Lecture 14: Scheduling
COS 598C – Advanced Compilers

Where are we?

- Code Generation
- Analysis
- Optimization
- Code Generation

Matt Bridges
Prof. David August
Department of Computer Science
Princeton University

Code Generation

- Map optimized “machine-independent” assembly to final assembly code
- Input code
  - Classical optimizations
  - ILP optimizations
  - Formed regions, applied if-conversion
- Virtual → physical binding
  - 2 big steps
  - 1. Scheduling
    - Determine when every operation executions
    - Create MultiOps
  - 2. Register allocation
    - Map virtual → physical registers
    - Spill to memory if necessary

What Do We Need to Schedule Operations?

- Information about the processor
  - Number of resources
  - Which resources are used by each operation
  - Operation latencies
  - Operan encoding limitations
  - For example:
    - 2 issue slots, 1 memory port, 1 adder/multiplier
    - load = 2 cycles, add = 1 cycle, mpy = 3 cycles; all fully pipelined
    - Each operand can be register or 6 bit signed literal
- Ordering constraints amongst operations
  - What order defines correct program execution?
  - Need a precedence graph – flow, anti, output deps
    - What about memory deps? control deps? Delay slots?
How Do We Schedule?

- When is it legal to schedule an instruction?
  - Correct execution is maintained
  - Resources not oversubscribed
- Given multiple operations that can be scheduled, how do you pick the best one?
  - How do you know it is the best one?
    - What about a good guess?
    - Does it matter, just pick one at random?
  - Are decisions final?, or is this an iterative process?
- How do we keep track of resources that are busy/free
  - Need a reservation table
  - Matrix (resources x time)

More Stuff to Worry About

- Model more resources
  - Register ports, output busses
  - Non-pipelined resources
- Dependent memory operations
- Multiple clusters
  - Cluster = group of FUs connected to a set of register files such that an FU in a cluster has immediate access to any value produced within the cluster
  - Multicluster = Processor with 2 or more clusters, clusters often interconnected by several low-bandwidth busses
    - Bottom line = Non-uniform access latency to operands
- Scheduler has to be fast
  - NP complete problem
  - So, need a heuristic strategy
  - What is better to do first, scheduling or register allocation?

Compiler Code Generation – 2nd try

- Map optimized “machine-independent” assembly to final assembly code
- Virtual → physical binding
  - Cannot do this all at once, too many decisions!?
  - Do slowly
  - Each step refines the binding by restricting previous choices
- Schedule both before and after register allocation
  - Initial scheduling is free of real processor register constraints
  - 2nd phase required due to spill code

Why Not Schedule After Register Allocation?

- virtual registers
  - r1 = load(r10)
  - r2 = load(r11)
  - r3 = r1 + 4
  - r4 = r1 – r12
  - r5 = r2 + r4
  - r6 = r5 + r3
  - r7 = load(r13)
  - r8 = r7 * 23
  - store (r8, r6)

- physical registers
  - R1 = load(R1)
  - R2 = load(R2)
  - R5 = R1 + 4
  - R1 = R1 – R3
  - R2 = R2 + R1
  - R2 = R2 + R5
  - R5 = load(R4)
  - R5 = R5 * 23
  - store (R5, R2)

Too many artificial ordering constraints!!!!
The 6 Step Program

1. Code selection, Literal handling
   - Semantic operations to generic operations
   - How to realize a specific function on this machine
   - Complement all bits → xor with –1
   - Can literal be encoded in operation, if not need load/move

2. Prepass operation binding
   - Partially bind operation to subset of resources
   - Resources are access equivalent
     - Any choice is equal to any other choice
   - Multi-cluster machine – bind operation to a cluster

3. Scheduling
   - What time the operation will be executed
   - What execution resources will be used
     - Chooses alternative

4. Register allocation
   - Assign physical registers
   - Bind each access-equivalent register to a specific physical register
   - Introduce additional code to spill registers to memory

5. Postpass scheduling
   - A second pass of scheduling to handle spill code
   - Resource assignments from first pass are ignored
   - But, registers are physical, so less code motion freedom

6. Code emission
   - Convert “fully qualified” operations into real assembly
   - A translator basically
   - Assembler converts this assembly to machine code

   Focus for now on 3, 4, 5, assume 1, 2, 6 are not needed

Machine Information

- Each step of code generation requires knowledge of the machine
  - Hard code it? – used to be common practice
  - Retargetability, then cannot

- What does the code generator need to know about the target processor?
  - Structural information?
    - No
  - For each opcode
    - What registers can be accessed as each of its operands
  - Other operand encoding limitations
  - Operation latencies
    - Read inputs, write outputs
  - Resources utilized
    - Which ones, when

Machine Description (mdes)

- Terminology
  - Generic opcode
    - Virtual opcode, machine supports k versions of it
    - ADD_W
  - Architecture opcode or unit specific opcode or sched opcode
    - Specific assembly operation of the processor
    - ADD_W.0 = add on function unit 0

- Each unit specific opcode has 3 properties
  - IO format
  - Latency
  - Resource usage
**IO Format**

- Registers, register files
  - Number, width, static or rotating
  - Read-only (hardwired 0) or read-write
- Operation
  - Number of source/dests
  - Predicated or not
  - For each source/dest/pred
    - What register file(s) can be read/written
    - Literals, if so, how big

  Multicluster machine example:
  - `ADD_W.0`: `gpr1, gpr1 : gpr1`
  - `ADD_W_L.0`: `gpr1, lit6 : gpr1`
  - `ADD_W.1`: `gpr2, gpr2 : gpr2`

**Latency Information**

- Multiply takes 3 cycles
  - No, not that simple!!!
- Differential input/output latencies
  - Earliest read latency for each source operand
  - Latest read latency for each source operand
  - Earliest write latency for each destination operand
  - Latest write latency for each destination operand
- Why all this?
  - Unexpected events may make operands arrive late or be produced early

- Compound op: part may finish early or start late
- Instruction re-execution by
  - Exception handlers
  - Interrupt handlers
- Ex: `mpyadd(d1, d2, s1, s2, s3)`
  - `d1 = s1 * s2, d2 = d1 + s3`

**Memory Serialization Latency**

- Ensuring the proper ordering of dependent memory operations
- Not the memory latency
  - But, point in the memory pipeline where 2 ops are guaranteed to be processed in sequential order
- Page fault – memory op is re-executed, so need
  - Earliest mem serialization latency
  - Latest mem serialization latency
- Remember
  - Compiler will use this, so any 2 memory ops that cannot be proven independent, must be separated by mem serialization latency.

**Branch Latency**

- Time relative to the initiation time of a branch at which the target of the branch is initiated
- What about branch prediction?
  - Can reduce branch latency
  - But, may not make it 1
- We will assume branch latency is 1 for this class (ie no delay slots!)

  Example:
  - 0: branch
  - 1: `xxx`
  - 2: `yyy`
  - 3: `target`
  - `delay slots = k – 1 (2)`
  - `branch latency = k (3)`
  - Note `xxx` and `yyy` are multiOps
A machine resource is any aspect of the target processor for which over-subscription is possible if not explicitly managed by the compiler.

- Scheduler must pick conflict free combinations

3 kinds of machine resources

- **Hardware resources** are hardware entities that would be occupied or used during the execution of an opcode
  - Integer ALUs, pipeline stages, register ports, busses, etc.

- **Abstract resources** are conceptual entities that are used to model operation conflicts or sharing constraints that do not directly correspond to any hardware resource
  - Abstract ALUs, instruction fields, etc.

- **Counted resources** are identical resources such that k are required to do something
  - Any 2 input busses

---

**Resources**

**Reservation Tables**

For each opcode, the resources used at each cycle relative to its initiation time are specified in the form of a table.

Res1, Res2 are abstract resources to model issue constraints.

---

**Data Dependences**

- **Data dependences**
  - If 2 operations access the same register, they are dependent
  - However, only keep dependences to most recent producer/consumer as other edges are redundant
  - Types of data dependences
More Dependences

- Memory dependences
  - Similar as register, but through memory
  - Memory dependences may be certain or maybe
- Control dependences
  - We discussed this earlier
  - Branch determines whether an operation is executed or not
  - Operation must execute after/before a branch
  - Note, control flow (C0) is not a dependence

```
Mem-flow
store (r1, r2)
    r3 = load(r1)

Mem-output
store (r1, r2)
    if (r1 != 0)
        r2 = load(r1)

Mem-anti
store (r1, r3)
    store (r1, r3)
    r2 = load(r1)

Control (C1)
```

 Dependence Graph

- Represent dependences between operations in a block via a DAG
  - Nodes = operations
  - Edges = dependences
- Single-pass traversal required to insert dependences
  1
- Example
  1: r1 = load(r2)
  2: r2 = r1 + r4
  3: store (r4, r2)
  4: p1 = cmp (r2 < 0)
  5: branch if p1 to BB3
  6: store (r1, r2)

```
BB3:
```

Dependence Edge Latencies

- *Edge latency* = minimum number of cycles necessary between initiation of the predecessor and successor in order to satisfy the dependence
- Register flow dependence, a → b
  - Latest_write(a) – Earliest_read(b)
- Register anti dependence, a → b
  - Latest_read(a) – Earliest_write(b) + 1
- Register output dependence, a → b
  - Latest_write(a) – Earliest_write(b) + 1
- Negative latency
  - Possible, means successor can start before predecessor
  - We will only deal with latency >= 0, so MAX any latency with 0

Dependence Edge Latencies (2)

- Memory dependences, a → b (all types, flow, anti, output)
  - latency = latest_serialization_latency(a) – earliest_serialization_latency(b) + 1
- Prioritized memory operations
  - Hardware orders memory ops by order in MultiOp
  - Latency can be 0 with this support
- Control dependences
  - branch → b
    - Op b cannot issue until prior branch completed
    - latency = branch_latency
  - a → branch
    - Op a must be issued before the branch completes
    - latency = 1 – branch_latency (can be negative)
    - conservative, latency = MAX(0, 1-branch_latency)
dependence graph

1. Draw dependence graph
2. Label edges with type and latencies

\[ \begin{align*}
    r_1 &= \text{load}(r_2) \\
    r_2 &= r_2 + 1 \\
    \text{store}(r_8, r_2) \\
    r_3 &= \text{load}(r_2) \\
    r_4 &= r_1 \times r_3 \\
    r_5 &= r_5 + r_4 \\
    r_2 &= r_6 + 4 \\
    \text{store}(r_2, r_5)
\end{align*} \]

**Dependence Graph Properties – \textit{Estart}**

- \textit{Estart} = earliest start time, (as soon as possible - ASAP)
  - Schedule length with infinite resources (dependence height)
  - \textit{Estart} = 0 if node has no predecessors
  - \textit{Estart} = \text{MAX}(\text{Estart}(\text{pred}) + \text{latency}) for each predecessor node
  - Example

**Lstart**

- \textit{Lstart} = latest start time, ALAP
  - Latest time a node can be scheduled s.t. sched length not increased beyond infinite resource schedule length
  - \textit{Lstart} = \textit{Estart} if node has no successors
  - \textit{Lstart} = \text{MIN}(\text{Lstart}(\text{succ}) - \text{latency}) for each successor node
  - Example

**Slack**

- \textit{Slack} = measure of the scheduling freedom
  - \textit{Slack} = \textit{Lstart} – \textit{Estart} for each node
  - Larger slack means more mobility
  - Example
### Critical Path

- **Critical operations = Operations with slack = 0**
  - No mobility, cannot be delayed without extending the schedule length of the block
  - Critical path = sequence of critical operations from node with no predecessors to exit node, can be multiple crit paths

![Critical Path Diagram]

### Class Problem

#### Node Table

<table>
<thead>
<tr>
<th>Node</th>
<th>Estart</th>
<th>Lstart</th>
<th>Slack</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>2</td>
<td>3</td>
<td>0.0</td>
</tr>
<tr>
<td>2</td>
<td>3</td>
<td>4</td>
<td>0.0</td>
</tr>
<tr>
<td>3</td>
<td>4</td>
<td>5</td>
<td>0.0</td>
</tr>
<tr>
<td>4</td>
<td>5</td>
<td>6</td>
<td>0.0</td>
</tr>
<tr>
<td>5</td>
<td>6</td>
<td>7</td>
<td>0.0</td>
</tr>
<tr>
<td>6</td>
<td>7</td>
<td>8</td>
<td>0.0</td>
</tr>
<tr>
<td>7</td>
<td>8</td>
<td>9</td>
<td>0.0</td>
</tr>
<tr>
<td>8</td>
<td>9</td>
<td>10</td>
<td>0.0</td>
</tr>
</tbody>
</table>

Critical path(s) =

### Operation Priority

- **Priority – Need a mechanism to decide which ops to schedule first (when you have multiple choices)**

**Common priority functions**

- **Height – Distance from exit node**
  - Give priority to amount of work left to do
- **Slackness – inversely proportional to slack**
  - Give priority to ops on the critical path
- **Register use – priority to nodes with more source operands and fewer destination operands**
  - Reduces number of live registers
- **Uncover – high priority to nodes with many children**
  - Frees up more nodes
- **Original order – when all else fails**

### Height-Based Priority

- **Height-based is the most common**
  - \( \text{priority}(op) = \text{MaxLstart} - \text{Lstart}(op) + 1 \)
List Scheduling (Cycle Scheduler)

- Build dependence graph, calculate priority
- Add all ops to UNSCHEDULED set
- time = -1
- while (UNscheduled is not empty)
  - time++
  - READY = UNSCHEDULED ops whose incoming dependences have been satisfied
  - Sort READY using priority function
  - For each op in READY (highest to lowest priority)
    - op can be scheduled at current time? (are the resources free?)
      - Yes, schedule it, op.issue_time = time
      - Mark resources busy in RU_map relative to issue time
      - Remove op from UNSCHEDULED/READY sets
    - No, continue

Cycle Scheduling Example

Machine: 2 issue, 1 memory port, 1 ALU
Memory port = 2 cycles, non-pipelined
ALU = 1 cycle

COS 598C - Advanced Compilers

Cycle Scheduling Example (2)

Cycle Scheduling Example (3)

COS 598C - Advanced Compilers
Machine: 2 issue, 1 memory port, 1 ALU
Memory port = 2 cycles, pipelined
ALU = 1 cycle

1. Calculate height-based priorities
2. Schedule using cycle scheduler