Constructing an Optimisation Phase Using Grammatical Evolution

Brad Alexander and Michael Gratton

Outline

- Problem
- Current Approaches
- Experimental Aim
- Ingredients
- Experimental Setup
- Experimental Results
- Conclusions/Future Work

Problem

- Optimising compilers work in a complex design space.
  - Difficult for the author of the optimiser configure well for all applications.
  - Static design is always a compromise.
- A Solution:
  - automatically adapt the optimiser to the set of programs it compiles!
- Problem:
  - the design space is huge and chaotic
  - however, can search this space using heuristic methods.

Current Approaches

- Phase seqencing

Loop Invariant Hoisting

↓

Common Subexpression Elimination

↓

Dead Code Elimination

↓

Block Reordering

↓

Current Approaches

- Phase seqencing

↓

Common Subexpression Elimination

↓

Loop Invariant Hoisting

↓

Dead Code Elimination

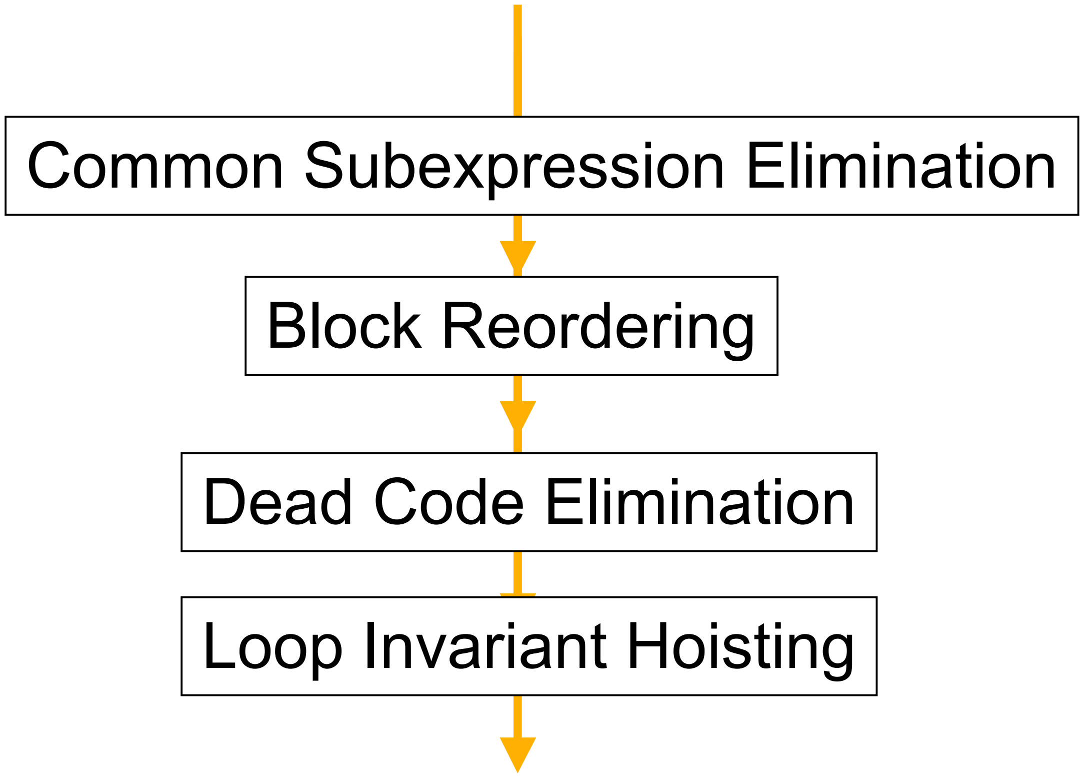
↓

Block Reordering

↓

Current Approaches

- Phase seqencing



Current Approaches

- Phase seqencing

Dead Code Elimination

↓

Block Reordering

↓

Common Subexpression Elimination

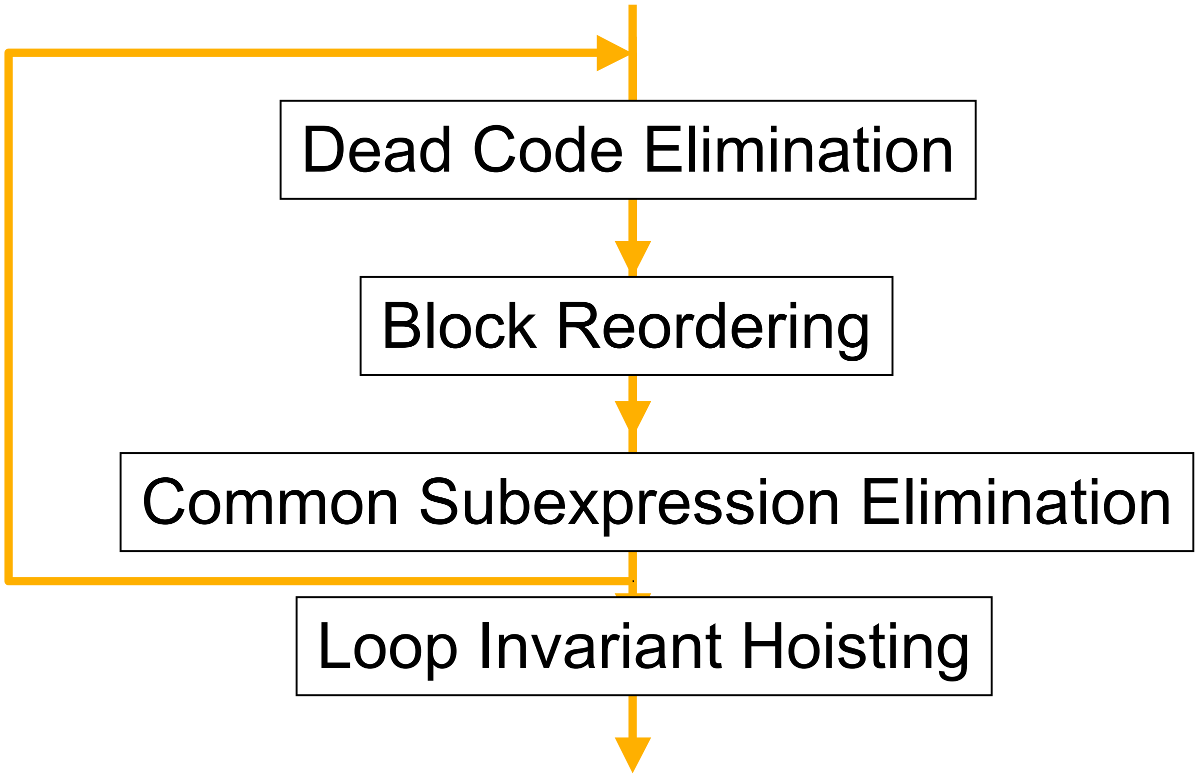
↓

Loop Invariant Hoisting

↓

Current Approaches

- Phase seqencing



Current Approaches

- Parameter Tuning

Loop unroll factor: 3
Loop tiling factor: 2

Current Approaches

- Parameter Tuning

Loop unroll factor: $\boxed{4}$
Loop tiling factor: $\boxed{3}$

Current Approaches

- Evolution of Control Code

Register Allocation

Current Approaches

- Evolution of Control Code

if(reg_size > &
spill_cost …)

Register Allocation

Current Approaches

- Evolution of Control Code

Register Allocation

if(reg_size > &
spill_cost …)

Experimental Aim

- All current work assumes that optimisation phases are pre-existing and atomic or parametric.
- Currently no work on the construction of these phases from smaller components.
- Aim of this experiment is a proof of concept:
- To attempt to build a safe, substantial, and effective optimisation phase using heuristic search.
  - We use Grammatical Evolution (GE) a form of Genetic Programming (GP).
  - The genotype to phenotype encoding in GE constrains the population to syntactically correct individuals.

Experimental Application

- Evolution of a phase of a compiler mapping a functional language (Adl) to a hardware definition language (Bluespec).
- The target phase is the Data Movement Optimiser (DMO) that reduces data flowing through a functional intermediate form (point-free code).
- There is an extant hand-written DMO that:
  - was non-trivial to construct.
  - can be used as a source of building blocks.
  - can be used as a benchmark
- The DMO is written in Stratego, a term-rewriting language consisting of rewrite rules and strategies for their application.

Ingredients

- Three ingredients in any GP exercise:
  1. The language grammar consisting of:
     - terminals
     - non terminals
  2. The evolutionary framework.
  3. The evaluation function
- We look at these in turn.

The Language Grammar (1)

- All individuals are expressed in Stratego
- Terminals
  - Consist of simple rewrite rules e.g.

    CompIntoMap: $f^* \circ g^* \rightarrow (f \circ g)^*$

    MapIntoComp: $(f \circ g)^* \rightarrow f^* \circ g^*$

    RemoveId: $id \circ f \rightarrow f$
  - grouped together using the left choice (<+) operator e.g.

    CompIntoMap <+ RemoveId
  - Semantics: try applying CompIntoMap to current node and, if that fails, try applying RemoveId.
- We use the same terminals as the handwritten DMO

The Language Grammar (2)

- Actual terminals include:

pushDownMap	(vectorise)
pushDownComp	(fuse loops)
simp	(apply simplifying rules)
leftAssociate	(left associate binary composition)

  – In most contexts, the order of rules within a group is of minor consequence
    - If they can be applied they eventually will be applied.

  – These terminals have little impact without strategies to apply them.

The Language Grammar (3)

- Non-terminals are strategies for rule application.
  - These take strategies or rule-groups as parameters and apply the them to the target AST in some order.
- Examples include:

  bottomup(s) : apply s to the current sub-tree bottomup

  innermost(s) : apply s to the current sub-tree bottomup until it can no longer be applied (fixpoint strategy)

  s ; t : apply s to current sub-tree followed by t

  repeatUntilCycle(s) : apply s to the current sub-tree until a result seen before in this invocation is detected.
- Example:

  bottomup(leftAssociate;innermost(simp))

The Evolutionary Framework

- We used LibGE in our experiments.
  - A popular framework for developing GE applications.
- LibGE (based on LibGA) takes:
  - A grammar definition and,
  - A fitness function
  - Some parameter settings

  and handles:
  - Population initialisation, application of the fitness function to individuals, application of genetic operators, collection of statistics and, genotype to phenotype mapping.
- The mapping works by using 8-bit numbers in the genotype string to select productions in the language grammar.

Fitness Function(1)

- Fitness is calculated by running evolved optimisers against up to six benchmark programs and their data against a dynamic cost-model.
  - Benchmarks needed to be carefully chosen to require multiple strategies and have a gradual gradient of difficulty.
- Fitness calculated relative to cost of hand-coded DMO on each benchmark *i* ($cost\_opt_i$):

$$fitness = \frac{\Sigma_{i=0}^{n}(cost\_opt_i / cost\_evo_i)}{n}$$

- Average fitness evaluation takes 5 seconds. Zero fitness for timeout or stack-overflow error.

Fitness Function(2)

- Hand Coded Benchmark:

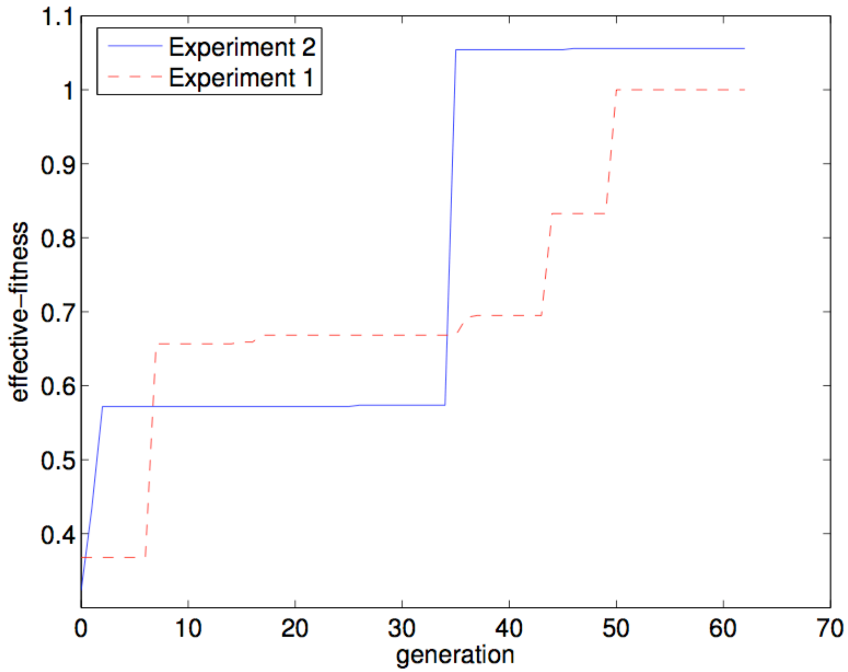
```
repeatUntilCycle(
    bottomup(
        repeatUntilCycle(
            innermost(LeftAssociate)
            ;innermost(pushDownComp)
            ;innermost(LeftAssociate)
            ;innermost(simp)
            ;innermost(LeftAssociate)
            ;innermost(pushDownMap)
            ;innermost(LeftAssociate)
            ;innermost(simp)))
    bottomup(
        repeatUntilCycle(
            innermost(LeftAssociate)
            ;innermost(pushDownAlltup)
            ;innermost(LeftAssociate)
            ;innermost(alltupSimp)
            ;innermost(LeftAssociate)
            ;innermost(convertAndRemoveIds))))
```

Experimental Setup

- All grammar elements pre-compiled into stratego libraries for faster running.
- Several runs conducted to tune fitness function.
- Final two runs:
  - Population approximately 250 individuals
  - Run for 80 generations and 63 generations respectively.
  - LibGE settings: Max tree depth 15. Read of genome can wrap-around twice.
  - Mostly default LibGA settings (for GE): Roulette wheel selection, 90% probability of crossover, 1% mutation probability, 1% replacement ratio and elitism switched on.

Experimental Results (1)

- Both runs evolved individuals at least as good as the handwritten DMO's on the benchmarks.



Experimental Results (2)

- Robustness
  - Take the fittest individuals and expose them to thirty benchmarks and measure their performances.
  - Most did not generalise well but the fittest did slightly better than hand coded optimiser.
- Correctness
  - 500 fittest individuals collected and tested.
  - None produced semantic errors.
- Code Size
  - Best individuals very large with much redundancy.

Conclusions and Future Work

- Evolving a non-trivial optimisation phase is feasible
  - Good results for effectiveness, robustness and correctness.
- Future work includes:
  - Pushing evolutionary process down to individual rules
  - Controlling code-size and efficiency.
  - Extending work to rewriting systems in other languages.