Optimizing Compilers:
Background

Structure of Optimizing Compilers

Source Program

Front ends
Front-end #1

Program Database

High-level Optimizer

Low-level Optimizer

Lowering of IL

Optimized HIL

Optimized LIL

Low-level Intermediate Language
LIL

High-level Intermediate Language
HIL

Target-1 Executable

Target-2 Executable

Target-3 Executable

Code Generator and Linker

Code Generator and Linker

Code Generator and Linker

Runtime Systems

Target-1 Executable

Target-2 Executable

Target-3 Executable

ECE 4100/6100 (1)
High-level Optimizer

The following steps may be performed more than once:

1. Global intraprocedural and interprocedural analysis of source program's control and data flow
2. Selection of high-level optimizations and transformations
3. Update of high-level intermediate language

Note: this optimization phase can optionally be bypassed, since its input and output use the same intermediate language

Lowering of Intermediate Language

- Linearized storage mapping of variables
- Array/structure references $\rightarrow$ load/store operations
- High-level control structures $\rightarrow$ low-level control flow
Low Level Optimizations

- Instruction Scheduling
  - ILP
- Register Allocation
  - Storage

Decreasing cost/bit

Cost of Instruction Scheduling

- Given a program segment, the goal is to execute it as quickly as possible
- The completion time is the *objective function* or cost to be minimized
- This is referred to as the *makespan* of the schedule
- It has to be balanced against the running time and space needs of the algorithm for finding the schedule, which translates to *compilation cost*
Instruction Scheduling Example

```c
main(int argc, char *argv[])
{
    int a, b, c;
    a = argc;
    b = a * 255;
    c = a * 15;
    printf("%d\n", b*b - 4*a*c);
}
```

Assumptions:
- max ILP = 4
- multiply instruction latency = 3 cycles
- all other instructions’ latencies = 1 cycle
Cost of Register Allocation

- The goal or Objective Function in this case is to maximize the duration that operands are in registers.

  Operations on register operands are much faster.

- Registers are unfortunately a limited resource.

- Therefore, when operands do not fit as is quite often the case in registers, then they are "spilled" to memory at the expense of additional cost.
Cost of Register Allocation (Contd.)

- Therefore, *maximizing* the duration of operands in registers or *minimizing* the amount of spilling, is the goal
- Once again, the running time (complexity) and space used, of the algorithm for doing this is the compilation cost

Register Allocation - An Example

Before

```
op 10 MPY vr2 ← param1, 255
op 12 MPY vr3 ← param1, 15
op 14 MPY vr8 ← vr2, vr2
op 15 SHL vr9 ← param1, 2
op 16 MPY vr10 ← vr9, r3
op 17 SUB param2 ← vr8, r10
op 18 MOV param1 ← addr("%d\n")
op 27 PBRR vb12 ← addr(printf)
op 20 BRL ret_addr ← vb12
```

After

```
spill code

op 10 MPY r2 ← param1, 255
op 12 MPY r3 ← param1, 15
op 15 SHL r4 ← param1, 2
op 27 PBRR b2 ← addr(printf)
spill code

op 18 MOV param1 ← addr("%d\n")
op 14 MPY r6 ← r2, r2
op 16 MPY r5 ← r4, r3
op 17 SUB param2 ← r6, r5
op 37 ADD temp_reg ← sp, 24
op 38 ST temp_reg, b2
op 20 BRL ret_addr ← b2
```

(assuming only 2 branch registers are available)
Execution Frequencies?

What are Execution Frequencies

- Branch probabilities
- Average number of loop iterations
- Average number of procedure calls

```c
if (a < b)
c = a
c = b
if (d < e)
f = d
f = e
```
How are Execution Frequencies Used?

- Focus optimization on most frequently used regions
  - region-based compilation
- Provides quantitative basis for evaluating quality of optimization heuristics

How are Execution Frequencies Obtained?

- Profiling tools:
  - Mechanism: sampling vs. counting
  - Granularity = procedure vs. basic block
- Compile-time estimation:
  - Default values
  - Compiler analysis
  - Goal is to select same set of program regions and optimizations that would be obtained from profiled frequencies
What are Execution Costs?

Cost of intermediate code operation parametrized according to target architecture:

- Number of target instructions
- Resource requirement template
- Number of cycles

How are Execution Costs Used?

In conjunction with execution frequencies:

- Identify most time-consuming regions of program
- Provides quantitative basis for evaluating quality of optimization heuristics
How are Execution Costs Obtained?

- Simplistic translation of intermediate code operation to corresponding instruction template for target machine

Graphs and Preliminaries
Basic Graphs

- A graph is made up of a set of nodes (V) and a set of edges (E)

- Each edge has a source and a sink, both of which must be members of the nodes set, i.e. $E = V \times V$

- Edges may be directed or undirected
  - A directed graph has only directed edges
  - A undirected graph has only undirected edges

Examples

Undirected graph

Directed graph
**Paths**

- **Undirected graph**
  - Source
  - Sink
  - Path

- **Directed graph**

**Cycles**

- **Undirected graph**
  - Directed graph
  - Acyclic Directed graph
Connected Graphs

Unconnected graph

Connected directed graph

Connectivity of Directed Graphs

- A strongly connected directed graph is one which has a path from each vertex to every other vertex.

- Is this graph strongly connected?
Data Dependence Analysis

If two operations have potentially interfering data accesses, data dependence analysis is necessary for determining whether or not an interference actually exists. If there is no interference, it may be possible to reorder the operations or execute them concurrently.

The data accesses examined for data dependence analysis may arise from array variables, scalar variables, procedure parameters, pointer dereferences, etc. in the original source program.

Data dependence analysis is conservative, in that it may state that a data dependence exists between two statements, when actually none exists.

Data Dependence: Definition

A *data dependence*, \( S_1 \rightarrow S_2 \), exists between CFG nodes \( S_1 \) and \( S_2 \) with respect to variable \( X \) if and only if

1. there exists a path \( P: S_1 \rightarrow S_2 \) in CFG, with no intervening write to \( X \), and

2. at least one of the following is true:

   (a) *(flow)* \( X \) is written by \( S_1 \) and later read by \( S_2 \), or

   (b) *(anti)* \( X \) is read by \( S_1 \) and later is written by \( S_2 \) or

   (c) *(output)* \( X \) is written by \( S_1 \) and later written by \( S_2 \)
Summary

- High level structure of optimizing compilers

- Our interest is on the development and incorporation of compile-time Optimizations
  - Driven by an objective metric – generally improvements in time, space or power.

- The use of execution profile data to
  - Find targets of optimization
    - Program regions, e.g., hot spots
    - Frequency and execution time
    - Resource demand
  - Evaluate the quality of an optimization