Scheduling

Introduction

• We shall discuss:
  — Straight line scheduling
  — Trace Scheduling
  — Kernel Scheduling (Software Pipelining)
  — Vector Unit Scheduling
  — Cache coherence in coprocessors
Introduction

• Scheduling: Mapping of parallelism within the constraints of limited available parallel resources

• Best Case Scenario: All the uncovered parallelism can be exploited by the machine

• In general, we must sacrifice some execution time to fit a program within the available resources

• Our goal: Minimize the amount of execution time sacrificed

Introduction

• Variants of the scheduling problem:
  — Instruction scheduling: Specifying the order in which instructions will be executed
  — Vector unit scheduling: Make most effective use of the instructions and capabilities of a vector unit. Requires pattern recognition and synchronization minimization

• We will concentrate on instruction scheduling (fine grained parallelism)

• Multiprocessor scheduling, which distributes work to asynchronous parallel coprocessors in an attempt to balance the load and minimize running time, was discussed in Section 6.6
Introduction

- All modern machine architectures have the capability of issuing several instructions each cycle, thereby have the theoretical capability of achieving a high degree of fine-grained parallelism.
- Effectively utilizing that capability requires that instructions be presented in an order that allows the process to find and issue those instructions that can be executed in parallel.
- There are two principal impediments to achieve that goal:
  - Dependences between instructions that force a sequential reordering
  - Resource limitations that force serialization of instructions that need the same resource.
- The goal of instruction scheduling for uniprocessors is to:
  - Generate instructions in an order that places dependent instruction far enough apart so that the dependence does not cause delays
  - Ensure that as many functional units as possible are busy on each cycle.

Introduction

- Categories of processors supporting fine-grained parallelism:
  - VLIW
    - A long instruction word that contains a field controlling each available functional unit
    - Compiler must explicitly specify parallelism and must be more observant of dependences
  - Superscalar processors
    - Multiple functional units controlled and scheduled by the hardware
    - Instruction decode reads a group of instructions at the same time and determines the dependences between them
    - The unit schedules the instructions across multiple functional units if there are free resources for doing the work and the instruction can be safely scheduled in parallel
    - Superscalar processors provide hazard protection by either stalling execution or using register renaming to avoid executing an instruction whose operands are not quite ready.
Introduction

• Scheduling in VLIW and Superscalar architectures:
  — Order instruction stream so that as many function units as possible are being used on every cycle

• Standard approach:
  — Emit a sequential stream of instructions
  — Reorder this sequential stream to utilize available parallelism
  — Reordering must preserve dependences

Issue: Creating a sequential stream must consider available resources. This may create artificial dependences

\[ a = b + c + d + e \]

• One possible sequential stream:
  add a, b, c
  add a, a, d
  add a, a, e

• And, another (better because of possible parallelism):
  add r1, b, c
  add r2, d, e
  add a, r1, r2
Fundamental Conflict in Scheduling

- Fundamental conflict in scheduling:
  - If the original instruction stream takes into account available resources, it will create artificial dependences.
  - If not, then there may not be enough resources to correctly execute the stream.

Focus of Transformations

- Arrange instructions of the object program in such a way as to take maximum advantage of the parallelism in the machine architecture.
  - Parallelism in instruction processing and function execution.
  - Without compromising the meaning of the program.

- Assumption: some sets of resources, registers in particular, have already been allocated while other resources, such as functional units, have not.
Machine Model

- Machine contains a number of issue units
  - Each issue unit corresponds to a machine resource such as an integer or floating-point arithmetic unit
  - Each issue unit is able to issue an operation per cycle on the functional unit it controls, assumed to be pipelined
- Issue unit has an associated
  - type - kind of resource it controls
  - delay - number of cycles required before the result of an operation, once issued, is available
- $I^k_j$ denotes the $j^{th}$ unit of type $k$
- Number of units of type $k = m_k$
- Total number of issue units: $M = \sum_{i=1}^{l} m_i$
  where, $l = \text{number of issue-unit types in the machine}$

---

Machine Model

- We assume a VLIW model
  - The wide instruction consists of of $M$ subinstructions
- Goal of compiler: select set of $\leq M$ instructions for each cycle such that the number of instructions of type $k$ is $\leq m_k$
- Note that code can be generated easily for an equivalent superscalar machine
- Scheduling problem is represented as a graph in which the vertices represent instructions of a given type and the edges represent dependences with a given delay equal to the delay for the type of operation at the source vertex of the edge
Straight Line Graph Scheduling

• Scheduling a basic block
• Use a dependence graph defined as $G = (N, E, \text{type}, \text{delay})$
  – $N$ is the set of instructions in the code
  – Each $n \in N$ has a type, $\text{type}(n)$, and a delay, $\text{delay}(n)$
  – $(n_1, n_2) \in E$ iff $n_2$ must wait for completion of $n_1$ due to a shared register (true, anti, and output dependences)

• A correct schedule is a mapping $S$ from vertices in the graph to nonnegative integers representing cycle numbers such that:
  1. $S(n) \geq 0$ for all $n \in N$ (all instructions are executed at some point)
  2. If $(n_1, n_2) \in E$, $S(n_1) + \text{delay}(n_1) \leq S(n_2)$ (no dependences are violated)
  3. For any type $t$, no more than $m_t$ vertices of type $t$ are mapped to a given integer (only available resources are in used during any cycle)

• The length of a schedule, $S$, denoted $L(S)$ is defined as:
  $L(S) = \max_{n \in N} (S(n) + \text{delay}(n))$

• Goal of straight-line scheduling: find a shortest possible correct schedule. A straight line schedule is said to be optimal if:
  $L(S) \leq L(S_i), \forall$ correct schedules $S_i$
List Scheduling

• Use variant of topological sort:
  — Maintain a list of instructions that have no predecessors in the graph
  — Schedule these instructions
  — This allows other instructions to be added to the list

• Algorithm for list scheduling:
  — Schedule an instruction at the first opportunity after all instructions it depends on have completed
  — count array determines how many predecessors are still to be scheduled
  — earliest array maintains the earliest cycle on which the instruction can be scheduled
  — Maintain a number of worklists which hold instructions to be scheduled for a particular cycle number.
List Scheduling

- Consider the following code:
  ```
  mult c,a,b  
mult f,d,e  
add f,f,g  
add f,f,h  
add f,f,c
  ```
- Which multiple should be chosen to execute first if there is only one multiply unit?
- The one that doesn’t increase the length of the schedule by delaying evaluation of critical additions.

---

List Scheduling

- How shall we select instructions from the worklist?
  - Random selection
  - Selection based on other criteria: Worklists are priority queues. Highest Level First (HLF) heuristic schedules more critical instructions first
List Scheduling Algorithm I

Idea: Keep a collection of worklists $W[c]$, one per cycle
— We need $MaxC = \text{max delay} + 1$ of any instruction

Code:

```plaintext
for each $n \in N$ do begin count[n] := 0; earliest[n] = 0 end
for each $(n_1, n_2) \in E$ do begin
    count[$n_2$] := count[$n_2$] + 1;
    successors[$n_1$] := successors[$n_1$] $\cup$ ($n_2$);
end
for $i := 0$ to $MaxC - 1$ do $W[i]$ := $\emptyset$;
Wcount := 0;
for each $n \in N$ do
    if count[n] = 0 then begin
        $W[0]$ := $W[0]$ $\cup$ ($n$); Wcount := Wcount + 1;
    end
$c$ := 0; // $c$ is the cycle number
$cW$ := 0; // $cW$ is the number of the worklist for cycle $c$
instr[c] := $\emptyset$
```

Priority

```plaintext
while Wcount > 0 do begin
    while $W[cW] = \emptyset$ do begin
        $c := c + 1$; instr[c] := $\emptyset$; $cW := \text{mod}(cW+1, MaxC)$;
    end
    nextc := $\text{mod}(c+1, MaxC)$;
    while $W[cW] \neq \emptyset$ do begin
        select and remove an arbitrary instruction $x$ from $W[cW]$;
        if $\exists$ free issue units of type($x$) on cycle $c$ then begin
            instr[c] := instr[c] $\cup$ ($x$); Wcount := Wcount - 1;
            for each $y \in \text{successors}[x]$ do begin
                count[y] := count[y] - 1;
                earliest[y] := $\text{max}(\text{earliest}[y], c + \text{delay}(x))$;
                if count[y] = 0 then begin
                    loc := $\text{mod}(\text{earliest}[y], MaxC)$;
                    $W[loc]$ := $W[loc] \cup \{y\}$; Wcount := Wcount + 1;
                end
            end
        end else $W[nextc] := W[nextc] \cup \{x\}$;
    end
end
```

List Scheduling Algorithm II
Trace Scheduling

• Problems with list scheduling?
  — Doesn't look across block boundaries
  — Transition points between basic blocks become a problem
  — Must insert enough instructions at the end of a basic block to
    ensure that results are available on entry into next basic block
  — Results in significant overhead!
• Alternative to list scheduling: trace scheduling
• Trace: is a collection of basic blocks that form a single path
  through all or part of the program:
  — Trace scheduling schedules an entire trace at a time
  — Traces are chosen based on their expected frequencies of execution
• Must insert fixup code at points where control flow enters or
  exits the trace
• Caveat: Cannot schedule cyclic graphs
  — Loops must be unrolled

Trace Scheduling

• Three steps for trace scheduling:
  — Selecting a trace
  — Scheduling the trace
  — Inserting fixup code
**Inserting Fixup Code**

- If the trace is scheduled in the order
  
  \[ j = j + 1 \]
  
  \[ i = i + 2 \]
  
  if \( e_1 \)
  
  \[ k = i + 3 \]
  
  then there is a problem with the last instruction

- Why?

**Trace Scheduling**

- Trace scheduling avoids moving operations above splits or below joins unless it can prove via dependence analysis that other instructions will not be adversely affected.
Trace Scheduling

• Because of the necessity of inserting code, will trace scheduling always converge?
• Trace scheduling will always converge
• However, in the worst case, a very large amount of fixup code may result
  —Worst case: operations increase to $O(n^{en})$

Straight-line Scheduling: Conclusion

• Issues in straight-line scheduling:
  —Relative order of register allocation and instruction scheduling
    - There are trade-offs which ever is done first
  —Dealing with loads and stores:
    - Memory loads are very slow if the value is not in the cache
    - Thus loads should be scheduled as early as possible
    - Without sophisticated analysis, almost no movement is possible among memory references
Kernel Scheduling

- Drawback of straight-line scheduling:
  - Loops are unrolled
  - Ignores parallelism among loop iterations

- Kernel scheduling: Try to maximize parallelism across loop iterations

Kernel Scheduling

- Scheduling a loop is done by reorganizing the loop into three parts:
  - Prolog - contains code that must be performed before steady state can be reached
  - Kernel - contains code that must be executed on every cycle of the loop
  - Epilog - contains code that must be executed to finish the loop once the kernel can no longer be executed

- The kernel scheduling problem seeks to find a minimal-length kernel for a given loop

- Issue: loops with small iteration counts?
Kernel Scheduling: Software Pipelining

- A kernel scheduling problem is a graph: 
  \[ G = (N, E, \text{delay}, \text{type}, \text{cross}) \]
  where cross \((n_1, n_2)\), defined for each edge in \(E\), is the number of iterations crossed by the dependence relating \(n_1\) and \(n_2\).
  — Cross is just the distance of the dependence

- Temporal movement of instructions through loop iterations
- Software Pipelining - body of one loop iteration is pipelined across multiple iterations.

Software Pipelining

- A solution to the kernel scheduling problem is a pair of tables \((S, I)\), where:
  - The schedule \(S\) maps each instruction \(n\) to a cycle within the kernel
  - The iteration \(I\) maps each instruction to an iteration offset from zero, such that:
    \[ S[n_1] + \text{delay}(n_1) \leq S[n_2] + (I[n_2] - I[n_1] + \text{cross}(n_1,n_2))L_k(S) \]
    for each edge \((n_1, n_2)\) in \(E\), where:
    \(L_k(S)\) is the length of the kernel for \(S\).
    \[ L_k(S) = \max_{n \in N} (S[n]) \]
Software Pipelining

- Example:
  ```
  lw  r1,0  
  lw  r2,400 
  if  fr1, c 
  l0  if  fr2,a(r1) 
  l1  addf fr3,fr2,fr1 
  l2  sf  fr3,b(r1) 
  l3  addi  r1,r1,8 
  l4  comp  r1,r2 
  l5  ble  l0
  ```

- Machine has three units:
  - Load-store unit – with two cycle delay for loads and one cycle delay for stores
  - Integer unit – handles branch instructions with a delay of one cycle on the branch taken side for blt. All others instructions, one cycle delay
  - Float unit – three cycle delay

<table>
<thead>
<tr>
<th>Load/Store</th>
<th>Integer</th>
<th>Floating Pt.</th>
</tr>
</thead>
<tbody>
<tr>
<td>l0: if fr2,a(r1)</td>
<td>l3: addi r1,r1,8</td>
<td>l1: addf fr3,fr2,fr1</td>
</tr>
<tr>
<td>l2: sf fr3,b-16(r1)</td>
<td>l4: comp r1,r2</td>
<td>l5: ble l0</td>
</tr>
</tbody>
</table>

lw  r1,0  
lw  r2,400  
if  fr1, c  
l0  if  fr2,a(r1)  
l1  addf fr3,fr2,fr1  
l2  sf  fr3,b(r1)  
l3  addi  r1,r1,8  
l4  comp  r1,r2  
l5  ble  l0
Software Pipelining

• Have to generate epilog and prolog to ensure correctness

• Prolog:
  lw r1, 0
  lw r2, 400
  if fr1, c
  p1 if fr2, a(r1); addi r1, r1, 8
  p2 comp r1, r2
  p3 bgt e1; add fr3, fr2, fr1

• Epilog:
  e1 nop
  e2 nop
  e3 sf fr3, b-8(r1)
Software Pipelining

• Let $N$ be the loop upper bound. Then, the schedule length $L(S)$ is given by:
  $$L(S) = N L_k(S) + \max_{n \in N} (S[n] + \text{delay}(n) + (I[n] - 1)L_k(S))$$

• Minimizing the length of kernel minimizes the length of the schedule

Kernel Scheduling Algorithm

• Is there an optimal kernel scheduling algorithm?

• To answer the question we need to establish a reasonable lower bound for how well scheduling can do.

• We will look at two lower bounds based on:
  — Available resources
  — Data dependences
Kernel Scheduling Algorithm

• Resource usage constraint:
  — No recurrence in the loop
  — \#t - number of instructions in each iteration that must issue in a unit of type t
  \[ L_k(S) \geq \max \left\lfloor \frac{\#t}{m} \right\rfloor \]

• We can always find a schedule S, such that
  \[ L_k(S) = \max \left\lfloor \frac{\#t}{m} \right\rfloor \]

• It is always possible, in the absence of recurrences, to find a kernel schedule of optimal length

Kernel Scheduling Algorithm

• Essentially a list scheduler
• Augmented with a scoreboard that tracks the use of resources across loop iterations
• As the scheduler assigns each instruction an issue slot, it also marks the corresponding resource on the scoreboard for the time it is busy.
Software Pipelining Algorithm

procedure loop_schedule(G, L, S, I)
  topologically sort G;
  for each instruction x in G in topological order do begin
    earlyS := 0; earlyl := 0;
    for each predecessor y of x in G do
      thisS := S[y] + delay(y); thisl := I[y];
      if thisS ≥ L then begin
        thisS := mod(thisS, L); thisl := thisl + ceil(thisS/L);
      end
      if thisl > earlyl or thisS > earlyS then begin
        earlyl := thisl; earlyS := thisS;
      end
    end
    starting at cycle earlyS, find the first cycle c₀ where the resource needed by x
    is available, wrapping to the beginning of the kernel if necessary;
    S[x] := c₀;
    if c₀ < earlyS then I[x] := earlyl + 1; else I[x] := earlyl;
  end
end min_loop_schedule

Software Pipelining Algorithm

10 lw a,x(i)
11 addi a,a,1
12 addi a,a,1
13 addi a,a,1
14 sw a,x(i)

<table>
<thead>
<tr>
<th>Memory1</th>
<th>Integer1</th>
<th>Integer2</th>
<th>Integer3</th>
<th>Memory2</th>
</tr>
</thead>
<tbody>
<tr>
<td>10: S=1; I=0</td>
<td>10: S=1; I=1</td>
<td>10: S=1; I=2</td>
<td>10: S=1; I=3</td>
<td>10: S=2; I=3</td>
</tr>
</tbody>
</table>

• If there are 3 integer units and 2 load/store unit, we should
  be able to schedule with one instruction
• But can we?
Software Pipelining Algorithm

• On each cycle, each unit has to work on a different iteration of the loop
• So we would rewrite it as
  \[\begin{align*}
  l_0 & \text{ lw } a, x(i) \\
  l_1 & \text{ addi } b, a, 1 \\
  l_2 & \text{ addi } c, b, 1 \\
  l_3 & \text{ addi } d, c, 1 \\
  l_4 & \text{ sw } d, x(i)
  \end{align*}\]
• But are there enough machine registers?

Cyclic Data Dependence Constraint

• Given a cycle of dependences \((n_1, n_2, ..., n_k)\):

\[
L_k(S) \geq \frac{\sum_i \text{delay}(n_i)}{\sum_i \text{cross}(n_i, n_{i+1})}
\]

— Right hand side is called the slope of the recurrence
— The numerator is the time to execute a single traversal of a recurrence
— The denominator is the number of iteration the recurrence crosses

\[
L_k(S) \geq \text{MAX}_c \left[ \frac{\sum_i \text{delay}(n_i)}{\sum_i \text{cross}(n_i, n_{i+1})} \right]
\]
Kernel Scheduling Algorithm

procedure kernel_schedule(G, S, I)
—use the all-pairs shortest path algorithm to find the cycle in the schedule graph G with the greatest slope;
—designate all cycles with this slope as critical cycles;
—mark every instruction in the G that is on a critical cycle as a critical instruction;
—compute the lower bound LB for the loop as the maximum of the slope of the critical recurrence given by Equation 10.10 and the hardware constraint as given in Equation 10.7
—N := the number of instructions in the original loop body;
—let G0 be G with all cycles broken by eliminating edges into the earliest instruction in the cycle within the loop body;

failed := true;
for L := LB to N while failed do begin
  // try to schedule the loop to length L
  loop_schedule(G0, L, S, I);
  // test to see if the schedule succeeded
  allOK := true;
  for each dependence cycle C while allOK do begin
    for each instruction v that is a part of C while allOK do begin
      if I[v] > 0 then allOK := false;
      else if v is the last instruction in the cycle C and v0 is the first instruction in the cycle and
      mod(S[v] + delay(v), L) > S[v0]
      then allOK = false;
    end
  end
  if allOK then failed := false;
end
end kernel_schedule
Prolog Generation

- Prolog:
  \[ \text{range}(S) = \max_{n \in N} (I[n]) + 1 \]
  - range = r = number of iterations executed for all instructions corresponding to a single iteration in the original loop to issue

- To get loop into steady state (priming the pipeline):
  - Lay out \((r-1)\) copies of the kernel
  - Any instruction with \(I[n] = i > r-1\) replaced by no-op in the first \(i\) copies

- Use list scheduling to schedule the prolog

Epilog Generation

- After last iteration of kernel, \(r-1\) iterations are required to wind down
  - Lay out \(r-1\) copies of the kernel
  - Any instruction \(n\) such that \(I[n] = I < r-1\) replaced by a no-op
  - Use list scheduling

- However, must also account for last instructions to complete to ensure all hazards outside the loop are accommodated

- Additional time required:
  \[ \Delta S = \left( \max_{n \in N} ((I[n] - 1)I_k(S) + S[n] + \text{delay}(n)) - rL_k(S) \right) \]

- Length of epilog has the following upper bound:
  \( (r-1) L_k(S) + \Delta S \)
Software Pipelining: Conclusion

• Issues to consider in software pipelining:
  — Increased register pressure: May have to resort to spills

• Control flow within loops:
  — Use If-conversion or construct control dependences
  — Put control dependence in data dependence graph and schedule as we have seen before
  — Schedule control flow regions using a non-pipelining approach and treat those areas as black boxes when pipelining

Vector Unit Scheduling

• Chaining:
  vload t1, a
  vload t2, b
  vadd t3, t1, t2
  vstore t3, c

• Assume each vector instruction takes 64 cycles
  — 192 cycles without chaining
  — 66 cycles with chaining

• Hardware usually takes care of this chaining, but the software can have some influence

• Proximity within instructions required for hardware to identify opportunities for chaining
Vector Unit Scheduling

vload a, x(i)  ·  2 load pipes
vload b, y(i)  ·  1 addition pipe
vadd t1, a, b  ·  1 multiplication pipe
vload c, z(i)
vmul t2, c, t1
vmul t3, a, b
vmul t4, c, t3

• Rearranging:
  vload a, x(i)
  vload b, y(i)
  vadd t1, a, b
  vmul t3, a, b
  vload c, z(i)
  vmul t2, c, t1
  vmul t4, c, t3

• Chaining problem solved by weighted fusion algorithm:
  — Variant of fusion algorithm seen in Chapter 8
  — Takes into consideration resource constraints of machine (number of pipes)
  — Weights are recomputed dynamically: For instance, if an addition and a subtraction are selected for chaining, then a load that is an input to both the addition and subtraction will be given a higher weight after fusion
We use weighted fusion in the following manner:

- Construct a dependence graph for the straight-line code to which chaining is to be applied.
- Weight each edge with the length of the vector it represents. If this cannot be determined, use the full vector register length.
- Apply the constrained weighted fusion algorithm to determine maximum fusion groups. At each step where the next fusion edge is selected, if several edges are tied for the heaviest weight, select one that is incident on the most recent fusion group, favoring the edge with the source and sink that are earliest in the original order.

```plaintext
vload a,x(i)
vload b,y(i)
vadd t1,a,b
vload c,z(i)
vmul t2,c,t1
vmul t3,a,b
vadd t4,c,t3
```
### Vector Unit Scheduling

- vload a, x(i)
- vload b, y(i)
- vadd t1, a, b
- vmul t3, a, b
- vload c, z(i)
- vmul t4, c, t3

After Fusion:
- vload a, x(i)
- vload b, y(i)
- vadd t1, a, b
- vmul t3, a, b
- vload c, z(i)
- vmul t2, c, t1
- vadd t4, c, t3

### Co-processors

- Co-processor can access main memory, but cannot see the cache
- Cache coherence problem
- Solutions:
  - Special set of memory synchronization operations
  - Stall processor on reads and writes (waits)
- Minimal number of waits essential for fast execution
Co-processors

- Use data dependence to insert these waits
  - True – main processor insures store are completed by the source processor before the reads are done on the sink processor
  - Anti – main processor ensures loads are completed on the source processor before the stores are done on the sink processor
  - Output – main processor ensures that the second store completes after the first store

- Positioning of waits important to reduce number of waits

```
store A(I)  ---
... region 1
store B(I)  ---
... region 2
load coprocessor A(I)
... region 3
load coprocessor B(I)
```

---

Co-processors

- Algorithm to insert waits:
  - Make a single pass starting from the beginning of the block
  - Note source of edges
  - When target reached, insert wait

- Produces minimum number of waits in absence of control flow

- Minimizing waits in presence of control flow is NP Complete. Compiler must use heuristics
Conclusion

- We looked at:
  - Straight line scheduling: For basic blocks
  - Trace Scheduling: Across basic blocks
  - Kernel Scheduling: Exploit parallelism across loop iterations
  - Vector Unit Scheduling
  - Issues in cache coherence for coprocessors