Lecture 1

Introduction and Course Overview

Daniel Sanchez and Joel Emer

6.888 Parallel and Heterogeneous Computer Architecture
Spring 2013
Why 6.888?

- The current revolution: Parallel computing

- The impending revolution: Heterogeneous computing

[Produced with CPUDP, cpudb.stanford.edu]
Classic CMOS Scaling

- Moore’s law + Denard scaling: Each generation (e.g., 90→65nm),

- 2x transistors, 1.4x frequency, same power \(\rightarrow\) area-constrained

\[P = P_{\text{dyn}} + P_{\text{leak}}\]

\[P_{\text{dyn}} = \frac{1}{2} CV^2 f\]

\[P_{\text{leak}} = V \times I_{\text{leak}} \ll P_{\text{dyn}}\]

[Adapted from “Advancing Systems Without Technology Progress” outbrief of DARPA/ISAT Workshop, 2012]
Current CMOS Scaling

- Frequency and supply voltage scaling are mostly exhausted

- 2.0x transistors, same frequency, 1.4x power → power-constrained
Parallelism and Heterogeneity Trade-offs

- **Good news:** Plenty of efficiency improvements
  - Simple cores have $\sim 10x$ lower energy/instruction than complex uniprocessors $\Rightarrow$ can scale to about $\sim 1000$ simple cores within power constraints
  - Specialized compute units have $\sim 10-1000x$ perf/energy savings over general-purpose cores

- **Bad news:** Harder to build and use, less general

- **Trillion-dollar question:** What is the right balance between efficiency, generality, and ease of use?
6.888 Goals

- Learn about the state of the art, both hardware and software aspects
  - Architectures and programming models
  - Hardware changes no longer transparent to software stack → must consider both to be successful!

- Improve on the state of the art
  - Lots of open problems!
6.888 Team

- Instructors: Daniel Sanchez and Joel Emer

- TA: Mieszko Lis

- Administrative support: Sally Lee
Class Basics

- Lectures: Mon & Wed, 1-2:30pm, room 1-135
  - Format: Short presentation + paper-based discussions
  - Need to read papers beforehand and contribute to discussion

  - Includes course info, calendar, readings, contact info, and office hours
Class Topics

- Structured in four parts (~1 month each):
  1. Parallel architectures and programming models
     - How current multicores are built, how to program and evaluate them
  2. Communication, synchronization, and the memory hierarchy
     - Advanced parallel systems, including techniques to ease parallel programming (e.g., TM, TLS)
  3. Specialized and heterogeneous computing
     - GPUs, vector, FPGAs, reconfigurable, and beyond
  4. Cross-cutting issues
Prerequisites

- Prerequisites: 6.004 or equivalent
  - Simple pipelined cores, caches, virtual memory, basic OS

- 6.823 (or similar) useful but not required
  - Today's lecture reviews 6.823 aspects needed in 6.888

- Parallel/performance-oriented programming (e.g., 6.172) useful but not required
Class Participation & Papers

- We expect you to participate regularly in class, and part of your grade depends on it.

- Syllabus lists readings for each lecture, plus a list of optional, additional readings.

- Tips for reading papers:
  - Read abstract, intro, and conclusions first.
  - Skim the paper first, then do a detailed reading.
  - Read critically, keep notes on questions and potential issues.
  - Look up references that seem important or missing.
Assignments

- Project: Research-oriented, should address an open question in the field
  - Propose your own topic or ask us for one
  - We’ll give you access to infrastructure (simulators, benchmarks, compute resources)
  - Milestones: Initial proposal (Mar 18), progress report (Apr 17), presentations (May 13), final report (May 15)
- Seminar: After the first month, select a topic from one of the upcoming lectures, develop a short presentation and lead the class discussion
- Homework: Single assignment during the first month
Grading & Rules

- **Grading breakdown:**
  - Project: 60%
  - Seminar: 15%
  - Class participation: 15%
  - Homework: 10%

- **Two late days for assignments**
  - Tip: reserve for project

- **Collaboration policy:** All collaboration OK, but
  - Must list all sources of external help
  - Follow MIT academic integrity rules
We Want Your Feedback!

- Aside from class participation...

- Small course, first time it’s taught → your feedback is really important
  - Should be challenging, but useful and fun

- We’re open to comments, suggestions, and willing to be dynamic
Rest of Today: Parallelism in Modern Multicores (ILP, TLP, and DLP)

- **Goals:**
  - Understand how general-purpose multicores exploit parallelism
  - Understand bottlenecks & insights into solving them

- **Today: Focus on Instruction-Level Parallelism**
  - Wide & superscalar pipelines
  - Prediction, renaming & out-of-order execution
  - Challenges and limitations of advanced processors

- **Next week: Thread and Data-level parallelism, memory hierarchy**
The Big Picture

[Slides 16-42 based on material from Sanchez & Kozyrakis]
Iron Law of Performance:

\[
\text{CPI} = \text{CPI}_{\text{ideal}} + \text{CPI}_{\text{stall}}
\]

- **CPI\text{\_ideal:** cycles per instruction if no stall

CPI\text{\_stall** contributors

- Data dependences: RAW, WAR, WAW
- Structural hazards
- Control hazards: branches, exceptions
- Memory latency: cache misses
5-stage Pipelined Processors
(MIPS R3000 circa 1985)

- **Advantages**
  - $CPI_{ideal}$ is 1 (pipelining)
  - No WAW or WAR hazards
  - Simple, elegant
    - Still used in ARM & MIPS processors

- **Shortcomings**
  - Upper performance bound is $CPI=1$
  - High latency instructions not handled well
    - 1 stage for accesses to large caches or multiplier
    - Clock cycle is high
  - Unnecessary stalls due to rigid pipeline
    - If one instruction stalls anything behind it stalls
Improving 5-stage Pipeline Performance

- Higher clock frequency (lower CCT): **deeper pipelines**
  - Overlap more instructions

- Higher $CPI_{\text{ideal}}$: **wider pipelines**
  - Insert multiple instructions in parallel in the pipeline

- Lower $CPI_{\text{stall}}$:
  - *Diversified pipelines* for different functional units
  - *Out-of-order execution*

- Balance conflicting goals
  - Deeper & wider pipelines $\Rightarrow$ more control hazards
  - *Branch prediction*

- It all works because of *instruction-level parallelism (ILP)*
Instruction Level Parallelism (ILP)

\[ D = 3(a - b) + 7ac \]

- **Sequential execution order**
  - ld a
  - ld b
  - sub a-b
  - mul 3(a-b)
  - ld c
  - mul ac
  - mul 7ac
  - add 3(a-b)+7ac
  - st d

- **Data-flow execution order**

```
ld b -> -
  v
ld a -> *
  v
mul 3(a-b)
  v
mul ac  
  v
mul 7ac  
  v
add 3(a-b)+7ac
  v
  st d
```
Deeper Pipelines

- Idea: break up instruction into N pipeline stages
  - Ideal CCT = 1/N compared to non-pipelined
  - So let’s use a large N!

- Other motivation for deep pipelines
  - Not all basic operations have the same latency
    - Integer ALU, FP ALU, cache access
  - Difficult to fit them in one pipeline stage
    - CCT must be large enough to fit the longest one
  - Break some of them into multiple pipeline stages
    - e.g., data cache access in 2 stages, FP add in 2 stage, FP mul in 3 stage…
Limits to Pipeline Depth

- Each pipeline stage introduces some overhead (\(O\))
  - Delay of pipeline registers
  - Inequalities in work per stage
    - Cannot break up work into stages at arbitrary points
  - Clock skew
    - Clocks to different registers may not be perfectly aligned

- If original CCT was \(T\), with \(N\) stages CCT is \(T/N + O\)
  - If \(N \to \infty\), speedup = \(T / (T/N + O) \to T/O\)
    - Assuming that IC and CPI stay constant
  - Eventually overhead dominates and deeper pipelines have diminishing returns
Pipelining Limits?

![Graph showing frequency, CPI, performance, and power vs. pipeline depth.][1]

[1]: edgrochowski.com/2013/06/01/57-some-good-some-bad.html
Deeper Pipelines Review

- Advantages: higher clock frequency
  - The workhorse behind multi-GHz processors
  - Opteron: 11, UltraSparc: 14, Power5: 17, Pentium4: 22/34; Nehalem: 16

- Cost
  - Complexity: more forwarding & stall cases

- Disadvantages
  - More overlapping $\Rightarrow$ more dependencies $\Rightarrow$ more stalls
    - $\text{CPI}_{\text{stall}}$ grows due to data and control hazards
  - Clock overhead becomes increasingly important
  - Power consumption
Wider or Superscalar Pipelines

- **Idea:** operate on $N$ instructions each clock cycle
  - Known as wide or superscalar pipelines
  - $CPI_{\text{ideal}} = 1/N$

- **Options (from simpler to harder):**
  - One integer and one floating-point instruction
  - Any $N=2$ instructions
  - Any $N=4$ instructions
  - Any $N=?$ Instructions
    - What are the limits here?
Superscalar Pipelines Review

- Advantages: lower $CPI_{\text{ideal}} \frac{1}{N}$
  - Opteron: 3, UltraSparc: 4, Power5: 8, Pentium4: 3; Core 2: 4; Nehalem: 4

- Cost
  - Need wider path to instruction cache
  - Need more ALUs, register file ports, ...
  - Complexity: more forwarding & stall cases to check

- Disadvantages
  - Parallel execution $\Rightarrow$ more dependencies $\Rightarrow$ more stalls
    - $CPI_{\text{stall}}$ grows due to data and control hazards
- **Idea:** decouple the execution portion of the pipeline for different instructions

- **Common approach:**
  - Separate pipelines for simple integer, integer multiply, FP, load/store

- **Advantage:**
  - Avoid unnecessary stalls
  - E.g. slow FP instruction does not block independent integer instructions

- **Disadvantages**
  - WAW hazards
  - Imprecise (out-of-order) exceptions
Putting it All Together: A Modern Superscalar Out-of-Order Processor

I-Cache

Branch Predict

Fetch Unit

Instruction Buffer

Decode/Rename

Dispatch

Reservation Stations

Int Int FP FP L/S L/S

Reorder Buffer

Retire

Write Buffer

D-Cache

In Order

Out Of Order

In Order
Branch Penalty

- >3 cycles to resolve a branch/jump
  - Latency of I-cache
  - Decode & execute latency
  - Buffering

- Cost of branch latency?
  - Assume 5 cycles to resolve & 4-way superscalar
  - Cost of branch = 5*4 instructions

- Typical programs:
  - 1 branch every 4 to 8 instructions
Branch Prediction

- Goal: eliminate stalls due to taken branches
  - Gets more critical as pipeline gets longer & wider

- Idea: dynamically predict the outcome of control-flow instructions
  - Predict both the branch condition and the target
  - Works well because several branches have repeated behavior
    - E.g. branches for loops are usually taken
    - E.g. termination/limit/error tests are usually not taken

- Why predict dynamically?
  - Branch behavior often difficult to analyze statically
  - Branch behavior may change during program execution
Predicting the Branch Condition: Simple Branch History Table (BHT)

- Basic idea:
  - Next branch outcome likely to be same as last one

- A $2^m \times 1$ bit table

- Algorithm:
  - Use $m$ least significant bits to access predictor
  - If Bit == 0 predict not-taken
  - If Bit == 1 predict taken
  - When prediction verified, update table if wrong
Predicting the Target Address: Branch Target Buffer (BTB)

- BTB: a cache for branch targets
  - Stores targets for taken branches, jr, function calls
  - Reduce size: don’t store prediction for not taken branches
  - Algorithm: access in parallel with I-cache
    - If hit, use predicted target
    - If miss, use PC+ 16 (assuming 4-way fetch)
  - Must update when prediction verified
Basic ideas

- Use >1b per BHT entry to add hysteresis
- Use PC & global branch history to address BHT
- Detect global and local correlation between branches
  - e.g. nested if-then-else statements
  - e.g. short loops
- Use multiple predictors and select most likely to be correct
  - Capture different patterns with each predictor
- Measure and use confidence in prediction
  - Avoid executing instructions after difficult to predict branch
- Neural-nets, filtering, separate taken/non-taken streams, ...

What happens on mispredictions

- Update prediction tables
- Flush pipeline & restart from mispredicted target (expensive)
Dealing with WAR & WAW: Register Renaming

- WAR and WAW hazards do not represent real data communication
  1. \( R1 = R2 + R3 \)
  2. \( R4 = R1 + R5 \)
  3. \( R1 = R6 + R7 \)

- If we had more registers, we could avoid them completely!

- Register renaming: use more registers than the 32 in the ISA
  - Architectural registers mapped to large pool of physical registers
  - Give each new “value” produced its own physical register

- Before & after renaming
  - Before: \( R1 = R2 + R3 \)  \( R4 = R1 + R5 \)  \( R1 = R6 + R7 \)  \( R6 = R1 + R3 \)
  - After: \( R8 = R6 + R7 \)  \( R9 = R8 + R3 \)
Dealing with Unnecessary Ordering: Out-of-Order Dispatch

- **In-order execution**: instruction dispatched to a functional unit when
  - All older instructions have been dispatched
  - All operands are available & FU available

- **Out-of-order execution**: instruction dispatched when
  - All operands are available & FU available

- **Out-of-order execution recreates the data-flow order**

- **Implementation**
  - Reservation stations or instruction window
  - Keep track when operands become available
Dealing with Memory Ordering

- When can a load read from the cache?
  - Option 1: when its address is available & all older stores done
  - Option 2: when its address is available, all older stores have address available, and no RAW dependency
  - Option 3: when its address is available
    - Speculate no dependency with older stores, must check later

- When can a store write to the cache?
  - It must have its address & data
  - All previous instructions must be exception-free
  - It must be exception-free
  - All previous loads have executed or have address
    - No dependency

- Implementation with ld/st buffers with associative search
Dealing with Precise Exceptions: Reorder Buffer

- Precise exceptions: Exceptions must occur in same order as in unpipelined, single-cycle processor
  - Older instruction first, no partial execution of younger instructions

- Reorder buffer: A FIFO buffer for recapturing order
  - Space allocated during instruction decode: in-order
  - Result updated when execution completes: out-of-order
  - Result written to registers or write-buffer: in-order
    - Older instruction first
    - If older instruction not done, stall
    - If older instruction has exception, flush buffer to eliminate results of incorrectly executed instructions
Putting it All Together: A Modern Superscalar Out-of-Order Processor
Memory Hierarchy in Modern Processors

- **Instruction cache:**
  - 8 to 64KB, 2 to 4 way associative, 16 to 64B blocks, wide access

- **Data cache**
  - 8 to 64KB, 2 to 8 way associative, 16 to 64B blocks, multiported

- **2nd level unified cache**
  - 256KB to 4MB, >4-way associative, multi-banked

- **Prefetch engines**
  - Sequential prefetching for instructions/data
    - When a cache line is accessed, fetch the next few consecutive lines
  - Strided prefetching for data
    - Detect a[i*k] type of accesses and prefetch proper cache lines

- **TLBs**
The Challenges of Superscalar Processors

- Clock frequency: getting close to pipelining limits
  - Clocking overheads, CPI degradation
- Branch prediction & memory latency limit the practical benefits of out-of-order execution
- Power grows superlinearly with higher clock & more OOO logic
- Design complexity grows exponentially with issue width

- Limited ILP \(\rightarrow\) Must exploit TLP and DLP
  - Thread-Level Parallelism: Multithreading and multicore
  - Data-Level Parallelism: SIMD instructions
Putting it all together: Intel Core i7 (Nehalem)

- 4 cores/chip, 2 threads/core
- 16 pipeline stages, ~3GHz
- 4-wide superscalar
- Out of order, 128-entry reorder buffer
- 2-level branch predictors
- Caches:
  - L1: 32KB I + 32KB D
  - L2: 256KB
  - L3: 8MB, shared
- Huge overheads vs simple, energy-optimized cores!
Modern processors rely on a handful of important techniques

- **Caching**
  - Instruction, data, page table

- **Prediction**
  - Branches, memory dependencies, values

- **Indirection**
  - Renaming, page tables

- **Dependence-based reordering**
  - Out-of-order execution

Modern processors: Main objective is high ILP

- High frequency, high power consumption
- Requires high memory bandwidth and low latency
- High price to pay for performance, but simple to use
Readings for Next Monday

- 3 short (~6 page) papers
  1. The Task of a Referee
  2. Roofline
  3. Niagara