Improving Adaptive Compilation

Jeff Sandoval
Keith D. Cooper

2006 LACSI Symposium
October 18, 2006

Department of Computer Science
Rice University
Optimizing Compilers

- Fixed sequence of optimization passes
  - Works well in most cases
  - Usually sub-optimal for individual programs

- Customized sequence of optimization passes
  - Works well on one program by exploiting program specific characteristics
  - Often does not work well on other programs
• Despite successful search techniques, this approach is still very expensive

• Two main approaches for reducing compile time cost
  – Prune search space [Kulkarni et al., Guo et al., O'Boyle et al.]
  – Reduce evaluation overhead [Cooper et. al]

• This work changes the search space by adding a new transformation, loop-unroll
Motivation

• Strange behavior: adaptive compiler selects loop-peel several times consecutively

Top 5 sequences for adpcm_coder:
pppppczpcc
pppppcclpc
pppppplpd
pppppplpg
ppppppclpm

c Constant Propagation
d Dead Code Elimination
g Global Value Numbering
l Partial Redundancy Elimination (PRE)
m Renaming (for PRE and LCM)
n Control Flow Cleanup
o Peephole Optimization
p Loop Peeling
r Algebraic Reassociation
s Register Coalescing
t Operator Strength Reduction
u Local Value Numbering
v SCC-based Value Numbering
x Dominator-based Value Numbering
y Extended-block Value Numbering
z Lazy Code Motion (LCM)
Motivation

Average Optimization Pass Frequency in Best 1% of Sequences

![Graph showing frequency of optimization passes for different letters (c, d, g, l, m, n, o, p, r, s, t, u, v, x, y, z).]
Reducing Loop Overhead

- Loop-peel reduces loop overhead for a single iteration
- Loop-unroll reduces loop overhead for every iteration

```c
/* Original Loop */
for(i = 0; i < 100; i++)
a[i] = a[i] + b[i];

/* Peeled Loop */
a[0] = a[0] + b[0];
for(i = 1; i < 100; i++)
a[i] = a[i] + b[i];

/* Unrolled Loop */
for(i = 0; i < 100; i += 2) {
    a[i] = a[i] + b[i];
    a[i+1] = a[i+1] + b[i+1];
}
```

- Adding loop-unroll to the adaptive compiler increases size of the search space
  - Can we find better solutions?
  - How long will the search take?
Experimental Setup

• Simulate benchmarks in the ILOC instruction set
  – RISC-like representation
  – Allows accurate instruction count
  – Focuses on machine independent transformations

• Examine loop-unrolling in two contexts
  – Fixed sequence: loop-unroll x4 followed by standard optimization sequence
  – Adaptive compiler: 16 transformation (including loop-unroll) hill climber

• Focus on two metrics for the adaptive compiler
  – Effectiveness: indicates the quality of the sequence discovered, measured by dynamic instruction count of output code
  – Efficiency: measures the effectiveness achieved per unit work
Loop Unrolling in a Fixed Sequence

![Bar chart showing dynamic instruction count normalized to standard sequence for various benchmarks.](chart.png)
Loop Unrolling in a Hill Climber

- The hill climber finds better quality sequences with loop-unrolling
Search Efficiency: Improvement

Average Hill Climber Effectiveness for adpcm_coder

- Without Unrolling
- With Unrolling

Dynamic Instruction Count (Normalized to Best)

Search Evaluations
Search Efficiency: Degradation

Average Hill Climber Effectiveness for solve

- Blue line: Without Unrolling
- Red line: With Unrolling

Dynamic Instruction Count (Normalized to Best)

Search Evaluations
Search Efficiency: Learning Overhead

Average Hill Climber Effectiveness for rkf45

- Without Unrolling
- With Unrolling

Dynamic Instruction Count (Normalized to Best) vs. Search Evaluations
Return to the Motivation

Average Optimization Pass Frequency in Best 1% of Sequences

- Blue: Without Unrolling
- Red: With Unrolling

Frequency

Optimization Pass
Conclusions

• Improved adaptive compiler by targeting available opportunities
  – Added loop-unroller to reduce loop overhead
• Changed the search space, allowing the hill climber to perform better than on the old space
  – On average, the best sequences found produced code that executed 10% fewer instructions
  – In all but two cases, the hill climber was more efficient, performing less work to find solutions of equal quality
• Showed that adaptive compilation effectiveness and efficiency depends on the available optimizations