Instruction-level Parallelism

- Most modern processors have the ability to execute several adjacent instructions simultaneously.
  - Pipelined machines.
  - Very-long-instruction-word machines (VLIW).
  - Superscalar machines.
  - Dynamic scheduling/out-of-order machines.
- ILP is limited by several kinds of execution constraints:
  - Data dependence constraints.
  - Resource constraints ("hazards")
  - Control hazards

Execution Constraints

- Data-dependence constraints:
  - If instruction A computes a value that is read by instruction B, then B cannot execute before A is completed.
- Resource hazards:
  - Limited # of functional units:
    - If there are \( n \) functional units (e.g., multipliers), then only \( n \) instructions of the same kind can execute at once.
  - Limited instruction issue:
    - If the instruction-issue unit can issue only \( n \) instructions at a time, then this limits ILP.
  - Limited register set:
    - Any schedule of instructions must have a valid register allocation.

Instruction Scheduling

- The purpose of instruction scheduling (IS) is to order the instructions for maximum ILP.
  - Keep all resources busy every cycle.
  - If necessary, eliminate data dependences and resource hazards to accomplish this.
- The IS problem is NP-complete (and bad in practice).
  - So heuristic methods are necessary.

How can you tell this is an old slide?
Instruction Scheduling

- There are many different techniques for IS.
  - Still an open area of research.
- Most optimizing compilers perform good local IS, and only simple global IS.
- The biggest opportunities are in scheduling the code for loops.

Should the Compiler Do IS?

- Many modern machines perform dynamic reordering of instructions.
  - Also called “out-of-order execution” (OOOE).
  - Not yet clear whether this is a good idea.
  - Pro:
    - OOOE can use additional registers and register renaming to eliminate data dependences that no amount of static IS can accomplish.
    - No need to recompile programs when hardware changes.
  - Con:
    - OOOE means more complex hardware (and thus longer cycle times and more wattage).
    - And can’t be optimal since IS is NP-complete.

What we will cover

- Scheduling basic blocks
  - List scheduling
  - Long-latency operations
  - Delay slots
- Scheduling for clusters architectures
- Software Pipelining (next week)

- What we need to know
  - pipeline structure
  - data dependencies
  - register renaming

Instruction Scheduling

- In the von Neumann model of execution an instruction starts only after its predecessor completes.

  instr 1  instr 2

  time

- This is not a very efficient model of execution.
  - von Neumann bottleneck or the memory wall.
Instruction Pipelines

- Almost all processors today use instructions pipelines to allow overlap of instructions (Pentium 4 has a 20 stage pipeline!!!).
- The execution of an instruction is divided into stages; each stage is performed by a separate part of the processor.

<table>
<thead>
<tr>
<th>instr</th>
<th>F</th>
<th>D</th>
<th>E</th>
<th>M</th>
<th>W</th>
</tr>
</thead>
</table>
  | time  | F: Fetch instruction from cache or memory.
  |       | D: Decode instruction.
  |       | E: Execute. ALU operation or address calculation.
  |       | M: Memory access.
  |       | W: Write back result into register.

- Each of these stages completes its operation in one cycle (shorter than the cycle in the von Neumann model).
- An instruction still takes the same time to execute.

However, we overlap these stages in time to complete an instruction every cycle.

Unconditional jump address available only after Decode.
Conditional branch address available only after Execute.

Jump/Branch Delay Slot(s)

- Control hazards, i.e. jump/branch instructions.

Pipeline Hazards

- Structural Hazards
  - two instructions need the same resource at the same time
  - memory or functional units in a superscalar.
- Data Hazards
  - an instruction needs the results of a previous instruction
    \[ r_1 = r_2 + r_3 \]
    \[ r_4 = r_1 + r_1 \]
    \[ r_1 = [r_2] \]
    \[ r_4 = r_1 + r_1 \]
  - solved by forwarding and/or stalling
  - cache miss?
- Control Hazards
  - jump & branch address not known until later in pipeline
  - solved by delay slot and/or prediction
Jump/Branch Delay Slot(s)

- One option is to stall the pipeline (hardware solution).

- Another option is to insert a no-op instructions (software).

- Both degrade performance!

Jump/Branch Delay Slots

- In other words, the instruction(s) in the delay slots of the jump/branch instruction always get(s) executed when the branch is executed (regardless of the branch result).

- Fetching from the branch target begins only after these instructions complete.

Branch Prediction

- Current processors will speculatively execute at conditional branches
  - if a branch direction is correctly guessed, great!
  - if not, the pipeline is flushed before instructions commit (WB).

- Why not just let compiler schedule?
  - The average number of instructions per basic block in typical C code is about 5 instructions.
  - branches are not statically predictable
  - What happens if you have a 20 stage pipeline?
Data Hazards

\[
\begin{align*}
  r_1 &= r_2 + r_3 \\
  r_4 &= r_1 + r_1 \\
  [r_2] &= r_1 \\
  r_4 &= r_1 + r_1
\end{align*}
\]

Defining Dependencies

- Flow Dependence \( W \rightarrow R \) \( \delta_f \) \( \text{true} \)
- Anti-Dependence \( R \rightarrow W \) \( \delta_a \) \( \text{false} \)
- Output Dependence \( W \rightarrow W \) \( \delta_o \)
- Input Dependence \( R \rightarrow R \) \( \delta_i \)

Example Dependencies

S1) \( a=0; \)
S2) \( b=a; \)
S3) \( c=a+d+e; \)
S4) \( d=b; \)
S5) \( b=5+e; \)

- \( S1 \) \( \delta_f S2 \) due to \( a \)
- \( S1 \) \( \delta_f S3 \) due to \( a \)
- \( S2 \) \( \delta_f S4 \) due to \( b \)
- \( S3 \) \( \delta_a S4 \) due to \( d \)
- \( S4 \) \( \delta_a S5 \) due to \( b \)
- \( S2 \) \( \delta_o S5 \) due to \( b \)
- \( S3 \) \( \delta_f S5 \) due to \( a \)

Renaming of Variables

- Sometimes constraints are not “real,” in the sense that a simple renaming of variables/registers can eliminate them.
  - Output dependence (WW): A and B write to the same variable.
  - Anti dependence (RW): A reads from a variable to which B writes.
- In such cases, the order of A and B cannot be changed unless variables are renamed.
- Can sometimes be done by the hardware, to a limited extent.
Register Renaming Example

• Can perform register renaming after register allocation
  • Constrained by available registers
  • Constrained by live on entry/exit
  • Instead, do scheduling before register allocation

Scheduling a BB

• Assume:
  • load 5
  • store 5
  • mult 2
  • others 1
  • operations are non-blocking

  • Latency of operations
  • # of registers
  • Assume:
    • load 5
    • store 5
    • mult 2
    • others 1
    • Also assume,
      • operations are non-blocking

We can do better

• Assume:
  • load 5
  • store 5
  • mult 2
  • others 1
  • operations are non-blocking

  1 r1 ← [fp+w]
  2 r2 ← [fp+x]
  3 r3 ← [fp+y]
  4 r4 ← [fp+z]
  5 r5 ← 2
  6 r1 ← r1 * r5
  8 r1 ← r1 * r2
  10 r1 ← r1 * r3
  12 r1 ← r1 * r4
  14 [fp+w] ← r1
  19 r1 can be used again

• x ← w * 2 * x * y * z
  1 r1 ← [fp+w]
  2 r2 ← 2
  6 r1 ← r1 * r2
  7 r2 ← [fp+x]
  12 r1 ← r1 * r2
  13 r2 ← [fp+y]
  18 r1 ← r1 * r2
  19 r2 ← [fp+z]
  24 r1 ← r1 * r2
  26 [fp+x] ← r1
  33 r1 can be used again

We can do even better if we assume what?
Defining Better

<table>
<thead>
<tr>
<th></th>
<th>r1   ← [fp+w]</th>
<th></th>
<th>r1   ← [fp+w]</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td></td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>r2 ← 2</td>
<td>2</td>
<td>r2 ← [fp+x]</td>
</tr>
<tr>
<td>3</td>
<td>r1 ← r1 * r2</td>
<td>3</td>
<td>r3 ← [fp+y]</td>
</tr>
<tr>
<td>4</td>
<td>r2 ← [fp+x]</td>
<td>4</td>
<td>r4 ← [fp+z]</td>
</tr>
<tr>
<td>5</td>
<td>r1 ← r1 * r2</td>
<td>5</td>
<td>r5 ← 2</td>
</tr>
<tr>
<td>6</td>
<td>r1 ← r1 * r2</td>
<td>6</td>
<td>r1 ← r1 * r5</td>
</tr>
<tr>
<td>7</td>
<td>r2 ← [fp+y]</td>
<td>7</td>
<td>r1 ← r1 * r2</td>
</tr>
<tr>
<td>8</td>
<td>r1 ← r1 * r2</td>
<td>8</td>
<td>r1 ← r1 * r3</td>
</tr>
<tr>
<td>9</td>
<td>r2 ← [fp+z]</td>
<td>9</td>
<td>r1 ← r1 * r4</td>
</tr>
<tr>
<td>10</td>
<td>r1 ← r1 * r2</td>
<td>10</td>
<td></td>
</tr>
<tr>
<td>11</td>
<td></td>
<td>11</td>
<td></td>
</tr>
<tr>
<td>12</td>
<td>[fp+w] ← r1</td>
<td>12</td>
<td></td>
</tr>
<tr>
<td>13</td>
<td></td>
<td>13</td>
<td></td>
</tr>
<tr>
<td>14</td>
<td>[fp+w] ← r1</td>
<td>14</td>
<td></td>
</tr>
<tr>
<td>15</td>
<td>r1 can be used again</td>
<td>15</td>
<td></td>
</tr>
</tbody>
</table>

The Scheduler

- Given:
  - Code to schedule
  - Resources available (FU and # of Reg)
  - Latencies of instructions
- Goal:
  - Correct code
  - Better code [fewer cycles, less power, fewer registers, ...]
  - Do it quickly

More Abstractly

- Given a graph $G = (V,E)$ where
  - nodes are operations
    - Each operation has an associated delay and type
  - edges between nodes represent dependencies
  - The number of resources of type $t$, $R(t)$
- A schedule assigns to each node a cycle number:
  - $S(n) \geq 0$
  - If $(n,m) \in G$, $S(m) \geq S(n) + \text{delay}(n)$
  - $|\{ n \mid S(n) = x \text{ and type}(n) = t\}| \leq R(t)$
- Goal is shortest length schedule, where length
  - $L(S) = \text{max over } n, S(n) + \text{delay}(n)$

List Scheduling

- Keep a list of available instructions, I.e.,
  - If we are at cycle $k$, then all predecessors, $p$, in graph have all been scheduled so that $S(p) + \text{delay}(p) \leq k$
- Pick some instruction, $n$, from queue such that there are resources for type($n$)
- Update available instructions and continue
- It is all in how we pick instructions
Lots of Heuristics

- forward or backward
- choose instructions on critical path
- ASAP or ALAP
- Balanced paths
- depth in schedule graph

DLS (1995)

- Aim: avoid pipeline hazards in load/store unit
  - load followed by use of target reg
  - store followed by load
- Simplifies in two ways
  - 1 cycle latency for load/store
  - includes all dependencies (WaW included)

The algorithm

- Construct Scheduling dag
- Make srcs of dag candidates
- Pick a candidate
  - Choose an instruction with an interlock
  - Choose an instruction with a large number of successors
  - Choose with longest path to root
- Add newly available instruction to candidate list

1) ld r1 ← [a]
2) ld r2 ← [b]
3) add r1 ← r1 + r2
4) ld r2 ← [c]
5) ld r3 ← [d]
6) mul r4 ← r2 * r3
7) add r1 ← r1 + r4
8) add r2 ← r2 + r3
9) mul r2 ← r2 * r3
10) add r1 ← r1 + r2
11) st [a] ← r1
Trace Scheduling

- Basic blocks typically contain a small number of instructions.
- With many FUs, we may not be able to keep all the units busy with just the instructions of a BB.
- Trace scheduling allows block scheduling across BBs.
- The basic idea is to dynamically determine which blocks are executed more frequently. The set of such BBs is called a trace.

The trace is then scheduled as a single BB.
- Blocks that are not part of the trace must be modified to restore program semantics if/when execution goes off-trace.

VLIW

- Very Long Instruction Word
- Multiple Function Units
- Statically scheduled
- Examples
  - Itanium
  - TI C6x
  - Memory
  - ALU
  - FPU
  - Branch
  - ALU

- Scalability Issues?
Why Clusters?

- Reduce number of register ports
- Reduce length of buses
- Example: C6x

Some more details

- Not all FUs the same
  - some overlap
  - Add on L,S,D
  - delay 1 mostly
  - load, 4 * mult, 2
- Not all srcs the same
  - both srcs from same RF
  - L,S,M: 1 from other RF
  - Only L&S used for copy
  - S &M: only right from other RF
- Not all dests the same
  - if 2 D ops, srcs and dests must be to different RFs

Phase-Ordering

- Valid assembly must
  - be properly partitioned
  - be properly scheduled
  - be properly register allocated
- What order to perform?

Partitioning/Scheduling Basics

- Objectives:
  - Balance workload per cluster
  - Minimize critical intercluster communication
Bottom-Up Greedy (BUG) 1985

- Assigns operations to cluster, then schedules
- Recurses down DFG
  - Assigns ops to cluster based on estimates of resource usage
  - Assigns ops on critical path first
  - Tries to schedule as early as possible

Leupers Approach

- Integrate partitioning and scheduling
- Use Simulated Annealing to determine partition
- The eval step in the SA loop is the scheduler!
- Deals with details of architecture

Integrated Approaches

- Leupers, 2000
  - Combine partitioning & scheduling
  - Iterative approach
- B-init/B-iter, 2002
  - Initial binding/scheduling
  - Iterative improvement
- RHOP, 2003
  - Region-based graph partitioning

Example Result
Approach: SA

1. Generate random partitioning & schedule
2. Pick one node to swap
3. Reduce T
4. Schedule

IF Newcost < oldcost * e^T THEN
   Undo swap
ELSE
   Reduce T
ENDIF

Basic Algorithm

Algorithm Partition
input DFG G with nodes;
output: DP: array [1..N] of 0, 1;
var int i, r, cost, mincost;
float T;
begin
   T = 10;
P := Random partitioning;
mincost := Listscheduling(G, P);
WHILE_LOOP;
   return DP;
end.

WHILE_LOOP:
   while T > 0.01 do
      for i = 1 to 50 do
         r := RANDOM(1,n);
P[r] := 1 - P[r];
cost := Listscheduling(G, P);
delta := cost - mincost;
   if delta < 0 or RANDOM(0, 1) < exp(-delta/T) then
      mincost := cost
   else
      P[r] := 1 - P[r]
   end if;
   end for;
   T := 0.9 * T;
end while;

Scheduling

- Use a List Scheduler
- Tie breaker for next ready node is min ALAP
- Heart of routine is ScheduleNode

ScheduleNode

- Goal: insert node m as early as possible
  - don’t violate resource constraints
  - don’t violate dependence constraints
- First try based on ASAP
- Until it is scheduled
  - See if there is an FU that can execute m
  - check source registers
    - if both from same RF as FU, done
    - if not: must decide what to do
Dealing with x-RF transfers

- Two ways to XFER:
  - Source can be Xfered this cycle
  - Source can be copied in previous cycle
- If neither is true
  - maybe commutative?
  - try to schedule next cycle

Handling loads

- After scheduling an op to a cluster, see if it is a load. Determine the partition of the result
- Scheduling of loads uses the RF of the address
- Scheduling the result
  - check to see if both units are free
  - if so, check to see where it is used most and schedule result in that RF

Basic Scheduling of a Node

```
algorithm ScheduleNode(S, m, P)
  input Schedule S, node m, partition P;
  output: new schedule with m
  var cs: control step
  begin
    cs = EarliestControlStep(m)-1;
    repeat
      cs++;
      f = GetNodeUnit(m, cs, P);
      if (f == ∅) continue;
      if (m needs arg from other RF) then
        CheckArgTransfer();
      if (no transfer possible) then continue;
      else TryScheduleTransfers();
    until (m is scheduled);
    ...

CheckArgTransfer:
  • if CSE, reuse
  • if room for move, do so
  • if X-path avail and valid, use it

TryScheduleTransfers:
  • reuse first
  • move if possible
  • use X-path if possible
  • try commuting args
  • Can fail
```

Benefits/Drawbacks

- Does not predetermine the partitioning
- handles many real world details
- local decisions only
- Time consuming
- Very specific to C6x
- may not scale to multiple clusters?
RHOP partitioning/scheduling

- 2003, Chu, Fan, Mahlke
- Global scheduling and partitioning
- Based on graph-partitioning
- Avoid local scheduling pitfalls
- Avoid “scheduling”

RHOP Approach

- Opposite approach to conventional clustering
- Global view
  - Graph partitioning strategy [Aletà ’01, ’02]
  - Identify tightly coupled operations - treat uniformly
- Non scheduler-centric mindset
  - Prescheduling technique
  - Doesn’t complicate scheduler
  - Enable global view of code
  - Estimate-based approach [Lapinskii ’01]

Region-based Hierarchical Operation Partitioning (RHOP)

- Code is considered region at a time
- Weight calculation creates guides for good partitions
- Partitioning clusters based on given weights

Edge Weights

- Slack distribution allocates slack to certain edges
  - Edge slack = Istart_{dest} - latency_{edge}
  - First come, first serve method used

1 - slack
8 - no slack after dist
10 - critical
RHOP - Partitioning Phase

- Modified Multilevel-KL algorithm [Kernighan ‘69]
- Multilevel graph partitioning consists of two stages
  1. Coarsening stage
  2. Refinement stage

  **Coasening:** meld partitions to reduce weight, thus, will keep edges on CP together.

Cluster Refinement

- 3 questions to answer:
  1. Which cluster should operations move from?
  2. How good is the current partition?
  3. How profitable is it to move X from cluster A to B?

Node Weights

- Create a metric to determine resource usage

  **Dedicated Resources**
  \[ \text{op wgt}_c = \frac{1}{\text{ops that can execute on c in 1 cycle}} \]
  Accounts for FU’s

  **Shared Resources**
  \[ \text{shared wgt}_c = \frac{\text{resource limited sched length on c}}{\# \text{ ops}} \]
  Accounts for buses, ports

Where Should Operations Move From?

\[ \text{lwt}_{ct} = \max_{\text{op groups}} \sum_{\text{ops at } t} \frac{\text{op wgt}_c}{\text{op slack} + 1} \]
Where Should Operations Move From?

\[ T_{\text{wgt}_{\text{c}}} = \frac{\#\text{ops in } c \text{ at } r \cdot \text{shared}_{\text{wgt}}}{\text{slack}_{\text{wgt}}} + 1 \]

\[ \text{cluster}_{\text{wgt}} = \left( \sum_{t=0}^{\text{max extant}} \max (\text{load}_{\text{wgt}}, \text{wgt}_{\text{c}}) - 1 \right) \]

\[ \text{cluster}_{\text{wgt}} = \max \left( \sum_{\text{op}_{\text{wgt}}} \text{op}_{\text{wgt}} + 1 \right) \]

How Good is this Partition?

\[ SL = \sum_{t=0}^{\text{max extant}} \max (\text{cwt}_{\text{t}}, \text{t} - 1) \]

\[ \text{Max} \]

\[ \text{Cluster 1} \]

\[ \text{Cluster 2} \]

How Good is This Proposed Move?

\[ E_{\text{gain}} = \sum_{\text{merged edges}} \text{edge}_{\text{wgt}} - \sum_{\text{cut edges}} \text{edge}_{\text{wgt}} \]

\[ L_{\text{gain}} = SL_{\text{(before)}} - SL_{\text{(after)}} \]

\[ M_{\text{gain}} = E_{\text{gain}} + (L_{\text{gain}} \times \text{CRITICAL EDGE COST}) \]

Experimental Evaluation

• Trimaran toolset: a re-targetable VLIW compiler

• Evaluated DSP kernels and SPECint2000

• 64 registers per cluster

• Latencies similar to Itanium

• Perfect caches

• For more detailed results, see paper
Conclusions

- A new, region-scoped method for clustering operations
  - Prescheduling technique
  - Estimates on schedule length used instead of scheduler
  - Combines slack distribution with multilevel-KL partitioning
- Performs better as number of resources increases

Average Improvement

<table>
<thead>
<tr>
<th>Machine</th>
<th>RHOP vs BUG</th>
</tr>
</thead>
<tbody>
<tr>
<td>2-1111</td>
<td>-1.8%</td>
</tr>
<tr>
<td>2-2111</td>
<td>3.7%</td>
</tr>
<tr>
<td>4-1111</td>
<td>14.3%</td>
</tr>
<tr>
<td>4-2111</td>
<td>15.3%</td>
</tr>
<tr>
<td>4-H</td>
<td>8.0%</td>
</tr>
</tbody>
</table>

Previous Work

<table>
<thead>
<tr>
<th>Algorithm</th>
<th>When (rel. to sched)</th>
<th>Scope</th>
<th>Desirability Metric</th>
<th>Grouping</th>
</tr>
</thead>
<tbody>
<tr>
<td>BUG</td>
<td>Before</td>
<td>Local</td>
<td>Flat</td>
<td>X</td>
</tr>
<tr>
<td>RHOP</td>
<td>Before</td>
<td>Local</td>
<td>Flat</td>
<td>X</td>
</tr>
<tr>
<td>B-ITER</td>
<td>Before</td>
<td>Local</td>
<td>Flat</td>
<td>X</td>
</tr>
<tr>
<td>Shop</td>
<td>Before</td>
<td>Local</td>
<td>Flat</td>
<td>X</td>
</tr>
</tbody>
</table>