

CS412/413

Introduction to Compilers Radu Rugina

Lecture 38: Compiling for Modern Architectures
07 May 04

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

CS 412/413 Spring 2003

Introduction to Compilers

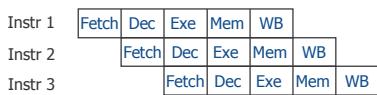
2

Pipelined Machines

- Example pipeline:
 - Fetch
 - Decode
 - Execute
 - Memory access
 - Write back



- Simultaneously execute stages of different instructions



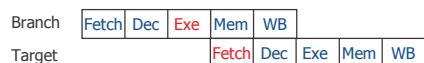
CS 412/413 Spring 2003

Introduction to Compilers

3

Stall the Pipeline

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



- Example 2: load instructions



CS 412/413 Spring 2003

Introduction to Compilers

4

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

CS 412/413 Spring 2003

Introduction to Compilers

5

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

CS 412/413 Spring 2003

Introduction to Compilers

6

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

CS 412/413 Spring 2003

Introduction to Compilers

7

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

CS 412/413 Spring 2003

Introduction to Compilers

8

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

CS 412/413 Spring 2003

Introduction to Compilers

9

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)

CS 412/413 Spring 2003

Introduction to Compilers

10

Data Dependencies

- Compiler must reason about dependence between instructions
- Three kinds of dependencies:
 - **True dependence**:
$$\begin{array}{l} (s1) \quad x = \dots \\ (s2) \quad \dots = x \end{array}$$
 - **Anti dependence**:
$$\begin{array}{l} (s1) \quad \dots = x \\ (s2) \quad x = \dots \end{array}$$
 - **Output dependence**:
$$\begin{array}{l} (s1) \quad x = \dots \\ (s2) \quad x = \dots \end{array}$$
- Cannot reorder instructions in any of these cases!

CS 412/413 Spring 2003

Introduction to Compilers

11

Data Dependencies

- 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)
$$\begin{array}{l} x = \dots \\ \dots = x \end{array}$$

CS 412/413 Spring 2003

Introduction to Compilers

12

Problem: Pointers

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

$$(s1) *p = \dots$$

$$(s2) \dots = *q$$
- s_1, s_2 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

CS 412/413 Spring 2003

Introduction to Compilers

13

Problem: Arrays

- Array accesses also problematic:

$$(s1) a[i] = \dots$$

$$(s2) \dots = a[j]$$
- s_1, s_2 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 2003

Introduction to Compilers

14

Iteration Vectors

- Must reason about nested loops


```
for (i1=1 to N)
  for (i2 = 1 to N)
    for (i3 = 1 to N)
      c[i1,i3] = a[i1,i2]*b[i2,i3]
```
- **Iteration vector:** describes multiple indices in nested loops
- Example: $i=\{i_1, i_2, i_3\}$
- **Lexicographic ordering:** iteration $i=\{i_1, \dots, i_n\}$ precedes $j=\{j_1, \dots, j_n\}$ if leftmost non-equal index k is such that $i_k < j_k$

CS 412/413 Spring 2003

Introduction to Compilers

15

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:


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

CS 412/413 Spring 2003

Introduction to Compilers

16

Dependence Testing

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

$$\begin{aligned} &\text{for } (i_1=L_1 \text{ to } U_1) \\ &\quad \dots \\ &\quad \text{for } (i_n = L_n \text{ to } U_n) \\ &\quad \dots = a[g_1(i_1, \dots, i_n), \dots, g_m(i_1, \dots, i_n)] \\ &\quad a[f_1(i_1, \dots, i_n), \dots, f_m(i_1, \dots, i_n)] = \dots \end{aligned}$$
- There is a dependence between the array accesses if there are two iteration vectors $i=\{i_1, \dots, i_m\}$ and $j=\{j_1, \dots, j_m\}$

$$f_k(i) = g_k(j), \text{ for all } k$$

CS 412/413 Spring 2003

Introduction to Compilers

17

Dependence Testing

- If f_k and g_k 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 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 2003

Introduction to Compilers

18

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]
    for (i=1 to N)
        c[i] = a[i]
```

CS 412/413 Spring 2003

Introduction to Compilers

19

Loop Parallelization

- Loop interchange:

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

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

- Scalar expansion:

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

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

CS 412/413 Spring 2003

Introduction to Compilers

20

Loop Parallelization

- Privatization:

```
int tmp
for (i=1 to N) {
    tmp = a[i]
    a[i] = b[i]
    b[i] = tmp
}
```

```
for (i=1 to N) {
    int tmp
    tmp = a[i]
    a[i] = b[i]
    b[i] = tmp
}
```

- Loop fusion:

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

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

CS 412/413 Spring 2003

Introduction to Compilers

21

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:

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

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

CS 412/413 Spring 2003

Introduction to Compilers

22

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 step B)
        for (k = 1 to N step B)
            for (ii=i to i+B-1)
                for (jj=j to j+B-1)
                    for (kk=k to k+B-1)
                        c[i,j] += a[i,k]*b[k,j]
```

CS 412/413 Spring 2003

Introduction to Compilers

23

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 2003

Introduction to Compilers

24

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

CS 412/413 Spring 2003

Introduction to Compilers

25

Predication

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

- Example:

```
cmp t1,t2
if (t1=t2)
  t3=t4+t5;
else  t6=t7+t8;
      jmp L2
      cmp.eq p4,p5=t1, t2
      <p4> add t3=t4, t5
      <p5> add t6=t7, t8
L1: mov t7, t6
      add t8, t6
L2:
```

CS 412/413 Spring 2003

Introduction to Compilers

26