Adaptive Inlining

Keith D. Cooper    Timothy J. Harvey    Todd Waterman

Department of Computer Science    Rice University    Houston, TX
Adaptive Compilation

- Iterative process
  - Compile with initial set of decisions
  - Evaluate code
  - Use previous decisions and results to guide new decisions
  - Repeat until...
Prior Work on Adaptive Compilation

• **Big focus on order of optimizations**
  — Intermediate optimizations can be applied in any possible order
  — Historically, the compiler writer selects a single, universal sequence of optimizations
  — Different sequences perform better for different programs
  — Use adaptive techniques to find a good sequence for a specific program (LACSI ’04)
Single-optimization adaptive techniques

• Can we use adaptive techniques to improve the performance of individual optimizations?
  — Need “flexible” optimizations
    - Expose a variety of decisions that impact the optimization’s performance
    - Different sets of decisions work better for different programs
  — Need to understand how to explore the space of decisions

• We examine procedure inlining
  — A poorly understood, complex, problem
    - Many different approaches and heuristics have been used
    - Mixed success that varies by input program
    - Potential for major improvements
Procedure Inlining

• Procedure inlining replaces a call site with the body of the procedure
• Wide variety of effects
  — Eliminates call overhead
  — Increases program size
  — Enables other optimizations
  — Changes register allocations
  — Cache performance
• Decisions are not independent

```c
int f() {
    int a = g(1);
    return a;
}

int g(x) {
    if (x == 0)
        return 0;
    else
        return x+1;
}

int f() {
    int a;
    if (1 == 0)
        a = 0;
    else
        a = 1 + 1;
    return a;
}

int g(x) {
    if (x == 0)
        return 0;
    else
        return x+1;
}

int f() {
    return 2;
}
```
Building the Inliner

- Built on top of Davidson and Holler's INLINER tool
  - Source-to-source C inliner
  - Need to modernize for ANSI C

- Parameterize the inliner
  - Find parameters that can positively impact inlining decisions
  - Group parameters together in condition strings
    - Specified at the command line
    - Use CNF
    - Example: `inliner -c "sc < 25 | ln > 0, sc < 100" foo.c`
Condition String Properties

- Statement count
- Loop nesting depth
- Static call count
  - If the static call count is one, inlining can be performed without increasing the code size
- Constant parameter count
  - Used to estimate the potential for enhancing optimizations
- Calls in the procedure
  - Introduced as a method for finding leaf procedures
- Dynamic call count
Preliminary Searches

- Perform one and two dimensional sweeps of the space
  - Get an idea of what the space looks like
  - Determine which parameters have a positive impact on performance
Results of Preliminary Searches

- Provided insight into the value of certain parameters
  - Calls in a procedure (CLC) and constant parameter count (CPC) proved very effective
  - Parameter count (PC) had little effect

- A hill climber is a good method for exploring the space
  - Relatively smooth search space
  - Was effective for optimization ordering work

- Bad sets of inlining decisions are expensive
  - Example: “clc < 3” provides great performance for Vortex, “clc < 4” exhausts memory during compilation
  - Unavoidable to some extent
Constraining the Search Space

• Search space is immense
  — Many possible combinations of parameters
  — A large number of possible values for certain parameters

• Need to constrain the search space
  — Eliminate obviously poor sets of decisions
    - Often the most time consuming as well
  — Make the search algorithm more efficient

• Fix the form of the condition string:
  “sc < A | sc < B, Ind > 0 | sc < C, scc = 1 | clc < D | cpc > E, sc < F | dcc > G”
  — Constrains the number of combinations and eliminates foolish possibilities
  — Still need to constrain the values of the individual parameters
Constraining Parameters

• Need reasonable minimum and maximum values
  – Easy case: parameters that have a limited range regardless of program (CPC & CLC)
  – Hard case: parameters that need to vary based on the program
    - General idea
      minimum value: no inlining will occur from the parameter
      maximum value: maximal amount of inlining
    - Statement count
      Set minimum value to 0
      Begin with a small value and increase until an unreasonable amount of inlining occurs for maximum value
    - Dynamic call count
      Set maximum value to the highest observed DCC
      Repeatedly decrease to find minimum value
Constraining Parameters

• Need reasonable granularity for the range of parameters
  — Some parameters can have extremely large ranges
  — Linear distribution of values doesn’t work well
    - Not uniform spaces
    - Want a smaller step value for smaller values
  — Purely quadratic has problems as well
    - Values too close at the low end

• Have a fixed number of ordinals for large ranges (20)

• Use quadratic equation with linear term to get value
Building the Hill Climber

- Each possible condition string is a point in the search space
  - Neighbors are strings with a single parameter increased or decreased by one ordinal (14 neighbors for each point)

- Hill climber descends to a local minimum
  - Examine neighbors until a better point is found and descend to that point
  - Evaluate points using a single execution of the code
    - Experiments show a single execution to be sufficient
  - Cutoff bad searches

- Perform multiple descents from random start points

- Try using limited patience
  - Only explore a percentage of the neighbors before giving up
  - Tradeoff the quality of a descent for the ability to perform more
Selection of Start Points

• **Problem:** Many possible start points are unsuitable
  — Massive amounts of inlining occur
  — Parameter bounds are designed to be individually reasonable but the combination can be unreasonable

• **Solution:** Limit start points to a subset of the total space
  — Require start points to have the property:
    \[ p_a^2 + p_b^2 + ... < (\text{max. ord})^2 \]
  — Much more successful in finding start points
  — Creates tendency to go from less inlining to more inlining
    - Faster searches
    - Prioritizes solutions with less code growth
Experimental Setup

• Used a 1 GHz G4 PowerMac
  — Running OS X server
  — 256kB L2 cache, 2MB L3 cache

• Tested using several SpecINT C benchmarks
  — Vortex - object oriented database program
  — Bzip2, Gzip - compression programs
  — Parser - recursive descent parser
  — Mcf - vehicle scheduling
Changing Patience

- Comparison using HC with varying levels of patience on Vortex
Normalized Execution Time

- Comparison of HC with 5-descents and 100% patience against no inlining and the GCC inliner
### Descents Chosen

- **Percentage each neighbor was chosen when a downward step was found by the hill climber**

<table>
<thead>
<tr>
<th>Step</th>
<th>Vortex</th>
<th>Parser</th>
<th>Bzip2</th>
<th>Gzip</th>
<th>Mcf</th>
</tr>
</thead>
<tbody>
<tr>
<td>SC Increased</td>
<td>7.88%</td>
<td>11.17%</td>
<td>15.74%</td>
<td>16.56%</td>
<td>9.30%</td>
</tr>
<tr>
<td>SC Decreased</td>
<td>9.07%</td>
<td>19.68%</td>
<td>21.30%</td>
<td>15.95%</td>
<td>22.10%</td>
</tr>
<tr>
<td>Loop SC Increased</td>
<td>8.11%</td>
<td>10.64%</td>
<td>0.93%</td>
<td>2.45%</td>
<td>1.16%</td>
</tr>
<tr>
<td>Loop SC Decreased</td>
<td>8.35%</td>
<td>8.51%</td>
<td>1.85%</td>
<td>5.52%</td>
<td>3.49%</td>
</tr>
<tr>
<td>SCC SC Increased</td>
<td>13.60%</td>
<td>10.11%</td>
<td>23.15%</td>
<td>6.75%</td>
<td>20.93%</td>
</tr>
<tr>
<td>SCC SC Decreased</td>
<td>5.25%</td>
<td>8.51%</td>
<td>12.04%</td>
<td>7.36%</td>
<td>34.88%</td>
</tr>
<tr>
<td>CLC Increased</td>
<td>3.82%</td>
<td>4.26%</td>
<td>8.33%</td>
<td>6.13%</td>
<td>2.33%</td>
</tr>
<tr>
<td>CLC Decreased</td>
<td>3.82%</td>
<td>2.12%</td>
<td>2.78%</td>
<td>0.61%</td>
<td>2.33%</td>
</tr>
<tr>
<td>CPC Increased</td>
<td>3.58%</td>
<td>5.32%</td>
<td>2.78%</td>
<td>6.13%</td>
<td>2.33%</td>
</tr>
<tr>
<td>CPC Decreased</td>
<td>4.06%</td>
<td>1.59%</td>
<td>4.63%</td>
<td>14.72%</td>
<td>1.16%</td>
</tr>
<tr>
<td>CPC SC Increased</td>
<td>6.44%</td>
<td>3.19%</td>
<td>0.00%</td>
<td>4.91%</td>
<td>0.00%</td>
</tr>
<tr>
<td>CPC SC Decreased</td>
<td>3.34%</td>
<td>0.53%</td>
<td>0.00%</td>
<td>2.45%</td>
<td>0.00%</td>
</tr>
<tr>
<td>DCC Increased</td>
<td>18.85%</td>
<td>4.26%</td>
<td>1.85%</td>
<td>0.61%</td>
<td>0.00%</td>
</tr>
<tr>
<td>DCC Decreased</td>
<td>3.82%</td>
<td>10.11%</td>
<td>4.63%</td>
<td>9.82%</td>
<td>0.00%</td>
</tr>
</tbody>
</table>
Conclusions

• Adaptive Inlining
  — Gets consistent improvement across programs
    - Magnitude limited by opportunity
  — Static techniques cannot compete
    - Results suggest against a universal static solution

• Adaptive Compilation
  — Design the optimization to expose opportunity for adaptivity
  — Understand the search space
  — Build the adaptive system accordingly