Produced Order Queue Compiler Design

m5131145 Masashi Masuda
Supervised by Prof. Ben A. Abderazek
Adaptive Systems Laboratory
Master of Computer Science and Engineering
Graduated School of the University of Aizu
Outline

■ Background
■ Research Motivation and Goal
■ Compiler Framework
  • 1-offset Code Generation
  • Offset Calculation
  • Instruction Scheduling
  • Statement Scheduling
  • Assembly Generation
■ Evaluation
■ Conclusion
■ Future Work
Queue computers are an alternative to conventional register architectures.

Queue computer has several characteristics:
- Instruction implicitly reference their operands
- No false-dependencies
- Short instructions
- Level-order scheduling
  - Expose instruction level parallelism
Produced Order Queue Computing

Equation

\[ x = \frac{a+b}{b-c} \]

Queue Register

<table>
<thead>
<tr>
<th>a</th>
<th>b</th>
<th>c</th>
<th>a+b</th>
<th>b-c</th>
</tr>
</thead>
<tbody>
<tr>
<td>QH-1</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>QH</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>QT</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Program

```
ld a
ld b
ld c
add 0 1
sub -1 0
div 0 1
st x
```
Research Motivation and Goal

■ Motivation
  • Need for a compiler to ease environment for queue research development

■ Goal
  • Development of a produced order queue compiler framework
Queue Compiler Framework

- Reconstruct the tree, remove redundant node and relink the node.
- Calculate the offset value and annotate the offset.
- Translate the tree into dependent language.
- Expose the instruction level parallelism.
- Generate assembly code.
1-offset Code Generation Phase

- Input the Middle-end generated tree
- Traverse the tree in post-order manner
- Eliminate the redundant nodes
  - Insert the **ghost nodes** instead of redundant nodes
- Eliminate the ghost nodes and relink the nodes
  - To protect the rule of 1-offset code generation insert the **duplicate** instruction instead of ghost nodes
Insert Ghost Nodes (1/3)

\[ x = \frac{a \cdot a}{-a + (b - a)} \]

Tree

Tree with ghosts

\[ a \]
\[ a \]
\[ b \]
\[ a \]
Insert Ghost Nodes (2/3)

\[ x = \frac{a \times a}{-a + (b-a)} \]

Tree

Tree with ghosts
Insert Ghost Nodes (3/3)

\[ x = \frac{(a*a)}{(-a+(b-a))} \]

Tree

Tree with ghosts
Eliminate Ghost Nodes (1/2)

Tree with ghosts
Eliminate Ghost Nodes (2/2)

Tree with ghosts

L0

L1

L2

L3

L4

ghost ghost

L0

L1

L2

L3

L4

b a

b a

*-

dup

X

/
Offset Calculation Phase (1/2)

L0 → x
L1 → /
L2 → *
L3 → -
L4 → dup

Tree with offset

<table>
<thead>
<tr>
<th></th>
<th>b</th>
<th></th>
<th>a</th>
<th></th>
</tr>
</thead>
</table>

QH

QT
Offset Calculation Phase (2/2)

Tree with offset

<table>
<thead>
<tr>
<th>b</th>
<th>a</th>
<th>a</th>
</tr>
</thead>
</table>

QH

QT
Instruction Scheduling Phase

Tree with offset

(QMARK_LEVEL)
(PUSH_Q: b)
(PUSH_Q: a)

(QMARK_LEVEL)
(COPY_P_Q (QT, QH)
(SUB_Q (QT, QH+0, QH+1))

(QMARK_LEVEL)
(MUL_Q (QT, QH+0, QH-1))
(SUB_Q (QT, QH+0, QH-2))

(QMARK_LEVEL)
(DIV_Q (QT, QH+0, QH+1))

(QMARK_LEVEL)
(POPQ_Q: x)
Statement Scheduling Phase (1/3)

Program Source

\[ x = a + b \]
\[ y = x + \left( \frac{1}{y} \right) \]
\[ z = 3 \]
Statement Scheduling Phase (2/3)
Statement Scheduling Phase (3/3)
Assembly Generation Phase

Program Source

\( x = (a^2)/(-a+(b-a)) \)

QIR

(QMARK_LEVEL)
(PUSH_Q: b)
(PUSH_Q: a)

(QMARK_LEVEL)
(COPY_Q (QT, QH)
(SUB_Q (QT, QH+0, QH+1))

(QMARK_LEVEL)
(MUL_Q (QT, QH+0, QH-1))
(SUB_Q (QT, QH+0, QH-2))

(QMARK_LEVEL)
(DIV_Q (QT, QH+0, QH+1))

(QMARK_LEVEL)
(POP_Q: x)

Assembly

main:

# PROLOGUE
sub $sp, $sp, 32
st ($sp)28, $ra
st ($sp)24, $fp
move $fp, $sp
sub $sp, $sp, 4
# END OF PROLOGUE

ld qt, ($fp)0
ld qt, ($fp)4
dup qt, qh
sub qt, qh, qh+1
mul qt, qh, qh-1
sub qt, qh, qh-2
div qt, qh, qh+1
st qh, ($fp)8

# EPILOGUE
move $sp, $fp
ld $ra, ($sp)28
ld $fp, ($sp)24
add $sp, $sp, 32
j $ra
Approach for Queue Compiler

- Developed with the GCC 4.0.2
- Constrain all instructions to use at most 1-offset reference
- Calculate offset references for all instructions
- Schedule the program in level-order manner
Evaluation Environment

- **Target architecture:**
  - 32-bits QueueCore with 16-bits wide instruction set
- **4 benchmark programs**
- **Comparison:**
  - Code size
  - Instruction level parallelism
Normalized Code Size

MIPS vs QueueCore

- fft8g
- livermore
- whetstone
- linpack

- 26% compact code than MIPS
Instruction Level Parallelism

- fft8g: 8
- livermore: 4
- whetstone: 4
- linpack: 4

-7% more parallelism than MIPS
Conclusion

- Proposed compiler framework for QueueCore architecture by:
  - Scheduling the program in level-order manner
  - Computing offset references for instructions

- Quality evaluation of generated codes
  - 26% compact codes than MIPS
  - 7% more parallelism than MIPS
Future Work

- Optimization techniques
  - Reduce the code size
  - Control the register file
  - Extract the ILP

- Aware of the available hardware resources
- Generate the more suitable code
Thank you for listening