
Bottleneck Identification and Scheduling in Multithreaded Applications

José A. Joao

M. Aater Suleman

Onur Mutlu

Yale N. Patt

Executive Summary

- **Problem:** Performance and scalability of multithreaded applications are limited by serializing bottlenecks
 - different types: critical sections, barriers, slow pipeline stages
 - importance (criticality) of a bottleneck can change over time
 - **Our Goal:** Dynamically identify the most important bottlenecks and accelerate them
 - How to identify the most critical bottlenecks
 - How to efficiently accelerate them
 - **Solution: Bottleneck Identification and Scheduling (BIS)**
 - Software: annotate bottlenecks (BottleneckCall, BottleneckReturn) and implement waiting for bottlenecks with a special instruction (BottleneckWait)
 - Hardware: identify bottlenecks that cause the most thread waiting and accelerate those bottlenecks on large cores of an asymmetric multi-core system
 - Improves multithreaded application performance and scalability, outperforms previous work, and performance improves with more cores
-

Outline

- Executive Summary
- The Problem: Bottlenecks
- Previous Work
- Bottleneck Identification and Scheduling
- Evaluation
- Conclusions

Bottlenecks in Multithreaded Applications

Definition: any code segment for which threads contend (i.e. wait)

Examples:

- **Amdahl's serial portions**
 - Only one thread exists → on the critical path
- **Critical sections**
 - Ensure mutual exclusion → likely to be on the critical path if contended
- **Barriers**
 - Ensure all threads reach a point before continuing → the latest thread arriving is on the critical path
- **Pipeline stages**
 - Different stages of a loop iteration may execute on different threads, slowest stage makes other stages wait → on the critical path

Observation: Limiting Bottlenecks Change Over Time

A=full linked list; B=empty linked list

repeat

Lock A

Traverse list A

Remove X from A

Unlock A

Compute on X

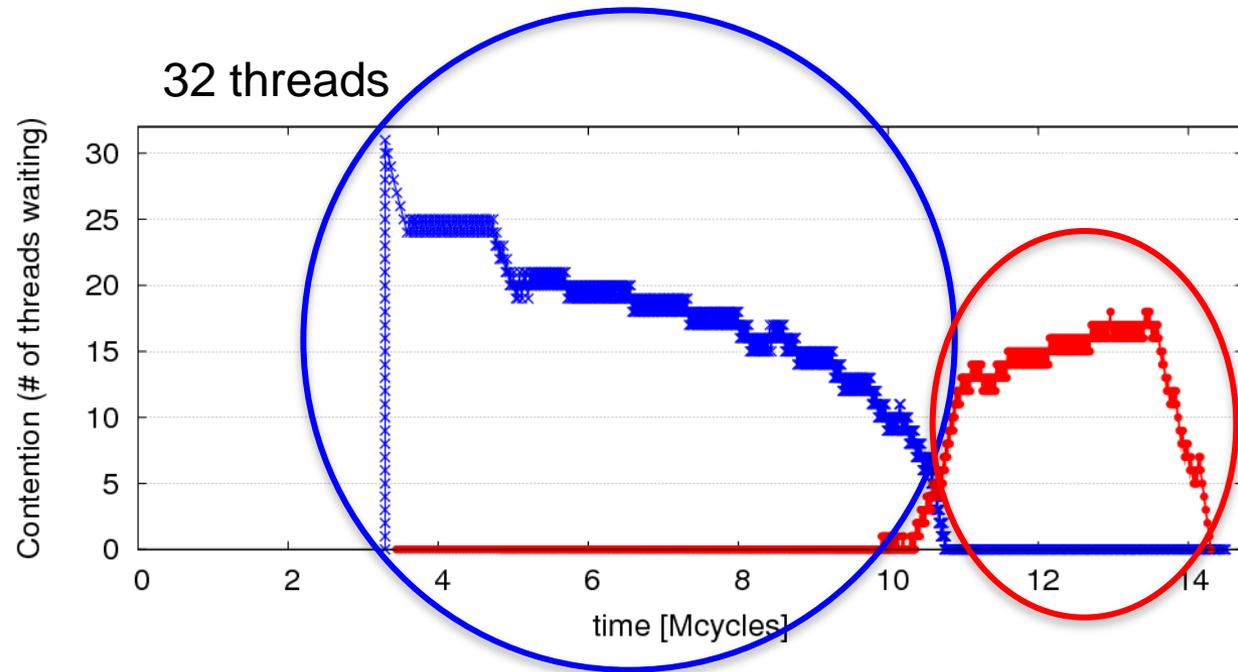
Lock B

Traverse list B

Insert X into B

Unlock B

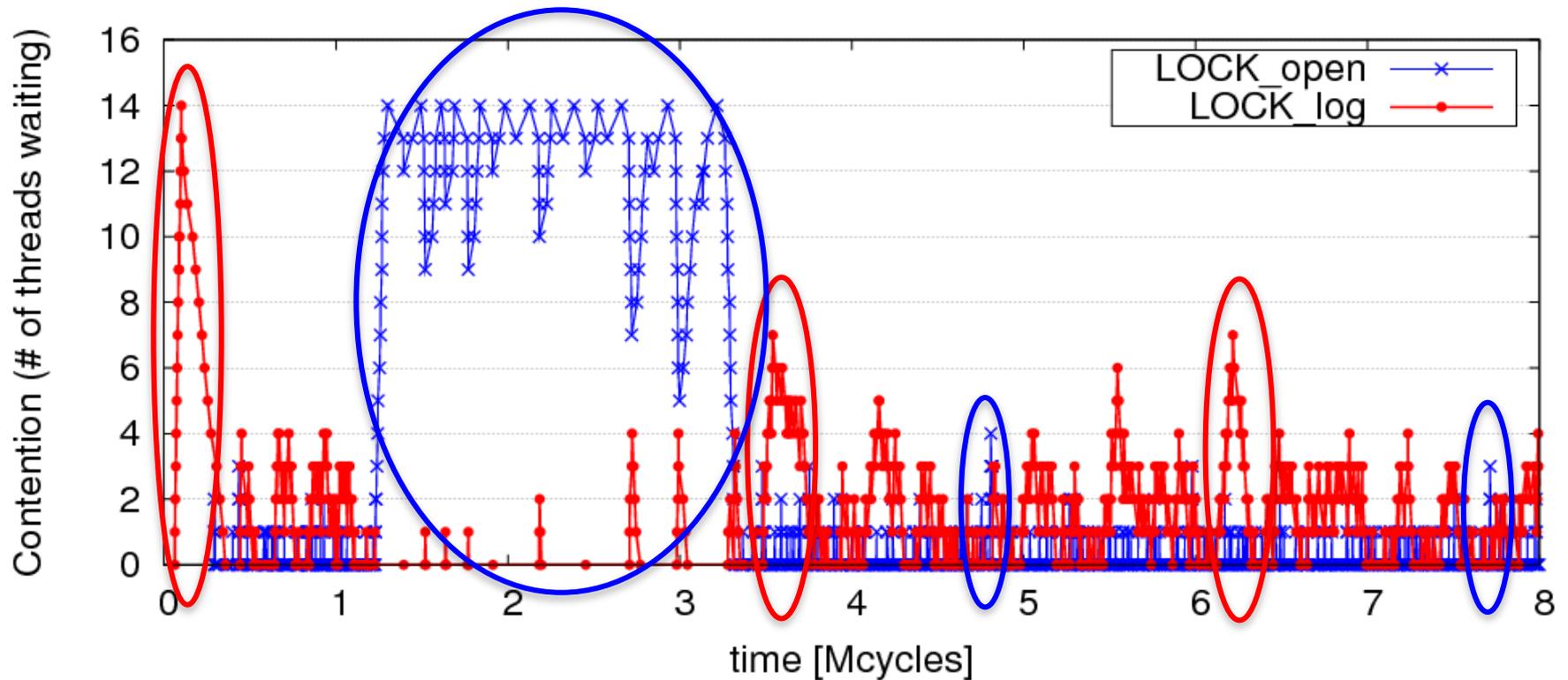
until A is empty



Lock A is limiter

Limiting Bottlenecks Do Change on Real Applications

MySQL running Sysbench queries, 16 threads



Outline

- Executive Summary
- The Problem: Bottlenecks
- Previous Work
- Bottleneck Identification and Scheduling
- Evaluation
- Conclusions

Previous Work

- Asymmetric CMP (ACMP) proposals [Annavaram+, ISCA'05] [Morad+, Comp. Arch. Letters'06] [Suleman+, Tech. Report'07]
 - Accelerate **only the Amdahl's bottleneck**
- Accelerated Critical Sections (ACS) [Suleman+, ASPLOS'09]
 - Accelerate **only critical sections**
 - **Does not take into account importance** of critical sections
- Feedback-Directed Pipelining (FDP) [Suleman+, PACT'10 and PhD thesis'11]
 - Accelerate **only stages with lowest throughput**
 - **Slow to adapt** to phase changes (software based library)

No previous work can accelerate all three types of bottlenecks or quickly adapts to fine-grain changes in the *importance* of bottlenecks

Our goal: general mechanism to identify performance-limiting bottlenecks of any type and accelerate them on an ACMP

Outline

- Executive Summary
- The Problem: Bottlenecks
- Previous Work
- Bottleneck Identification and Scheduling (BIS)
- Methodology
- Results
- Conclusions

Bottleneck Identification and Scheduling (BIS)

- Key insight:
 - Thread waiting reduces parallelism and is likely to reduce performance
 - Code causing the most thread waiting → likely critical path

- Key idea:
 - Dynamically identify bottlenecks that cause the most thread waiting
 - Accelerate them (using powerful cores in an ACMP)

Bottleneck Identification and Scheduling (BIS)

Compiler/Library/Programmer

1. Annotate *bottleneck* code
2. Implement *waiting* for bottlenecks

Binary containing
BIS instructions

Hardware

1. Measure *thread waiting cycles (TWC)* for each bottleneck
2. Accelerate bottleneck(s) with the highest TWC

Critical Sections: Code Modifications

...

while cannot acquire lock

 Wait loop for watch_addr

acquire lock

...

release lock

...

Critical Sections: Code Modifications

...

BottleneckCall *bid*, targetPC

... Wait loop for watch_addr

targetPC:

while cannot acquire lock

... **BottleneckWait** *bid*, watch_addr

acquire lock

...

release lock

BottleneckReturn *bid*

Critical Sections: Code Modifications

targetPC: ...
BottleneckCall *bid*, targetPC
... Wait loop for watch_addr
while cannot acquire lock
... BottleneckWait *bid*, watch_addr
acquire lock
...
release lock
BottleneckReturn *bid*

Used to enable
acceleration

Used to keep track of
waiting cycles

Barriers: Code Modifications

...

BottleneckCall *bid*, targetPC

enter barrier

while not all threads in barrier

BottleneckWait *bid*, watch_addr

exit barrier

...

targetPC: code running for the barrier

...

BottleneckReturn *bid*

Pipeline Stages: Code Modifications

BottleneckCall *bid*, targetPC

...

targetPC:

while not done

 while empty queue

BottleneckWait prev_bid

 dequeue work

 do the work ...

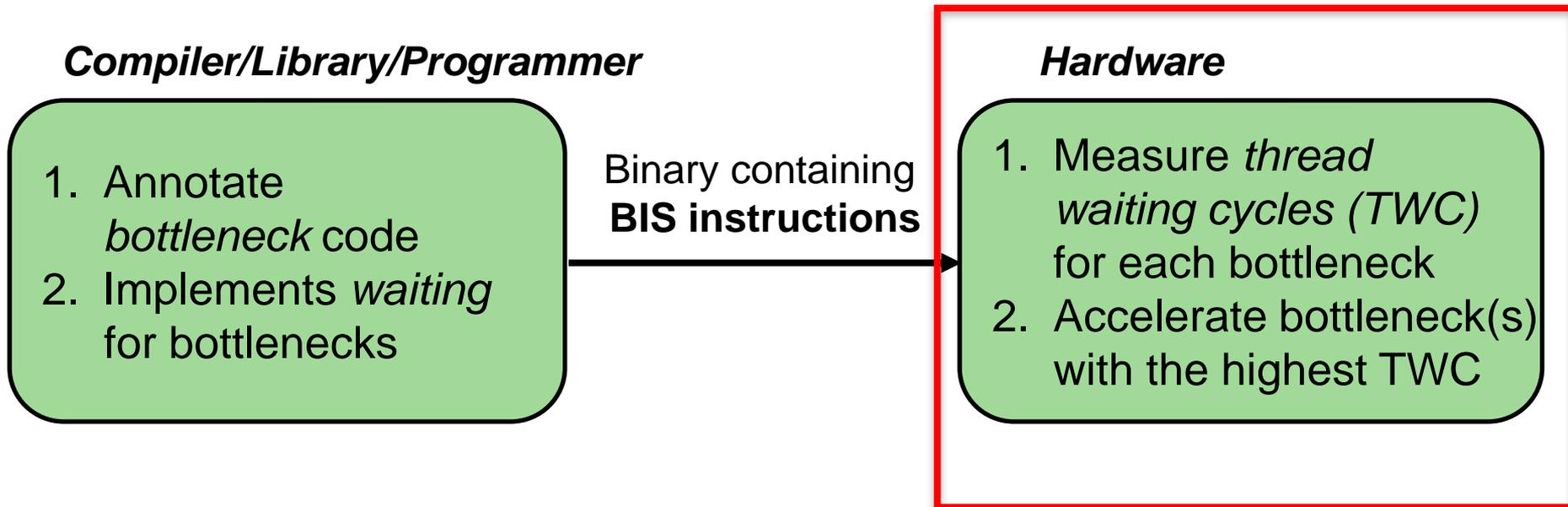
 while full queue

BottleneckWait next_bid

 enqueue next work

BottleneckReturn *bid*

Bottleneck Identification and Scheduling (BIS)



BIS: Hardware Overview

- Performance-limiting bottleneck **identification and acceleration are independent tasks**
- Acceleration can be accomplished in multiple ways
 - Increasing core frequency/voltage
 - Prioritization in shared resources [Ebrahimi+, MICRO'11]
 - **Migration to faster cores in an Asymmetric CMP**

Small core	Small core	Large core	
Small core	Small core		
Small core	Small core	Small core	Small core
Small core	Small core	Small core	Small core

Bottleneck Identification and Scheduling (BIS)

Compiler/Library/Programmer

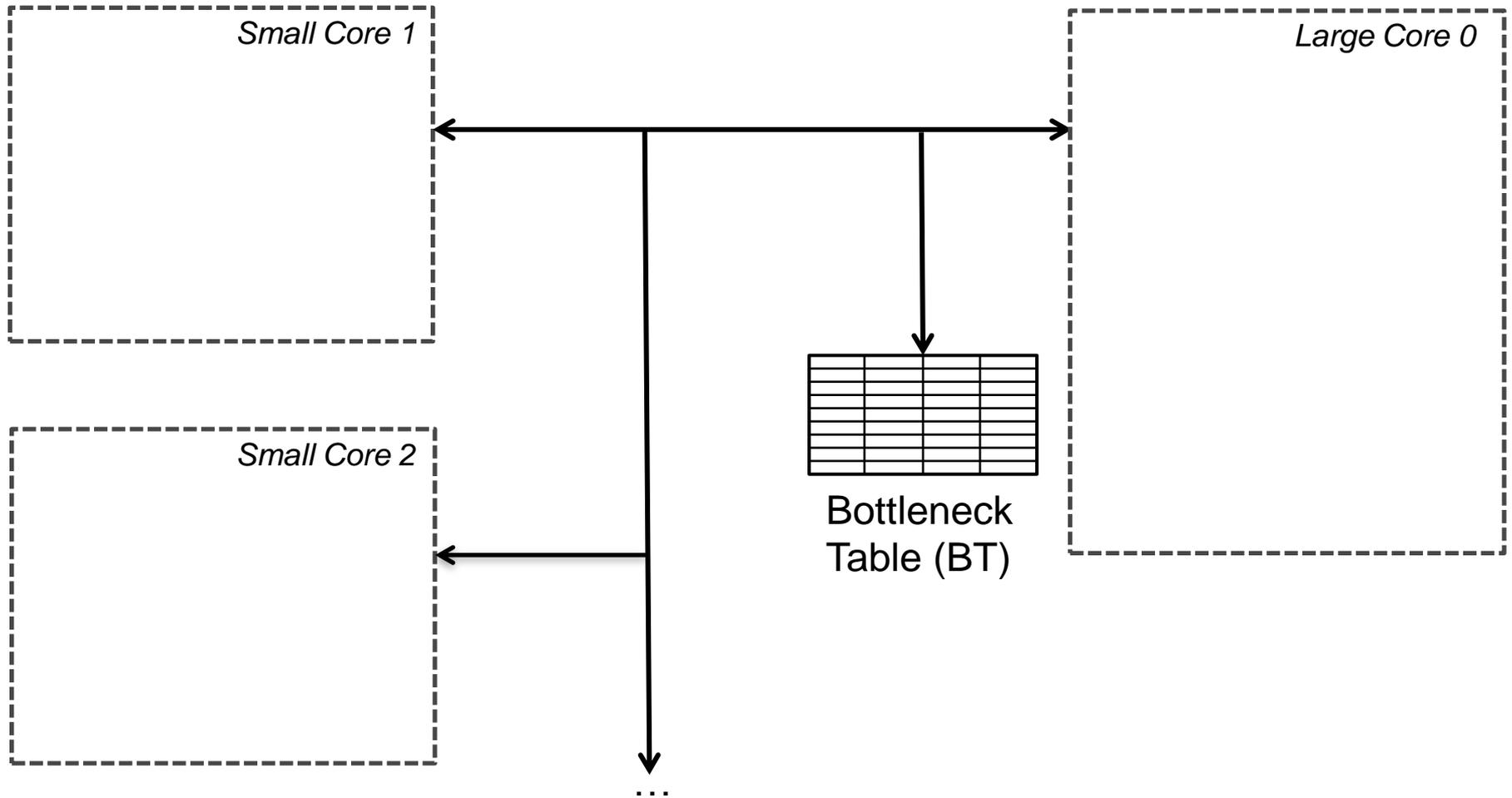
1. Annotate *bottleneck* code
2. Implements *waiting* for bottlenecks

Binary containing
BIS instructions

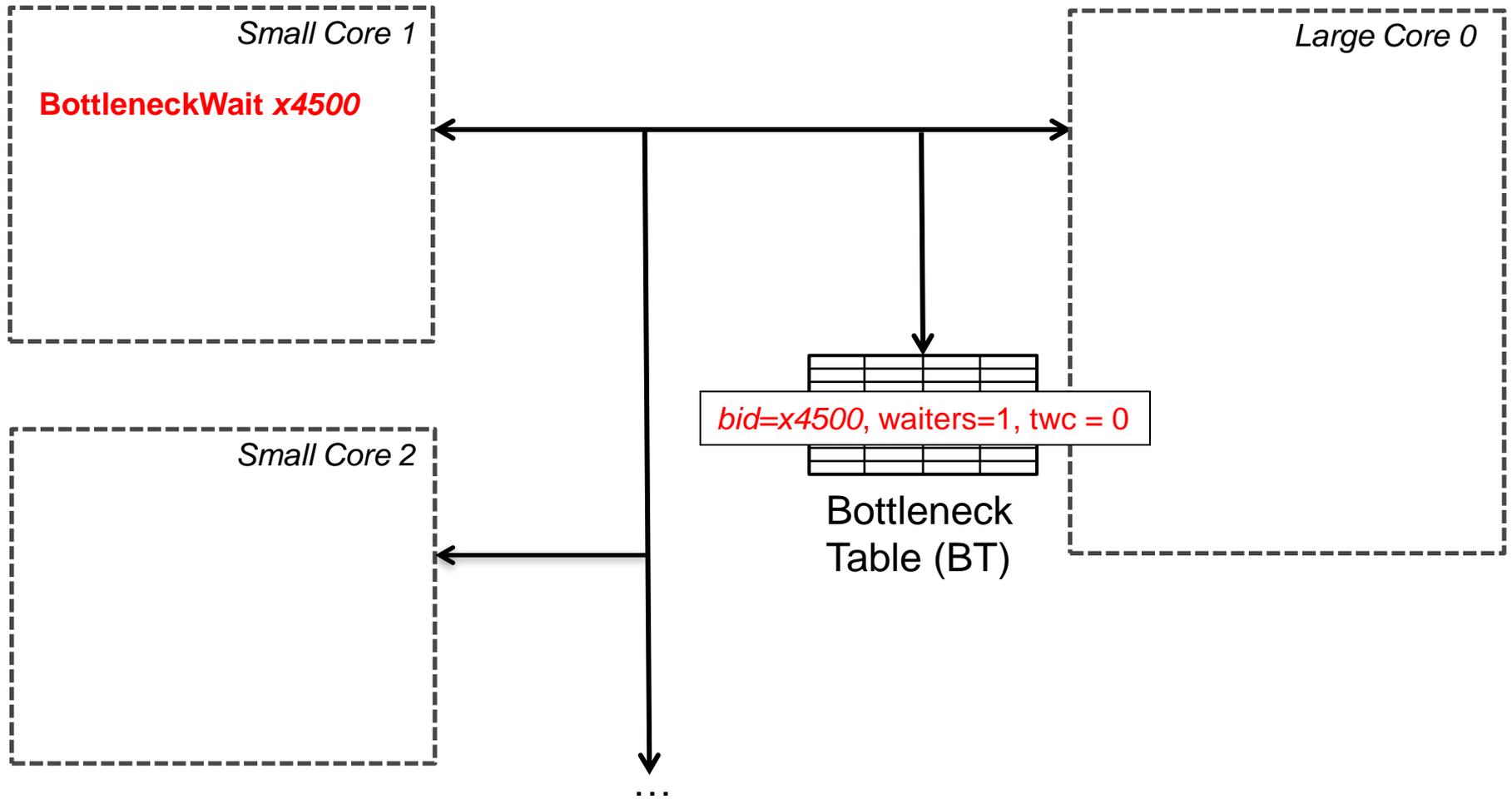
Hardware

1. Measure *thread waiting cycles (TWC)* for each bottleneck
2. Accelerate bottleneck(s) with the highest TWC

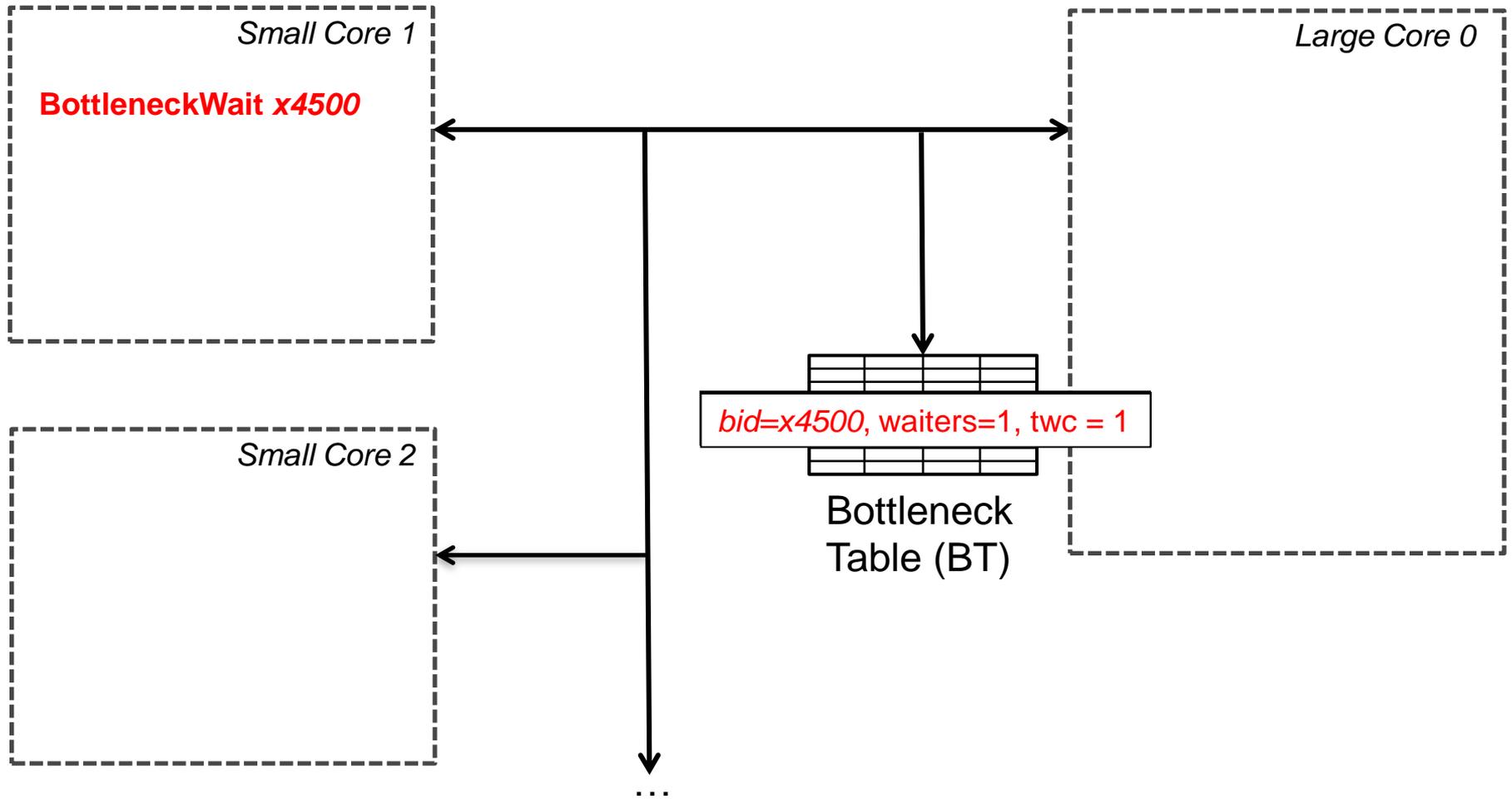
Determining Thread Waiting Cycles for Each Bottleneck



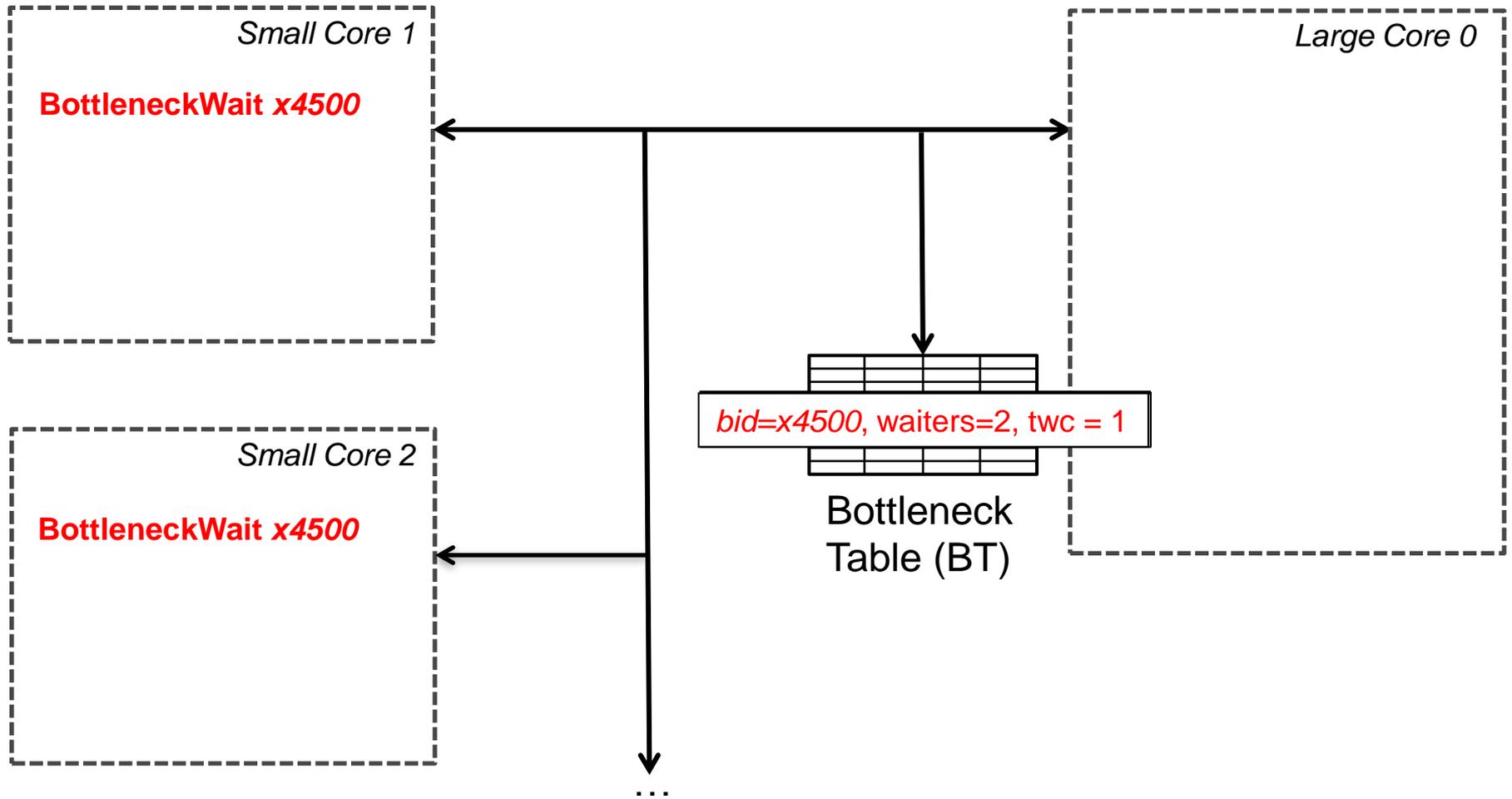
Determining Thread Waiting Cycles for Each Bottleneck



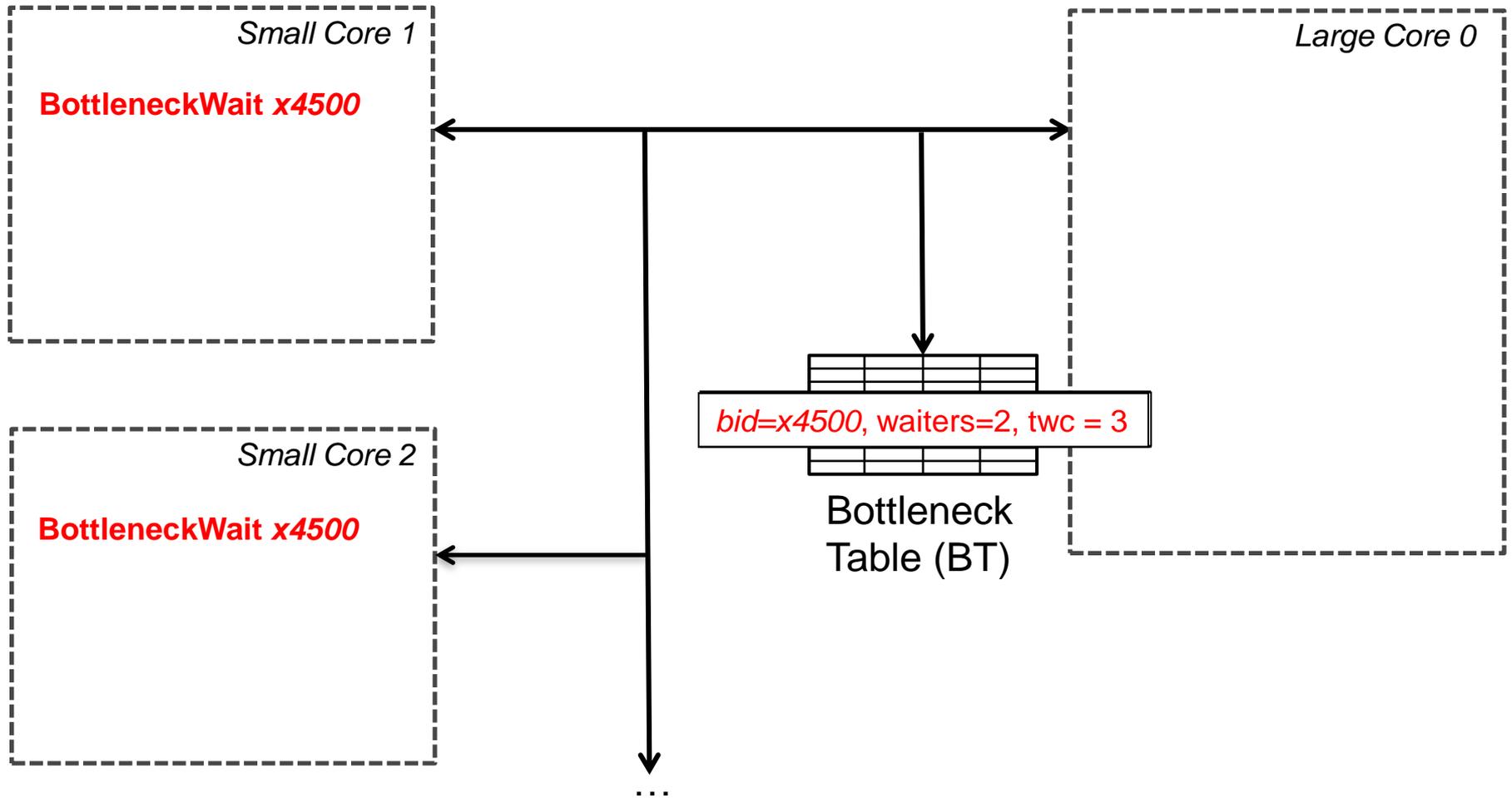
Determining Thread Waiting Cycles for Each Bottleneck



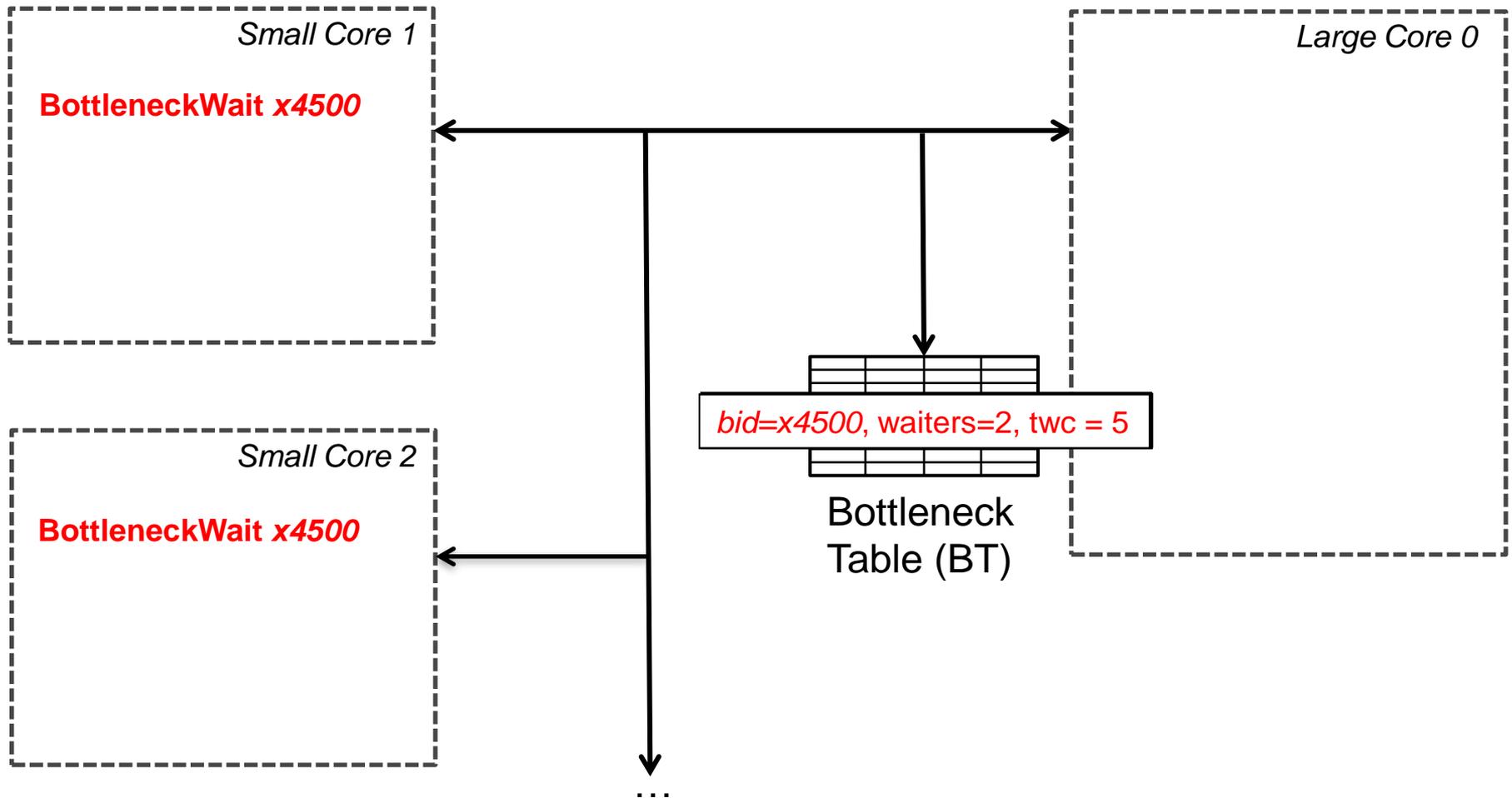
Determining Thread Waiting Cycles for Each Bottleneck



Determining Thread Waiting Cycles for Each Bottleneck



Determining Thread Waiting Cycles for Each Bottleneck



Bottleneck Identification and Scheduling (BIS)

Compiler/Library/Programmer

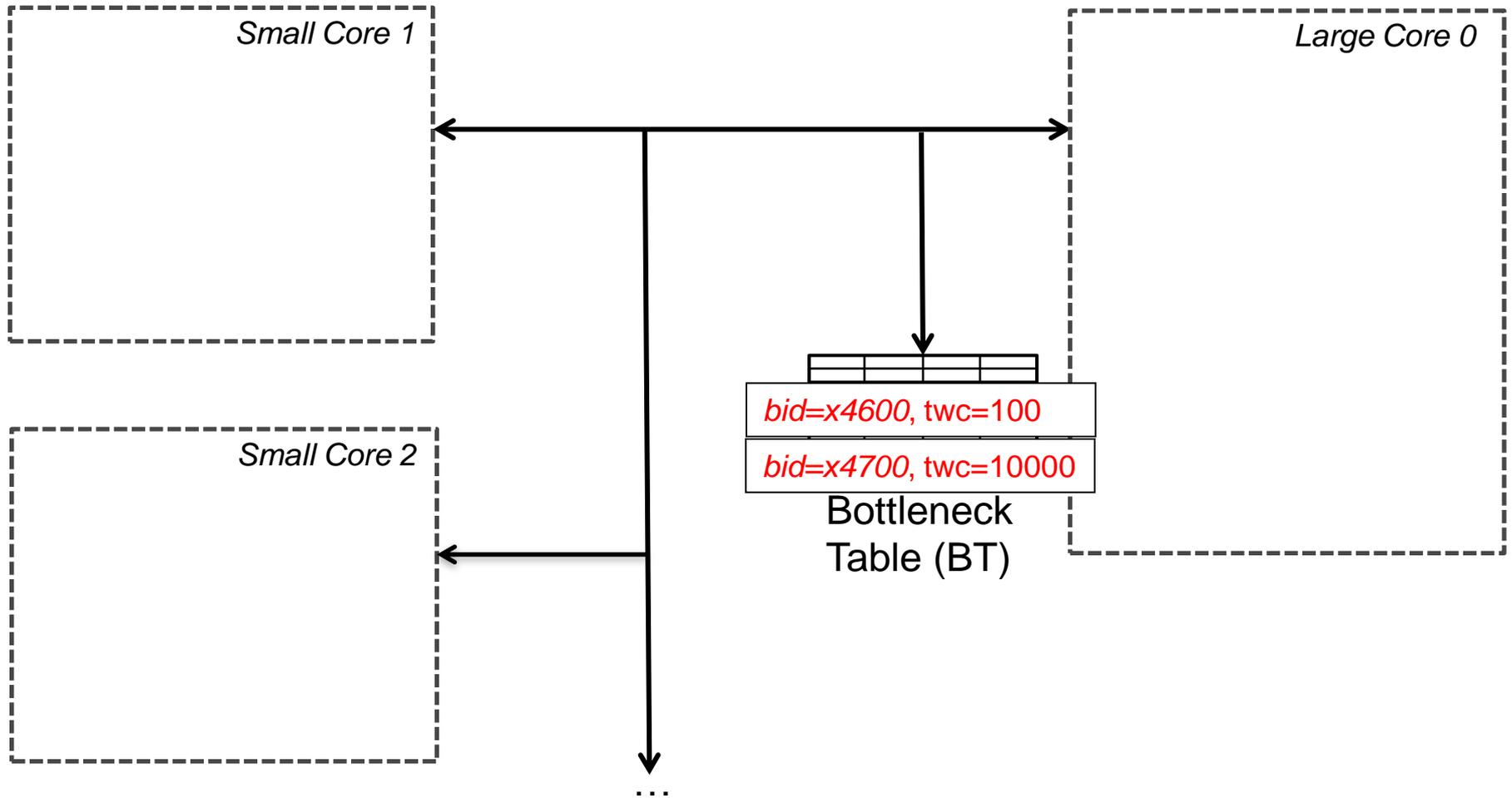
1. Annotate *bottleneck* code
2. Implements *waiting* for bottlenecks

Binary containing
BIS instructions

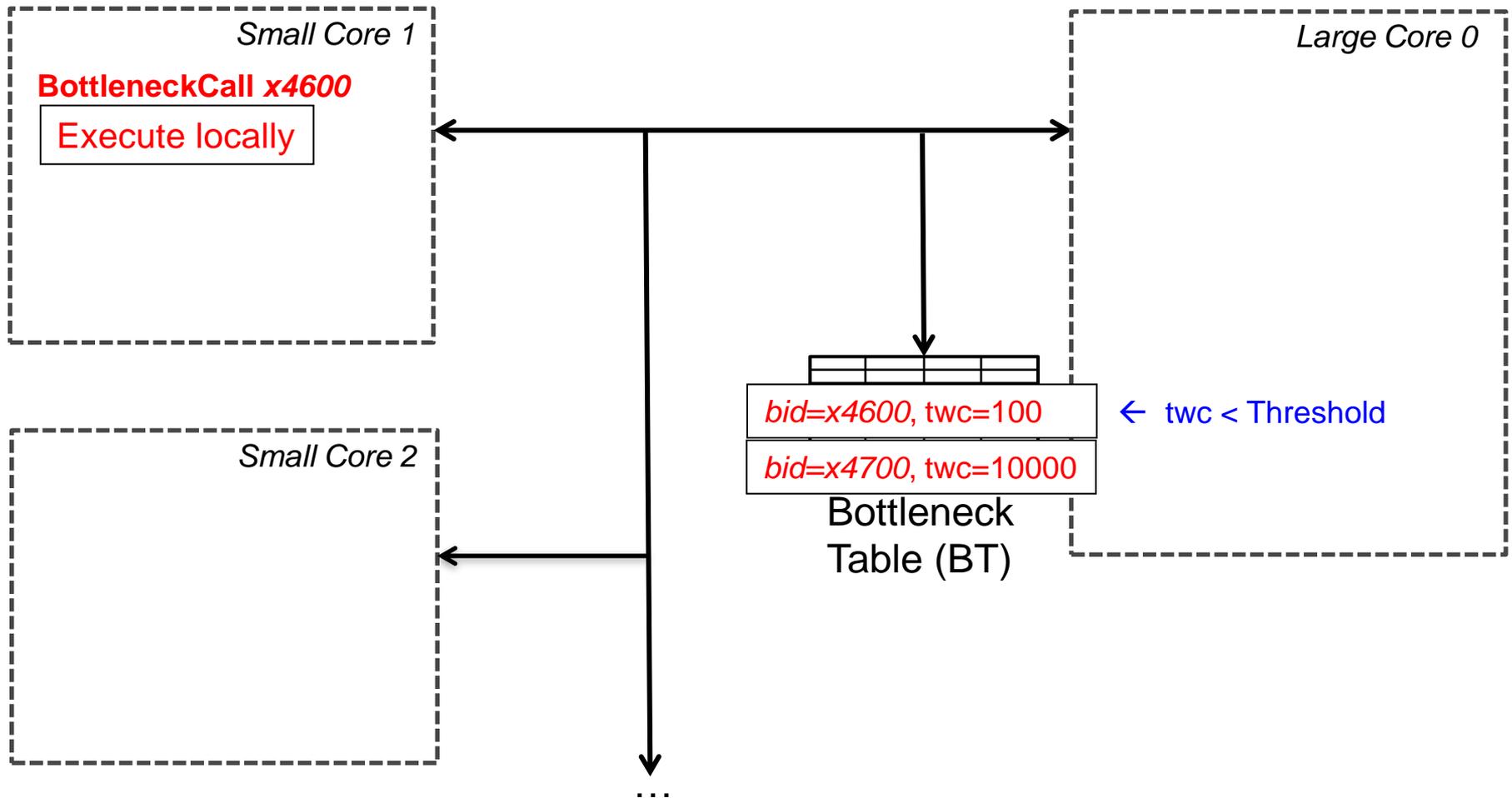
Hardware

1. Measure *thread waiting cycles (TWC)* for each bottleneck
2. Accelerate bottleneck(s) with the highest TWC

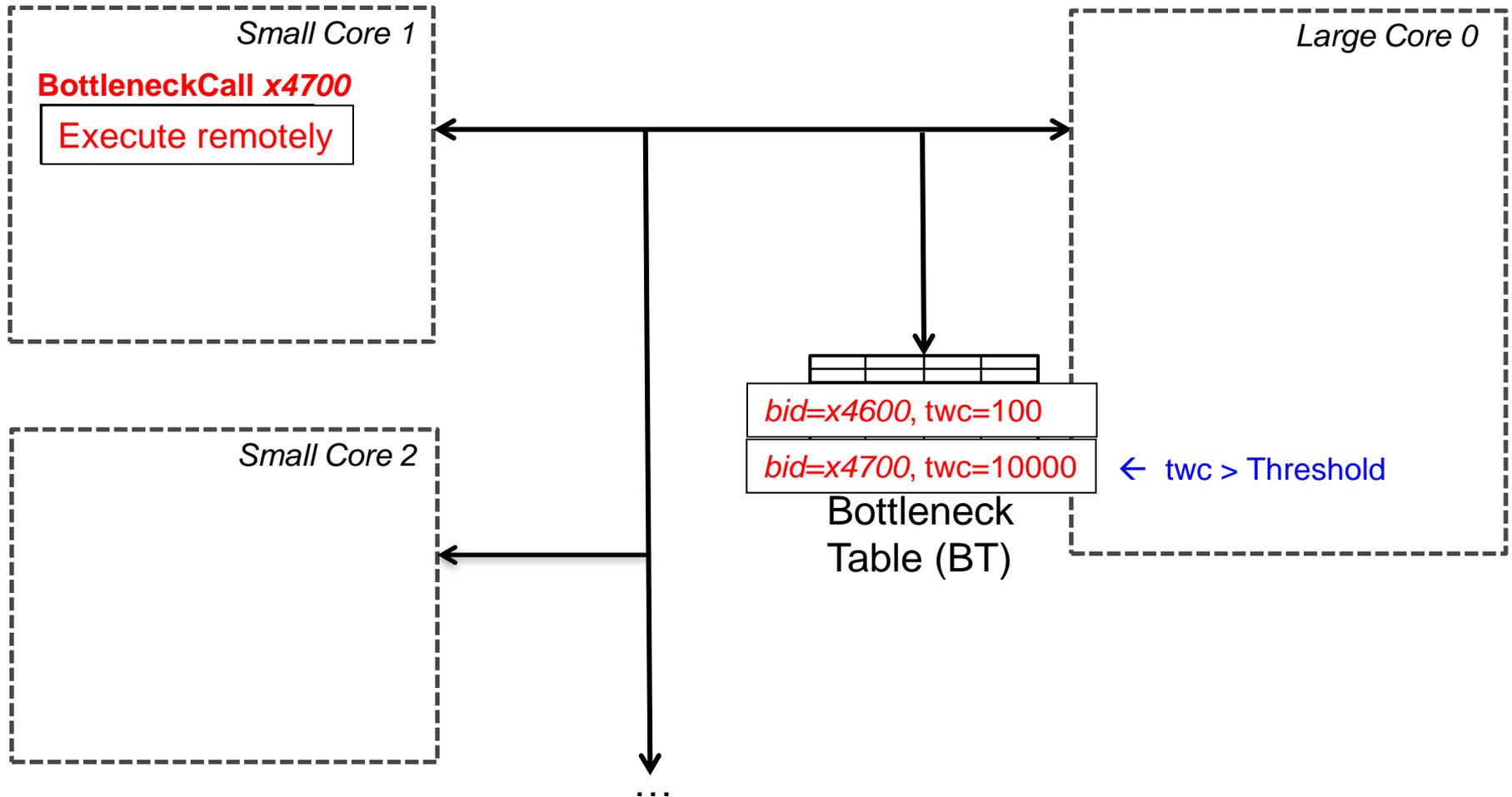
Bottleneck Acceleration



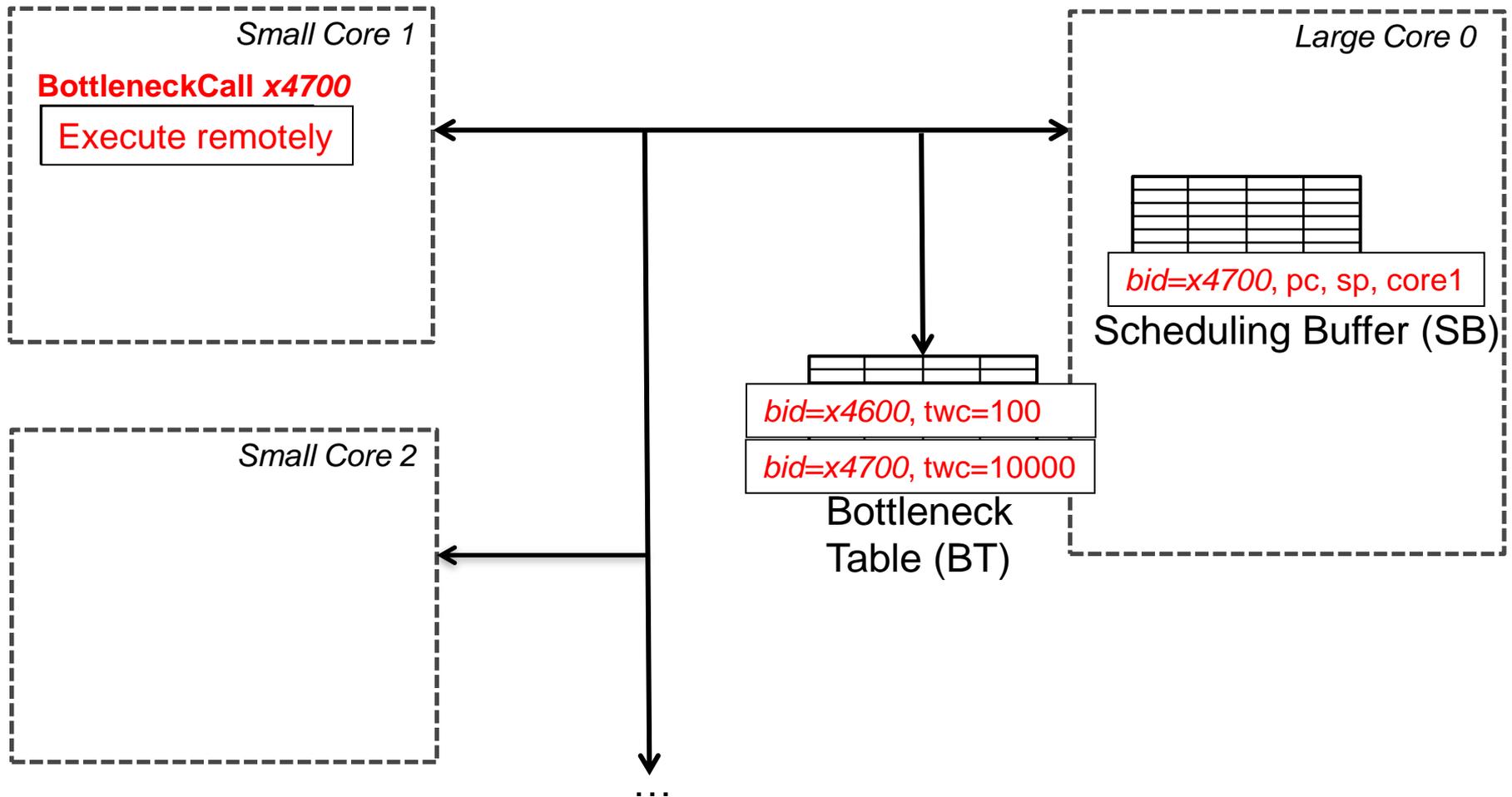
Bottleneck Acceleration



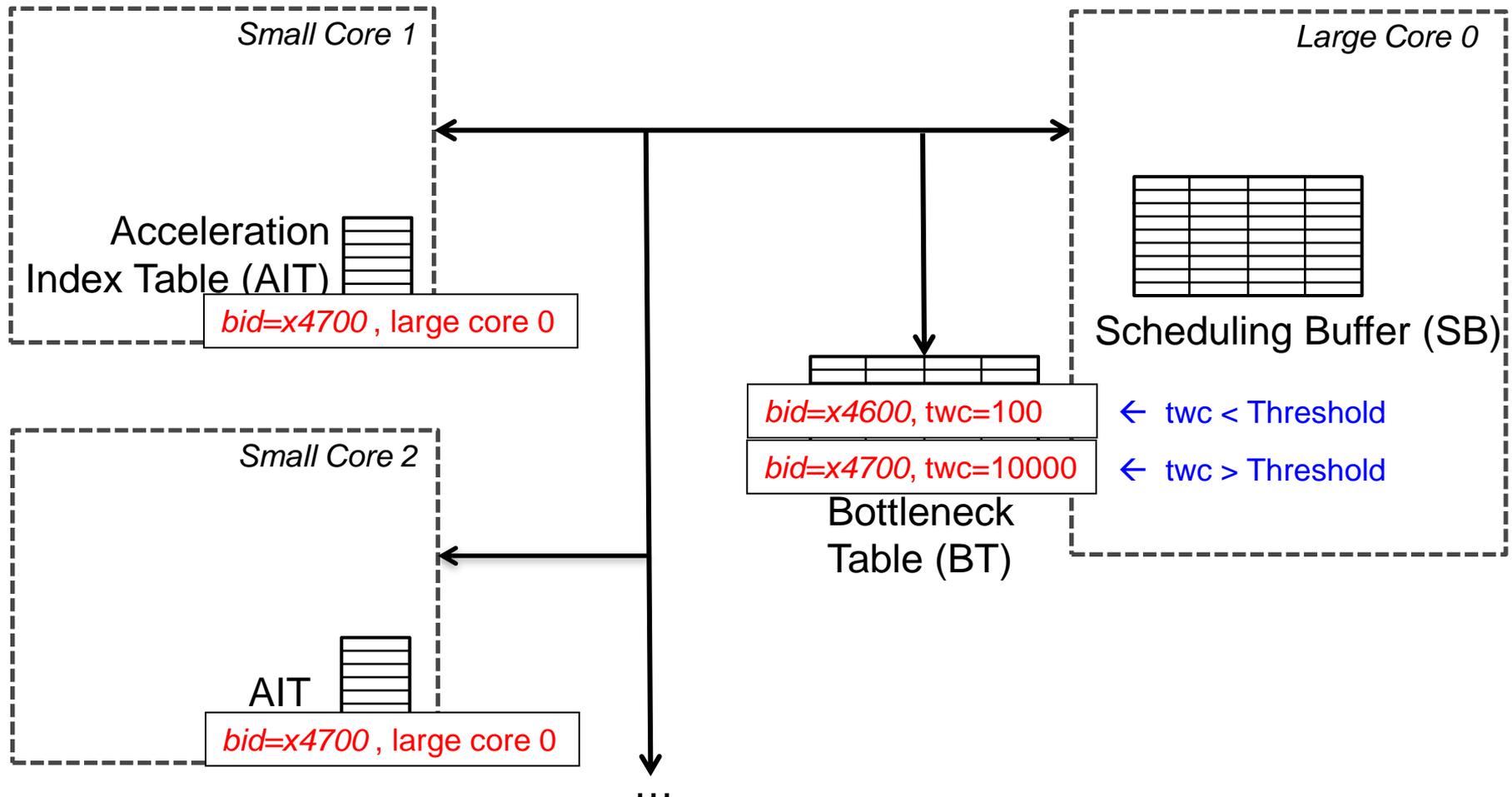
Bottleneck Acceleration



Bottleneck Acceleration



Bottleneck Acceleration



BIS Mechanisms

- Basic mechanisms for BIS:
 - Determining Thread Waiting Cycles ✓
 - Accelerating Bottlenecks ✓
- Mechanisms to improve performance and generality of BIS:
 - Dealing with false serialization
 - Preemptive acceleration
 - Support for multiple large cores

False Serialization and Starvation

- **Observation:** Bottlenecks are picked from Scheduling Buffer in Thread Waiting Cycles order
- **Problem:** An independent bottleneck that is ready to execute has to wait for another bottleneck that has higher thread waiting cycles → **False serialization**
- **Starvation:** Extreme false serialization
- **Solution:** Large core detects when a bottleneck is ready to execute in the Scheduling Buffer but it cannot → sends the bottleneck back to the small core

Preemptive Acceleration

- **Observation:** A bottleneck executing on a small core can become the bottleneck with the highest thread waiting cycles
 - **Problem:** This bottleneck should really be accelerated (i.e., executed on the large core)
 - **Solution:** The Bottleneck Table detects the situation and sends a preemption signal to the small core. Small core:
 - saves register state on stack, ships the bottleneck to the large core
 - Main acceleration mechanism for barriers and pipeline stages
-

Support for Multiple Large Cores

- **Objective:** to accelerate independent bottlenecks
- Each large core has its own Scheduling Buffer (shared by all of its SMT threads)
- Bottleneck Table assigns each bottleneck to a fixed large core context to
 - preserve cache locality
 - avoid busy waiting
- Preemptive acceleration extended to send multiple instances of a bottleneck to different large core contexts

Hardware Cost

- Main structures:
 - Bottleneck Table (BT): global 32-entry associative cache, minimum-Thread-Waiting-Cycle replacement
 - Scheduling Buffers (SB): one table per large core, as many entries as small cores
 - Acceleration Index Tables (AIT): one 32-entry table per small core
- Off the critical path
- Total storage cost for 56-small-cores, 2-large-cores < 19 KB

BIS Performance Trade-offs

- Bottleneck **identification**:
 - Small cost: BottleneckWait instruction and Bottleneck Table
- Bottleneck **acceleration** on an ACMP (execution migration):
 - Faster bottleneck execution vs. fewer parallel threads
 - Acceleration offsets loss of parallel throughput with large core counts
 - Better shared data locality vs. worse private data locality
 - Shared data stays on large core (good)
 - Private data migrates to large core (bad, but latency hidden with Data Marshaling [Suleman+, ISCA' 10])
 - Benefit of acceleration vs. migration latency
 - Migration latency usually hidden by waiting (good)
 - Unless bottleneck not contended (bad, but likely to not be on critical path)

Outline

- Executive Summary
- The Problem: Bottlenecks
- Previous Work
- Bottleneck Identification and Scheduling
- Evaluation
- Conclusions

Methodology

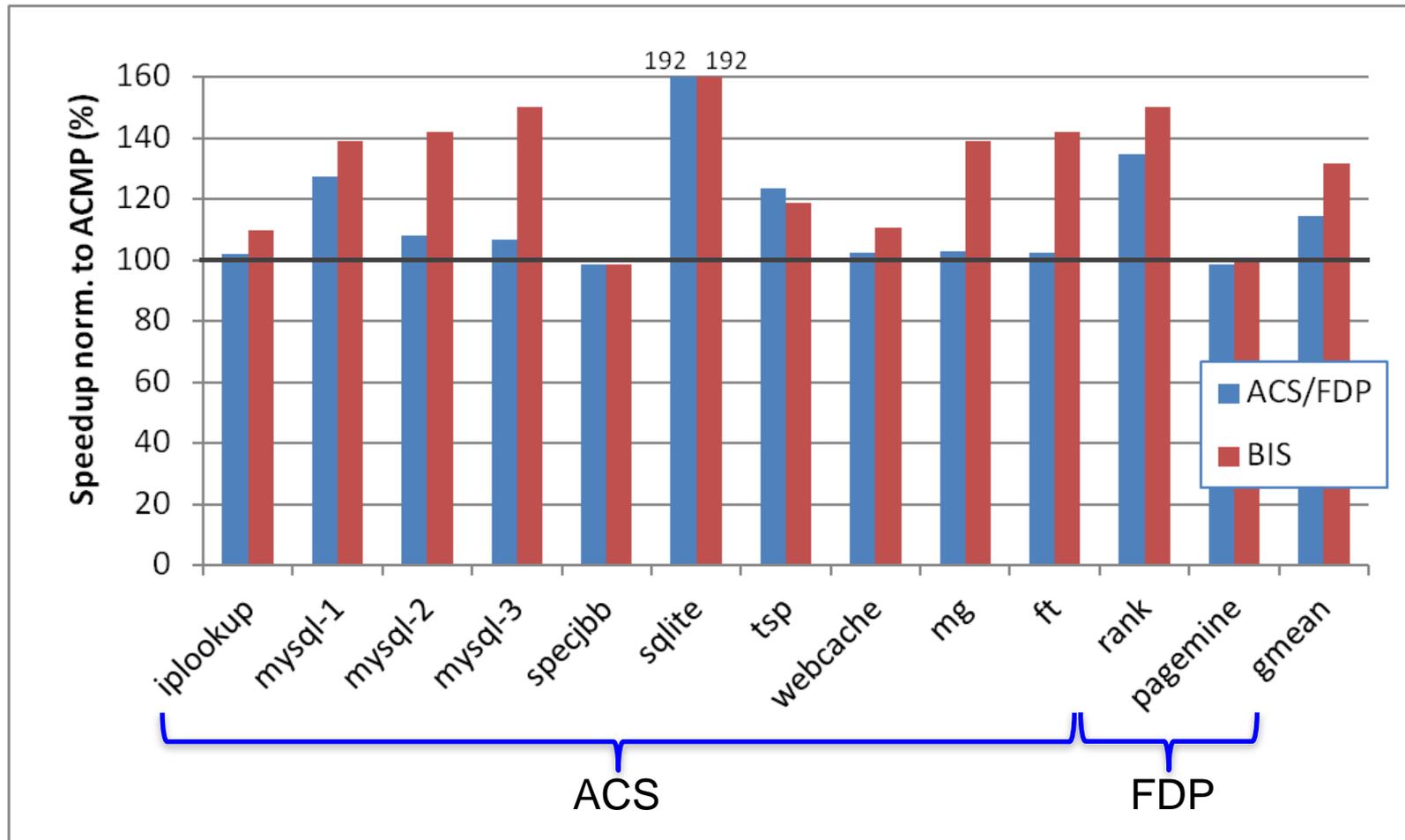
- Workloads: 8 critical section intensive, 2 barrier intensive and 2 pipeline-parallel applications
 - Data mining kernels, scientific, database, web, networking, specjbb
- Cycle-level multi-core x86 simulator
 - 8 to 64 small-core-equivalent area, 0 to 3 large cores, SMT
 - 1 large core is area-equivalent to 4 small cores
- Details:
 - Large core: 4GHz, out-of-order, 128-entry ROB, 4-wide, 12-stage
 - Small core: 4GHz, in-order, 2-wide, 5-stage
 - Private 32KB L1, private 256KB L2, shared 8MB L3
 - On-chip interconnect: Bi-directional ring, 2-cycle hop latency

Comparison Points (Area-Equivalent)

- SCMP (Symmetric CMP)
 - All small cores
 - Results in the paper
- ACMP (Asymmetric CMP)
 - Accelerates only Amdahl's serial portions
 - Our baseline
- ACS (Accelerated Critical Sections)
 - Accelerates only critical sections and Amdahl's serial portions
 - Applicable to multithreaded workloads
([iplookup](#), [mysql](#), [specjbb](#), [sqlite](#), [tsp](#), [webcache](#), [mg](#), [ft](#))
- FDP (Feedback-Directed Pipelining)
 - Accelerates only slowest pipeline stages
 - Applicable to pipeline-parallel workloads ([rank](#), [pagemine](#))

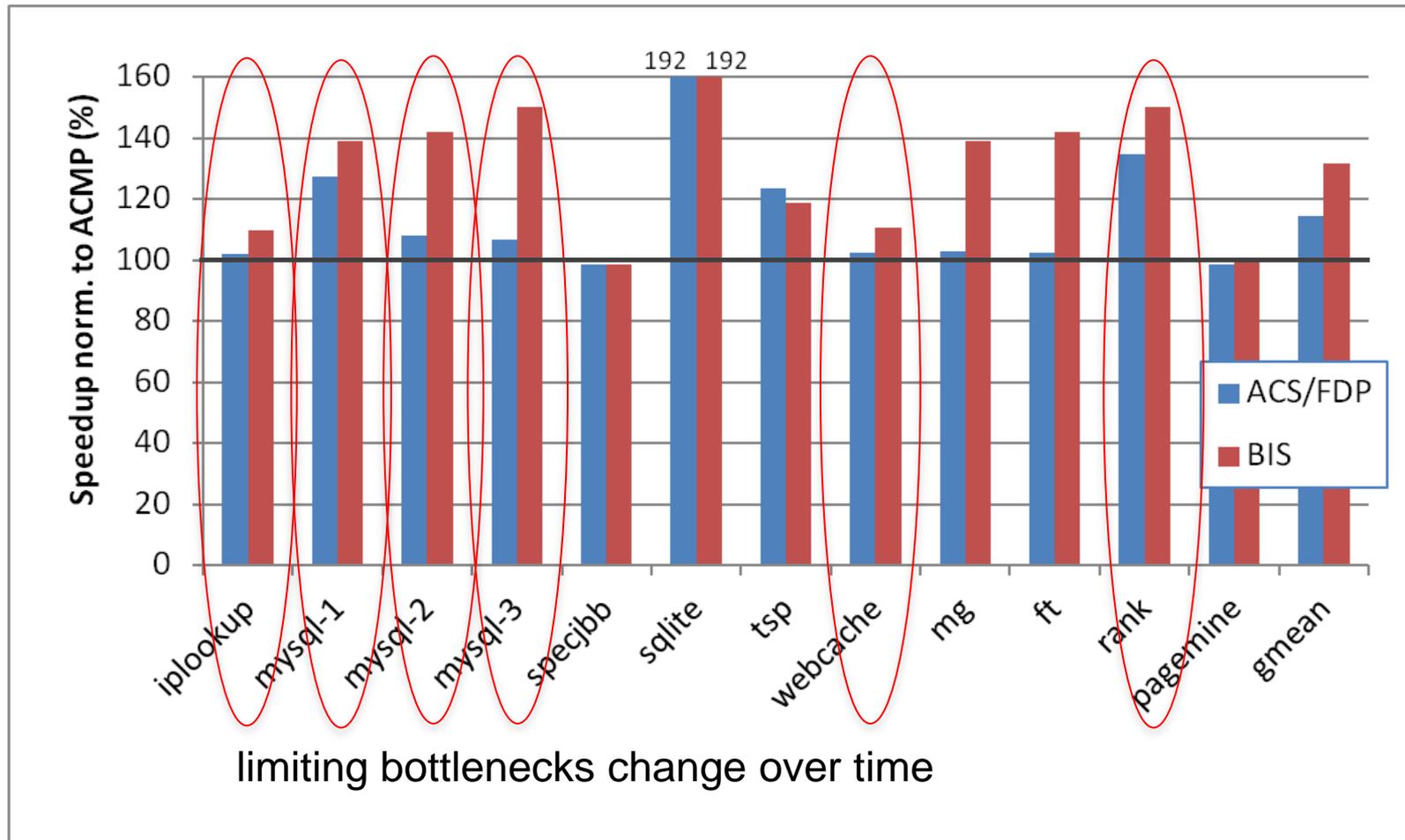
BIS Performance Improvement

Optimal number of threads, 28 small cores, 1 large core



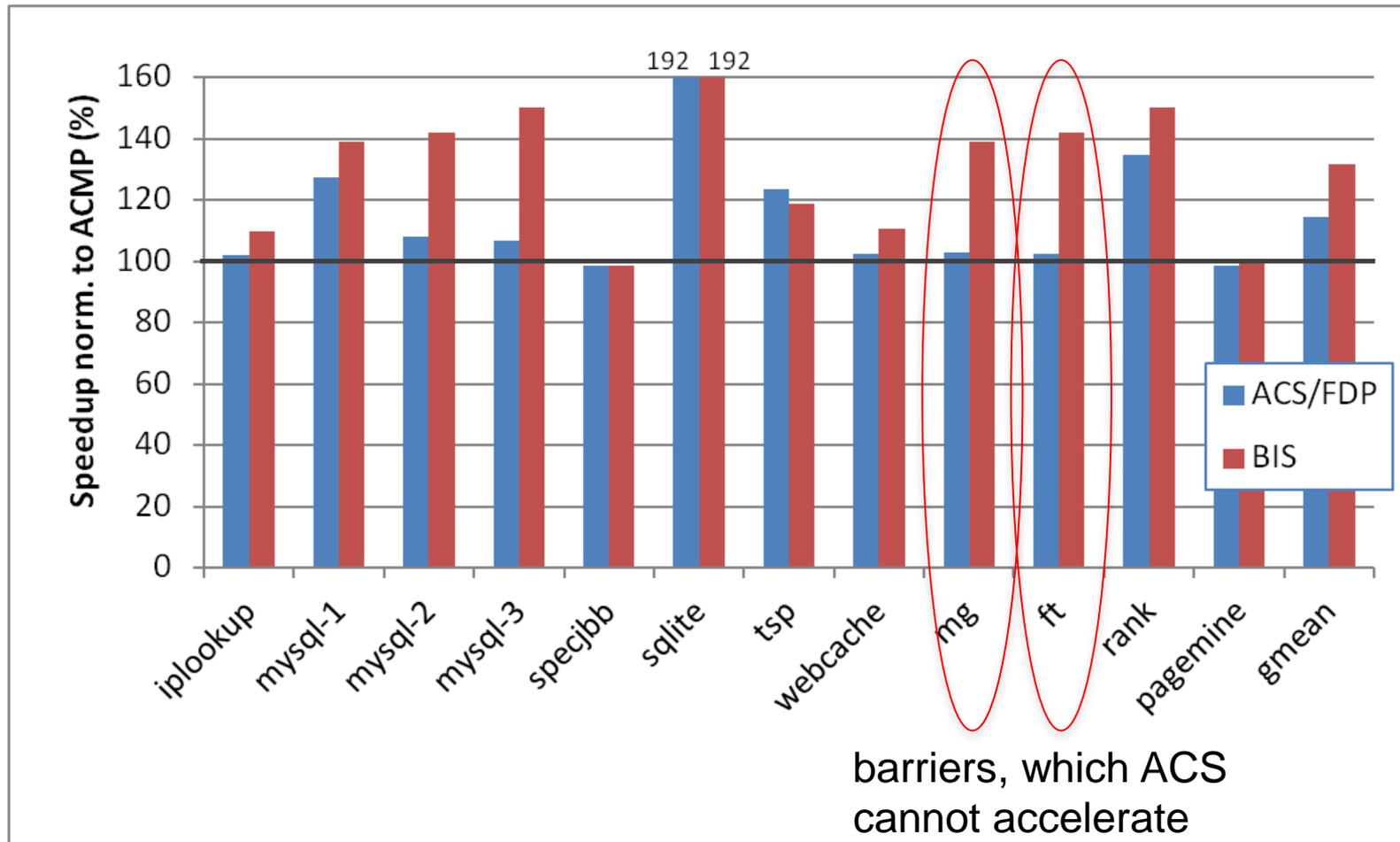
BIS Performance Improvement

Optimal number of threads, 28 small cores, 1 large core



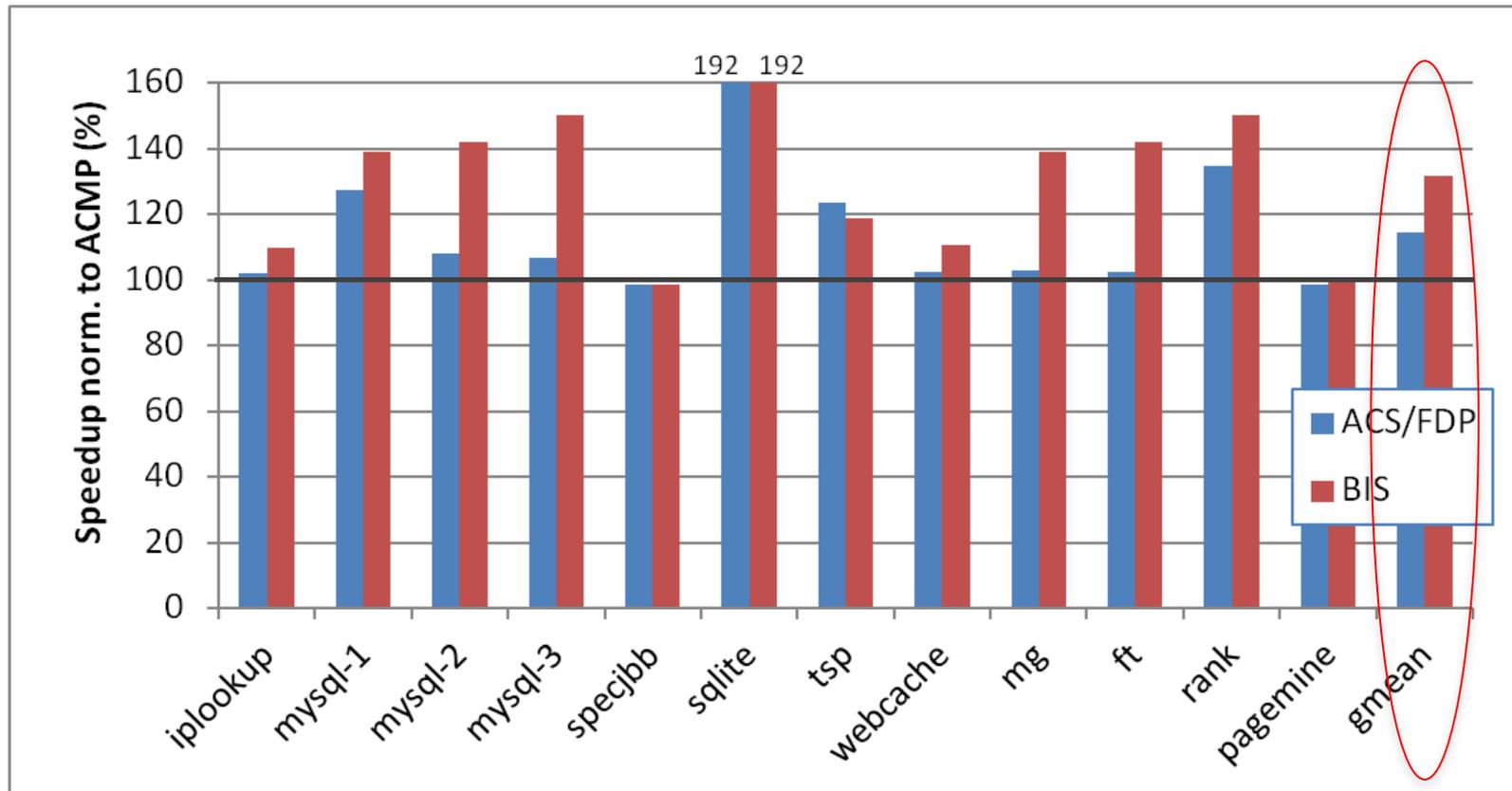
BIS Performance Improvement

Optimal number of threads, 28 small cores, 1 large core



BIS Performance Improvement

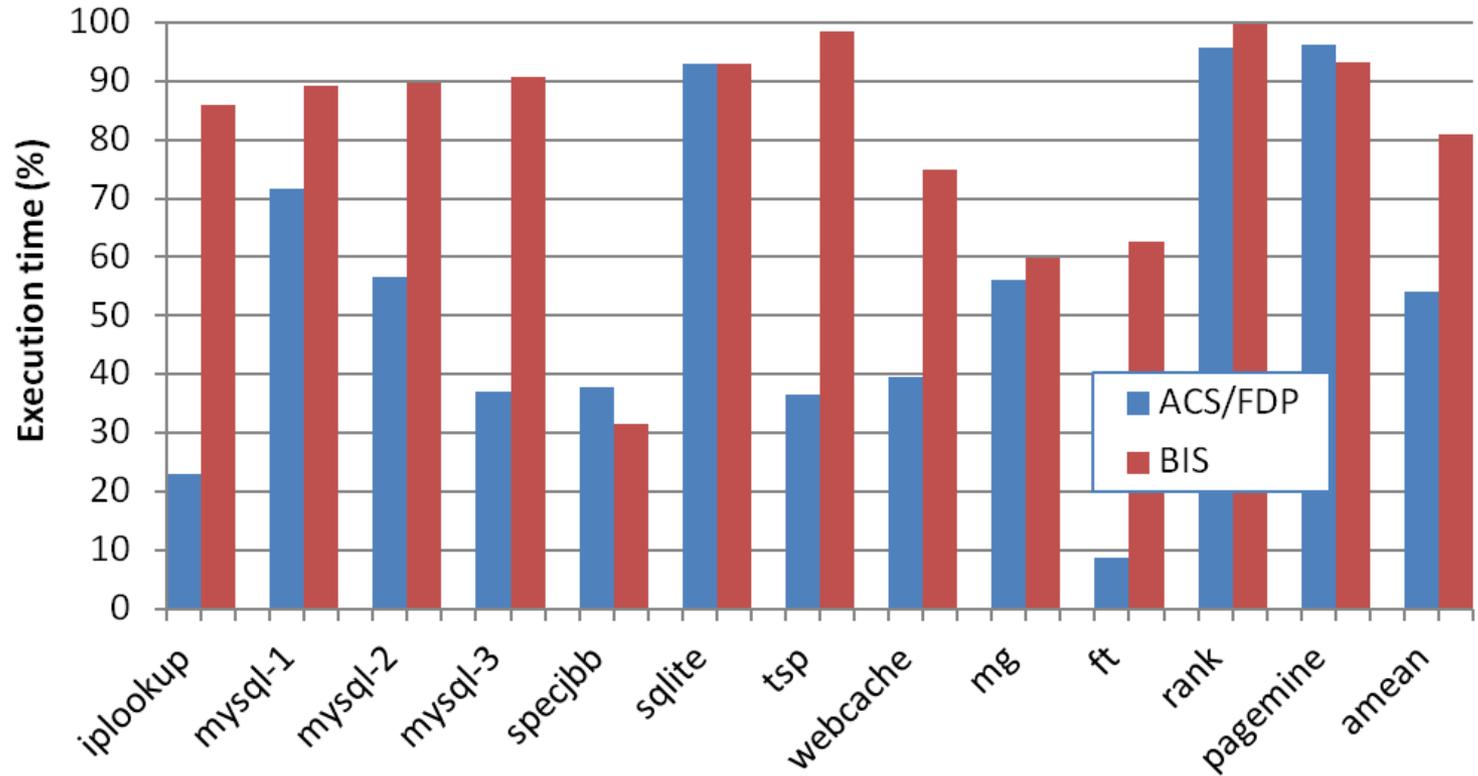
Optimal number of threads, 28 small cores, 1 large core



- BIS outperforms ACS/FDP by 15% and ACMP by 32%
- BIS improves scalability on 4 of the benchmarks

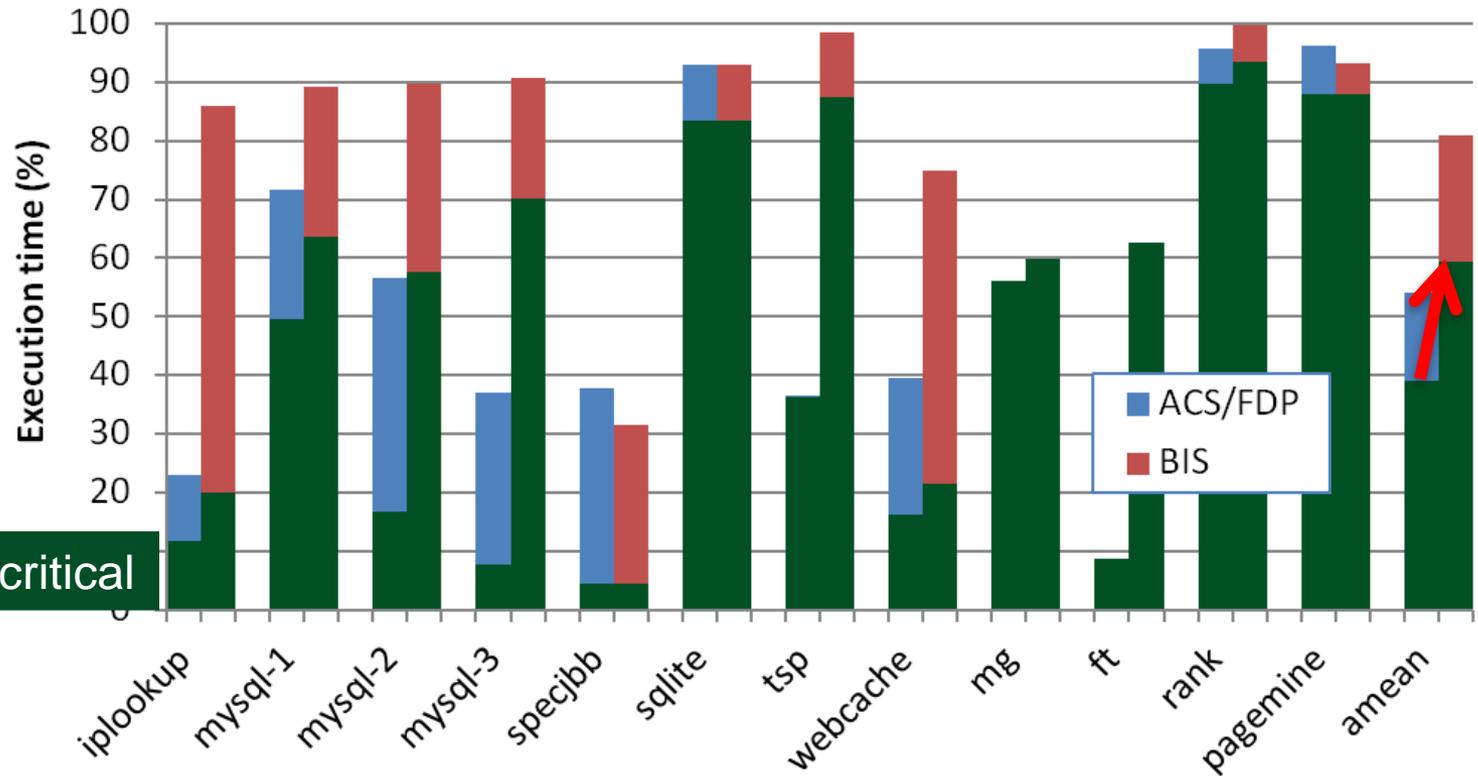
Why Does BIS Work?

Fraction of execution time spent on predicted-important bottlenecks



Why Does BIS Work?

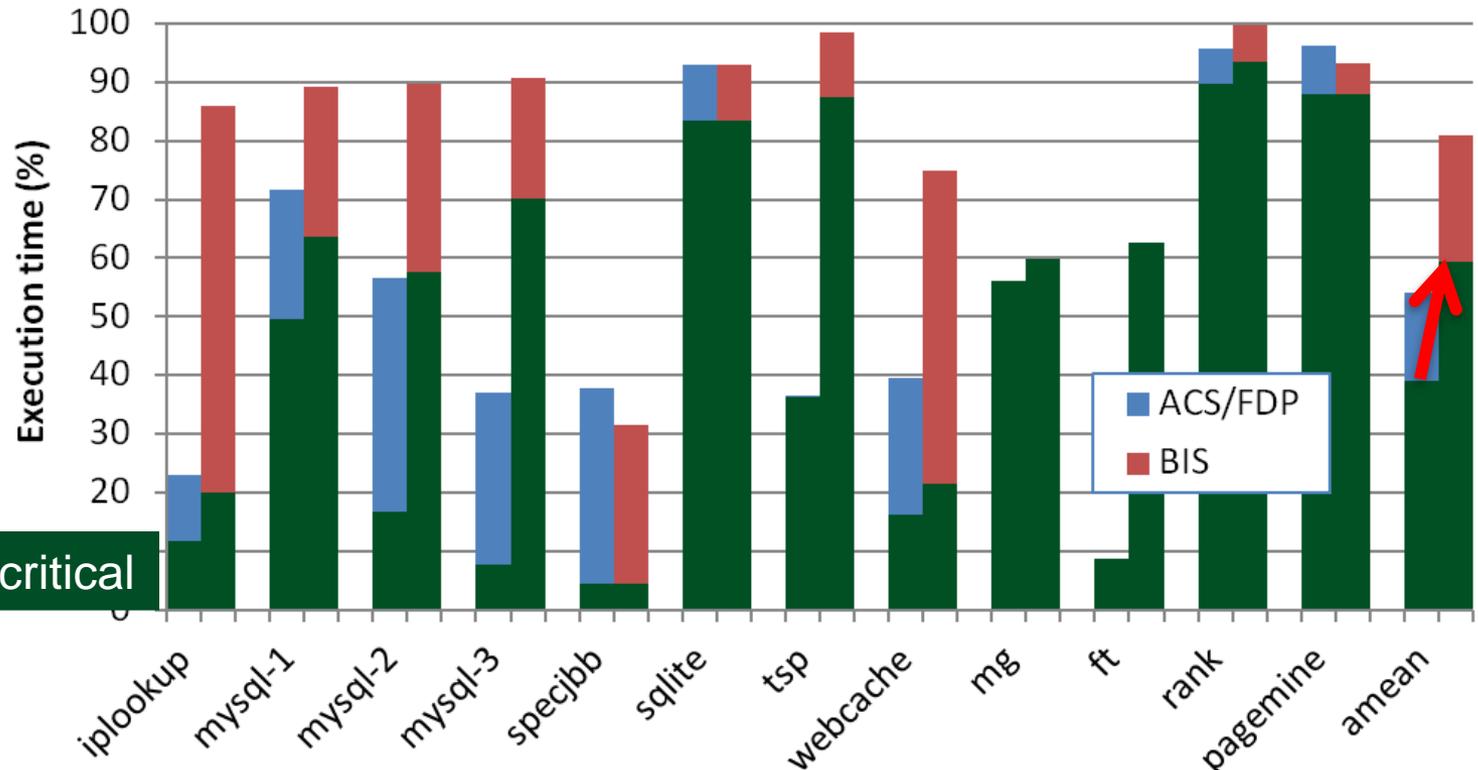
Fraction of execution time spent on predicted-important bottlenecks



Actually critical

Why Does BIS Work?

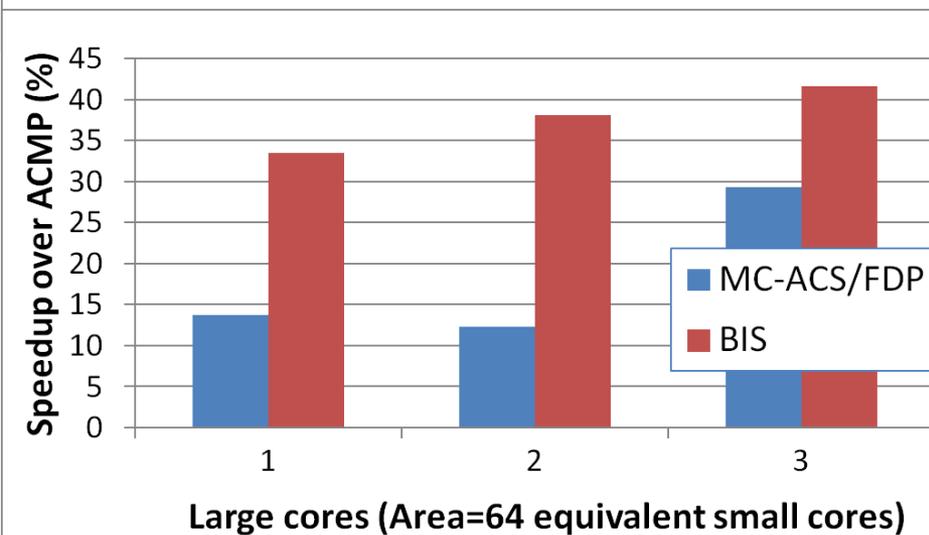
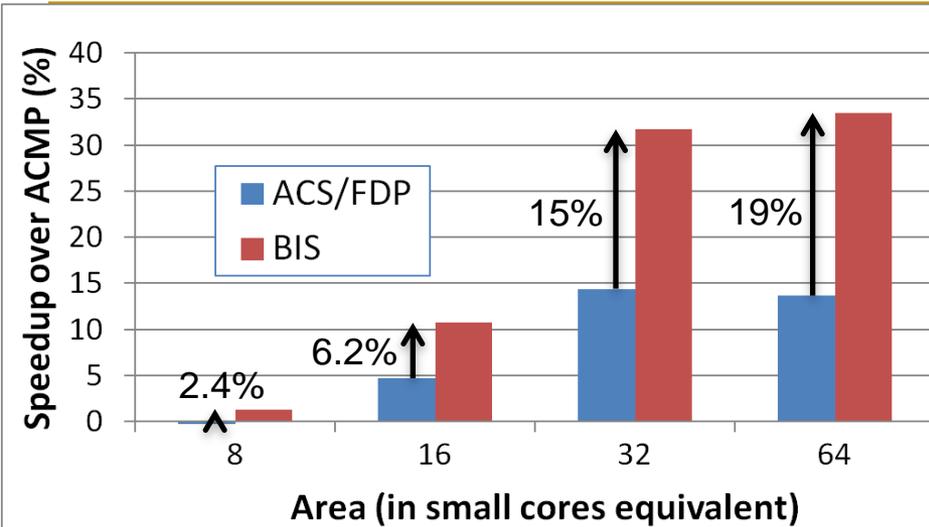
Fraction of execution time spent on predicted-important bottlenecks



Actually critical

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

Scaling Results



Performance increases with:

1) More small cores

- Contention due to bottlenecks increases
- Loss of parallel throughput due to large core reduces

2) More large cores

- Can accelerate independent bottlenecks
- *Without reducing parallel throughput (enough cores)*

Outline

- Executive Summary
- The Problem: Bottlenecks
- Previous Work
- Bottleneck Identification and Scheduling
- Evaluation
- Conclusions

Conclusions

- **Serializing bottlenecks of different types** limit performance of multithreaded applications: **Importance changes over time**
 - BIS is a hardware/software cooperative solution:
 - **Dynamically identifies bottlenecks** that cause the **most thread waiting** and **accelerates** them on large cores of an ACMP
 - Applicable to critical sections, barriers, pipeline stages
 - BIS improves application performance and scalability:
 - 15% speedup over ACS/FDP
 - Can accelerate multiple independent critical bottlenecks
 - Performance benefits increase with more cores
 - Provides **comprehensive fine-grained bottleneck acceleration for future ACMPs** without programmer effort
-

Thank you.

Bottleneck Identification and Scheduling in Multithreaded Applications

José A. Joao

M. Aater Suleman

Onur Mutlu

Yale N. Patt

Backup Slides

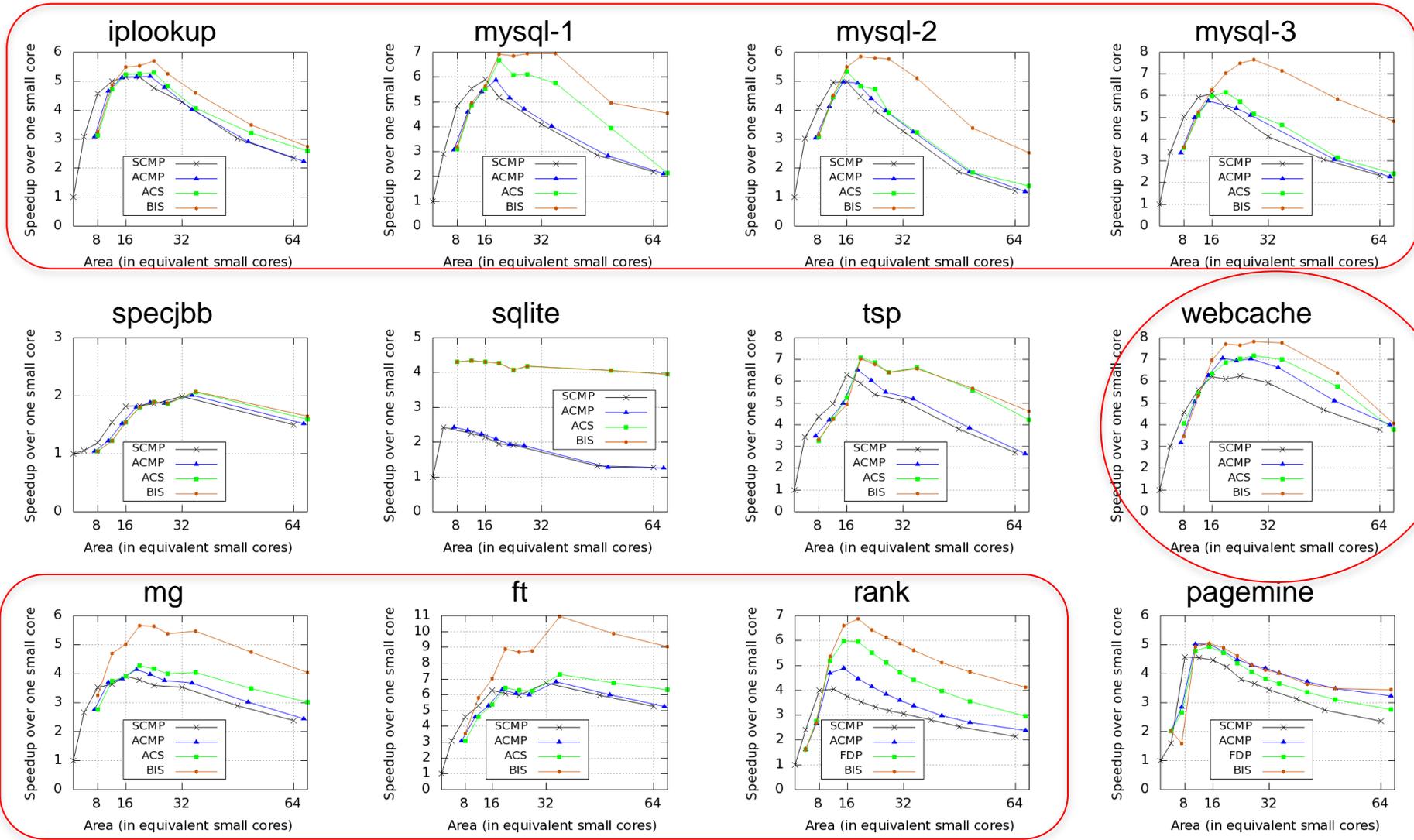
Major Contributions

- New bottleneck criticality predictor: thread waiting cycles
 - New mechanisms (compiler, ISA, hardware) to accomplish this
- Generality to multiple bottlenecks
- Fine-grained adaptivity of mechanisms
- Applicability to multiple cores

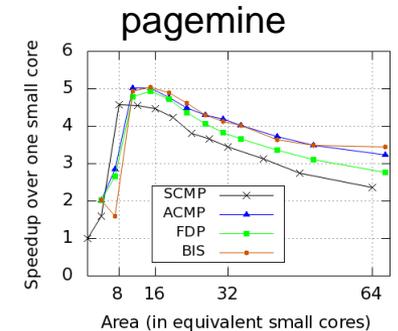
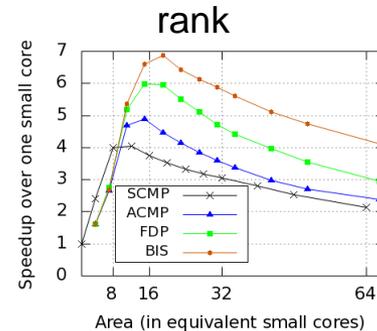
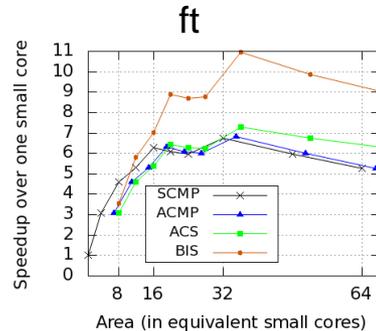
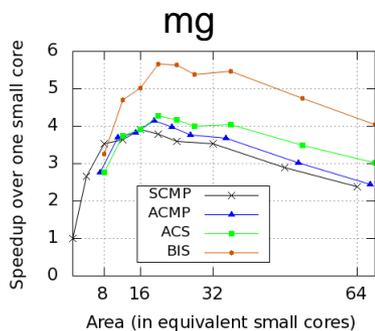
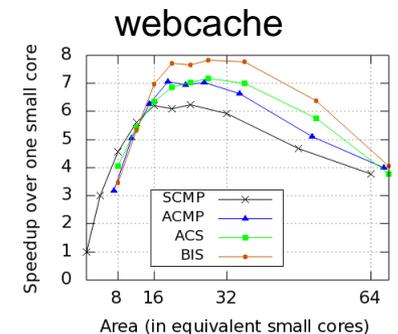
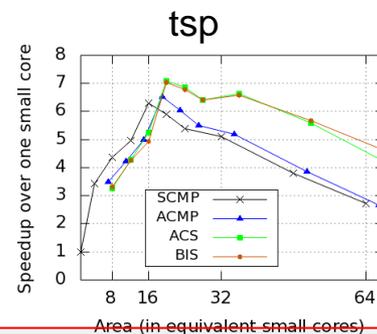
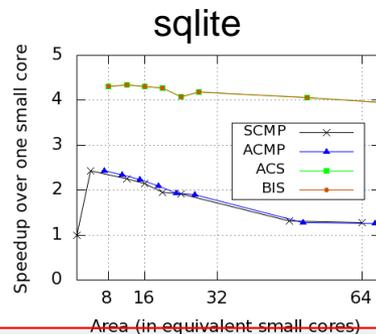
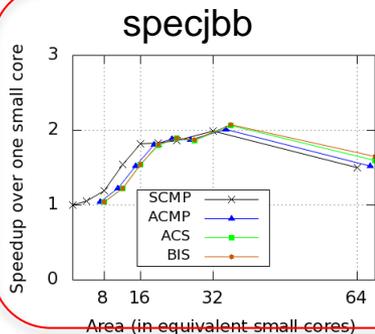
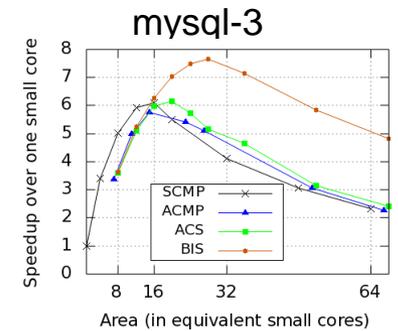
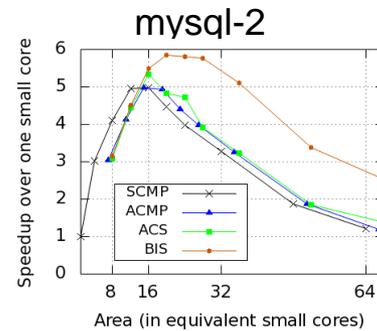
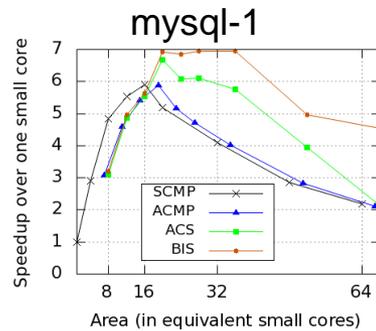
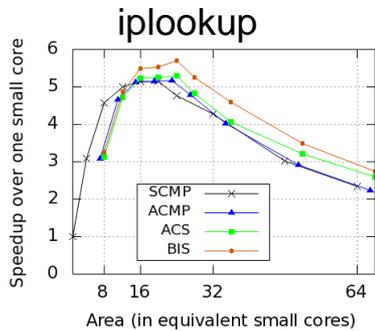
Workloads

Workload	Description	Source	Input set	# of Bottl.	Bottleneck description
iplookup	IP packet routing	[39]	2.5K queries	# of threads	Critical sections (CS) on routing tables
mysql-1	MySQL server [1]	SysBench [3]	OLTP-complex	29	CS on meta data, tables
mysql-2	MySQL server [1]	SysBench [3]	OLTP-simple	20	CS on meta data, tables
mysql-3	MySQL server [1]	SysBench [3]	OLTP-nontrx	18	CS on meta data, tables
specjbb	JAVA business benchmark	[33]	5 seconds	39	CS on counters, warehouse data
sqlite	sqlite3 [2] DB engine	SysBench [3]	OLTP-simple	5	CS on database tables
tsp	Traveling salesman	[20]	11 cities	2	CS on termination condition, solution
webcache	Cooperative web cache	[38]	100K queries	33	CS on replacement policy
mg	Multigrid solver	NASP [6]	S input	13	Barriers for omp single/master
ft	FFT computation	NASP [6]	T input	17	Barriers for omp single/master
rank	Rank strings based on similarity to another string	[37]	800K strings	3	Pipeline stages
pagemine	Computes a histogram of string similarity	Rsearch [27]	1M pages	5	Pipeline stages

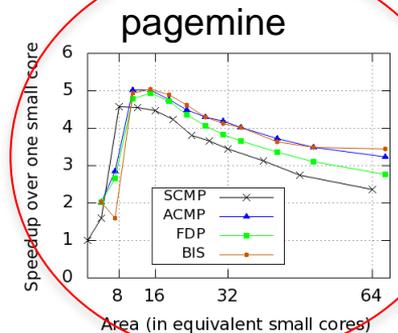
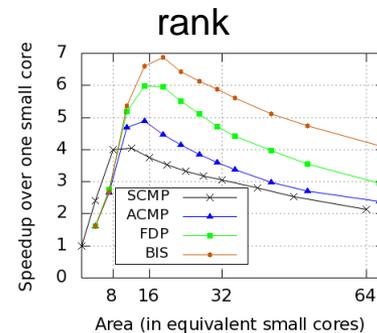
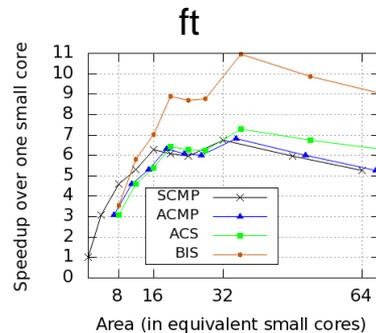
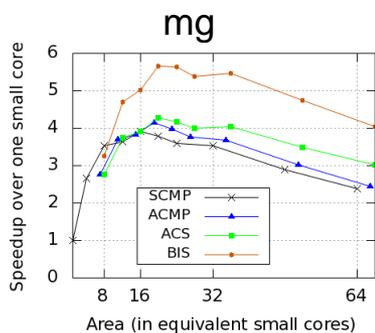
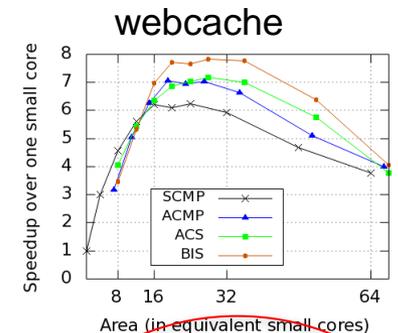
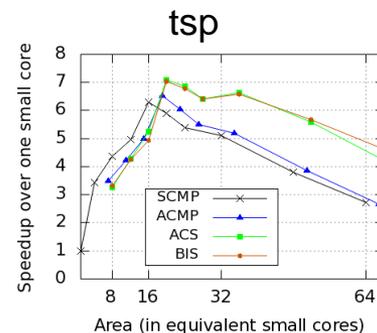
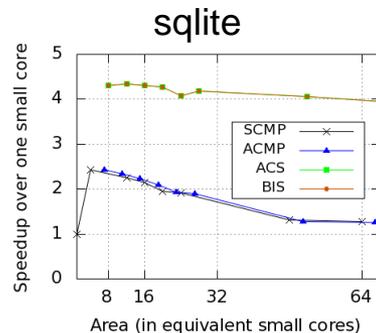
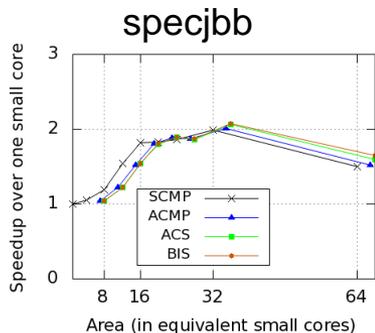
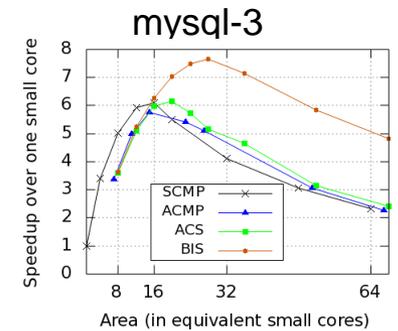
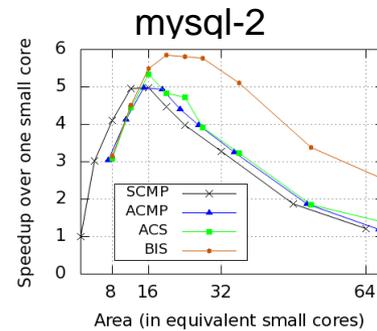
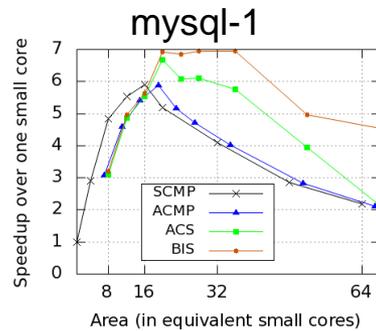
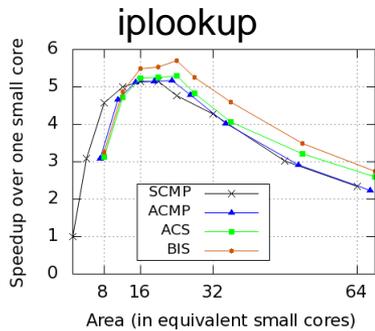
Scalability at Same Area Budgets



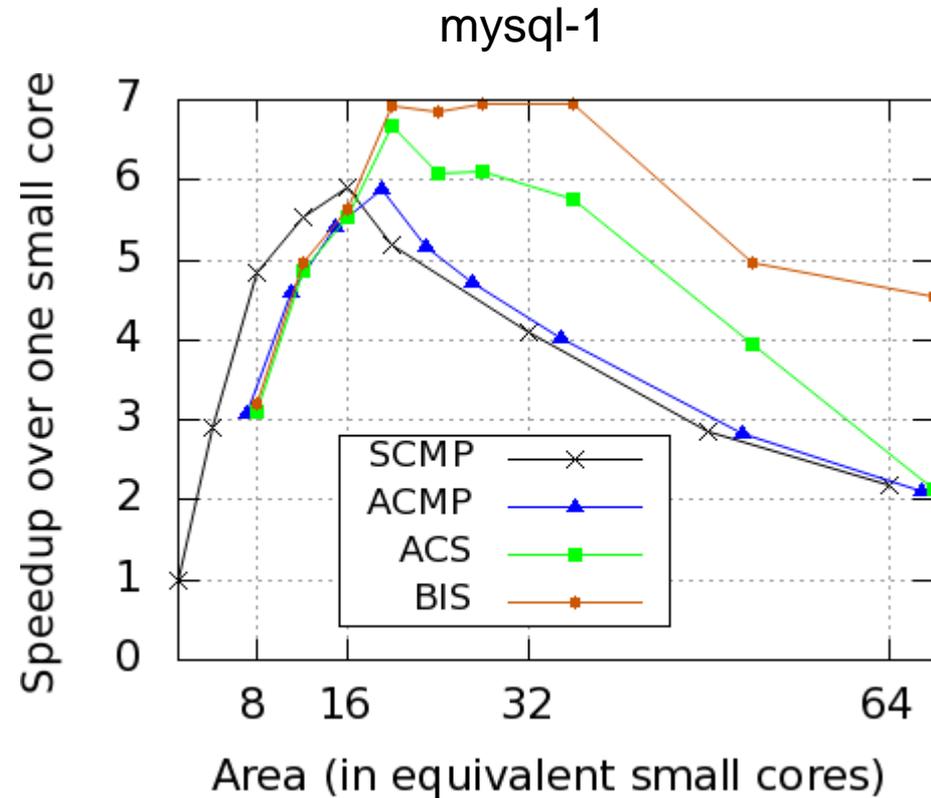
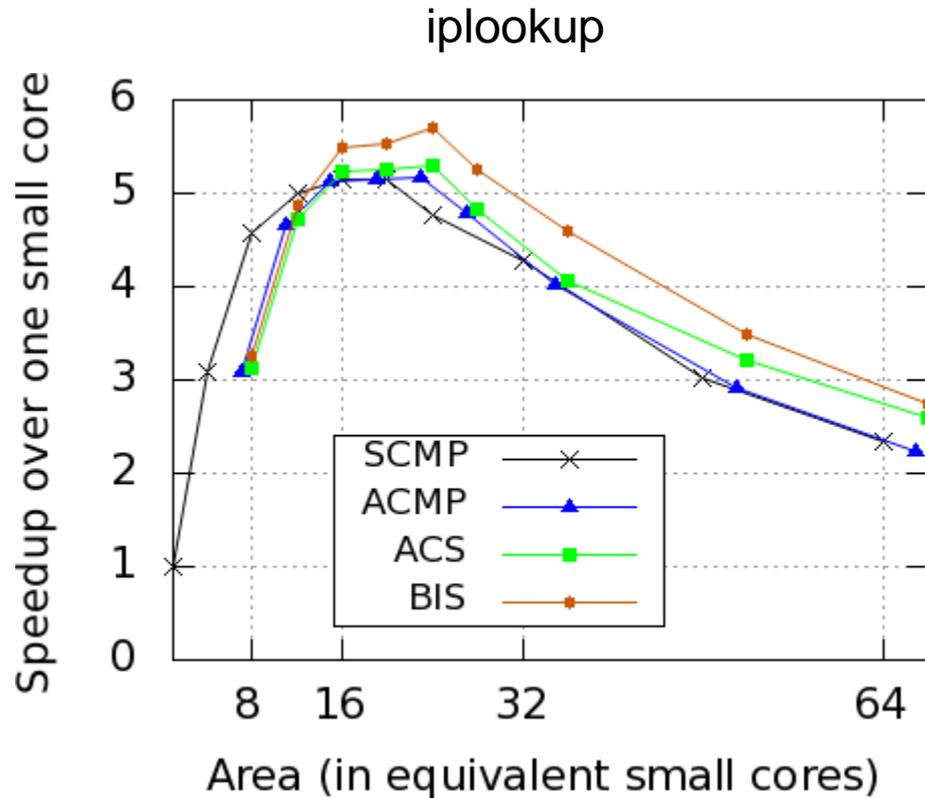
Scalability at Same Area Budgets



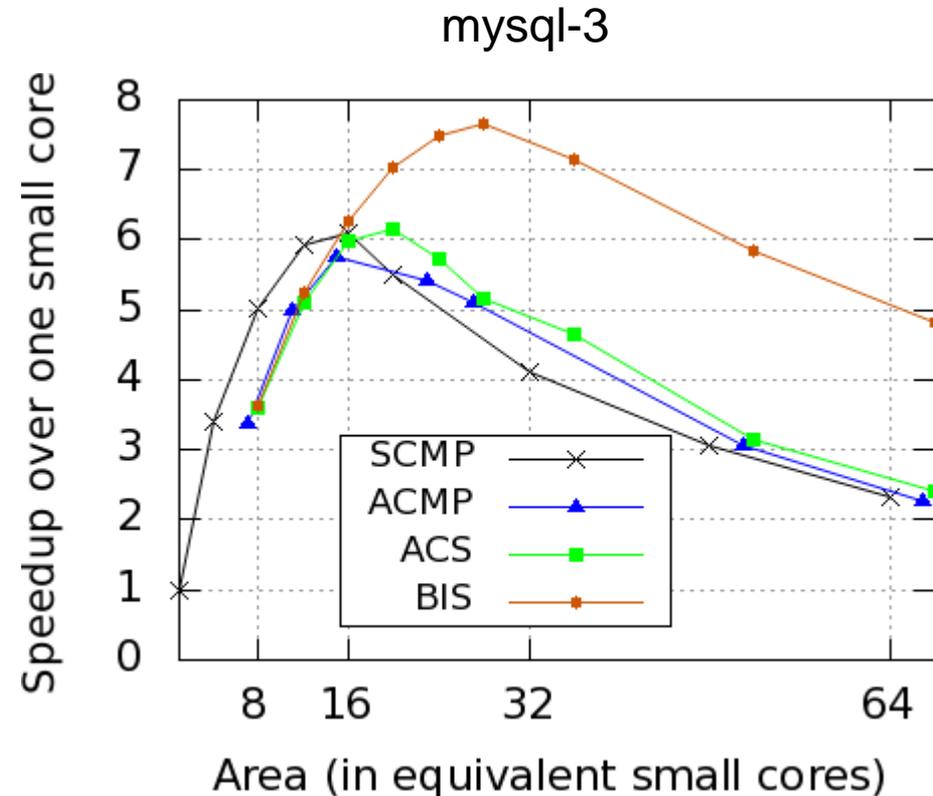
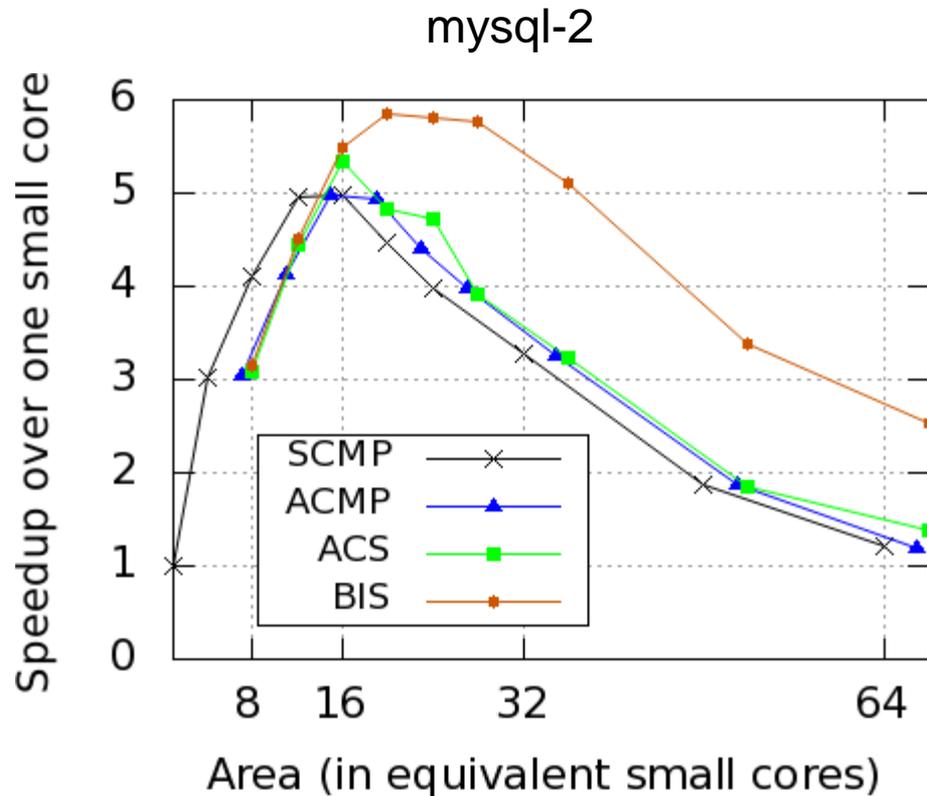
Scalability at Same Area Budgets



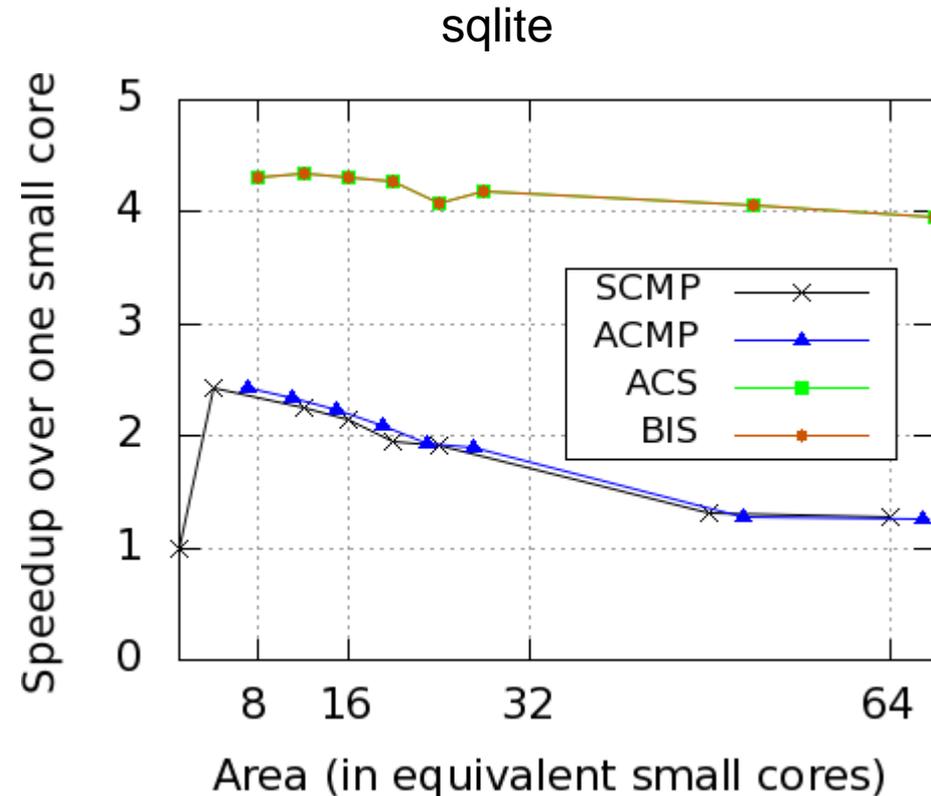
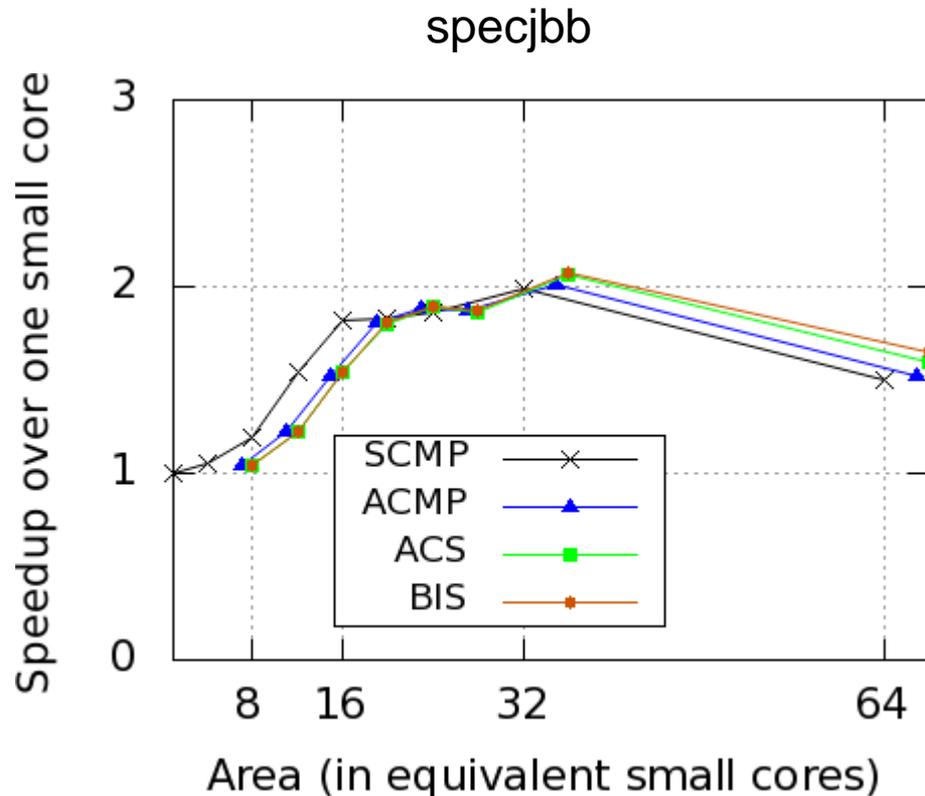
Scalability with #threads = #cores (I)



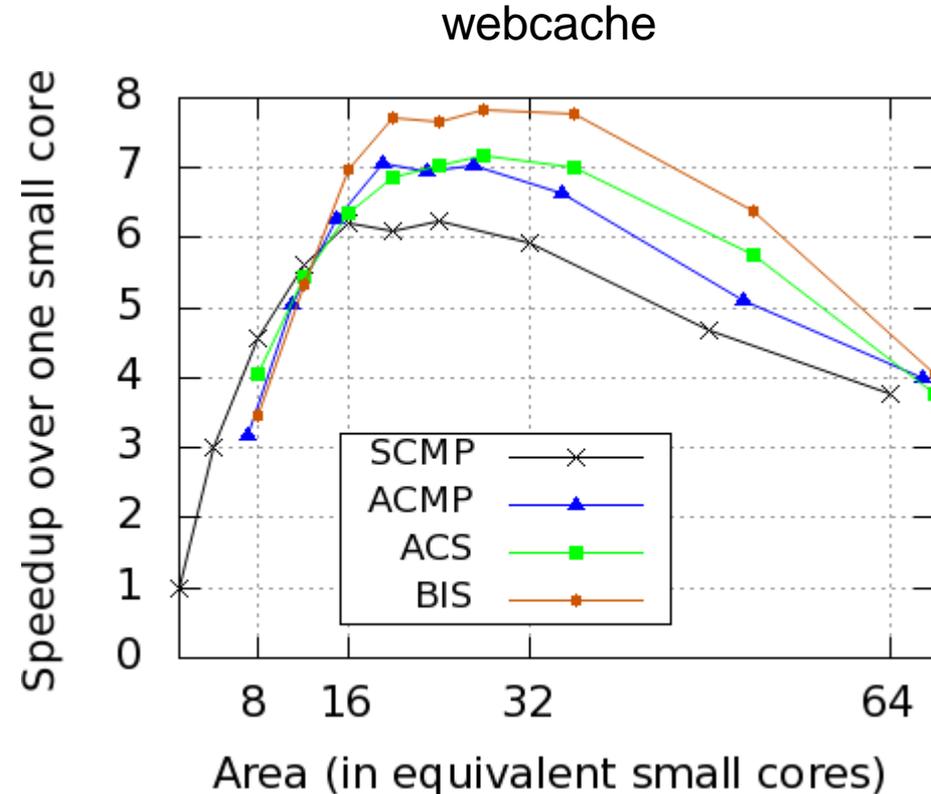
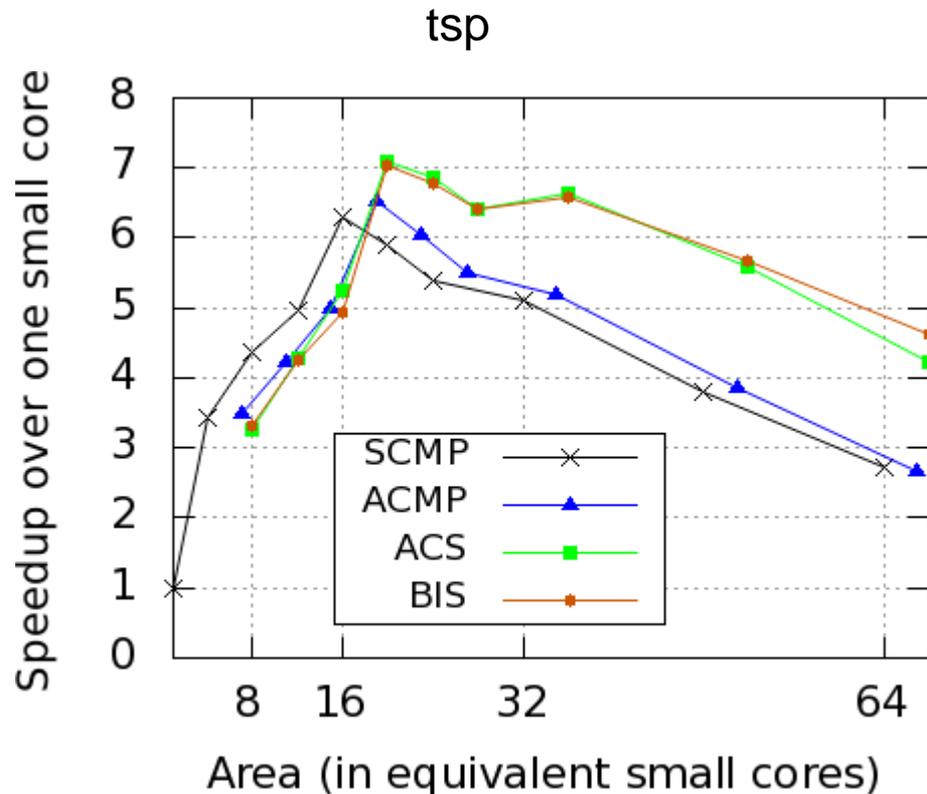
Scalability with #threads = #cores (II)



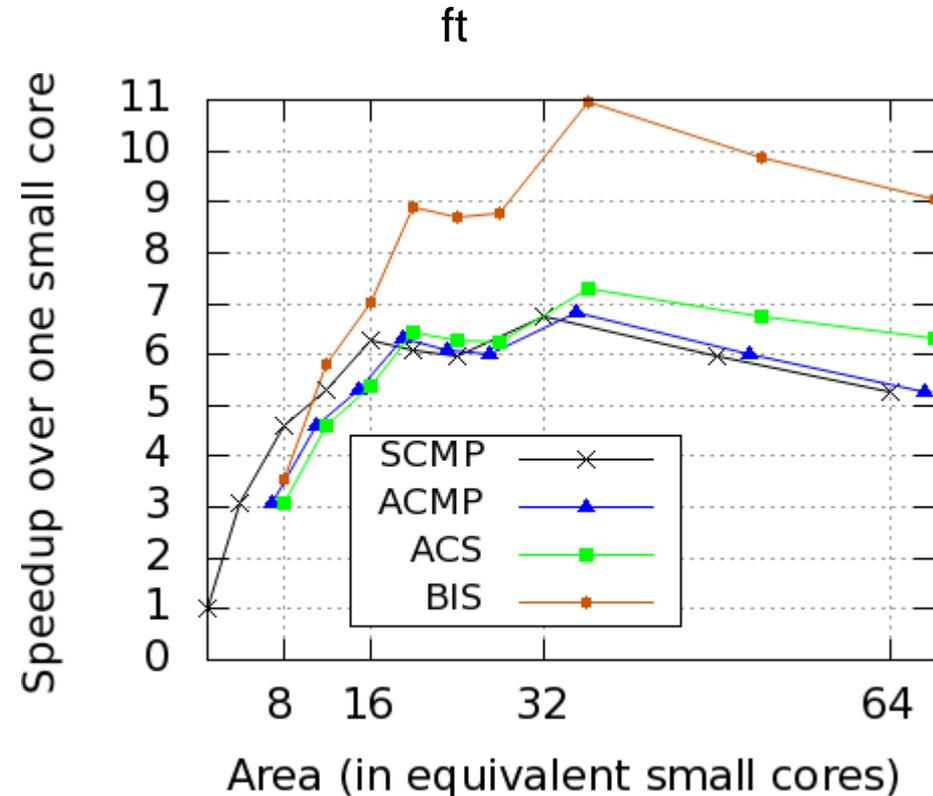
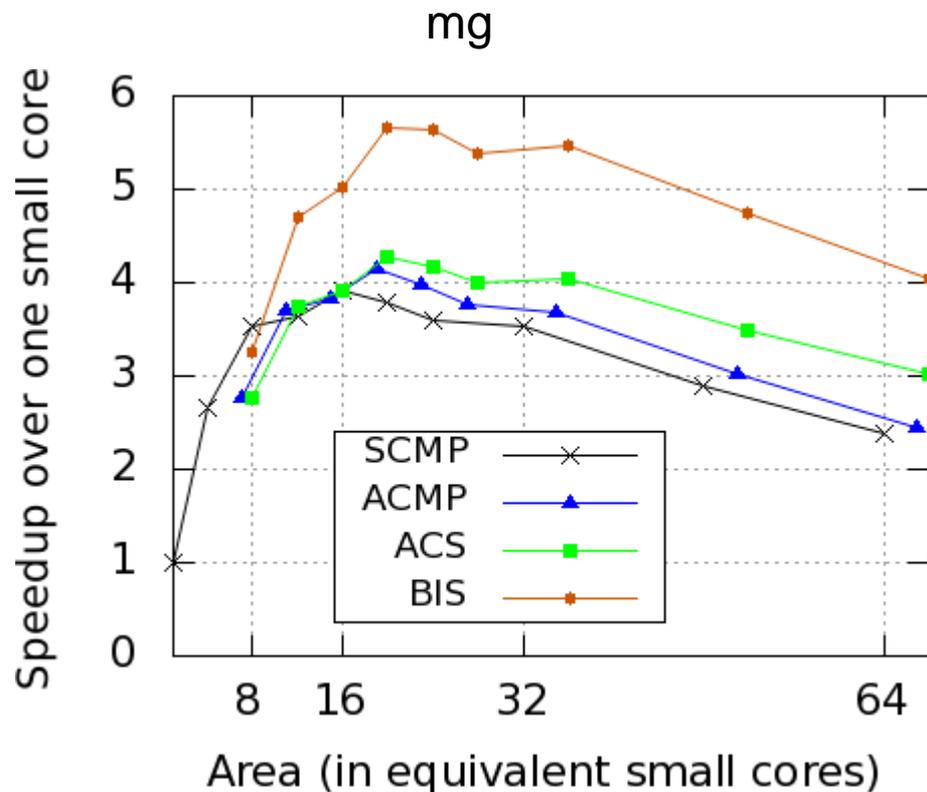
Scalability with #threads = #cores (III)



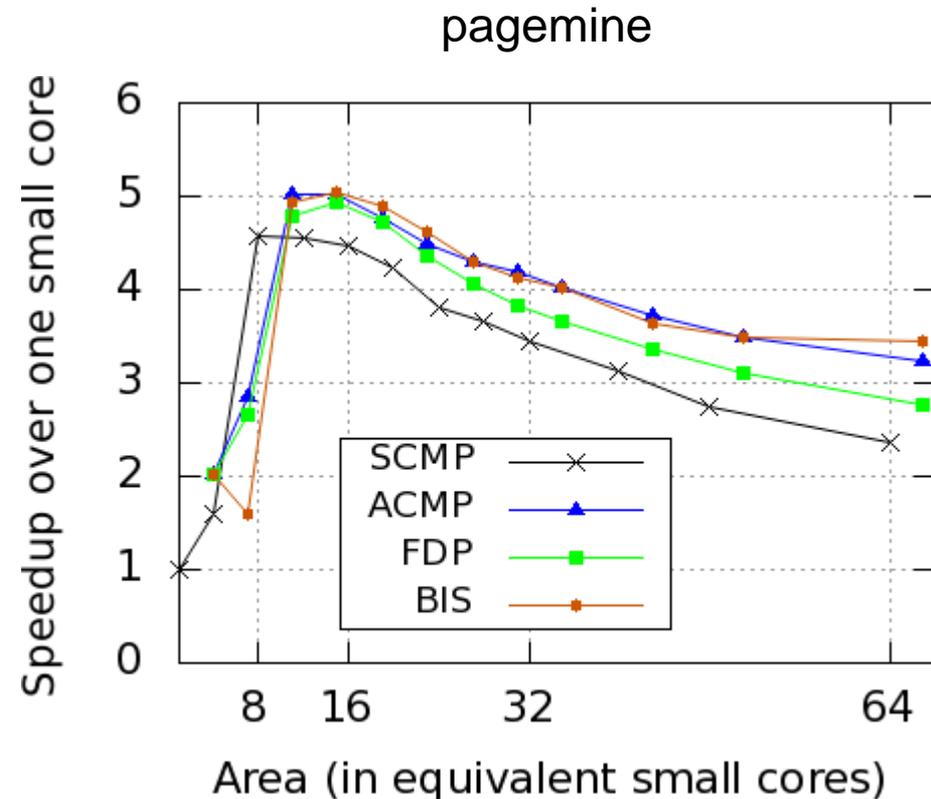
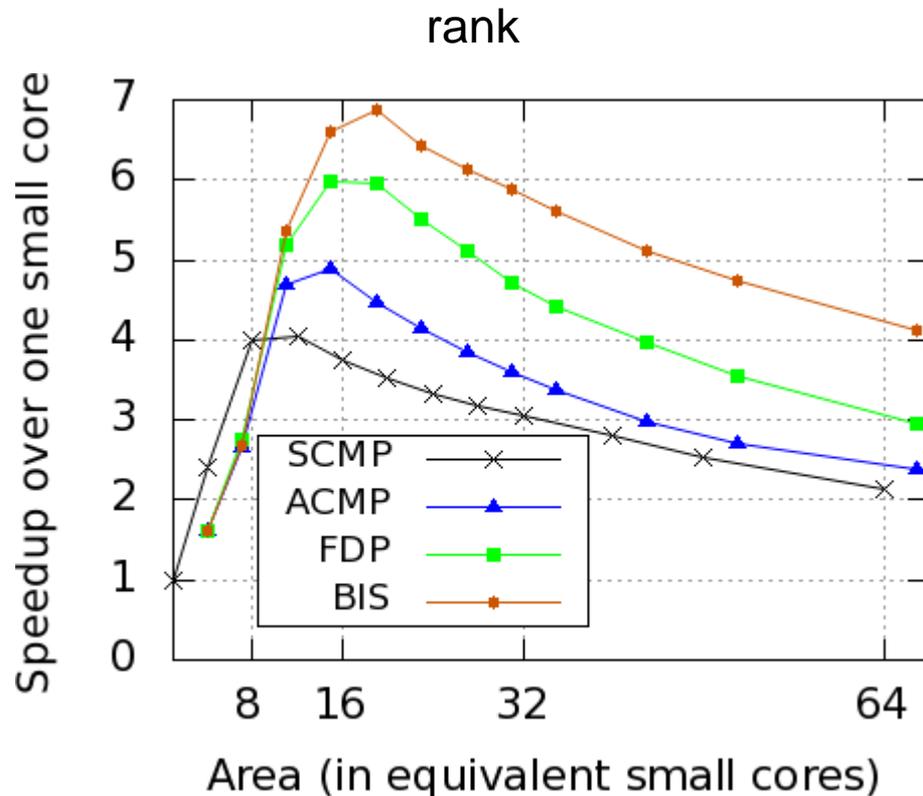
Scalability with #threads = #cores (IV)



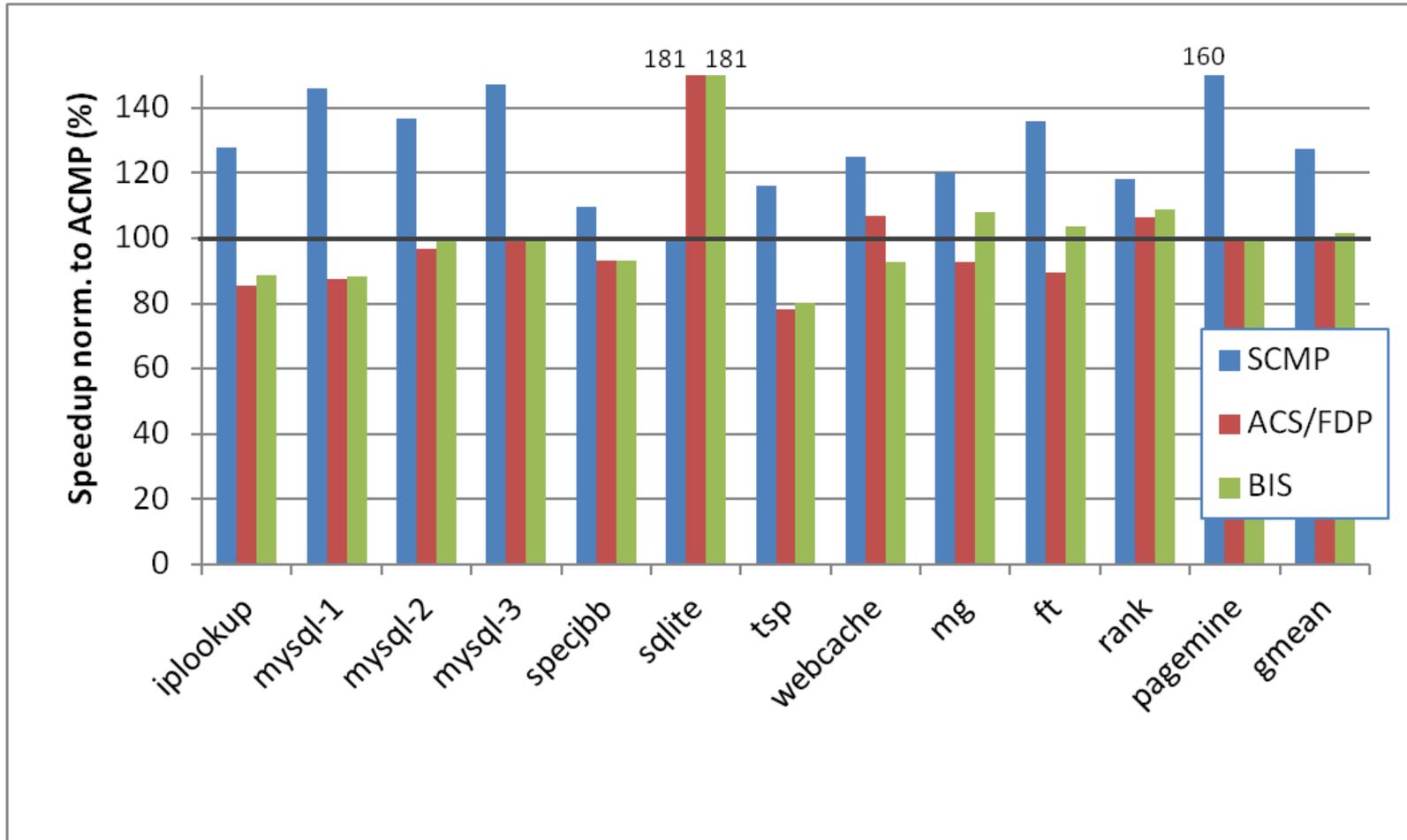
Scalability with #threads = #cores (V)



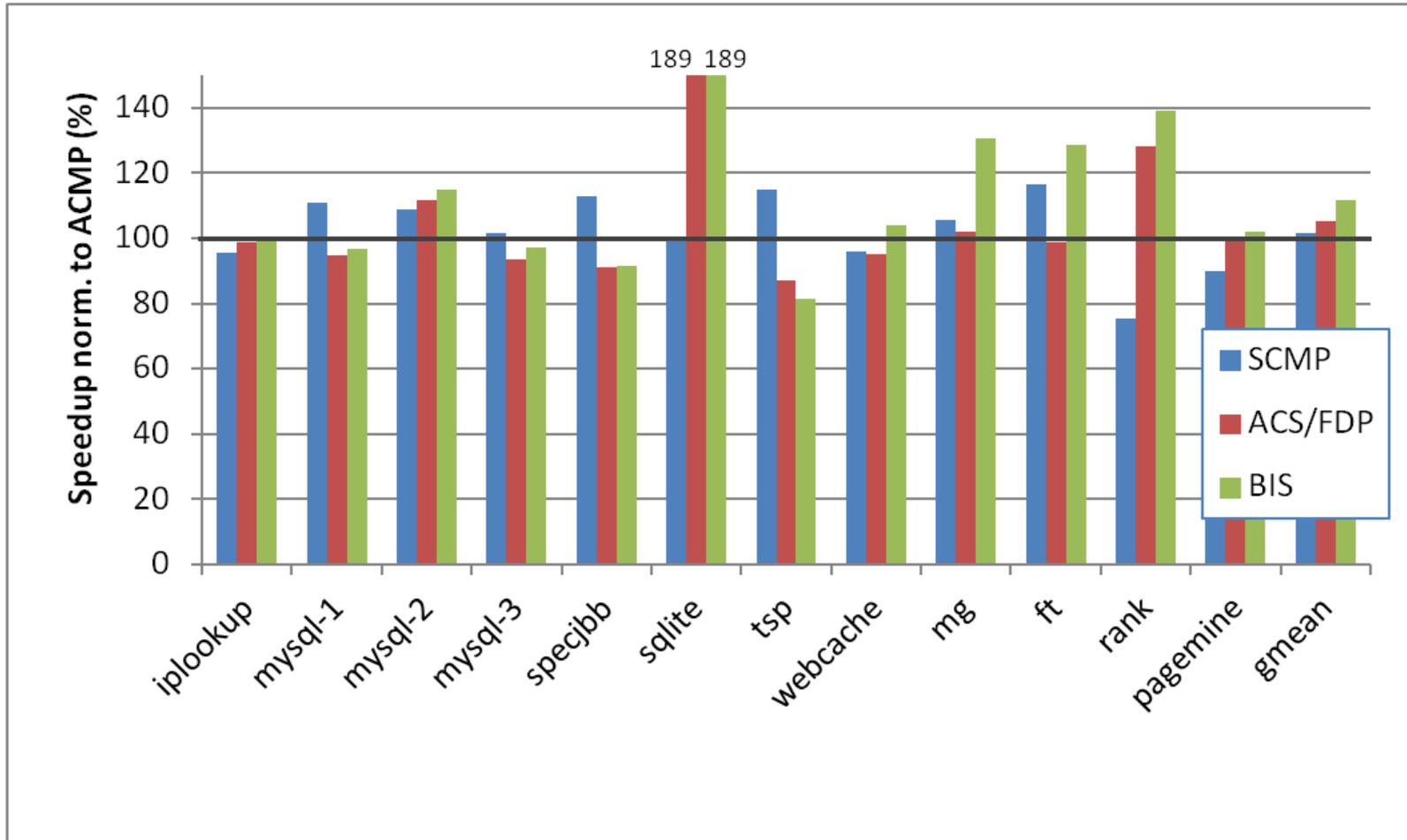
Scalability with #threads = #cores (VI)



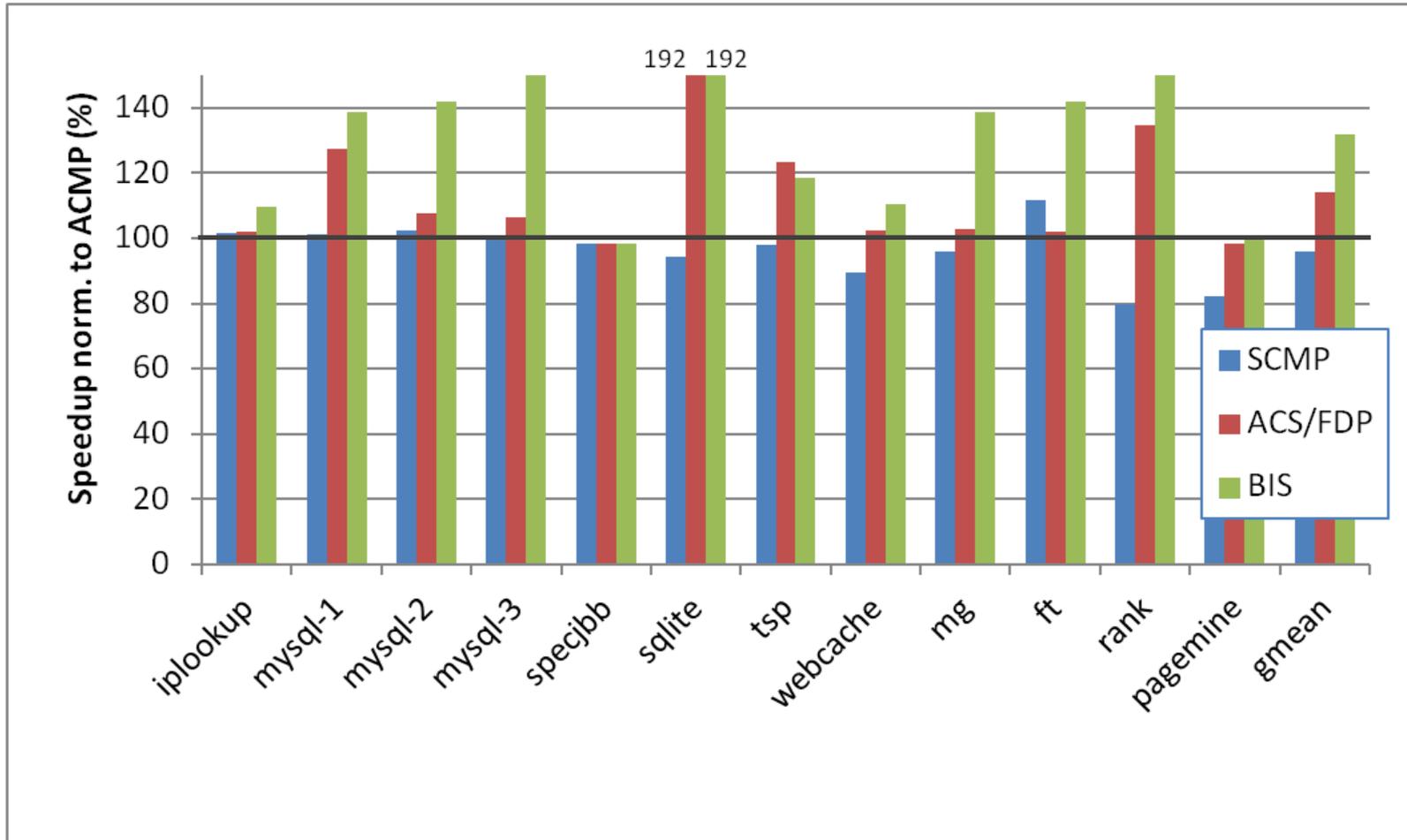
Optimal number of threads – Area=8



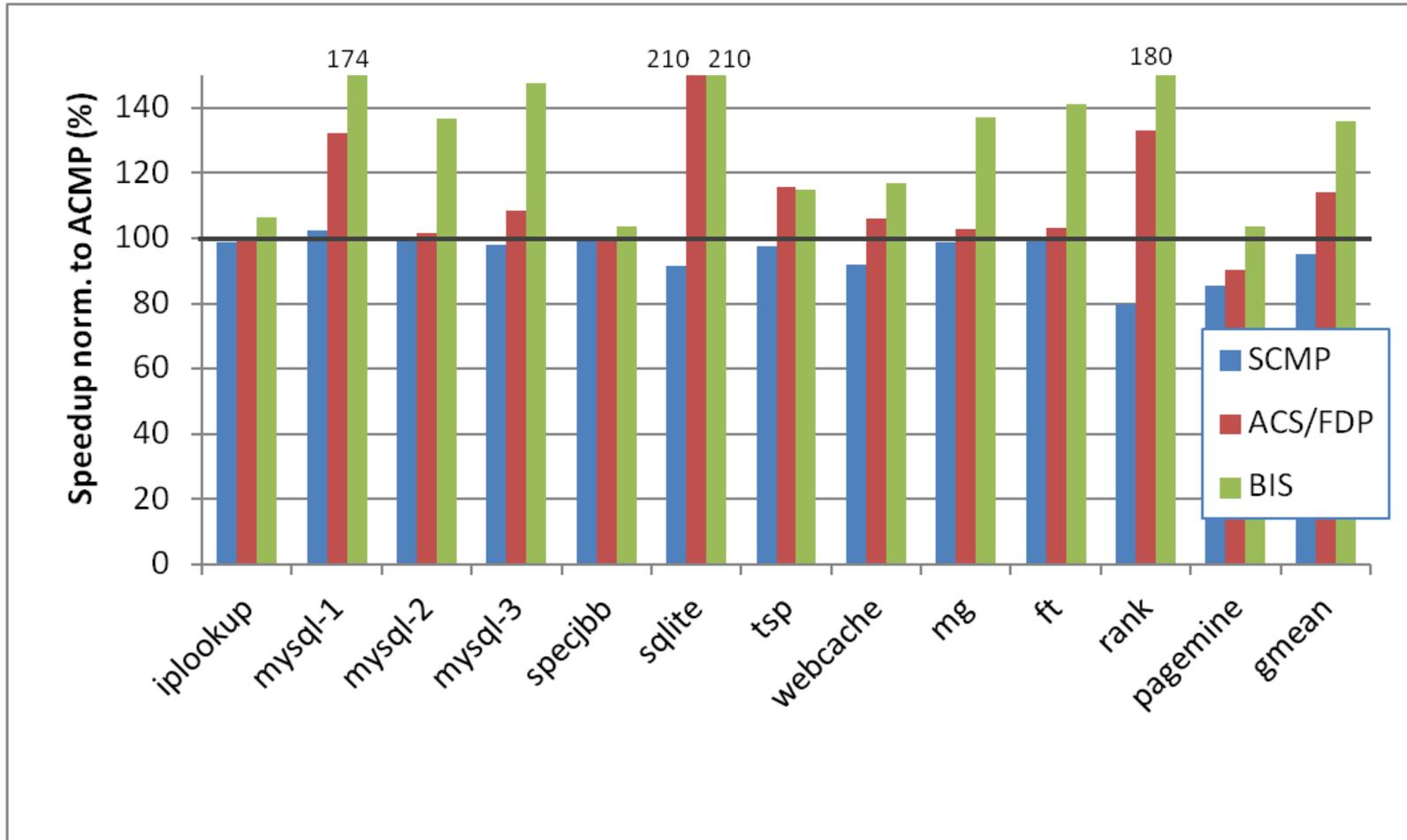
Optimal number of threads – Area=16



Optimal number of threads – Area=32



Optimal number of threads – Area=64



BIS and Data Marshaling, 28 T, Area=32

