Understanding Optimization Phase Interactions to Reduce the Phase Order Search Space

Michael Jantz

Prasad Kulkarni (Advisor)

KU THE UNIVERSITY OF KANSAS

Introduction

Optimization Phases

- Conventional optimizing compilers contain several *optimization phases*
  - Phases apply transformations to improve the code
  - Phases require specific code patterns and / or resources (e.g. machine registers) to do their work
- Phases *interact* with each other
- No single ordering is best for all programs

The Phase Ordering Problem

- Conventional compilers are plagued with the *phase ordering problem*:

  > "How to determine the ideal sequence of optimization phases to apply to each function or program so as to maximize the gain in speed, code-size, power, or any combination of these performance constraints."

- Different orderings can have major performance implications
- Particularly important in performance-critical applications (e.g. embedded systems)

Iterative Phase Order Search

- Most common solution employs iterative search
  - Evaluate performance produced by many phase sequences
  - Choose the best one
- Problem: Extremely large phase order search spaces are infeasible or impractical to exhaustively explore
- Thus, we must reduce compilation time of iterative phase order search to harness most benefit from today's optimizing compilers.

Speeding Up the Search

- Two complementary approaches:
  - Develop techniques to reduce the exhaustive search space
  - Perform a partial exploration of the search space using machine learning algs. (most research *solely* focused here)
- Our approach: analyze and attempt to address most common phase interactions, and then develop solutions to reduce the search space
  - Can exhaustive approaches more practical
  - May enable better predictability and efficiency for intelligent heuristic searches

KU THE UNIVERSITY OF KANSAS

Experimental Setup

Compiler Framework

- The Very Portable Optimizer (VPO) Compiler
  - Compiler backend that performs all transformations on a single low-level intermediate representation called RTL's
  - 15 *optional* code-improving phases
  - Optimizations applied repeatedly, in any order
  - Compilation performed one function at a time
  - For our experiments, targeted to produce code for the StrongARM SA-100 processor running Linux.
- SimpleScalar ARM simulator used for performance measurements

Our Benchmark Set

- A subset of the *MiBench* benchmark suite.
  - C applications targeted to the embedded systems market.
- Selected 2 benchmarks from each category in this suite, for a total of 12 benchmarks.
  - VPO compiles and optimizes one function at a time.
  - 246 functions, 86 of which were executed with the input data provided with each benchmark.

Experimental Framework

- Experiments run on a high-performance computer cluster (Bioinformatics Cluster at ITTC)
  - 174 nodes (4GB to 16GB of main memory per node)
  - 768 processors (frequencies range from 2.8GHz to 3.2GHz)
- Phase order searches parallelized by running each exhaustive search on different nodes of the cluster
- Could not enumerate search space for all functions due to time/space restrictions
  - Ran for more than 2 weeks
  - Generated raw data files larger than the max. allowed on our 32 bit system (2.1GB)

False Phase Interactions

Register Conflicts

- Architectural registers play a key role in how optimization phases interact.
- Phases may be enabled or disabled due to:
  - Register availability (only a limited number of registers)
  - Requirements that particular program values (e.g. function arguments) must be held in specific registers
- How do register availability and assignment affect phase interactions?
  - How do these interactions affect the size of the phase order search space?

False Phase Interactions

- Manually analyzed most common phase interactions
- Found that many interactions not due to limited number of registers
  - But due to different register assignments produced by different phase orderings
- False register dependency may disable optimizations in some phase orderings, while not for others.

Example of False Register Dependency

1. r[12] = r[12] – 8;
2. r[1] = r[12];
3. r[1] = r[1]{2;
4. r[12] = r[13] + .LOC;
5. r[12] = Load[r[12] + r[1]];

(a) original code

1. r[12] = r[12] – 8;
2. r[1] = r[12];
3. r[1] = r[1]{2;
4. r[12] = r[13] + .LOC;
5. r[12] = Load[r[12] + r[1]];

(b) original code

1. r[12] = r[12] – 8;
2. r[1] = r[12];
3. r[1] = r[1]{2;
4. r[12] = r[13] + .LOC;
5. r[12] = Load[r[12] + r[1]];

(c) original code

Example of False Register Dependency

1. r[12] = r[12] – 8;
2. r[1] = r[12];
3. r[1] = r[1]{2;
4. r[12] = r[13] + .LOC;
5. r[12] = Load[r[12] + r[1]];

(a) original code

1. r[12] = r[12] – 8;
2. r[1] = r[12];
3. r[1] = r[1]{2;
4. r[12] = r[13] + .LOC;
5. r[12] = Load[r[12] + r[1]];

(b) applying instruction selection

1. r[12] = r[12] – 8;
2. r[1] = r[12];
3. r[1] = r[1]{2;
4. r[12] = r[13] + .LOC;
5. r[12] = Load[r[12] + r[1]];

(c) original code

Example of False Register Dependency

1. r[12] = r[12] – 8;
2. r[1] = r[12];
3. r[1] = r[1]{2;
4. r[12] = r[13] + .LOC;
5. r[12] = Load[r[12] + r[1]];

(a) original code

2. r[1] = r[12] – 8;
3. r[1] = r[1]{2;
4. r[12] = r[13] + .LOC;
5. r[12] = Load[r[12] + r[1]];

(b) applying instruction selection

1. r[12] = r[12] – 8;
2. r[1] = r[12];
3. r[1] = r[1]{2;
4. r[12] = r[13] + .LOC;
5. r[12] = Load[r[12] + r[1]];

(c) original code

Example of False Register Dependency

1. r[12] = r[12] – 8;
2. r[1] = r[12];
3. r[1] = r[1]{2;
4. r[12] = r[13] + .LOC;
5. r[12] = Load[r[12] + r[1]];

(a) original code

2. r[1] = r[12] – 8;
3. r[1] = r[1]{2;
4. r[12] = r[13] + .LOC;
5. r[12] = Load[r[12] + r[1]];

(b) applying instruction selection

1. r[12] = r[12] – 8;
2. r[1] = r[12];
3. r[1] = r[1]{2;
4. r[12] = r[13] + .LOC;
5. r[12] = Load[r[12] + r[1]];

(c) original code

Example of False Register Dependency

1. r[12] = r[12] – 8;
2. r[1] = r[12];
3. r[1] = r[1]{2;
4. r[12] = r[13] + .LOC;
5. r[12] = Load[r[12] + r[1]];

(a) original code

2. r[1] = r[12] – 8;

4. r[12] = r[13] + .LOC;
5. r[12] = Load[r[12] + (r[1]{2)];

(b) applying instruction selection

1. r[12] = r[12] – 8;
2. r[1] = r[12];
3. r[1] = r[1]{2;
4. r[12] = r[13] + .LOC;
5. r[12] = Load[r[12] + r[1]];

(c) original code

Example of False Register Dependency

1. r[12] = r[12] – 8;
2. r[1] = r[12];
3. r[1] = r[1]{2;
4. r[12] = r[13] + .LOC;
5. r[12] = Load[r[12] + r[1]];

(a) original code

2. r[1] = r[12] – 8;

4. r[12] = r[13] + .LOC;
5. r[12] = Load[r[12] + (r[1]{2)];

(b) instruction selection followed by common subexpression elimination

1. r[12] = r[12] – 8;
2. r[1] = r[12];
3. r[1] = r[1]{2;
4. r[12] = r[13] + .LOC;
5. r[12] = Load[r[12] + r[1]];

(c) original code

Example of False Register Dependency

1. r[12] = r[12] – 8;
2. r[1] = r[12];
3. r[1] = r[1]{2;
4. r[12] = r[13] + .LOC;
5. r[12] = Load[r[12] + r[1]];

(a) original code

2. r[1] = r[12] – 8;

4. r[12] = r[13] + .LOC;
5. r[12] = Load[r[12] + (r[1]{2)];

(b) instruction selection followed by common subexpression elimination

1. r[12] = r[12] – 8;
2. r[1] = r[12];
3. r[1] = r[1]{2;
4. r[12] = r[13] + .LOC;
5. r[12] = Load[r[12] + r[1]];

(c) applying common subexpression elimination

Example of False Register Dependency

1. r[12] = r[12] – 8;
2. r[1] = r[12];
3. r[1] = r[1]{2;
4. r[12] = r[13] + .LOC;
5. r[12] = Load[r[12] + r[1]];

(a) original code

2. r[1] = r[12] – 8;

4. r[12] = r[13] + .LOC;
5. r[12] = Load[r[12] + (r[1]{2)];

(b) instruction selection followed by common subexpression elimination

1. r[12] = r[12] – 8;

3. r[1] = r[12]{2;
4. r[12] = r[13] + .LOC;
5. r[12] = Load[r[12] + r[1]];

(c) applying common subexpression elimination

Example of False Register Dependency

1. r[12] = r[12] – 8;
2. r[1] = r[12];
3. r[1] = r[1]{2;
4. r[12] = r[13] + .LOC;
5. r[12] = Load[r[12] + r[1]];

(a) original code

2. r[1] = r[12] – 8;

4. r[12] = r[13] + .LOC;
5. r[12] = Load[r[12] + (r[1]{2)];

(b) instruction selection followed by common subexpression elimination

1. r[12] = r[12] – 8;

3. r[1] = r[12]{2;
4. r[12] = r[13] + .LOC;
5. r[12] = Load[r[12] + r[1]];

(c) applying instruction selection

Example of False Register Dependency

1. r[12] = r[12] – 8;
2. r[1] = r[12];
3. r[1] = r[1]{2;
4. r[12] = r[13] + .LOC;
5. r[12] = Load[r[12] + r[1]];

(a) original code

2. r[1] = r[12] – 8;

4. r[12] = r[13] + .LOC;
5. r[12] = Load[r[12] + (r[1]{2)];

(b) instruction selection followed by common subexpression elimination

1. r[12] = r[12] – 8;

3. r[1] = r[12]{2;
4. r[12] = r[13] + .LOC;
5. r[12] = Load[r[12] + r[1]];

(c) common subexpression elimination followed by instruction selection

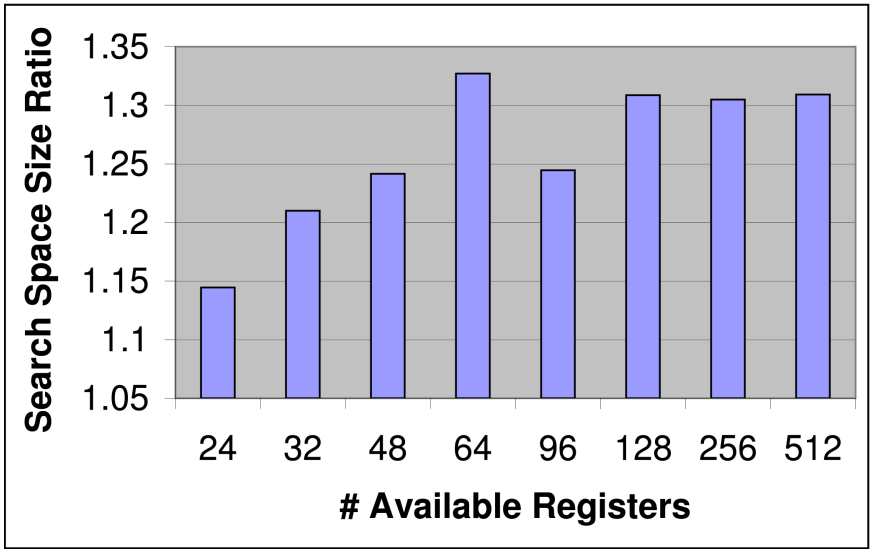
Register Pressure

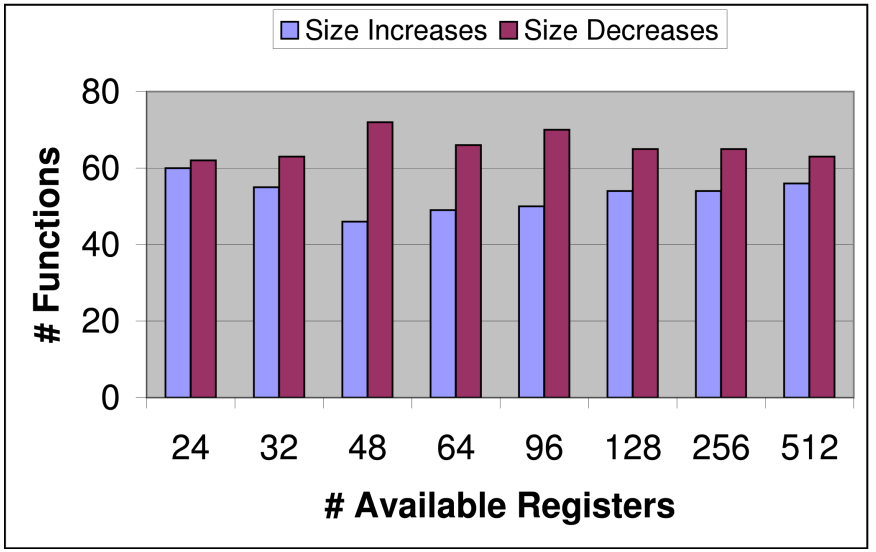
- False register dependence is often a result of limited number of available registers.
- Register scarcity forces optimizations to be implemented in a way that reassigns the same registers often and as soon as they are available.
- Hypothesis: decreasing register pressure should decrease false register dependence
  - Which *should* decrease the phase order search space
  - But more available registers could *enable* additional phase transformations increasing the total search space size.

Study on Register Availability

- Designed experiments to test the effect of the number of available registers on the size of the phase order search space.
- Modified VPO to produce code with register configurations ranging from 24 to 512 registers.
- Able to enumerate entire phase order search space in all configurations in 234 (out of 236) functions.
- Could not simulate code for new register configs
  - Able to estimate performance for 73 (out of 81) executed functions

Effect of Different Numbers of Available Registers





- Performance for most of the 73 executed functions either improves or remains the same, resulting in an average improvement of 1.9% in all register configs over the default

Observations

- Expansion caused by additional optimization opportunities exceeds the decrease (if any) caused by reduced phase interactions.
- VPO assumes limited registers and naturally reuses registers regardless of register pressure.
- Thus, limited number of registers is not sole cause of false register dependences.
- More informed optimization phase implementations may be able to minimize false register dependences.

Eliminating False Register Dependences

- Rather than alter all VPO optimization phases, we propose and implement two new optimization phases:
  - *Register Remapping* – reassign registers to live ranges
  - *Copy Propagation* – remove copy instructions by replacing the occurrences of targets of assignments with their values
- Apply these after every reorderable phase during our iterative search algorithm.
- Perform experiments in compiler configuration with 512 registers to avoid register pressure issues.

Register Remapping Removes False Register Dependency

1. r[12] = r[12] – 8;
2. r[1] = r[12];
3. r[1] = r[1]{2;
4. r[12] = r[13] + .LOC;
5. r[12] = Load[r[12] + r[1]];

(a) original code

2. r[1] = r[12] – 8;

4. r[12] = r[13] + .LOC;
5. r[12] = Load[r[12] + (r[1]{2)];

(b) instruction selection followed by common subexpression elimination

1. r[12] = r[12] – 8;

3. r[1] = r[12]{2;
4. r[12] = r[13] + .LOC;
5. r[12] = Load[r[12] + [r[1]];

(c) common subexpression elimination followed by instruction selection

Register Remapping Removes False Register Dependency

1. r[12] = r[12] – 8;
2. r[1] = r[12];
3. r[1] = r[1]{2;
4. r[12] = r[13] + .LOC;
5. r[12] = Load[r[12] + r[1]];

(a) original code

2. r[1] = r[12] – 8;

4. r[12] = r[13] + .LOC;
5. r[12] = Load[r[12] + (r[1]{2)];

(b) instruction selection followed by common subexpression elimination

1. r[12] = r[12] – 8;

3. r[1] = r[12]{2;
4. r[0] = r[13] + .LOC;
5. r[12] = Load[r[0] + r[1]];

(c) common subexpression elimination followed by instruction selection

Register Remapping Removes False Register Dependency

1. r[12] = r[12] – 8;
2. r[1] = r[12];
3. r[1] = r[1]{2;
4. r[12] = r[13] + .LOC;
5. r[12] = Load[r[12] + r[1]];

(a) original code

2. r[1] = r[12] – 8;

4. r[12] = r[13] + .LOC;
5. r[12] = Load[r[12] + (r[1]{2)];

(b) instruction selection followed by common subexpression elimination

1. r[12] = r[12] – 8;

3. r[1] = r[12]{2;
4. r[0] = r[13] + .LOC;
5. r[12] = Load[r[0] + r[1]];

(c) common subexpression elimination followed by instruction selection

Register Remapping Removes False Register Dependency

1. r[12] = r[12] – 8;
2. r[1] = r[12];
3. r[1] = r[1]{2;
4. r[12] = r[13] + .LOC;
5. r[12] = Load[r[12] + r[1]];

(a) original code

2. r[1] = r[12] – 8;

4. r[12] = r[13] + .LOC;
5. r[12] = Load[r[12] + (r[1]{2)];

(b) instruction selection followed by common subexpression elimination

1. r[12] = r[12] – 8;

4. r[0] = r[13] + .LOC;
5. r[12] = Load[r[0] + (r[12]{2)];

(c) common subexpression elimination followed by instruction selection

Register Remapping Removes False Register Dependency

```
1. r[12] = r[12] – 8;
2. r[1] = r[12];
3. r[1] = r[1]{2;
4. r[12] = r[13] + .LOC;
5. r[12] = Load[r[12] + r[1]];
```

(a) original code

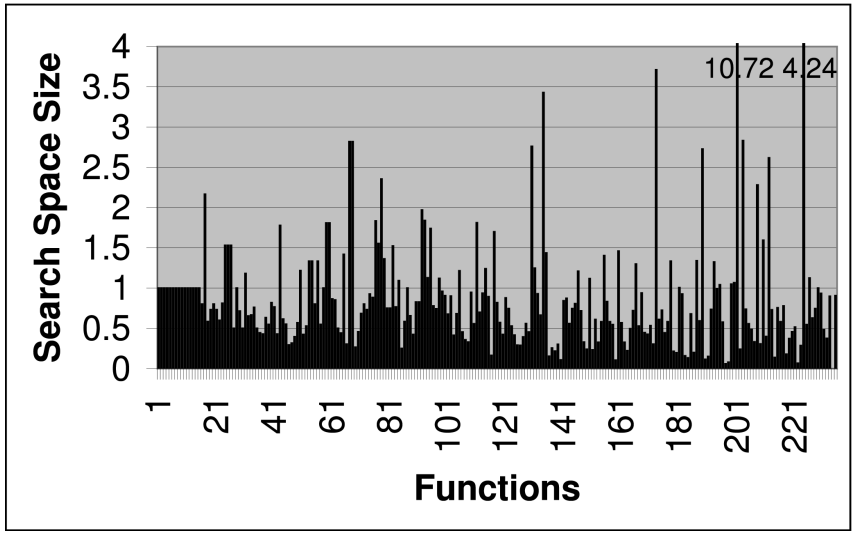
```
2. r[1] = r[12] – 8;

4. r[12] = r[13] + .LOC;
5. r[12] = Load[r[12] + (r[1]{2)];
```

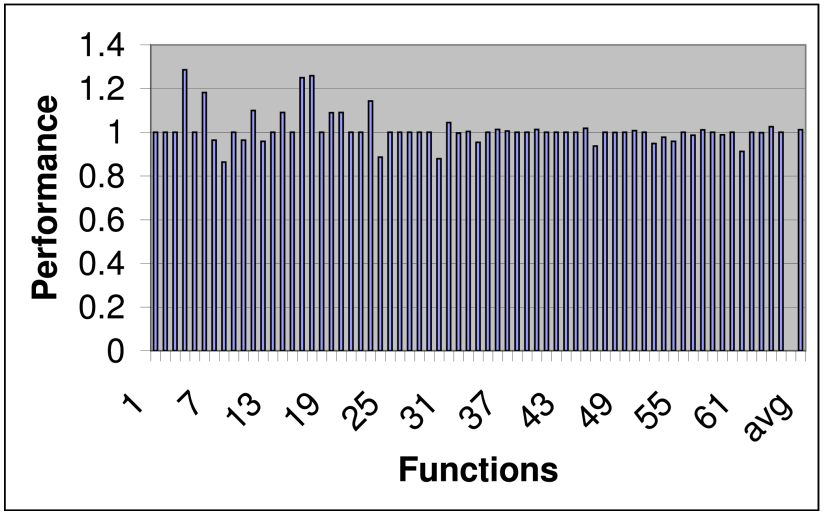
(b) instruction selection followed by common subexpression elimination

```
1. r[12] = r[12] – 8;

4. r[0] = r[13] + .LOC;
5. r[12] = Load[r[0] + (r[12]{2)];
```

(c) common subexpression elimination followed by instruction selection

Effect of Register Remapping (512 Registers)





- Avg search space size impact (233 functions): 9.5% per function reduction, 13% total reduction
- Avg performance impact (65 functions): 1.24% degradation

Other Notes on Register Remapping

- Register remapping is an *enabling* phase that can provide more opportunities for later optimizations.
- Including register remapping as the 16th reorderable phase in VPO causes an unmanageable increase in search space size for most functions.

Copy Propagation Removes False Register Dependences

```
1. r[18] = Load [L1];
2. r[7] = r[18];
3. r[21] = r[7];
4. r[24] = Load[r[21]];
5. r[5] = r[24];

6. ..... = r[7];
```

(a). original code

```
2. r[7] = Load[L1];

5. r[5] = Load[r[7]];

6. ..... = r[7];
```

(b) instruction selection followed by common subexpression elimination

```
1. r[18] = Load [L1];
2. r[7] = r[18];

5. r[5] = Load [r[18]];

6. ..... = r[7];
```

(c) common subexpression elimination followed by instruction selection

```
1. r[18] = Load [L1];
2. r[7] = r[18];

5. r[5] = Load [r[18]];

6. ..... = r[7];
```

(d) copy propagation removes false register dependence

Copy Propagation Removes False Register Dependences

```
1. r[18] = Load [L1];
2. r[7] = r[18];
3. r[21] = r[7];
4. r[24] = Load[r[21]];
5. r[5] = r[24];

6. ..... = r[7];
```

(a). original code

```
2. r[7] = Load[L1];

5. r[5] = Load[r[7]];

6. ..... = r[7];
```

(b) instruction selection followed by common subexpression elimination

```
1. r[18] = Load [L1];
2. r[7] = r[18];

5. r[5] = Load [r[18]];

6. ..... = r[7];
```

(c) common subexpression elimination followed by instruction selection

```
1. r[18] = Load [L1];
2. r[7] = r[18];

5. r[5] = Load [r[18]];

6. ..... = r[7];
```

(d) copy propagation removes false register dependence

Copy Propagation Removes False Register Dependences

```
1. r[18] = Load [L1];
2. r[7] = r[18];
3. r[21] = r[7];
4. r[24] = Load[r[21]];
5. r[5] = r[24];

6. ..... = r[7];
```

(a). original code

```
2. r[7] = Load[L1];

5. r[5] = Load[r[7]];

6. ..... = r[7];
```

(b) instruction selection followed by common subexpression elimination

```
1. r[18] = Load [L1];
2. r[7] = r[18];

5. r[5] = Load [r[18]];

6. ..... = r[7];
```

(c) common subexpression elimination followed by instruction selection

```
1. r[18] = Load [L1];

5. r[5] = Load [r[18]];

6. ..... = r[18];
```

(d) copy propagation removes false register dependence

Copy Propagation Removes False Register Dependences

```
1. r[18] = Load [L1];
2. r[7] = r[18];
3. r[21] = r[7];
4. r[24] = Load[r[21]];
5. r[5] = r[24];

6. ..... = r[7];
```

(a). original code

```
2. r[7] = Load[L1];

5. r[5] = Load[r[7]];

6. ..... = r[7];
```

(b) instruction selection followed by common subexpression elimination

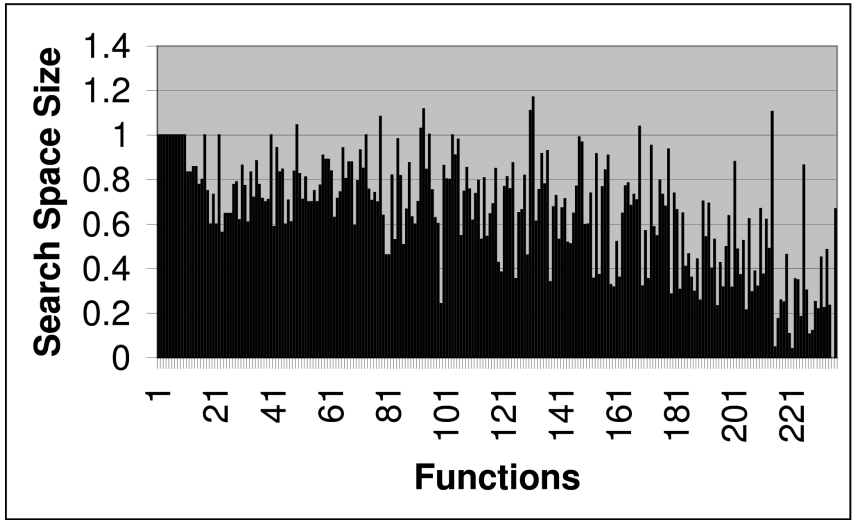
```
1. r[18] = Load [L1];
2. r[7] = r[18];

5. r[5] = Load [r[18]];

6. ..... = r[7];
```

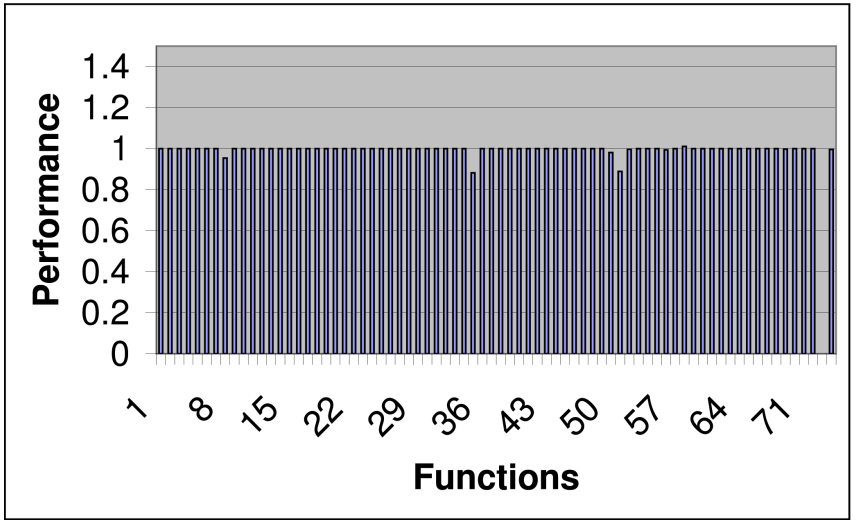
(c) common subexpression elimination followed by instruction selection

```
1. r[18] = Load [L1];

5. r[5] = Load [r[18]];

6. ..... = r[18];
```

(d) copy propagation removes false register dependence

Effect of Copy Propagation (512 Registers)



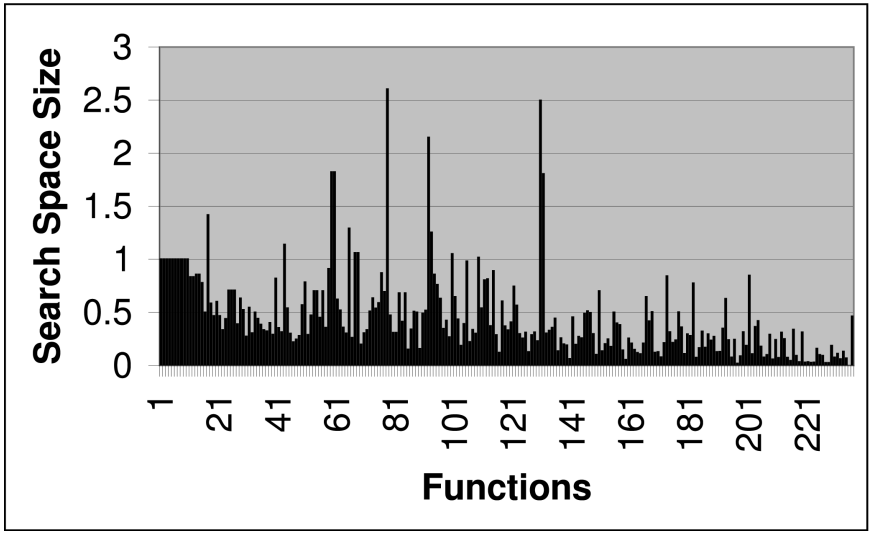


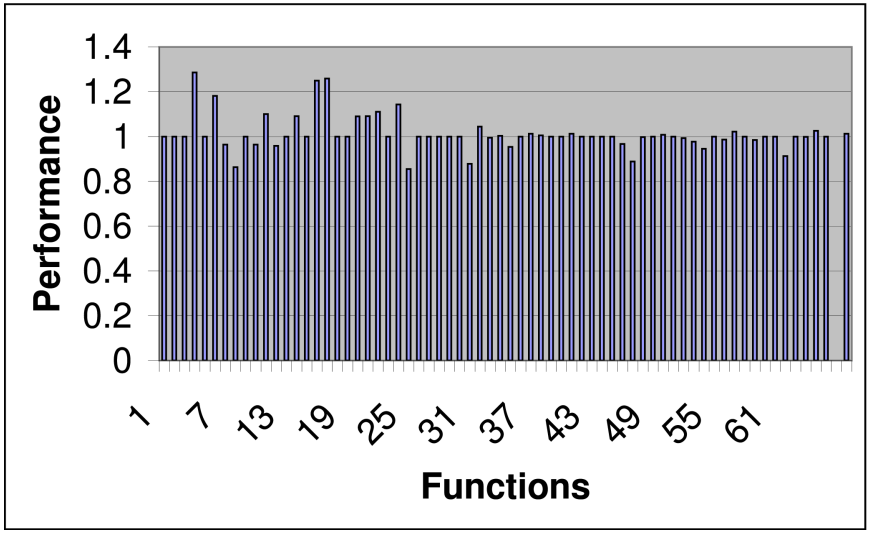
- Avg search space size impact (234 functions): 33% per function reduction, 67% total reduction
- Avg performance impact (72 functions): 0.41% improvement

Other Notes on Copy Propagation

- Copy propagation directly improves performance by eliminating copy instructions.
- Including copy propagation as the 16th reorderable phase during the phase order search:
  - Almost doubles the size of the phase order search (an increase of 98.8%) compared to the default VPO config
  - Has a negligible effect on the quality of code instances (0.06% improvement over the configuration with copy propagation implicitly applied)

Combining Register Remapping and Copy Propagation (512 Registers)





- Avg search space size impact (234 functions): 56.7% per function reduction, 88.9% total reduction
- Avg performance impact (66 functions): 1.24% degradation

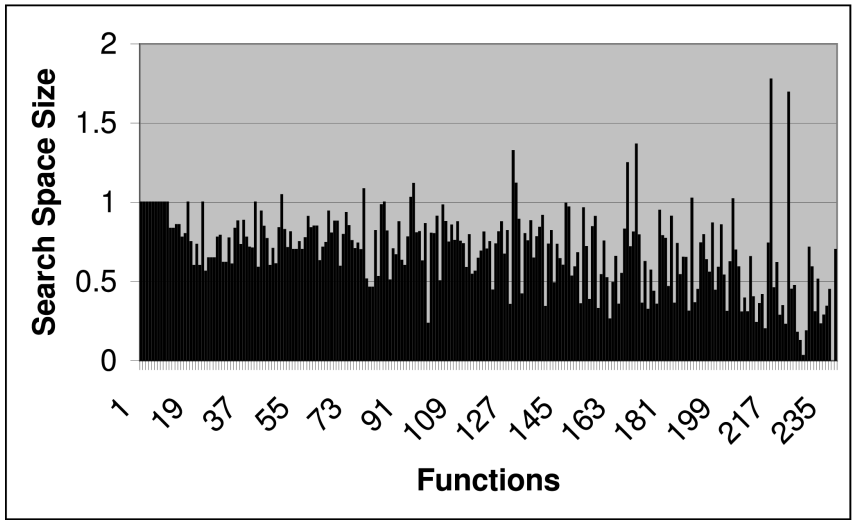
False Register Dependences on Real Embedded Architectures

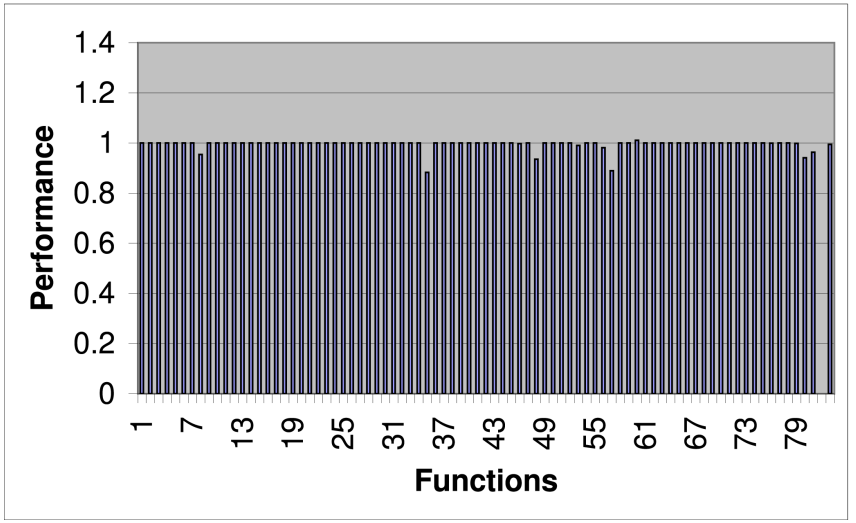
- Register remapping and copy propagation reduce the search space in a machine with unlimited registers
- Both transformations tend to increase register pressure, which affects the operation of successive phases.
- How can we adapt the behavior and application of these transformations to reduce search search space size on real embedded hardware?

Conservative Copy Propagation

- Aggressive application of copy propagation can increase register pressure and introduce spills.
- Can affect other optimizations, and may ultimately change the shape of the phase order search space.
- Thus, we develop a *conservative copy propagation*:
  - Only successful in situations where the copy instruction becomes redundant and can later be removed.
  - Always avoids increasing register pressure.

Effect of Conservative Copy Propagation (16 Registers)





- Avg search space size impact (236 functions): 30% per function reduction, 56.7% total reduction
- Avg performance impact (81 functions): 0.56% improvement

Conclusions

- Huge phase order search space is partly a result of interactions due to false register dependences.
  - Decreasing register pressure (by increasing the available registers) does not sufficiently eliminate false register dependences.
  - Register remapping and copy propagation:
    - Reduce false register dependences and substantially reduce the size of the phase order search space.
    - Increase register pressure to a point not sustainable on real machines
  - Prudent application of these techniques (e.g. conservative implementation of copy propagation) can be very effective on real machines.

KU THE UNIVERSITY OF KANSAS

Phase Independence

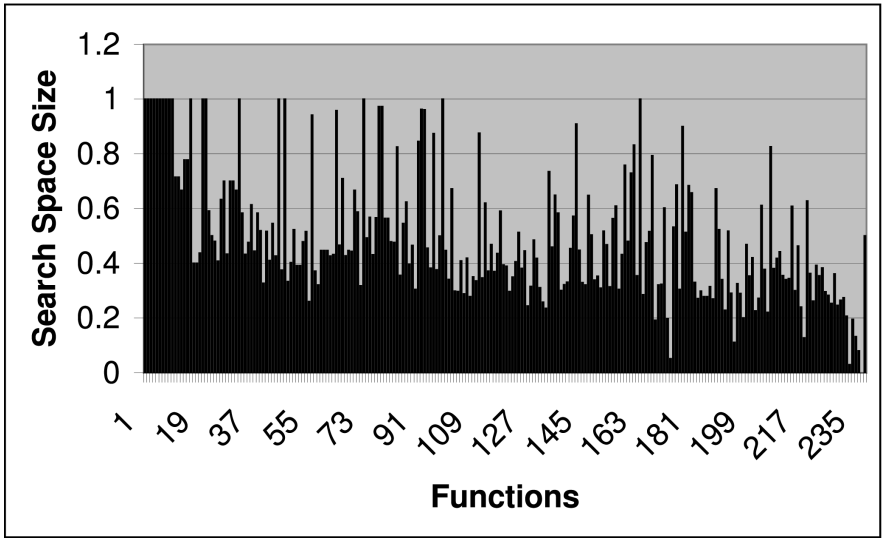
Phase Independence

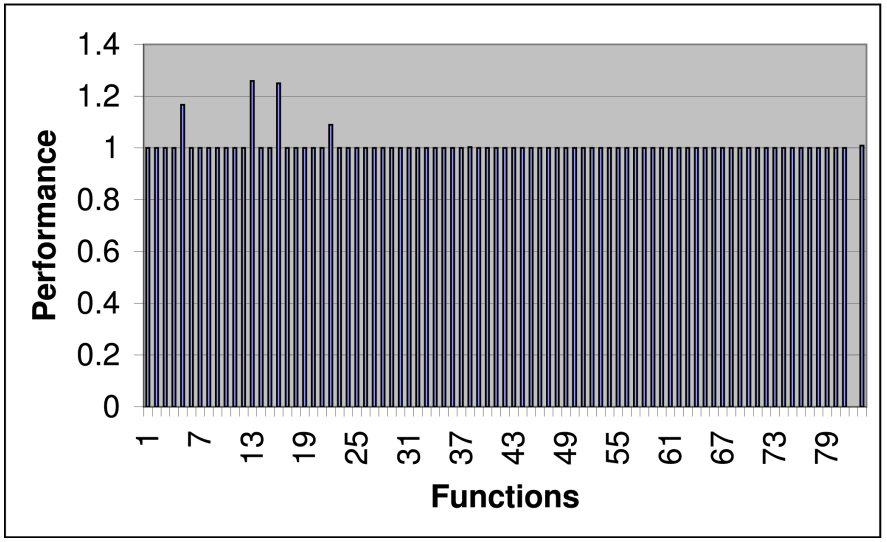
- Two phases are *independent* if applying them in different orders to any input code always leads to the same output code.
- Completely independent phases can be removed from the set employed during the exhaustive search and applied implicitly after every relevant phase
- In VPO, we have observed that very few phases are completely independent of each other.
- However, several phases show none to very sparse interaction activity.

Eliminating Cleanup Phases

- Cleanup phases:
  - e.g. *dead assignment elimination*, *dead code elimination*
  - Do not consume machine resources
  - Assist other phases by cleaning junk instructions / blocks left behind by other phases
  - Should be naturally independent from other phases
- Thus, implicit application of cleanup phases during the exhaustive phase order search should not have any impact on performance of other phases.

Effect of Removing DAE





- Avg search space size impact (236 functions): 50% per function reduction, 77% total reduction
- Avg performance impact (81 functions): 0.95% degradation

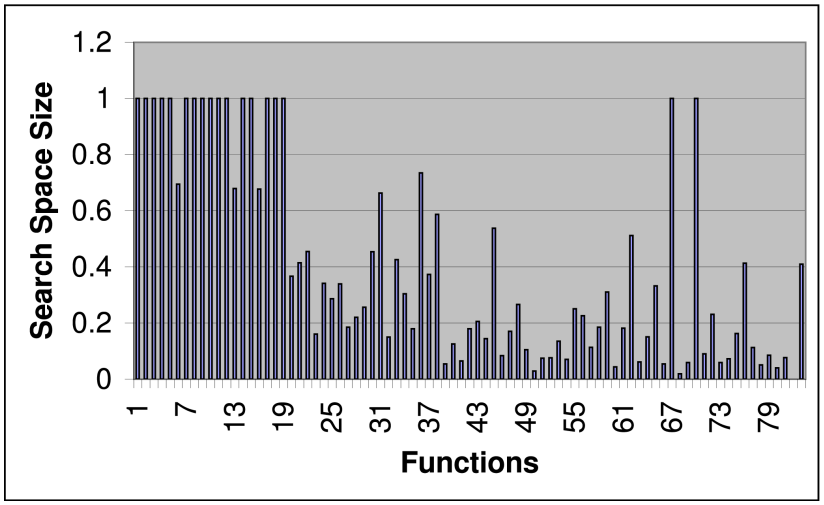
Branch and Non-branch Optimization Phases

- Set of optimization phases can be naturally partitioned into two subsets:
  - Phases that affect control-flow (branch optimizations)
  - Phases that depend on registers (non-branch optimizations)
- Intuitively, branch optimizations should be naturally independent from non-branch optimizations

Multi-stage Phase Order Search

- Branch optimizations may interact with other branch optimizations, and thus, cannot be simply removed from the set employed during the exhaustive search.
- Multi-stage Phase Order Search
  - Partition optimization set into branch / non-branch sets
  - In first stage, perform phase order search using only branch optimizations in the optimization set
  - Find best performing function instance(s)
  - Perform phase order search(es) using the non-branch optimizations and with the best performing function instance(s) as the starting code

Results of Multi-stage Phase Order Search



- Avg search space size impact (81 functions): 59% per function reduction, 88.4% total reduction
- Performance not affected in all 81 functions

Notes on the Multi-Stage Phase Order Search

- Performance degradations can occur if later stages do not include branch optimizations
  - Non-branch optimizations can *enable* branch optimizations by changing the control flow (e.g. by removing an instruction that results in removing a block).
- Partitioning of branch / non-branch optimizations requires intricate knowledge of compiler phases
  - Developed an algorithm to perform this partitioning automatically by analyzing independence relationships among phases.

Conclusions

- Removing cleanup phases (such as DAE) from the optimization set and applying these implicitly after every phase:
  - Reduces the search space size significantly
  - Does cause large performance degradations in a few cases
- Partitioning the optimization set into branch and non-branch optimizations and applying these in a staged fashion:
  - Reduces the search space to a fraction of its original size
  - Does not sacrifice performance in any function we tested

KU THE UNIVERSITY OF KANSAS

Future Work

Future Work

- Study other causes of false phase interactions
- Removal of false phase interactions should make remaining interactions more predictable
  - Does this improve the the efficiency / quality of solution of heuristic algorithms?
- Can we account for all performance degradations when cleanup phases are removed?
- What independence relationships can we deduce from data gathered by heuristic algorithms?

Thank you for listening.

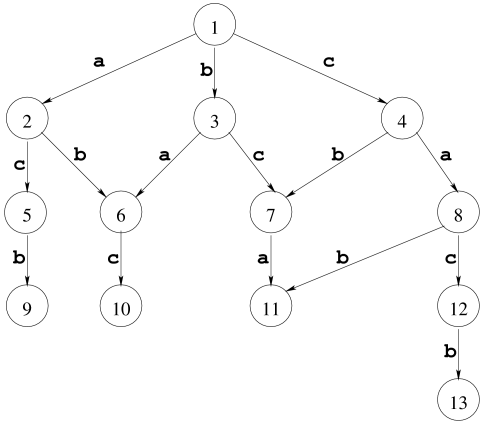
Questions?

Background

Search Space Enumeration

- Many optimization phase sequences produce the same code (*function instance*).
- Our approach for enumerating the search space:
  - Enumerate all possible function instances that can be produced by any combination of optimization phases for any possible sequence length.
  - Phase ordering search space viewed as a DAG of distinct function instances.
- Allows us to generate / evaluate the entire search space and determine the *optimal* function instance.

An Example DAG



- Nodes are function instances, edges are transition from one function instance to another on application of some phase
- Rooted at unoptimized function instance.
- Each successive level produced by applying all possible phases to distinct nodes at the preceding level.
- Terminate when no additional phase creates a new distinct function instance.

The Optimization Space

- 15 *optional* code-improving phases (see next slide)
- Can mostly be applied in arbitrary order, but there are a few restrictions
  - Optimizations that analyze values in registers must be performed after *register allocation*
- All optimizations, except *loop unrolling*, can be successfully applied only a limited number of times.
- *Loop unrolling* always uses a loop unroll factor of two and is only attempted once for each loop.

Optimization Phase	Code	Description
branch chaining	b	Replaces a branch/jump target with the target of the last jump in the chain.
common subexpression elimination	c	Performs global analysis to eliminate fully redundant calculations, which also includes global constant and copy propagation.
dead code elimination	d	Removes basic blocks that cannot be reached from the function entry block.
loop unrolling	g	To potentially reduce the number of comparisons and branches at run time and to aid scheduling at the cost of code size increase.
dead assignment elimination	h	Uses global analysis to remove assignments when the assigned value is never used.
block reordering	i	Removes a jump by reordering blocks when the target of the jump has only a single predecessor.
loop jump minimization	j	Removes a jump associated with a loop by duplicating a portion of the loop.
register allocation	k	Uses graph coloring to replace references to a variable within a live range with a register.
loop transformations	l	Performs loop-invariant code motion, recurrence elimination, loop strength reduction, and induction variable elimination on each loop ordered by loop nesting level.
code abstraction	n	Performs cross-jumping and code-hoisting to move identical instructions from basic blocks to their common predecessor or successor.
evaluation order determination	o	Reorders instructions within a single basic block in an attempt to use fewer registers.
strength reduction	q	Replaces an expensive instruction with one or more cheaper ones. For this version of the compiler, this means changing a multiply by a constant into a series of shift, adds, and subtracts.
branch reversal	r	Removes an unconditional jump by reversing a conditional branch when it branches over the jump.
instruction selection	s	Combines pairs or triples of instructions that are are linked by set/use dependencies. Also performs constant folding.
useless jump removal	u	Removes jumps and branches whose target is the following positional block.

Category	Program	#Lines	Description
auto	bitcount	584	test processor bit manipulation abilities
	qsort	45	sort strings using the quicksort sorting algorithm
network	dijkstra	172	Dijkstra's shortest path algorithm
	patricia	538	construct patricia trie for IP traffic
telecomm	fft	331	fast fourier transform
	adpcm	281	compress 16-bit linear PCM samples to 4-bit samples
consumer	jpeg	3575	image compression and decompression
	tiff2bw	401	convert color *tiff* image to b&w image
security	sha	241	secure hash algorithm
	blowfish	97	symmetric block cipher with variable length key
office	string-search	3037	searches for given words in phrases
	ispell	8088	fast spelling checker

```
1. r[12] = r[12] – 8;
2. r[1] = r[12];
3. r[1] = r[1]{2;
4. r[12] = r[13] + .LOC;
5. r[12] = Load[r[12] + [r[1]];
```

(a) original code

```
2. r[1] = r[12] – 8;
4. r[12] = r[13] + .LOC;
5. r[12] = Load [r[12] + (r[1]{2)];
```

(b) instruction selection followed by common subexpression elimination

```
1. r[12] = r[12] – 8;
3. r[1] = r[12]{2;
4. r[12] = r[13] + .LOC;
5. r[12] = Load [r[12] + r[1]];
```

(c) common subexpression elimination followed by instruction selection

- Example shows false register dependences produce distinct function instances
- Later and repeated application of optimization phases often cannot correct the effects of such register assignments
- Successive optimizations working on unique function instances produce even more distinct points, causing an explosion in the size of the phase order search space

Other Notes on Conservative Copy Propagation

Notes on Removing DAE

- Performance impact in 5 of 81 executed functions
- Degradations range from 0.4% to 25.9%
- Performance impact more significant in functions with smaller default search space size
- Detailed analysis suggests degradations stem from non-orthogonal use of registers and machine independent implementation of phases.