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, 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
- 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

Fetch	Dec	Exe	Mem	WB

- Simultaneously execute stages of different instructions

Instr 1	Fetch	Dec	Exe	Mem	WB		
Instr 2		Fetch	Dec	Exe	Mem	WB	
Instr 3			Fetch	Dec	Exe	Mem	WB

Stall the Pipeline

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

Branch	Fetch	Dec	Exe	Mem	WB			
Target				Fetch	Dec	Exe	Mem	WB

- Example 2: load instructions

Load	Fetch	Dec	Exe	Mem	WB		
Use			Fetch	Dec	Exe	Mem	WB

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

- 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

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:
    (s1) x = …
    (s2) … = x
  - Anti dependence:
    (s1) … = x
    (s2) x = …
  - Output dependence:
    (s1) x = …
    (s2) x = …
- Cannot reorder instructions in any of these cases!

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)

  x = …
  … = x

Problem: Pointers

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

```
(s1)  *p = …
(s2)   … = *q
```

- s1, s2 may be dependent if Ptr(p) ∩ Ptr(q) ≠ ∅
- 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:

```
(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

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={i1, i2, i3}
- 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 to N) {
  a[i+1] = b[i]
  b[i+1] = a[i]
}
```


Dependence Testing

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

for ($i_1=L_1$ to U_1)
  …
    for ($i_n = L_n$ to U_n)
      … = a[$g_1(i_1,\ldots,i_n)$, …, $g_m(i_1,\ldots,i_n)$]
      a[$f_1(i_1,\ldots,i_n)$, …, $f_m(i_1,\ldots,i_n)$] = …

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

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

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?

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]
```


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]
}
```


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]
}
```


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
}
```


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

```
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[ii,jj] += a[ii,kk]*b[kk,jj]
```


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:
```

```
cmp $1, t1
cmovz t3, t2
cmovn t5, t4
```


Predication

- 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 t1,t2
    jne L1
    mov t4, t3
    add  t5, t3
    jmp L2
L1: mov t7, t6
     add t8, t6
L2:
```

```
cmp.eq p4,p5=t1, t2
<p4> add t3=t4, t5
<p5> add t6=t7, t8
```