Continuous Compilation: A New Approach to Aggressive and Adaptive Code Transformation

Bruce R. Childers
University of Pittsburgh
Pittsburgh, Pennsylvania, USA
childers@cs.pitt.edu
http://www.cs.pitt.edu/coco

This work is a collaboration of many people!
Sponsored by Next Generation Systems, National Science Foundation.

Code Optimization

- Sophisticated algorithms exist for many optimizations that do quite well
- We are at the point of diminishing returns in applying optimizations - small gains are considered good
- The challenge is to go beyond current optimization improvements

Continuous Compilation

- A new approach: Apply optimizations both statically at compile-time and dynamically at run-time with static planning
- Plan for both static and dynamic optimizations
  - Understand interactions of existing optimizations
  - Efficacy of both static and dynamic optimizations
- Determine what optimizations to apply, where to apply them, the order in which to apply them, and their parameters

Outline

- Introduction: Continuous Compilation
- CoCo Framework
  - Profit-driven Optimization
    - Loop optimizations
    - Scalar optimizations
  - CoCo Run-time System
    - Software dynamic translation
    - Program instrumentation and optimization
- Summary

Continuous Compilation

Static Compilation (using profiles, estimation models)
Phase 1
Program Execution and Dynamic Optimization (using monitor and optimization plans)
Phase 2
Offline Adaptation and Retargetment of Monitor and Optimization Plans
Phase 3
Recompilation and Regeneration of Monitor and Optimization Plans
Phase 4

At time passes, the continuous compiler moves through phases, possibly revisiting earlier ones.
Target applications: Long running programs that have different phases of execution.

CoCo Framework
Motivation

- Performance problems of the optimizations
  - Optimizations may degrade performance in some circumstances
  - Optimizations interact with one another by enabling and disabling other optimizations
- Optimizations often applied in an ad hoc fashion
  - Simple heuristic: always apply if applicable
  - Predetermined order to apply optimizations
  - Fixed configurations of optimizations

Challenges

- Performance varies widely, based on
  - Code context (e.g., loop trip count)
  - Configuration of optimizations (e.g., loop unrolling factor)
  - Machine configurations (e.g., cache configuration)
  - The order of optimizations
- Many resources impact overall performance
  - Cache configuration
  - Instruction scheduling rules
  - Register numbers and types

Our Approach

- Build and develop analytic models to predict when to apply an optimization, without actually applying the optimization
  - Need models of particular optimizations
  - Need models of the code
  - Need models of the resources that are effected
- Based on analytic models, make decisions about what optimizations to apply
  - We don’t need accurate models, just the trend needs to be accurate enough to do the estimates

Framework for Predicting Optimizations

FPO: a Framework, consisting of models, for Predicting the impact of Optimizations

Consider both loop and scalar optimizations (e.g., PRE)

Loop Code Model

\[
\text{lb} = A \times \text{I} + C
\]

For loops and cache: 1. loop header, 2. array references, 3. reference sequence
Loop Optimization Model

Model how optimization transforms code model

For loops and cache: sequence of functions that effect aspects of the loop’s representation (code model)

E.g., Interchange
\[ f \circ \cdots \circ g(\mathcal{R}) \]
\[ g(\mathcal{R}) = \{v \in \mathcal{R} : h(v)\} \]
\[ h(v) = (l(A), C) \]
\[ l(A) = A[i][i] \leftrightarrow A[i][j] \]

Loop Resource Model (Cache)

How the code model effects machine resources

For loops and cache: with code model, how the array reference pattern effects both cache misses and hits

1. Group references by temporal or spatial reuse and compute conflicts
2. Pick representative ref from each group
3. Compute misses for each representative by reuse and conflicts
Simplified Ghosh model

Loop Optimization Prediction

Compare transformed & non-transformed code

For loops and cache: Before & after loop optimization and whether optimization had reduction/increase in cache misses

Prediction of the benefit or penalty is the difference between unoptimized and optimized

Applications of FPO

Selectively apply optimizations
Choose most beneficial optimizations
Search best order or configuration
Combine optimizations

Experiments

- Benchmarks (MediaBench, DSPStone, and others)
- Machine model
  - SimpleScalar microarchitecture simulator
  - In-order, single issue, non-blocking cache
  - 1KB cache, direct mapped and 32B line size
  - Similar to ARM's 94x, IBM Power PC 405
- Usefulness of FPO for loop optimizations
- Prediction accuracy of FPO for loop optimizations

Ad hoc: Always Applying

- Interchange
- Tiling
- Reversal
**Selectively Applying Loop Optimizations**

-50%
-25%
0%
25%
50%

alv (100)
lgsi (98)
lgsi (128)
smsi (124)
tfsi (42)
biquad_N (90)
gdevcdj (100)
pegwit (100)

**Choose Most Beneficial Loop Optimization**

0%
20%
40%
60%
80%
100%

alv irkernel lgsi gdevcdj smsi srsi biquad_N tfsi tms pegwit

**Loop Opt. Prediction Accuracy**

- Accurate trend
  - Whether an optimization is beneficial or not
- Correct prediction
  - When prediction matches actual execution behavior
- Prediction accuracy
  - Single loop nest: varying trip count
  - Multiple loop nests: the number of loop nests

**FPO: Scalar Optimizations**

- Transformations that operate on scalar code
  - E.g., constant propagation, dead code elimination, partial redundancy elimination
- Can have several impacts
  - Reduce amount of computation
  - Change register pressure (for the better or for the worse!)
  - May change memory referencing pattern and cache behavior
- FPO (initially) considers
  - Affect on computation
  - How register pressure helps or hurts spills and reloads

**Impact of PRE**

---

**Framework for Predicting Optimizations**

("Prediction Engine")

Resource Model (regs)

OFT Model for register allocation

# of load & stores increased or decreased
Using a Heuristic to Make Decisions

<table>
<thead>
<tr>
<th>Bench</th>
<th>Heuristic-driven PRE</th>
<th>Heuristic-driven LCIM</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>H 0</td>
<td>H 4</td>
</tr>
<tr>
<td>gzip</td>
<td>3.50</td>
<td>3.75</td>
</tr>
<tr>
<td>vpr</td>
<td>1.22</td>
<td>0.75</td>
</tr>
<tr>
<td>mcf</td>
<td>2.37</td>
<td>2.35</td>
</tr>
<tr>
<td>parser</td>
<td>1.25</td>
<td>1.50</td>
</tr>
<tr>
<td>vortex</td>
<td>4.73</td>
<td>5.25</td>
</tr>
<tr>
<td>bzip2</td>
<td>7.33</td>
<td>7.52</td>
</tr>
<tr>
<td>twolf</td>
<td>1.07</td>
<td>0.88</td>
</tr>
</tbody>
</table>

Appaches for PRE

Scalar Opt. Prediction Accuracy

<table>
<thead>
<tr>
<th>Benchmark</th>
<th>PREs</th>
<th>Correct</th>
<th>%Accuracy</th>
</tr>
</thead>
<tbody>
<tr>
<td>gzip</td>
<td>48</td>
<td>43</td>
<td>89.58</td>
</tr>
<tr>
<td>vpr</td>
<td>303</td>
<td>291</td>
<td>96.04</td>
</tr>
<tr>
<td>mcf</td>
<td>51</td>
<td>44</td>
<td>86.27</td>
</tr>
<tr>
<td>parser</td>
<td>293</td>
<td>210</td>
<td>87.87</td>
</tr>
<tr>
<td>vortex</td>
<td>530</td>
<td>431</td>
<td>81.13</td>
</tr>
<tr>
<td>bzip2</td>
<td>56</td>
<td>44</td>
<td>78.57</td>
</tr>
<tr>
<td>twolf</td>
<td>475</td>
<td>433</td>
<td>91.12</td>
</tr>
<tr>
<td>Average</td>
<td></td>
<td></td>
<td>87.23</td>
</tr>
</tbody>
</table>

CoCo Run-Time System

- Based on Software Dynamic Translation
- Layer of software between application binary and the OS/CPU.
- Application’s instructions are examined and modified before being executed on the CPU.
- Uses include binary translation, dynamic optimization, & others

CoCo Framework

CoCo Run-Time System

- Strata SDT developed at UVA & Pitt
- Provides basic functionality for run-time optimization
  - Memory management
  - Caching
  - Code analysis
  - Overhead reduction
  - Program instrumentation
- Includes target Interface
  - Handles all interactions between VM, application binary, and target OS/CPU

Current targets: MIPS/Irix, SPARC/Solaris, x86/Linux, MIPS/Playstation 2
Software Dynamic Translation

Translation and Execution
1. Fetch gets next instruction
2. Decode classifies instruction
3. Translate performs any necessary modification and rewriting and write translated instruction into fragment cache
4. Terminate fragment on control transfer and execute fragment

Overhead Reduction in Strata

- Improve run-time performance of translated code, without applying code optimizations
- Efficient execution in fragment cache
  - Linking branches in the translated cache
  - Handling indirect branches inline in the translated code
- Program instrumentation
  - Reduce amount of instrumentation inserted
  - Reduce the cost of an individual piece of instrumentation code

Strata Software Dynamic Translation

Operating System
CPU
Context Capture
New PC Context Switch
Cached?
New Fragment Fetch Decode Translate Next PC
Dynamic Translator Finished?

Overhead Reduction in Strata

• Improve run-time performance of translated code, without applying code optimizations

Strata Performance

Program Instrumentation

- CoCo monitors for run-time conditions
  - When conditions are found: Apply some optimization plan
- Program monitoring needs instrumentation
  - Instrumentation code is inserted into translated binary
  - Gathers information, may invoke some action
  - Event-driven model
- Challenges
  - Dynamic instrumentation: Insert and remove at run-time
  - Portable mechanisms (i.e., retargetability)
  - Low run-time overhead (including the insertion & removal)

Approach: Instrumentation Optimization

- Costs associated instrumentation
  - Probe count: Number of probes executed
  - Probe cost: Intercept program execution
- Transform instrumentation to reduce costs
  - Dynamic probe coalescing: Reduce probe count
  - Partial context switch: Reduce probe cost
  - Payload partial inlining: Reduce probe cost for hot code
- Applied for static or dynamic instrumentation

Example: Cache Simulation

- Cache simulation can be sloooow
- Direct-execution cache simulators
  - Popular (& compiled simulation) – Shade, Embra, Fast-Sim
  - Instrument every load and store – call data cache simulator
  - Instrument every basic block – call instruction cache simulator
- Used INSOP and Strata to build a very fast direct-execution simulator
### Comparison of Cache Simulators

<table>
<thead>
<tr>
<th>Benchmarks</th>
<th>Native (Secs.)</th>
<th>Shade</th>
<th>Sim-cache</th>
<th>Strata-Embra</th>
<th>Strata-Embra-O6</th>
</tr>
</thead>
<tbody>
<tr>
<td>mcf</td>
<td>1.813</td>
<td>10.3x</td>
<td>23.4x</td>
<td>5.5x</td>
<td>2.5x</td>
</tr>
<tr>
<td>twolf</td>
<td>2.534</td>
<td>74.9x</td>
<td>N/A</td>
<td>15.7x</td>
<td>7.3x</td>
</tr>
<tr>
<td>gcc</td>
<td>1.364</td>
<td>111.1x</td>
<td>112.4x</td>
<td>25.2x</td>
<td>11.1x</td>
</tr>
<tr>
<td>vpr</td>
<td>831</td>
<td>113x</td>
<td>109.8x</td>
<td>20.4x</td>
<td>10.1x</td>
</tr>
<tr>
<td>parser</td>
<td>1.979</td>
<td>48.4x</td>
<td>123.2x</td>
<td>21.6x</td>
<td>8.6x</td>
</tr>
<tr>
<td>vortex</td>
<td>2.747</td>
<td>101.9x</td>
<td>114.4x</td>
<td>21.5x</td>
<td>10.6x</td>
</tr>
<tr>
<td>gzip</td>
<td>1.192</td>
<td>228.1x</td>
<td>227.9x</td>
<td>34x</td>
<td>13.1x</td>
</tr>
<tr>
<td>baip</td>
<td>1.325</td>
<td>169.2x</td>
<td>194.1x</td>
<td>26.7x</td>
<td>8.1x</td>
</tr>
</tbody>
</table>

Average improvement over Strata-Embra is 2.4x

### Related Work
- **Effective optimization**
  - Adaptive optimizing compilers: [Cooper02,04 & Whalley04]
  - Iterative compilation: [Knijnenburg03]
  - Optimization space exploration: [Triantafyllos03]
  - Analytic models: [Wu91, Sarkar 97, McKinley96, Hu02]
- **Software dynamic translation**
  - Dynamo/RIO, Dynamo, Mojo, Vulcon, Walkabout, DELI
- **Cache simulation**
  - Shade, Embra, Fast-Cache, FastSim

### Summary
- **A new planning-based approach to compilation called Continuous Compilation (CoCo)**
- **Apply whole suite of optimizations with constant refinement of optimizations and plans for them**
- **Results**
  - Highly accurate predictions for simple loop optimizations
  - Highly accurate predictions for scalar optimizations
  - Low overhead run-time system based on SDT
  - INSOP reduces instrumentation cost (cache simulation)

### Collaborators
- Many students have participated, including:
  - **FPO:** Min Zhao (Pitt)
  - **Program instrumentation:** Naveen Kumar (Pitt)
  - **Strata:** Kevin Scott (UVA/Google) and Naveen Kumar
  - **Overhead reduction:** Kevin Scott, Naveen Kumar, Jason Mars (Pitt)

Other Faculty: Mary Lou Soffa, Jack Davidson (UVA)
- **Sponsored by the National Science Foundation, Next Generation Systems, 2002-2003 and 2003-2006**

### Current Areas of Focus
- **Continuous compilation**
- **Software dynamic translation**
  - Low overhead dynamic translation
  - New applications to architecture simulation, security
- **Debugging of dynamically translated code**
  - Dynamically optimized code
  - Security checking
  - Dynamically compiled simulation
- **Soft error detection & recovery based on SDT**
- **Power-aware memory systems**

### Current Projects
- **Debugging dyn. translated/optimized code** – N. Kumar
- **On-demand structural software testing with dynamic instrumentation** – J. Misurda and J. Clause
- **Static/dynamic optimization planning** – S. Zhou
- **Optimization checking** – Y. Huang
- **Compiler-driven power management** – N. Aboughazaleh
- **Memory systems for cognitive processing**
- **Reuse through Speculation on Traces** – M. Pilla (from UFRGS)
Selected Past Projects

• Instruction code compression/decompression
• On-demand code downloading for Smartcards
• Program profiling primitives and profiling language
• Software based value reuse on traces
• Power on/Shut down of superscalar functional units
• Memory bus reordering for power reduction
• Processor-driven DVS (based on IPC/peak demands)
• Data width-sensitive VLIWs & scheduling
• Application-specific processors (automatic design and target architectures)

CS2002 Projects?

• Debugging for code security with SDT
• Dynamically compiled & sampled architecture simulation
• Profit-driven optimization for other constraints (e.g., power, code size)
• Self-checking programs (for soft errors; e.g., memory bit flips)
• Domain specific languages for structural testing and automatic planning
• Reconfigurable / custom memory systems
• And many others… or your ideas??

Let’s Talk!

• 6409 Sennott Square
• Office hours: MW 1-3 PM
• Or by appointment

• Send e-mail…. childers@cs.pitt.edu

• See selected papers online…. http://www.cs.pitt.edu/~childers