

# Bottleneck Identification and Scheduling in Multithreaded Applications

ASPLOS XVII, March 2012

Authors: José A. Joao, M. Aater Suleman, Onur Mutlu, Yale N. Patt

Presenter: Roman Meier

04.11.2021

# Executive Summary

---

- Problem: Multi-threaded programs contain «bottlenecks»
  - Bottlenecks force execution to be serialized
  - Bottlenecks can vary in importance over time
- Goal: Identify bottlenecks & accelerate the most critical bottlenecks using fast cores on an Asymmetric Chip Multiprocessor (ACMP)
- Solution: Cooperative hardware/software “Bottleneck Identification & Scheduling” **BIS**
  - Use special instructions to mark bottlenecks in software
  - Accelerate most critical bottlenecks at runtime in hardware by scheduling them on large cores in an ACMP system
  - Outperforms previous approaches by 15%

# Outline

---

- Background
- Previous Work
- Bottleneck Identification and Scheduling
  - Bottleneck Identification
  - Bottleneck Acceleration
- Improvements & Details
- Evaluation
  
- Critique

# Outline

---

- Background
- Previous Work
- Bottleneck Identification and Scheduling
  - Bottleneck Identification
  - Bottleneck Acceleration
- Improvements & Details
- Evaluation
  
- Critique

# Types of Bottlenecks

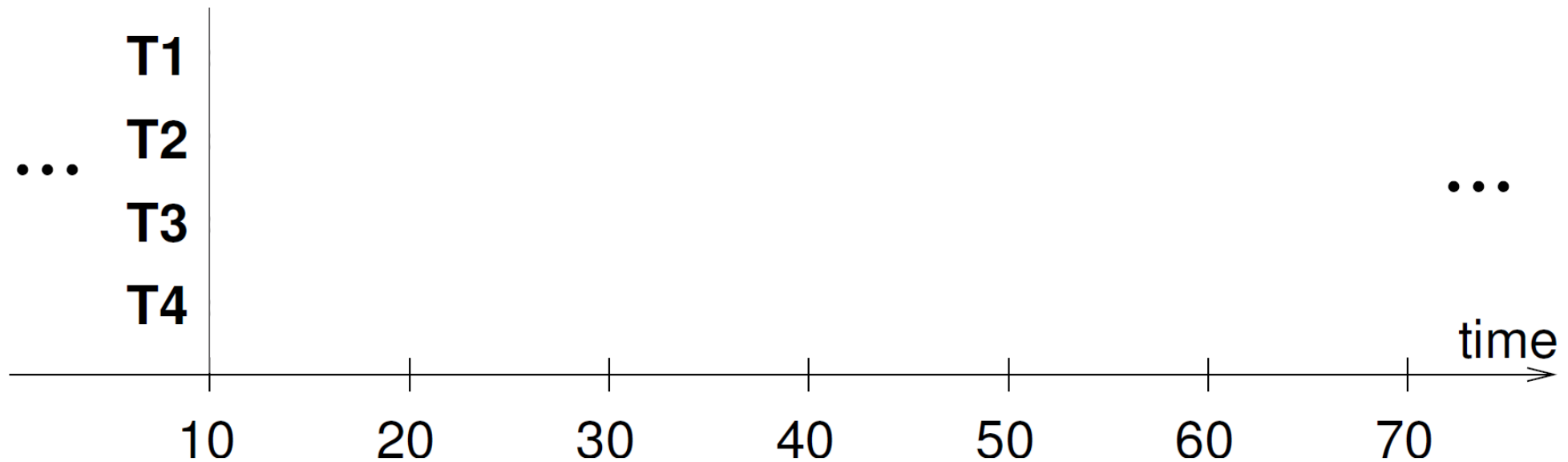
---

- Amdahl's serial portions
  - Sections of the program with only one thread
- Critical Sections
- Barriers
- Pipeline Stages

# Critical Sections

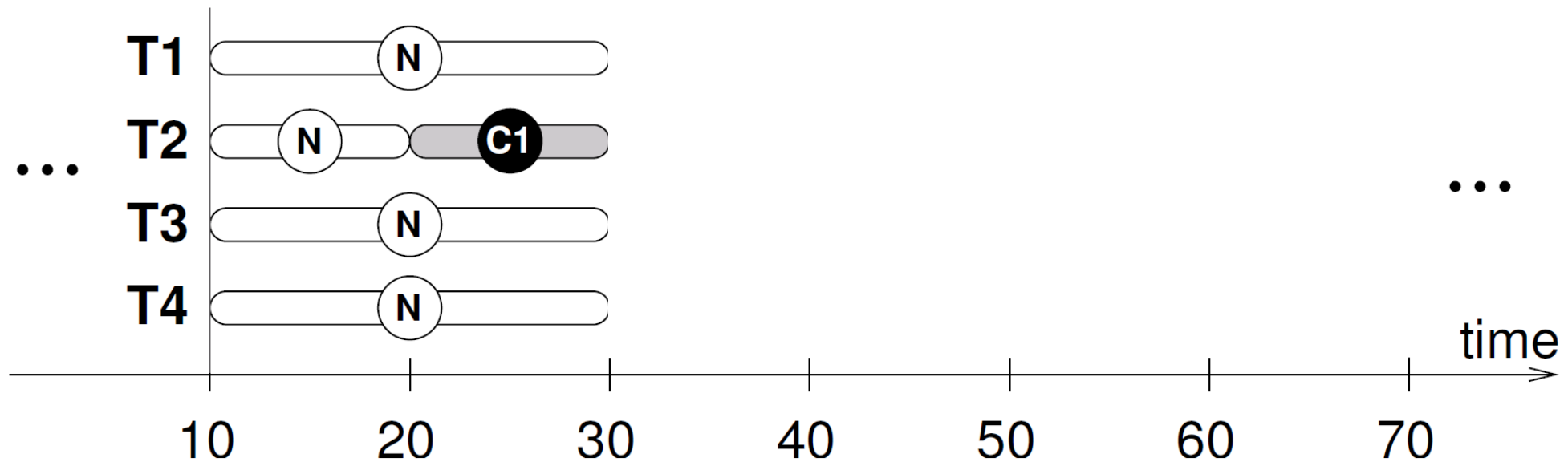
---

- Only one thread may enter the Critical Section at any time



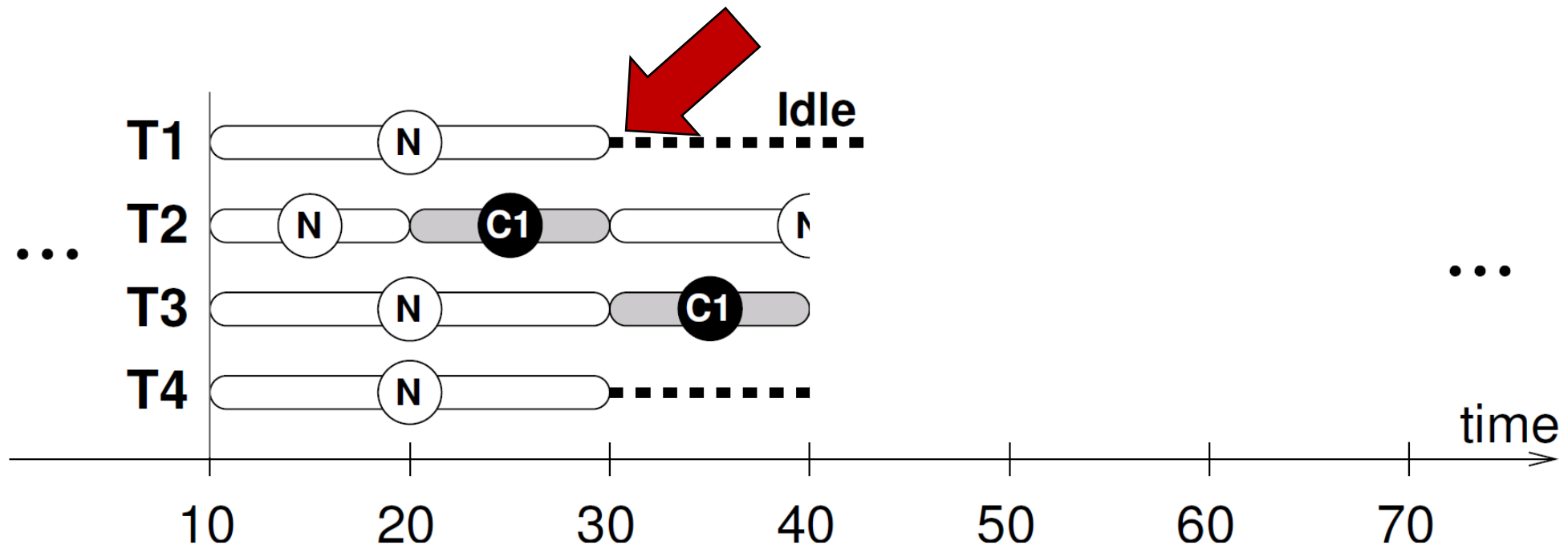
# Critical Sections

- Only one thread may enter the Critical Section at any time



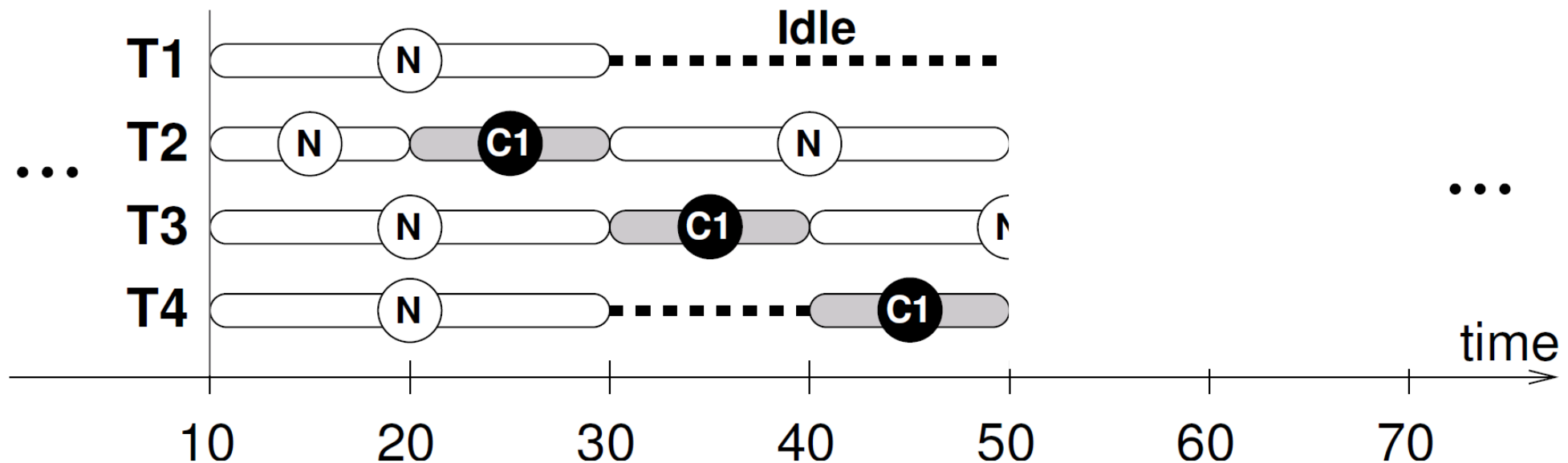
# Critical Sections

- Only one thread may enter the Critical Section at any time



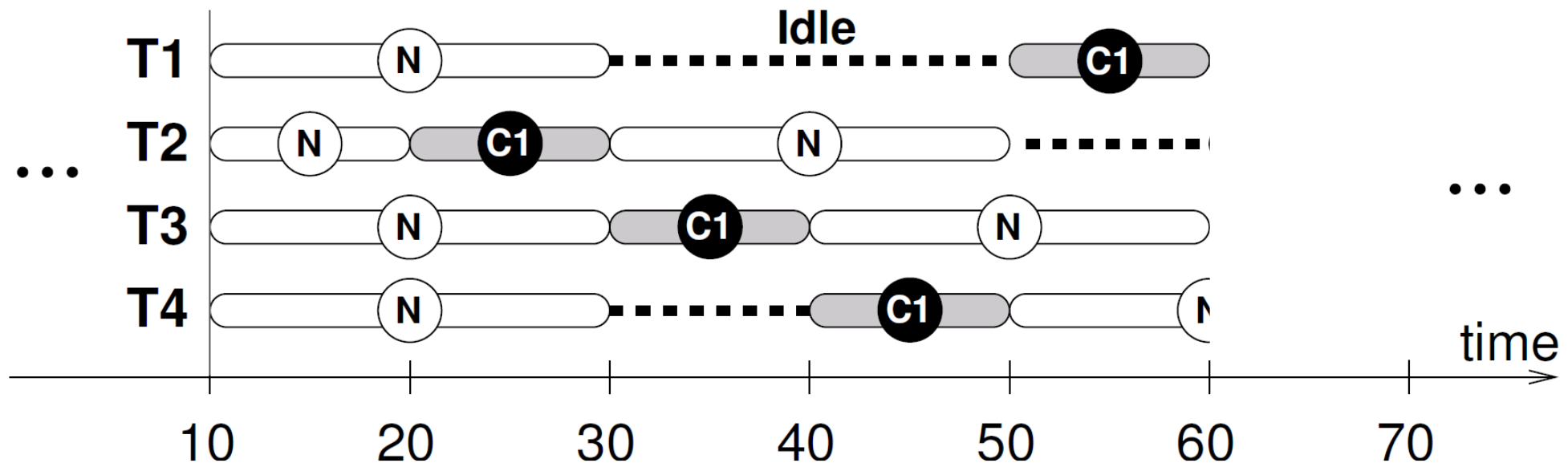
# Critical Section

- Only one thread may enter the Critical Section at any time



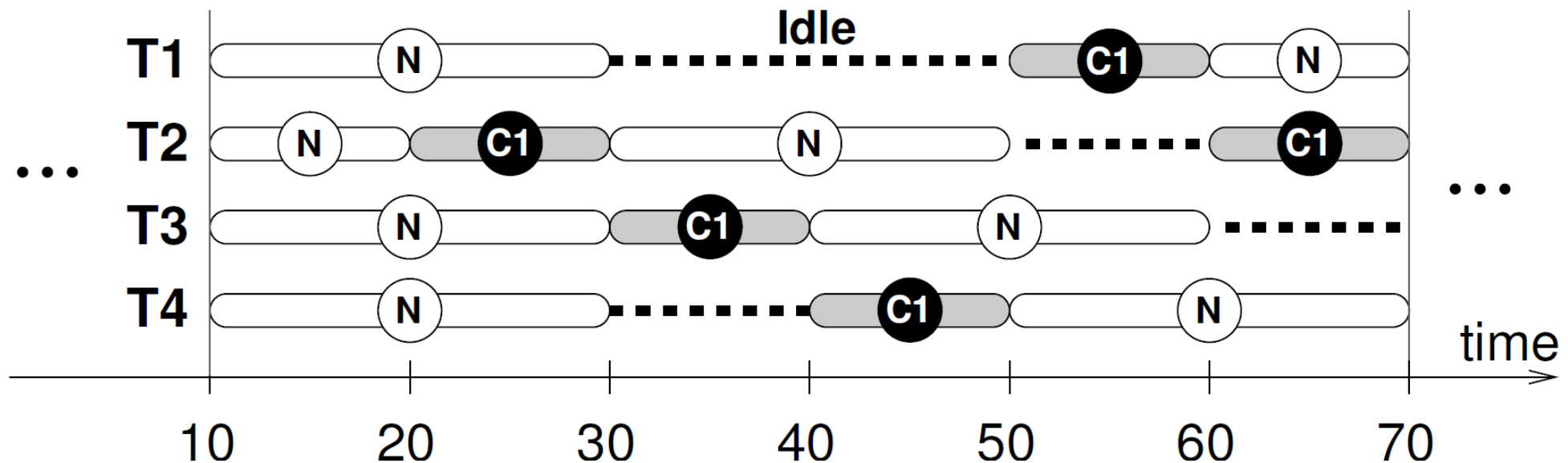
# Critical Section

- Only one thread may enter the Critical Section at any time



# Critical Sections (CT)

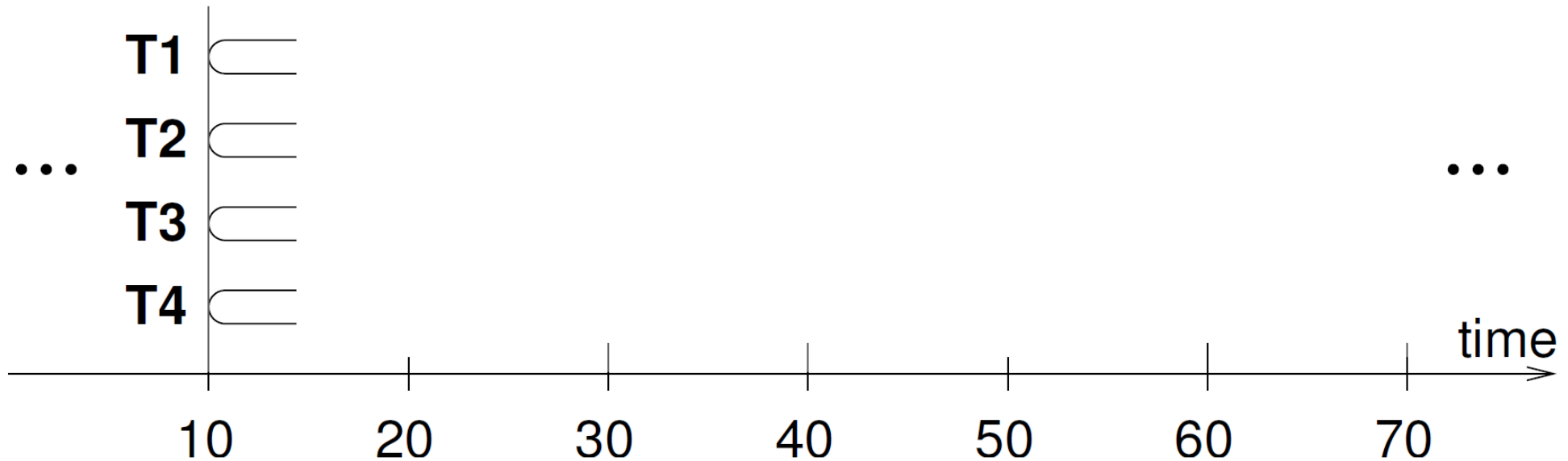
- Idea: Execute Critical Section faster than the rest



# Barriers

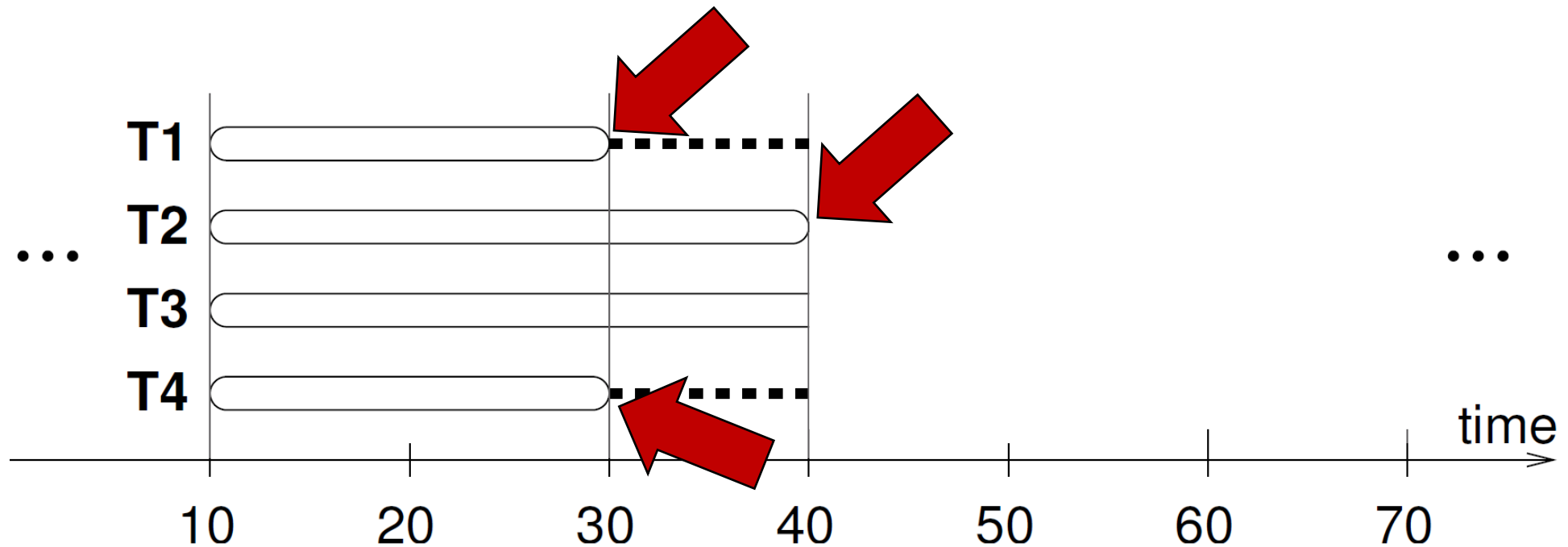
---

- Ensure that all threads synchronize before proceeding



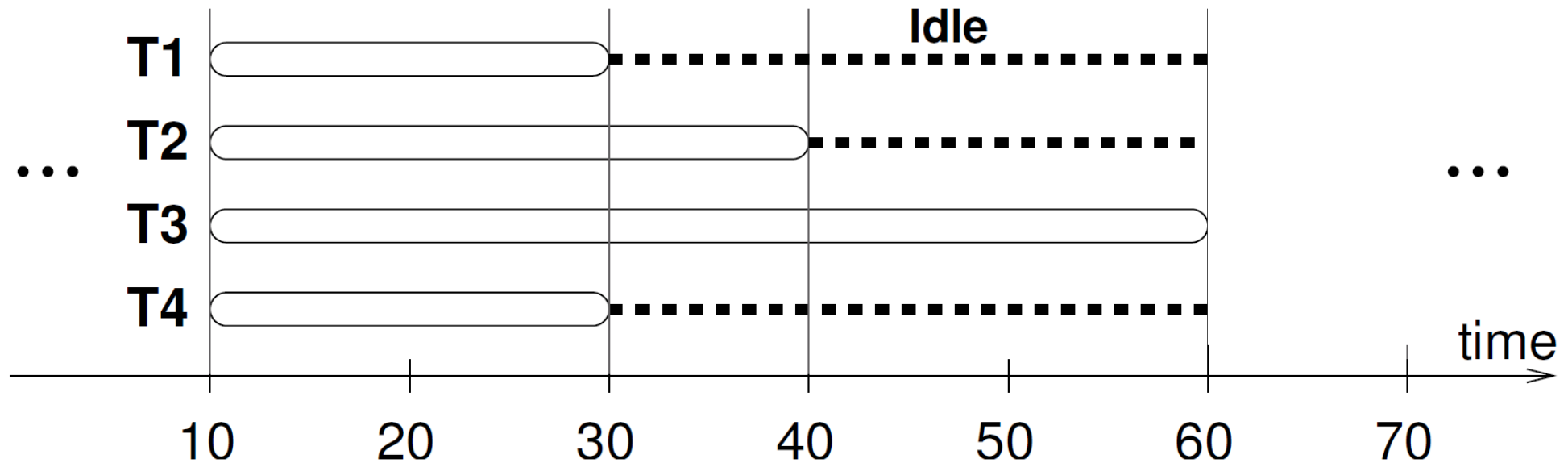
# Barriers

- Threads may not finish at the same time



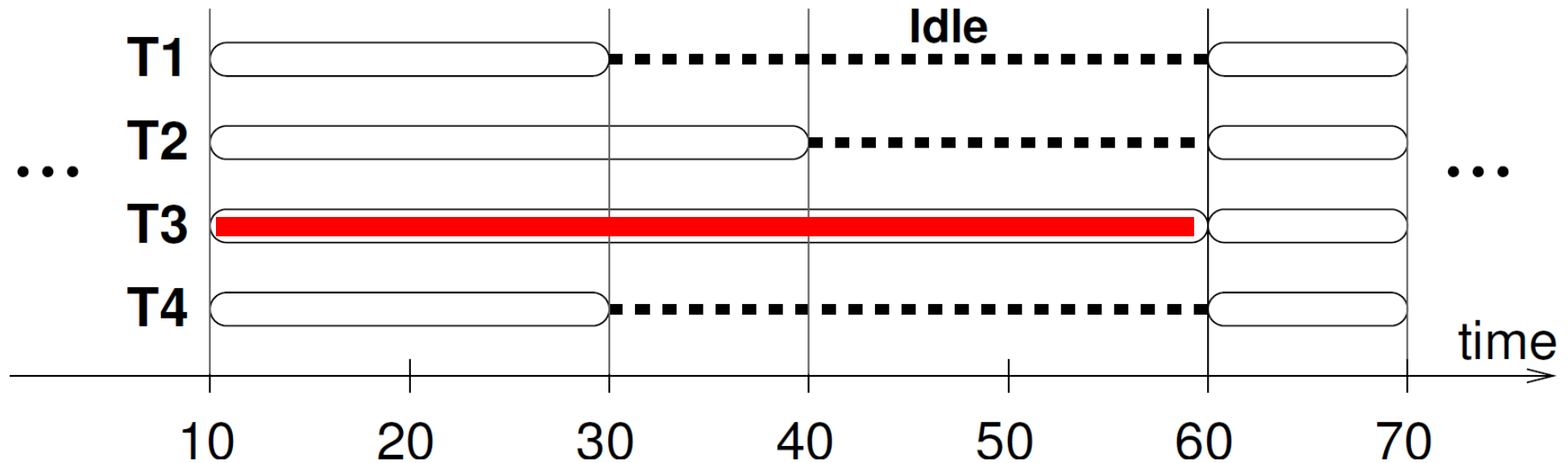
# Barriers

- Result: Wasted time on all threads that finish early



# Barriers

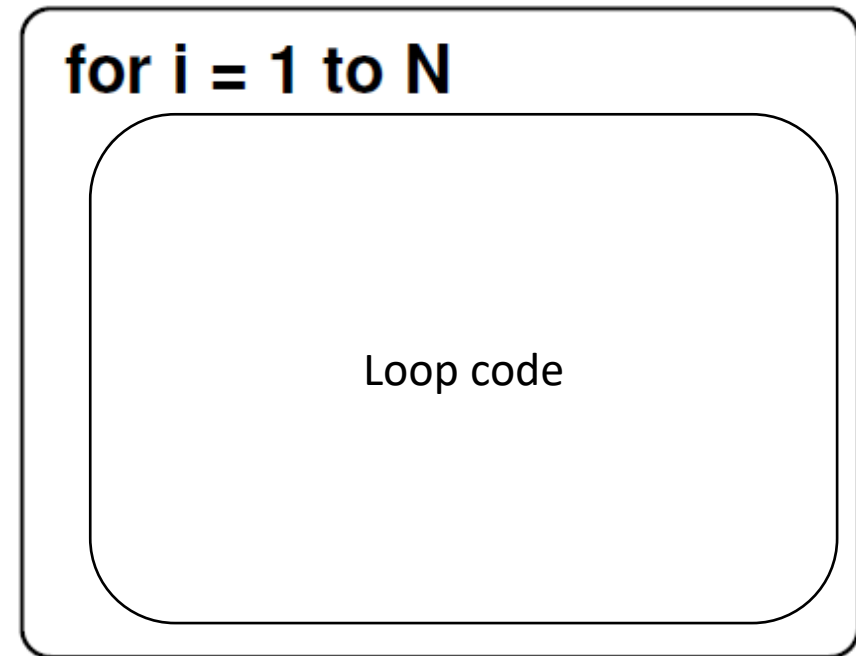
- Idea: Speed up the slowest thread



# Pipelining

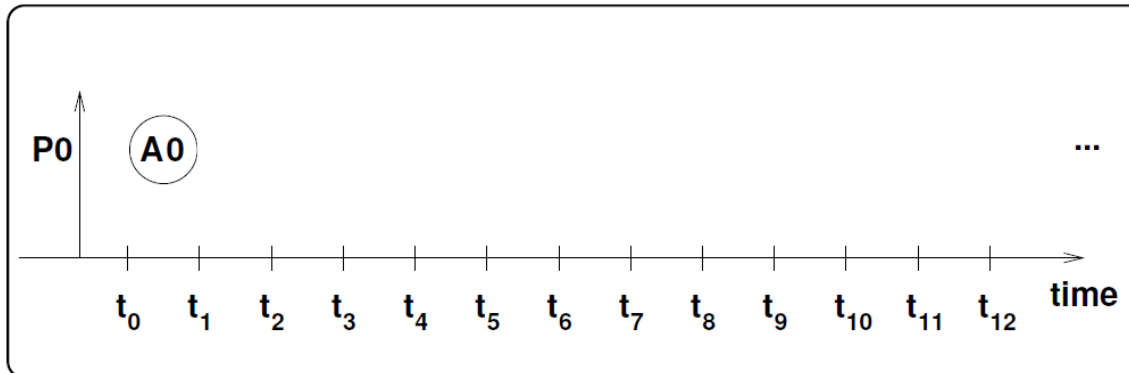
---

- Programming Paradigm to **parallelize for-loops** and similar
- Code in loop **split into  $M$  stages**



# Pipelining

- Programming Paradigm to **parallelize for-loops** and similar
- Code in loop **split into  $M$  stages**



**for  $i = 1$  to  $N$**

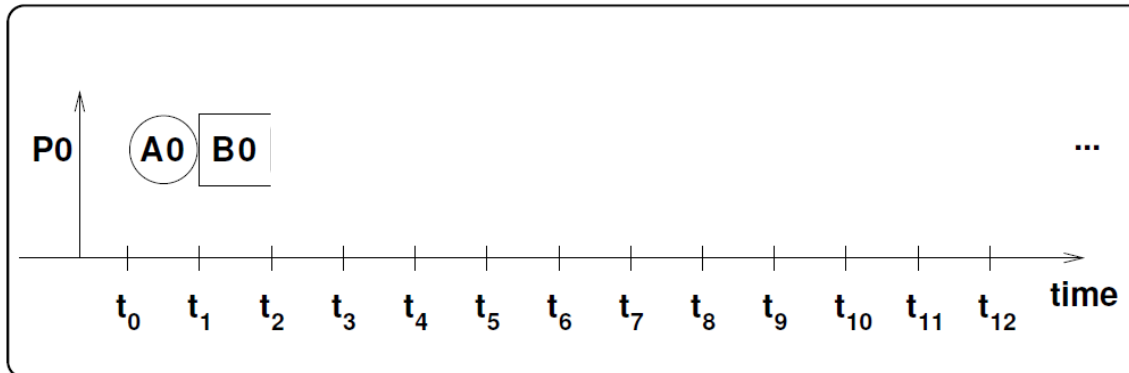
**... // code in stage A**

**... // code in stage B**

**... // code in stage C**

# Pipelining

- Programming Paradigm to **parallelize for-loops** and similar
- Code in loop **split into  $M$  stages**



**for  $i = 1$  to  $N$**

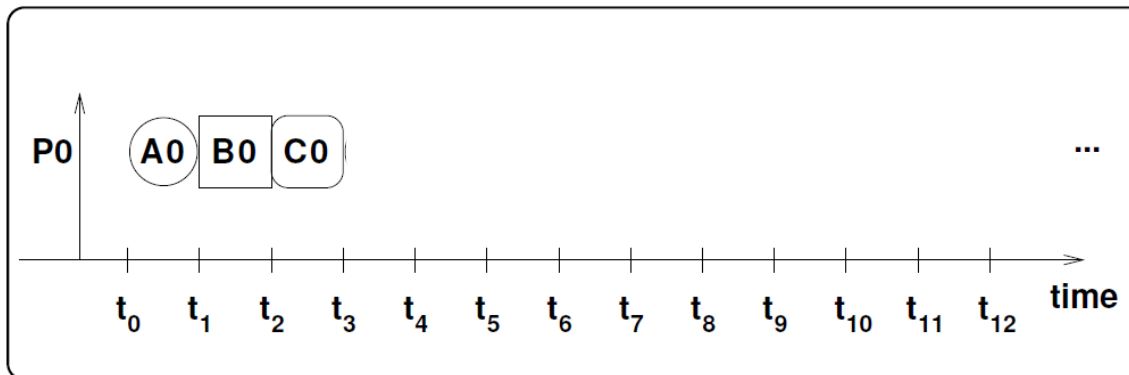
**... // code in stage A**

**... // code in stage B**

**... // code in stage C**

# Pipelining

- Programming Paradigm to **parallelize for-loops** and similar
- Code in loop **split into  $M$  stages**



**for i = 1 to N**

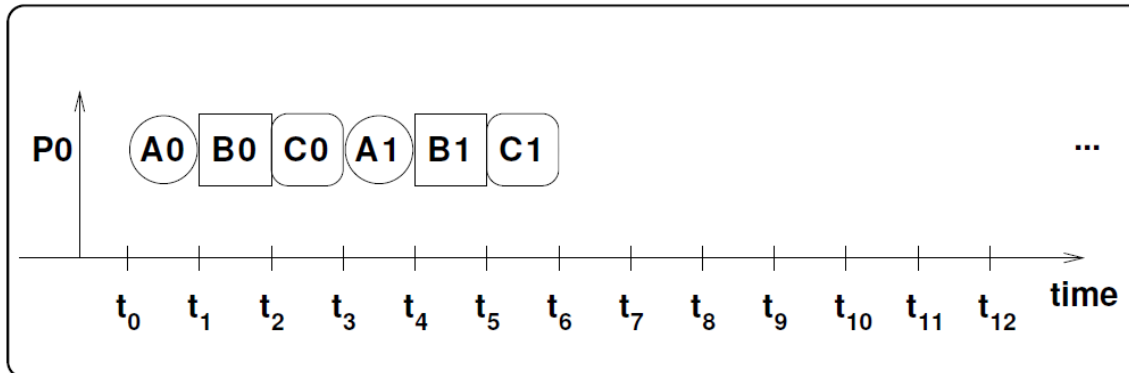
**... // code in stage A**

**... // code in stage B**

**... // code in stage C**

# Pipelining

- Programming Paradigm to **parallelize for-loops** and similar
- Code in loop **split into  $M$  stages**



**for  $i = 1$  to  $N$**

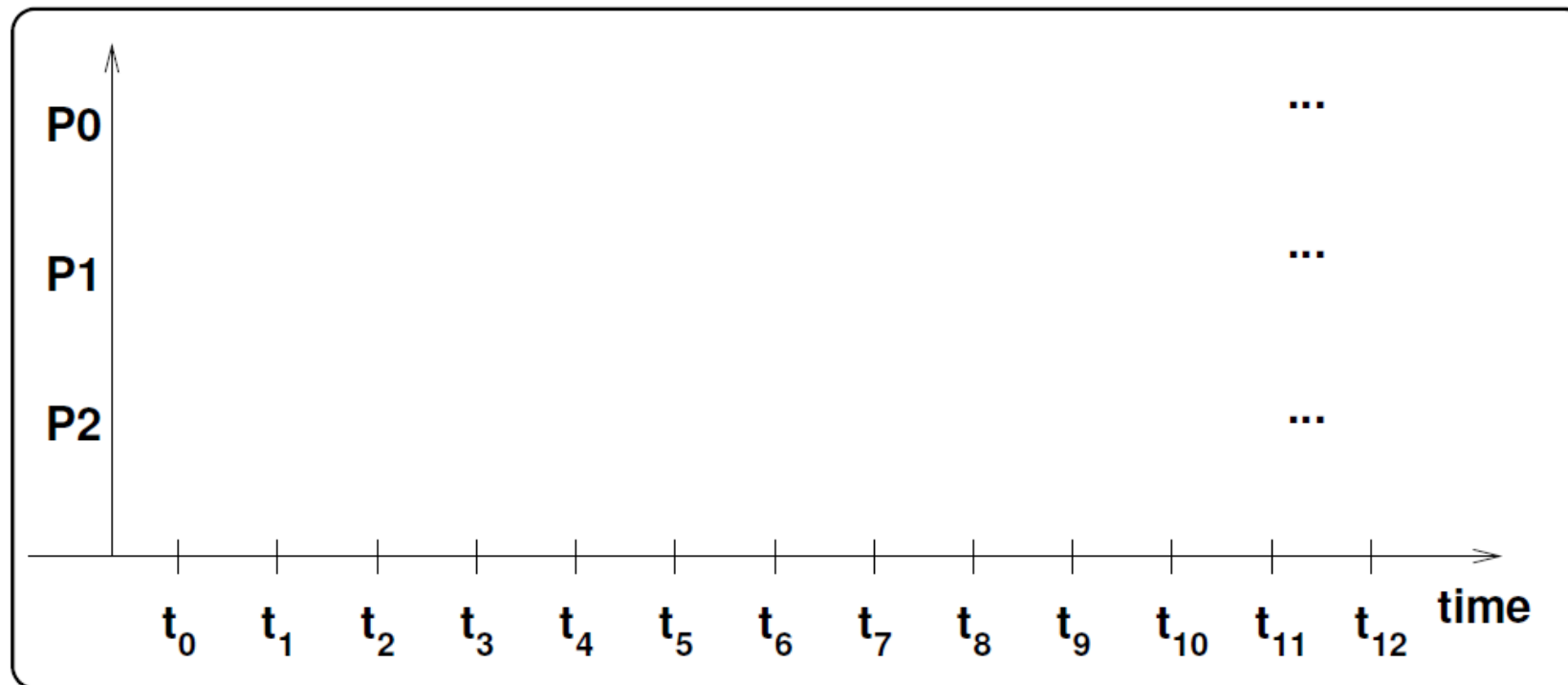
**... // code in stage A**

**... // code in stage B**

**... // code in stage C**

# Pipelining – Ideal Parallel Scenario

- Idea: Run the stages in parallel



for  $i = 1$  to  $N$

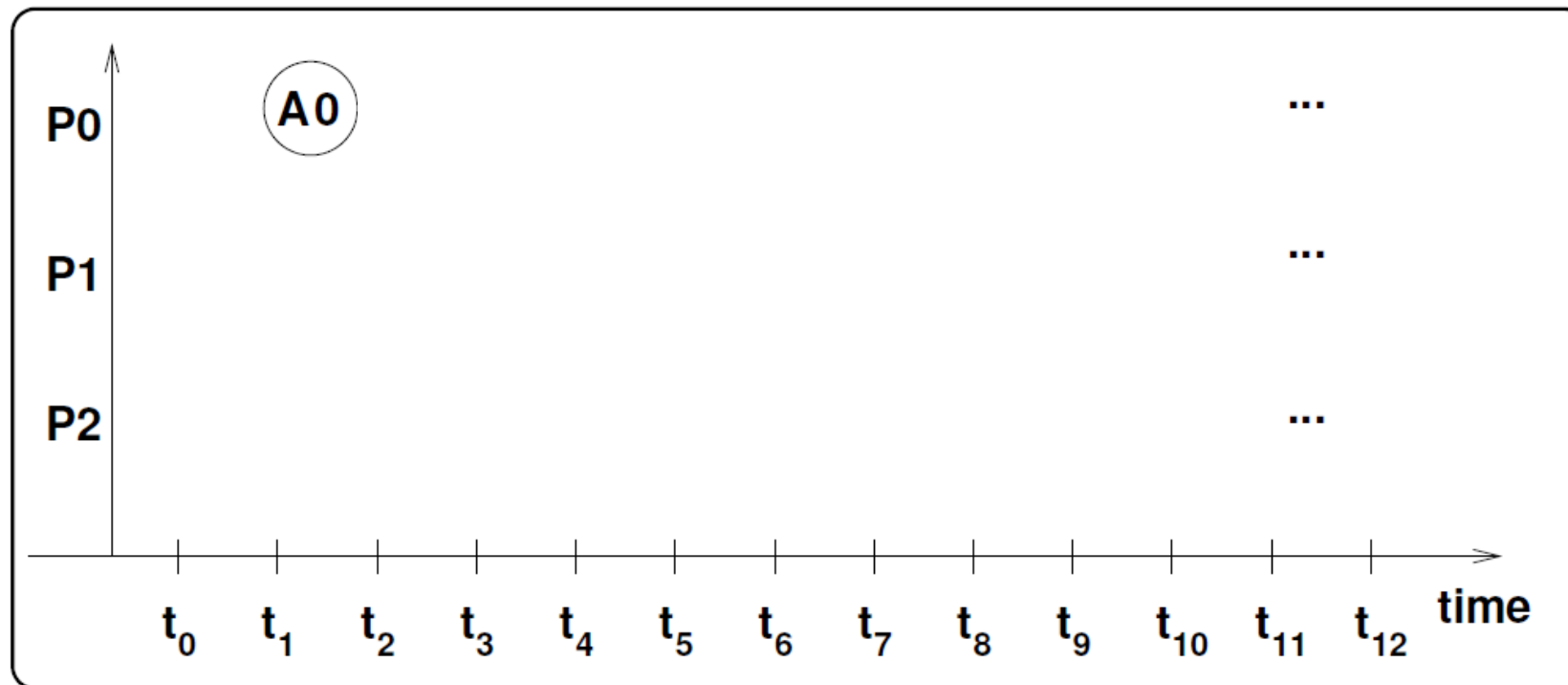
... // code in stage A

... // code in stage B

... // code in stage C

# Pipelining – Ideal Parallel Scenario

- Idea: Run the stages in parallel



for  $i = 1$  to  $N$

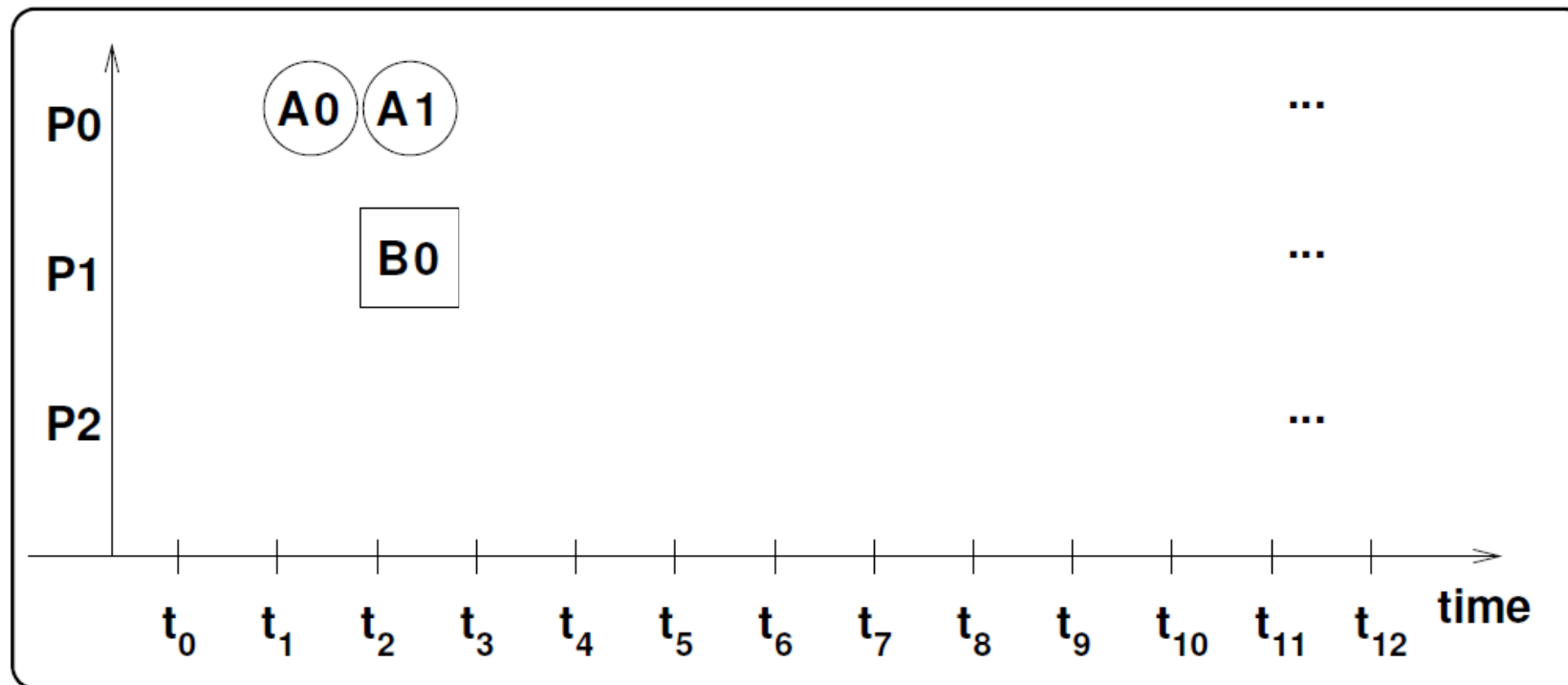
... // code in stage A

... // code in stage B

... // code in stage C

# Pipelining – Ideal Parallel Scenario

- Idea: Run the stages in parallel



for  $i = 1$  to  $N$

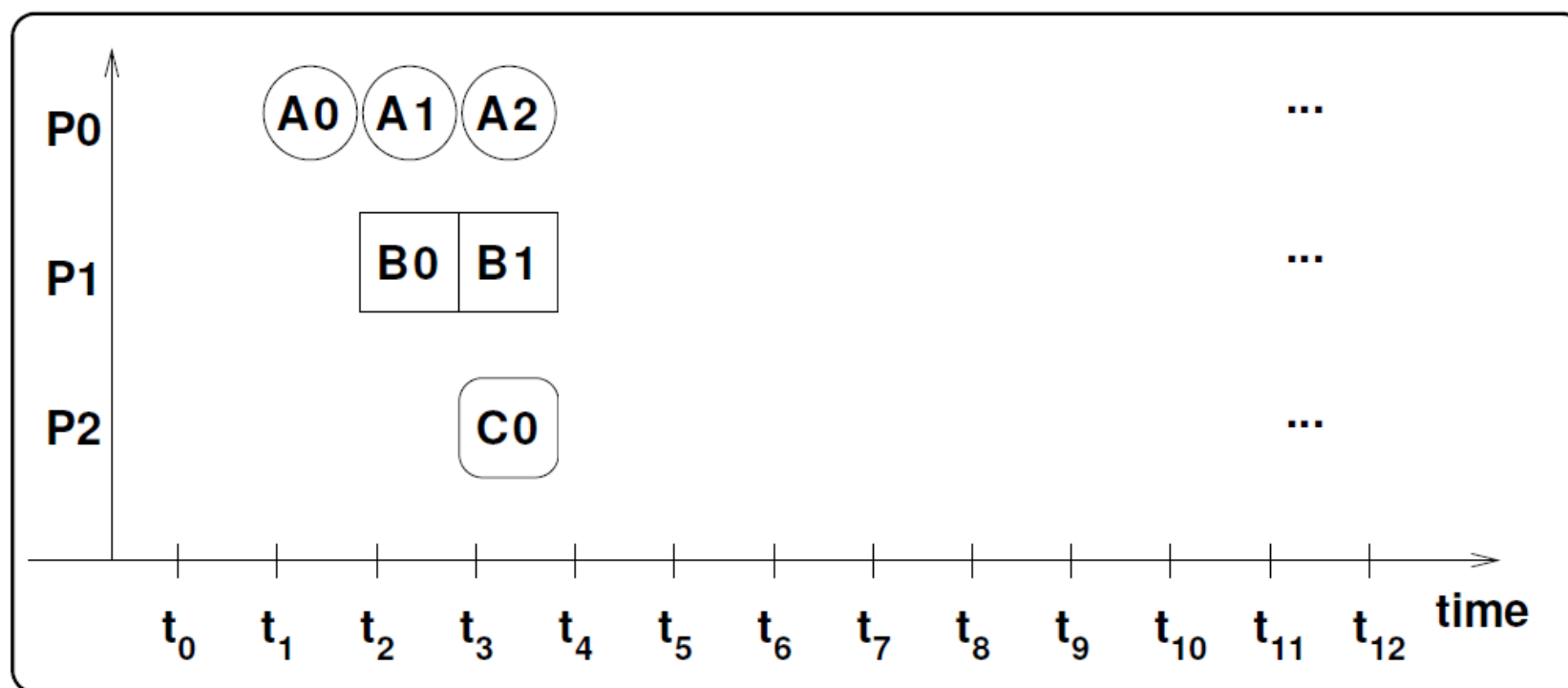
... // code in stage A

... // code in stage B

... // code in stage C

# Pipelining – Ideal Parallel Scenario

- Idea: Run the stages in parallel



for i = 1 to N

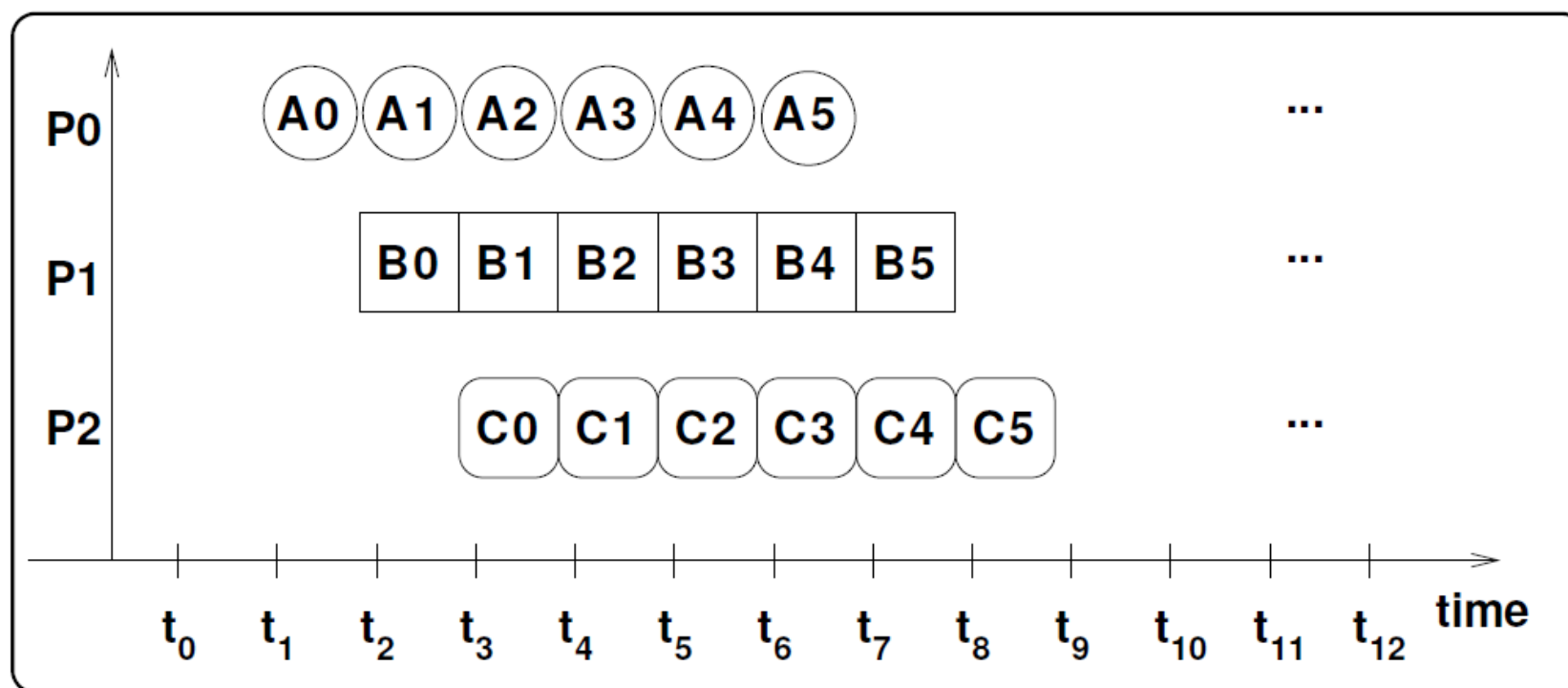
... // code in stage A

... // code in stage B

... // code in stage C

# Pipelining – Ideal Parallel Scenario

- Run the stages in parallel



for i = 1 to N

... // code in stage A

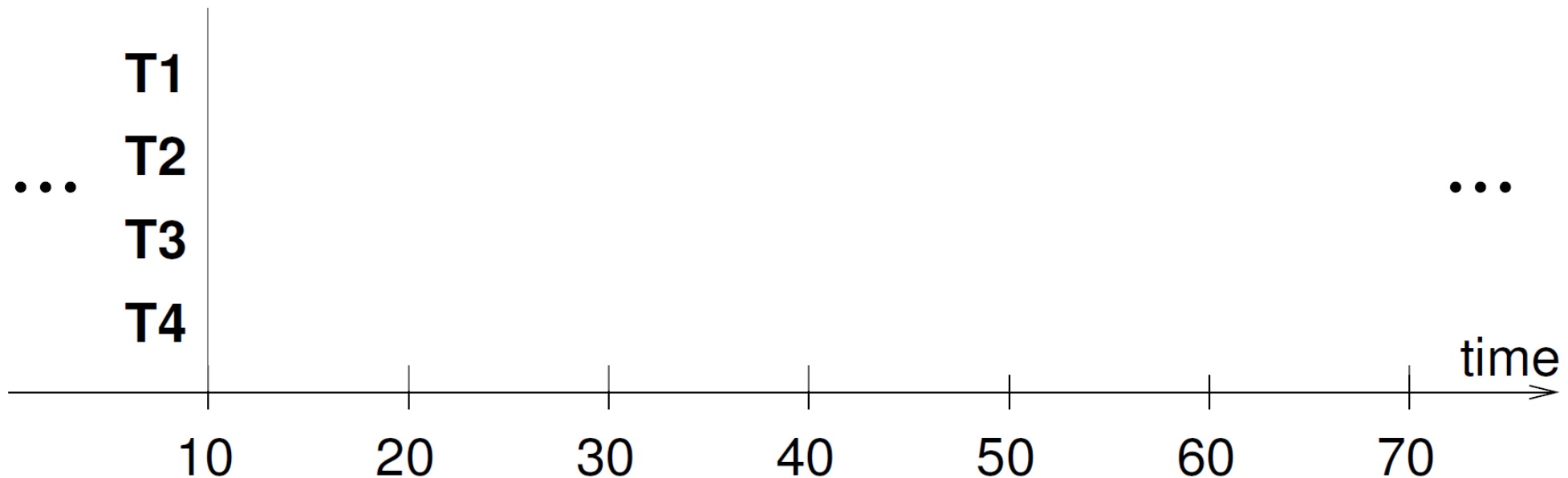
... // code in stage B

... // code in stage C

# Pipeline Stages in the Real World

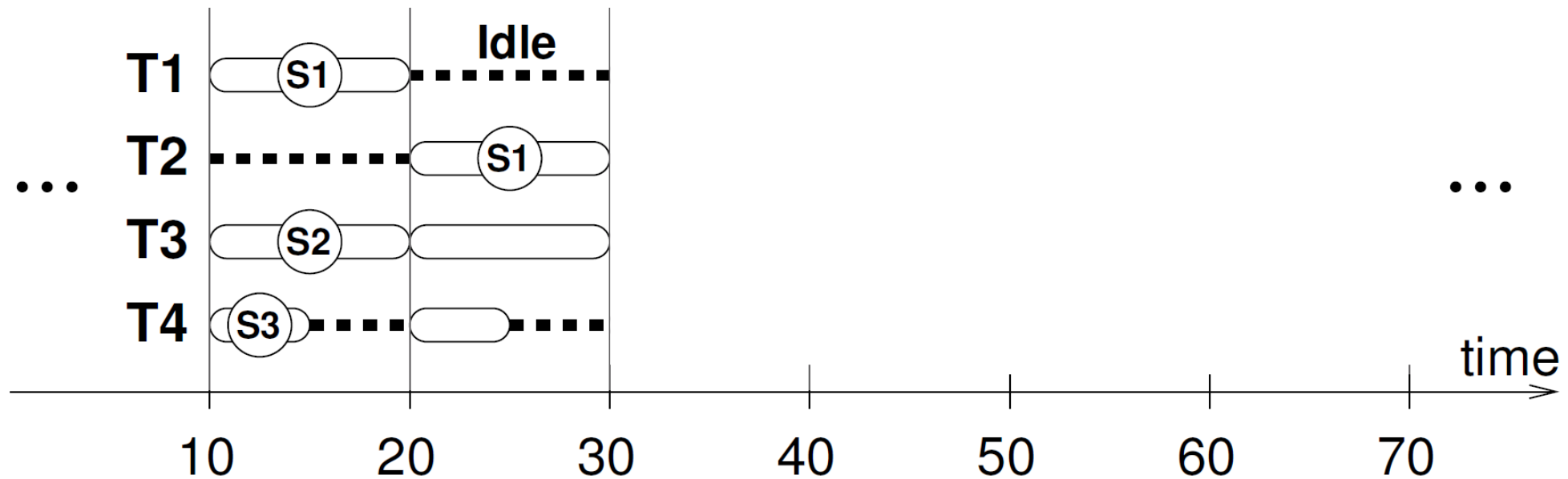
---

- Again: stages may take non-uniform time
- We can vary the distribution of stages-to-cores as we like



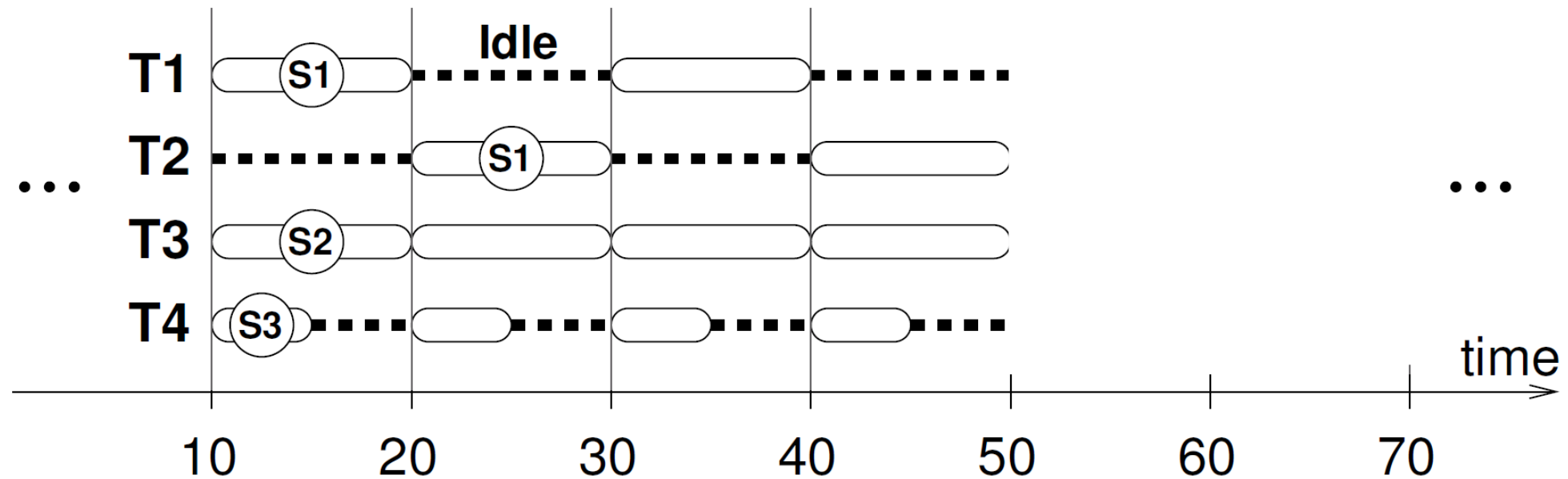
# Pipeline Stages in the Real World

- Again: **stages may take non-uniform time**
- We can **vary the distribution of stages-to-cores** as we like



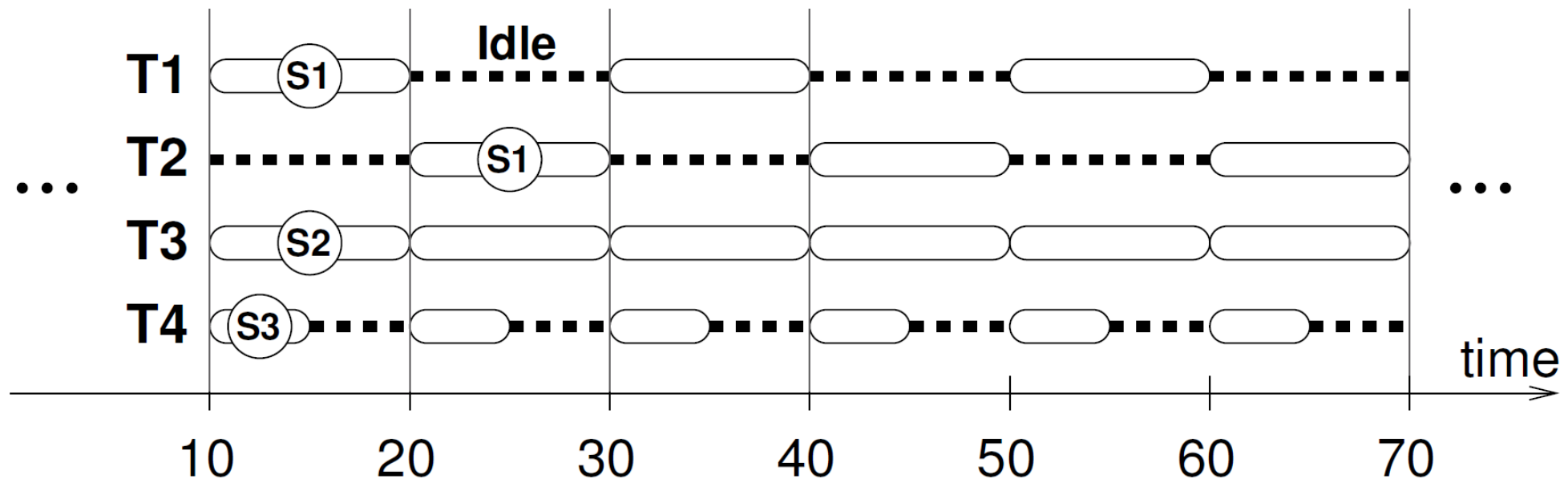
# Pipeline Stages in the Real World

- The **slowest stage causes** others to **wait**



# Pipeline Stages in the Real World

- The **slowest stage causes** others to **wait**
- Idea: **accelerate stages causing bottlenecks**



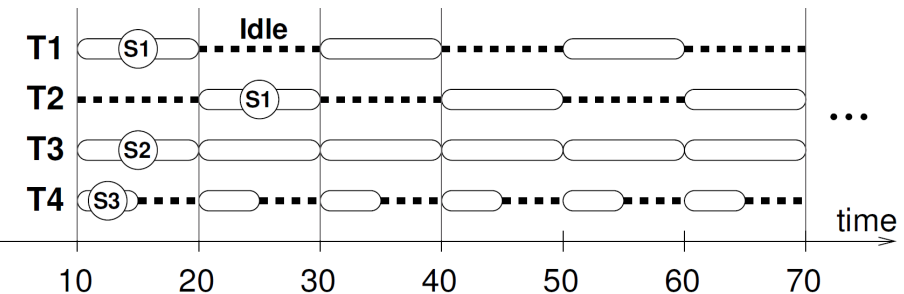
# Outline

---

- Background
- **Previous Work**
- Bottleneck Identification and Scheduling
  - Bottleneck Identification
  - Bottleneck Acceleration
- Improvements & Details
- Evaluation
- Critique

# Previous Work

- Asymmetric Chip Multiprocessor
  - Execute **serial phases on a large core**
  - Execute **parallel phases on multiple large/small cores**
- Feedback Directed Pipelining (FDP)
  - Pure **software framework**
  - **Accelerates pipelined workloads** using core-to-stage allocation selection
- Accelerated Critical Sections (ACS)
  - **Modifies an ACMP hardware system**
    - Adds Instructions to mark Critical Sections
    - Adds a “Critical Section Request Buffer”
  - **Accelerates Critical Sections** using



# Outline

---

- Problem & Background
- Previous Work
- Bottleneck Identification and Scheduling
- Improvements & Details
- Evaluation
  
- Critique

# Bottleneck Identification and Scheduling

---

- Goal: **Identify and accelerate bottlenecks** in multithreaded applications to speed up execution overall.
- Key idea:
  - Identification: The **most critical bottlenecks** make other threads wait the **longest**
  - Acceleration: Use (multiple) **large cores to accelerate bottlenecks**
- BIS overview:
  - **Mark** potential **bottlenecks in software**
  - **Identify** critical bottlenecks **at runtime**
  - **Accelerate** critical bottlenecks **on large cores**

# Bottleneck Identification and Scheduling

---

## 1. Identification

- Annotation
- Hardware Components

## 2. Acceleration

- Critical Bottleneck Selection

# 1. Bottleneck Identification

---

- Move bottlenecks into own **function (de-inline)**
- Mark bottlenecks in software using three new instructions:
  - BottleneckCall *bid*, targetPC
    - Marks the **beginning of a new bottleneck** with a bottleneck-id
    - TargetPC is the PC of the start of the bottleneck code
  - BottleneckWait *bid*
    - **Waits for memory** to change
    - Similar to **mwait**
  - BottleneckReturn *bid*
    - **Ends a bottleneck** function
    - Returns like normal function return
- **Identify** critical bottlenecks **at run-time**

# Critical Section Annotation

---

```
call targetPC
```

```
targetPC: while cannot acquire lock
```

```
    mwait
```

```
    acquire lock
```

```
    (...)
```

```
    release lock
```

```
    return
```

# Critical Section Annotation

---

**BottleneckCall** *bid, targetPC*

```
targetPC: while canot acquire lock
    mwait
    acquire lock
    (...)
    release lock
    return
```

# Critical Section Annotation

---

**BottleneckCall** *bid*, *targetPC*

targetPC: while cannot acquire lock

**BottleneckWait** *bid*

acquire lock

(...)

release lock

return

# Critical Section Annotation

---

**BottleneckCall** *bid*, *targetPC*

targetPC: while cannot acquire lock

**BottleneckWait** *bid*

acquire lock

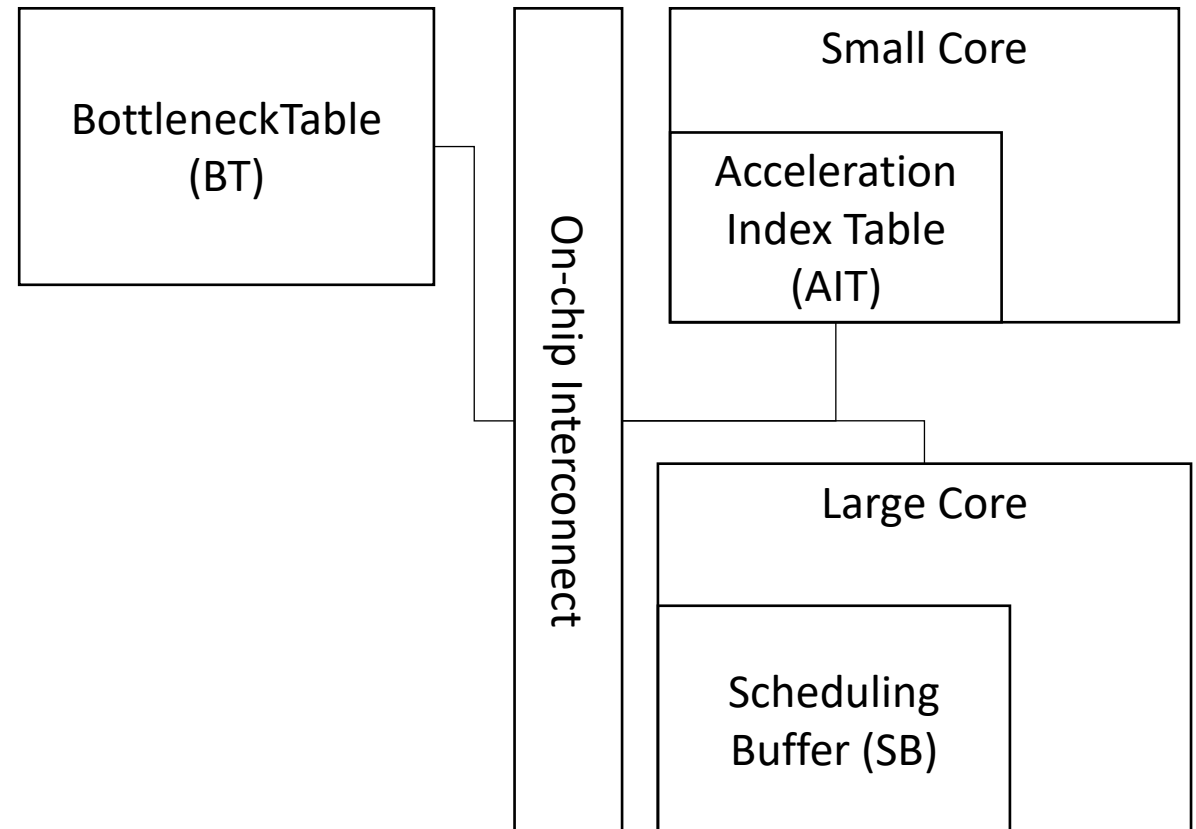
(...)

release lock

**BottleneckReturn** *bid*

# Hardware – Single Large Core

- One *Bottleneck Table (BT)*
  - Saves metadata of bottlenecks
- Each small core has *Acceleration Index Table (AIT)*
  - Avoids accesses to BT
  - Caches *bid* and *accel\_enable* bit for bottlenecks
- Large core has a *Scheduling Buffer (SB)*
  - Saves which bottlenecks are to be executed on large core



# Hardware – Bottleneck Table

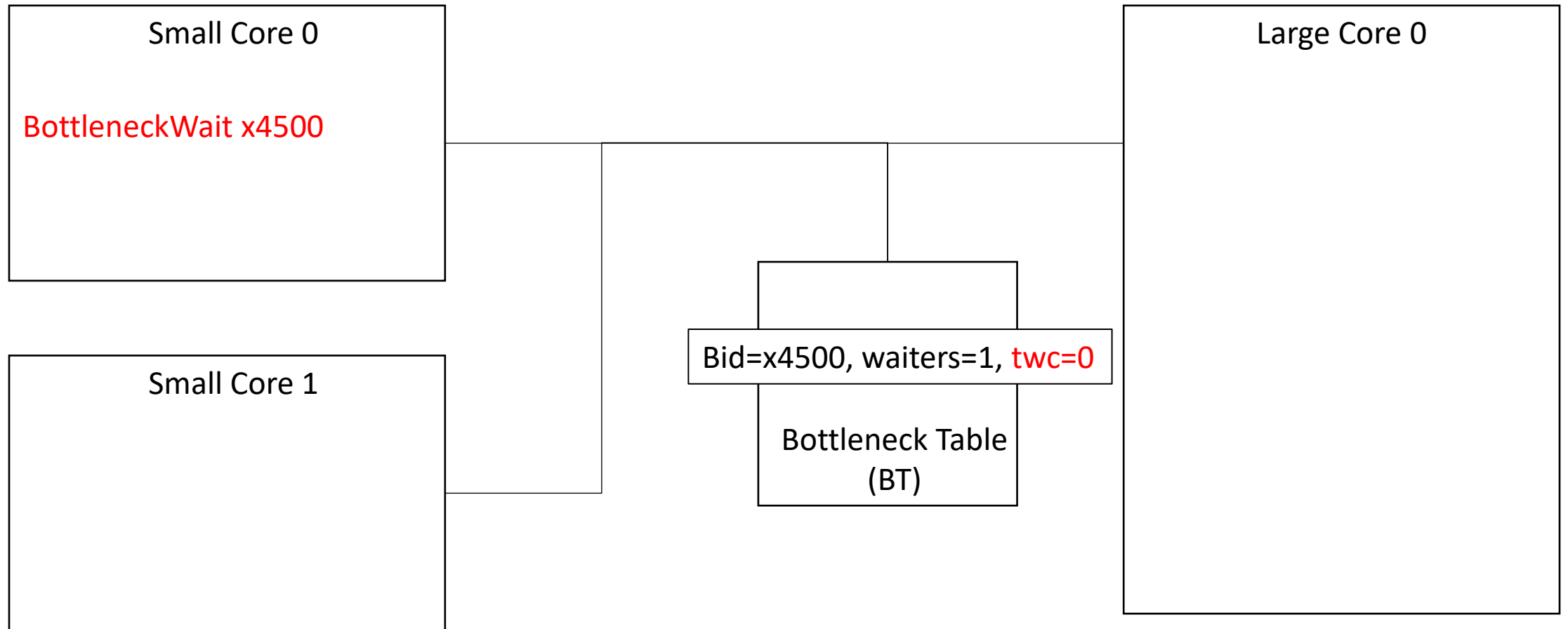
- Bottleneck Table holds **metadata for bottlenecks**
- Implemented as an **associative cache**
  - Evict bottleneck with smallest number of *Thread Waiting Cycles*
- Halve *Thread Waiting Cycles* every 100k cycles to replace stale entries



Field	Description
bid	Bottleneck ID
pid	Process ID
executers	Current # of threads running <i>bid</i>
executer_vec	Bit vect of threads running <i>bid</i>
waiters	Current # of threads waiting for <i>bid</i>
waiters_sb	Current # of threads on SB waiting for <i>bid</i>
TWC	Thread waiting cycles
large_core_id	ID of large core

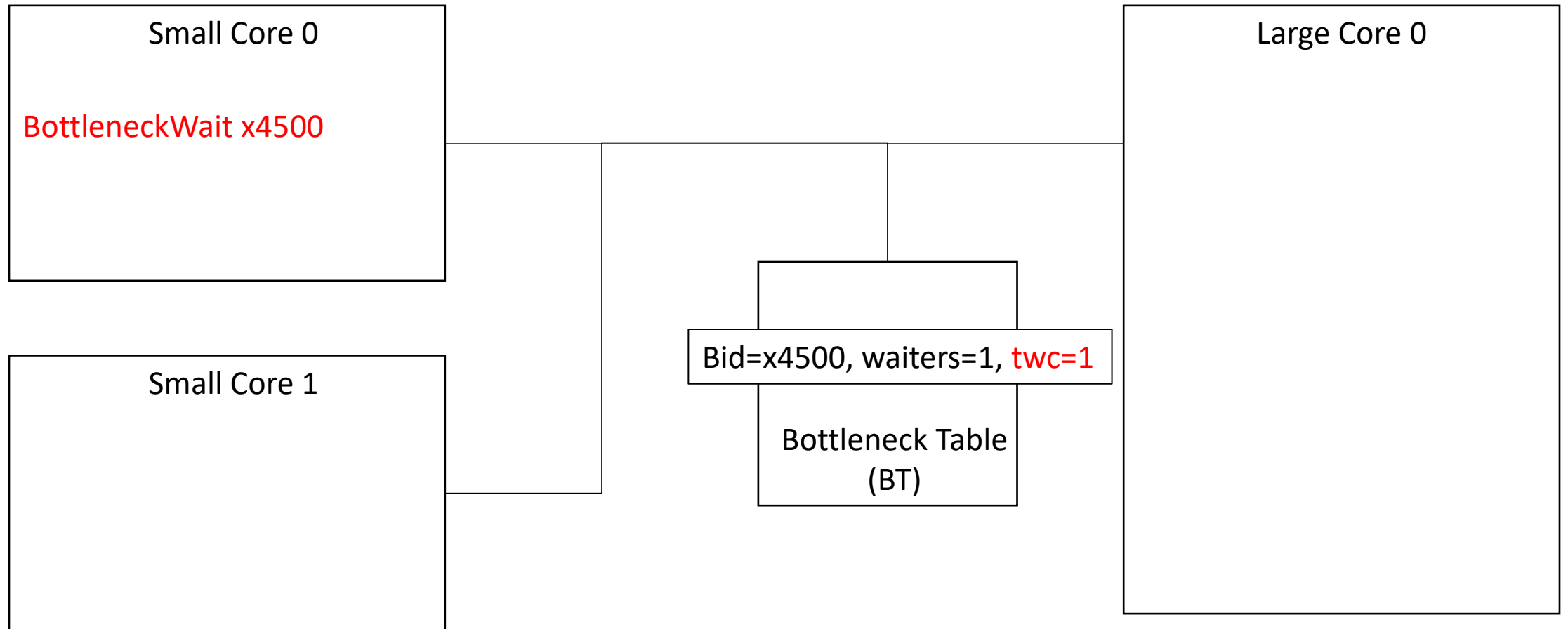
# Determine *TWC* for Bottlenecks

---



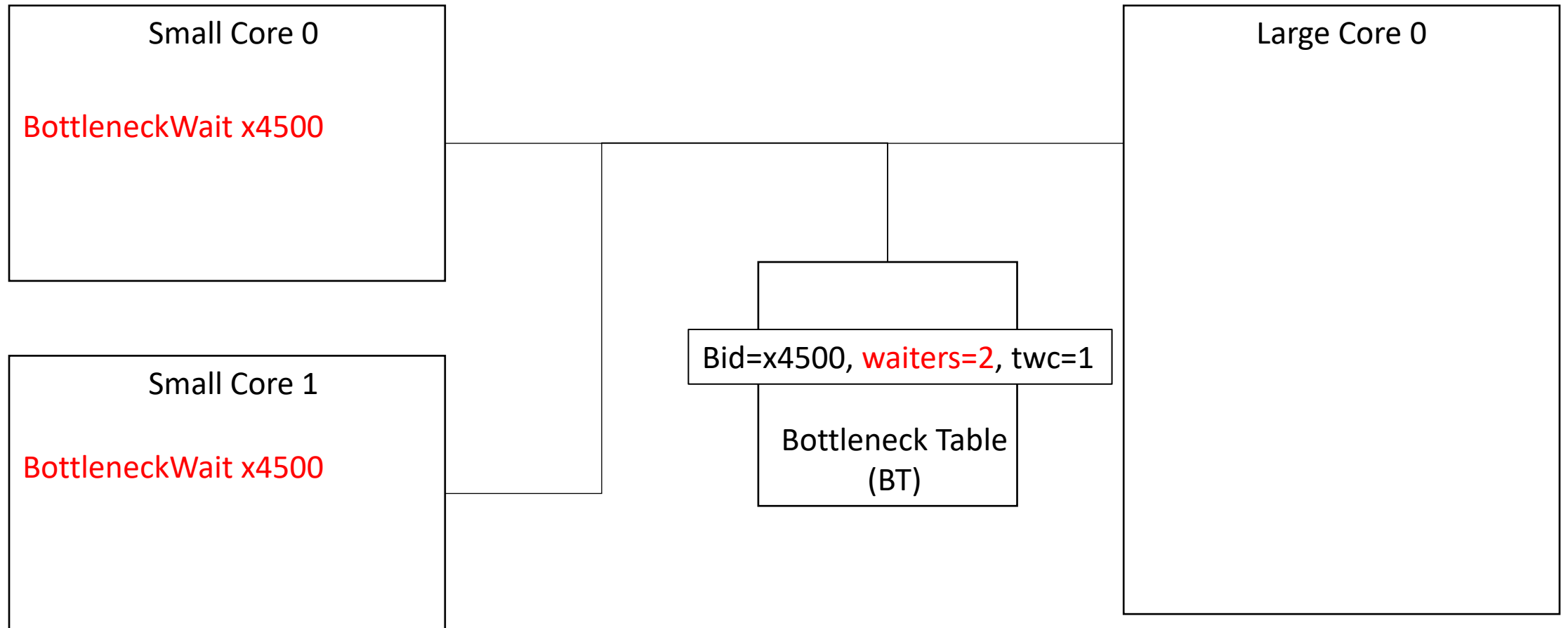
# Determine *TWC* for Bottlenecks

---

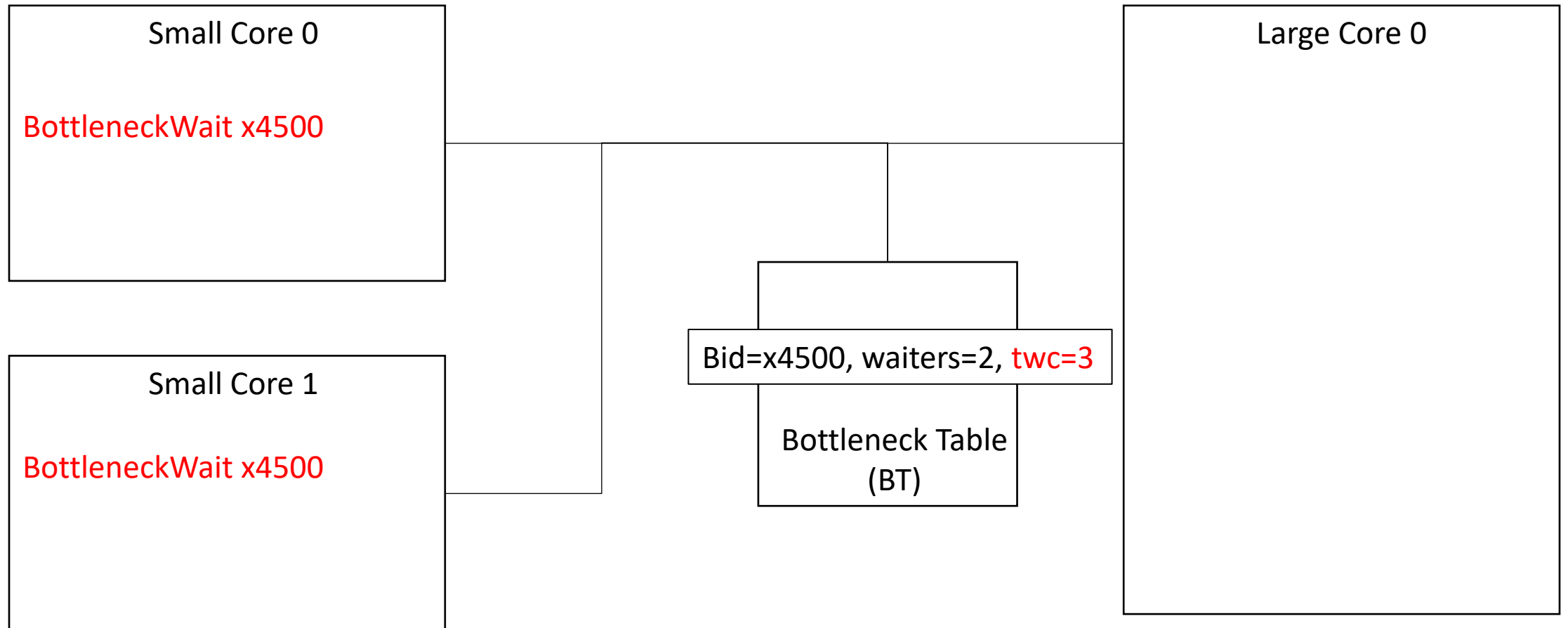


# Determine *TWC* for Bottlenecks

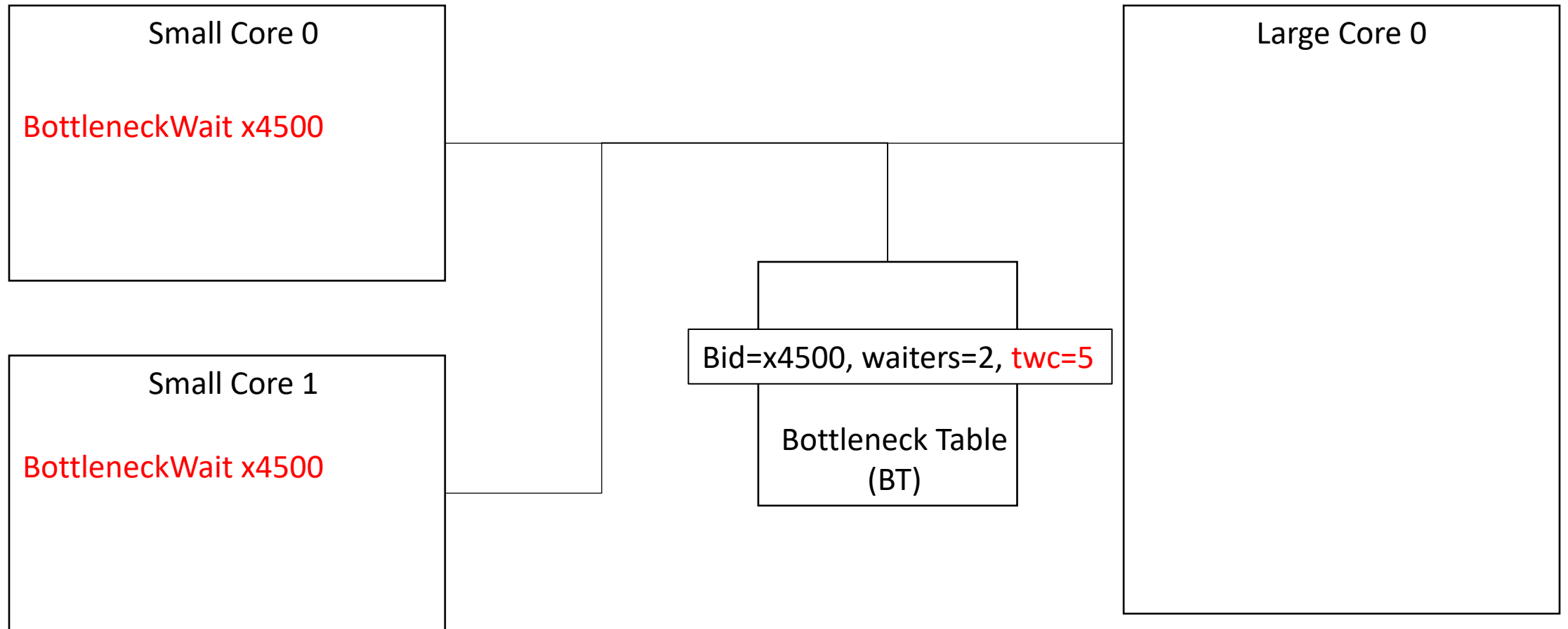
---



# Determine *TWC* for Bottlenecks



# Determine *TWC* for Bottlenecks



# Bottleneck Identification and Scheduling

---

## 1. Identification

- Annotation
- Hardware Components

## 2. Acceleration

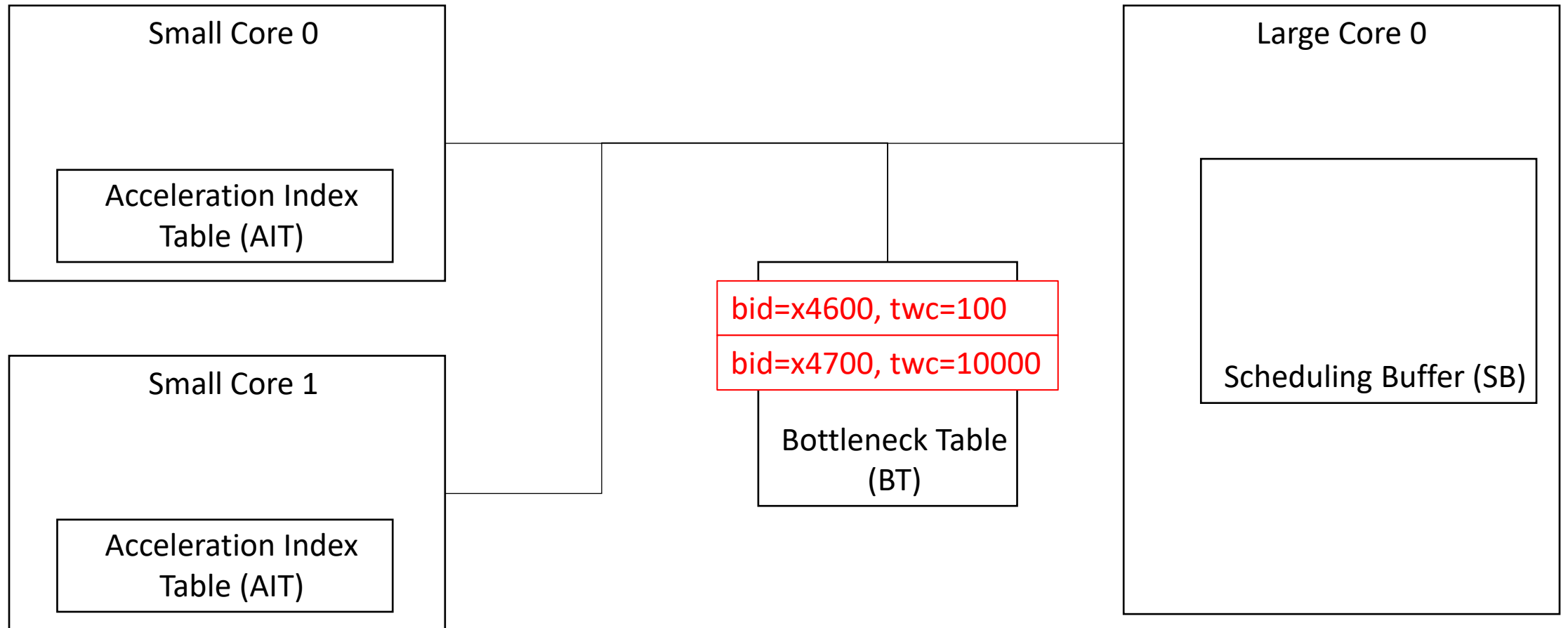
- Critical Bottleneck Selection

## 2. Acceleration

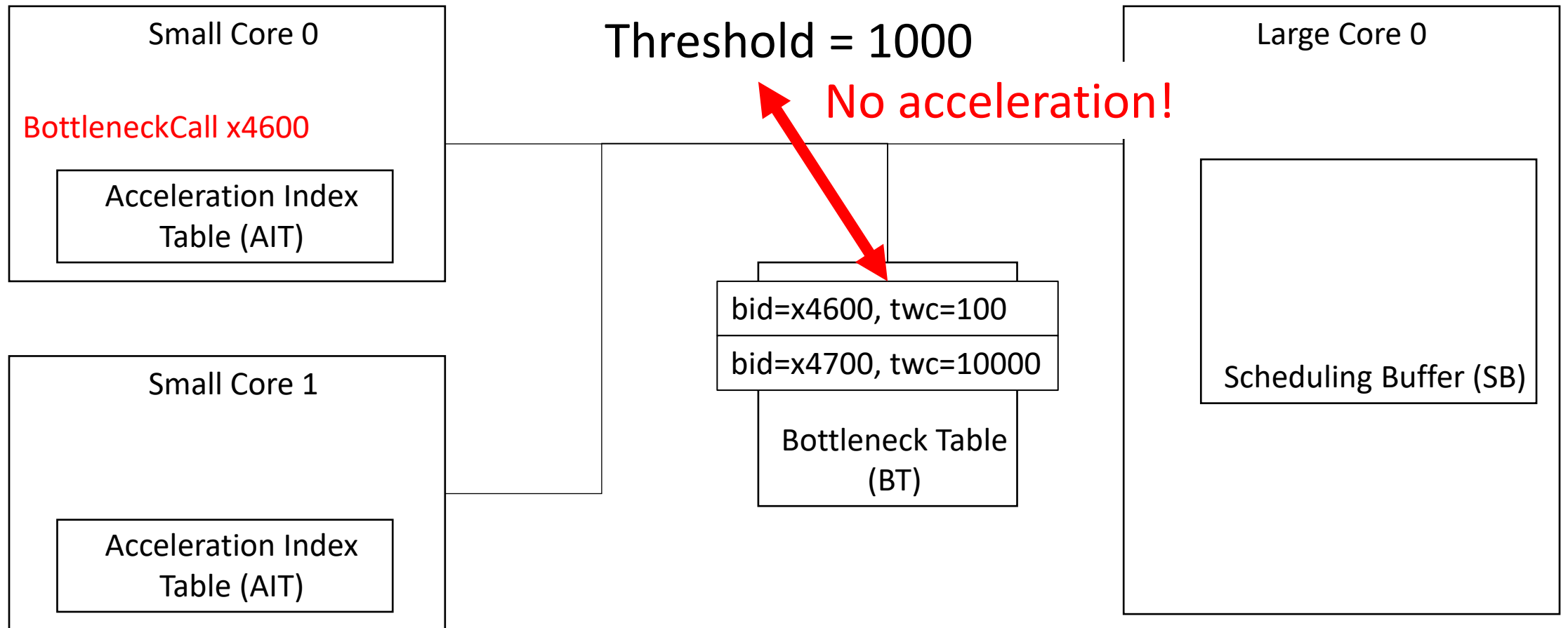
---

- Accelerate bottleneck **with highest Thread Waiting Cycles** (above threshold)
- Driven by insight that the **most critical bottleneck** is the one that **makes other threads wait the longest**

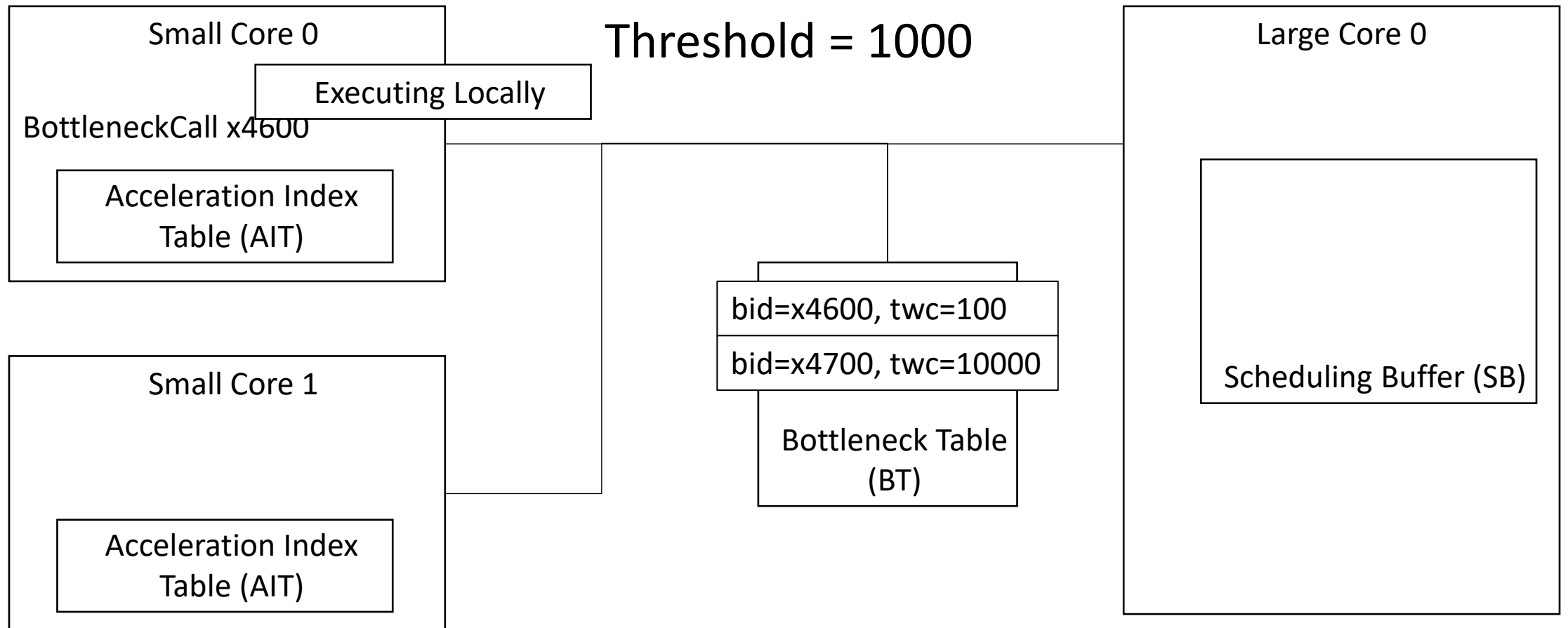
# Bottleneck Acceleration



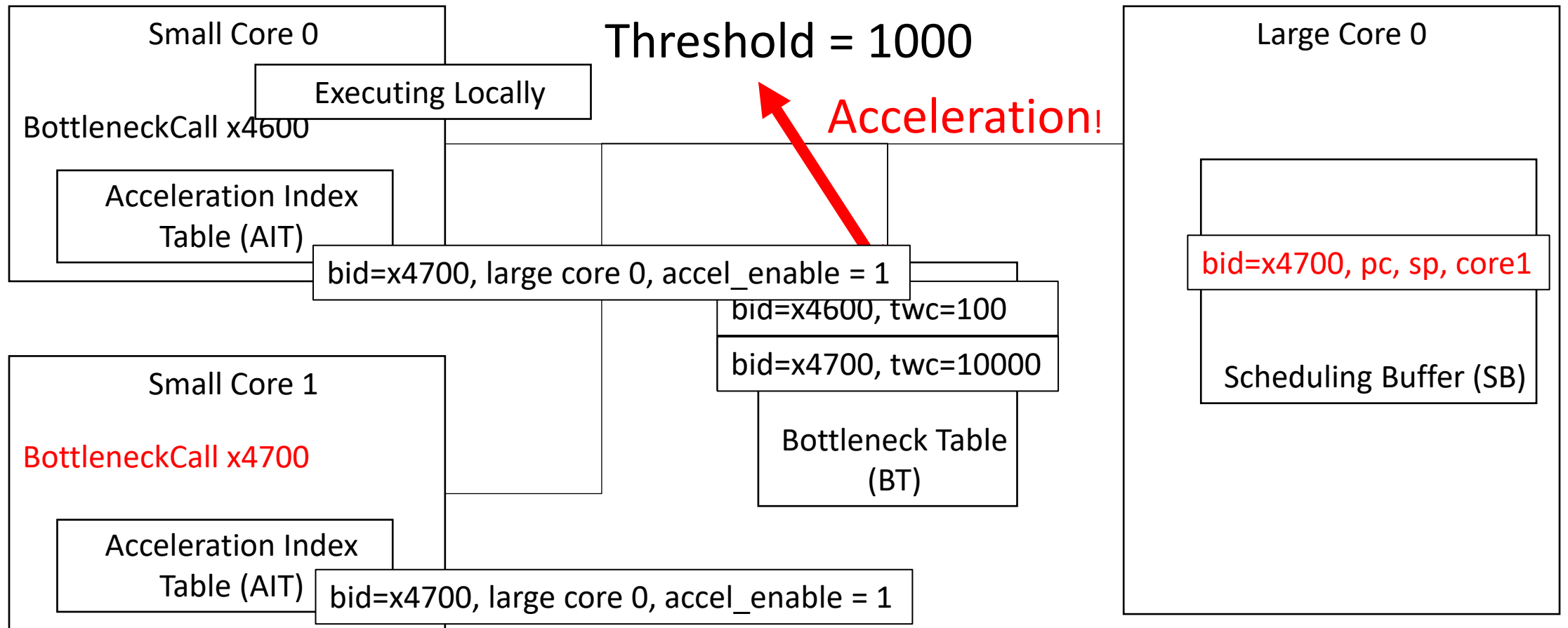
# Bottleneck Acceleration



# Bottleneck Acceleration



# Bottleneck Acceleration



## 2. Acceleration – Multiple Large Cores

---

- One Scheduling Buffer per core
- Each enabled **bottleneck assigned to fixed large core**
  - **Preserve cache locality**
  - **Avoid large cores waiting on each other** for same bottleneck
  - Bottleneck entry gains large core ID
- How to **accelerate**:
  - **Top  $N$  bottlenecks assigned to  $N$  large cores**
  - Rest assigned uniformly at random
- For Simultaneous Multi-Threading (SMT): execute different bottlenecks from same Scheduling Buffer

# Outline

---

- Background
- Previous Work
- Bottleneck Identification and Scheduling
- Improvements & Details
- Evaluation
- Critique

# False Serialization

---

- Situation:
  - $BT_1 > BT_2$ ,  $time(BT_1) = 4$ ,  $time(BT_2) = 2$ ,  $speedup_{large} = 2$
  - Both are scheduled on the large core
  - $BT_1$  starts executing on large core, takes 2 seconds
  - $BT_2$  has to wait, ultimately takes 3 seconds until complete
  - Better: Execute  $BT_2$  on small core and  $BT_1$  on large core
- Solution: Abort bottleneck in Scheduling Buffer if
  1. Bottleneck does not have highest Thread Waiting Cycles
  2. Bottleneck could be run on small core

# Pre-emptive Acceleration

---

- Situation:
  - We **schedule  $BT_1$  on small core**
  - Other **bottlenecks** start executing but **start waiting for  $BT_1$**
  - **Thread Waiting Cycles of  $BT_1$  increase**, but it **remains on small core**
  - Better: **Run  $BT_1$  on large core** if  $BT_1$  becomes critical
- Solution: Pre-emptive Mechanism
  - On update of Thread Waiting Cycles:
    1. If  $BT_1$  has become the most critical bottleneck
    2. If the number of executors is  $\leq$  number of large cores
  - **Pre-empt small core** and **ship  $BT_1$  to large core** for execution
    - Save “architectural state” on stack etc.
- **Primary acceleration** mechanism for both **barriers and pipeline stages**

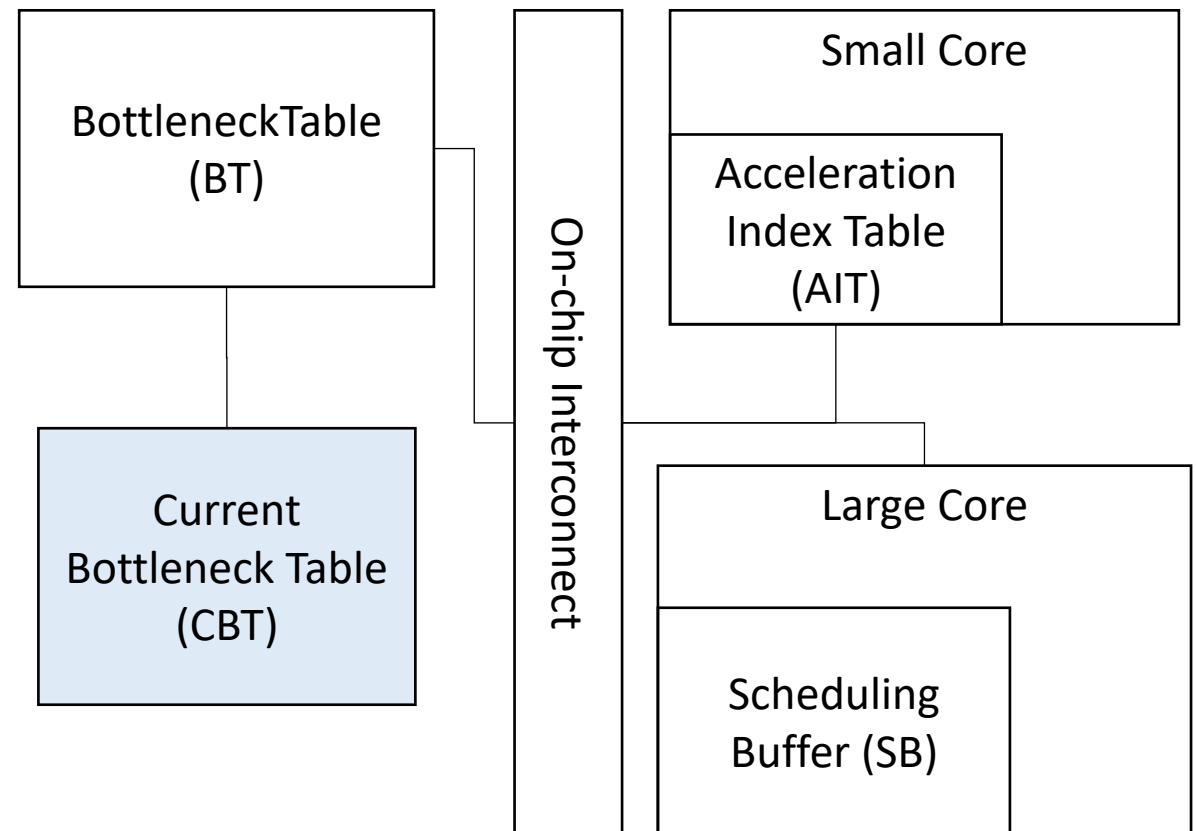
# Nested & Dependent Bottlenecks

---

- Situation:
  - $T_1$  is running  $BT_1$ , but waits for  $BT_2$
  - $T_2$  is waiting for  $BT_1$
  - $T_2$  is indirectly waiting for  $T_1$ ,  $BT_1$  is indirectly waiting for  $BT_2$ !
- Solution:
  - Follow dependency chain between bottlenecks until we find a «root bottleneck»
  - Add current number of waiters for «child bottlenecks» to root
  - Need to know:
    1. Which thread is executing which bottleneck
      - Add `executer_vec` for each Bottleneck Table entry, one bit per hardware thread
    2. Which bottleneck is being waited for
      - Add Current Bottleneck Table (CBT)

# Current Bottleneck Table

- Add Current Bottleneck Table (CBT)
- Maps hardware thread ids to *bid* of bottlenecks currently being waited for by the thread



# Data Marshaling

---

- Situation:
  - When bottleneck is moved from small to large core cache state is lost
  - Execution on large core will incur unnecessary cache misses
- Solution: Data Marshaling
  - Identify and «marshal» cache lines to remote core

# Outline

---

- Background
- Previous Work
- Bottleneck Identification and Scheduling
- Improvements & Details
- **Evaluation**
- Critique

# Experimental Methodology

---

- Workloads: 8 with critical sections, 2 with barriers, 2 pipelined applications
- Simulations on x86 cycle-level simulator
  - Small cores modeled after Intel Pentium
    - 4GHz, in-order
  - Fast cores modeled after Intel Core 2
    - 4GHz, out-of-order
  - Caches: Private 32KB L1, private 256KB L2, shared 8MB L3

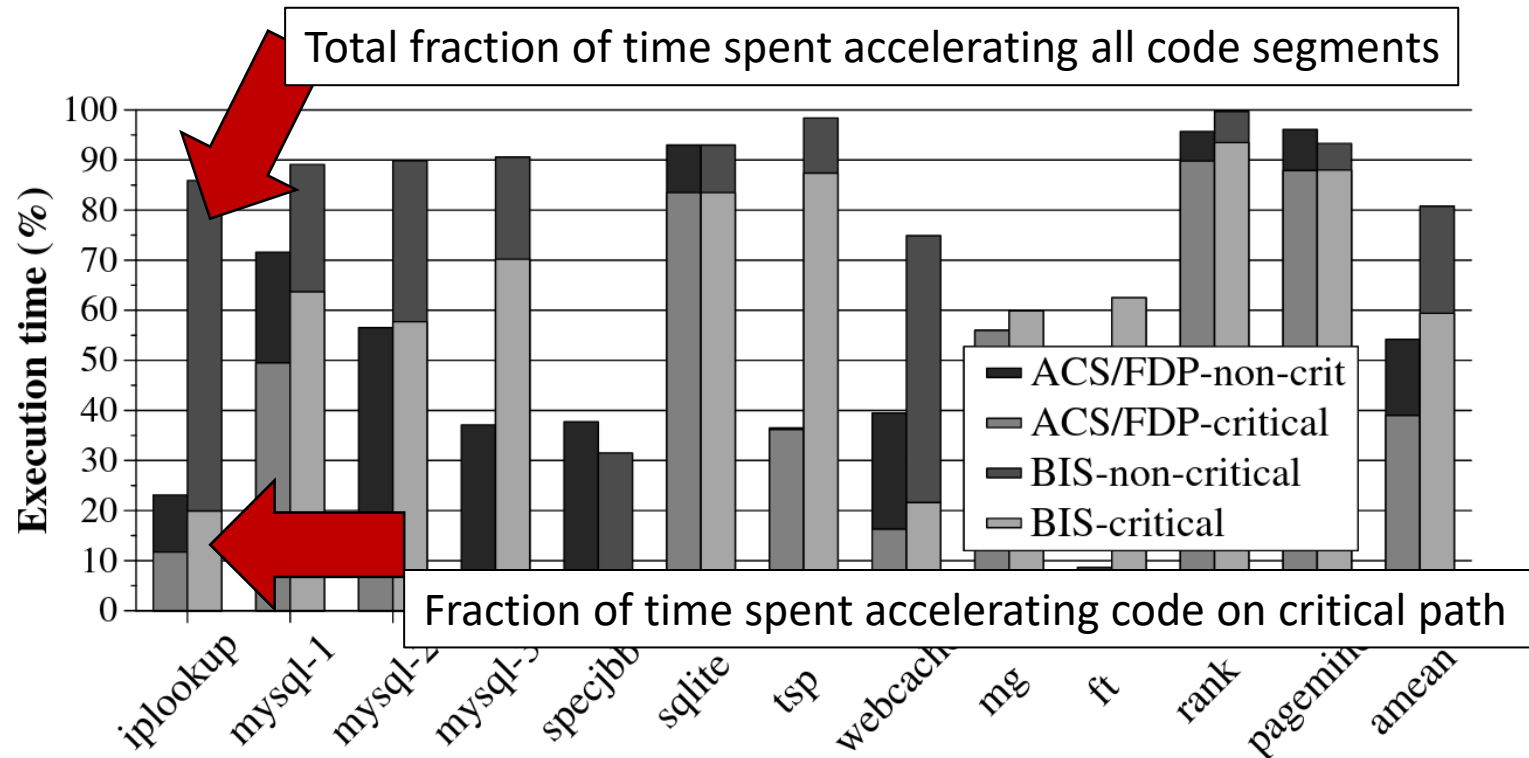
# Experimental Methodology

---

- SCMP — Symmetric Core Multi-Processor
  - $N$  small cores
- ACMP — Asymmetric Core Multi-Processor
  - 1 large core,  $N - 4$  small cores
  - Large core always runs single-threaded code
- ACS — Accelerated Critical Sections
  - 1 large core,  $N - 4$  small cores
  - Large core always runs single-threaded code
  - Large core accelerates critical sections
- BIS
  - $L$  large cores
  - $S = N - 4L$  small cores
  - 1 large core always runs single-threaded code
  - 32-entry BT
  - $N$ -entry CBT
  - Each large core:  $S$ -entry SB
  - Each small core: 32-entry AT

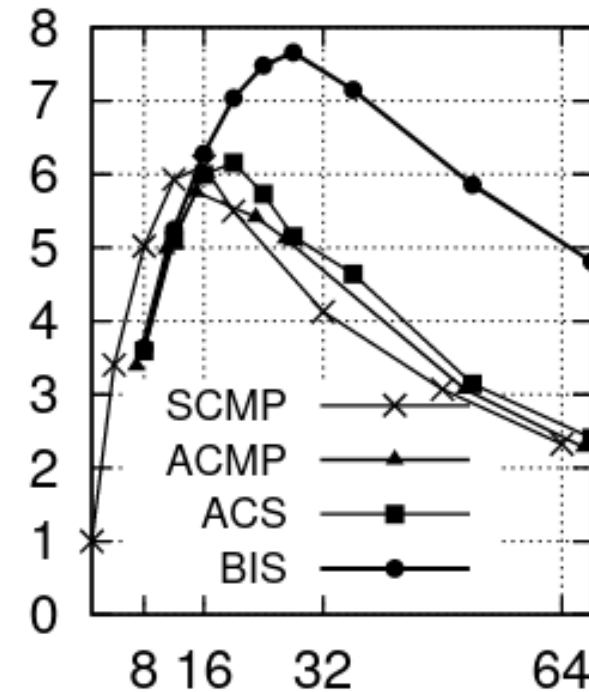
# Bottleneck Identification

- Accuracy: identified bottlenecks on the critical path over total identified bottlenecks
  - 72% (ACS/FDP) to 73.5% (BIS)
- **Coverage**: fraction of program critical path that is actually identified as bottlenecks
  - 39% (ACS/FDP) to 59% (BIS)



# Speedup over a single small core

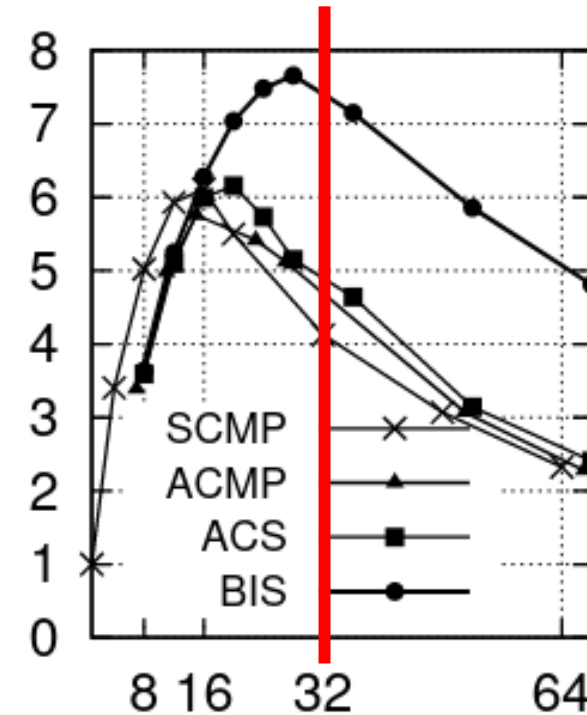
- Use **as many threads as cores**



(d) mysql-3

# Speedup over a single small core

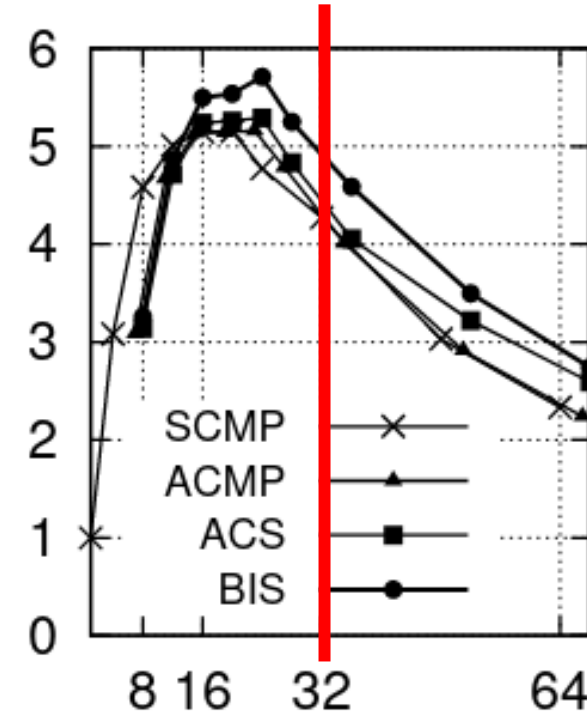
- Use **as many threads as cores**
- For 32 cores, **B/S matches or outperforms** all other approaches



(d) mysql-3

# Speedup over a single small core

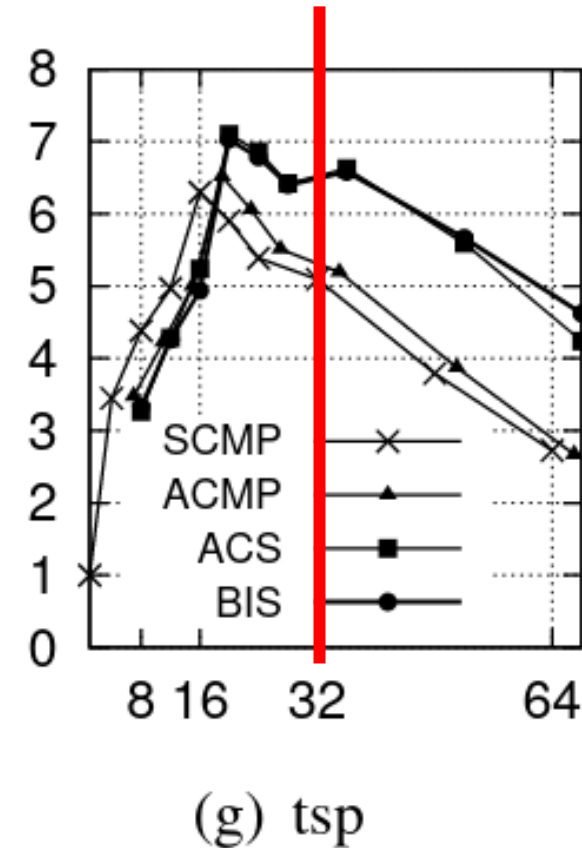
- Use **as many threads as cores**
- For 32 cores, **B/S matches or outperforms** all other approaches



(a) iplookup

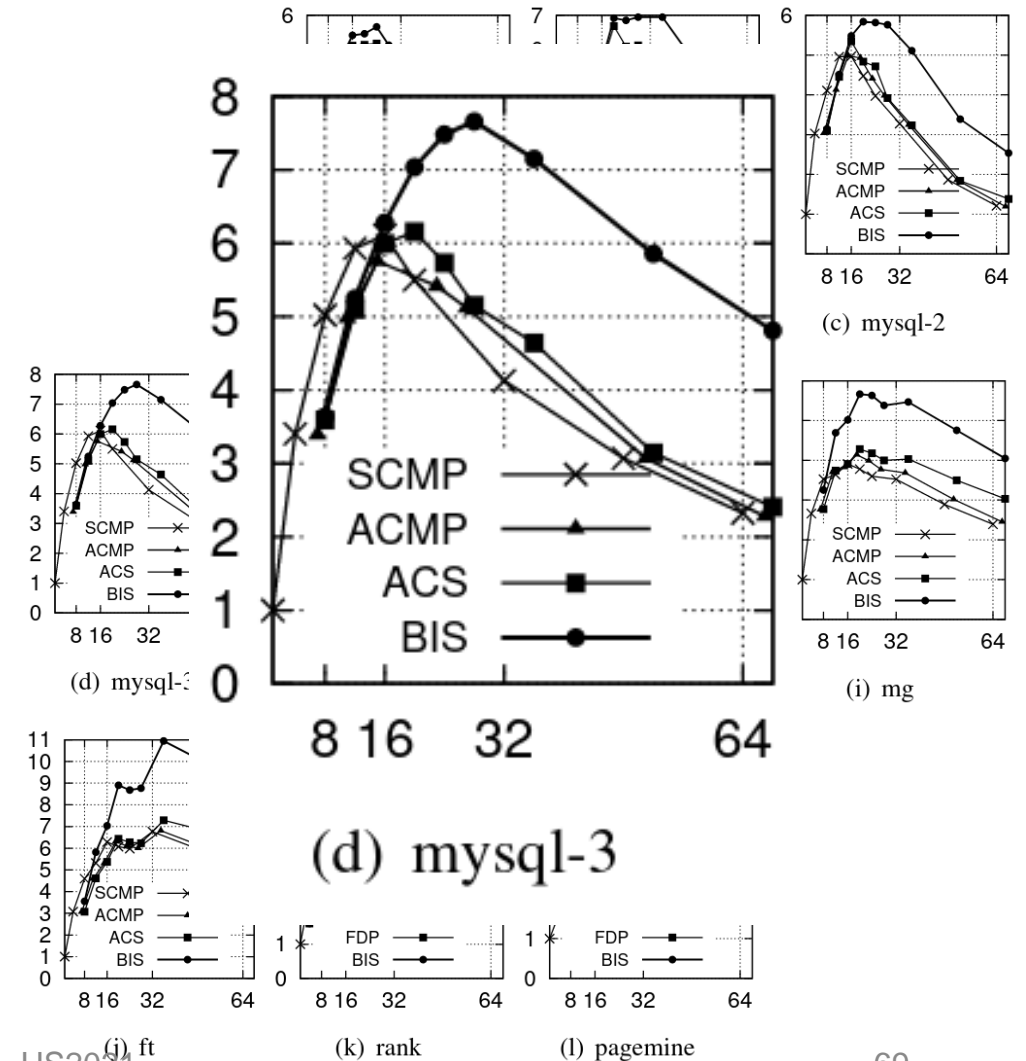
# Speedup over a single small core

- Use **as many threads as cores**
- For 32 cores, **B/S matches or outperforms** all other approaches
- For *tsp*, **ACS accelerates fewer bottlenecks**, incurring **fewer cache misses** on the large core.



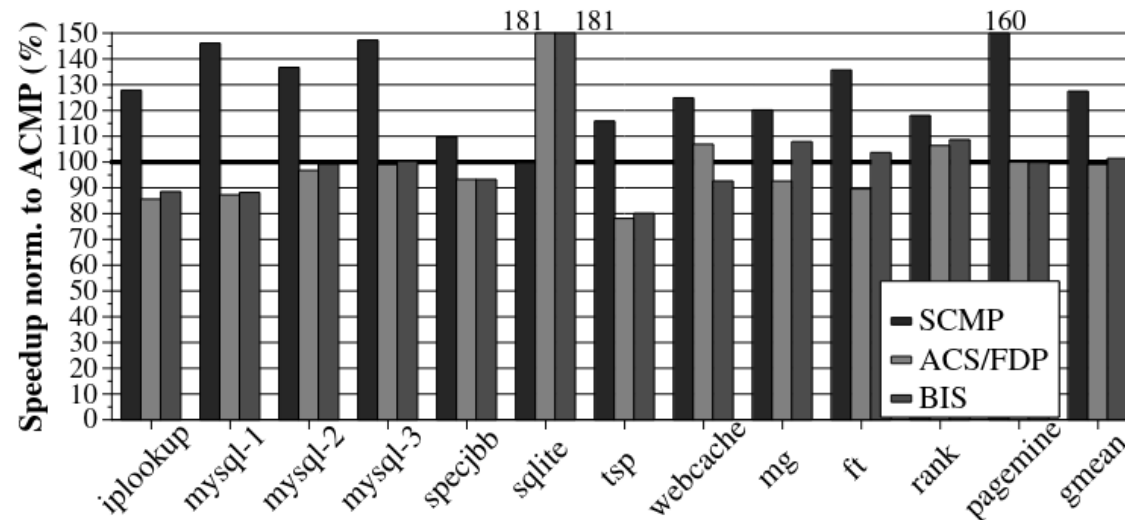
# Speedup Over a Single Small Core

- The more cores, the better *BIS*
- For small area budgets, **large core replaces 3 small cores for ACMP/FDP, 4 small cores for ACS/BIS**
  - Loss of general purpose cores
  - ACMP/FDP run only one thread on the large core
  - ACS/BIS dedicate large core for critical sections and bottlenecks



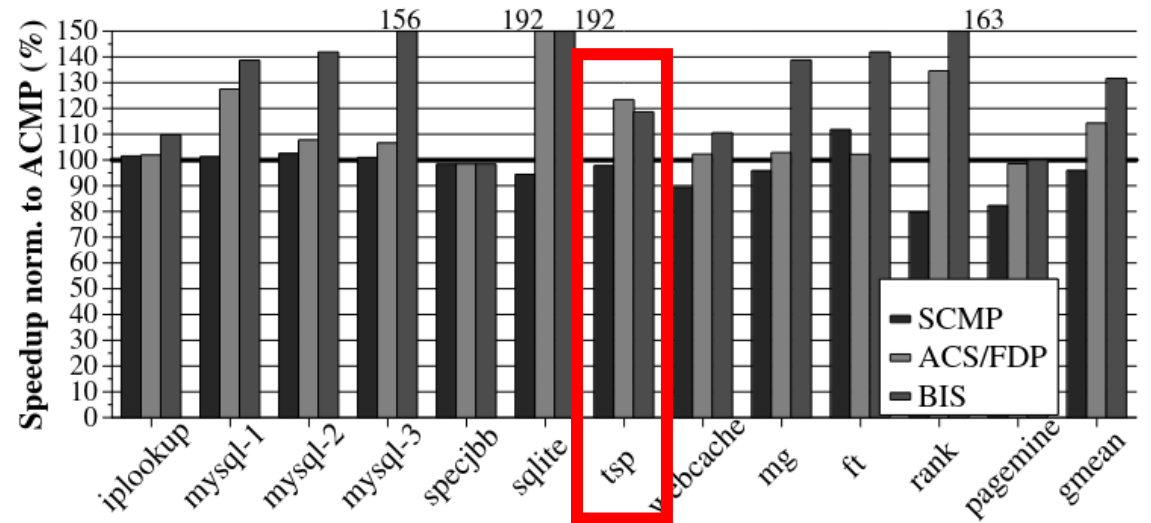
# Optimal Number of Threads

- Area budget = 8; 4 small cores, 1 large core
- For small area budgets, **large core replaces 3 small cores** for **ACMP/FDP, 4 for ACS/BIS**
  - **SCMP uses all cores** as normal



# Optimal Number of Threads

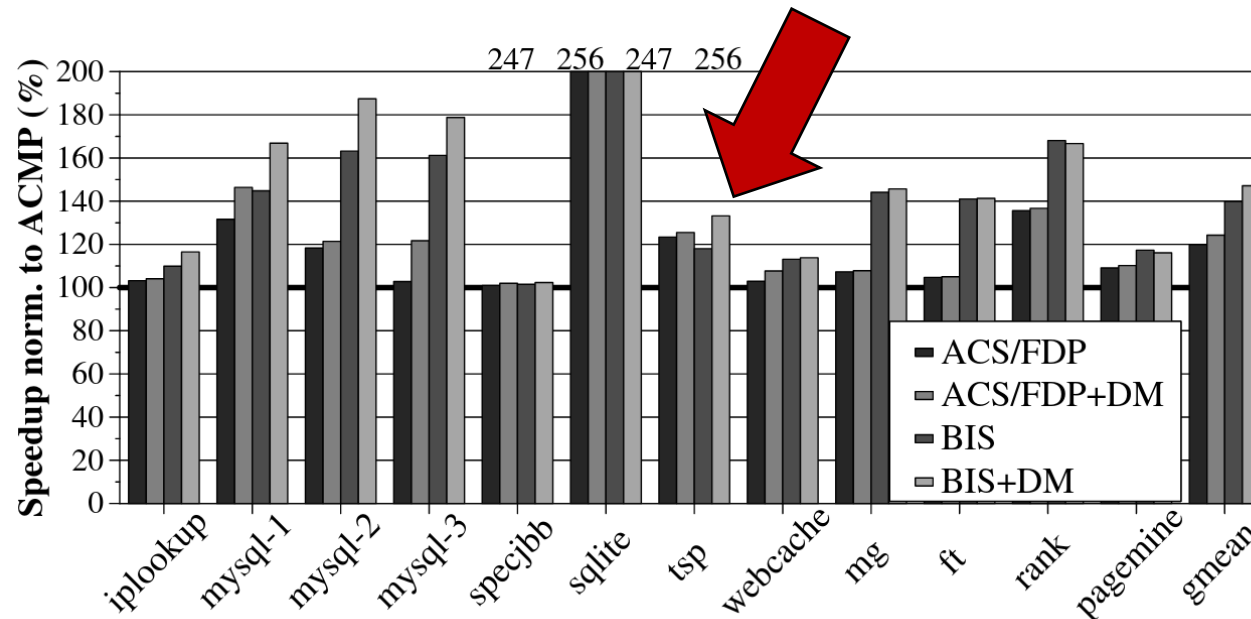
- 28 small cores, 1 large core
- SCMP loses its advantage



- For *tsp* we see that BIS underperforms ACS
  - BIS accelerates more bottlenecks than ACS
  - *tsp* bottlenecks are only 52 instructions long on average
  - BIS incurs cache misses without Data Marshalling

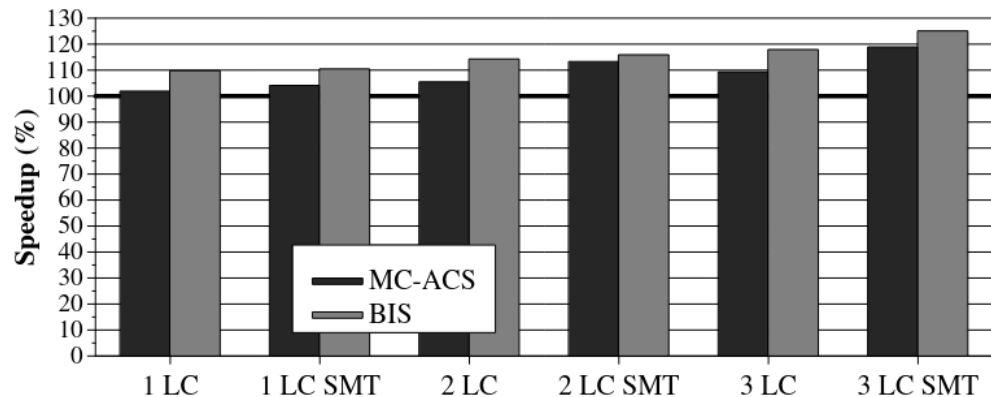
# Data Marshaling

- **Workloads with many small bottlenecks** can incur «inter-segment» cache penalties
  - The benefit of acceleration does not overcome the cost of cache misses
- With DM on average: BIS+DM +5.2%, ACS+DM +3.8%
- Especially *tsp* profits

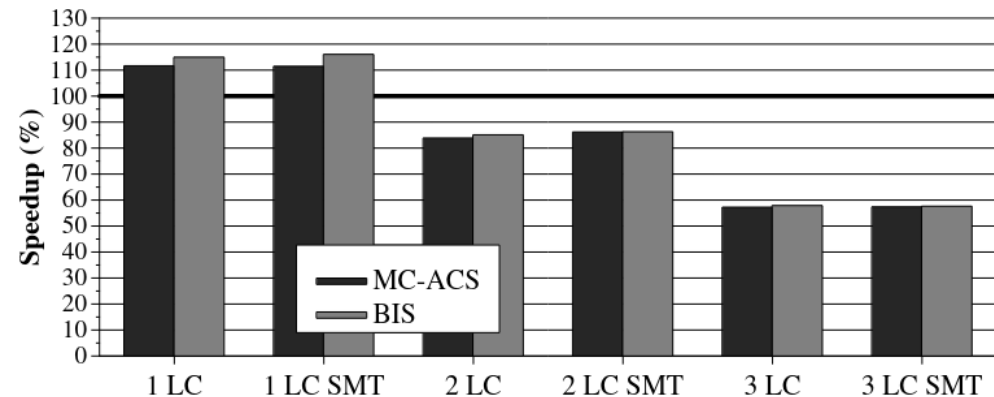


# No free lunch

- SMT = Simultaneous Multithreading
- *Iplookup* executes many independent critical sections
  - Benefits from more large cores to accelerate
- *Mysql-2* does not benefit from more large cores
  - Cost of reducing number of concurrent threads is larger than the benefit of accelerating multiple critical sections



(a) iplookup at 32-core area budget



(b) mysql-2 at 16-core area budget

# Conclusion

---

- *Bottleneck Identification and Scheduling (BIS)* is the **first general mechanism to identify most critical bottlenecks** and accelerate them using Asymmetric Core Multi-Processor (ACMP)
- Particularly, *BIS* is the **first approach to use multiple large cores** for acceleration; with success
- *BIS'* **identification step improves coverage of bottlenecks significantly** over *ACS/FDP*
- *BIS* **improves performance over ACS and FDP by 15%** on average in bottleneck-intensive applications
- *BIS* benefits increase as number of cores increase

# Outline

---

- Background
- Previous Work
- Bottleneck Identification and Scheduling
- Improvements & Details
- Evaluation
- Critique

# Strengths

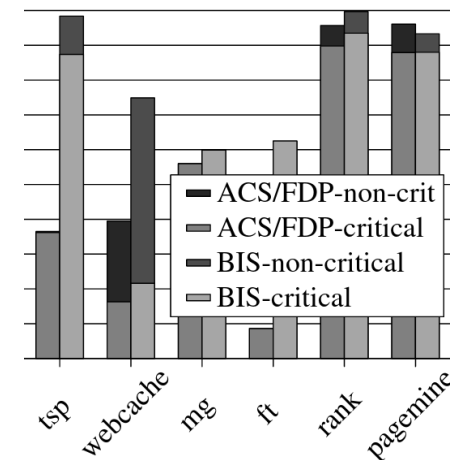
---

- Bottlenecks in multithreaded applications are important
- Simple mechanisms for identification & acceleration
  - Minimal changes to software
- Comprehensive analysis of results
  - Care taken to do fair comparisons
  - Used representative workloads
  - In-depth explanations of results
- Performance increase is significant

# Weaknesses

---

- Not designed to accelerate bottlenecks from multiple applications
- Performance is sensitive to workload and number of large/small cores
- «Large» number of cores needed to overcome benefit of more small cores
- Needs additional hardware (tables etc.)
- Black & white plots difficult to read



# Discussion

---

- How does varying the performance/cost multiple of the large cores change the evaluation?
- What would a complete system using BIS look like?
  - How do you solve the problem of multiple applications wanting to use cores at the same time?
  - What kind of additional performance costs do multiple applications introduce?
- How practical is the addition of hardware + 3 new instructions?
  - Can this approach work in general or is it only worth it in specialized contexts?
  - Which hardware environments use ACMP-like systems today?
- Which specific thing would have been the most important thing for future work to focus on?

# Bottleneck Identification and Scheduling in Multithreaded Applications

ASPLOS XVII, March 2012

Authors: José A. Joao, M. Aater Suleman, Onur Mutlu, Yale N. Patt

Presenter: Roman Meier

04.11.2021

# Backup Slides

# Interrupts

---

- Small core gets interrupted while waiting for large core:
  - Wait until `BottleneckDone` or `BottleneckCallAbort` received
  - Service interrupt
- Small core interrupted while `BottleneckWaiting`:
  - Force finish instruction
  - Service interrupt
  - Re-execute instruction
- Large core interrupted while accelerating:
  - Abort all bottlenecks on Scheduling Buffer
  - Finish current bottleneck
  - Service interrupt