CSC2/458 Parallel and Distributed Systems
Automatic Parallelization in Hardware

Sreepathi Pai

January 25, 2018

URCS

Outline

Pipelining

Superscalar and Out-of-order Execution

Speculation

Outline

Primes

```
for(int i = 2; i * i <= num; i++) {
  if(num % i ==  0)  {
    is_prime = 0;
    divisor = i;
    break;
  }
}
```

Primes – Assembly

```
        movl    $2, -8(%rbp)
        jmp     .L3
.L6:
        movl    -4(%rbp), %eax
        cltd
        idivl   -8(%rbp)
        movl    %edx, %eax
        testl   %eax, %eax
        jne     .L4
        movl    $0, -16(%rbp)
        movl    -8(%rbp), %eax
        movl    %eax, -12(%rbp)
        jmp     .L5
.L4:
        addl    $1, -8(%rbp)
.L3:
        movl    -8(%rbp), %eax
        imull   -8(%rbp), %eax
        cmpl    -4(%rbp), %eax
        jle     .L6
.L5:
        cmpl    $0, -16(%rbp)
        je      .L7
```

gcc -S

Primes – Machine code

```
4006b0:       c7 45 f8 02 00 00 00    movl   $0x2,-0x8(%rbp)
4006b7:       eb 20                   jmp    4006d9
4006b9:       8b 45 fc                mov    -0x4(%rbp),%eax
4006bc:       99                      cltd
4006bd:       f7 7d f8                idivl  -0x8(%rbp)
4006c0:       89 d0                   mov    %edx,%eax
4006c2:       85 c0                   test   %eax,%eax
4006c4:       75 0f                   jne    4006d5
4006c6:       c7 45 f0 00 00 00 00    movl   $0x0,-0x10(%rbp)
4006cd:       8b 45 f8                mov    -0x8(%rbp),%eax
4006d0:       89 45 f4                mov    %eax,-0xc(%rbp)
4006d3:       eb 10                   jmp    4006e5
4006d5:       83 45 f8 01             addl   $0x1,-0x8(%rbp)
4006d9:       8b 45 f8                mov    -0x8(%rbp),%eax
4006dc:       0f af 45 f8             imul   -0x8(%rbp),%eax
4006e0:       3b 45 fc                cmp    -0x4(%rbp),%eax
4006e3:       7e d4                   jle    4006b9
4006e5:       83 7d f0 00             cmpl   $0x0,-0x10(%rbp)
4006e9:       74 16                   je     400701
```

objdump -d

Primes – What the machine sees

```
4006b0:

c7 45 f8 02 00 00 00 eb 20 8b 45 fc
99 f7 7d f8 89 d0 85 c0 75 0f c7 45
f0 00 00 00 00 8b 45 f8 89 45 f4 eb
10 83 45 f8 01 8b 45 f8 0f af 45 f8
3b 45 fc 7e d4 83 7d f0 00 74 16
```

Executing an instruction

- **Fetch** instruction at Program Counter
- **Decode** fetched instruction
- **Execute** decoded instruction
  - Dispatch to functional units
  - Functional units include ALU, Floating Point, etc.
- **Memory** access for data loads/stores
- **Writeback** results of execution to registers

Assuming each task above takes 1 cycle, how many cycles will a non-memory instruction take?

Instruction Execution Pipeline

See “animation” on board.

Pipelining Performance

- What is latency of executing a single instruction?
- What is the latency of executing all instructions?
- What is the throughput of instruction execution once first instruction has finished executing?

Data “Hazards”

```
1:      movl    -4(%rbp), %eax
2:      cltd
3:      idivl   -8(%rbp)
4:      movl    %edx, %eax
```

- Instructions 1 and 3 are ”variable” latency
  - May take more than once cycle to execute
- Instruction 2 takes one cycle and writes results to EAX, EDX
- How to pipeline instructions 2 and 4?

Solving Data "Hazards"

- Bubble
  - Don't issue the instruction
- Bypassing
  - Forward results to previous stages in pipeline

Design Issues

- When is pipelining useful?
- What characteristics should the pipeline stages have?
- How would you implement pipelines in software?

Outline

Superscalar Execution

- Superscalar: ability to fetch, decode, execute and writeback more than one instruction at the same time
- Conceptually simple
  - More pipelines
  - More ALUs
- Central question: Which instructions should be executed together?
  - Which instructions allow parallel execution?

Dependences

A dependence exists between two instructions if they both access the same register or memory location, and if one of the accesses is a write.

Types of Dependences

- True dependence (or Read after Write)

```
R1 = R2 + R3
R4 = R1 + 1
```

- Anti-dependence (or Write after Read)

```
R1 = R2 + R3
R3 = R4 * R5
```

- Output-dependence (or Write after Write)

```
R1 = R2 + R3
R1 = R4 * R5
```

Here Rx indicates a register.

Algorithm to find and execute multiple instructions

- Fetch multiple instructions
- Decode multiple instructions
- Find instructions that are not dependent on earlier instructions
    - Earlier in *program order*
- Execute them
- Write back results

Finding Independent Instructions

Instruction	Reads	Writes
movl $2, -8(%rbp)	%rbp	MEM
jmp .L3	-	%eip
.L6: movl -4(%rbp), %eax	%rbp, MEM	%eax
cltd	%eax	%eax, %edx
idivl -8(%rbp)	%rbp, MEM, %eax, %edx	%eax, %edx
movl %edx, %eax	%edx	%eax
testl %eax, %eax	%eax	%eflags
jne .L4	-	%eip?
movl $0, -16(%rbp)	%rbp	MEM

Issues with superscalar execution

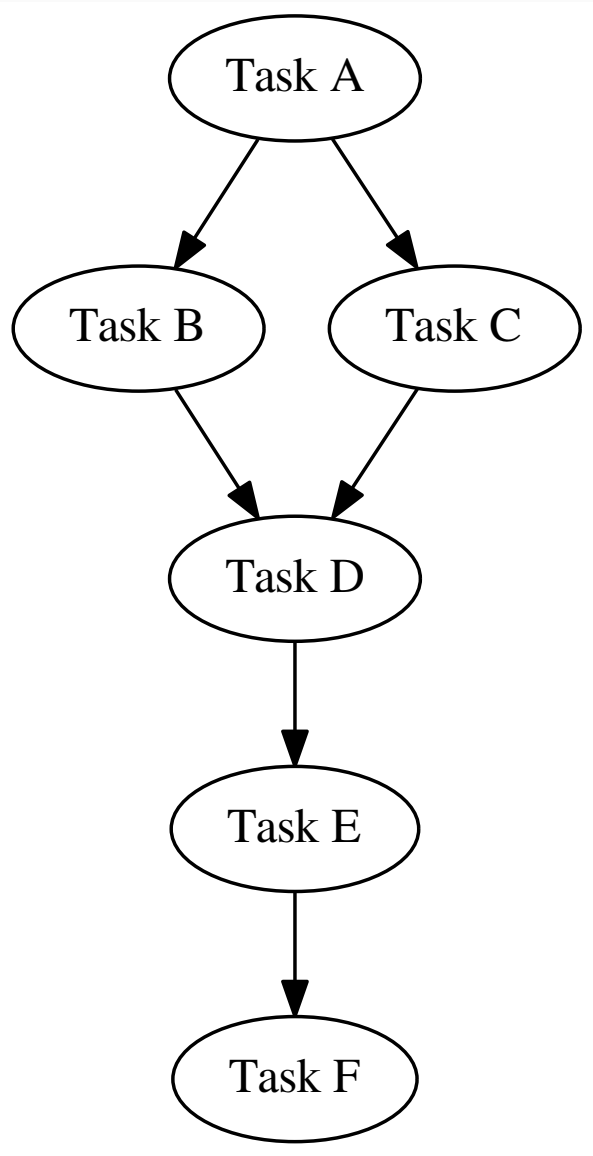
- Are there independent instructions?
  - Data dependence
- How to handle branches?
  - I.e. how to fetch "beyond" a branch?
  - Control dependence
- How to handle machine limitations?
  - E.g. 8 independent ADD instructions, but only 4 ALUs
  - "Structural Hazard"

Software Implementations

How to implement out-of-order execution of tasks in software?

Dependence Graphs

- Node represents a graph
- Edge represents dependence
- Algorithm to execute
  - STEP 1: Find independent tasks (tasks with no incoming edges)
  - STEP 2: Execute these tasks
  - STEP 3: Remove edges from executed tasks, and repeat from STEP 1 until no tasks remain



Outline

Basic Blocks

- Unit of code
- Single entry and single exit

```
.L6:
        movl    -4(%rbp), %eax
        cltd
        idivl   -8(%rbp)
        movl    %edx, %eax
        testl   %eax, %eax
        jne     .L4
```

Parallelizing Basic Blocks

- All instructions in basic block can be executed in parallel
  - if independent
- Only data dependences between instructions in the same basic block

- How big are most basic blocks?
- Alternatively, how often do branch instructions occur?

Predicting Branches

- Different types of branch instructions
  - Unconditional
  - Conditional
  - Returns
- Can we predict PC of instruction after branch?
  - Can immediately start executing instructions without waiting for branch

What happens if we're wrong?

- All instructions dependent on predicted branch are *speculative*
- When branch is resolved:
  - if predication was correct: commit/writeback speculative instructions
  - if incorrect: throw away all speculative instructions

Speculation in software?

How do we do speculation in software?

How will a processor parallelize this?

For more on processor parallelization, take Advanced Computer Architecture.

```
for(i = 0; i < A; i++) {
   sum1 = sum1 + i;
}

for(i = 0; i < B; i++) {
   sum2 = sum2 + i;
}
```