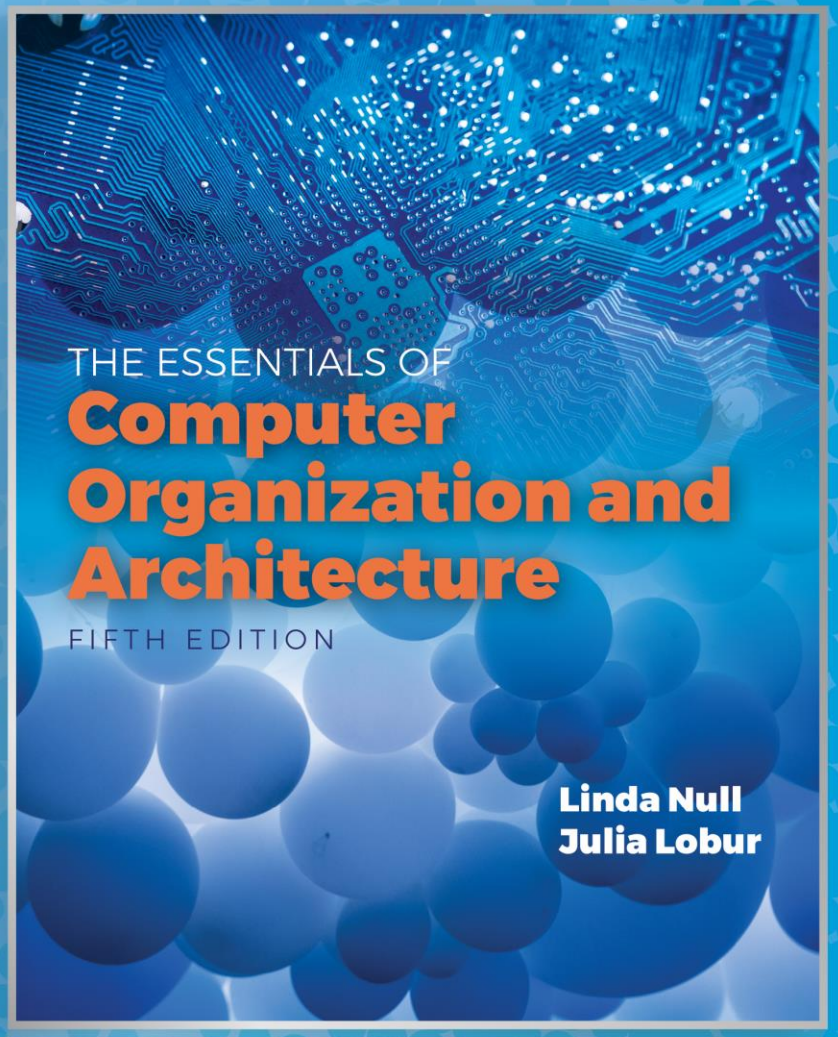
This is the fourth lecture of Chapter 11

Chapter 11

Performance Measurement and Analysis (D)



Quick review of last lecture

- Benchmarking
  - Clock rate, MIPS, and FLOPS
  - Synthetic Benchmarks: Whetstone, Linpack, and Dhrystone
  - Standard Performance Evaluation Corporation (SPEC) benchmarks
  - Transaction Performance Council (TPC) benchmarks
  - System simulation

11.5 CPU Performance Optimization (1 of 15)

- CPU optimization includes many of the topics that have been covered in preceding chapters.
  - CPU optimization includes topics such as pipelining, parallel execution units, and integrated floating-point units.
- We have not yet explored two important CPU optimization topics: Branch optimization and user code optimization.
- Both of these can affect performance in dramatic ways.

11.5 CPU Performance Optimization (2 of 15): Branch Optimization

- We know that pipelines offer significant execution speedup when the pipeline is kept full.
- Conditional branch instructions are a type of pipeline hazard that can result in flushing the pipeline.
  - Other hazards are include conflicts, data dependencies, and memory access delays.
- Delayed branching offers one way of dealing with branch hazards.
- With delayed branching, one or more instructions following a conditional branch are sent down the pipeline regardless of the outcome of the statement.

11.5 CPU Performance Optimization (3 of 15)

- The responsibility for setting up delayed branching most often rests with the compiler.
- It can choose the instruction to place in the delay slot in a number of ways.
- The first choice is a useful instruction that executes regardless of whether the branch occurs.
- Other possibilities include instructions that execute if the branch occurs, but do no harm if the branch does not occur.
- Delayed branching has the advantage of low hardware cost.

11.5 CPU Performance Optimization (4 of 15)

- Branch prediction is another approach to minimizing branch penalties.
- Branch prediction tries to avoid pipeline stalls by guessing the next instruction in the instruction stream.
  - This is called speculative execution.
- Branch prediction techniques vary according to the type of branching. If/then/else, loop control, and subroutine branching all have different execution profiles.

11.5 CPU Performance Optimization (5 of 15)

- There are various ways in which a prediction can be made:
  - Fixed predictions do not change over time.
  - True predictions result in the branch being always taken or never taken.
  - Dynamic prediction uses historical information about the branch and its outcomes.
  - Static prediction does not use any history.

11.5 CPU Performance Optimization (6 of 15)

- When fixed prediction assumes that a branch is not taken, the normal sequential path of the program is taken.
- However, processing is done in parallel in case the branch occurs.
- If the prediction is correct, the preprocessing information is deleted.
- If the prediction is incorrect, the speculative processing is deleted and the preprocessing information is used to continue on the correct path.

11.5 CPU Performance Optimization (7 of 15)

- When fixed prediction assumes that a branch is always taken, state information is saved before the speculative processing begins.
- If the prediction is correct, the saved information is deleted.
- If the prediction is incorrect, the speculative processing is deleted and the saved information is restored allowing execution to continue on the correct path.

11.5 CPU Performance Optimization (8 of 15)

- Dynamic prediction employs a high-speed branch prediction buffer to combine an instruction with its history.
- The buffer is indexed by the lower portion of the address of the branch instruction that also contains extra bits indicating whether the branch was recently taken.
  - One-bit dynamic prediction uses a single bit to indicate whether the last occurrence of the branch was taken.
  - Two-bit branch prediction retains the history of the previous to occurrences of the branch along with a probability of the branch being taken.

11.5 CPU Performance Optimization (9 of 15)

- The earliest branch prediction implementations used static branch prediction.
- Most newer processors (including the Pentium, PowerPC, UltraSparc, and Motorola 68060) use two-bit dynamic branch prediction.
- Some superscalar architectures include branch prediction as a user option.
- Many systems implement branch prediction in specialized circuits for maximum throughput.

11.5 CPU Performance Optimization (10 of 15): Use of Good Algorithms

- The best hardware and compilers will never equal the abilities of a human being who has mastered the science of effective algorithm and coding design.
- People can see an algorithm in the context of the machine it will run on.
  - For example a good programmer will access a stored column-major array in column-major order.
- We end this section by offering some tips to help you achieve optimal program performance.

11.5 CPU Performance Optimization (11 of 15)

- Operation counting can enhance program performance.
- With this method, you count the number of instruction types executed in a loop then determine the number of machine cycles for each instruction.
- The idea is to provide the best mix of instruction types for a particular architecture.
- Nested loops provide a number of interesting optimization opportunities.

11.5 CPU Performance Optimization (12 of 15)

- *Loop unrolling* is the process of expanding a loop so that each new iteration contains several of the original operations, thus performing more computations per loop iteration.
- For example:

```
for (i = 1; i <= 30; i++)
   a[i] = a[i] + b[i] * c;
```

- becomes

```
for (i = 1; i <= 30; i+=3)
  { a[i] = a[i] + b[i] * c;
    a[i+1] = a[i+1] + b[i+1] * c;
    a[i+2] = a[i+2] + b[i+2] * c; }
```

11.5 CPU Performance Optimization (13 of 15)

- *Loop fusion* combines loops that use the same data elements, possibly improving cache performance. For example:

```
for (i = 0; i < N; i++)
     C[i] = A[i] + B[i];
for (i = 0; i < N; i++)
     D[i] = E[i] + C[i];
```

- becomes

```
for (i = 0; i < N; i++)
  { C[i] = A[i] + B[i];
    D[i] = E[i] + C[i]; }
```

11.5 CPU Performance Optimization (14 of 15)

- *Loop fission* splits large loops into smaller ones to reduce data dependencies and resource conflicts.
- A loop fission technique known as loop peeling removes the beginning and ending loop statements.

For example:

```
for (i = 1; i < N+1; i++)
{ if (i==1)
      A[i] = 0;
  else if (i == N)
      A[i] = N;
  else A[i] = A[i] + 8; }
```

Becomes:

```
A[1] = 0;
for (i = 2; i < N; i++)
  A[i] = A[i] + 8;
A[N] = N;
```

11.5 CPU Performance Optimization (15 of 15)

- The text lists a number of rules of thumb for getting the most out of program performance.
- Optimization efforts pay the biggest dividends when they are applied to code segments that are executed the most frequently.
- In short, try to make the common cases fast.

11.6 Disk Performance (1 of 23)
1. Understanding the problem

- Optimal disk performance is critical to system throughput.
- Disk drives are the slowest memory component, with the fastest access times one million times longer than main memory access times.
- A slow disk system can choke transaction processing and drag down the performance of all programs when virtual memory paging is involved.
- Low CPU utilization can actually indicate a problem in the I/O subsystem, because the CPU spends more time waiting than running.