Combinatorial Spill Code Optimization and Ultimate Coalescing

Roberto Castañeda Lozano – SICS
Gabriel Hjort Blindell – KTH
Mats Carlsson – SICS
Christian Schulte – KTH

LCTES 2014
Outline

1 Introduction
2 Background
3 Alternative Temporaries
4 Results
5 Conclusion
Combinatorial Code Generation

- **Traditional approach**
  - heuristics, staging: suboptimal, complex

- **Combinatorial approach**
  - model: variables, constraints, objective
  - solve: integer programming, constraint programming . . .

- optimization, integration: potentially optimal, flexible
Register Allocation

- Global register allocation has many subproblems
- Competitive approaches must capture all of them
- Focus of this presentation:
  - **spill code optimization**
    - remove unnecessary spill instructions
  - **coalescing**
    - remove unnecessary register-to-register moves
    - *basic*: coalesce temps related by moves
    - *ultimate*: even if their live ranges overlap
Our Approach

- Alternative temporaries
  - program representation
  - combinatorial structure

- Extends combinatorial reg. allocation and scheduling with
  - spill code optimization
  - ultimate coalescing

- Yields better code than
  - previous combinatorial approaches
  - traditional heuristic approaches

- Scales despite increased solution space
Related Approaches

- Some models include spill code optimization
  - (Chang et al., 1997)
  - (Bashford and Leupers, 1999)
  - (Nagarakatte and Govindarajan, 2007)
  - (Eriksson and Kessler, 2012)
  - typically via a quadratic number of Boolean variables

- Some models include basic coalescing
  - (Wilson et al., 1994)
  - (Bashford and Leupers, 1999)
  - (Castañeda et al., 2012)

- No model includes ultimate coalescing
  - non-trivial when combined with scheduling
Program Representation

- Dependency graph with processor instructions
Spill Code Optimization

- Remove unnecessary spill load instructions

Before spilling
Spill Code Optimization

- Remove unnecessary spill load instructions

- Spill everywhere: a load before each use
Spill Code Optimization

- Remove unnecessary spill load instructions

- Spill code optimization: reuse temp $t_3$ to remove a load
Ultimate Coalescing

- Remove unnecessary register-to-register moves
  - even if the respective temp live ranges overlap

- Basic: move’s temps \((t_1, t_2)\) interfere, cannot coalesce
Ultimate Coalescing

- Remove unnecessary register-to-register moves
  - even if the respective temp live ranges overlap

- Ultimate: they hold the same value, can coalesce
1 Introduction
2 Background
3 Alternative Temporaries
4 Results
5 Conclusion
Alternative Temporaries

- Program representation and combinatorial structure
- Augments model with
  - spill code optimization
  - ultimate coalescing
- Allows connection of alternative temps to each instruction
  - invariant: alternative temps hold the same value
Instruction $k$ can be connected (dashed) to $t_3$ or $t_4$.
Alternative Temporaries: Spill Code Optimization

- If $k$ is connected to $t_3$, $t_4$ is not used
Alternative Temporaries: Spill Code Optimization

If $t_4$ is not used, its definer `load` becomes inactive
Instruction $k$ can be connected to $t_1$ or $t_2$
Alternative Temporaries: Ultimate Coalescing

- If $k$ is connected to $t_1$, $t_2$ is not used
If $t_2$ is not used, its definer move becomes inactive.
Alternative Temporaries: Construction

1. Extend program with optional copies
   - after definition: reg-to-reg move or memory store
   - before use: reg-to-reg move or memory load

2. Replace each temporary use with alternatives
   - \{t_1, t_2, t_3, t_4\} all hold the same value
   - due to copy semantics of move, store, and load
Combinatorial Model

\[
\text{minimize } \sum_{b \in B} \text{weight}(b) \times \text{cost}(b) \quad \text{subject to}
\]

\[
l_t \iff \exists p \in P : (\text{use}(p) \land y_p = t) \quad \forall t \in T
\]

\[
a_{\text{definer}}(t) \iff l_t \quad \forall t \in T
\]

\[
a_o \iff y_p \neq \bot \quad \forall o \in O, \forall p \in \text{operands}(o)
\]

\[
a_o \iff i_o \neq \bot \quad \forall o \in O
\]

\[
r_{yp} \in \text{class}(i_o, p) \quad \forall o \in O, \forall p \in \text{operands}(o)
\]

\[
\text{disjoint2}(\{(r_t, r_t + \text{width}(t) \times l_t, l_{st}, l_{et}) : t \in T(b)\}) \quad \forall b \in B
\]

\[
r_{yp} = r \quad \forall p \in P : p \triangleright r
\]

\[
r_{yp} = r_{yq} \quad \forall p, q \in P : p \equiv q
\]

\[
l_t \implies l_{st} = c_{\text{definer}}(t) \quad \forall t \in T
\]

\[
l_t \implies l_{et} = \max_{o \in \text{users}(t)} c_o \quad \forall t \in T
\]

\[
a_o \implies c_o \geq c_{\text{definer}}(y_p) + \text{lat}(i_{\text{definer}}(y_p)) \quad \forall o \in O, \forall p \in \text{operands}(o) : \text{use}(p)
\]

\[
\text{cumulative}(\{(c_o, \text{con}(i_o, r), \text{dur}(i_o, r)) : o \in O(b)\}, \text{cap}(r)) \quad \forall b \in B, \forall r \in R
\]

- Generic objective function: speed, code size, . . .
- See the paper for details
Introduction

Background

Alternative Temporaries

Results

Conclusion
Experiment Setup

- 10 functions from each DSP application in MediaBench
  - medium size: 10 to 1000 instructions
  - sampled by clustering (size, register pressure)

- Selected Hexagon V4 instructions with LLVM 3.3
  - VLIW DSP in Qualcomm’s Snapdragon system-on-chip

- Constraint-based code generator
  - uses Gecode 4.2.1 as the underlying constraint solver
  - iterative scheme: finds better solutions every iteration
  - fixed to 10 iterations (point of convergence)

- LLVM as a traditional code generator
  - register allocation by priority-based coloring
  - instruction scheduling by list scheduling
Impact of Alternative Temporaries

Optimal solution improvement due to alternative temps (compared to model by Castañeda et al., 2012)

- 62% of the functions are faster
- None is slower – as expected
- 2% geometric mean improvement
Code Quality Compared to Traditional Approaches

Estimated speed up over LLVM

- 7% geometric mean improvement
- Provably optimal code (■) for 29% of the functions
- Model limitation: no rematerialization
Scalability

Solving time to reach LLVM’s quality

- Quadratic average complexity up to 1000 instructions
- Comparable to approach **without** alternative temps
Different Optimization Criteria

Code size improvement over LLVM

- 1% geometric mean improvement
- Low development effort to adapt the code generator
Conclusion

- Alternative temporaries completes combinatorial code generation with
  - spill code optimization
  - ultimate coalescing

- Yields a code generator that
  - delivers faster code than traditional ones
  - is robust and scales to medium-size functions
  - adapts easily to different optimization criteria

- Lots of future work
  - rematerialization
  - global instruction scheduling
  - handle unknown instruction latencies
  - improve runtime with different solving techniques