#### CS412/413

### Introduction to Compilers Radu Rugina

Lecture 38: Compiling for Modern Architectures 03 May 02

#### 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, whithout 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

CS 412/413 Spring 2002

Introduction to Compilers

## **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

CS 412/413 Spring 2002

Introduction to Compilers

# 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

CS 412/413 Spring 2002

Introduction to Compilers

# 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

CS 412/413 Spring 2002

Introduction to Compilers

#### **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)

CS 412/413 Spring 2002

Introduction to Compilers

1

## **Data Dependencies**

- · Compiler must reason about dependence between instructions
- · Three kinds of dependencies:

- True dependence:

(s1) x = ...

- Anti dependence:

(s1) ... = x

- Output dependence:

(s1) x = ...(s2) x = ..

• Cannot reorder instructions in any of these cases!

CS 412/413 Spring 2002

Introduction to Compilers

#### **Data Dependences**

- In the context of hardware design, dependences are called hazards
  - True dependence = RAW hazard (read after write)
     Anti dependence = WAR hazard (write after read)

  - Output dependence = WAW hazard (write after read)
- A transformation is correct if it preserves all dependences in the program
- · How easy is it to determine dependences?
- Trivial for scalar variables (variables of primitive types)

CS 412/413 Spring 2002

Introduction to Compilers

#### **Problem: Pointers**

- Data dependences not obvious for pointer-based accesses
- · Pointer-based loads and stores:

- s1, s2 may be dependent if Ptr(p) ∩ Ptr(q) ≠ Ø
- Need pointer analysis to determine dependent instructions!
- More precise analyses compute smaller pointer sets, can detect (and parallelize) more independent instructions

CS 412/413 Spring 2002

Introduction to Compilers

### Problem: Arrays

· Array accesses also problematic:

$$(s1) \ a[i] = ...$$
  
 $(s2) \ ... = a[j]$ 

- 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

CS 412/413 Spring 2002

Introduction to Compilers

#### **Iteration Vectors**

· Must reason about nested loops

- Iteration vector: describes multiple indices in nested loops
- Example: i={i1, i2, i3}
- Lexicographic ordering: iteration  $i = \{i_1,...,i_n\}$  precedes  $j = \{j_1,...,j_n\}$  if leftmost non-equal index k is such that  $i_k < j_k$

CS 412/413 Spring 2002

Introduction to Compilers

11

#### **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 to N) {
  a[i+1] = b[i]
 b[i+1] = a[i]
```

CS 412/413 Spring 2002

Introduction to Compilers

12

10

# **Dependence Testing**

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

$$\begin{cases} \text{for } (i_1 \! = \! L_1 \text{ to } U_1) \\ \cdots \\ \text{for } (i_n = L_n \text{ to } U_n) \\ \text{a} [f_1(i_1, \dots, i_n), \, \dots, \, f_m(i_1, \dots, i_n)] = \dots \\ \dots = \text{a} [g_1(i_1, \dots, i_n), \, \dots, \, g_m(i_1, \dots, i_n)] \end{cases}$$

• There is a dependence between the array accesses if there are two iteration vectors  $i = \{i_1,...,i_m\}$  and  $j = \{j_1,...,j_m\}$ 

$$f_k(i) = g_k(j)$$
, for all k

CS 412/413 Spring 2002

Introduction to Compilers

13

15

17

# **Dependence Testing**

- If f<sub>k</sub> and g<sub>k</sub> 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)
  for (j = 1 to N) {
a[3i+5, 2*j] = ...
      ... = a[j+3, i+j]
```

· Are there any dependences?

CS 412/413 Spring 2002

Introduction to Compilers

# **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:

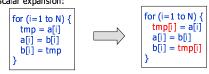
CS 412/413 Spring 2002

Introduction to Compilers

### **Loop Parallelization**

· Loop interchange:

· Scalar expansion:



CS 412/413 Spring 2002

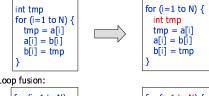
Introduction to Compilers

16

18

# **Loop Parallelization**

• Privatization:



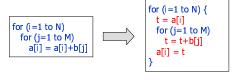
· Loop fusion:

CS 412/413 Spring 2002

Introduction to Compilers

#### **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:



CS 412/413 Spring 2002

Introduction to Compilers

# **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)

```
\begin{array}{c} \text{for (i=1 \ to \ N \ step \ B)} \\ \text{for (j=1 \ to \ N) \ step \ B)} \\ \text{for (i=1 \ to \ N)} \\ \text{for (i=1 \ to \ N)} \\ \text{for (k=1 \ to \ N)} \\ \text{for (k=1 \ to \ N)} \\ \text{c[i,j] += a[i,k]*b[k,j]} \\ \end{array} \begin{array}{c} \text{for (i=i \ to \ l+B-1)} \\ \text{for (kk=k \ to \ k+B-1)} \\ \text{c[ii,jj] += a[i,kk]*b[kk,jj]} \end{array}
```

Introduction to Compilers

19

21

# 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

CS 412/413 Spring 2002

Introduction to Compilers

20

22

#### **Predication**

• Predicated instructions:

CS 412/413 Spring 2002

- 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):

CS 412/413 Spring 2002

Introduction to Compilers

#### **Predication**

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

• Example:

CS 412/413 Spring 2002

Introduction to Compilers