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

These transformations require knowledge about dependencies between program instructions

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 of 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

- The compiler must reason about dependencies between instructions.
- Three kinds of dependencies:
  - True dependence:
    \[
    (s_1) \quad x = \ldots \\
    (s_2) \quad y = x
    \]
  - Anti dependence:
    \[
    (s_1) \quad y = x \\
    (s_2) \quad x = \ldots
    \]
  - Output dependence:
    \[
    (s_1) \quad x = \ldots \\
    (s_2) \quad y = \ldots
    \]
- Cannot reorder instructions in any of these cases!

Problem: Pointers

- Data dependencies not obvious for pointer-based accesses.
- Pointer-based loads and stores:
  \[
  (s_1) \quad *p = \ldots \\
  (s_2) \quad *q = \ldots
  \]
  - \( s_1, s_2 \) may be dependent if \( Pr(p) \cap Pr(q) \neq \emptyset \)
  - 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
  - \( \text{for} (i=1 \text{ to } N) \text{ for } (j=2 \text{ to } N) \text{ for } (k=1 \text{ to } N) \) \( a[i][j][k] = a[i][j][k] \)
  - Iteration vectors describe multiple indices in nested loops
- Example: \( (i, j, k) \)
- Lexicographic ordering: \( (i, j, k) \) precedes \( (i', j', k') \) if for all non-equal index \( k \), \( i < i' \)

Loop-Carried Dependences

- There is a dependence between statements \( s_1 \) and \( s_2 \) if they access the same location
  - In different iterations
  - In the same iteration
- Loop-carried dependence = dependence between accesses in different iterations
- Example:
  - \( \text{for } (i=1 \text{ to } N) \{ \\
  a[i][i+1] = b[i] \\
  b[i][i+1] = a[i] \\
  \} \)
Dependence Testing

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

  \[
  \begin{align*}
  &\text{for } (i_1 = l_1 \text{ to } u_1) \\
  &\quad \cdots
  \\
  &\quad \text{for } (i_n = l_n \text{ to } u_n)
  \\
  &\quad \quad a[i_1 \cdots i_n] \rightarrow f(i_1 \cdots i_n) = \cdots
  \\
  &\quad \quad \cdots
  \\
  &\quad \quad \vdots
  \\
  &\text{there is a dependence between the array accesses if there}
  \\
  &\quad \quad \text{are two iteration vectors } i = (i_1, \ldots, i_n)
  \\
  &\quad \quad \text{and } j = (j_1, \ldots, j_n)
  \\
  &\quad \quad \quad f(i) = g(j), \text{ for all } k
  \end{align*}
  \]

- If \( f_i \) and \( g_i \) are all linear functions, then dependence testing
  
- Example:
  
  \[
  \begin{align*}
  &\text{for } (i = 1 \text{ to } N)
  \\
  &\quad \text{for } (j = 1 \text{ to } N) \{ \text{...} \}
  \\
  &\quad \quad a[i+j, 2j] = \cdots
  \\
  &\quad \quad \vdots
  \\
  &\quad \text{Are there any dependences?}
  \end{align*}
  \]

---

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:

\[
\begin{align*}
&\text{for } (i = 1 \text{ to } N) \\
&\quad a[i+1] = b[i] \\
&\quad c[i] = a[i]
\end{align*}
\]

---

Memory Hierarchy Optimizations

- Many ways to improve memory accesses
- One way is to improve register usage
  
- Example:

\[
\begin{align*}
&\text{for } (i = 1 \text{ to } M) \\
&\quad t = a[i] \\
&\quad a[i] = a[i] + b[i]
\end{align*}
\]
**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)

```c
for (i = 1 to N step 6)
  for (j = 1 to N step 6)
    for (k = 1 to k)  
      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 access.
- 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):

```assembly
if (t1=0)
  jmp L1
else
  mov L1, L2
```

**Predication**

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

```assembly
if (t1=2)
  jmp L1, L2
else
  jmp L1
```

- Add instructions:
  - cmp eq p45 p6 = 1, 12
  - add t3 = 5, 13
  - jmp = 5, 15, 16
  - add 16 = t6, 18
  - add 15 = 15, 16
  - jmp L2

*CS 412/413 Spring 2002 Introduction to Compilers*