New Review Assignments

- Due: Friday, September 21, 11:59pm.


Last Lecture: Multi-Core Alternatives

- Bigger, more powerful single core
- Bigger caches
- (Simultaneous) multithreading
- Integrate platform components on chip instead
- More scalable superscalar, out-of-order engines
- Traditional symmetric multiprocessors
- Dataflow?
- Vector processors (SIMD)?
- Integrating DRAM on chip?
- Reconfigurable logic? (general purpose?)
- Other alternatives?
An Early History of Multi-Core

Homogeneous Multi-Core Evolution

From Symmetry to Asymmetry
Multi-Core Evolution
(An Early History)
Piranha Chip Multiprocessor


- An early example of a symmetric multi-core processor
- Large-scale server based on CMP nodes
- Designed for commercial workloads

Read:
Commercial Workload Characteristics

- Memory system is the main bottleneck
  - Very high CPI
  - Execution time dominated by memory stall times
  - Instruction stalls as important as data stalls
  - Fast/large L2 caches are critical

- Very poor Instruction Level Parallelism (ILP) with existing techniques
  - Frequent hard-to-predict branches
  - Large L1 miss ratios
  - Small gains from wide-issue out-of-order techniques

- No need for floating point and multimedia units
Piranha Processing Node

Alpha core: 1-issue, in-order, 500MHz

Next few slides from Luiz Barroso’s ISCA 2000 presentation of Piranha: A Scalable Architecture Based on Single-Chip Multiprocessing
Piranha Processing Node

Alpha core:
1-issue, in-order,
500MHz

L1 caches:
I&D, 64KB, 2-way
Piranha Processing Node

Alpha core:
1-issue, in-order, 500MHz
L1 caches:
I&D, 64KB, 2-way
Intra-chip switch (ICS)
32GB/sec, 1-cycle delay
Piranha Processing Node

Alpha core:
1-issue, in-order, 500MHz
L1 caches:
I&D, 64KB, 2-way
Intra-chip switch (ICS) 32GB/sec, 1-cycle delay
L2 cache:
shared, 1MB, 8-way
Piranha Processing Node

Alpha core:
1-issue, in-order, 500MHz
L1 caches:
I&D, 64KB, 2-way
Intra-chip switch (ICS)
32GB/sec, 1-cycle delay
L2 cache:
shared, 1MB, 8-way
Memory Controller (MC)
RDRAM, 12.8GB/sec

8 banks @1.6GB/sec
**Piranha Processing Node**

**CPU**
- Alpha core: 1-issue, in-order, 500MHz
- L1 caches: I&D, 64KB, 2-way
  - Intra-chip switch (ICS) 32GB/sec, 1-cycle delay
- L2 cache: shared, 1MB, 8-way

**Memory Controller (MC)**
- RDRAM, 12.8GB/sec

**Protocol Engines (HE & RE)**
- µprog., 1K µinstr., even/odd interleaving
Piranha Processing Node

Alpha core:
- 1-issue, in-order,
- 500MHz
L1 caches:
- I&D, 64KB, 2-way
Intra-chip switch (ICS)
- 32GB/sec, 1-cycle delay
L2 cache:
- shared, 1MB, 8-way
Memory Controller (MC)
- RDRAM, 12.8GB/sec
Protocol Engines (HE & RE):
- μprog., 1K μinstr.,
  even/odd interleaving
System Interconnect:
- 4-port Xbar router
  topology independent
  32GB/sec total
  bandwidth
Piranha Processing Node

Alpha core:
  1-issue, in-order,
  500MHz

L1 caches:
  I&D, 64KB, 2-way
  Intra-chip switch (ICS)
    32GB/sec, 1-cycle
    delay

L2 cache:
  shared, 1MB, 8-way

Memory Controller (MC)
  RDRAM, 12.8GB/sec

Protocol Engines (HE & RE):
  \( \mu \)prog., 1K \( \mu \)instr.,
  even/odd interleaving

System Interconnect:
  4-port Xbar router
  topology independent

Single Chip

bandwidth
Piranha Processing Node
Inter-Node Coherence Protocol Engine

**Figure 4.** Block diagram of a protocol engine.
Figure 3. Example configuration for a Piranha system with six processing (8 CPUs each) and two I/O chips.
Figure 2. Block diagram of a single-chip Piranha I/O node.
Sun Niagara (UltraSPARC T1)

Niagara Core

- 4-way fine-grain multithreaded, 6-stage, dual-issue in-order
- Round robin thread selection (unless cache miss)
- Shared FP unit among cores
Niagara Design Point

- Also designed for commercial applications

<table>
<thead>
<tr>
<th>Benchmark</th>
<th>Application category</th>
<th>Instruction-level parallelism</th>
<th>Thread-level parallelism</th>
<th>Working set</th>
<th>Data sharing</th>
</tr>
</thead>
<tbody>
<tr>
<td>Web99</td>
<td>Web server</td>
<td>Low</td>
<td>High</td>
<td>Large</td>
<td>Low</td>
</tr>
<tr>
<td>JBB</td>
<td>Java application server</td>
<td>Low</td>
<td>High</td>
<td>Large</td>
<td>Medium</td>
</tr>
<tr>
<td>TPC-C</td>
<td>Transaction processing</td>
<td>Low</td>
<td>High</td>
<td>Large</td>
<td>High</td>
</tr>
<tr>
<td>SAP-2T</td>
<td>Enterprise resource planning</td>
<td>Medium</td>
<td>High</td>
<td>Medium</td>
<td>Medium</td>
</tr>
<tr>
<td>SAP-3T</td>
<td>Enterprise resource planning</td>
<td>Low</td>
<td>High</td>
<td>Large</td>
<td>High</td>
</tr>
<tr>
<td>TPC-H</td>
<td>Decision support system</td>
<td>High</td>
<td>High</td>
<td>Large</td>
<td>Medium</td>
</tr>
</tbody>
</table>

Diagram showing single issue, ILP, and TLP with memory latency and compute latency.
Sun Niagara II (UltraSPARC T2)

- 8 SPARC cores, 8 threads/core. 8 stages. 16 KB I$ per Core. 8 KB D$ per Core. FP, Graphics, Crypto, units per Core.

- 4 MB Shared L2, 8 banks, 16-way set associative.

- 4 dual-channel FBDIMM memory controllers.

- X8 PCI-Express @ 2.5 Gb/s.

- Two 10G Ethernet ports @ 3.125 Gb/s.
Chip Multithreading (CMT)


- **Idea:** Chip multiprocessor where each core is multithreaded
  - Niagara 1/2: fine grained multithreading
  - IBM POWER5: simultaneous multithreading

- **Motivation:** Tolerate memory latency better
  - A simple core stays idle on a cache miss
  - Multithreading enables tolerating cache miss latency when there is TLP
Advantages of adding multithreading to each core

+ Better memory latency tolerance when there are enough threads
+ Fine grained multithreading can simplify core design (no need for branch prediction, dependency checking)
+ Potentially better utilization of core, cache, memory resources
  + Shared instructions and data among threads not replicated
  + When one thread is not using a resource, another can

Disadvantages

- Reduced single-thread performance (a thread does not have the core and L1 caches to itself)
- More pressure on the shared resources (cache, off-chip bandwidth) \(\rightarrow\) more resource contention
- Applications with limited TLP do not benefit
Sun ROCK


Goals:
- Maximize throughput when threads are available
- Boost single-thread performance when threads are not available and on cache misses

Ideas:
- Runahead on a cache miss → ahead thread executes miss-independent instructions, behind thread executes dependent instructions
- Branch prediction (gshare)
Sun ROCK

- 16 cores, 2 threads per core (fewer threads than Niagara 2)
- 4 cores share a 32KB instruction cache
- 2 cores share a 32KB data cache
- 2MB L2 cache (smaller than Niagara 2)
Runahead Execution (I)

- A simple pre-execution method for prefetching purposes

When the oldest instruction is a long-latency cache miss:
- Checkpoint architectural state and enter runahead mode

In runahead mode:
- Speculatively pre-execute instructions
- The purpose of pre-execution is to generate prefetches
- L2-miss dependent instructions are marked INV and dropped

Runahead mode ends when the original miss returns
- Checkpoint is restored and normal execution resumes
Runahead Execution (II)

**Small Window:**
- Load 1 Miss
  - Compute
  - Stall
- Load 2 Miss
  - Compute
  - Stall

**Runahead:**
- Load 1 Miss
  - Compute
  - Runahead
- Load 2 Miss
  - Compute
  - Runahead
- Load 1 Hit
  - Compute
  - Runahead
- Load 2 Hit
  - Compute
  - Runahead

**Saved Cycles**
Runahead Execution (III)

**Advantages**
- Very accurate prefetches for data/instructions (all cache levels)
  - Follows the program path
- Simple to implement, most of the hardware is already built in

**Disadvantages**
- Extra executed instructions

**Limitations**
- Limited by branch prediction accuracy
- Cannot prefetch dependent cache misses. Solution?
- Effectiveness limited by available Memory Level Parallelism

Performance of Runahead Execution

- No prefetcher, no runahead
- Only prefetcher (baseline)
- Only runahead
- Prefetcher + runahead

Graph shows the performance of runahead execution with different configurations across various applications.
Sun ROCK Cores

- Load miss in L1 cache starts parallelization using 2 HW threads
- Ahead thread
  - Checkpoints state and executes speculatively
  - Instructions independent of load miss are speculatively executed
  - Load miss(es) and dependent instructions are deferred to behind thread
- Behind thread
  - Executes deferred instructions and re-defers them if necessary

- Memory-Level Parallelism (MLP)
  - Run ahead on load miss and generate additional load misses

- Instruction-Level Parallelism (ILP)
  - Ahead and behind threads execute independent instructions from different points in program in parallel
ROCK Pipeline
More Powerful Cores in Sun ROCK

- Advantages
  + Higher single-thread performance (MLP + ILP)
  + Better cache miss tolerance → Can reduce on-chip cache sizes

- Disadvantages
  - Bigger cores → Fewer cores → Lower parallel throughput (in terms of threads).
    How about each thread’s response time?
  - More complex than Niagara cores (but simpler than conventional out-of-order execution) → Longer design time?
More Powerful Cores in Sun ROCK

- Chaudhry talk, Aug 2008.

![Graph showing normalized IPC against L2 cache size, with labels for 'Scout', 'No Scout', 'Bays 12 MB', 'Bays 7 MB', and '40% Better Performance']
More Powerful Cores in Sun ROCK


**Figure 9: Commercial Performance.**

Another symmetric multi-core chip...

But, fewer and more powerful cores
IBM POWER4

- 2 cores, out-of-order execution
- 100-entry instruction window in each core
- 8-wide instruction fetch, issue, execute
- Large, local+global hybrid branch predictor
- 1.5MB, 8-way L2 cache
- Aggressive stream based prefetching
IBM POWER5


Figure 4. Power5 instruction data flow (BXU = branch execution unit and CRL = condition register logical execution unit).
IBM POWER6


- 2 cores, in order, high frequency (4.7 GHz)
- 8 wide fetch
- Simultaneous multithreading in each core
- Runahead execution in each core
  - Similar to Sun ROCK
IBM POWER7

- 8 out-of-order cores, 4-way SMT in each core
- TurboCore mode
  - Can turn off cores so that other cores can be run at higher frequency
Large vs. Small Cores

- Out-of-order
- Wide fetch e.g. 4-wide
- Deeper pipeline
- Aggressive branch predictor (e.g. hybrid)
- Multiple functional units
- Trace cache
- Memory dependence speculation

- In-order
- Narrow Fetch e.g. 2-wide
- Shallow pipeline
- Simple branch predictor (e.g. Gshare)
- Few functional units

Large Cores are power inefficient: e.g., 2x performance for 4x area (power)
Large vs. Small Cores


<table>
<thead>
<tr>
<th>Feature</th>
<th>Large core</th>
<th>Small core</th>
</tr>
</thead>
<tbody>
<tr>
<td>Microarchitecture</td>
<td>Out-of-order, 128-256 entry ROB</td>
<td>In-order</td>
</tr>
<tr>
<td>Width</td>
<td>3-4</td>
<td>1</td>
</tr>
<tr>
<td>Pipeline depth</td>
<td>20-30</td>
<td>5</td>
</tr>
<tr>
<td>Normalized performance</td>
<td>5-8x</td>
<td>1x</td>
</tr>
<tr>
<td>Normalized power</td>
<td>20-50x</td>
<td>1x</td>
</tr>
<tr>
<td>Normalized energy/inSTRUCTION</td>
<td>4-6x</td>
<td>1x</td>
</tr>
</tbody>
</table>
Tile-Large Approach

- Tile a few large cores
- IBM Power 5, AMD Barcelona, Intel Core2Quad, Intel Nehalem
  + High performance on single thread, serial code sections (2 units)
  - Low throughput on parallel program portions (8 units)
Tile-Small Approach

- Tile many small cores
- Sun Niagara, Intel Larrabee, Tilera TILE (tile ultra-small)
  + High throughput on the parallel part (16 units)
  - Low performance on the serial part, single thread (1 unit)
Can We Get the Best of Both worlds?

- **Tile Large**
  + High performance on single thread, serial code sections (2 units)
  - Low throughput on parallel program portions (8 units)

- **Tile Small**
  + High throughput on the parallel part (16 units)
  - Low performance on the serial part, single thread (1 unit), reduced single-thread performance compared to existing single thread processors

- **Idea:** Have both large and small on the same chip → Performance asymmetry
Asymmetric Chip Multiprocessor (ACMP)

- Provide one large core and many small cores
- Accelerate serial part using the large core (2 units)
- Execute parallel part on all cores for high throughput (14 units)
Accelerating Serial Bottlenecks

Single thread $\rightarrow$ Large core

ACMP Approach
Performance vs. Parallelism

Assumptions:

1. Small core takes an area budget of 1 and has performance of 1

2. Large core takes an area budget of 4 and has performance of 2
### ACMP Performance vs. Parallelism

Area-budget = 16 small cores

<table>
<thead>
<tr>
<th></th>
<th>Large Cores</th>
<th>Small Cores</th>
<th>Serial Performance</th>
<th>Parallel Throughput</th>
</tr>
</thead>
<tbody>
<tr>
<td>Large</td>
<td>4</td>
<td>0</td>
<td>2</td>
<td>2 x 4 = 8</td>
</tr>
<tr>
<td>Small</td>
<td>0</td>
<td>16</td>
<td>1</td>
<td>1 x 16 = 16</td>
</tr>
<tr>
<td></td>
<td>1</td>
<td>12</td>
<td>2</td>
<td>1x2 + 1x12 = 14</td>
</tr>
</tbody>
</table>

ACMP

ACMP area-budget = 16 small cores
Some Analysis

- Each Chip Bounded to N BCEs (Base Core Equivalents)
- One R-BCE Core leaves N-R BCEs
- Use N-R BCEs for N-R Base Cores
- Therefore, 1 + N - R Cores per Chip
- For an N = 16 BCE Chip:

Symmetric: Four 4-BCE cores
Asymmetric: One 4-BCE core & Twelve 1-BCE base cores
Amdahl’s Law Modified

- Serial Fraction 1-F same, so time = (1 – F) / Perf(R)

- Parallel Fraction F
  - One core at rate Perf(R)
  - N-R cores at rate 1
  - Parallel time = F / (Perf(R) + N - R)

- Therefore, w.r.t. one base core:

\[
\text{Asymmetric Speedup} = \frac{1}{\frac{1-F}{\text{Perf}(R)}} + \frac{F}{\text{Perf}(R) + N - R}
\]
Asymmetric Multicore Chip, $N = 256$ BCEs

- Number of Cores = 1 (Enhanced) + $256 - R$ (Base)
Symmetric Multicore Chip, $N = 256$ BCEs

Recall $F=0.9$, $R=28$, Cores=9, $\text{Speedup}=26.7$
Asymmetric Multicore Chip, N = 256 BCEs

- Asymmetric multi-core provides better speedup than symmetric multi-core when N is large

![Graph showing asymmetry speedup comparison between symmetric and asymmetric cores](image-url)

- Speedup for asymmetric cores:
  - F=0.999: R=118 (vs. 28), Cores= 139 (vs. 9), Speedup=65.6 (vs. 26.7)
  - F=0.99: R=41 (vs. 3), Cores=216 (vs. 85), Speedup=166 (vs. 80)
  - F=0.975
  - F=0.9
  - F=0.5
Asymmetric vs. Symmetric Cores

- Advantages of Asymmetric
  + Can provide better performance when thread parallelism is limited
  + Can be more energy efficient
  + Schedule computation to the core type that can best execute it

- Disadvantages
  - Need to design more than one type of core. Always?
  - Scheduling becomes more complicated
    - What computation should be scheduled on the large core?
    - Who should decide? HW vs. SW?
  - Managing locality and load balancing can become difficult if threads move between cores (transparently to software)
  - Cores have different demands from shared resources
How to Achieve Asymmetry

- **Static**
  - Type and power of cores fixed at design time
  - Two approaches to design “faster cores”:
    - High frequency
    - Build a more complex, powerful core with entirely different uarch
  - Is static asymmetry natural? (chip-wide variations in frequency)

- **Dynamic**
  - Type and power of cores change dynamically
  - Two approaches to dynamically create “faster cores”:
    - Boost frequency dynamically (limited power budget)
    - Combine small cores to enable a more complex, powerful core
    - Is there a third, fourth, fifth approach?
Asymmetry via Boosting of Frequency

- **Static**
  - Due to process variations, cores might have different frequency
  - Simply hardwire/design cores to have different frequencies

- **Dynamic**
  - Dynamic voltage and frequency scaling
EPI Throttling

- **Goal:** Minimize execution time of parallel programs while keeping power within a fixed budget
- For best scalar and throughput performance, vary energy expended per instruction (EPI) based on available parallelism
  - \( P = \text{EPI} \times \text{IPS} \)
  - \( P = \) fixed power budget
  - \( \text{EPI} = \) energy per instruction
  - \( \text{IPS} = \) aggregate instructions retired per second

- **Idea:** For a fixed power budget
  - Run sequential phases on high-EPI processor
  - Run parallel phases on multiple low-EPI processors
EPI Throttling via DVFS

- DVFS: Dynamic voltage frequency scaling

- In phases of low thread parallelism
  - Run a few cores at high supply voltage and high frequency

- In phases of high thread parallelism
  - Run many cores at low supply voltage and low frequency
Possible EPI Throttling Techniques


<table>
<thead>
<tr>
<th>Method</th>
<th>EPI Range</th>
<th>Time to Alter EPI</th>
<th>Throttle Action</th>
</tr>
</thead>
<tbody>
<tr>
<td>Voltage/frequency scaling</td>
<td>1:2 to 1:4</td>
<td>100us (ramp Vcc)</td>
<td>Lower voltage and frequency</td>
</tr>
<tr>
<td>Asymmetric cores</td>
<td>1:4 to 1:6</td>
<td>10us (migrate 256KB L2 cache)</td>
<td>Migrate threads from large cores to small cores</td>
</tr>
<tr>
<td>Variable-size core</td>
<td>1:1 to 1:2</td>
<td>1us (fill 32KB L1 cache)</td>
<td>Reduce capacity of processor resources</td>
</tr>
<tr>
<td>Speculation control</td>
<td>1:1 to 1:1.4</td>
<td>10ns (pipeline latency)</td>
<td>Reduce amount of speculation</td>
</tr>
</tbody>
</table>
Boosting Frequency of a Small Core vs. Large Core

- Frequency boosting implemented on Intel Nehalem, IBM POWER7

- Advantages of Boosting Frequency
  + Very simple to implement; no need to design a new core
  + Parallel throughput does not degrade when TLP is high
  + Preserves locality of boosted thread

- Disadvantages
  - Does not improve performance if thread is memory bound
  - Does not reduce Cycles per Instruction (remember the performance equation?)
  - Changing frequency/voltage can take longer than switching to a large core
We did not cover the following slides in lecture. These are for your preparation for the next lecture.
EPI Throttling (Annavaram et al., ISCA’ 05)

- **Static AMP**
  - Duty cycles set once prior to program run
  - Parallel phases run on 3P/1.25GHz
  - Sequential phases run on 1P/2GHz
  - Affinity guarantees sequential on 1P and parallel on 3
  - Benchmarks that rapidly transition between sequential and parallel phases

- **Dynamic AMP**
  - Duty cycle changes during program run
  - Parallel phases run on all or a subset of four processors
  - Sequential phases of execution on 1P/2GHz
  - Benchmarks with long sequential and parallel phases
EPI Throttling (Annavaram et al., ISCA’05)

- Evaluation on Base SMP: 4 Base SMP: 4-way 2GHz Xeon, 2MB L3, 4GB Memory

- Hand-modified programs
  - OMP threads set to 3 for static AMP
  - Calls to set affinity in each thread for static AMP
  - Calls to change duty cycle and to set affinity in dynamic AMP

<table>
<thead>
<tr>
<th>AMP Configuration</th>
<th>Programs</th>
</tr>
</thead>
<tbody>
<tr>
<td>Static AMP: 1P/2GHz or 3P/1.25GHz</td>
<td>wupwise, swim, mgrid, equake, fma3d, art, ammp, BLAST, HMMER</td>
</tr>
<tr>
<td>Dynamic AMP: 1P/2GHz to 4P/1GHz</td>
<td>applu, apsi, FFTW, TPC-H</td>
</tr>
</tbody>
</table>
EPI Throttling (Annavaram et al., ISCA’05)

- Frequency boosting AMP improves performance compared to 4-way SMP for many applications
EPI Throttling

- Why does Frequency Boosting (FB) AMP not always improve performance?

- Loss of throughput in static AMP (only 3 processors in parallel portion)
  - Is this really the best way of using FB-AMP?

- Rapid transitions between serial and parallel phases
  - Data/thread migration and throttling overhead

- Boosting frequency does not help memory-bound phases
Review So Far

- Symmetric Multicore
  - Evolution of Sun’s and IBM’s Multicore systems and design choices
  - Niagara, Niagara 2, ROCK
  - IBM POWERx

- Asymmetric multicore
  - Motivation
  - Functional vs. Performance Asymmetry
  - Static vs. Dynamic Asymmetry
  - EPI Throttling
Design Tradeoffs in ACMP (I)

- **Hardware Design Effort vs. Programmer Effort**
  - ACMP requires more design effort
  - Performance becomes less dependent on length of the serial part
  - Can reduce programmer effort: Serial portions are not as bad for performance with ACMP

- **Migration Overhead vs. Accelerated Serial Bottlenecks**
  + Performance gain from faster execution of serial portion
  - Performance loss when architectural state is migrated/switched in when the master changes
    - Can be alleviated with multithreading and hidden by long serial portion
  - Serial portion incurs cache misses when it needs data generated by the parallel portion
  - Parallel portion incurs cache misses when it needs data generated by the serial portion
Design Tradeoffs in ACMP (II)

- Fewer threads vs. accelerated serial bottleneck
  - + Performance gain from accelerated serial portion
  - - Performance loss due to unavailability of L threads in parallel portion

- This need not be the case → Large core can implement Multithreading to improve parallel throughput
- As the number of cores (threads) on chip increases, fractional loss in parallel performance decreases
Uses of Asymmetry

- So far:
  - Improvement in serial performance (sequential bottleneck)

- What else can we do with asymmetry?
  - Energy reduction?
  - Energy/performance tradeoff?
  - Improvement in parallel portion?
Use of Asymmetry for Energy Efficiency


- Idea:
  - Implement multiple types of cores on chip
  - Monitor characteristics of the running thread (e.g., sample energy/performance on each core periodically)
  - Dynamically pick the core that provides the best energy/performance tradeoff for a given phase
    - “Best core” → Depends on optimization metric
Use of Asymmetry for Energy Efficiency

Figure 1. Relative sizes of the Alpha cores scaled to 0.10 μm. EV8 is 80 times bigger but provides only two to three times more single-threaded performance.

Table 1. Power and relative performance of Alpha cores scaled to 0.10 μm. Performance is expressed normalized to EV4 performance.

<table>
<thead>
<tr>
<th>Core</th>
<th>Peak power (Watts)</th>
<th>Average power (Watts)</th>
<th>Performance (norm. IPC)</th>
</tr>
</thead>
<tbody>
<tr>
<td>EV4</td>
<td>4.97</td>
<td>3.73</td>
<td>1.00</td>
</tr>
<tr>
<td>EV5</td>
<td>9.83</td>
<td>6.88</td>
<td>1.30</td>
</tr>
<tr>
<td>EV6</td>
<td>17.8</td>
<td>10.68</td>
<td>1.87</td>
</tr>
<tr>
<td>EV8</td>
<td>92.88</td>
<td>46.44</td>
<td>2.14</td>
</tr>
</tbody>
</table>
Use of Asymmetry for Energy Efficiency

- **Advantages**
  + More flexibility in energy-performance tradeoff
  + Can execute computation to the core that is best suited for it (in terms of energy)

- **Disadvantages/Issues**
  - Incorrect predictions/sampling → wrong core → reduced performance or increased energy
  - Overhead of core switching
  - Disadvantages of asymmetric CMP (e.g., design multiple cores)
  - Need phase monitoring and matching algorithms
    - What characteristics should be monitored?
    - Once characteristics known, how do you pick the core?
Use of ACMP to Improve Parallel Portion Performance

- Mutual Exclusion:
  - Threads are not allowed to update shared data concurrently

- Accesses to shared data are encapsulated inside *critical sections*

- Only one thread can execute a critical section at a given time

- **Idea**: Ship critical sections to a large core