18-447: Computer Architecture
Lecture 24: Runahead and Multiprocessing

Prof. Onur Mutlu
Carnegie Mellon University
Spring 2012, 4/23/2012
Reminder: Lab Assignment 6

- Due Today

- Implementing a more realistic memory hierarchy
  - L2 cache model
  - DRAM, memory controller models
  - MSHRs, multiple outstanding misses

- Extra credit: Prefetching
Reminder: Lab Assignment 7

- Cache coherence in multi-core systems
  - MESI cache coherence protocol

- Due May 4

- Extra credit: Improve the protocol (open-ended)
Midterm II

- Midterms will be distributed today

- Please give me a 15-minute warning!
Final Exam

- May 10

- Comprehensive (over **all topics** in course)

- Three cheat sheets allowed

- We will have a review session (stay tuned)

- Remember this is 30% of your grade
  - I will take into account your improvement over the course
  - Know the previous midterm concepts by heart
A Belated Note on Course Feedback

- We have taken into account your feedback

- Some feedback was contradictory
  - Pace of course
    - Fast or slow?
    - Videos help
  - Homeworks
    - Love or hate?
  - Workload
    - Too little/easy
    - Too heavy
  - Many of you indicated you are learning a whole lot
Readings for Today

- Required

- Recommended
Readings for Wednesday

- **Required cache coherence readings:**
  - Culler and Singh, *Parallel Computer Architecture*
    - Chapter 5.1 (pp 269 – 283), Chapter 5.3 (pp 291 – 305)
  - P&H, *Computer Organization and Design*
    - Chapter 5.8 (pp 534 – 538 in 4th and 4th revised eds.)

- **Recommended:**
Last Lecture

- Wrap up prefetching
  - Markov prefetching
  - Content-directed prefetching
  - Execution-based prefetching

- Runahead execution
Today

- Wrap-up runahead execution
- Multiprocessing fundamentals
- The cache coherence problem
Memory Latency Tolerance and Runahead Execution
How Do We Tolerate Stalls Due to Memory?

- Two major approaches
  - Reduce/eliminate stalls
  - Tolerate the effect of a stall when it happens

- Four fundamental techniques to achieve these
  - Caching
  - Prefetching
  - Multithreading
  - Out-of-order execution

- Many techniques have been developed to make these four fundamental techniques more effective in tolerating memory latency
Review: Execution-based Prefetchers

- **Idea:** Pre-execute a piece of the (pruned) program solely for prefetching data
  - Only need to distill pieces that lead to cache misses

- **Speculative thread:** Pre-executed program piece can be considered a “thread”

- Speculative thread can be executed
  - On a separate processor/core
  - On a separate hardware thread context (think fine-grained multithreading)
  - On the same thread context in idle cycles (during cache misses)
Review: Thread-Based Pre-Execution


Review: Runahead Execution

**Perfect Caches:**

- Load 1 Hit
- Load 2 Hit

**Small Window:**

- Load 1 Miss
- Load 2 Miss

**Runahead:**

- Load 1 Miss
- Load 2 Miss
- Load 1 Hit
- Load 2 Hit

*Saved Cycles*
Review: Runahead Execution Pros and Cons

- **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
  + Versus other pre-execution based prefetching mechanisms:
    + Uses the same thread context as main thread, no waste of context
    + No need to construct a pre-execution thread

- **Disadvantages/Limitations:**
  -- **Extra executed instructions**
  -- Limited by branch prediction accuracy
  -- Cannot prefetch dependent cache misses. Solution?
  -- Effectiveness limited by available “memory-level parallelism” (MLP)
  -- Prefetch distance limited by memory latency

- Implemented in IBM POWER6, Sun “Rock”
Execution-based Prefetchers Pros and Cons

+ Can prefetch pretty much any access pattern
+ Can be very low cost (e.g., runahead execution)
  + Especially if it uses the same hardware context
  + Why? The processor is equipped to execute the program anyway
+ Can be bandwidth-efficient (e.g., runahead execution)

-- Depend on branch prediction and possibly value prediction accuracy
  - Mispredicted branches dependent on missing data throw the thread off the correct execution path
-- Can be wasteful
  -- speculatively execute many instructions
  -- can occupy a separate thread context
Performance of Runahead Execution

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

Micro-operations Per Cycle

- S95
- FP00
- INT00
- WEB
- MM
- PROD
- SERV
- WS
- AVG

Performance metrics for different scenarios and applications.
Runahead Execution vs. Large Windows

![Graph comparing different window sizes and their impact on micro-operations per cycle.]

- **128-entry window (baseline)**
- **128-entry window with Runahead**
- **256-entry window**
- **384-entry window**
- **512-entry window**

The graph shows the performance of different window sizes in terms of micro-operations per cycle for various applications and services.
Effect of Runahead in Sun ROCK

- Shailender Chaudhry talk, Aug 2008.

![Graph showing the effect of runahead on normalized IPC with varying L2 cache sizes. The graph illustrates that increasing the L2 cache size from 256KB to 64MB improves performance, with a significant boost from 256KB to 512KB.]
Limitations of the Baseline Runahead Mechanism

- **Energy Inefficiency**
  - A large number of instructions are speculatively executed
  - *Efficient Runahead Execution* [ISCA’05, IEEE Micro Top Picks’06]

- **Ineffectiveness for pointer-intensive applications**
  - Runahead cannot parallelize dependent L2 cache misses
  - *Address-Value Delta (AVD) Prediction* [MICRO’05]

- **Irresolvable branch mispredictions in runahead mode**
  - Cannot recover from a mispredicted L2-miss dependent branch
  - *Wrong Path Events* [MICRO’04]
The Efficiency Problem

- % Increase in IPC
- % Increase in Executed Instructions

- Average:
  - 22%
  - 27%
Causes of Inefficiency

- Short runahead periods
- Overlapping runahead periods
- Useless runahead periods

Overall Impact on Executed Instructions

Increase in Executed Instructions:

- **bzip2**: 235%
- **crafty**: 26.5%
- **eon**: 6.2%
- **gap**:
- **gcc**:
- **gzip**:
- **mcf**:
- **parser**:
- **perlbench**:
- **twolf**:
- **vortex**:
- **vvp**:
- **amp**:
- **applu**:
- **apsi**:
- **art**:
- **equake**:
- **farcerek**:
- **fma3d**:
- **galgel**:
- **lucas**:
- **mesa**:
- **mgrid**:
- **sixtrack**:
- **swim**:
- **wupwise**:
- **AVG**:

Legend:
- **baseline runahead**
- **all techniques**
The Problem: Dependent Cache Misses

Runahead: **Load 2 is dependent on Load 1**

![Diagram showing load misses and runahead execution](image)

- Runahead execution cannot parallelize dependent misses
  - wasted opportunity to improve performance
  - wasted energy (useless pre-execution)

- Runahead performance would improve by 25% if this limitation were ideally overcome
Parallelizing Dependent Cache Misses

- **Idea:** Enable the parallelization of dependent L2 cache misses in runahead mode with a low-cost mechanism.

- **How:** Predict the values of L2-miss **address (pointer) loads**
  - **Address load:** loads an address into its destination register, which is later used to calculate the address of another load.
  - as opposed to **data load**

- **Read:**
Parallelizing Dependent Cache Misses

Compute Load 1 Miss Load 2 INV Load 1 Hit Load 2 Miss

Cannot Compute Its Address!

Value Predicted Can Compute Its Address

Load 1 Miss Load 2 Miss Load 1 Hit Load 2 Hit

Saved Speculative Instructions

Saved Cycles
Readings

- Required

- Recommended
Multiprocessors and
Issues in Multiprocessing
Readings for Today

- Required

- Recommended
Readings for Wednesday

- **Required cache coherence readings:**
  - Culler and Singh, *Parallel Computer Architecture*
    - Chapter 5.1 (pp 269 – 283), Chapter 5.3 (pp 291 – 305)
  - P&H, *Computer Organization and Design*
    - Chapter 5.8 (pp 534 – 538 in 4th and 4th revised eds.)

- **Recommended:**
Remember: Flynn’s Taxonomy of Computers


- **SISD**: Single instruction operates on single data element
- **SIMD**: Single instruction operates on multiple data elements
  - Array processor
  - Vector processor
- **MISD**: Multiple instructions operate on single data element
  - Closest form: systolic array processor, streaming processor
- **MIMD**: Multiple instructions operate on multiple data elements (multiple instruction streams)
  - Multiprocessor
  - Multithreaded processor
Why Parallel Computers?

- **Parallelism**: Doing multiple things at a time
- **Things**: instructions, operations, tasks

**Main Goal**
- Improve performance (Execution time or task throughput)
  - Execution time of a program governed by Amdahl’s Law

**Other Goals**
- Reduce power consumption
  - (4N units at freq F/4) consume less power than (N units at freq F)
  - Why?
- Improve cost efficiency and scalability, reduce complexity
  - Harder to design a single unit that performs as well as N simpler units
- Improve dependability: Redundant execution in space
Types of Parallelism and How to Exploit Them

- **Instruction Level Parallelism**
  - Different instructions within a stream can be executed in parallel
  - Pipelining, out-of-order execution, speculative execution, VLIW
  - Dataflow

- **Data Parallelism**
  - Different pieces of data can be operated on in parallel
  - SIMD: Vector processing, array processing
  - Systolic arrays, streaming processors

- **Task Level Parallelism**
  - Different “tasks/threads” can be executed in parallel
  - Multithreading
  - Multiprocessing (multi-core)
Task-Level Parallelism: Creating Tasks

- **Partition a single problem into multiple related tasks (threads)**
  - Explicitly: Parallel programming
    - Easy when tasks are natural in the problem
      - Web/database queries
    - Difficult when natural task boundaries are unclear
  - Transparently/implicitly: Thread level speculation
    - Partition a single thread speculatively

- **Run many independent tasks (processes) together**
  - Easy when there are many processes
    - Batch simulations, different users, cloud computing workloads
  - Does not improve the performance of a single task
MIMD Processing Overview
MIMD Processing

- Loosely coupled multiprocessors
  - No shared global memory address space
  - Multicomputer network
    - Network-based multiprocessors
  - Usually programmed via message passing
    - Explicit calls (send, receive) for communication

- Tightly coupled multiprocessors
  - Shared global memory address space
  - Traditional multiprocessing: symmetric multiprocessing (SMP)
  - Existing multi-core processors, multithreaded processors
  - Programming model similar to uniprocessors (i.e., multitasking uniprocessor) except
    - Operations on shared data require synchronization
Main Issues in Tightly-Coupled MP

- **Shared memory synchronization**
  - Locks, atomic operations

- **Cache consistency**
  - More commonly called cache coherence

- **Ordering of memory operations**
  - What should the programmer expect the hardware to provide?

- **Resource sharing, contention, partitioning**
- **Communication: Interconnection networks**
- **Load imbalance**
Aside: Hardware-based Multithreading

- Coarse grained
  - Quantum based
  - Event based (switch-on-event multithreading)

- Fine grained
  - Cycle by cycle

- Simultaneous
  - Can dispatch instructions from multiple threads at the same time
  - Good for improving execution unit utilization
Parallel Speedup Example

- \(a_4x^4 + a_3x^3 + a_2x^2 + a_1x + a_0\)

- Assume each operation 1 cycle, no communication cost, each op can be executed in a different processor

- How fast is this with a single processor?

- How fast is this with 3 processors?
$R = a_4 x^4 + a_3 x^3 + a_2 x^2 + a_1 x + a_0$

Single processor: 11 operations (data flow graph)

$T_1 = 11$ cycles
\[ R = a_4 x^4 + a_3 x^3 + a_2 x^2 + a_1 x + a_0 \]

Three processors: \( T_3 \) (execute with 3 proc.)

\[ T_3 = 5 \text{ cycles} \]
Speedup with 3 Processors

\[
T_3 = \frac{5 \text{ cycles}}{}
\]

\[
\text{Speedup with 3 processors} = \frac{11}{5} = 2.2
\]

\[
\left( \frac{T_1}{T_3} \right)
\]

Is this a fair comparison?
Revisiting the Single-Processor Algorithm

Better single-processor algorithm:

\[ R = a_4 x^4 + a_3 x^3 + a_2 x^2 + a_1 x + a_0 \]

\[ R = (((a_4 x + a_3) x + a_2) x + a_1) x + a_0 \]

(Horner’s method)

$T_1 = 8\text{ cycles}$

\[
\frac{\text{Speedup with 3 proc.}}{\text{best}} = \frac{T_1}{T_3} = \frac{8}{5} = 1.6
\] (not 2.2)
Superlinear Speedup

- Can speedup be greater than P with P processing elements?

- Cache effects
- Working set effects

- Happens in two ways:
  - Unfair comparisons
  - Memory effects
Utilization, Redundancy, Efficiency

- Traditional metrics
  - Assume all P processors are tied up for parallel computation

- Utilization: How much processing capability is used
  - \( U = \frac{\text{# Operations in parallel version}}{\text{processors} \times \text{Time}} \)

- Redundancy: how much extra work is done with parallel processing
  - \( R = \frac{\text{# of operations in parallel version}}{\text{# operations in best single processor algorithm version}} \)

- Efficiency
  - \( E = \frac{\text{Time with 1 processor}}{\text{processors} \times \text{Time with P processors}} \)
  - \( E = \frac{U}{R} \)
Utilization of Multiprocessor

**Multiprocessor metrics**

**Utilization**: How much processing capability we use.

![Diagram of multiprocessor utilization]

\[
U = \frac{10 \text{ operations (in parallel version)}}{3 \text{ processors} \times 5 \text{ time units}} = \frac{10}{15}
\]

\[
U = \frac{\text{Ops with proc.}}{p \times T_p}
\]
Redundancy: How much extra work due to multiprocessing

\[ R = \frac{\text{Ops with } p \text{ proc. best}}{\text{Ops with 1 proc. best}} = \frac{10}{8} \]

\( R \) is always \( \geq 1 \)

Efficiency: How much resource we use compared to how much resource we can get away with

\[ E = \frac{1 \cdot T_1^\text{best}}{p \cdot T_p^\text{best}} \] (tying up 1 proc for \( T_p \) time units)

\[ = \frac{8}{15} \]  

\( E = \frac{U}{R} \)
Caveats of Parallelism (I)

Why the reality? (diminishing returns)

\[ T_p = \alpha \cdot \frac{T_1}{p} + (1-\alpha) \cdot T_1 \]

parallelizable part/fraction of the single-processor program

non-parallelizable part
Amdahl’s Law

\[
\text{Speedup with } p \text{ proc.} = \frac{T_1}{T_p} = \frac{1}{\frac{\alpha}{p} + (1-\alpha)}
\]

\[
\text{Speedup as } p \to \infty = \frac{1}{1-\alpha} \quad \text{but not for parallel Speedup}
\]

Amdahl’s Law Implication 1

\[ \text{Speedup} = \frac{1}{\frac{1}{\alpha} + \frac{1}{1 - \alpha}} \]

- \( \alpha = 0.98 \)
- \( \alpha = 0.95 \)
- \( \alpha = 0.9 \)

Adding more and more processors gives less and less benefit if \( \alpha < 1 \).
Amdahl’s Law Implication 2

The benefit (speedup) is small until $\alpha \approx 1$.
Caveats of Parallelism (II)

- Amdahl’s Law
  - $f$: Parallelizable fraction of a program
  - $N$: Number of processors
  
  \[
  \text{Speedup} = \frac{1}{1 - f + \frac{f}{N}}
  \]


- Maximum speedup limited by serial portion: Serial bottleneck

- Parallel portion is usually not perfectly parallel
  - Synchronization overhead (e.g., updates to shared data)
  - Load imbalance overhead (imperfect parallelization)
  - Resource sharing overhead (contention among $N$ processors)
Midterm II Results
Midterm II Scores

- **Average**: 148 / 315
- **Minimum**: 58 / 315
- **Maximum**: 258 / 315
- **Std. Dev.**: 52

Midterm 2 Distribution
Midterm II Per-Question Statistics (I)

I. Potpourri

II. Vector Processing

III. DRAM Refresh

IV. Dataflow

V. Memory Scheduling
Midterm II Per-Question Statistics (II)

VI. Caches and Virtual Memory

VII. Memory Hierarchy (Bonus)
We did not cover the following slides in lecture. These are for your preparation for the next lecture.
Sequential Bottleneck

![Graph showing speedup against f (parallel fraction) for different N values: N=10, N=100, N=1000. The graph illustrates the increase in speedup as f approaches 1.](image-url)
Why the Sequential Bottleneck?

- Parallel machines have the sequential bottleneck

- Main cause: **Non-parallelizable operations on data** (e.g. non-parallelizable loops)
  
  for (i = 0; i < N; i++)
  

- Single thread prepares data and spawns parallel tasks (usually sequential)
Another Example of Sequential Bottleneck

```
InitPriorityQueue(PQ);
SpawnThreads();  A

ForEach Thread:

while (problem not solved)

    Solve(SubProblem);
    If(problem solved) break;
    NewSubProblems = Partition(SubProblem);

    Lock(X)
    PQ.insert(NewSubProblem)
    Unlock(X)

    ...

PrintSolution(); E

LEGEND
A,E: Amdahl's serial part
B: Parallel Portion
C1,C2: Critical Sections
D: Outside critical section
```
Bottlenecks in Parallel Portion

- **Synchronization:** Operations manipulating shared data cannot be parallelized
  - Locks, mutual exclusion, barrier synchronization
  - **Communication:** Tasks may need values from each other
    - Causes thread serialization when shared data is contended

- **Load Imbalance:** Parallel tasks may have different lengths
  - Due to imperfect parallelization or microarchitectural effects
    - Reduces speedup in parallel portion

- **Resource Contention:** Parallel tasks can share hardware resources, delaying each other
  - Replicating all resources (e.g., memory) expensive
    - Additional latency not present when each task runs alone
Difficulty in Parallel Programming

- Little difficulty if parallelism is natural
  - “Embarrassingly parallel” applications
  - Multimedia, physical simulation, graphics
  - Large web servers, databases?

- Difficulty is in
  - Getting parallel programs to work correctly
  - Optimizing performance in the presence of bottlenecks

- Much of parallel computer architecture is about
  - Designing machines that overcome the sequential and parallel bottlenecks to achieve higher performance and efficiency
  - Making programmer’s job easier in writing correct and high-performance parallel programs
Cache Coherence
Cache Coherence

- Basic question: If multiple processors cache the same block, how do they ensure they all see a consistent state?
The Cache Coherence Problem

P1

ld r2, x

Interconnection Network

Main Memory

P2

1000

Id r2, x
The Cache Coherence Problem

P1

ld r2, x

1000

P2

ld r2, x

1000

Interconnection Network

Main Memory

ld r2, x

1000

x 1000
The Cache Coherence Problem

```
l d r2, x
a dd r1, r2, r4
s t x, r1
```

```
l d r2, x
```

```
2000
```

```
1000
```

Interconnection Network

Main Memory
The Cache Coherence Problem

ld r2, x
add r1, r2, r4
st x, r1

ld r5, x

ld r2, x

Should NOT load 1000

ld r5, x

Interconnection Network

P1

2000

P2

1000

Main Memory

x 1000
Cache Coherence: Whose Responsibility?

- Software
  - Can the programmer ensure coherence if caches are invisible to software?
  - What if the ISA provided the following instruction?
    - FLUSH-LOCAL A: Flushes/invalidates the cache block containing address A from a processor’s local cache
    - When does the programmer need to FLUSH-LOCAL an address?
  - What if the ISA provided the following instruction?
    - FLUSH-GLOBAL A: Flushes/invalidates the cache block containing address A from all other processors’ caches
    - When does the programmer need to FLUSH-GLOBAL an address?

- Hardware
  - Simplifies software’s job
  - One idea: Invalidate all other copies of block A when a processor writes to it
Snoopy Cache Coherence

- Caches “snoop” (observe) each other’s write/read operations
- A simple protocol:

- Write-through, no-write-allocate cache
- Actions: PrRd, PrWr, BusRd, BusWr