Main Problems

- Need special compiler technology to generate efficient code on modern architectures
- Pipelined machines: scheduling to expose instructions which can run in parallel in the pipeline, without stalls
- Superscalar, VLIW: scheduling to expose instruction which can run fully in parallel
- Symmetric multiprocessors (SMP): transformations to expose coarse-grain parallelism
- Memory hierarchies: transformations to improve memory system performance
- Need knowledge about dependencies between instructions
- Book: "Optimizing Compilers for Modern Architectures", by Kennedy, Allen

Pipelined Machines

- Example pipeline:
  - Fetch
  - Decode
  - Execute
  - Memory access
  - Write back
- Simultaneously execute stages of different instructions

<table>
<thead>
<tr>
<th>Instr 1</th>
<th>Fetch</th>
<th>Dec</th>
<th>Exe</th>
<th>Mem</th>
<th>WB</th>
</tr>
</thead>
<tbody>
<tr>
<td>Instr 2</td>
<td>Fetch</td>
<td>Dec</td>
<td>Exe</td>
<td>Mem</td>
<td>WB</td>
</tr>
<tr>
<td>Instr 3</td>
<td>Fetch</td>
<td>Dec</td>
<td>Exe</td>
<td>Mem</td>
<td>WB</td>
</tr>
</tbody>
</table>

Stall the Pipeline

- It is not always possible to pipeline instructions
- Example 1: branch instructions

<table>
<thead>
<tr>
<th>Branch Fetch</th>
<th>Dec</th>
<th>Exe</th>
<th>Mem</th>
<th>WB</th>
</tr>
</thead>
<tbody>
<tr>
<td>Target</td>
<td>Fetch Dec</td>
<td>Exe</td>
<td>Mem</td>
<td>WB</td>
</tr>
</tbody>
</table>

- Example 2: load instructions

<table>
<thead>
<tr>
<th>Load Fetch</th>
<th>Dec</th>
<th>Exe</th>
<th>Mem</th>
<th>WB</th>
</tr>
</thead>
<tbody>
<tr>
<td>Use</td>
<td>Fetch Dec</td>
<td>Exe</td>
<td>Mem</td>
<td>WB</td>
</tr>
</tbody>
</table>

Instruction Scheduling

- Instruction scheduling = reorder instructions to improve the parallel execution of instructions
- Essentially, compiler detects parallelism in the code
- Instruction Level Parallelism (ILP) = parallelism between individual instructions
  - Instruction scheduling: reorder instructions to expose ILP

Pipelined Machines

- Instructions cannot be executed concurrently in the pipeline because of hazards:
  - Control hazard: target of branch not known in the early stages of the pipeline, cannot fetch next instruction
  - Data hazard: results of an instruction not available for a subsequent instruction
  - Structural hazard: machine resources restrict the number of possible combinations of instructions in the pipeline
- Hazards produce pipeline stalls
- Instruction scheduling = reorder instructions to avoid hazards
Instruction Scheduling

- Many techniques for instruction scheduling
- List scheduling
  - Build dependence graph
  - Schedule an instruction if all its predecessors have been scheduled
  - Many choices at each step: need heuristics
- Scheduling across basic blocks
  - Move instructions past control flow split/join points
  - Move instruction to successor blocks
  - Move instructions to predecessor blocks

Superscalar, VLIW

- Processor can issue multiple instructions in each cycle
- Need to determine instructions which don’t depend on each other
  - VLIW: programmer/compiler finds independent instructions
  - Superscalar: hardware detects if instructions are independent; but compiler must maximize independent instructions close to each other
- Out-of-order superscalar: burden of instruction scheduling and ILP detection is partially moved to the hardware
- Must detect and reorder instructions to expose fully independent instructions

Symmetric Multiprocessors

- Multiple processing units (as in VLIW)
- ...which execute asynchronously (unlike VLIW)
- Problems:
  - Overhead of creating and starting threads of execution
  - Overhead of synchronizing threads
- Conclusion:
  - Inefficient to execute single instructions in parallel
  - Need coarse grain parallelism (not ILP)
  - Compiler must detect larger pieces of code (not just instructions) which are independent

Memory Hierarchies

- Memory system is hierarchically structured: register, L1 cache, L2 cache, RAM, disk
- Top the hierarchy: faster, but fewer
- Bottom of the hierarchy: more resources, but slower
- Memory wall problem: processor speed increases at a higher rate than memory latency
- Effect: memory accesses have a bigger impact on the program efficiency
- Need compiler optimizations to improve memory system performance (e.g. increase cache hit rate)

Data Dependencies

- Compiler must reason about dependence between instructions
- Three kinds of dependencies:
  - True dependence: \( \frac{s1}{s2} \) \( x = \ldots \)
  - Anti dependence: \( \frac{s1}{s2} \) \( x = \ldots \)
  - Output dependence: \( \frac{s1}{s2} \) \( x = \ldots \)
- Cannot reorder instructions in any of these cases!
Problem: Pointers

- Data dependences not obvious for pointer-based accesses
- Pointer-based loads and stores:
  \[(s1) \quad *p = \ldots\]
  \[(s2) \quad \ldots = *q\]
- \(s1, s2\) may be dependent if \(\text{Ptr}(p) \cap \text{Ptr}(q) \neq \emptyset\)
- Need pointer analysis to determine dependent instructions!
- More precise analyses compute smaller pointer sets, can detect (and parallelize) more independent instructions

Problem: Arrays

- Array accesses also problematic:
  \[
  \begin{align*}
  (s1) \quad a[i] &= \ldots \\
  (s2) \quad \ldots &= a[j]
  \end{align*}
  \]
- \(s1, s2\) may be dependent if \(i=j\) in some execution of the program
- Usually, array elements accessed in nested loops, access expressions are linear functions of the loop indices
- Lot of existing work to formalize the array data dependence problem in this context

Iteration Vectors

- Must reason about nested loops
  
  for \((i=1\text{ to }N)\) 
  for \((i=1\text{ to }N)\) 
  for \((j=1\text{ to }N)\)
  
  \(c[i,j] = a[i][j]\)

- Iteration vector: describes multiple indices in nested loops
- Example: \(i=(1, i, 13)\)
- Lexicographic ordering: iteration \(i=(i_1, \ldots, i_n)\) precedes \(j=(j_1, \ldots, j_n)\) if leftmost non-equal index \(k\) is such that \(i_k < j_k\)

Loop-Carried Dependences

- There is a dependence between statements \(s1\) and \(s2\) if they access the same location
  - In different iterations
  - In the same iteration
- Loop carried dependence = dependence between accesses in different iterations
- Example:
  
  for \((i=1\text{ to }N)\) \{ 
  \[
  \begin{align*}
  a[i+1] &= b[i] \\
  b[i+1] &= a[i]
  \end{align*}
  \}

Dependence Testing

- Goal: determine if there are dependences between array accesses in the same loop nest
  
  for \((i_e=L_\text{e}\text{ to }U_\text{e})\) 
  
  for \((i_k=L_k\text{ to }U_k)\) 
  
  \[
  \begin{align*}
  a[g_i(l_1, \ldots, l_k)] &= \ldots \\
  a[f_i(l_1, \ldots, l_k)] &= \ldots
  \end{align*}
  \]

- There is a dependence between the array accesses if there are two iteration vectors \(i=(i_1, \ldots, i_k)\) and \(j=(j_1, \ldots, j_k)\)
  
  \(f_i(j) = g_i(j)\), for all \(k\)

Dependence Testing

- If \(f_i\) and \(g_i\) are all linear functions, then dependence testing = finding integer solutions of a system of linear equations (which is an NP-complete problem)

- Example:
  
  for \((i=1\text{ to }N)\) \{ 
  \[
  \begin{align*}
  a[3+i, 2*i] &= \ldots \\
  a[j+3, i+j] &= \ldots
  \end{align*}
  \}

- Are there any dependences?
Loop Parallelization

- Can parallelize a loop if there is no loop-carried dependence
- If there are dependences, compiler can perform transformations to expose more parallelism
- Loop distribution:
  
  ```
  for (i=1 to N) {
    a[i+1] = b[i]
    c[i] = a[i]
  }
  ```

  ```
  for (i=1 to N) {
    a[i+1] = b[i]
    c[i] = a[i]
  }
  ```

Loop Parallelization

- Loop interchange:
  ```
  for (i=1 to M) for (j=1 to N)
  a[i,j+1] = b[i,j]

  for (i=1 to N) for (j=1 to M)
  a[i,j+1] = b[i,j]
  ```

- Scalar expansion:
  ```
  for (i=1 to N) {
    tmp = a[i]
    a[i] = b[i]
    b[i] = tmp
  }
  ```

Memory Hierarchy Optimizations

- Many ways to improve memory accesses
- One way is to improve register usage
  - Register allocation targets scalar variables
  - Performs transformations to improve allocation of array elements to registers
- Example:
  ```
  for (i=1 to N) for (j=1 to M)
  a[i] = a[i]+b[j]
  t = a[i]
  ```

Blocking

- Another class of transformations: reorder instructions in different iterations such that program accesses same array elements in iterations close to each other
- Typical example: blocking (also called tiling)
  ```
  for (i=1 to N step B)
  for (j=1 to N)
  for (k = 1 to N)
  c[i,j] += a[i,k]*b[k,j]
  ```

Software Prefetching

- Certain architectures have prefetch instructions which bring data into the cache
- Compiler can insert prefetch instructions in the generated code to improve memory accesses
- Issues:
  - Must accurately determine which memory accesses require prefetching
  - Compiler must insert prefetch instructions in such a way that the required data arrive in the cache neither too late, nor too soon
Predication

- Predicated instructions:
  - Have a condition argument
  - Instruction always executed
  - Result discarded if condition is false
- Predication can significantly reduce number of branch instructions (and the associated pipeline stalls)
- Example (Pentium):

  ```
  if (t1=0)
    t2=t3;
  else
    t4=t5;
  cmp $1, t1
  jne L1
  mov t3, t2
  jmp L2
  L1: mov t5, t4
  L2:
  ```

- Itanium processor: all instructions are predicated
- Can generate predicated code for arbitrary computation
- Example:

  ```
  if (t1=t2)
    t3=t4+t5;
  else
    t6=t7+t8;
  cmp $1, t1
  jne L1
  mov t4, t3
  add t5, t3
  jmp L2
  L1: mov t7, t6
  add t8, t6
  L2:
  ```