RUTGERS
THE STATE UNIVERSITY
OF NEW JERSEY

CS415 Compilers

Compiler Optimizations

These slides are based on slides copyrighted by
Keith Cooper, Ken Kennedy & Linda Torczon at Rice
University

Announcements

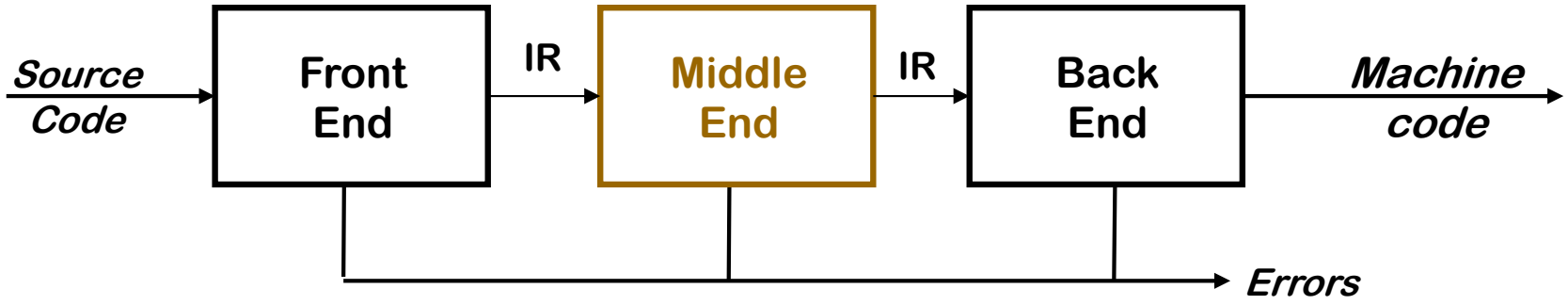
Roadmap for the remainder of the course

- Project #2 – Bottom-up parser and compiler
  New due date: Wednesday April 20
- Homework #5 due today.
- Project #3 – Will be posted by Thursday
- Final exam on May 10 , 1:00pm, (60 minutes in class)
- Grading Scheme
  - → Exams: 2 x 30% (best two exams count)
  - → Projects: 3 x 10%
  - → Homeworks: 5 x 2% (best five homeworks count)

Compiler Optimizations

ALSU Chapter 9

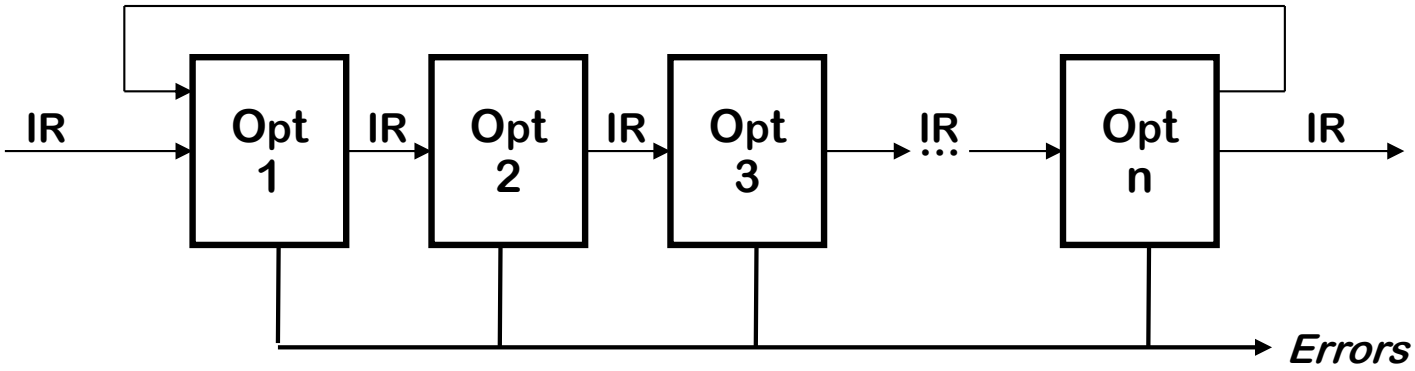
Traditional Three-pass Compiler



Code Improvement (or *Optimization*)

- Analyzes IR and rewrites (or *transforms*) IR
- Primary goal is to reduce running time of the compiled code
  - → May also improve space, power dissipation, energy consumption, …
- Must preserve "meaning" of the code (may include approximations, i.e., quality of outcomes tradeoffs)
  - → Measured by values of named variables or produced output

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

Optimization - Benefits

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

(S) Safety: Program semantics has to be preserved (true or false)

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

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

Benefit = [(100 – O) + O/P] if S = true

Examples:

The transformation "a" is **safe** and improves the execution time of **10%** of the executed code by a factor of **5**.

Benefit: execution time reduced to 92%

The transformation "b" is **not safe** and improves the execution time **40%** of the executed code by a factor of **2.**

Benefit is not defined

If "b" were **safe**, benefit: execution time reduction to 80%

The Optimizer – Interactions

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.

Performance Optimizations

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

Optimizations

What other optimization goals are there?

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

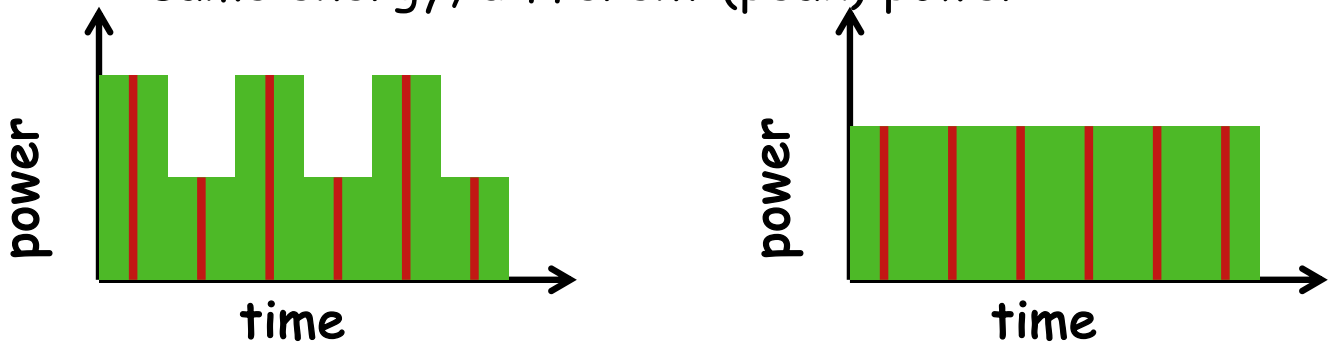
How do these different optimization goals interact?

- 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)

Power vs. Energy vs. Thermal

power (when): activity level at a given point in time
energy (what): total amount of activity
thermal (where): location of activity / power density

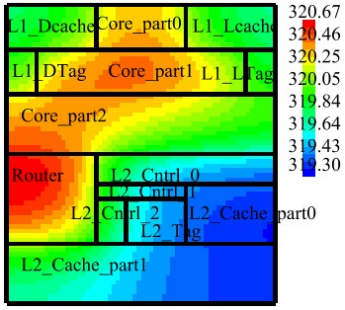
same energy, different (peak) power



Thermal optimization: Spread activities across spatial dimensions: Where to do things? A larger surface is easier to cool!

Thermal map of a multi-core system

Source: Zhu et al., DATE'15



A10 Gatling gun

Plate of hot mashed potatoes

There is no "free" lunch!

You cannot have everything, so something has to give (Pareto optimal)

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)
}
```

The Optimizer – Scope/Granularity

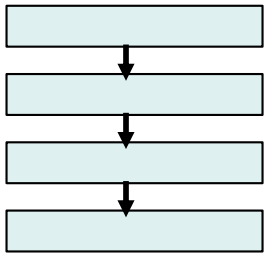
Example: Discover & propagate some constant value (constant folding / propagation)

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

Local: Basic block within a procedure

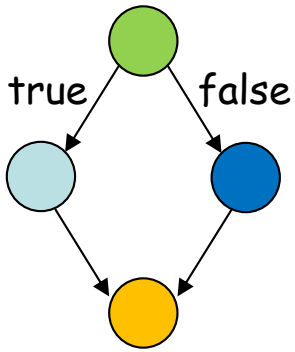
```
a := 2
b := 3
c := a + b
print (c)
```

List of statements



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)
```



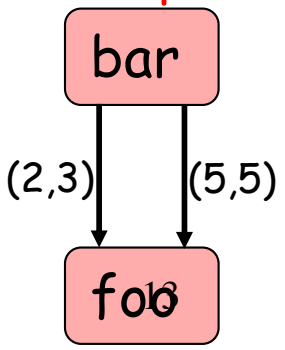
Control Flow Graph (CFG)

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)
}
```

Call Multi-Graph



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 := 5
print (5)
```

optimization results

Global: Control flow between basic blocks within a procedure

```
if (…) then {
   a := 2
   b := 3
} else {
   a := 3
   b := 2
}
c := 5
print (5)
```

Inter-procedural: Control flow across procedure calls

```
procedure foo (a, b) {
c := a + b // no side effects
return (c) }

procedure bar {
…
c := 5
print (5)
…
d := 10
print (10)
}
```

The Optimizer – Scope/Granularity

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)

Things to do and next class

Work on the project!

Procedure abstraction

Read EaC: Chapter 6.1 – 6.5