CS 516 Compilers and Programming Languages II

Compiler Optimizations (cont.)
The Optimizer (or Middle End)

Modern optimizers are structured as a series of passes

Typical Transformations

- Discover & propagate some constant value
- Move a computation to a less frequently executed place
- Specialize some computation based on context
- Discover a redundant computation & remove it
- Remove useless or unreachable code
- Encode an idiom in some particularly efficient form
The Optimizer - Scope/Granularity

Example: Discover & propagate some constant value (constant folding / propagation)

Local, global (intra-procedural), and inter-procedural optimization

Local: Basic block within a procedure

```
a := 2
b := 3
c := a + b
print (c)
```

Global: Control flow between basic blocks within a procedure

```
if (...) then {
  a := 2
  b := 3
} else {
  a := 3
  b := 2
}
c := a + b
print (c)
```

Inter-procedural: Control flow across procedure calls

```
procedure foo (a, b) {
  c := a + b // no side effects
  return (c)
}
procedure bar {
  ...
c := foo(2, 3)
  print (c)
  ...
d := foo(5, 5)
  print (d)
}
```
Local, global (intra-procedural), and inter-procedural optimization

**Local**: Basic block within a procedure

```
a := 2
b := 3
c := a + b
print (c)
```

**Global**: Control flow between basic blocks within a procedure

```
if (...) then {
    a := 2
    b := 3
} else {
    a := 3
    b := 2
}
c := a + b
print (c)
```

**Inter-procedural**: Control flow across procedure calls

```
procedure foo (a, b) {
    c := a + b // no side effects
    return (c)
}
procedure bar {
    ...
c := foo(2, 3)
    print (c)
    ...
d := foo(5, 5)
    print (d)
}
```

Control Flow Graph (CFG)

- Call Multi-Graph
  - bar
  - foo

Example: Discover & propagate some constant value (constant folding / propagation)
Example: Discover & propagate some constant value (constant folding / propagation)

Local, global (intra-procedural), and inter-procedural optimization

Local: Basic block within a procedure

\[ a := 2 \]
\[ b := 3 \]
\[ c := 5 \]

Optimization results

\[ \text{print (5)} \]

Global: Control flow between basic blocks within a procedure

\[ \text{if (...) then \{ } \]
\[ \quad a := 2 \]
\[ \quad b := 3 \]
\[ \text{\}} \text{ else \{ } \]
\[ \quad a := 3 \]
\[ \quad b := 2 \]
\[ \text{\}} \]
\[ c := 5 \]
\[ \text{print (5)} \]

Inter-procedural: Control flow across procedure calls

\[ \text{procedure foo (a, b) \{ } \]
\[ \quad c := a + b // no side effects \]
\[ \quad \text{return (c)} \} \]

\[ \text{procedure bar \{ } \]
\[ \quad \ldots \]
\[ \quad c := 5 \]
\[ \quad \text{print (5)} \]
\[ \quad \ldots \]
\[ \quad d := 10 \]
\[ \quad \text{print (10)} \} \]
Constant propagation/folding (local, global, inter-procedural)

Dead code elimination (local, global, inter-procedural)

CSE: common subexpression elimination (local, global)

Invariant code motion (global, inter-procedural)

Strength reduction, idioms recognition (local)

Procedure inlining (inter-procedural)
Modern restructuring compilers:
- Perform optimizations at different abstraction levels
- Start with highest abstractions, and the lower abstraction level
- Highest level: source-to-source transformations using AST

Typical Restructuring (source-to-source) Transformations:
- Blocking for memory hierarchy and register reuse
- Vectorization
- Parallelization
- Full and partial inlining
Static single assignment (value numbering)

Static Single Assignment (SSA) form is one of the key program representations in optimizing compilers

Dependence graph (blocking for cache, parallelization, register allocation, list scheduling, communication optimization)

Knob dependence graph (approximations)

Covered in a separate lectures

Some representations are complete, i.e., encode the full semantics of the program. Most representations are partial and focus on some program properties only (important summary information to support a specific optimization).
Optimization - Benefits

How to assess any technique (transformation) that will improve the overall program outcome or its \textit{dynamic} execution?

\textbf{(S) Safety:} Program semantics has to be preserved (true or false)

\textbf{(O) Opportunity:} How often can the optimization be safely applied during the execution of the program (percentage)

\textbf{(P) Profitability:} If the optimization is applied, what is the expected average benefit in terms of the target metric?

\textbf{Benefit} = \[ (100 - O) + \frac{O}{P} \] if \( S = \text{true} \)

Examples:

The transformation “a” is \textbf{safe} and improves the execution time of 10\% of the executed code by a factor of \textbf{5}.

Benefit: execution time reduced to 92\%.

The transformation “b” is \textbf{not safe} and improves the execution time 40\% of the executed code by a factor of \textbf{2}.

Benefit is not defined.

If “b” were \textbf{safe}, benefit: execution time reduction to 80\%.
How do these optimizations interact?

A significant body of research tries to find the best sequence of optimizing transformations for different application domains. These transformation are not Church-Rosser, i.e., the particular order of the transformations impact the overall outcome.

Some of the optimizations are used as “clean-up” passes (e.g.: constant propagation, dead code elimination). This allows implementers of other transformations to use simpler algorithms and data abstractions that are easier to reason about.

When you design an optimization pass, keep in mind that the program your optimizing pass is presented with may have run through many previous transformations, significantly changing the program’s code shape. Most likely, this code shape would not have been generated directly by any human programmer. Make sure your optimization path algorithms and data structures can deal with these “un-natural” shapes.
What do these optimizations have in common?

Their goal is to reduce the number of machine cycles needed to execute the program (reduce dynamic execution count).

Note: Reducing dynamic execution cycles does not always imply reducing static program size. In fact, many optimizations increase the program size significantly. This in turn can have negative impact on (dynamic) performance (e.g.: caches, failure of “standard” algorithms to generate good code).

Examples:
- Procedure inlining
- Blocking for memory hierarchy (in particular caches)
- Loop unrolling to increase basic block sizes
- Trace scheduling to increase size of basic blocks
What other optimization goals are there?

- Performance (dynamic execution time)
- Size of executable
- Power (peak power dissipation)
- Energy (battery life)
- Thermal (cooling)

How do these different optimization goals interact?

- Does one optimization goal subsumes another, or are they all different?
- Can one optimization goal conflict with another? (e.g.: power vs. performance, thermal vs. performance)
Things to do / Next time

Read papers posted on our web site

More detailed discussion of compile-time power/energy optimization