In Search of Near-Optimal Optimization Phase Orderings

Prasad A. Kulkarni
David B. Whalley
Gary S. Tyson
Jack W. Davidson
Optimization Phase Ordering

- Optimizing compilers apply several optimization phases to improve the performance of applications.
- Optimization phases interact with each other.
- Determining the order of applying optimization phases to obtain the best performance has been a long standing problem in compilers.
Exhaustive Phase Order Evaluation

- Determine the performance of all possible orderings of optimization phases.
- Exhaustive phase order evaluation involves
  - generating all distinct function instances that can be produced by changing optimization phase orderings (CGO ’06)
  - determining the dynamic performance of each distinct function instance for each function
Outline

- Experimental framework
- Exhaustive phase order space enumeration
- Accurately determining dynamic performance
- Correlation between dynamic frequency measures and processor cycles
- Genetic algorithm performance results
- Future work and conclusions
Outline

- Experimental framework
- Exhaustive phase order space enumeration
- Accurately determining dynamic performance
- Correlation between dynamic frequency measures and processor cycles
- Genetic algorithm performance results
- Future work and conclusions
**Experimental Framework**

- We used the VPO compilation system
  - established compiler framework, started development in 1988
  - comparable performance to gcc –O2
- VPO performs all transformations on a single representation (RTLs), so it is possible to perform most phases in an arbitrary order.
- Experiments use all the 15 available optimization phases in VPO.
- Target architecture was the StrongARM SA-100 processor.
Disclaimers

- Instruction scheduling and predication not included.
- VPO does not contain optimization phases normally associated with compiler front ends
  - no memory hierarchy optimizations
  - no inlining or other interprocedural optimizations
- Did not vary how phases are applied.
- Did not include optimizations that require profile data.
Benchmarks

- Used one program from each of the six MiBench categories.
- Total of 111 functions.

<table>
<thead>
<tr>
<th>Category</th>
<th>Program</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>auto</td>
<td>bitcount</td>
<td>test processor bit manipulation abilities</td>
</tr>
<tr>
<td>network</td>
<td>dijkstra</td>
<td>Dijkstra’s shortest path algorithm</td>
</tr>
<tr>
<td>telecomm</td>
<td>fft</td>
<td>fast fourier transform</td>
</tr>
<tr>
<td>consumer</td>
<td>jpeg</td>
<td>image compression / decompression</td>
</tr>
<tr>
<td>security</td>
<td>sha</td>
<td>secure hash algorithm</td>
</tr>
<tr>
<td>office</td>
<td>stringsearch</td>
<td>searches for given words in phrases</td>
</tr>
</tbody>
</table>
Outline

• Experimental framework
• Exhaustive phase order space enumeration
• Accurately determining dynamic performance
• Correlation between dynamic frequency measures and processor cycles
• Genetic algorithm performance results
• Future work and conclusions
Exhaustive Phase Order Enumeration

- Exhaustive enumeration is difficult
  - Compilers typically contain many different optimization phases
  - Optimizations may be successful multiple times for each function / program
- On average, we would need to evaluate $15^{12}$ different phase orders per function.
Naive Optimization Phase Order Space

- All combinations of optimization phase sequences are attempted.
Eliminating Dormant Phases

- Get feedback from the compiler indicating if any transformations were successfully applied in a phase.
Identical / Equivalent Function Instances

- Some optimization phases are independent
  - example: branch chaining and register allocation
- Different phase sequences can produce the same code.
- Two function instances can be identical except for register numbers or basic block numbers used.
Resulting Search Space

- Merging equivalent function instances transforms the tree to a DAG.
Outline

- Experimental framework
- Exhaustive phase order space enumeration
- Accurately determining dynamic performance
- Correlation between dynamic frequency measures and processor cycles
- Genetic algorithm performance results
- Future work and conclusions
Finding the Best Dynamic Function Instance

- On average, there were over 25,000 distinct function instances for each studied function.
- Executing all distinct function instances would be too time consuming.
- Many embedded development environments use simulation instead of direct execution.
- Use data obtained from a few executions to estimate the performance of all remaining function instances.
Quickly Obtaining Dynamic Frequency Measures

- Two different instances of the same function having identical control-flow graphs will execute each block the same number of times.
- Statically estimate the number of cycles required to execute each basic block.
- \[ \text{dynamic frequency measure} = \sum (\text{static cycles} \times \text{block frequency}) \]
## Dynamic Frequency Statistics

<table>
<thead>
<tr>
<th>Function</th>
<th>Instrs.</th>
<th>CF</th>
<th>Leaf</th>
<th>% from optimal</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Batch</td>
</tr>
<tr>
<td>AR_bitbl...(b)</td>
<td>40</td>
<td>88</td>
<td>2</td>
<td>0.00</td>
</tr>
<tr>
<td>BW_bitbl...(b)</td>
<td>56</td>
<td>198</td>
<td>4</td>
<td>0.00</td>
</tr>
<tr>
<td>bit_count(b)</td>
<td>155</td>
<td>72</td>
<td>4</td>
<td>1.40</td>
</tr>
<tr>
<td>bit_shifter(b)</td>
<td>147</td>
<td>82</td>
<td>3</td>
<td>0.00</td>
</tr>
<tr>
<td>bitcount(b)</td>
<td>86</td>
<td>63</td>
<td>10</td>
<td>2.40</td>
</tr>
<tr>
<td>main(b)</td>
<td>92834</td>
<td>45</td>
<td>171</td>
<td>8.33</td>
</tr>
<tr>
<td>ntbl_bitcnt(b)</td>
<td>253</td>
<td>50</td>
<td>20</td>
<td>18.69</td>
</tr>
<tr>
<td>ntbl_bit...(b)</td>
<td>48</td>
<td>33</td>
<td>8</td>
<td>4.09</td>
</tr>
<tr>
<td>dequeue(d)</td>
<td>102</td>
<td>59</td>
<td>14</td>
<td>0.00</td>
</tr>
<tr>
<td>dijkstra(d)</td>
<td>86370</td>
<td>44</td>
<td>1168</td>
<td>0.04</td>
</tr>
<tr>
<td>enqueue(d)</td>
<td>570</td>
<td>40</td>
<td>9</td>
<td>0.20</td>
</tr>
<tr>
<td>main(d)</td>
<td>8566</td>
<td>30</td>
<td>143</td>
<td>4.29</td>
</tr>
</tbody>
</table>

*...* indicates values not provided.

**Average:** 25362.6 27.5 182.8 4.60 47.64
Outline

- Experimental framework
- Exhaustive phase order space enumeration
- Accurately determining dynamic performance
- Correlation between dynamic frequency measures and processor cycles
- Genetic algorithm performance results
- Future work and conclusions
Cycle level Simulation

- *SimpleScalar* toolset includes several different simulators
  - *sim-uop* - functional simulator, relatively fast, provides only dynamic instruction counts
  - *sim-outorder* – cycle accurate simulator, much slower, also model microarchitecture
- Extended *sim-outorder* to switch to a functional mode when not in the function of interest.
Complete Function Correlation
Complete Function Correlation
Leaf Function Correlation

- Leaf function instances are generated from optimization sequences when no additional phases can be successfully applied.
- On average there are only about 183 leaf function instances, as compared to over 25,000 total instances.
- Leaf function instances represent possible code that can be generated from an iterative compiler when the phase order is varied.
Leaf versus Nonleaf Performance

% function instances

perf./optimal perf.

- leaf instance
- non-leaf instance

Florida State University

Languages, Compilers, and Tools for Embedded Systems
Leaf Function Correlation Statistics

- Pearson’s correlation coefficient

\[ P_{corr} = \frac{\Sigma xy - (\Sigma x \Sigma y)/n}{\sqrt{\left(\Sigma x^2 - (\Sigma x)^2/n\right) \times \left(\Sigma y^2 - (\Sigma y)^2/n\right)}} \]

- \( L_{corr} = \) cycle count for best leaf

\( \) cy. cnt for leaf with best dynamic freq count
### Leaf Function Correlation Statistics (cont…)

<table>
<thead>
<tr>
<th>Function</th>
<th>Pcorr</th>
<th>Lcorr 0%</th>
<th>Lcorr 1%</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>Ratio</td>
<td>Leaves</td>
</tr>
<tr>
<td>AR_btbl...(b)</td>
<td>1.00</td>
<td>1.00</td>
<td>1</td>
</tr>
<tr>
<td>BW_btbl...(b)</td>
<td>1.00</td>
<td>1.00</td>
<td>2</td>
</tr>
<tr>
<td>bit_count(b)</td>
<td>1.00</td>
<td>1.00</td>
<td>2</td>
</tr>
<tr>
<td>bit_shifter(b)</td>
<td>1.00</td>
<td>1.00</td>
<td>2</td>
</tr>
<tr>
<td>bitcount(b)</td>
<td>0.89</td>
<td>0.92</td>
<td>1</td>
</tr>
<tr>
<td>main(b)</td>
<td>1.00</td>
<td>1.00</td>
<td>6</td>
</tr>
<tr>
<td>ntbl_bitcnt(b)</td>
<td>1.00</td>
<td>0.95</td>
<td>2</td>
</tr>
<tr>
<td>ntbl_bit...(b)</td>
<td>0.99</td>
<td>1.00</td>
<td>2</td>
</tr>
<tr>
<td>dequeue(d)</td>
<td>0.99</td>
<td>1.00</td>
<td>6</td>
</tr>
<tr>
<td>dijkstra(d)</td>
<td>1.00</td>
<td>0.97</td>
<td>4</td>
</tr>
<tr>
<td>enqueue(d)</td>
<td>1.00</td>
<td>1.00</td>
<td>2</td>
</tr>
<tr>
<td>main(d)</td>
<td>0.98</td>
<td>1.00</td>
<td>4</td>
</tr>
<tr>
<td>....</td>
<td>....</td>
<td>....</td>
<td>....</td>
</tr>
<tr>
<td><strong>average</strong></td>
<td><strong>0.96</strong></td>
<td><strong>0.98</strong></td>
<td><strong>4.38</strong></td>
</tr>
</tbody>
</table>
Outline

- Experimental framework
- Exhaustive phase order space enumeration
- Accurately determining dynamic performance
- Correlation between dynamic frequency measures and processor cycles
- Genetic algorithm performance evaluation
- Future work and conclusions
Genetic Algorithm Properties

- *Genes* are phases, *chromosomes* are sequences.
- There are 20 chromosomes per generation.
- *Crossover* is used to replace 4 poorly performing chromosomes per generation.
- All, except the best sequence and the 4 newly generated sequences are subject to *mutation*.
- We modified our GA to use phase *enabling* and *disabling* relationships during the mutation phase of the GA.
## GA Evaluation Results

<table>
<thead>
<tr>
<th>Function</th>
<th>Original GA</th>
<th>Modified GA</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Opt</td>
<td>Diff</td>
</tr>
<tr>
<td>AR_btbl...(b)</td>
<td>Y</td>
<td>0.00</td>
</tr>
<tr>
<td>BW_btbl...(b)</td>
<td>Y</td>
<td>0.00</td>
</tr>
<tr>
<td>bit_count.(b)</td>
<td>Y</td>
<td>0.00</td>
</tr>
<tr>
<td>bit_shifter(b)</td>
<td>Y</td>
<td>0.00</td>
</tr>
<tr>
<td>bitcount(b)</td>
<td>Y</td>
<td>0.00</td>
</tr>
<tr>
<td>main(b)</td>
<td>Y</td>
<td>0.00</td>
</tr>
<tr>
<td>ntbl_bitcnt(b)</td>
<td>N</td>
<td>6.55</td>
</tr>
<tr>
<td>ntbl_bit...(b)</td>
<td>Y</td>
<td>0.00</td>
</tr>
<tr>
<td>dequeue(d)</td>
<td>Y</td>
<td>0.00</td>
</tr>
<tr>
<td>dijkstra(d)</td>
<td>Y</td>
<td>0.00</td>
</tr>
<tr>
<td>enqueue(d)</td>
<td>Y</td>
<td>0.00</td>
</tr>
<tr>
<td>main(d)</td>
<td>N</td>
<td>3.96</td>
</tr>
<tr>
<td>average</td>
<td>0.87</td>
<td>0.51</td>
</tr>
</tbody>
</table>
Outline

• Experimental framework
• Exhaustive phase order space enumeration
• Accurately determining dynamic performance
• Correlation between dynamic frequency measures and processor cycles
• Genetic algorithm performance evaluation
• Future work and conclusions
Future Work

• Find more equivalent performing function instances to further reduce the phase order space.
• Study effect of limiting scope of phases so that the most deeply nested loops of a function are optimized first.
• Improve conventional compilation speed and performance.
Conclusions

• We demonstrated how a near-optimal phase ordering can be obtained in a short period of time.
• We showed that our measure of *dynamic frequency counts* correlate extremely well to simulator cycles.
• We also showed how the enumerated space can be used to evaluate the effectiveness of heuristic phase order search algorithms.
Optimization Space Properties

- Phase ordering problem can be made more manageable by exploiting certain properties of the optimization search space
  - optimization phases might not apply any transformations
  - many optimization phases are independent
- Thus, many different orderings of optimization phases produce the same code.
Re-stating the Phase Ordering Problem

- Rather than considering all attempted phase sequences, the phase ordering problem can be addressed by enumerating all distinct function instances that can be produced by combination of optimization phases.
- We were able to exhaustively enumerate 109 out of 111 functions, in a few minutes for most.
Detecting Identical Function Instances

- Some optimization phases are independent
  - example: branch chaining & register allocation
- Different phase sequences can produce the same code

\[
\begin{align*}
  r[2] &= 1; \\
  \Rightarrow &\text{ instruction selection} \\
\end{align*}
\]

\[
\begin{align*}
  r[2] &= 1; \\
  \Rightarrow &\text{ constant propagation} \\
  r[2] &= 1; \\
\end{align*}
\]

\[
\begin{align*}
  \Rightarrow &\text{ dead assignment elimination} \\
\end{align*}
\]
VPO Optimization Phases

- Register assignment (assigning pseudo registers to hardware registers) is implicitly performed before the first phase that requires it.
- Some phases are applied after the sequence
  - fixing the entry and exit of the function to manage the run-time stack
  - exploiting predication on the ARM
  - performing instruction scheduling
### VPO Optimization Phases

<table>
<thead>
<tr>
<th>ID</th>
<th>Optimization Phase</th>
<th>ID</th>
<th>Optimization Phase</th>
</tr>
</thead>
<tbody>
<tr>
<td>b</td>
<td>branch chaining</td>
<td>l</td>
<td>loop transformations</td>
</tr>
<tr>
<td>c</td>
<td>common subexpr. elim.</td>
<td>n</td>
<td>code abstraction</td>
</tr>
<tr>
<td>d</td>
<td>remv. unreachable code</td>
<td>o</td>
<td>eval. order determin.</td>
</tr>
<tr>
<td>g</td>
<td>loop unrolling</td>
<td>q</td>
<td>strength reduction</td>
</tr>
<tr>
<td>h</td>
<td>dead assignment elim.</td>
<td>r</td>
<td>reverse branches</td>
</tr>
<tr>
<td>i</td>
<td>block reordering</td>
<td>s</td>
<td>instruction selection</td>
</tr>
<tr>
<td>j</td>
<td>minimize loop jumps</td>
<td>u</td>
<td>remv. useless jumps</td>
</tr>
<tr>
<td>k</td>
<td>register allocation</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Eliminating Consecutively Applied Phases

- A phase just applied in our compiler cannot be immediately active again.
Detecting Equivalent Function Instances

sum = 0;
for (i = 0; i < 1000; i++)
sum += a[i];

Source Code

<table>
<thead>
<tr>
<th>L3</th>
<th>L5</th>
<th>L01</th>
</tr>
</thead>
<tbody>
<tr>
<td>( r[10] = 0; )</td>
<td>( r[11] = 0; )</td>
<td>( r[32] = 0; )</td>
</tr>
<tr>
<td>( r[12] = HI[a]; )</td>
<td>( r[10] = HI[a]; )</td>
<td>( r[33] = HI[a]; )</td>
</tr>
<tr>
<td>( r[12] = r[12] + LO[a]; )</td>
<td>( r[10] = r[10] + LO[a]; )</td>
<td>( r[33] = r[33] + LO[a]; )</td>
</tr>
<tr>
<td>( r[1] = r[12]; )</td>
<td>( r[1] = r[10]; )</td>
<td>( r[34] = r[33]; )</td>
</tr>
<tr>
<td>( r[9] = 4000 + r[12]; )</td>
<td>( r[9] = 4000 + r[10]; )</td>
<td>( r[35] = 4000 + r[33]; )</td>
</tr>
<tr>
<td>( L3 )</td>
<td>( L5 )</td>
<td>( L01 )</td>
</tr>
<tr>
<td>( r[8] = M[r[1]]; )</td>
<td>( r[8] = M[r[1]]; )</td>
<td>( r[36] = M[r[34]]; )</td>
</tr>
<tr>
<td>IC = r[1] ? r[9];</td>
<td>IC = r[1] ? r[9];</td>
<td>IC = r[34] ? r[35];</td>
</tr>
<tr>
<td>PC = IC &lt; 0, L3;</td>
<td>PC = IC &lt; 0, L5;</td>
<td>PC = IC &lt; 0, L01;</td>
</tr>
</tbody>
</table>
Case when No Leaf is Optimal

(a) Before Code Motion

PC = L1;

L2:
  r[0]=r[4]+28;
  r[1]=64;
  call reverse;
  r[0]=r[4]+28;
  r[1]=r[5];
  call memcpy;

L1:
  PC=c[0]<0, L2;

(b) After Code Motion

PC = L1;

L2:
  r[0]=r[7];
  r[1]=64;
  call reverse;
  r[0]=r[7];
  r[1]=r[5];
  call memcpy;

L1:
  PC=c[0]<0, L2;