Introduction to CS422

Agenda

- Course outline
- Performance measurement
  - CPI equation
- Research goals and practices

Goal of the course

- Goal is to understand the amazing performance growth that has taken place in the last 50+ years of semiconductor chip industry
  - Focus more on architectural aspects
  - Little bit on integrated circuits (enough to be able to evaluate the complexity of an architectural enhancement)
- What is computer architecture?
  - *The structure of a computer that a machine language programmer must understand to write a correct (timing independent) program for that machine* [Amdahl, Blaauw and Brookes, 1964 (IBM 360 team)]
  - Loosely speaking, it is the science of designing computers; online algorithms that exploit application's run-time behavior

53-year journey

- Glorious failures and some notable successes
- Started with Intel 4004 in 1971
  - http://www.intel4004.com/
- Today we talk about more than one billion transistors on a chip
- Die size has increased steadily SiO_2 slab
- Minimum feature size has shrunk from 10 micron in 1971 to 7 nm today G any component on a
- Various innovations made easy by Moore's law
  - Transistor density doubles every 18–24 months
  - What has an architect done with so many transistors in these 50+ years?

Die generally Transistor length is min

The leaps

- Pipelining
- Caches
- Multi-issue (superscalar) → parallel execution
- Out-of-order scheduling → reorder
- Control speculation → branch predictions
- Data speculation → prefetch,
- Multiprocessors, multi-cores and many-cores
- Cross-cutting problems: verification complexity, power consumption, interconnects, extracting parallelism from applications
- Enablers: shrinking process, compilers, and OS to some extent

buffering of instr to check bad me

more small processors to do the above techniques

Performance Measurement

Data speculation → if we can speculate what data will be req, prefetch them.

Speculate value of a variable under execution

Address speculation → no correctness problem, just cache polluting

Value speculation → not yet commercial

↓ req buffering to check bad me of instr

Why?

- Primarily two reasons
  - To measure how good a computer is
  - To understand the bottlenecks in every stage of a design and allocate resources accordingly

Metrics

- For desktop and workstations in most cases it is execution time
  - Also known as response time $P = \frac{1}{RT}$
  - Reciprocal of performance
- For servers it is normally throughput
  - How many jobs can get done in unit time
  - Almost always a response time limit (per job) is also imposed
  - Parallel execution time or FLOPS in HPC workloads

amazon etc

supercomputers

- For embedded processors it is also execution time
  - In many situations a hard or soft real time deadline is imposed
  - Hard deadlines cannot be missed, soft deadlines can be missed in a limited cases
- Goal: maximize performance within power budget

rocket launcher at count=0 (Hard deadline)

Benchmark

- Want to compare two processors by measuring their performance
  - Need some standardized set of programs
  - These are called benchmark programs
  - Each market sector has a different focus: needs different set of benchmark

Benchmark

- Types of benchmark
  - Real applications: taken from day to day applications; may have portability problems; ideal for final performance report, not good for diagnosis purpose: in many cases overly complex and do not provide any insight into what's wrong
  - Modified applications: enhanced portability, possible to focus on particular aspects of CPU (e.g. less I/O if purpose is to measure CPU)
  - Kernels: frequently used code segments e.g., Livermore loops, Linpack; can focus on one CPU feature at a time
  - Toys: small code snippets to quickly test your intuition; must not be used for performance measurement
  - Synthetic: same philosophy as kernels, but not part of any real application; cooked up to model average application behavior; should be avoided
  - Microbenchmark: idea is to quickly get a feel

Performance measurement

final evaluation

debugging and diagnosing

Desktop benchmarks

- SPEC CPU is the standard
  - Provided by Standard Performance Evaluation Corporation (SPEC) *http://www.spec.org/*
  - Currently in the 6th generation: SPEC89, SPEC92, SPEC95, SPEC2000, SPEC2006, SPEC2017
  - Has two types of programs: integer and floating-point to stress different CPU units (C, C++, Fortran)
  - If you are introducing a new processor for desktop or uniprocessor workstation/server, you are supposed to report performance of all programs in the suite
  - For graphics performance two available benchmarks: SPECviewperf and SPECapc
  - For Java run-time environment's performance: SPECjvm

database ops

int only → int ops

floating point → both int & FP ops, but mainly FP

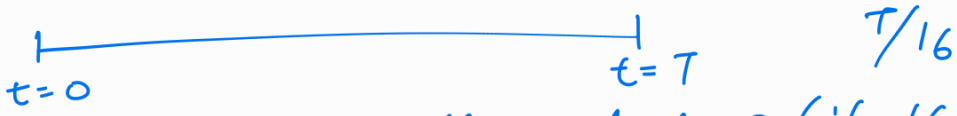
SPECrate INT2017

- 10 applications covering a wide range
  - perl
  - gcc (part of GNU C compiler)
  - mcf (public transit scheduling)
  - omnetpp (discrete event simulation of ethernet network)
  - xalancbmk (XML processing)
  - x264 (video compression)
  - deepsjeng (alpha-beta tree search for game of chess)
  - leela (monte-carlo tree search for game of go)
  - exchange2 (recursive solution generator for Sudoku)
  - xz (data compression)
- 10 applications in SPECspeed INT2017
- 13 applications in SPECrate FP2017
- 10 applications in SPECspeed FP2017

Processors today are multi-core. The SPEC rate are single threaded programs → only one core occupied. So we run 16 copies of SPEC rate apps to keep all cores busy & we tell throughput → overall / time of each appl. (if 16 core machine)

SPEC speed → parallelise the apps above & run only one copy on multicore to evalute performance

SPEC rate



We start all 16 copies together at $t=0$ (if 16 core machine) and report $T/16$ where T = last copy to finish

What is the reality?

- SPEC2017 applications may not capture an average PC user's demands
  - Three benchmarks are available for PC running Windows
  - Business Winstone: A script running Netscape and other Office products; tries to simulate multiprogramming among these
  - CC Winstone: Runs multiple applications focused on content creation e.g. Photoshop, Premiere, audio-editing programs, Navigator
  - Winbench: Set of kernels to measure CPU performance, video performance, disk performance etc.
  - Check out *http://www.etestinglabs.com/benchmarks/*

Server benchmarks

- Typically two metrics
  - Throughput, important for concurrent independent transactions
  - Parallel execution time or FLOPS, important in HPC community
- SPECrate
  - Not the most reliable measure: run a copy of the same SPEC application on each processor and report average number of jobs finished per unit time
  - One big aspect of servers that is missing in SPECrate is the I/O throughput (disk as well as network)
- SPECSFS server file system
  - Tests the performance of NFS using a series of file server requests; measures CPU, disk and network throughput; also puts response time limit on each request

Server benchmarks

- SPECWeb doesn't provide web server
  - Web server benchmark; simulates multiple clients requesting both static and dynamic pages from a server; also clients can upload data on the server
  - Provides the input string only; must be run on some web server such as apache
- SPECjAppServer, SPECjbb, SPECjEnterprise, SPECjms
  - Java client/server
- LAPACK, ScaLAPACK, BLAS
  - Linear algebra routines typically used to measure supercomputer performance
- PARSEC, SPLASH, FFTW, SPEC MPI, SPEC OMP
  - Parallel execution performance measurement on multiprocessors, multi-cores, many-cores

Server benchmarks

- Transaction processing (TP)
  - Probably the most widely used benchmark in server community
  - Measures database access and update throughput of a server such as MySQL, Oracle, IBM DB2, PostgreSQL
  - Simple examples: airline reservation, bank ATM
  - Complex systems: complicated query processing e.g. online book shops (amazon)
  - Provided by Transaction Processing Council (TPC)
  - TPC-A was the first one; now we have TPC-C (complex queries), TPC-H (unrelated queries), TPC-R (DBMS is optimized based on past query pattern), TPC-W (business-oriented transactional web server), TPC-E (OLTP workload of a brokerage firm)
  - The measured metric is transactions per second along with response time limit for individual transactions
  - Check out *http://www.tpc.org/*

Embedded sector

- Quite difficult to come up with a good benchmark
  - Wide range of applications
  - Requirements vary widely from one system to another
  - Reality: in many cases an embedded system designer designs his/her own benchmarks which are either the target applications or kernels extracted from them
  - Today the best known benchmark set is offered by EEMBC (Embedded Microprocessor Benchmark Consortium)
  - EEMBC has five types of applications: automotive/industrial (16 kernels), consumer (5 multimedia kernels), networking (3 kernels), office automation (4 graphics and text processing kernels), telecommunications (6 filtering and DSP kernels)
  - Check out *http://www.eembc.org/*

Performance comparison

- Root of most debates and confusion
  - Consider three processors A, B, C and two benchmark applications P1, P2
  - A executes P1 in 1 second and P2 in 1000 seconds
  - B executes P1 in 10 seconds and P2 in 100 seconds
  - C executes P1 in 20 seconds and P2 in 20 seconds
  - How do you summarize these?
  - One consistent way: report total time to execute P1 and P2, and compare them
  - Along the same line it is possible to report arithmetic mean of the total e.g. in this case 0.5(t(P1) + t(P2))
  - On average A takes 500.5s, B takes 55s, C takes 20s to execute P1 and P2

Performance comparison

- Is it a fair comparison?
  - What could be wrong?
  - Weighted execution time: AM assigns equal weight
  - Could assign equal-time weights with respect to a particular machine: equal-time weights with respect to B are 0.909 and 0.091
  - Possible to play foul if input is not properly specified
- Normalized performance
  - Normalize execution times to some baseline processor
  - So now we are really talking about ratios
  - Arithmetic mean or geometric mean of ratios?
  - Does geometric mean correspond to execution time?

Performance comparison

- Geometric mean could be misleading
  - Suppose designer of A does an optimization that brings down the time to execute P2 to 500s
  - Designer of B does an optimization that brings down the time to execute P1 to 5s
  - Geometric mean will continue to show these two processors at the same level
  - Geometric mean is oblivious to absolute savings
  - But designer of A is probably smarter than that of B
- In summary
  - Provide geometric mean or harmonic mean of performance with respect to a baseline; the type of mean really depends on metric
  - Don't cheat intentionally

$\frac{E_{A\,new}}{E_{A\,old}}$

$\frac{E_{B\,new}}{E_{B\,old}}$

both = 2

$\{X_1, X_2, \ldots X_n\}$

$\{Y_1, Y_2, \ldots Y_n\}$

$\left(\frac{X_1}{Y_1} \cdot \frac{X_2}{Y_2} \cdot \ldots \frac{X_n}{Y_n}\right)^{1/n} = \frac{(X_1 \ldots X_n)^{1/n}}{(Y_1 \ldots Y_n)^{1/n}}$

Amdahl's law

- Make the common case fast
  - Quite intuitive: no point investing time and money to optimize part of a system that gets invoked 1% of time
  - Suppose a program takes time *t* to execute on a processor
  - A particular section of the program can be enhanced by some optimization in the processor; suppose *x* fraction of entire execution time is spent in this particular section of the program
  - The optimization in the processor can speed up execution of this section by *y* times
  - Amdahl's law says that overall speedup due to this optimization is $t / t_{new} = t / (t - tx + tx/y) = 1 / (1 - x + x / y)$
  - Remember that fraction *x* is measured before the enhancement is applied

Amdahl's law

- Look for portions of program that takes maximum amount of time to execute
  - Allocate resources and design time proportionate to execution time
  - As *x* increases the achieved speedup goes up for a fixed *y*; as *y* increases, speedup remains limited by *x*
  - Amdahl's law is usually used to compare design alternatives i.e. which design would bring more performance
  - Example: FP square root is critical in graphics applications; two design choices: implement a FP square root hardware to improve *sqrt* execution by 10 times or improve all FP instructions by 2 times; suppose FP sqrt takes 20% of execution time while 50% time is spent in all FP instructions in the current processor; which design choice is better?

Amdahl's law

- Option#1
  - x = 0.2, y = 10
  - Speedup = 1/(1 – x + x/y) = 1/0.82
- Option#2
  - x = 0.5, y = 2
  - Speedup = 1/(1 – x + x/y) = 1/0.75
- Option#2 has higher speedup

Amdahl's law

- Amdahl's law can be used to derive upper bound on achievable speedup in a parallel computer
  - Suppose a sequential program takes time t to run on a single processor
  - A profiler shows that a fraction s of this time is spent in executing inherently sequential portions of the program
  - The remaining time can be perfectly parallelized on arbitrary number of processors
  - Maximum achievable speedup = t / (s*t + (1 – s)*t / P) which is 1 / (s + (1 – s) / P) on P processors
  - In the limit, speedup gets capped at 1 / s
    - Even if s is 0.05, speedup cannot be more than 20 (still ignores communication overhead)
    - Need almost embarrassingly parallel algorithms with near-zero communication to fully utilize even a medium-scale parallel computer

CPI equation

- We have seen
  - How to compare two processors
  - How to decide between feasible optimizations
  - Next questions: how do we measure execution time? Which are the determinant factors?
- Assume that we want to calculate the execution time of a program
  - Execution time = Clock cycles to execute × cycle time
  - Executed clock cycles = number of executed instructions × average cycles per instruction
  - Execution time = instruction count × CPI × cycle time
  - Cycle time is also same as reciprocal of frequency (in appropriate unit)
  - Execution time equally depends on three components

dynamic instruction count

Execution cycles = instr count × CPI

cycle time fixed later.

CPI equation

- Each component can be improved to get reduced execution time
  - Reducing instruction count of a program normally depends on the instruction set of the processor and the smartness of the compiler e.g. separate equality check and branch instructions can be fused into one instruction such as bne or beq; similarly compiler can identify simple optimizations: ANDing with a mask and check instead of shift and AND and check
  - Example#1: if (x == y) {…} can be translated using beq or bne
  - Example#2: if ((x >> n) & 0x1) can be written as if (x & mask) where mask is prepared by the compiler as (1 << n) provided n is known at compile time
    - If n <= 16, one andi instruction is enough; else two instructions (lui, and)

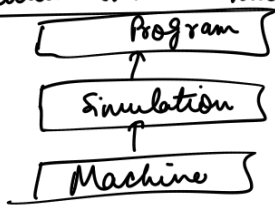
CPI equation

- Each component can be improved to get reduced execution time
  - Reducing instruction count of a program normally depends on the instruction set of the processor and the smartness of the compiler e.g. separate equality check and branch instructions can be fused into one instruction such as bne or beq; similarly compiler can identify simple optimizations: ANDing with a mask and check instead of shift and AND and check
  - Reducing CPI depends on processor architecture, namely, how much parallelism it can extract
  - Frequency of a processor depends on semiconductor technology as well as processor architecture
  - Architectural enhancements (such as deep pipelines) to improve frequency may increase CPI if not careful

CPI equation

- Example
  - Consider the last example of GPU with following additional data
  - Current GPU does not have FPSQRT instruction and FPSQRT is emulated in software (large CPI)
  - Frequency of FP instructions is 25%
  - Average CPI of these instructions is 4.0
  - Average CPI of non-FP instructions is 1.33
  - Frequency of FPSQRT operation is 2%
  - CPI of FPSQRT operation is 20.0
  - One design alternative is to reduce CPI of FPSQRT to 2
  - Other alternative is to reduce CPI of all FP operations to 2.0
  - Which one is better?

Execution driven Simulation



compiler → static instruction count
we want dynamic

CPI equation

- Baseline CPI = 0.25*4 + 0.75*1.33 = 1.9975
- CPI of first design (CPI of FPSQRT from 20 to 2) = 1.9975 – 20*0.02 + 2*0.02 = 1.6375
- CPI of second design (CPI of FP from 4 to 2) = 1.9975 – 0.25*4 + 0.25*2 = 1.4975
- Second design has lower CPI and hence, better

MAINAK CS422 29

mimics everything: out-of-order, pipelines etc

CPI equation

- How to get these three parameters? → mimic behavior of a processor
  - CPU designers normally use detailed simulators to get exact behavior of program execution
  - Simulations can be done at different levels of accuracy
  - Trace-driven simulation: obtain a trace of executed instructions; complex interaction in pipeline is not possible to model
  - Execution-driven simulation: an accurate model of the processor and memory system is designed in software and programs are run on this simulator; most accurate and most time-consuming
  - A user can exploit the performance counters to get a rough estimate of time spent on certain code segments and the number of instructions in those segments (frequency is known) to check the chip made
  - Static profiling of the program can also provide some information about instruction distribution

reg running the prog

diff from assembly as it is exactly the trace (and simulator consumes it.

4 counters only → no. of cycles spent

Functional Simulator – mimics the operations / functions. No memory
↳ mimics the func behaviour

Simulator gets accurate ⇒ becomes slower.

may not be exactly the seq of instr that CPU executes for that prog

Trace – reg running the prog to generate trace. Trace doesn't contain the non-committed instrs. Committed instr → executed by CPU completely

Principle of locality

- Programs are not random pieces of code → loops/ recursions or both
  - Rule of thumb: 90% of time is spent in 10% of code
  - Same locality principle applies to data accesses also
  - Spatial locality: closely spaced data are accessed closely in time
  - Temporal locality: currently accessed data are likely to be accessed in near future
  - Exploit locality in design e.g. caches try to exploit temporal locality while prefetching exploits spatial locality

MAINAK CS422 31

Exploit parallelism

- Mantra of today's computer systems
  - Parallelism at different levels
  - Have more disks to improve I/O throughput
  - Have more memory banks to support parallel data access
  - Process multiple instructions in parallel
  - Digital circuits are inherently parallel systems (individual bits get operated on in parallel)
  - Have more ALUs to carry out parallel additions
  - Finally, speculation is the ultimate solution for extracting parallelism: main idea is to do multiple possible operations in parallel without knowing which one is correct while correctness detection proceeds in parallel (e.g. cache lookup)

this leads to extra instructions that the CPU also executes, and which may be thrown away.

MAINAK CS422 32

Trace Driven → no. of instrs = count of trace length

CPI =

Counters (4 only): (i) no. of instrs executed
(ii) cycles spend for each instr
(iii) no. of types of instrs executed
} counters count the CPU events

Static : if → 90% else → 10%. the compiler can encode this
for (i=1 → 100){ ⇒ 100th time branch does not go back, compiler knows.
}

To keep in mind

- Make the common case fast
- Smaller is faster
- 90% of time is spent in 10% of code

Research goals

- Recall
  - Computer architecture is about designing online algorithms to exploit application behavior (static or run-time)
- Computer architecture innovations are always driven by the needs of emerging software
  - Guided by thorough study of benchmark applications and sometimes the underlying algorithms
  - Key questions that an architect tries to answer, once application bottlenecks (in terms of performance, power consumption, communication traffic, or hardware complexity) on current architecture are understood
    - With what minimal hardware changes can I eliminate the bottlenecks? What property/behavior can I exploit?
    - Can the compiler help in simplifying the solution?
    - Can I get any help from the operating system?

which apps will become imp in 5-6 y.

Research practices

Queuing theory →

- Empirical in nature
  - Any theoretical model amenable to mathematical treatment needs to make unreasonable assumptions as the underlying interactions are fairly complicated
- Experiments are done on a detailed carefully designed processor simulator (written in HLL)
  - Benchmark applications are run on the simulator to gather execution statistics
  - Often simulating a full application takes unreasonably large amount of time; representative regions of the application are selected for simulation
    - One possible selection procedure: SimPoint (L_2 dist = dist measure)
    - Representative regions should be large enough to capture different phase behavior of an application

Research practices

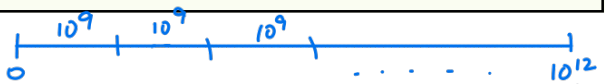
- Possible structure of a sequential simulator

```
main () {
  // Initialize
  while (!exit) {  // exit flag is set by syscall emulation layer
    for each core {
      retire ();
      execute ();
      read_register ();
      issue ();
      decode_and_rename ();
      fetch();
    }
    cycle++;
    Adjust resources and compute enable/stall signals for next cycle
  }
  // Cleanup
}
```

simulator's way to model all sys calls of linux kernel so don't depend on OS

} queues in b/w

if queue is full

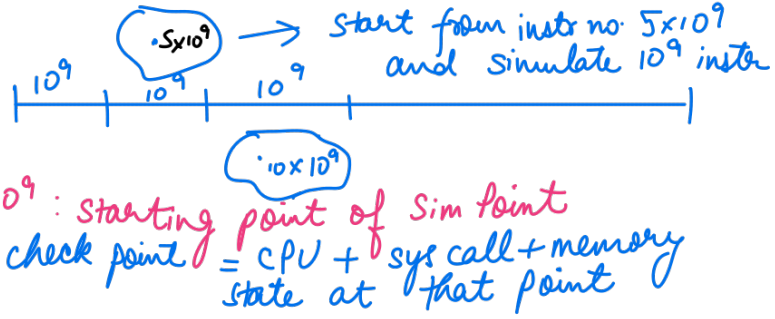
Trace = 

SimPoint : clustering based on similarity

Program ≡ N BB. (Total no. of BBs)

each region = N-dim vector : each number tells how many times that basic block is executed. then we do clustering. So for any BB does not appear in a region → 0 for that BB in that region N-dim vector

Assume : 6 stage pipeline and executed for each core in reverse order so as to model pipeline
first cycle → fetch
second → decode + fetch



$5x10^9$: starting point of Sim Point + check point = CPU + sys call + memory state at that point

Research practices

- Possible structure of a parallel simulator
  - Each pipe stage of a core is implemented as a thread
  - A clock signal (two-phase) synchronizes the threads
  - Typical structure of a fetch thread:

```
fetch () {
  // Initialize
  while (1) {  // exit is called by main()
    AWAIT_PHI0;   // Wait for positive edge of clock
                  // Advanced by an event/thread scheduler
    // Sample all necessary signals e.g., PC, branch mispredict, any stall signal, etc.
    AWAIT_PHI1;   // Wait for negative edge of clock
    if (!stalled) {// Fetch instruction(s), prepare decode packet, and update PC
   based on all sampled PC modifying signals }
    // Update stats, etc.   PC ← PC+4
  }
}
```

MAINAK CS422 37

pipeline = 6 thread
4 core ⇒ 24 threaded model

similar to FF : sample all variables on a posedge

no work on pos edge in simulation of software so that next stage does not see the just prev stage and race is avoided

Basic block : block of instruc → a block of instrs that ends with a branch instr } executed completely then exit to other basic block