Loop Optimizations

Instruction Scheduling
Outline

- Scheduling for loops
- Loop unrolling
- Software pipelining
- Interaction with register allocation
- Hardware vs. Compiler
- Induction Variable Recognition
- loop invariant code motion
Scheduling Loops

• Loop bodies are small
• But, lot of time is spend in loops due to large number of iterations
• Need better ways to schedule loops
Loop Example

- **Machine**
  - One load/store unit
    - load 2 cycles
    - store 2 cycles
  - Two arithmetic units
    - add 2 cycles
    - branch 2 cycles
    - multiply 3 cycles
  - Both units are pipelined (initiate one op each cycle)

- **Source Code**
  
  ```
  for i = 1 to N
  ```
Loop Example

• Source Code

\[
\text{for } i = 1 \text{ to } N \\
A[i] = A[i] \times b
\]

• Assembly Code

\[
\text{loop:} \\
\text{ mov } (\%rdi,\%rax), \%r10 \\
\text{ imul } \%r11, \%r10 \\
\text{ mov } \%r10, (\%rdi,\%rax) \\
\text{ sub } $4, \%rax \\
\text{ jz } \text{ loop}
\]
Loop Example

- **Assembly Code**
  
  ```
  loop:
  mov (%rdi,%rax), %r10
  imul %r11, %r10
  mov %r10, (%rdi,%rax)
  sub $4, %rax
  jz loop
  ```

- **Schedule (9 cycles per iteration)**

<table>
<thead>
<tr>
<th>mov</th>
<th>mov</th>
<th>mov</th>
<th>mov</th>
</tr>
</thead>
<tbody>
<tr>
<td>mov</td>
<td>mov</td>
<td>mov</td>
<td>mov</td>
</tr>
<tr>
<td>imul</td>
<td>imul</td>
<td>imul</td>
<td>imul</td>
</tr>
<tr>
<td>imul</td>
<td>bge</td>
<td>bge</td>
<td></td>
</tr>
<tr>
<td>imul</td>
<td>sub</td>
<td>sub</td>
<td></td>
</tr>
</tbody>
</table>

©MIT Fall 1998
Outline

• Scheduling for loops
• **Loop unrolling**
• Software pipelining
• Interaction with register allocation
• Hardware vs. Compiler
• Induction Variable Recognition
• loop invariant code motion
Loop Unrolling

• Unroll the loop body \( y \) few times
• Pros:
  – Create a much larger basic block for the body
  – Eliminate few loop bounds checks
• Cons:
  – Much larger program
  – Setup code (\# of iterations < unroll factor)
  – beginning and end of the schedule can still have unused slots
Loop Example

loop:
  mov   (%rdi,%rax), %r10
  imul  %r11, %r10
  mov   %r10, (%rdi,%rax)
  sub   $4, %rax
  jz    loop
Loop Example

loop:
  mov (%rdi,%rax), %r10
  imul %r11, %r10
  mov %r10, (%rdi,%rax)
  sub $4, %rax
  mov (%rdi,%rax), %r10
  imul %r11, %r10
  mov %r10, (%rdi,%rax)
  sub $4, %rax
  jz loop
Loop Example

loop:
    mov (%rdi,%rax), %r10
    imul %r11, %r10
    mov %r10, (%rdi,%rax)
    sub $4, %rax
    mov (%rdi,%rax), %r10
    imul %r11, %r10
    mov %r10, (%rdi,%rax)
    sub $4, %rax
    jz loop

• Schedule (8 cycles per iteration)
Loop Unrolling

• Rename registers
  – Use different registers in different iterations
Loop Example

loop:
  mov (%rdi,%rax), %r10
  imul %r11, %r10
  mov %r10, (%rdi,%rax)
  sub $4, %rax
  mov (%rdi,%rax), %r10
  imul %r11, %r10
  mov %r10, (%rdi,%rax)
  sub $4, %rax
  jz loop
Loop Example

```
loop:
    mov   (%rdi,%rax), %r10
    imul  %r11, %r10
    mov   %r10, (%rdi,%rax)
    sub   $4, %rax
    mov   (%rdi,%rax), %rcx
    imul  %r11, %rcx
    mov   %rcx, (%rdi,%rax)
    sub   $4, %rax
    jz    loop
```
Loop Unrolling

• Rename registers
  – Use different registers in different iterations

• Eliminate unnecessary dependencies
  again, use more registers to eliminate true, anti and output dependencies
  – eliminate dependent-chains of calculations when possible
Loop Example

```
loop:
  mov (%rdi, %rax), %r10
  imul %r11, %r10
  mov %r10, (%rdi, %rax)
  sub $4, %rax
  mov (%rdi, %rax), %rcx
  imul %r11, %rcx
  mov %rcx, (%rdi, %rax)
  sub $4, %rax
  jz loop
```
Loop Example

loop:

mov (%rdi, %rax), %r10
imul %r11, %r10
mov %r10, (%rdi, %rax)
sub $8, %rax
mov (%rdi, %rbx), %rcx
imul %r11, %rcx
mov %rcx, (%rdi, %rbx)
sub $8, %rbx
jz loop
Loop Example

```c
loop:
    mov (%rdi,%rax), %r10
    imul %r11, %r10
    mov %r10, (%rdi,%rax)
    sub $8, %rax
    mov (%rdi,%rbx), %rcx
    imul %r11, %rcx
    mov %rcx, (%rdi,%rbx)
    sub $8, %rbx
    jz loop
```

- Schedule (4.5 cycles per iteration)
Outline

• Scheduling for loops
• Loop unrolling
• **Software pipelining**
• Interaction with register allocation
• Hardware vs. Compiler
• loop invariant code motion
• Induction Variable Recognition
Software Pipelining

• Try to overlap multiple iterations so that the slots will be filled
• Find the steady-state window so that:
  – all the instructions of the loop body is executed
  – but from different iterations
Loop Example

• Assembly Code

```assembly
loop:
    mov (%rdi,%rax), %r10
    imul %r11, %r10
    mov %r10, (%rdi,%rax)
    sub $4, %rax
    jz loop
```

• Schedule

```
<table>
<thead>
<tr>
<th>mov</th>
<th>mov</th>
<th>mov</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>mov</td>
<td></td>
</tr>
<tr>
<td></td>
<td>mul</td>
<td>mov</td>
</tr>
<tr>
<td></td>
<td>mul</td>
<td>jz</td>
</tr>
<tr>
<td></td>
<td>mul</td>
<td>jz</td>
</tr>
<tr>
<td></td>
<td>mul</td>
<td></td>
</tr>
<tr>
<td></td>
<td>sub</td>
<td></td>
</tr>
<tr>
<td></td>
<td>sub</td>
<td></td>
</tr>
</tbody>
</table>
```
Loop Example

• Assembly Code

```assembly
loop:
    mov (%rdi,%rax), %r10
    imul %r11, %r10
    mov %r10, (%rdi,%rax)
    sub $4, %rax
    jz loop
```

• Schedule

<table>
<thead>
<tr>
<th>mov</th>
<th>mov1</th>
<th>mov2</th>
<th>mov</th>
<th>mov3</th>
<th>mov4</th>
<th>mov5</th>
<th>mov6</th>
</tr>
</thead>
<tbody>
<tr>
<td>mov</td>
<td>mov1</td>
<td>mov2</td>
<td>mov</td>
<td>mov3</td>
<td>mov1</td>
<td>mov4</td>
<td>mov2</td>
</tr>
<tr>
<td>mul</td>
<td>mul1</td>
<td>mul2</td>
<td>jz</td>
<td>mul3</td>
<td>jz1</td>
<td>mul4</td>
<td>jz2</td>
</tr>
<tr>
<td>mul</td>
<td>mul1</td>
<td>mul2</td>
<td>jz</td>
<td>mul3</td>
<td>jz1</td>
<td>mul4</td>
<td>jz2</td>
</tr>
<tr>
<td>mul</td>
<td>mul1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>sub</td>
<td>sub1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>sub</td>
<td>sub1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

©MIT Fall 1998
Loop Example

- Assembly Code

  ```
  loop: 
  mov  (%rdi,%rax), %r10
  imul %r11, %r10
  mov  %r10, (%rdi,%rax)
  sub $4, %rax
  jz   loop
  ```

- Schedule (2 cycles per iteration)

<table>
<thead>
<tr>
<th>mov4</th>
<th>mov2</th>
</tr>
</thead>
<tbody>
<tr>
<td>mov1</td>
<td>mov4</td>
</tr>
<tr>
<td>mul3</td>
<td>jz1</td>
</tr>
<tr>
<td>jz</td>
<td>mul3</td>
</tr>
<tr>
<td>mul2</td>
<td></td>
</tr>
<tr>
<td>sub1</td>
<td></td>
</tr>
</tbody>
</table>

©MIT Fall 1998
Loop Example

- 4 iterations are overlapped
  - value of %r11 don’t change
  - 4 regs for (%rdi, %rax)
  - each addr. incremented by 4*4
  - 4 regs to keep value %r10
  - Same registers can be reused after 4 of these blocks

generate code for 4 blocks, otherwise need to move

```
loop:
  mov (%rdi, %rax), %r10
  imul %r11, %r10
  mov %r10, (%rdi, %rax)
  sub $4, %rax
  jz loop
```
Software Pipelining

- Optimal use of resources
- Need a lot of registers
  - Values in multiple iterations need to be kept
- Issues in dependencies
  - Executing a store instruction in an iteration before branch instruction is executed for a previous iteration (writing when it should not have)
  - Loads and stores are issued out-of-order (need to figure-out dependencies before doing this)
- Code generation issues
  - Generate pre-amble and post-amble code
  - Multiple blocks so no register copy is needed
Outline

• Scheduling for loops
• Loop unrolling
• Software pipelining
• Interaction with register allocation
• Hardware vs. Compiler
• Induction Variable Recognition
• loop invariant code motion
Register Allocation and Instruction Scheduling

• If register allocation is before instruction scheduling
  – restricts the choices for scheduling
Example

1: mov 4(%rbp), %rax
2: add %rax, %rbx
3: mov 8(%rbp), %rax
4: add %rax, %rcx
Example

1: mov 4(%rbp), %rax
2: add %rax, %rbx
3: mov 8(%rbp), %rax
4: add %rax, %rcx
Example

1: mov \(4(\%rbp), \%rax\)
2: add \(%rax, \%rbx\)
3: mov \(8(\%rbp), \%rax\)
4: add \(%rax, \%rcx\)
Example

1: mov 4(%rbp), %rax
2: add %rax, %rbx
3: mov 8(%rbp), %rax
4: add %rax, %rcx

Anti-dependence
How about a different register?
Example

1: mov 4(%rbp), %rax
2: add %rax, %rbx
3: mov 8(%rbp), %r10
4: add %r10, %rcx

Anti-dependence
How about a different register?
Example

1: mov 4(%rbp), %rax
2: add %rax, %rbx
3: mov 8(%rbp), %r10
4: add %r10, %rcx
Register Allocation and Instruction Scheduling

• If register allocation is before instruction scheduling
  – restricts the choices for scheduling
Register Allocation and Instruction Scheduling

• If register allocation is before instruction scheduling
  – restricts the choices for scheduling

• If instruction scheduling before register allocation
  Register allocation may spill registers
  – Will change the carefully done schedule!!!
Outline

• Scheduling for loops
• Loop unrolling
• Software pipelining
• Interaction with register allocation
• **Hardware vs. Compiler**
• Induction Variable Recognition
• loop invariant code motion
Superscalar: Where have all the transistors gone?

• Out of order execution
  – If an instruction stalls, go beyond that and start executing non-dependent instructions
  – Pros:
    • Hardware scheduling
    • Tolerates unpredictable latencies
  – Cons:
    • Instruction window is small
Superscalar: Where have all the transistors gone?

• Register renaming
  – If there is an anti or output dependency of a register that stalls the pipeline, use a different hardware register
  – Pros:
    • Avoids anti and output dependencies
  – Cons:
    • Cannot do more complex transformations to eliminate dependencies
Hardware vs. Compiler

- In a superscalar, hardware and compiler scheduling can work hand-in-hand
- Hardware can reduce the burden when not predictable by the compiler
- Compiler can still greatly enhance the performance
  - Large instruction window for scheduling
  - Many program transformations that increase parallelism
- Compiler is even more critical when no hardware support
  - VLIW machines (Itanium, DSPs)
Outline

• Scheduling for loops
• Loop unrolling
• Software pipelining
• Interaction with register allocation
• Hardware vs. Compiler
• **Induction Variable Recognition**
• loop invariant code motion
Induction Variables

• Example

\[ i = 200 \]

\[ \text{for } j = 1 \text{ to } 100 \]

\[ a(i) = 0 \]

\[ i = i - 1 \]
Induction Variables

• Example

\[ i = 200 \]
\[
\text{for } j = 1 \text{ to } 100 \\
\quad a(i) = 0 \\
\quad i = i - 1
\]

Basic Induction variable:
\[ J = 1, 2, 3, 4, \ldots \]

Index Variable i in a(i):
\[ I = 200, 199, 198, 197, \ldots \]
Induction Variables

• Example

\[ i = 200 \]

\[ \text{for } j - 1 \text{ to } 100 \]

\[ a(i) = 0 \]

\[ i = i - 1 \]

Basic Induction variable:

\[ J = 1, 2, 3, 4, \ldots \]

Index Variable \( i \) in \( a(i) \):

\[ I = 200, 199, 198, 197, \ldots = 201 - J \]
Induction Variables

• Example

\[ i = 200 \]

\[ \text{for } j = 1 \text{ to } 100 \]

\[ a(201 - j) = 0 \]

\[ i = i - 1 \]

Basic Induction variable:
\[ J = 1, 2, 3, 4, \ldots. \]

Index Variable \( i \) in \( a(i) \):
\[ I = 200, 199, 198, 197, \ldots. = 201 - J \]
Induction Variables

• Example

for \( j - 1 \) to 100

\[ a(201 - j) = 0 \]

Basic Induction variable:

\[ J = 1, 2, 3, 4, \ldots \]

Index Variable \( i \) in \( a(i) \):

\[ I = 200, 199, 198, 197, \ldots = 201 - J \]
What are induction variables?

• x is an induction variable of a loop L if
  – variable changes its value every iteration of the loop
  – the value is a function of number of iterations of the loop

• In compilers this function is normally a linear function
  – Example: for loop index variable j, function c*j + d
What can we do with induction variables?

- Use them to perform strength reduction

- Get rid of them
Classification of induction variables

• Basic induction variables
  – Explicitly modified by the same constant amount once during each iteration of the loop
  – Example: loop index variable

• Dependent induction variables
  – Can be expressed in the form: \( ax + b \) where \( a \) and \( b \) are loop invariant and \( x \) is an induction variable
  – Example: \( 202 - 2\times j \)
Classification of induction variables

• Class of induction variables: All induction variables with same basic variable in their linear equations

• Basis of a class: the basic variable that determines that class
Finding Basic Induction Variables

- Look inside loop nodes
- Find variables whose only modification is of the form \( j = j + d \) where \( d \) is a loop constant
Finding Dependent Induction Variables

- Find all the basic induction variables
- Search variable \( k \) with a single assignment in the loop
- Variable assignments of the form \( k = e \text{ op } j \) or \( k = -j \) where \( j \) is an induction variable and \( e \) is loop invariant
Finding Dependent Induction Variables

- **Example**
  
  ```
  for i = 1 to 100
    j = i*c
    k = j+1
  ```
A special case

t = 202
for j = 1 to 100
    t = t - 2
    a(j) = t
    t = t - 2
    b(j) = t
A special case

\[
\begin{align*}
t &= 202 \\
\text{for } j = 1 \text{ to } 100 \\
& \quad t = t - 2 \\
& \quad a(j) = t \\
& \quad t = t - 2 \\
& \quad b(j) = t
\end{align*}
\]

\[
\begin{align*}
u1 &= 200 \\
u2 &= 202 \\
\text{for } j = 1 \text{ to } 100 \\
u1 &= u1 - 4 \\
a(j) &= u1 \\
u2 &= u2 - 4 \\
b(j) &= u2
\end{align*}
\]
Outline

• Scheduling for loops
• Loop unrolling
• Software pipelining
• Interaction with register allocation
• Hardware vs. Compiler
• Induction Variable Recognition
• **Loop invariant code motion**
Loop Invariant Code Motion

- If a computation produces the same value in every loop iteration, move it out of the loop
Loop Invariant Code Motion

- If a computation produces the same value in every loop iteration, move it out of the loop

```plaintext
for i = 1 to N
    x = x + 1
for j = 1 to N
    a(i,j) = 100*N + 10*i + j + x
```
Loop Invariant Code Motion

• If a computation produces the same value in every loop iteration, move it out of the loop

```plaintext
for i = 1 to N
    x = x + 1
    for j = 1 to N
        a(i,j) = 100*N + 10*i + j + x
```
Loop Invariant Code Motion

• If a computation produces the same value in every loop iteration, move it out of the loop

\[ t1 = 100*N \]

\[ \text{for } i = 1 \text{ to } N \]
\[ x = x + 1 \]

\[ \text{for } j = 1 \text{ to } N \]
\[ a(i,j) = 100*N + 10*i + j + x \]
Loop Invariant Code Motion

• If a computation produces the same value in every loop iteration, move it out of the loop

\[
t_1 = 100*N
\]

for i = 1 to N
    \[x = x + 1\]
for j = 1 to N
    \[a(i,j) = t_1 + 10*i + j + x\]
Loop Invariant Code Motion

• If a computation produces the same value in every loop iteration, move it out of the loop

\[
\begin{align*}
t1 &= 100*N \\
\text{for } i = 1 \text{ to } N & \\
& \quad x = x + 1 \\
& \quad \text{for } j = 1 \text{ to } N \\
& \quad a(i,j) = t1 + 10*i + j + x
\end{align*}
\]
Loop Invariant Code Motion

• If a computation produces the same value in every loop iteration, move it out of the loop

\[ t1 = 100 \times N \]
\[
\text{for } i = 1 \text{ to } N \\
\quad x = x + 1 \\
\text{for } j = 1 \text{ to } N \\
\quad a(i,j) = t1 + 10 \times i + j + x \]
Loop Invariant Code Motion

• If a computation produces the same value in every loop iteration, move it out of the loop

\[
t1 = 100*N
\]

for \( i = 1 \) to \( N \)

\[
x = x + 1
\]

\[
t2 = t1 + 10*i + x
\]

for \( j = 1 \) to \( N \)

\[
a(i,j) = t1 + 10*i + j + x
\]
Loop Invariant Code Motion

- If a computation produces the same value in every loop iteration, move it out of the loop

```plaintext
t1 = 100*N
for i = 1 to N
    x = x + 1
    t2 = t1 + 10*i + x
for j = 1 to N
    a(i,j) = t2 + j
```
6.035 Computer Language Engineering
Spring 2010

For information about citing these materials or our Terms of Use, visit: http://ocw.mit.edu/terms.