Exploiting Phase Inter-Dependencies for Faster Iterative Compiler Optimization Phase Order Searches

Michael R. Jantz    Prasad A. Kulkarni

Electrical Engineering and Computer Science, University of Kansas
{mjantz,kulkarni}@ittc.ku.edu

September 30, 2013
Optimization Phase Ordering

- Optimization Phase Ordering – finding the best set and combination of optimizations to apply to each function / program to generate the best quality code
- Earlier research has shown customized phase sequences can significantly improve performance – by as much as 33% (VPO) or 350% (GCC)[1]
- Iterative searches are most common solution
- *Problem*: exhaustive searches are extremely time consuming
Hypothesis

- Exhaustive search algorithms assume all optimization phases interact with each other.
- Optimization phases do not always interact with each other.
- **Hypothesis**: it is possible to reduce search times by considering well-known relationships between phases during the search.
- Focus on two categories of phase relationships:
  - *Cleanup phases* are unlikely to interact with other phases.
  - "Branch" and "non-branch" phases are likely to interact with phases within their own group, but show minimal interaction with phases in the other set.
Objective and Contributions

- **Primary Objective**: evaluate iterative search algorithms that exploit phase relationships
- Create two variations of our exhaustive search algorithm
  - Remove cleanup phases from the search and apply implicitly
  - Partition optimizations into branch and non-branch sets and conduct exhaustive (multi-stage) phase order searches over the partitioned sets
- Use these observations to find a common set of near-optimal phase sequences, and improve heuristic GA-based searches
Compiler and Benchmarks

- All experiments use the Very Portable Optimizer (VPO)
  - Compiler backend that performs all optimizations on a single low-level IR known as RTLs
  - Contains 15 reorderable optimizations
  - Compiles and optimizes individual functions one at a time

- VPO targeted to generate code for ARM running Linux

- Use a subset of applications from MiBench
  - Randomly selected two benchmarks from each of the six categories for a total of 12 benchmarks
  - Evaluate with the standard small input data set
  - 246 functions, 87 of which are executed at least once
Setup for Exhaustive Search Space Enumeration

- Default exhaustive search uses all 15 phases in VPO
- Implement the framework proposed by Kulkarni et al. [2]
- Main idea: generate all possible function instances that can be produced by applying any combination of phases of any possible sequence length
Setup for Evaluating Search Space Enumeration

Figure 1: DAG for Hypothetical Function with Optimization Phases a, b, and c

- Nodes represent distinct function instances, edges represent transition from one function instance to another on application of an optimization phase
- Unoptimized function is at the root
- Each successive level is produced by applying all possible phases to distinct nodes at the previous level
Setup for Evaluating Search Space Enumeration

Figure 1: DAG for Hypothetical Function with Optimization Phases a, b, and c

- Employ redundancy detection techniques to find when phase orderings generate duplicate function instances
- Terminate when no additional phase is successful in creating a new distinct function instance at the next level
Evaluating the Default Exhaustive Search Configuration

- **Search space size** measured as the number of distinct function instances (nodes) generated by the exhaustive search

- **Performance Evaluation**
  - Per-function perf. in terms of dynamic instruction counts
  - Whole program simulated processor cycles
  - Whole program (native) run-time

- Exhaustively enumerated 236 (of 246) benchmark functions, 81 (of 87) executed functions
  - Search space size varies from a few to several million nodes
  - Maximum performance improvement is 33%, average is 4.0%
Phase Independence

- Phases are *independent* if their order of application does not affect the final code that is produced.
- In Figure 1, phases a and b are independent of each other.
- Removing independent phases from the search, and applying them implicitly will make no difference to final code produced.

Figure 1: DAG for Hypothetical Function with Optimization Phases a, b, and c
Implicit Application of Cleanup Phases During Exhaustive Phase Order Search

- *Dead Code Elimination* (DCE) and *Dead Assignment Elimination* (DAE) designated as *cleanup phases*
- Cleanup phases independent from other phases in the search
- Modified exhaustive search excludes DCE and DAE, and *implicitly* applies these phases after each phase during the exhaustive search
Implicit Application of Cleanup Phases During Exhaustive Phase Order Search

Figure 2: Comparison of search space size (over 236 benchmark functions) achieved by our configuration with cleanup phases implicitly applied to the default exhaustive phase order search space

- Per-function average reduction is 45%, total reduction is 78%
- Search reaches same best performance for 79 out of 81 functions
- Worst performance degradation for one function is 9%, average is 0.1%
Exploiting Phase Independence Between Sets of Phases

- Re-ordering phases that work on distinct code regions and do not share resources should not affect performance.
- Phases in VPO can be partitioned into control-flow changing *branch* phases, and phases that do not affect control flow (*non-branch* phases).
- *Multi-stage* exhaustive search strategy:
  - First stage: search over only branch phases and find function instances that produce the best code.
  - Second stage: continue search from best function instances found by the first stage using only the non-branch phases.
Exploiting Phase Independence Between Sets of Phases

Figure 3: Comparison of search space size achieved by our multi-stage search algorithm as compared to the default exhaustive phase order search space.

- Per-function average reduction is 76%, total reduction is 90%
- Only two of 81 functions do not reach optimal performance (with degradations of 3.47% and 3.84%)
- No performance losses if we include all phases in the second stage, search space reductions are similar (75% per-function average, 88% total)
Exploiting Phase Inter-Dependencies for Faster Iterative Compiler Optimization Phase Order Searches

Exploiting Phase Independence Between Sets of Phases

Finding a Covering Set of Phase Sequences

Finding a Covering Set of Phase Sequences

- No single sequence achieves optimal perf. for all functions
  - Can a small number of sequences achieve near-optimal performance for all functions?

- Difficult to explore due to exorbitantly large search space sizes

- Our approach: employ the observation that phase partitioning over the search space does not impact the best performance
Finding a Covering Set of Phase Sequences

- Step 1: Find minimal number of best branch-only phase orderings (expressed as the classical set cover problem)
  - Apply all possible branch-only sequences of length 6
  - Generate a set of functions for each branch-only sequence where a function is in a sequence’s set if that sequence achieves the best branch-only solution for that function
  - Set-cover problem: find the minimal number of branch-only phase orderings whose corresponding sets cover all functions
  - Only three branch-only sequences needed to cover all functions
Finding a Covering Set of Phase Sequences

Step 2: Combine best branch-only sequences with non-branch sequences to find the best sequences of length 15

- Generate all non-branch sequences of length nine, append each to the best branch-only sequences
- Create sets of functions that reach the best performance for each sequence and again approximate the minimal set cover

Yields 19 sequences that are needed to reach the best phase ordering performance for the 81 functions
Finding a Covering Set of Phase Sequences

- Covering Sequence Evaluation
  - Incrementally apply covering sequences in descending order of the number of functions they cover
  - Employ standard *leave-one-out* cross validation
  - Experiment applies $N$ covering sequences *one-at-a-time* and reports best performance for each function
Finding a Covering Set of Phase Sequences

Figure 4: Average performance improvement of 19 covering sequences compared to default compiler sequence

- Points along the X-axis show the best perf. of the \( N \) covering sequences compared to default compiler sequence
- Horizontal line shows average perf. of best phase ordering sequences
Finding a Covering Set of Phase Sequences

Figure 4: Average performance improvement of 19 covering sequences compared to default compiler sequence

- Batch sequence achieves better performance than any one covering sequence
- Only three covering sequences required to improve performance over default compiler sequence
- Applying all 19 sequences yields 3.1% average improvement
Better Genetic Algorithm Searches

- Popular method to develop heuristic searches is to use a genetic algorithm (GA)

- Terminology
  - **Chromosomes** are phase ordering sequences
  - The **population** is the set of chromosomes under consideration
  - Populations are evaluated over several **generations**

- Genetic algorithm-based heuristic search algorithm
  - Population in the first generation is randomly initialized
  - Evaluate performance with each chromosome in the population
  - Sort chromosomes in decreasing order of performance, and use **cross-over** and **mutation** to create population for the next gen.
Better Genetic Algorithm Searches

- Experiments evaluate 20 chromosomes over 200 generations
- Implement and evaluate three GA variants
  - Default – first population randomly generated
  - Branch bias – first population prepends branch-only covering sequences to randomly generated sequences
  - Full bias – first population prepends covering sequences with both branch and non-branch phases to 19 randomly generated sequences, one chromosome generated randomly
Better Genetic Algorithm Searches

Figure 5: Improvement achieved by the default and biased GA configurations over default compiler sequence in each generation (averaged over all 81 benchmark functions)

- Branch bias focuses search to more useful portions of the search space and allows for faster convergence
- Full bias shows improvements because covering sequences achieve very good performance by themselves
Conclusions

- **Primary contribution**
  - Phases do not necessarily interact with each other
  - This observation can be exploited to reduce exhaustive and heuristic search times

- Evaluated exhaustive search variations
  - Implicitly applying cleanup phases
    - 55% avg. reduction, 78% total reduction, 0.1% perf. loss
  - Multi-stage search over independent subsets of phases
    - 75% avg. reduction, 88% total reduction, no perf. loss

- Developed new algorithms to find a small set of near-optimal phase orderings and improve GA performance
Questions

Thank you for your time. Questions?
References
