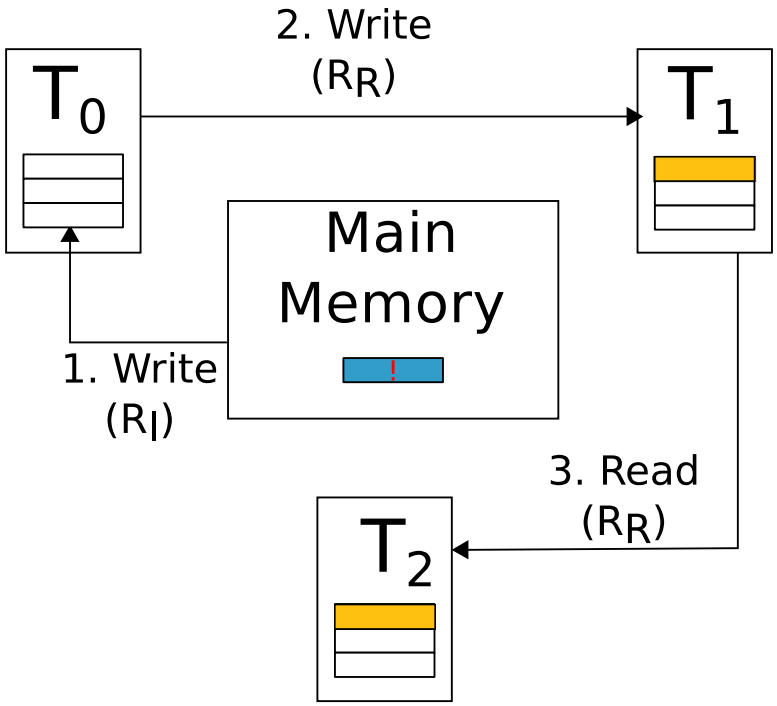
Min-Max Modeling

- **Example:**
  - T_0 + T_1 write CL
  - T_2 polls for completion



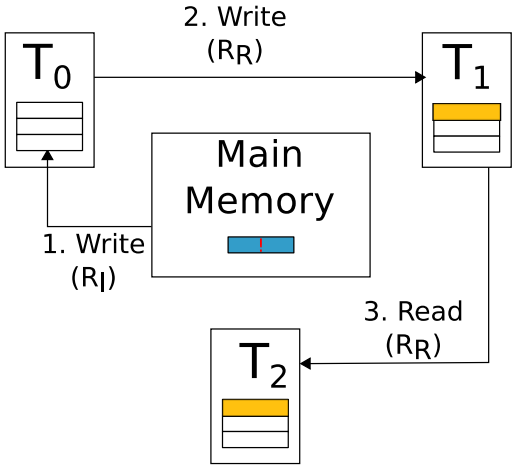
Min-Max Modeling

- **Example:**
  - T_0 + T_1 write CL
  - T_2 polls for completion



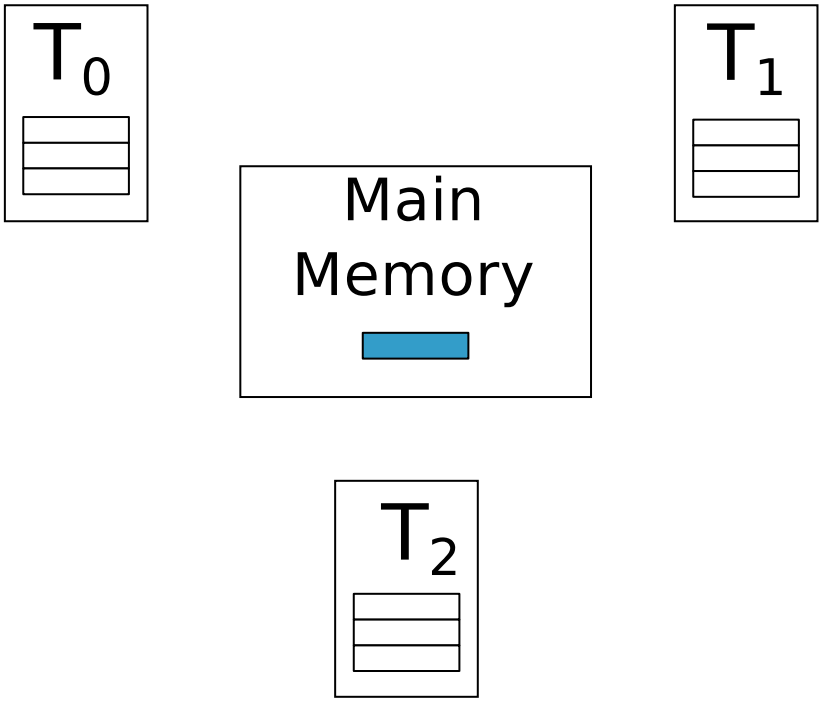
Min-Max Modeling

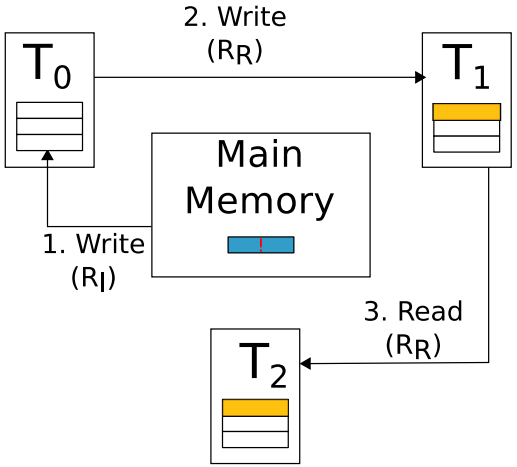
- **Example:**
  - T_0 + T_1 write CL
  - T_2 polls for completion



Min-Max Modeling

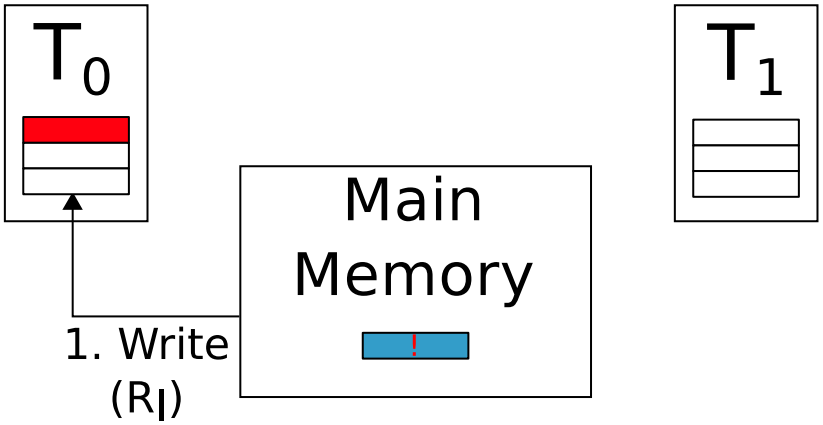
- **Example:**
  - T_0 + T_1 write CL
  - T_2 polls for completion

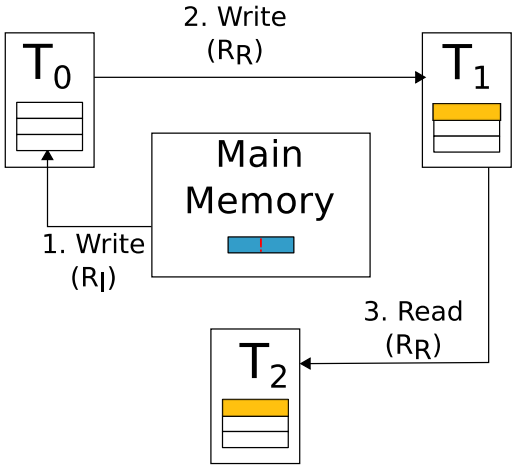


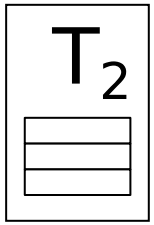


Min-Max Modeling

- **Example:**
  - T_0 + T_1 write CL
  - T_2 polls for completion

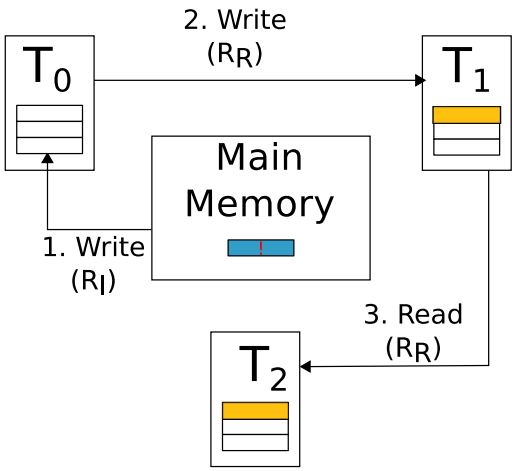


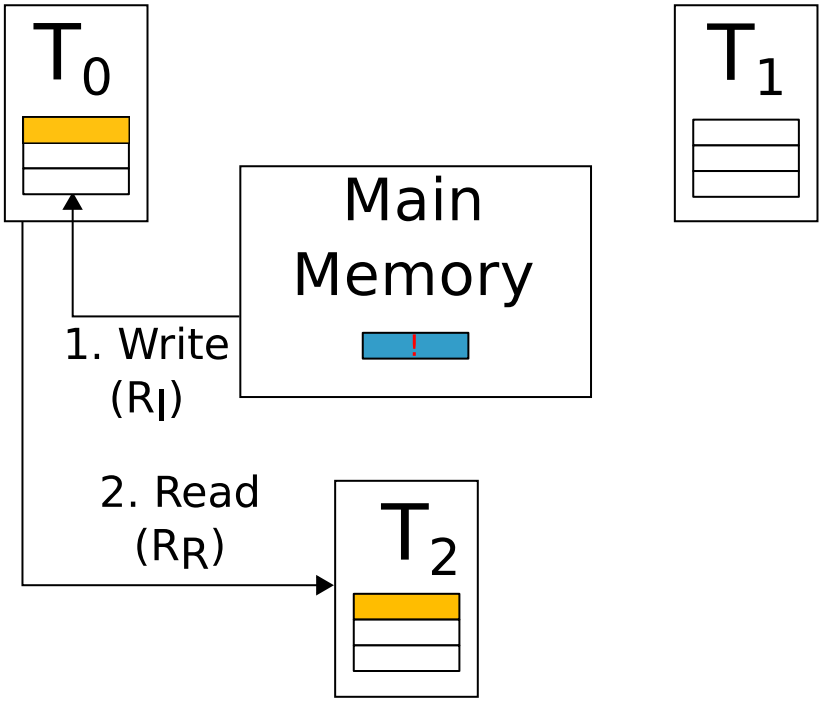




Min-Max Modeling

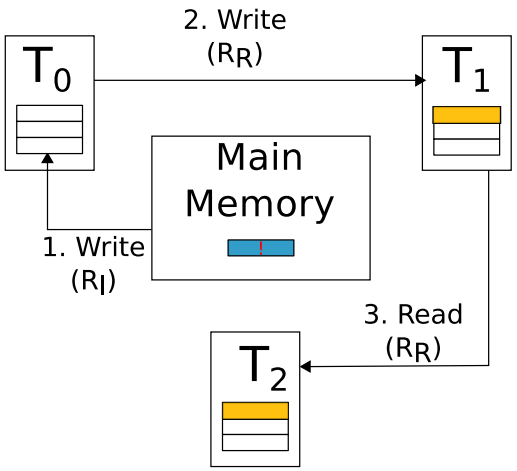
- **Example:**
  - T_0 + T_1 write CL
  - T_2 polls for completion

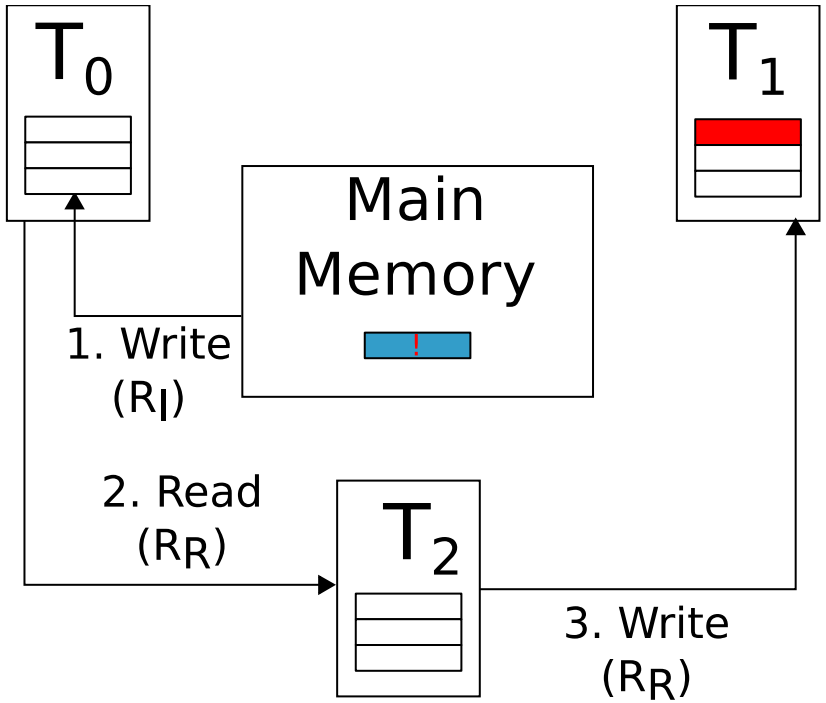




Min-Max Modeling

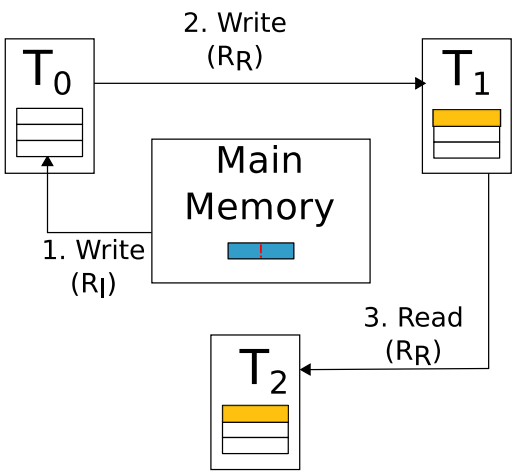
- **Example:**
  - T_0 + T_1 write CL
  - T_2 polls for completion

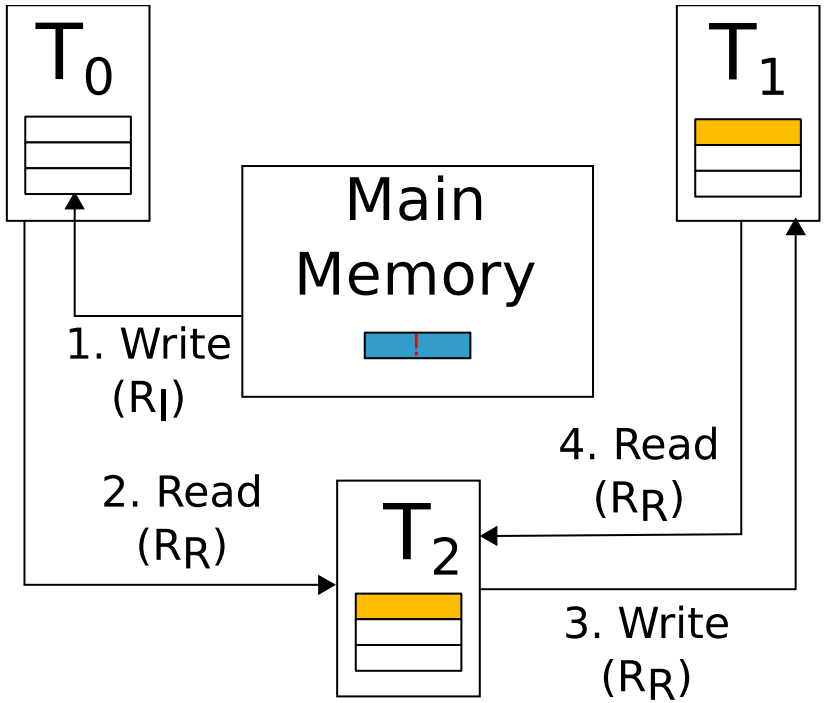




Min-Max Modeling

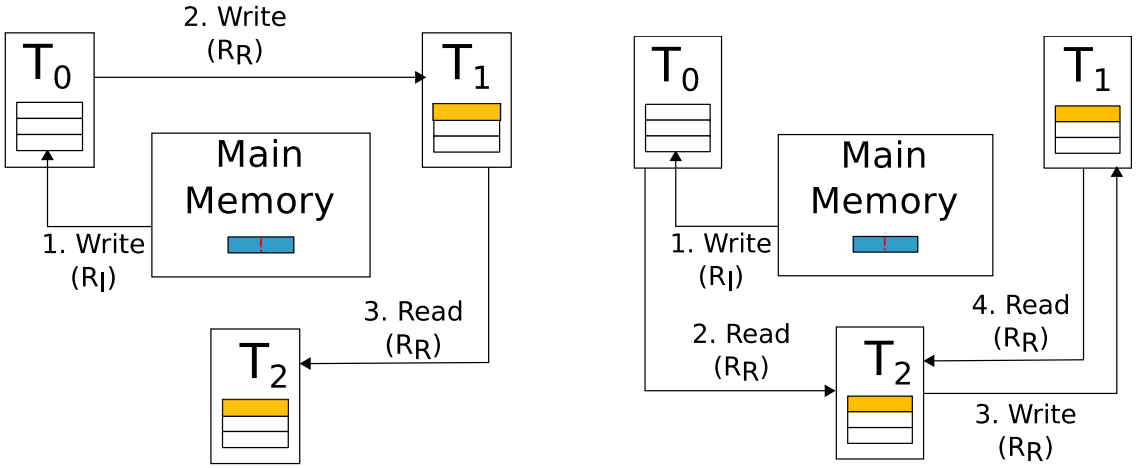
- **Example:**
  - T_0 + T_1 write CL
  - T_2 polls for completion





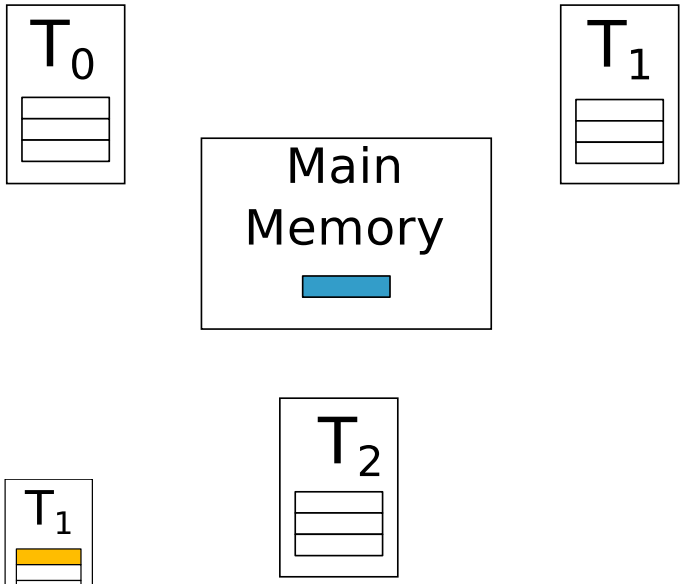
Min-Max Modeling

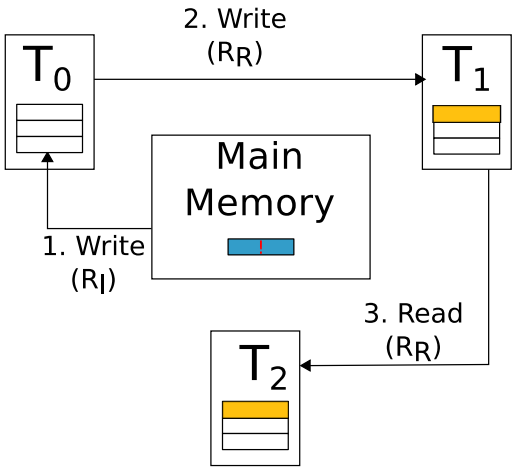
- **Example:**
  - T_0 + T_1 write CL
  - T_2 polls for completion

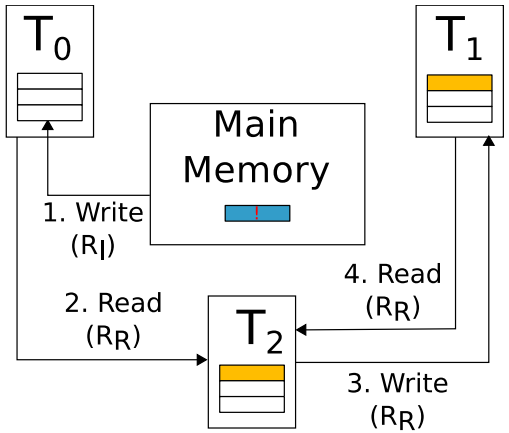


Min-Max Modeling

- **Example:**
  - T_0 + T_1 write CL
  - T_2 polls for completion

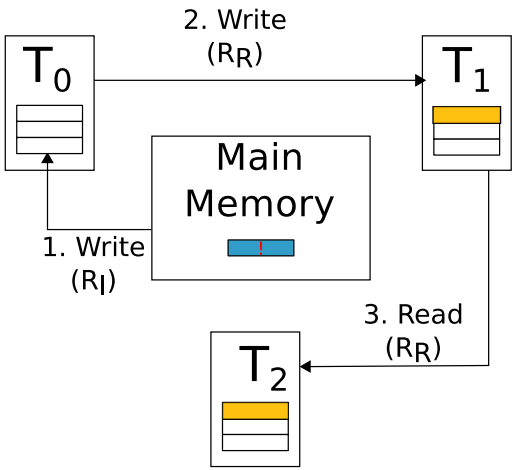


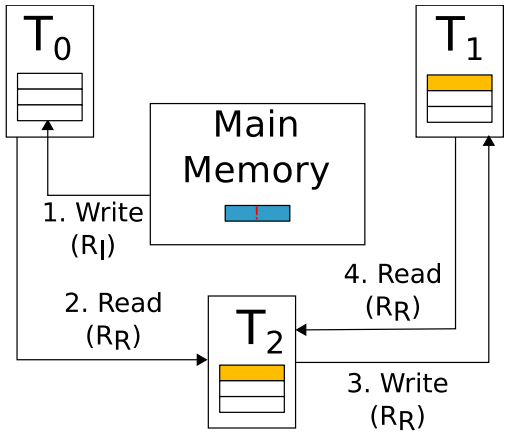


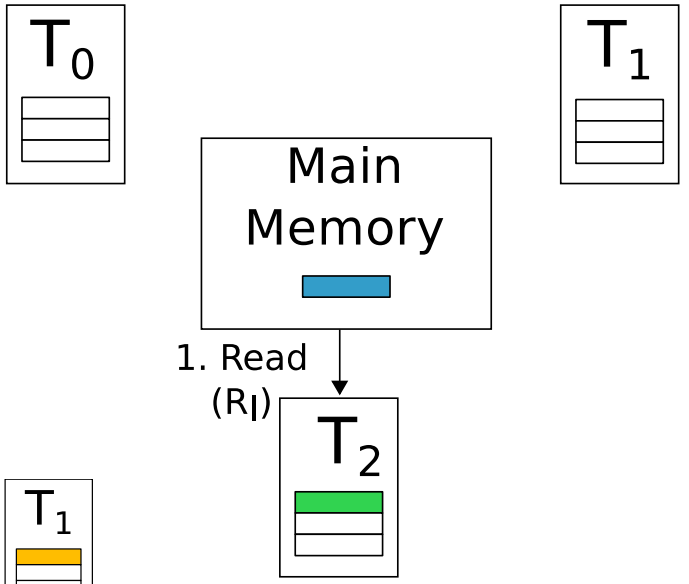


Min-Max Modeling

- **Example:**
  - T_0 + T_1 write CL
  - T_2 polls for completion

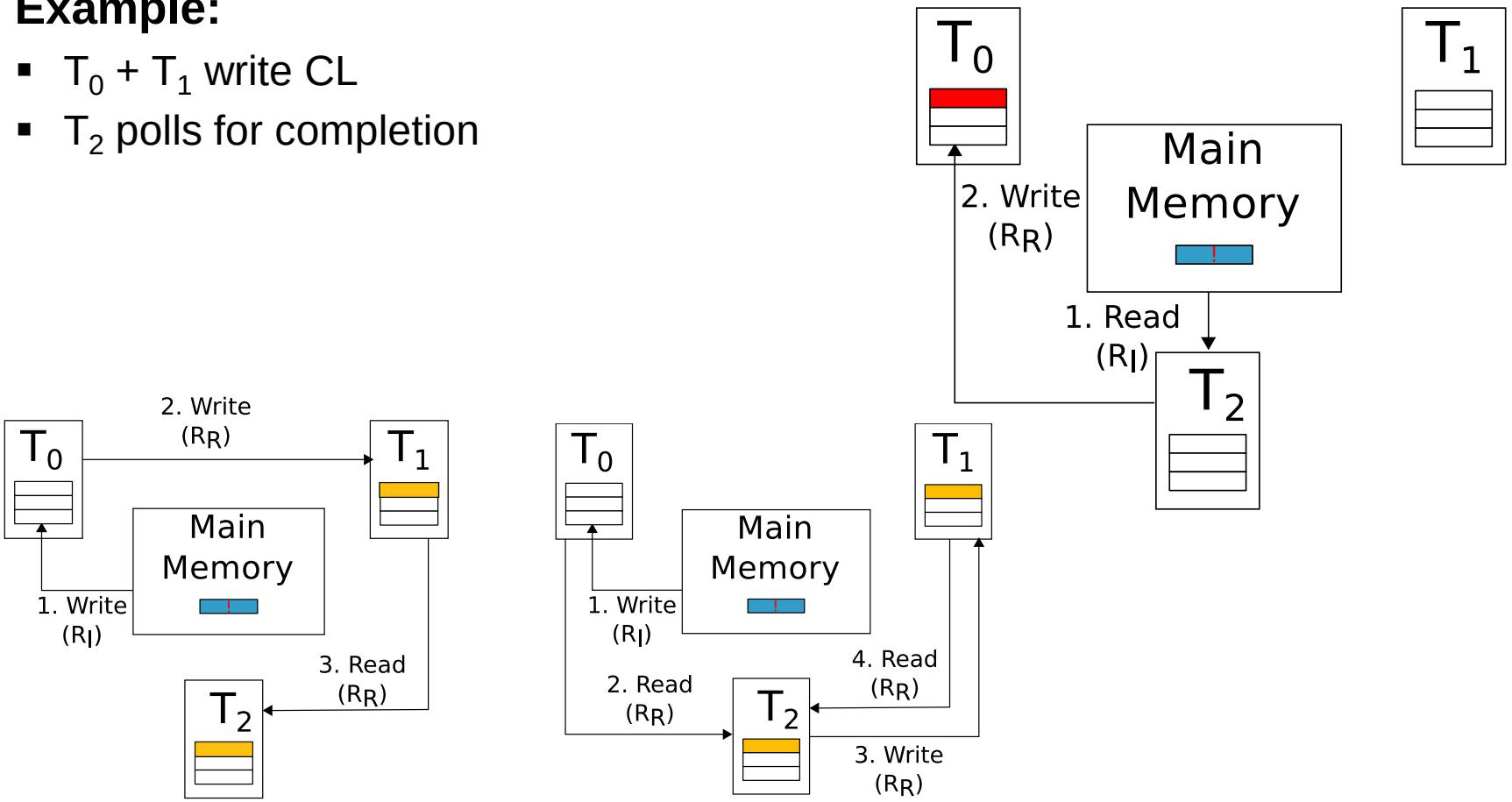






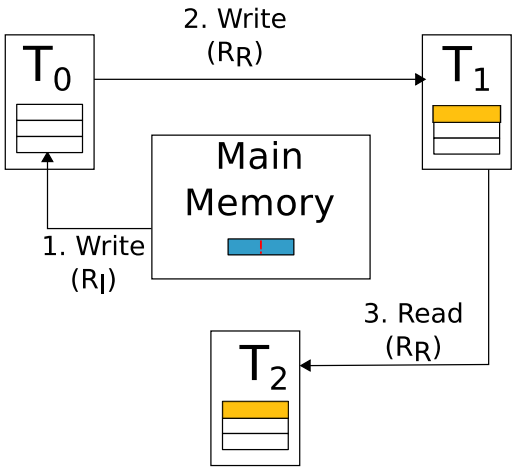
Min-Max Modeling

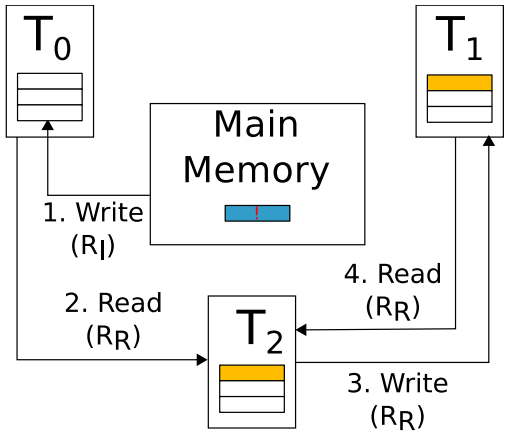
- **Example:**
  - T_0 + T_1 write CL
  - T_2 polls for completion

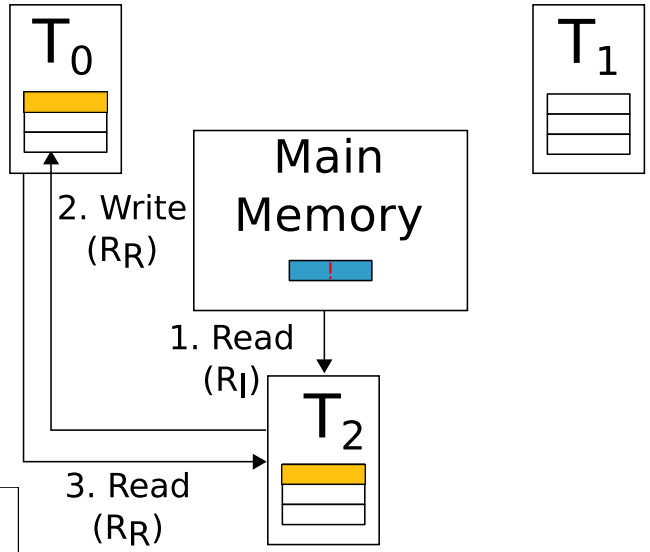


Min-Max Modeling

- **Example:**
  - T_0 + T_1 write CL
  - T_2 polls for completion

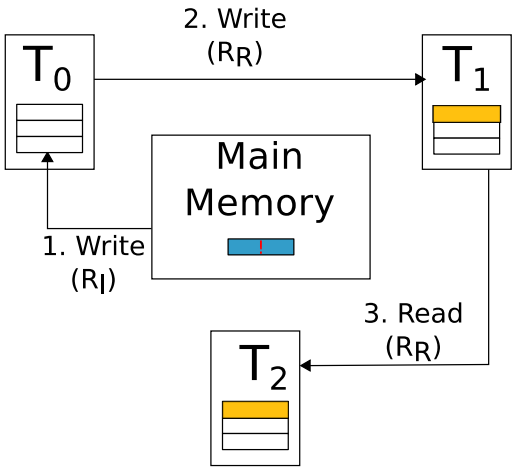


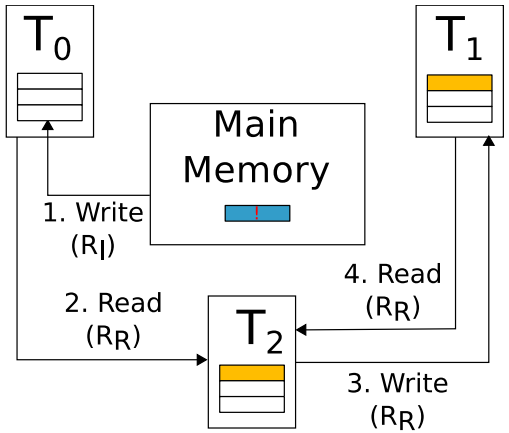


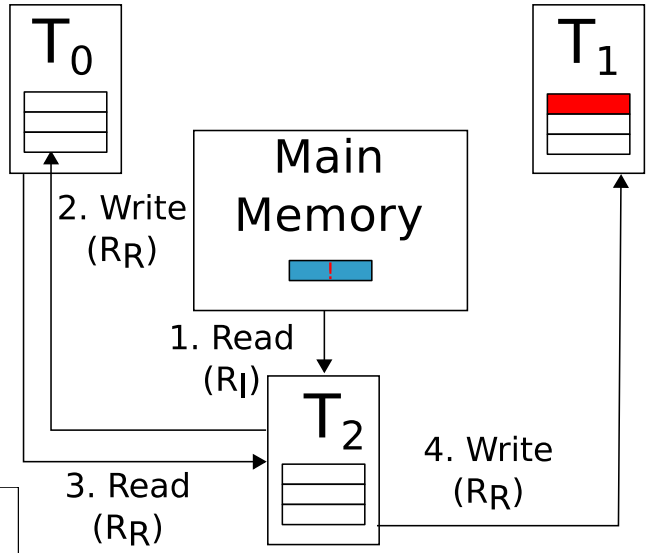


Min-Max Modeling

- **Example:**
  - T_0 + T_1 write CL
  - T_2 polls for completion

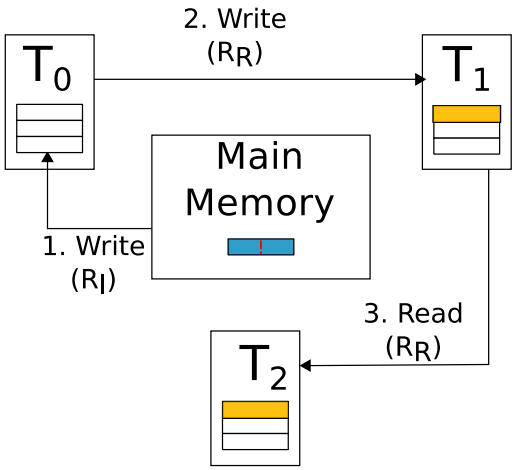


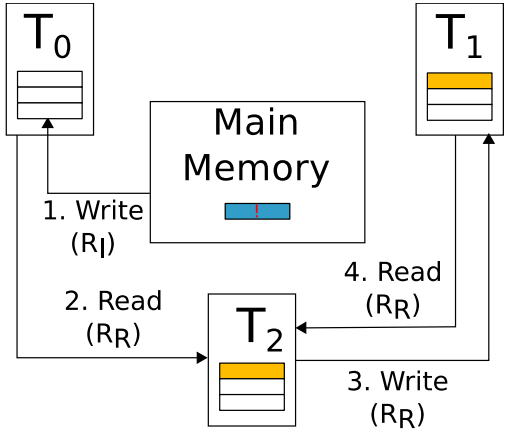


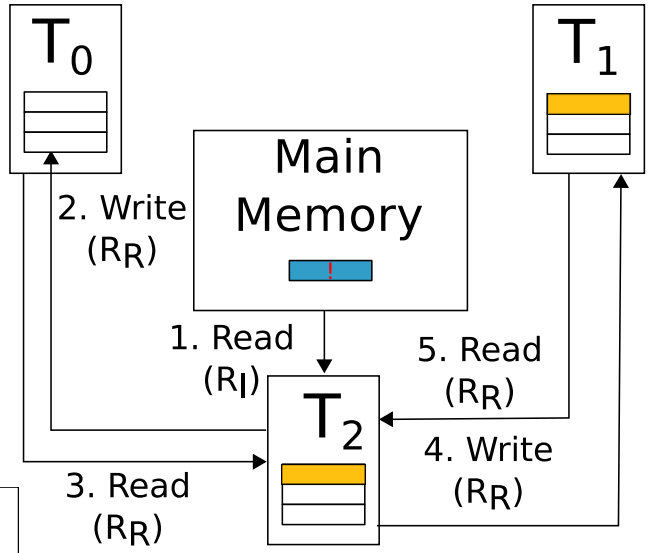


Min-Max Modeling

- **Example:**
  - T_0 + T_1 write CL
  - T_2 polls for completion

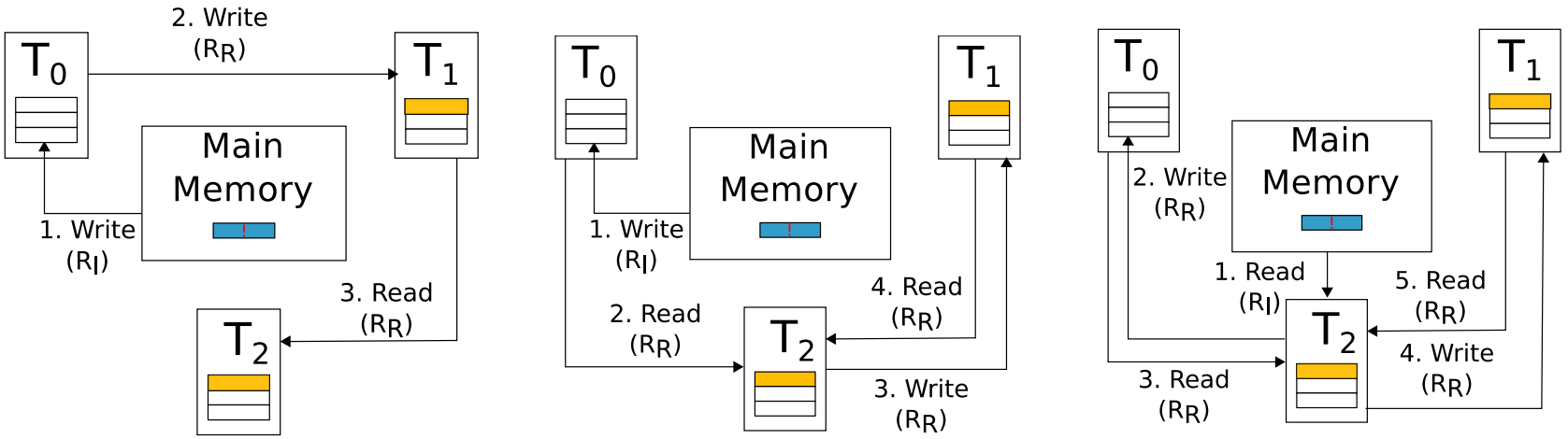




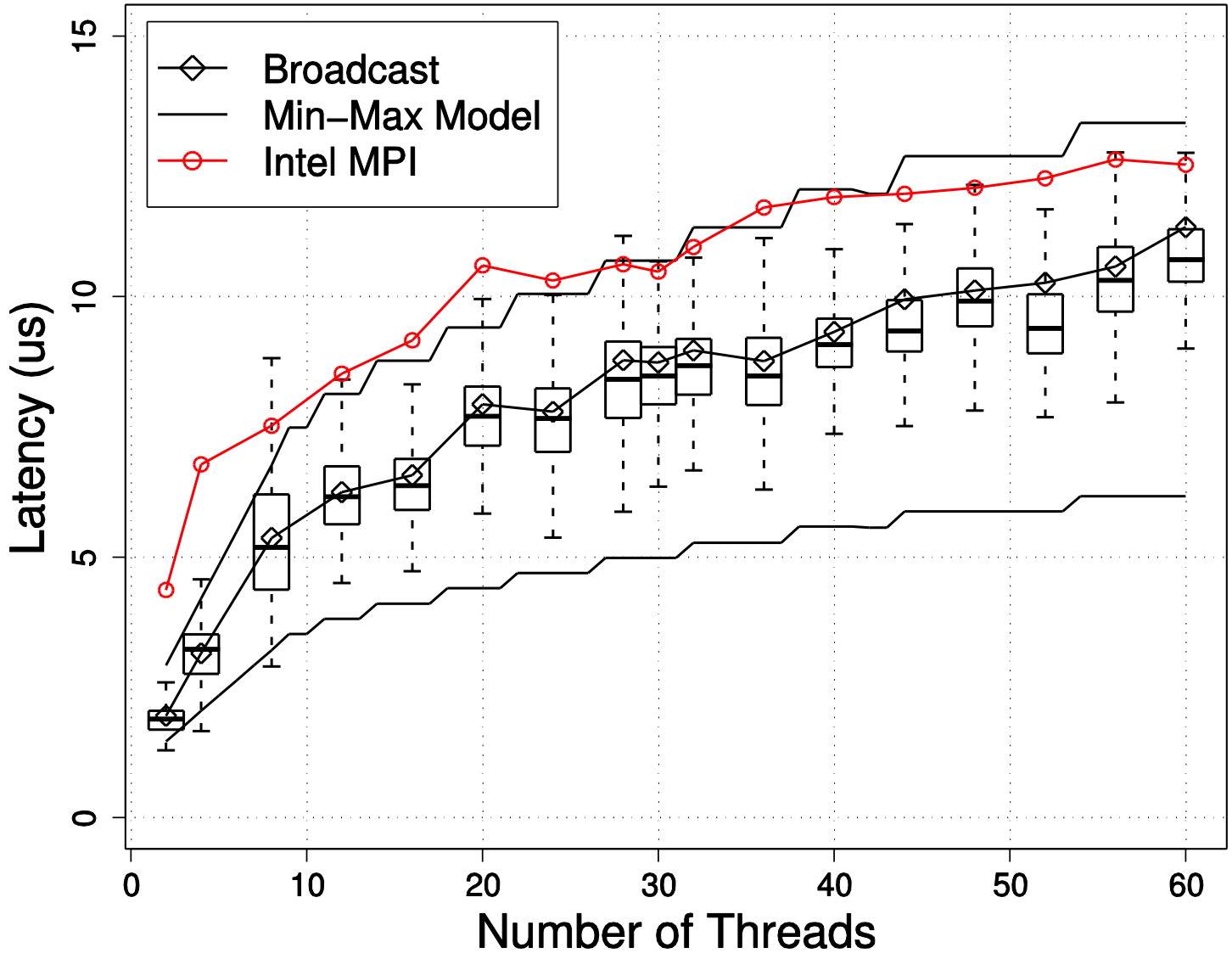


Min-Max Modeling

- **Example:**
  - T_0 + T_1 write CL
  - T_2 polls for completion

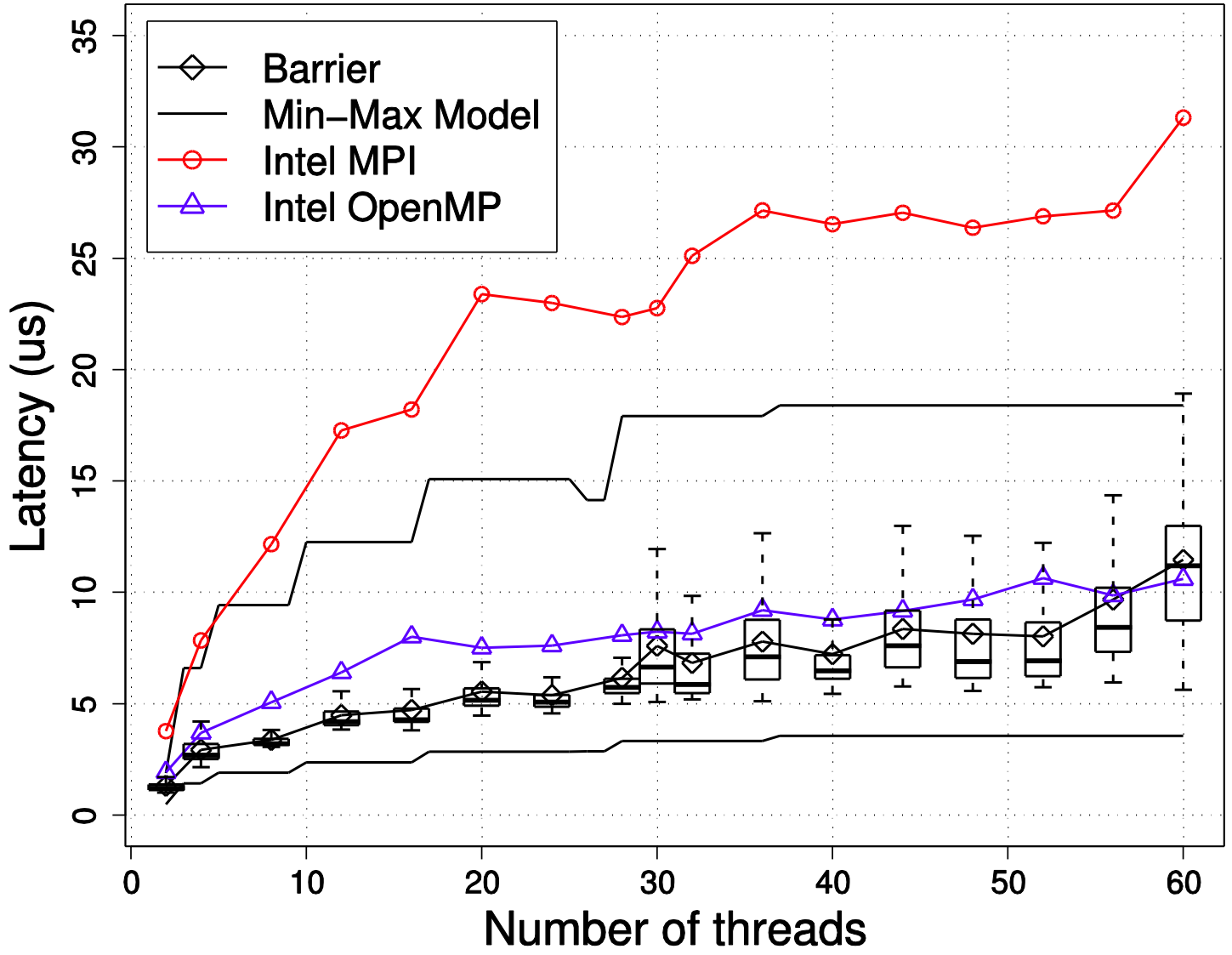


Small Broadcast (8 Bytes)



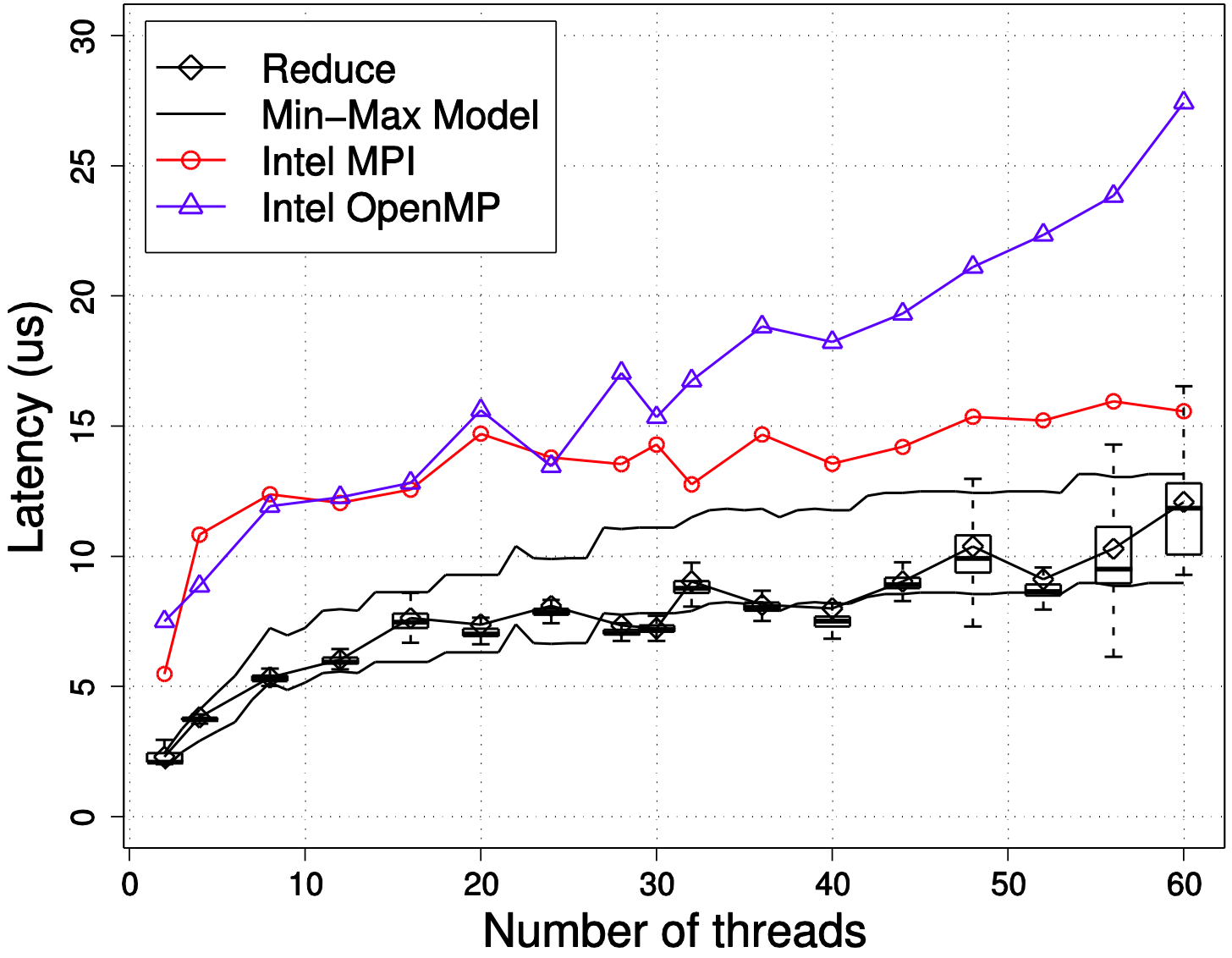
Ramos, Hoefler: "Modeling Communication in Cache-Coherent SMP Systems - A Case-Study with Xeon Phi ", HPDC'13

Barrier



Ramos, Hoefler: "Modeling Communication in Cache-Coherent SMP Systems - A Case-Study with Xeon Phi ", HPDC'13

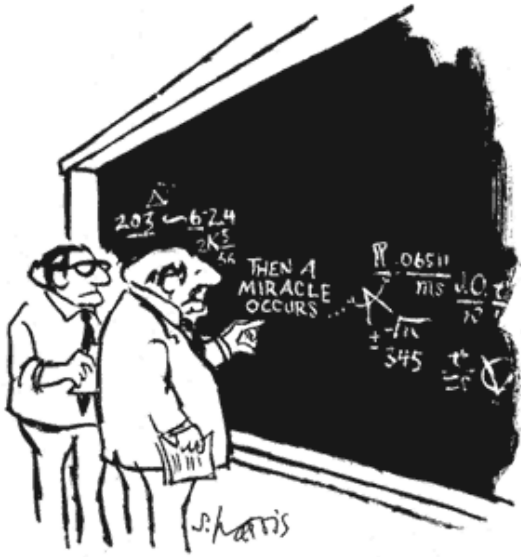
Small Reduction



Ramos, Hoefler: "Modeling Communication in Cache-Coherent SMP Systems - A Case-Study with Xeon Phi ", HPDC'13

Lessons learned

- **Rigorous modeling has large potential**
  - Coming with great cost (working on tool support [1])
- **Understanding cache-coherent communication performance is incredibly complex (but fun)!**
  - Many states, min-max modeling, NUMA, …
  - Have models for Sandy Bridge now (QPI, worse!)
    → *up to 10x improvements*
- **Cache coherence really gets in our way here ☹**



"I THINK YOU SHOULD BE MORE EXPLICIT HERE IN STEP TWO."

- **Obvious question: why do we need cache coherence?**
  - Answer: well, we don't! If we program right!
  - One option: Remote Memory Access (RMA) programming [2]

[1]: Calotoiu et al.: Using Automated Performance Modeling to Find Scalability Bugs in Complex Codes, SC13
[2]: Gerstenberger et al.: Enabling Highly-Scalable Remote Memory Access Programming with MPI-3 One Sided, SC13, Best Paper