Near-Optimal Instruction Selection on DAGs

David Ryan Koes
Seth Copen Goldstein

CGO 2008

4/7/2008
Embedded Processors by the Numbers

Microprocessors Sold by Type

Most embedded processors are resource constrained.
Most embedded processors are resource constrained.
Embedded Processors by the Numbers

Microprocessors Sold by Type

- 8-Bit
- 4-Bit
- DSP
- 32-Bit
- 16-Bit

Resource Constraint: Memory

Example: Microchip PIC16F819
- SRAM: 256 bytes
- EEPROM: 256 bytes
- Flash Memory: 3584 bytes

Most embedded processors are resource constrained

Limited instruction memory → code size critical
Architecting for Code Size

4 Byte Instructions
- Ample bits for accessing registers, supporting addressing modes, supporting ISA extensions
- Large code size

2 Byte Instructions
- Small code size, better instruction fetch
- Limited support for addressing modes, accessing registers; instruction count increases

Variable Sized Instructions
- Small code size; full support for addressing modes, accessing registers, ISA extensions
- Increases complexity of decoder, compiler
Complex Instruction Sets →
Complex Compilers

Complex Instruction Sets
– variable length instructions
– full complement of addressing modes
– redundant instructions

x86 Example: \( t+1 \)

- `incl t` 1 byte
- `addl $1,t` 3 bytes
- `leal 1(t),t` 3 bytes

The compiler must select the best instruction based upon its context
Instruction Selection

IR

Compiler Backend

instruction selection

register allocation

ASM
Intermediate Representations

\[(a+8) + (b+8)\];

**Expression Tree**

\[
\begin{array}{c}
\text{+} \\
\text{+} \\
a \rightarrow 8 \\
8 \rightarrow 8 \\
8 \rightarrow b \\
\end{array}
\]

**Expression DAG**

\[
\begin{array}{c}
\text{+} \\
\text{+} \\
a \rightarrow 8 \\
8 \rightarrow b \\
\end{array}
\]

Explicitly encodes redundant computations

Linear IRs such as three address pseudo-assembly can be easily converted to a structural IR.
Instruction Selection = Tiling

add in1, in2 → out
add in1, in2 → out
add in, reg → out
add in, reg → out
move const → out
move const → out
add 8, a → t1
add 8, b → t2
add a, b → t3
add 8, a → t1
add 8, b → t2
add a, b → t3

Architecture specific set of tiles mapping IR to instructions + tiling algorithm = instruction selector

What is the best tiling?
Instruction Selection = Tiling

Architectures specific set of tiles + tiling algorithm = instruction selector

What is the best tiling?
Assign cost to each tile. Minimize cost.
Optimal Tiling on Trees: Bottom Up Dynamic Programming

Given the optimum tiling of each subtrees, generate optimum tiling of the current tree

– consider all tiles for the root of the current tree

– sum cost of best subtree tiles and each tile

– choose tile with minimum total cost
Optimal Tiling on Trees: Bottom Up Dynamic Programming

Given the optimum tiling of each subtrees, generate optimum tiling of the current tree

– consider all tiles for the root of the current tree
– sum cost of best subtree tiles and each tile
– choose tile with minimum total cost
Optimal Tiling on Trees: Bottom Up Dynamic Programming

Given the optimum tiling of each subtrees, generate optimum tiling of the current tree
– consider all tiles for the root of the current tree
– sum cost of best subtree tiles and each tile
– choose tile with minimum total cost
Optimal Tiling on Trees: Bottom Up Dynamic Programming

Given the optimum tiling of each subtrees, generate optimum tiling of the current tree

– consider all tiles for the root of the current tree
– sum cost of best subtree tiles and each tile
– choose tile with minimum total cost
Optimal Tiling on Trees: Bottom Up Dynamic Programming

Given the optimum tiling of each subtrees, generate optimum tiling of the current tree
– consider all tiles for the root of the current tree
– sum cost of best subtree tiles and each tile
– choose tile with minimum total cost
Tiling on Directed Acyclic Graphs

Expression DAGs better representation

– explicitly encode redundant expressions
Tiling on Directed Acyclic Graphs

Expression DAGs better representation
  – explicitly encode redundant expressions

Tiling NP-complete
  – Heuristic: convert DAG into tree

Expression DAG

\[
\begin{align*}
&+ \\
&+ \\
&\text{a} & \text{8} & \text{b} \\
\end{align*}
\]
Turning a DAG into a Tree

This can be done conceptually without modifying the underlying DAG data structure.

This is common subexpression elimination.
# Instruction Selection: State of the Art

<table>
<thead>
<tr>
<th>Method</th>
<th>DAG Support</th>
<th>Fast</th>
<th>Optimal</th>
</tr>
</thead>
<tbody>
<tr>
<td>Dynamic Programming: Trees</td>
<td>N</td>
<td>Y</td>
<td>Y</td>
</tr>
<tr>
<td>Dynamic Programming: DAGS</td>
<td>Y</td>
<td>Y</td>
<td>N</td>
</tr>
<tr>
<td>Greedy Matching</td>
<td>Y</td>
<td>Y</td>
<td>N</td>
</tr>
<tr>
<td>Peephole Matcher</td>
<td>Y</td>
<td>Y</td>
<td>N</td>
</tr>
<tr>
<td>Binate Covering</td>
<td>Y</td>
<td>N</td>
<td>Y</td>
</tr>
</tbody>
</table>
# Instruction Selection: State of the Art

<table>
<thead>
<tr>
<th>Method</th>
<th>DAG Support</th>
<th>Fast</th>
<th>Optimal</th>
</tr>
</thead>
<tbody>
<tr>
<td>Dynamic Programming: Trees</td>
<td>N</td>
<td>Y</td>
<td>Y</td>
</tr>
<tr>
<td>Dynamic Programming: DAGS</td>
<td>Y</td>
<td>Y</td>
<td>N</td>
</tr>
<tr>
<td>Greedy Matching</td>
<td>Y</td>
<td>Y</td>
<td>N</td>
</tr>
<tr>
<td>Peephole Matcher</td>
<td>Y</td>
<td>Y</td>
<td>N</td>
</tr>
<tr>
<td>Binate Covering</td>
<td>Y</td>
<td>N</td>
<td>Y</td>
</tr>
<tr>
<td><strong>NOLTIS</strong></td>
<td>Y</td>
<td>Y</td>
<td>Nearly</td>
</tr>
</tbody>
</table>
**NOLTIS:** Near Optimal Linear Time Instruction Selection

1. Run dynamic programming on DAG
   - implicitly duplicate all shared nodes

2. “Fix” shared nodes
   - mark nodes for which decomposition appears more beneficial

3. Rerun dynamic programming
   - “fixed” nodes must be at root of a tile
Dynamic Programming First Pass

Compute best tiling cost in bottom-up pass

– result is optimal for fully duplicated DAG
– linear time

Obtain tiling in top-down pass

– avoid redundant overlap
– linear time
Fixing Shared Nodes

Would the overall solution be improved if a shared node was decomposed into the root of a tree?

– assuming rest of tiling remains the same, what happens to the cost if we “cut” the tiles overlapping this shared node?

– if cost improves, “fix” the node
Fixing Shared Nodes

Would the overall solution be improved if a shared node was decomposed into the root of a tree?

– assuming rest of tiling remains the same, what happens to the cost if we “cut” the tiles overlapping this shared node?

– if cost improves, “fix” the node
Fixing Shared Nodes

Would the overall solution be improved if a shared node was decomposed into the root of a tree?

– assuming rest of tiling remains the same, what happens to the cost if we “cut” the tiles overlapping this shared node?

– if cost improves, “fix” the node
Fixing Shared Nodes

Would the overall solution be improved if a shared node was decomposed into the root of a tree?

– assuming rest of tiling remains the same, what happens to the cost if we “cut” the tiles overlapping this shared node?

– if cost improves, “fix” the node

\[
\text{cost} = 5 + 5 = 10
\]

\[
\text{cost} = 5 + 1 + 1 = 7
\]
Fixing Shared Nodes

Would the overall solution be improved if a shared node was decomposed into the root of a tree?

– assuming rest of tiling remains the same, what happens to the cost if we “cut” the tiles overlapping this shared node?

– if cost improves, “fix” the node

cost = 5 + 5 = 10

cost = 5 + 1 + 1 = 7
Dynamic Programming Second Pass

Compute best tiling in bottom-up pass
– tiles not allowed to span fixed nodes

Obtain tiling in top-down pass
NOLTIS Implementation

LLVM 2.1 compiler infrastructure targeting x86

Algorithms implemented:

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>default</td>
<td>greedily select largest tile in top-down topological traversal of DAG</td>
</tr>
<tr>
<td>cse-all</td>
<td>decompose entire DAG into trees then perform dynamic programming</td>
</tr>
<tr>
<td>cse-leaves</td>
<td>decompose non-leaf expressions into trees, duplicate leaf expressions and perform dynamic programming</td>
</tr>
<tr>
<td>cse-none</td>
<td>perform dynamic programming on DAG treating shared nodes as duplicated</td>
</tr>
<tr>
<td>NOLTIS</td>
<td>near optimal linear-time instruction selection</td>
</tr>
</tbody>
</table>
Evaluating NOLTIS: Optimality

Compute optimal instruction tiling using integer linear programming and ILOG CPLEX 10.0

Evaluated nearly half a million functions
Evaluating NOLTIS: Optimality

Compute optimal instruction tiling using integer linear programming and ILOG CPLEX 10.0

Evaluated nearly half a million functions

NOLTIS Optimal Functions 99.7%

0.3%
Evaluating NOLTIS: Compile Time

Two pass algorithm results in 2X slowdown
– each linear time pass is ideally a small part of total compile-time
Evaluating NOLTIS: Code Size After Instruction Selection

Percent Improvement Over Default

-4%  -2%  0%  2%  4%  6%

cse-all  cse-leaves  cse-none  NOLTIS
Evaluating NOLTIS: Code Size After Instruction Selection

- Average Code Size Improvement:
  - cse-all: -0.75%
  - cse-leaves: 0.50%
  - cse-none: 1.75%
  - NOLTIS: 3.00%

- Comparison:
  - cse-all vs. cse-leaves: -2.00%
  - cse-leaves vs. cse-none: 1.75%
  - cse-none vs. NOLTIS: 3.00%
Evaluating NOLTIS: Final Code Size

Percent Improvement Over Default

-10% -8% -6% -4% -2% 0% 2% 4% 6% 8% 10%

400.perlbench 401.bzip2 403.gcc 429.mcf 433.milc 444.namd 445.gobmk 450.soplex 453.povray 456.hmmer 458.sjeng 462.libquantum 464.h264ref 470.lbm 471.omnetpp 473.astar 482.sphinx3 483.xalancbmk average

cse-all cse-leaves cse-none NOLTIS
Conclusions

NOLTIS is **fast**, **effective**, and **easy to implement**

Expression DAGs are better than trees

*But*, need to further investigate interaction between instruction selection and register allocation
Conclusions

NOLTIS is fast, effective, and easy to implement.

Expression DAGs are better than trees.

But, need to further investigate interaction between instruction selection and register allocation.

My thesis topic!
Conclusions

NOLTIS is fast, effective, and easy to implement

Expression DAGs are better than trees

But, need to further investigate interaction between instruction selection and register allocation

My thesis topic!

Questions?
http://www.cs.cmu.edu/~dkoes
Questions?
http://www.cs.cmu.edu/~dkoes
Performance Improvement

Here the graph shows the performance improvement for various benchmarks. The x-axis represents the benchmarks, and the y-axis shows the percentage improvement. The bars indicate the improvement under different conditions: 'cse-all', 'cse-none', and 'NOLTIS'.

- **400.perlbench**: 401. bzip2 403.gcc 429.gobmk 456.hmmer 458.sjeng 462.libquantum 464.h264ref 471.omnetpp 473.astar 483.xalancbmk 484.libquantum 485.milc 490.namd 500.soplex 505.sphinx3 average
# Impact of ISA on Code Size

<table>
<thead>
<tr>
<th>Architecture</th>
<th>Instruction Size</th>
<th>Integer Registers</th>
<th>FP Registers</th>
</tr>
</thead>
<tbody>
<tr>
<td>68k (68040)</td>
<td>2-14</td>
<td>16 (8/8)</td>
<td>8</td>
</tr>
<tr>
<td>Alpha</td>
<td>4</td>
<td>32</td>
<td>32</td>
</tr>
<tr>
<td>Arm</td>
<td>4</td>
<td>16</td>
<td>8</td>
</tr>
<tr>
<td>Arm Thumb</td>
<td>2</td>
<td>8*</td>
<td></td>
</tr>
<tr>
<td>Coldfire (V4e)</td>
<td>2, 4, 6</td>
<td>16 (8/8)</td>
<td>8</td>
</tr>
<tr>
<td>MIPS32</td>
<td>4</td>
<td>32</td>
<td>32</td>
</tr>
<tr>
<td>NEC v850</td>
<td>2, 4</td>
<td>32</td>
<td>0</td>
</tr>
<tr>
<td>PowerPC (750)</td>
<td>4</td>
<td>32</td>
<td>32</td>
</tr>
<tr>
<td>s390</td>
<td>2, 4</td>
<td>16</td>
<td>16</td>
</tr>
<tr>
<td>Sparc</td>
<td>4</td>
<td>32</td>
<td>32</td>
</tr>
<tr>
<td>SuperH (SH4)</td>
<td>2</td>
<td>16</td>
<td>16+16</td>
</tr>
<tr>
<td>x86</td>
<td>1-15</td>
<td>8</td>
<td>8</td>
</tr>
</tbody>
</table>

*Additional registers can be accessed inefficiently*
Impact of ISA on Code Size

Results obtained using gcc 4.2.1 compiling the 403.gcc benchmark of SPEC2006 using the -Os option