CS 701

Charles N. Fischer

Fall 2014

http://www.cs.wisc.edu/~fischer/cs701.html

In Memoriam

Susan B. Horwitz

1955 - 2014

Class Meets

Mondays and Wednesdays,
11:00 — 12:15

2540 Engineering Hall

Instructor

Charles N. Fischer

5393 Computer Sciences

Telephone: 262-1204

E-mail: fischer@cs.wisc.edu

Office Hours:

10:30 - Noon, Tuesdays and Thursdays, or by appointment

Key Dates

- **September 22: Project 1 due**
- **October 15: Project 2 due**
- **November 5: Project 3 due**
- **November 27: Midterm (tentative)**
- **December 12: Project 4 due**
- **December ??: Final Exam, date to be determined**

Class Text

- **Crafting a Compiler**

  Fischer, Cytron, LeBlanc

  ISBN-10: 0136067050

  ISBN-13: 9780136067054

  Publisher: Addison-Wesley

- **Handouts and Web-based reading will also be used.**

Reading Assignment

- **Section 14.1 - 14.2.2 of CaC**
- **Pages 1 - 30 of "Automatic Program Optimization"**
- **Assignment 1**

Class Notes

- **The lecture notes used in each lecture will be made available prior to that lecture on the class Web page (under the "Lecture Nodes" link).**

Piazza

Piazza is an interactive online platform used to share class-related information. We recommend you use it to ask questions and track course-related information. If you are enrolled (or on the waiting list) you should have already received an email invitation to particapate (about one week ago).

Instructional Computers

We have access to departmental 64 bit Linux boxes (macaroni-01 to macaroni-09) for general class-related computing. These machines have access to LLVM 3.3 at

`/unsup/llvm-3.3`

If you have access to a Linux box in your office connected to AFS, it will probably work fine for class projects.

CS701 Projects

1. Introduction to LLVM and Simple Local Optimization
2. Dataflow Analysis and Optimization
3. Natural Loops and Loop-Invariant Code Motion
4. Graph Coloring Register Allocation

Academic Misconduct Policy

- **You must do your assignments—no copying or sharing of solutions.**
- **You may discuss general concepts and Ideas, especially on Piazza.**
- **All cases of Misconduct *must* be reported.**
- **Penalties may be severe.**

Partnership Policy

Projects may be done individually *or* by two person teams (your choice).

Guest Lecturers

1. **Tom Reps**
2. **Somesh Jha**
3. **Ben Liblit**

Overview of Course Topics

1. Register Allocation

Local Allocation

Avoid unnecessary loads and stores within a *basic block*. Remember and reuse register contents.
Consider effects of *aliasing*.

Global Allocation

Allocate registers within a single subprogram. Choose "most profitable" values. Map several values to the *same* register.

Interprocedural Allocation

Avoid saves and restores across calls. Share globals in registers.

2. Code Scheduling

We can reorder code to reduce latencies and to maximize ILP (*Instruction Level Parallelism***). We must respect** *data dependencies* **and** *control dependencies***.**

```
ld  [a],%r1          ld[a],%r1
add %r1,1,%r2        mov 3,%r3
mov 3,%r3            add %r1,1,%r2
```

(before) **(after)**

3. Automatic Instruction Selection

How do we map an IR (*Intermediate Representation*) into Machine Instructions?

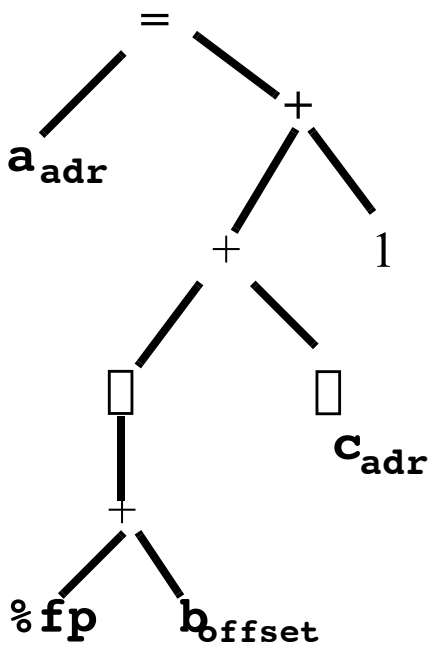
Can we guarantee the *best* instruction sequence?

Idea—Match instruction patterns (represented as trees) against an IR that is a low-level tree. Each match is a generated instruction; the best overall match is the best instruction sequence.

Example:

```
a=b+c+1;
```

In IR tree form:



Generated code:

```
ld   [%fp+b_offset],%r1
ld   [c_adr],%r2
add  %r1,%r2,%r3
add  %r3,1,%r4
st   %r4,[a_adr]
```

Why use four *different* registers?

4. Peephole Optimization

Inspect generated code sequences and replace pairs/triples/tuples with better alternatives.

```
ld  [a],%r1          ld  [a],%r1
mov const,%r2        add %r1,const,%r3
add %r1,%r2,%r3
```

(before) **(after)**

```
mov 0,%r1            OP  %g0,%r2,%r3
OP  %r1,%r2,%r3
```

(before) **(after)**

But why not just generate the better code sequence to begin with?

5. Cache Improvements

We want to access data & instructions from the L1 cache whenever possible; misses into the L2 cache (or memory) are *expensive*!

We will layout data and program code with consideration of cache sizes and access properties.

6. Local & Global Optimizations

Identify unneeded or redundant code.

Decide where to place code.

Worry about debugging issues (how reliable are current values and source line numbers after optimization?)

7. Program representations

- **Control Flow Graphs**
- **Program Dependency Graphs**
- **Static Single Assignment Form (SSA)**

  Each program variable is assigned to in only *one* place.

  After an assignment $x_i = y_j$, the relation $x_i = y_j$ *always* holds.

  Example:

```
if (a)                  if (a)
     x = 1                   x1 =1
else x = 2;             else x2 =2;
print(x)                x3 = □(x1,x2)
                        print(x3)
```

8. Data Flow Analysis

Determine invariant properties of subprograms; analysis can be extended to entire programs.

Model abstract execution.

Prove correctness and efficiency properties of analysis algorithms.

9. Points-To Analysis

All compiler analyses and optimizations are limited by the potential effects of assignments through pointers and references.

Thus in C:

```
b = 1;
*p = 0;
print(b);
```

is 1 or 0 printed?

Similarly, in Java:

```
a[1] = 1;
b[1] = 0;
print(a[1]);
```

is 1 or 0 printed?

Points-to analysis aims to determine what variables or heap objects a pointer or reference may access. Exact analysis is impossible (why?). But fast and reasonably accurate analyses are known.

Review of Compiler Optimizations

1. Redundant Expression Elimination (Common Subexpression Removal)

Use an address or value that has been previously computed. Consider control and data dependencies.

2. Partially Redundant Expression (PRE) Elimination

A variant of Redundant Expression Elimination. If a value or address is redundant along *some* execution paths, add computations to other paths to create a fully redundant expression (which is then removed).

Example:

```
if (i > j)
   a[i] = a[j];
a[i] = a[i] * 2;
```

3. Constant Propagation

If a variable is known to contain a particular constant value at a particular point in the program, replace references to the variable at that point with the constant value.

4. Copy Propagation

After the assignment of one variable to another, a reference to one variable may be replaced with the value of the other variable (until one or the other of the variables is reassigned).

(This may also "set up" dead code elimination. Why?)

5. Constant Folding

An expression involving constant (literal) values may be evaluated and simplified to a constant result value. Particularly useful when constant propagation is performed.

6. Dead Code Elimination

Expressions or statements whose values or effects are unused may be eliminated.

7. Loop Invariant Code Motion

An expression that is *invariant* in a loop may be moved to the loop's header, evaluated once, and reused within the loop.

***Safety* and *profitability* issues may be involved.**

8. Scalarization (Scalar Replacement)

A field of a structure or an element of an array that is repeatedly read or written may be copied to a local variable, accessed using the local, and later (if necessary) copied back.

This optimization allows the local variable (and in effect the field or array component) to be allocated to a register.

9. Local Register Allocation

Within a *basic block* **(a straight line sequence of code) track register contents and reuse variables and constants from registers.**

10. Global Register Allocation

Within a subprogram, frequently accessed variables and constants are allocated to registers. Usually there are *many more* **register candidates than available registers.**

11. Interprocedural Register Allocation

Variables and constants accessed by more than one subprogram are allocated to registers. This can *greatly* reduce call/ return overhead.

12. Register Targeting

Compute values directly into the intended target register.

13. Interprocedural Code Motion

Move instructions across subprogram boundaries.

14. Call Inlining

At the site of a call, insert the body of a subprogram, with actual parameters initializing formal parameters.

15. Code Hoisting and Sinking

If the same code sequence appears in two or more alternative execution paths, the code may be *hoisted* to a common ancestor or *sunk* to a common successor. (This reduces code size, but does not reduce instruction count.)

16. Loop Unrolling

Replace a loop body executed N times with an expanded loop body consisting of M copies of the loop body. This expanded loop body is executed N/M times, reducing loop overhead and increasing optimization possibilities within the expanded loop body.

17. Software Pipelining

A value needed in iteration *i* of a loop is computed during iteration *i-1* (or *i-2*, ...). This allows long latency operations (floating point divides and square roots, low hit-ratio loads) to execute in parallel with other operations. Software pipelining is sometimes called *symbolic loop unrolling*.

18. Strength Reduction

Replace an expensive instruction with an equivalent but cheaper alternative. For example a division may be replaced by multiplication of a reciprocal, or a list append may be replaced by cons operations.

19. Data Cache Optimizations

- **Locality Optimizations**

  Cluster accesses of data values both spacially (within a cache line) and temporally (for repeated use).
  Loop interchange **and** *loop tiling* **improve temporal locality.**

- **Conflict Optimizations**

  Adjust data locations so that data used consecutively and repeatedly don't share the same cache location.

20. Instruction Cache Optimizations

Instructions that are repeatedly executed should be accessed from the instruction cache rather than the secondary cache or memory. Loops and "hot" instruction sequences should fit within the cache.

Temporally close instruction sequences should not map to conflicting cache locations.