CS 701

Charles N. Fischer

Fall 2003

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

Class Meets

Tuesdays & Thursdays, 11:00 — 12:15

2321 Engineering Hall

Instructor

Charles N. Fischer

5397 Computer Sciences

Telephone: 262-6635

E-mail: fischer@cs.wisc.edu

Office Hours:

10:30 - Noon, Mondays & Wednesdays, or by appointment

Teaching Assistant

Kent Hunter

1308 Computer Sciences

Telephone: 262-6602

E-mail: khunter@cs.wisc.edu

Office Hours:

10:00 - 11:00
Tuesdays and Thursdays,
or by appointment

Key Dates

- September 23: Project 1 due
- October 21: Project 2 due (tentative)
- October 23: Midterm (tentative)
- November 18: Project 3 due (tentative)
- December 11: Project 4 due
- December ??: Final Exam, date to be determined

Class Text

There is no required text.

Handouts and Web-based reading will be used.

Suggested reference:

Advanced Compiler Design & Implementation,
by Steven S. Muchnick,
published by Morgan Kaufman.

Instructional Computers

Departmental SPARC Processors (nova1-nova60)

You may use your own workstation if it is has a SPARC processor
(test using **`dmesg|grep cpu`**)

Otherwise log onto a SPARC processor to do SPARC-specific assignments

CS701 Projects

1. SPARC Code Optimization
2. Global Register Allocation
   (using Graph Coloring)
3. Global Code Optimizations
4. Individual Research Topics

Academic Misconduct Policy

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

Reading Assignment

- Get Handout #2 (Chapter 15, Code Optimization) from DoIt.
- Read Chapters 0-6 and Appendices G&H of the SPARC Architecture Manual. Also skim Appendix A.
- Read section 15.2 of Chapter 15.
- Read Assignment #1

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*.

(before)	(after)
`ld [a],%r1`	`ld [a],%r1`
`add %r1,1,%r2`	`mov 3,%r3`
`mov 3,%r3`	`add %r1,1,%r2`

3. Automatic Instruction Selection

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

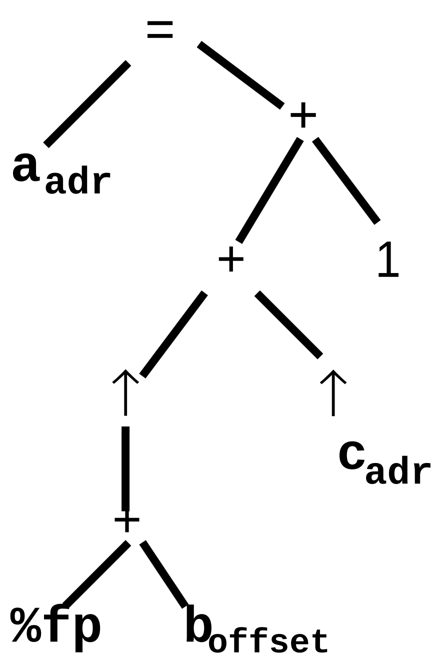
Can we guarantee the *best* instruction sequence?

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 = φ(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.