CS 701
Charles N. Fischer
Fall 2003

http://www.cs.wisc.edu/~fischer/cs703.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 #1 (SPARC Architecture Information) and Handout #2 (Chapter 15, Code Optimization) from DoIt.

- Review Handout #1 and 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:  
   
   ```
   ld [a],%r1  
   add %r1,1,%r2  
   mov 3,%r3  
   mov 3,%r3  
   add %r1,1,%r2
   ```

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

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:

```
  ___________
 /              \
|                |
v                v
%fp            %fp+b_offset
    |         |
    |         v
    v                l
    | +   | +   |
    |     |     |
    %adr  %adr  %adr
```

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 %r1,1,%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) x = 1
else x = 2;
print(x)
```

```
if (a) x_1 = 1
else x_2 = 2;
x_3 = \phi(x_1, x_2)
print(x_3)
```
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.

Review of Compiler Optimizations

1. Redundant Expression Elimination (Common Subexpression Elimination)
   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 that 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. 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 \( i-2, \ldots \)). 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 spatially (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 re-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.