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

32 threads

5

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

**Hardware**

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

Binary containing BIS instructions
... while cannot acquire lock
    Wait loop for watch_addr
acquire lock
...
release lock
...
Critical Sections: Code Modifications

... BottleneckCall \textit{bid}, targetPC
... Wait loop for watch\_addr

\textbf{targetPC:} while cannot acquire lock
... BottleneckWait \textit{bid}, watch\_addr
acquire lock

... release lock
BottleneckReturn \textit{bid}
Critical Sections: Code Modifications

... 
\text{BottleneckCall} \; bid, \; \text{targetPC} 
...
\text{wait loop for watch\_addr} 
\text{targetPC:} \; \text{while cannot acquire lock} 
\text{BottleneckWait} \; bid, \; \text{watch\_addr} 
\text{acquire lock} 
... 
\text{release lock} 
\text{BottleneckReturn} \; bid 

\underline{Used to enable acceleration}\n
\underline{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

... 

**BottleneckWait**  prev\_bid

dequeue work

do the work ... 

**BottleneckWait**  next\_bid

enqueue next work

**BottleneckReturn**  \( bid \)
1. Annotate *bottleneck* code
2. Implements *waiting* for bottlenecks

**Compiler/Library/Programmer**

Binary containing BIS instructions

---

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

**Hardware**
# 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

<table>
<thead>
<tr>
<th>Small core</th>
<th>Small core</th>
<th>Large core</th>
</tr>
</thead>
<tbody>
<tr>
<td>Small core</td>
<td>Small core</td>
<td></td>
</tr>
<tr>
<td>Small core</td>
<td>Small core</td>
<td>Small core</td>
</tr>
<tr>
<td>Small core</td>
<td>Small core</td>
<td>Small core</td>
</tr>
<tr>
<td>Small core</td>
<td>Small core</td>
<td>Small core</td>
</tr>
<tr>
<td>Small core</td>
<td>Small core</td>
<td>Small core</td>
</tr>
<tr>
<td>Small core</td>
<td>Small core</td>
<td>Small core</td>
</tr>
<tr>
<td>Small core</td>
<td>Small core</td>
<td>Small core</td>
</tr>
</tbody>
</table>
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

Small Core 1

Large Core 0

Small Core 2

Bottleneck Table (BT)
Determining Thread Waiting Cycles for Each Bottleneck

Small Core 1

BottleneckWait *4500

Small Core 2

Bottleneck Table (BT)

bid=*4500, waiters=1, twc = 0

Large Core 0
Determining Thread Waiting Cycles for Each Bottleneck

Small Core 1

BottleneckWait x4500

Small Core 2

BottleneckTable (BT)

Large Core 0

bid=x4500, waiters=1, twc = 1
Determining Thread Waiting Cycles for Each Bottleneck

Small Core 1

BottleneckWait x4500

Small Core 2

BottleneckWait x4500

Large Core 0

Bottleneck Table (BT)

\( bid=x4500, \) waiters=2, twc = 1
### Determining Thread Waiting Cycles for Each Bottleneck

<table>
<thead>
<tr>
<th>Bottleneck</th>
<th>Small Core 1</th>
<th>Large Core 0</th>
</tr>
</thead>
<tbody>
<tr>
<td>BottleneckWait x4500</td>
<td></td>
<td></td>
</tr>
<tr>
<td>BottleneckWait x4500</td>
<td></td>
<td></td>
</tr>
<tr>
<td>BottleneckWait Table (BT)</td>
<td>bid=x4500, waiters=2, twc = 3</td>
<td></td>
</tr>
</tbody>
</table>

**Diagram:**
- Small Core 1 and Large Core 0 are connected by arrows indicating the flow of data or tasks.
- The Bottleneck Wait table (BT) is shown with the conditions: bid=x4500, waiters=2, twc = 3.
Determining Thread Waiting Cycles for Each Bottleneck

Small Core 1

BottleneckWait x4500

Bottleneck Table (BT)

bid=x4500, waiters=2, twc = 5

Small Core 2

BottleneckWait x4500

Large Core 0
1. Annotate **bottleneck** code
2. Implements **waiting** for bottlenecks

Binary containing **BIS instructions**

**Compiler/Library/Programmer**

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

**Hardware**
Bottleneck Acceleration

Small Core 1

Large Core 0

Small Core 2

Bottleneck Table (BT)

- bid=x4600, twc=100
- bid=x4700, twc=10000
Bottleneck Acceleration

Small Core 1

BottleneckCall $\times 4600$

Execute locally

Small Core 2

Bottleneck Call Table (BT)

$\leftarrow$ twc $<$ Threshold

$\begin{align*}
  bid &= x4600, \text{ twc} = 100 \\
  bid &= x4700, \text{ twc} = 10000
\end{align*}$
Bottleneck Acceleration

Small Core 1

**BottleneckCall x4700**

Execute remotely

Small Core 2

Large Core 0

Bottleneck Table (BT)

- \( \text{bid}=x4600, \text{twc}=100 \)
- \( \text{bid}=x4700, \text{twc}=10000 \)

\( \leftarrow \text{twc > Threshold} \)
Bottleneck Acceleration

Small Core 1

BottleneckCall x4700
Execute remotely

Small Core 2

Large Core 0

Scheduling Buffer (SB)

Bid=x4700, pc, sp, core1

Bottleneck Table (BT)

Bid=x4600, twc=100
Bid=x4700, twc=10000
Bottleneck Acceleration

Small Core 1

Acceleration Index Table (AIT)

bid=x4700, large core 0

Small Core 2

AIT

bid=x4700, large core 0

Bottleneck Table (BT)

bid=x4600, twc=100

bid=x4700, twc=10000

Large Core 0

Scheduling Buffer (SB)

← twc < Threshold
← twc > Threshold
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

Speedup norm. to ACMP (%)

ACS/FDP  BIS

ipltlookup  mysql-1  mysql-2  mysql-3  specbb  sqlite  tsp  webcache  mg  ft  rank  pagemine  gmean

ACS  FDP
BIS Performance Improvement

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

Limiting bottlenecks change over time
BIS Performance Improvement

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

Barriers, which ACS cannot accelerate
BIS Performance Improvement

- 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

Execution time (%)

Actually critical
Why Does BIS Work?

- **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)
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

<table>
<thead>
<tr>
<th>Workload</th>
<th>Description</th>
<th>Source</th>
<th>Input set</th>
<th># of Bott.</th>
<th>Bottleneck description</th>
</tr>
</thead>
<tbody>
<tr>
<td>iplookup</td>
<td>IP packet routing</td>
<td>[39]</td>
<td>2.5K queries</td>
<td>29</td>
<td>Critical sections (CS) on routing tables</td>
</tr>
<tr>
<td>specjbb</td>
<td>JAVA business benchmark</td>
<td>[33]</td>
<td>5 seconds</td>
<td>39</td>
<td>CS on counters, warehouse data</td>
</tr>
<tr>
<td>tsp</td>
<td>Traveling salesman</td>
<td>[20]</td>
<td>11 cities</td>
<td>2</td>
<td>CS on termination condition, solution</td>
</tr>
<tr>
<td>webcache</td>
<td>Cooperative web cache</td>
<td>[38]</td>
<td>100K queries</td>
<td>33</td>
<td>CS on replacement policy</td>
</tr>
<tr>
<td>mg</td>
<td>Multigrid solver</td>
<td>NASP [6]</td>
<td>S input</td>
<td>13</td>
<td>Barriers for omp single/master</td>
</tr>
<tr>
<td>ft</td>
<td>FFT computation</td>
<td>NASP [6]</td>
<td>T input</td>
<td>17</td>
<td>Barriers for omp single/master</td>
</tr>
<tr>
<td>rank</td>
<td>Rank strings based on similarity to another string</td>
<td>[37]</td>
<td>800K strings</td>
<td>3</td>
<td>Pipeline stages</td>
</tr>
<tr>
<td>pagemine</td>
<td>Computes a histogram of string similarity</td>
<td>Rsearch [27]</td>
<td>1M pages</td>
<td>5</td>
<td>Pipeline stages</td>
</tr>
</tbody>
</table>
Scalability at Same Area Budgets

- iplookup
- mysql-1
- mysql-2
- mysql-3
- specjbb
- sqlite
- tsp
- webcache
- mg
- ft
- rank
- pagemine
Scalability at Same Area Budgets

- iplookup
- mysql-1
- mysql-2
- mysql-3

- specjbb
- sqlite
- tsp
- webcache

- mg
- ft
- rank
- pagemine
Scalability at Same Area Budgets

- **iplookup**
- **mysql-1**
- **mysql-2**
- **mysql-3**
- **specjbb**
- **sqlite**
- **tsp**
- **webcache**
- **mg**
- **ft**
- **rank**
- **pagemine**
Scalability with \#threads $=$ \#cores (I)
Scalability with \#threads = \#cores (II)
Scalability with \#threads = \#cores (III)

**specjbb**

**sqlite**

<table>
<thead>
<tr>
<th>Area (in equivalent small cores)</th>
<th>Speedup over one small core</th>
</tr>
</thead>
<tbody>
<tr>
<td>8</td>
<td>0</td>
</tr>
<tr>
<td>16</td>
<td>1</td>
</tr>
<tr>
<td>32</td>
<td>2</td>
</tr>
<tr>
<td>64</td>
<td>3</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Speedup over one small core</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
</tr>
<tr>
<td>1</td>
</tr>
<tr>
<td>2</td>
</tr>
<tr>
<td>3</td>
</tr>
</tbody>
</table>

Legend:
- SCMP
- ACMP
- ACS
- BIS
Scalability with \#threads = \#cores (IV)
Scalability with \( \#\text{threads} = \#\text{cores} \) (V)

![Graphs for mg and ft showing scalability with varying cores and threads.](Image)
Scalability with \#threads = \#cores (VI)
Optimal number of threads – Area=8

![Graph showing speedup norm. to ACMP (%) for different benchmarks and thread numbers. The x-axis represents the benchmarks: iplookup, mysql-1, mysql-2, mysql-3, specbb, sqlite, tsp, webcache, mg, ft, rank, pagemine, gmean. The y-axis represents the speedup in percentage. The graph shows various bars for SCMP, ACS/FDP, and BIS. The optimal number of threads for Area=8 is indicated by the highest bar in each benchmark category.]
Optimal number of threads – Area=16
Optimal number of threads – Area=32

![Graph showing speedup normalized to ACMP (%) for different benchmarks and thread counts. The graph compares SCMP, ACS/FDP, and BIS with 192 threads marked as the optimal number.](image-url)
Optimal number of threads – Area=64
BIS and Data Marshaling, 28 T, Area=32