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

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 is partially moved to 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
  - 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:
    \[
    \begin{align*}
    s1 & \ x = \ \ldots \\
    s2 & \ \ldots = \ x
    \end{align*}
    \]
  - Anti dependence:
    \[
    \begin{align*}
    s1 & \ \ldots = \ x \\
    s2 & \ x = \ \ldots
    \end{align*}
    \]
  - Output dependence:
    \[
    \begin{align*}
    s1 & \ x = \ \ldots \\
    s2 & \ x = \ \ldots
    \end{align*}
    \]
- Cannot reorder instructions in any of these cases!

Problem: Pointers

- Data dependences not obvious for pointer-based accesses
- Pointer-based loads and stores:
  \[
  \begin{align*}
  s1 & \ x \ p = \ \ldots \\
  s2 & \ \ldots = \ q
  \end{align*}
  \]
- \(s1, s2\) 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

Iteration Vectors

- Must reason about nested loops
  \[
  \begin{align*}
  \text{for } (i=1 \text{ to } N) \\
  \text{for } (j=1 \text{ to } N) \\
  \text{for } (k=1 \text{ to } N) \\
  c[i,j,k] = a[i,j,k] \times b[i,j,k]
  \end{align*}
  \]
- Iteration vector: describes multiple indices in nested loops
- Example: \(i=(1, 2, 3)\)
- Lexicographic ordering: iteration \(i=(j_1, \ldots j_n)\) precedes \(j=(j_1, \ldots j_n)\) if leftmost non-equal index \(k\) is such that \(j_k < j_{-k}\)

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:
  \[
  \begin{align*}
  \text{for } (i=1 \text{ to } N) \{ \\
  a[i+1] = b[i] \\
  b[i+1] = a[i] \\
  \}
  \end{align*}
  \]
Dependence Testing

- Goal: determine if there are dependences between array accesses in the same loop nest

\[
\text{for } (i_1 = L_1 \text{ to } U_1) \\
\vdots \\
\text{for } (i_n = L_n \text{ to } U_n) \\
\ldots = a(g_1(l_1, \ldots, l_n)) \ldots g_n(l_1, \ldots, l_n)) \\
a_1(i_1, \ldots, i_n) \rightarrow \ldots \rightarrow a_n(i_1, \ldots, i_n) = \ldots
\]

- There is a dependence between the array accesses if there are two iteration vectors \(i_1, \ldots, i_n\) \(\text{and } j_1, \ldots, j_m\) where \(f_i(i) = g_j(j)\) for all \(k\)

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

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

Loop Parallelization

- Loop interchange:

\[
\text{for } (i=1 \text{ to } N) \{ \\
\quad a[i, j+1] = b[i, j]
\}
\]

- Scalar expansion:

\[
\text{for } (i=1 \text{ to } N) \{ \\
\quad \text{tmp} = a[i] \\
\quad a[i] = b[i] \\
\quad b[i] = \text{tmp}
\}
\]

Loop Parallelization

- Privatization:

```c
int tmp; 
\text{for } (i=1 \text{ to } N) \{ \\
\quad a[i] = b[i] \\
\quad b[i] = \text{tmp}
\}
```

- Loop fusion:

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

Memory Hierarchy Optimizations

- Many ways to improve memory accesses
- One way is to improve register usage
  - Register allocation targets scalar variables
  - Perform transformations to improve allocation of array elements to registers

- Example:

```c
\text{for } (i=1 \text{ to } N) \{ \\
\quad t = a[i] \text{ for } (i=1 \text{ to } M) \\
\quad a[i] = a[i] + b[j] \\
\quad t = t + b[j] \\
\}
```
### 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)

\[
\text{for } (i=1 \text{ to } N \text{ step } B) \\
\quad \text{for } (j=1 \text{ to } N) \\
\quad \text{for } (k=1 \text{ to } N) \\
\quad c[i,j,k] += a[i,k]*b[j,k]
\]

### 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.
- Predication can significantly reduce number of branch instructions (and the associated pipeline stalls).
- Example (Pentium):

```assembly
if (t1=0) 
  cmp $1, t1 
  jne L1 
  mov $1, t2 
  jmp L2 
L1: mov $1, t4 
L2: 
else 
  t4=t5; 
  cmp $1, t1 
  cmovz $3, t2 
  cmovn $5, t4 
```

### Predication

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

**Example:**

```assembly
if (t1=12) 
  cmp $1, t2 
  jne L1 
  mov $3, t5 
  add $5, $3, $t4 
  jmp L2 
L1: mov $7, $t6 
L2: 
else 
  cmp $1, t2 
  jne L1 
  mov $3, t5 
  add $5, $3, $t4 
  jmp L2 
L1: mov $7, $t6 
L2: 
```