263-2810: Advanced Compiler Design

Thomas R. Gross

Computer Science Department
ETH Zurich, Switzerland
Topics

- Program optimization
  - “Optimization”
  - Optimize for (execution) speed
  - Optimize for (code) size
  - Optimize for energy consumption
Topics

- **Program optimization**
  - “Optimization”
  - Optimize for (execution) speed
  - Optimize for (code) size
  - Optimize for energy consumption

- **Program analysis**

- **Program transformation**

- **Compiler engineering**
  - Compile time vs. code quality
Topics -- continued

- Ahead-of-time compilers
- Just-in-time compilers
  - Multi-tier compilation systems
- Binary translators
  - “Special purpose compilers”
- Source language(s): imperative, typed object-oriented (class-based) programming languages
  - Excursions to other language worlds
Logistics

- **Lectures**
  - Wednesday 10-12
  - Friday 13-14

- **Recitations**
  - Friday 14-16

- **Place:** CAB G 51

[www.lst.inf.ethz.ch](http://www.lst.inf.ethz.ch)
Learning by doing

- Lectures present concepts, algorithms, tradeoffs
- Recitations repeat, provide context

Course credit given if you

- Do homework assignments (3 planed)
  - 50% of the grade
- Develop and present a (compiler) project
  - 50% of the grade
- Assignments and projects done in teams of 2 students
Assignments

- Build/extend a compiler
- Focus on program optimization
  - Preparation for project
- Assignments: build compiler infrastructure, implement optimizations
- Project: define and investigate a compiler question
Learning by doing

- The work you turn in must be your work.
  - You must review your partner’s work
- Turning in the work of others is cheating.

- ETH has a set of rules that you should know. Bottomline: do not copy a design or code (or text ...) without proper attribution.
  - In papers or a report you must identify and cite work not performed by you.
  - For programs, you must identify your source(s)
    - Attribution must be clearly visible
    - Code reuse is fine – but not presenting the work of others as your work
  - Talk to the instructor/TA if necessary
Recitations

- Review material, introduce framework, etc.

- Meetings when announced in class

- May swap lectures and recitations on Friday
  - Esp. early in semester
  - Cover material for later projects
  - *Friday is not optional*
Question

- How do we want to organize Friday afternoon?
  - **Scenario 1: one or two hours of lecture**
    - Start at 13:15
    - Start at 14:15
    - Start at 15:15
  
  - **Scenarios 2: only recitation**
    - Start at 13:15
    - Start at 14:15
    - Start at 15:15
1.0 Introduction

- **Central topic: optimization techniques for modern programming languages**
  - Key issues for real compilers
  - Techniques have many uses

- **“Optimization” – resulting code is rarely optimal**
  - Often impossible to prove optimality
  - May have proofs for an abstract execution engine (virtual machine) but tie to real systems difficult
  - Interested in improvement
    - “better than naïve translation”
    - “pretty good”
    - incremental transformations
1.1 Intermediate representation

- IR (Intermediate representation): output of front-end, input to backend

- Optimizer IR $\rightarrow$ IR
  - Or separate IRs: IR1 and IR2 (or high-level IR, low-level IR)
IR concerns

- Abstract from unnecessary features of programming language
  - Support multiple front-ends
- Decouple front-end and back-end
  - Multiple back-ends
- Keep essential information
  - Structure
  - Types
- Expose operations
Simple compiler: Tree-based IR

- One assignment stmt: one *binary* tree
  ```java
  int A, B, C, D;
  A = B + C * D;
  ```

- A tree has a root
  - Top node
  - Right and left subtrees

- Sequences of statements: *forest* of trees
\[ x = a + b; \]
\[ d = x + 1; \]
\[ b = a + c; \]
\[ \text{if} \ (\ldots) \ \{ \ldots \} \]
\[ y = b + x \]
Assessment of trees

+ simple
  easy to debug
  local - small can be analyzed in isolation

- LR (tree) is not suited for optimization
  no reuse of values
  X is loaded multiple times,
  X is computed, loaded
  no model of storage hierarchy (register,)
  no local info must be gathered (cert data)
  overspecification (3rd cert proc and 2nd)
Assessment of trees

- Simple
- Not a good IR for optimization
  - No reuse of values
  - No model of storage hierarchy
- Overspecification
- Global aspects (e.g., last writer for a variable) hidden
1.2 Requirements for IR

- Make data dependences explicit
  - Capture producer – consumer relationship

- Make control dependences explicit
  - Determine essential order of execution that *must* be honored
Control dependence

Captures if execution of one \( \{ \text{block} \} \) depends on the function call

execution of some other \( \{ \text{block} \} \) function call
Requirements for IR

- Make data dependences explicit
  - Capture producer – consumer relationship

- Make control dependences explicit
  - Determine essential order of execution that *must* be honored

- Ideally the IR contains only true (essential) constraints
  - No superfluous constraints
Control dependence graph

- CDG (control dependence graph) captures control dependences

- Nodes: basic blocks (BB)

- A basic block is a *maximal* sequence of statements that are always executed together
  - Unless there is an exception – these are usually handled differently
IR concerns

- Make data dependences explicit
  - Capture producer – consumer relationship

```plaintext
// 1 // x = a + b;
// 2 // d = x + 1;
// 3 // b = a + c;
```

- x produced by 1, consumed by 2
- b in 1 must be read before b is written in 3
Simple solution: execute 1, 2, and 3 in order

What matters is 1 reads the *old* value and 3 produces a *new* value

Idea: rename variable and/or make copies

```plaintext
// 1 //   x = a + foo ;
// 2 //   d = x + 1;
// 3 //   bar = a + c;
```
Compiler introduces a new name whenever a new value is produced.

Could use any name but usually use subscripts: \(b_1, b_2, \ldots\)

```plaintext
// 1 //  x = a + b_{34} ;
// 2 //  d = x + 1;
// 3 //  b_{56} = a + c;
```

IR used in optimizing compilers: static single assignment (SSA)