Instruction Scheduling

Ph.D Candidate: Curtis D'Alves

Supervisors: Dr. Wolfram Kahl, Dr. Christopher Anand

Table of Contents

Instruction Scheduling

Problem: Given a set of instructions and dependencies, designate an order (find a *schedule*) satisfying the dependencies and optimizing performance

Instruction Scheduling

Problem: Given a set of instructions and dependencies, designate an order (find a *schedule*) satisfying the dependencies and optimizing performance

Known **NP-Complete** problem, practically solved by

- Heuristics
- Approximation Algorithms

Computational Complexity

```
for i in 1..n
   x[i] = x[i] + 1


   O(n)
```

```
for i in 1..n
   for j in i..n
        x[i] += x[j]


     O(n^2)
```

- Bounds how a algorithm **scales**
- Algorithms with a polynomial bound (i.e **P**) are **do-able**
- Algorithms with a exponential bound (i.e **NP**) are take too long as their input grows

NP-Complete

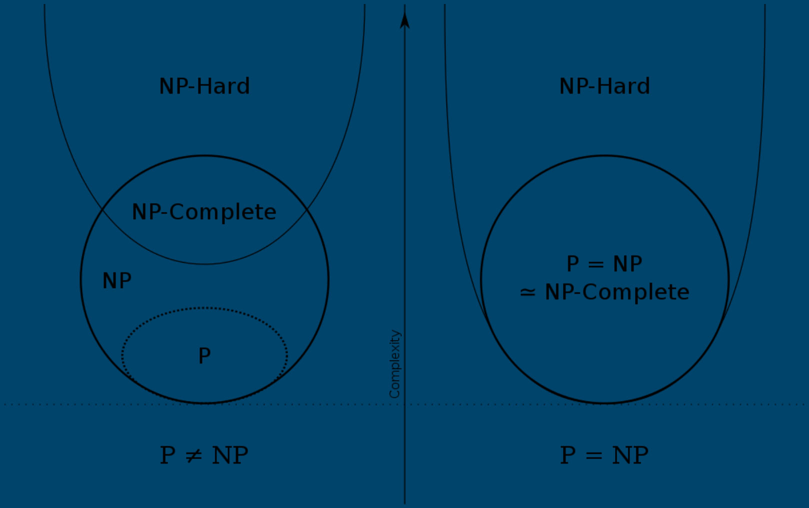


Figure: Reference https://en.wikipedia.org/wiki/NP-completeness

Heuristics

Don't find the optimal solution, but rather a **good-enough** solution

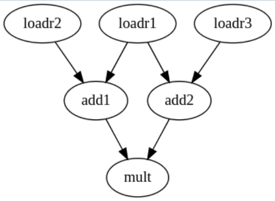
Often seeks to **solve a simpler problem** then the actual task at hand

Heuristics for scheduling are often just **rules of thumb**

Example: Dependency Graph (DAG)

```
load r1 0xFFFF
load r2 0xFFF1
load r3 0xFFF2
add r4 r2 r1
add r5 r3 r1
mult r6 r4 r5
```

```
(r1 + r2) * (r1 + r3)
```



Instruction Schedules

```
load r1 0xFFFF      load r1 0xFFFF
load r2 0xFFF1      load r2 0xFFF1
load r3 0xFFF2      add r3 r2 r1
add r4 r2 r1        load r2 0xFFF2
add r5 r3 r1        add r4 r2 r1
mult r6 r4 r5       mult r5 r4 r3
```

- Both of the above orders (i.e schedules) are **valid** (i.e don't break dependencies)
- What's the difference:
  - **peformance**
  - **number of registers used**

Complication: Branching

```
load r1 0xFFFF
load r2 0xFFF1
load r3 0xFFF2
add r4 r2 r1 <---
add r5 r3 r1    |
mult r6 r4 r5   |
branch 0x0003 ---
add r2 r4 r1
```

Control flow (like **loops** and **conditionals**) complicate scheduling

Types of Scheduling

- **Basic Block:** break code into blocks within branches **(most commonly performed scheduling)**

Types of Scheduling

- **Basic Block:** break code into blocks within branches **(most commonly performed scheduling)**

- **Global Scheduling:** schedule across basic block boundaries

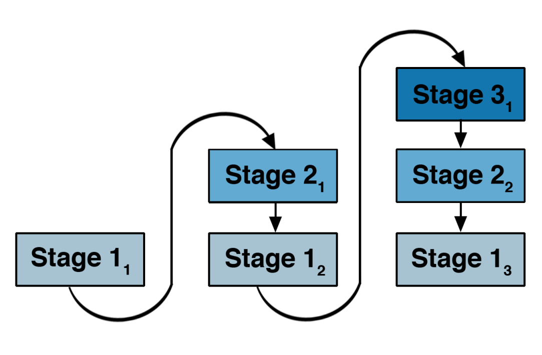
Types of Scheduling

- **Basic Block:** break code into blocks within branches **(most commonly performed scheduling)**
- **Global Scheduling:** schedule across basic block boundaries
- **Modulo Scheduling:** an algorithm to increase pipelining of loops by interleaving different iterations

Types of Scheduling

- **Basic Block:** break code into blocks within branches **(most commonly performed scheduling)**
- **Global Scheduling:** schedule across basic block boundaries
- **Modulo Scheduling:** an algorithm to increase pipelining of loops by interleaving different iterations
- **Trace Scheduling:** tries to optimize control flow by predicting routes taken on branches

Modulo Scheduling: Staged Loop



When performing **modulo scheduling**, a basic block of a loop can be broken into stages and the loop can be **rolling** to interleave stages between iterations

Table of Contents

Classic RISC Pipeline

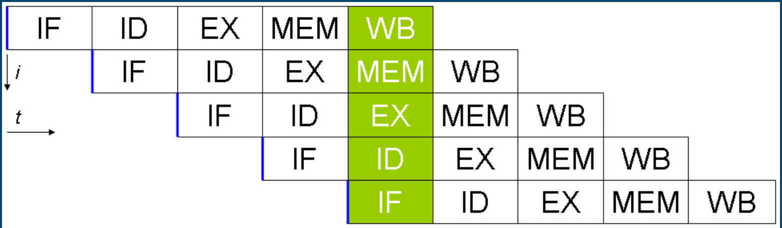


Figure: Example Pipeline

SuperScaler Pipelining

IF	ID	EX	MEM	WB				
IF	ID	EX	MEM	WB				
	IF	ID	EX	MEM	WB			
	IF	ID	EX	MEM	WB			
		IF	ID	EX	MEM	WB		
		IF	ID	EX	MEM	WB		
			IF	ID	EX	MEM	WB	
			IF	ID	EX	MEM	WB	
				IF	ID	EX	MEM	WB
				IF	ID	EX	MEM	WB

Figure: Example SuperScaler Pipeline

Out-of-order Execution

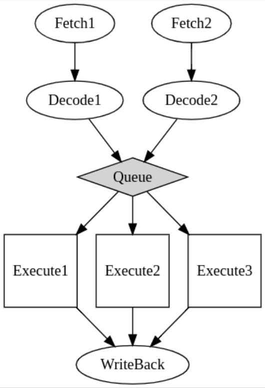


Figure: Example SuperScaler Pipeline

Out-of-order Execution

1. Instruction fetch
2. Dispatch to instruction queue
3. Instruction waits until its input is available, then allowed to leave queue (in whatever order)
4. Instruction is executed
5. Results are queued
6. Only after all older instructions have their results written back to registers, the instruction's result is written back to registers

Hazards

- **Data Hazards**
  - read after write (RAW)
  - write after read (WAR)
  - write after write (WAW)

Hazards

- **Data Hazards**
  - read after write (RAW)
  - write after read (WAR)
  - write after write (WAW)

- **Structural Hazards** - occurs when an aspect of hardware is accessed at the same time

Hazards

- **Data Hazards**
  - read after write (RAW)
  - write after read (WAR)
  - write after write (WAW)
- **Structural Hazards** - occurs when an aspect of hardware is accessed at the same time
- **Control Hazards** - caused by branching, next instruction unknown

Pipeline Stalls / Bubbles

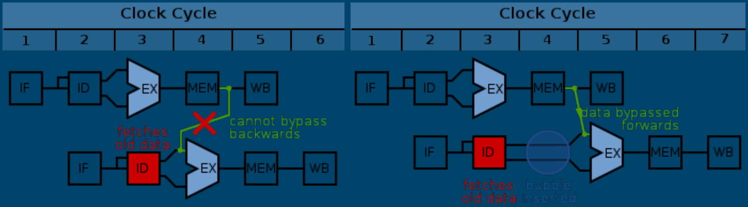


Figure: Pipeline Stall

A **Ideal Schedule** contains NO bubbles (often not possible)

Table of Contents

Register Allocation

Problem: Given a schedule, assign registers keeping in mind

- limited # of registers
- can't rewrite a register until consumed by dependent instructions

Once again, known **NP-Complete** problem. Practically solved by using non-optimal **Graph Coloring** problems

Graph Coloring

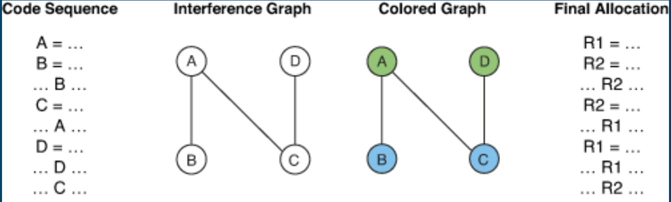


Figure: Register Allocation via Graph Coloring

Find a **k-Coloring** for the interference graph, where $k = \#\textbf{Registers}$

Spilling

- What if a **k-Coloring** can't be found? Must **Spill** memory

Spilling

- What if a **k-Coloring** can't be found? Must **Spill** memory
- Simply insert new **Load / Store** instructions as needed

Spilling

- What if a **k-Coloring** can't be found? Must **Spill** memory
- Simply insert new **Load / Store** instructions as needed
- Potentially **creates new bubbles** in the pipeline, need to re-perform scheduling

Spilling

- What if a **k-Coloring** can't be found? Must **Spill** memory
- Simply insert new **Load / Store** instructions as needed
- Potentially **creates new bubbles** in the pipeline, need to re-perform scheduling
- An **Ideal Schedule** has no spilling

Table of Contents

List Scheduling

Simple heuristic. Choose a **prioritized topological order** that

- Respects the edges in the data-dependence graph (**topological**
- Heuristic choice among options, e.g pick first the node with the longest path extending from that node **prioritized**

Most commonly used method for scheduling. Efficient but yields far less than optimal schedules

Issues with List Scheduling

- Many factors to consider when constructing a schedule (everything listed in this presentation and more!)

Issues with List Scheduling

- Many factors to consider when constructing a schedule (everything listed in this presentation and more!)

- Difficult (or more accurately impossible!) to consider all these aspects into a single choice heuristic

Issues with List Scheduling

- Many factors to consider when constructing a schedule (everything listed in this presentation and more!)
- Difficult (or more accurately impossible!) to consider all these aspects into a single choice heuristic
- Combinations of heuristics can be used, and multiple iterations performed, but each will usually undo the work of the other

Table of Contents

My Research

- Scheduling of pre-compiled binaries

My Research

- Scheduling of pre-compiled binaries
- Since pre-compiled, my algorithm can afford to be **less efficient** and seek **near-optimal performance**

My Research

- Scheduling of pre-compiled binaries
- Since pre-compiled, my algorithm can afford to be **less efficient** and seek **near-optimal performance**
- Uses a **Continuous Optimization Model**

Continuous Optimization

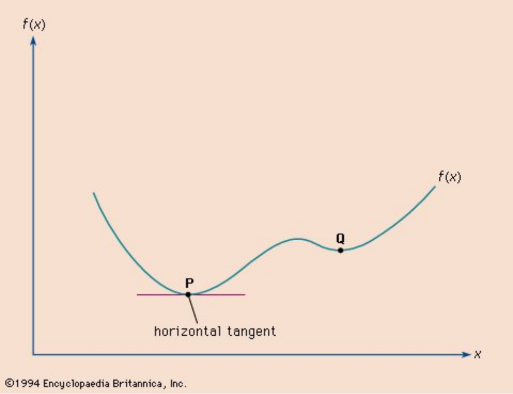


Figure: Ref https://www.britannica.com/topic/nonlinear-programming

Find a minumum of a function $f(x)$

Relaxed Continuous Optimization based Scheduling

Per Instruction i, perform a relaxation of scheduled position to dispatch and completion times t_i,b_i

Objective Variables	t_i, b_i, f_i :	$\mathbb{R}$
Constants	II :	$\mathbb{R}$
Indicator Function	$\mathbb{IN}$:	$\mathbb{R} \rightarrow \mathbb{R}$
	t_i :	dispatch time
	b_i :	completion time
	f_i :	FIFO use $0 \leq f_i \leq 1$
	II :	iteration interval $\dfrac{\#instructions}{dispatches/cycle}$

Relaxed Continuous Optimization based Scheduling

Hard Constraints
$$t_i + \epsilon \le t_j \qquad \forall i, j \cdot i \rightarrow j \tag{1}$$
$$0 \le t_i \le b_i \le \#\text{stages} \cdot \mathrm{II} \tag{2}$$
$$b_i + \epsilon \le t_i + \mathrm{II} \tag{3}$$

Objective Function
$$\min \sum_i (b_i - t_i + f_i) + \text{Penalties} \tag{4}$$

Key Idea: Encode choice heuristics as penalties, adjust preference by between heuristics scaling

Relaxed Continuous Optimization based Scheduling

Other Key Idea: Need to construct penalty to prevent **Spilling**. Need to prevent clobbering of certain types of instructions

Solution: Indicator function to detect penalize clobbering

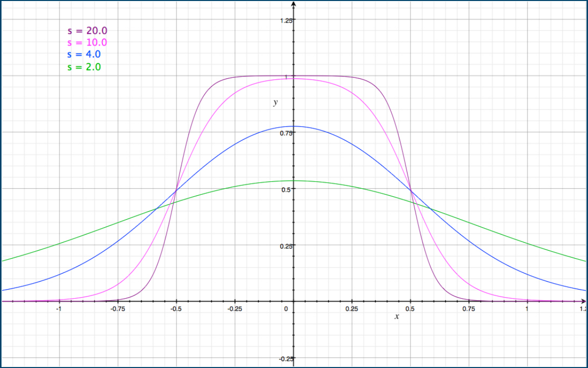


Figure: Altered Sigmoid Indicator Function

Table of Contents

Questions?