18-447
Computer Architecture
Guest Lecture: Multi-Core Systems

Prof. Onur Mutlu
Carnegie Mellon University
Unexpected Slowdowns in Multi-Core

![Graph showing slowdowns for MATLAB and GCC on Core 0 and Core 1. High priority tasks on Core 1 experience a slowdown of 3.04, while low priority tasks on Core 0 experience a slowdown of 1.07.](image-url)
Agenda

- Intro to Multi-Core Systems

- Multi-Core Design Issues
  - Shared Main Memory Systems
  - Shared Caches
  - Core Organization
  - Interconnects

- Announcements
  - CALCM reading group
  - ECE 18-742: Parallel Computers (offered Spring 2010)
  - Interested in summer and future research in computer architecture and multi-core systems?
Announcements

- Weekly CALCM Reading Group
  - Will start the first week of May (May 5)
  - Readings and brainstorming on cutting-edge research in comp arch and related areas
  - + snacks

- Email me or join CALCM mailing list if you are interested in attending and receiving announcements
  - [https://sos.ece.cmu.edu/mailman/listinfo/calcm-list](https://sos.ece.cmu.edu/mailman/listinfo/calcm-list)
Announcements (II)

- Interested in more Computer Architecture classes?
  - 18-740: Advanced Comp Arch (Fall 2009, Prof. Mowry)
  - 18-742: Parallel Comp Arch (Spring 2010, Prof. Mutlu)

- Interested in Summer of Future Research in Comp Arch?
  - Talk to me. Some sample projects:
    - **MS-Manic**: Memory systems for 1000-core processors
    - **On-chip security**: attacks, defenses, many-core resource management
    - **BLESS**: Bufferless on-chip networks
    - **Asymmetric Multi-Core Design**
    - Architectural support for safe/managed programming languages
    - Hardware/software/system support for tolerating hardware defects and bugs
The State of Computer Architecture

- Computer architecture is the science/art of designing high-performance processing systems under many different constraints (power, cost, size, battery life, reliability, etc)

- Processor performance improvements enabled innovation in software development for decades

- Single-thread performance has become very difficult to improve
  - Complexity wall
  - Memory wall
  - Power wall
  - Reliability wall (soon)

- Chip-multiprocessor architectures
  - Reduce mainly the “complexity wall” by tiling cores
  - Create new problems
    - shared resources, parallel programming, off-chip bandwidth, serial bottleneck

Flynn & Hung, IEEE Micro, 2005

Moore, FCRC, 2007
Virtuous Cycle, 1950-2005 (per Jim Larus)

- Increased processor performance
- Larger, more feature-full software
- Larger development teams
- Higher-level languages & abstractions
- Slower programs

World-Wide Software Market (per IDC):
$212b (2005) \rightarrow $310b (2010)

*Slide credit: Mark Hill, HPCA 2007
Virtuous Cycle, 2005+

Thread Level Parallelism & Multicore Chips
World-Wide Software Market: $212b (2005) → ???

*Slide credit: Mark Hill, HPCA 2007
An Example Multi-Core System

Multi-Core Chip

*Die photo credit: AMD Barcelona
A Future Multi-Core Chip
Designing Multi-Core Chips is Difficult

- Designers must confront single-core design options
  - Instruction fetch, decode, wakeup, select, out-of-order execution
  - Execution unit configuration, operand bypass, SIMD extensions
  - Load/store queues, data cache, L2 caches
  - Checkpoint, runahead, commit
  - Speculative execution: Prefetching, branch prediction

- As well as additional design degrees of freedom
  - How many cores? How big each? Heterogeneous/homogeneous?
  - Shared caches: levels? How many banks? How to share?
  - Shared memory interface: How many controllers? How to share?
  - On-chip interconnect: bus, switched, ordered? How to share?
  - Prefetching: how to manage prefetchers across cores?
Problems in Multi Core Chips

- Simplify the design complexity problem
  - Somewhat...
  - Stamping multiple of the same cores side by side and connect them with some interconnection network easier

- However, create many other (new) problems
  - **Shared resources** among multiple cores: how to design and manage?
  - More cores, **NOT faster cores**: single-thread performance suffers, serial code performance suffers
  - **Memory bandwidth**: How to supply all the cores with enough data
  - **Parallel programming**: How to write programs that can benefit from multiple cores? How to ease parallel programming?
  - **How to design the cores**: what kind? homogeneous or heterogeneous?
  - How to design the **interconnect** between cores/caches/memory?
Let’s Take a Look at Some of These Problems
Unexpected Slowdowns in Multi-Core

**Memory Performance Hog**
- Low priority (Core 0)
- High priority (Core 1)

**Graph:**
- Matlab (Core 0) slowdown: 1.07
- Gcc (Core 1) slowdown: 3.04
Why the Disparity in Slowdowns?

Multi-Core Chip

Shared DRAM Memory System

unfairness
Access Address:
(Row 0, Column 0)
(Row 0, Column 1)
(Row 0, Column 85)
(Row 1, Column 0)

Row decoder

Columns

Rows

Row 1

Column mux

Data

Column address 05

CONFLICT!
DRAM Controllers

- A row-conflict memory access takes 2-3 times longer than a row-hit access

- Current controllers take advantage of the row buffer

- Commonly used scheduling policy (FR-FCFS) [Rixner 2000]*
  1. Row-hit first: Service row-hit memory accesses first
  2. Oldest-first: Then service older accesses first

- This scheduling policy aims to maximize DRAM throughput

---

The Problem

- Multiple threads share the DRAM controller
- DRAM controllers designed to maximize DRAM throughput

- **DRAM scheduling policies are thread-unfair**
  - Row-hit first: unfairly prioritizes *threads with high row buffer locality*
    - Threads that keep on accessing the same row
  - Oldest-first: unfairly prioritizes *memory-intensive threads*

- **DRAM controllers vulnerable to denial of service**
  - Can write programs that deny memory service to others
  - Memory performance hogs
An Example Memory Performance Hog

STREAM
- Sequential memory access
- Very high row buffer locality (96% hit rate)
- Memory intensive

RANDOM
- Random memory access
- Very low row buffer locality (3% hit rate)
- Similarly memory intensive

// initialize large arrays A, B
for (j=0; j<N; j++) {
    index = rand();
    A[index] = B[index];
    ...
}

// initialize large arrays A, B
for (j=0; j<N; j++) {
    index = j*linesize;
    A[index] = B[index];
    ...
}
What does the MPH do?

Row size: 8KB, cache block size: 64B
128 (8KB/64B) requests of T0 serviced before T1
Effect of the MPH

Results on Intel Pentium D running Windows XP
(Similar results for Intel Core Duo and AMD Turion, and on Fedora Linux)
Can Be a Bigger Problem with More Cores

DRAM memory is the only shared resource

- High priority: 7.74
- Low priority: 1.05
- Me: Low priority

- libquantum (Core 0): 1.05
- hmmer (Core 1): 1.85
- h264ref (Core 2): 4.72
- omnetpp (Core 3): 7.74
Problems Caused by MPHs

- **Vulnerability to denial of service** [Usenix Security 2007]

- **Inability to enforce thread priorities** [MICRO 2007, ISCA 2008]

- **System performance loss** [MICRO 2007, ISCA 2008]
Preventing Memory Performance Hogs

- Fundamentally hard to distinguish between malicious and unintentional MPHs
  - MATLAB’s memory access behavior is very similar to STREAM’s

- Unfair DRAM scheduling is the fundamental cause of MPHs
  - MPHs exploit the unfairness in the DRAM controller

- Solution: Prevent DRAM unfairness
  - Contain and limit MPHs by providing fair memory scheduling
Solution: Hardware-Software Cooperation

- Hardware provides a fair scheduler that is
  - Configurable by software
  - High-performance
  - Simple to implement (cost- and power-efficient)

- System software decides policy
  - Configures the fair scheduler to enforce thread priorities and quality of service policies

- But, what is fairness in shared DRAM systems?
A DRAM system is fair if it equalizes the slowdown of equal-priority threads relative to when each thread is run alone on the same system.

DRAM-related stall-time: The time a thread spends waiting for DRAM memory.

$\text{ST}_{\text{shared}}$: DRAM-related stall-time when the thread runs with other threads.

$\text{ST}_{\text{alone}}$: DRAM-related stall-time when the thread runs alone.

**Memory-slowdown** = $\frac{\text{ST}_{\text{shared}}}{\text{ST}_{\text{alone}}}$

- Relative increase in stall-time

*Stall-Time Fair Memory scheduler (STFM)* aims to equalize Memory-slowdown for interfering threads, without sacrificing performance.

- Considers inherent DRAM performance of each thread
- Aims to allow proportional progress of threads
STFM Scheduling Algorithm [MICRO’07]

- For each thread, the DRAM controller
  - Tracks $ST_{\text{shared}}$
  - Estimates $ST_{\text{alone}}$

- Each cycle, the DRAM controller
  - Computes $\text{Slowdown} = \frac{ST_{\text{shared}}}{ST_{\text{alone}}}$ for threads with legal requests
  - Computes $\text{unfairness} = \frac{\text{MAX Slowdown}}{\text{MIN Slowdown}}$

- If $\text{unfairness} < \alpha$
  - Use DRAM throughput oriented scheduling policy

- If $\text{unfairness} \geq \alpha$
  - Use fairness-oriented scheduling policy
    - (1) requests from thread with MAX Slowdown first
    - (2) row-hit first, (3) oldest-first
How Does STFM Prevent Unfairness?

<table>
<thead>
<tr>
<th>T0</th>
<th>T1</th>
</tr>
</thead>
<tbody>
<tr>
<td>Row 0</td>
<td>Row 5</td>
</tr>
<tr>
<td>Row 0</td>
<td>Row 0</td>
</tr>
<tr>
<td>Row 0</td>
<td>Row 111</td>
</tr>
<tr>
<td>Row 0</td>
<td>Row 06</td>
</tr>
</tbody>
</table>

T0 Slowdown: 1.00
T1 Slowdown: 1.06
Unfairness: 1.06
\( \alpha \): 1.05

Row Buffer
Data
Row 161
Containing the Memory Performance Hog

14% improvement in system performance

<table>
<thead>
<tr>
<th>Slowdown</th>
<th>matlab (Core 0)</th>
<th>gcc (Core 1)</th>
</tr>
</thead>
<tbody>
<tr>
<td>1.76</td>
<td></td>
<td>1.85</td>
</tr>
</tbody>
</table>

(matlab vs. gcc)
STFM Implementation

- **Tracking $ST_{\text{shared}}$**
  - Increase $ST_{\text{shared}}$ if the thread cannot commit instructions due to an outstanding DRAM access

- **Estimating $ST_{\text{alone}}$**
  - Difficult to estimate directly because thread not running alone
  - Observation: $ST_{\text{alone}} = ST_{\text{shared}} - ST_{\text{interference}}$
  - Estimate $ST_{\text{interference}}$: Extra stall-time due to interference
  - Update $ST_{\text{interference}}$ when a thread incurs delay due to other threads
    - When a row buffer hit turns into a row-buffer conflict
      (keep track of the row that would have been in the row buffer)
    - When a request is delayed due to bank or bus conflict
Support for System Software

- System-level thread weights (priorities)
  - OS can choose thread weights to satisfy QoS requirements
  - Larger-weight threads should be slowed down less
  - OS communicates thread weights to the memory controller
  - Controller scales each thread’s slowdown by its weight
  - Controller uses weighted slowdown used for scheduling
    - Favors threads with larger weights

- $\alpha$: Maximum tolerable unfairness set by system software
  - Don’t need fairness? Set $\alpha$ large.
  - Need strict fairness? Set $\alpha$ close to 1.
  - Other values of $\alpha$: trade off fairness and throughput
Enforcing Thread Priorities

![Graph showing slowdown for low and high priority threads](image)

- **Low priority**
  - Matlab (Core 0): 2.78
  - Gcc (Core 1): 1.06

- **High priority**
  - Matlab (Core 0)
  - Gcc (Core 1)
Some Issues in Multi-Core Design

- Shared Main Memory System

- Shared vs. Private Caches

- Interconnect Design

- Amdahl’s Law: Asymmetric Multi-Core Chips
Multi-core Issues in Caching

- How does the cache hierarchy change in a multi-core system?
- **Private** cache: Cache belongs to one core
- **Shared** cache: Cache is shared by multiple cores
Shared Caches Between Cores

- Advantages:
  - Dynamic partitioning of available cache space
    - No fragmentation due to static partitioning
  - Easier to maintain coherence
  - Shared data and locks do not ping pong between caches

- Disadvantages
  - Cores incur conflict misses due to other cores’ accesses
    - Misses due to inter-core interference
    - Some cores can destroy the hit rate of other cores
      - What kind of access patterns could cause this?
  - Guaranteeing a minimum level of service (or fairness) to each core is harder (how much space, how much bandwidth?)
  - High bandwidth harder to obtain (N cores → N ports?)
Handling Shared Data in Private Caches

- Shared data and locks ping-pong between processors if caches are private
  - Increases latency to fetch shared data/locks
  - Reduces cache efficiency (many invalid blocks)
  - Scalability problem: maintaining coherence across a large number of private caches is costly

How to do better?

- Idea: Store shared data and locks only in one special core’s cache. Divert all critical section execution to that core/cache.
  - Essentially, a specialized core for processing critical sections
Multi-Core Cache Efficiency: Bandwidth Filters

- Caches act as a filter that reduce memory bandwidth requirement
  - Cache hit: No need to access memory
  - This is in addition to the latency reduction benefit of caching
  - GPUs use caches to reduce memory BW requirements

- Efficient utilization of cache space becomes more important with multi-core
  - Memory bandwidth is more valuable
    - Pin count not increasing as fast as # of transistors
      - 10% vs. 2x every 2 years
  - More cores put more pressure on the memory bandwidth
Some Issues in Multi-Core Design

- Shared Main Memory System
- Shared vs. Private Caches
- Interconnect Design
- Amdahl’s Law: Asymmetric Multi-Core Chips
On-Chip Interconnects

- Or Networks-On-Chip (NoC)

- Each node on chip consists of
  - A core and caches associated with the core

- How should we connect the nodes?
  - A shared bus is not scalable
  - A crossbar is too expensive
  - A ring?
  - A 2D mesh?
  - A torus?
On-Chip Interconnects

- What we want
  - Fast communication
  - No congestion
    - Many paths or good routing
  - Small area overhead
  - Small energy consumption
2D Mesh

+ Easy to layout in a 2D chip
+ Many paths, yet relatively simple
-- Large diameter, maximum distance
-- Large energy and area overhead
  -- Compared to rings, buses
  -- Many buffers in router
How to Make a 2D Mesh More Efficient

- NoC consumes 20-40% of system power in prototype chips
- Problem: Buffers consume energy, occupy area, increase router/NoC complexity/latency
- Question: When are buffers most helpful? Congestion.
- Observation: On-chip networks lightly loaded

- Idea: **Eliminate Buffers**
- Misroute a packet upon congestion instead of buffering it
  - Called **Hot Potato** routing
  - Deflected/misrouted packets eventually reach destination
Bufferless On-Chip Networks

- **Benefits**
  - Network Energy Savings: ~40%
  - Performance Increase: ~2%
    - Reduced router latency
  - Network Area Savings: ~40%
  - Simpler network/router design
  - Adaptivity, deadlock freedom

- **Many remaining research issues**
  - How to provide fairness to cores?
  - How to provide quality of service guarantees?
  - Better routing and flow-control algorithms to handle congestion
  - Prototyping in FPGAs
  - How to apply it to other topologies?
Some Issues in Multi-Core Design

- Shared Main Memory System
- Shared vs. Private Caches
- Interconnect Design
- Amdahl’s Law: Asymmetric Multi-Core Chips
Remember Amdahl’s Law?

- Begins with Simple Software Assumption (Limit Arg.)
  - Fraction $F$ of execution time perfectly parallelizable
  - No Overhead for
    - Scheduling
    - Communication
    - Synchronization, etc.
  - Fraction $1 - F$ Completely Serial

- Time on 1 core = $\frac{(1 - F)}{1} + \frac{F}{1} = 1$
- Time on $N$ cores = $\frac{(1 - F)}{1} + \frac{F}{N}$
- Speedup limited by the serial fraction of the program

*Slide credit: Mark Hill, HPCA 2007*
### Accelerating Serial Program Portions

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

#### “Tile-Large” Approach

- **Tile-large**: Good at serial program portions
- **Niagara**: Good at exploiting thread-level parallelism
- **ACMP (Asymmetric Multi-Core)**
  - Good at both
  - Serial: on large core, Parallel: on many small cores

#### “Niagara” Approach

#### ACMP Approach
Asymmetric Multi-Core Approach
Performance vs. Parallel Fraction
Performance vs. Parallel Fraction (II)

At low parallelism, ACMP and P6-Tile outperform Niagara.
Performance vs. Parallel Fraction (III)

At high parallelism, Niagara outperforms ACMP
Performance vs. Parallel Fraction (IV)

At medium parallelism, ACMP wins
Asymmetric Multi-Core Chips

- Powerful execution engines are needed to execute
  - Single-threaded applications
  - Serial sections of multithreaded applications (remember Amdahl’s law)
  - Where single thread performance matters (e.g., transactions, game logic)
  - Accelerate multithreaded applications (e.g., critical sections)

- Corollary: Core design and enhancements still very important in multi-core chips

- Many research questions
  - How many types of cores? How many “powerful” cores?
  - Specialized accelerator cores? For what kernels/applications?
  - How to allocate cores to threads and applications?
  - What should be shipped to and executed on powerful cores?
Summary

- Multi-core chips bring about many new challenges

In **Computer Architecture**
- Design of uncore components
- Design of cores
- Allocation of chip real-estate to types of cores and uncore

In **System Software**
- Hardware resource allocation and management
- Virtualization and QoS support

In **Programming Languages and Compilers**
- Parallelization, thread extraction, easy parallel programming


Suleman et al., "Accelerating Critical Section Execution with Asymmetric Multi-Core Architectures," ASPLOS 2009.