Compiler Design and Construction Optimization
Generating Code via Macro Expansion

- Macroexpand each IR tuple or subtree
- A := B + C;
- D := A * C;

lw $t0, B,
lw $t1, C,
add $t2, $t0, $t1
sw $t2, A
lw $t0, A
lw $t1, C
mul $t2, $t0, $t1
sw $t2, D
Generating Code via Macro Expansion

- \( D := (B+C) \times C; \)

\[
\begin{align*}
\text{t1} &= B + C \\
\text{lw} &\quad \text{t0}, B, \\
\text{lw} &\quad \text{tl}, C \\
\text{add} &\quad \text{t2}, \text{t0}, \text{tl} \\
\text{sw} &\quad \text{t2}, \text{tl} \\
\text{t2} &= \text{t1} \times C \\
\text{lw} &\quad \text{t0}, \text{t1} \\
\text{lw} &\quad \text{tl}, C \\
\text{mul} &\quad \text{t2}, \text{t0}, \text{tl} \\
\text{sw} &\quad \text{t2}, \text{t2} \\
\text{d} &= \text{t2} \\
\text{lw} &\quad \text{t0}, \text{t2} \\
\text{sw} &\quad \text{t0}, \text{d}
\end{align*}
\]
Generating Code via Macro Expansion

- Macroexpansion gives poor quality code if each tuple expanded separately
  - Ignoring state (values already loaded)

- What if more than 1 tuple can be replaced with 1 instruction
  - Powerful addressing modes
  - Powerful instructions
  - Loop construct that decrements, tests and jumps if necessary
Register and Temporary Management

- Efficient use of registers
  - Values used often remain in registers
  - Temp values reused if possible

- Define Register classes
  - Allocatable
    - Explicitly allocated and freed
  - Reserved
    - Have a fixed function
  - Volatile
    - Can be used at any time
    - Good for temp values (A:=B)
Temporaries

- Usually in registers (for quick access)
- Storage temporaries
  - Reside in memory
  - Save registers or large data objects
- Pseudoregisters
  - Load into volatile, then save back out
  - Generates poor code
  - Moving temp from register to pseudo register is called spilling
Code Generation

- A separate generator for each tuple
  - Modularized
  - Simpler
  - Harder to generate good code
  - Easy to add to yacc!

- A single generator
  - More complex
Code Generation

- Instruction selection
  - Addressing modes, intermediates
  - R-R, R-M, M-M, RI...
- Address-mode selection
  - Remember all the types!
- Register allocation
- These are tightly coupled
  - Address-mode affects instruction
  - Instruction can affect register
- See handout for a “+” code generator (following slides)
  - Doesn't handle 0 or same oprnd twice
Code Generation for Integer Add

From Fischer Leblanc, Fig 15.1

Generate code for integer add: (+,A,B,C)

A,B operands,  C is destination

Possible operand modes for A and B are:
1. Literal (stored in value field)
2. Indexed (stored in adr field as (Reg,Displacement) pair; indirect=F)
3. Indirect (stored in adr field as (Reg,Displacement) pair, indirect=T)
4. Live register (stored in Reg field)
5. Dead register (stored in Reg field)

Possible operand modes for C are:
1. Indexed (stored in adr field as (Reg,Displacement) pair, indirect=F)
2. Indirect (stored in adr field as (Reg,Displacement) pair, indirect=T)
3. Live register (stored in Reg field)
4. Unassigned register (stored in Reg field, when assigned)
(a) Swap operands (knowing addition is commutative)

if (B.mode == DEAD_REGISTER || A.mode == LITERAL)
Swap A and B;
/* This may save a load or store since addition overwrites the first operand. */
(b) “Target” result directly into C (if possible)

```c
switch (C.mode) {
    case LIVE_REGISTER: Target = C.reg; break;
    case UNASSIGNED_REGISTER:
        if (A.mode == DEAD_REGISTER)
            C.reg = A.reg; /* Compute into A's reg, then assign it to C. */
        else
            Assign a register to C.reg;
        C.mode = LIVE_REGISTER;
        Target = C.reg;
        break;
    case INDIRECT:
    case INDEXED:
        if (A.mode == DEAD_REGISTER)
            Target = A.reg;
        else
            Target = v2; /* vi is the i-th volatile register. */
        break;
}
```
(c) Map operand B to right operand of add instruction (the "Source")

if (B.mode == INDIRECT) {
    /* Use indexing to simulate indirection. */
    generate(LOAD,B.adr,v1,"""""); // v1 is a volatile register.
    B.mode = INDEXED;
    B.adr = (address)
    B.reg = v1;
    B.displacement = 0;
}
Source = B;
(d) Now generate the add instruction

switch (A.mode) {
    /* Load operand A (if necessary). */
    case LITERAL:
        if (B.mode == LITERAL) // Do we need to fold?
            generate(LOAD, #(A.val+B.val), Target, "") ; break;
        else
            generate(LOAD, #A, val.Target);
    case INDEXED: generate(LOAD, A.adr, Target, "") ; break;
    case LIVE_REGISTER: generate(LOAD, A.reg, Target, "") ; break;
    case INDIRECT:
        generate(LOAD, A.adr, v2, "");
        t.reg = v2; t.displacement = 0;
        generate(LOAD, t, Target, "") ; break;
    case DEAD_REGISTER:
        if (Target != A.reg)
            generate(LOAD, A.reg, Target, "") ; break;
}
generate(ADD, Source, Target, "");
if (C.mode == INDEXED)
    generate(STORE,C.adr,Target,"");
else if (C.mode == INDIRECT) {
    generate(LOAD,C.adr,v3,""');
    t.reg = v3; t.displacement = 0;
    generate(STORE,t,Target,"");
}
Improving Code

- **Removing extra loads and stores**
  \[
  r_2 := r_1 + 5 \\
  i := r_2 \\
  r_3 := i \\
  r_3 := r_3 \times 3
  \]

- **Copy propagation**
  \[
  r_2 := r_1 \\
  r_3 := r_1 + r_2 \\
  r_2 := 5
  \]

- **What about?**
  \[
  \text{if (?) then } A := B + C; \\
  D := B \times C; \quad \text{// Can we use previous loads?}
  \]
Improving Code

- **Constant folding**
  \[ r_2 := 4 \times 3 \quad \Rightarrow \quad r_2 := 12 \]

- **Constant propagation**
  \[ r_2 := 4 \quad \Rightarrow \quad r_2 := 4 \]
  \[ r_3 := r_1 + r_2 \quad \Rightarrow \quad r_3 := r_1 + 4 \]
  \[ r_2 := \ldots \quad \Rightarrow \quad r_2 := \ldots \]
  \[ r_3 := r_1 + 4 \quad \Rightarrow \quad r_2 := \ldots \]
  \[ r_3 := r_1 + 4 \quad \Rightarrow \quad r_2 := \ldots \]
Redundant computations

- Common subexpression (CSE)
  \[
  A := b+c; \\
  D := 3 \times (b+c);
  \]

- \(b+c\) already calculated, so don't do again

- But what about?
  \[
  A := b+c; \\
  b := 3; \\
  D := 3 \times (b+c);
  \]

- Need to know if the CSE is alive or dead
  - This also applies with copy propagation

- Array indexing often causes CSEs
Redundant computations

- Common subexpression (CSE)
  
  \[ A := b+c; \]
  \[ D := b+f+c; \]

- \( b+c \) already calculated, don't do again
  
  \[ A := b+c; \]
  \[ D := A+f; \]

- Problem is to identify the CSEs
  
  - Store \( A+B+C, A+C+B, B+C+A \ldots \) all in the same form
    
    - \( E = A + C; \)
    - \( D = A + B + C \)
Redundant computations

- To take advantage of CSEs, keep track of what values are already in the temp registers and when they “die”

- This can be complex
  - Can use a simple stack approach
  - More complex allocation scheme
    - allocation/deallocation with spilling
    - Allocation with auto deallocate based on usage pattern.

- What about

  $$A(i) := b+c;$$
  $$D := A(j);$$

  - A(j) is already stored if (i == j)
    - This is aliasing and can cause problems
    - If A(j) gets set, A(i) should be killed
Peephole Optimization

- As in the “+” example in the handout, we could have special cases in all of the semantic routines

- Or we could worry about it later and look at the generated code for special cases

- Pattern-replacement pairs can be used
  - A pattern replace pair is of the form
    - Pattern $\Rightarrow$ replacement
    - If Pattern is seen, it is replaced with replacement
Peephole Optimization: Useful Replacement Rules

- **Constant folding** – don’t calc constants
  - \((+,\text{Lit}_1,\text{Lit}_2,\text{Result}) \Rightarrow (=:,\text{Lit}_1 + \text{Lit}_2,\text{Result})\)
  - \((=:,\text{Lit}_1,\text{Result}_1),(+,\text{Lit}_2,\text{Result},\text{Result}_2) \Rightarrow (=:,\text{Lit}_1,\text{Result}_1),(:,=,\text{Lit}_1 + \text{Lit}_2,\text{Result}_2)\)

- **Strength reduction** - slow op to fast op
  - \((*,\text{Oprnd},2,\text{Res}) \Rightarrow (\text{ShiftLeft},\text{Oprnd},1,\text{Res})\)
  - \((*,\text{Oprnd},4,\text{Res}) \Rightarrow (\text{ShiftLeft},\text{Oprnd},2,\text{Res})\)

- **Null sequences** - delete useless calcs
  - \((+,\text{Oprnd},0,\text{Res}) \Rightarrow (=:,\text{Oprnd},\text{Res})\)
  - \((*,\text{Oprnd},1,\text{Res}) \Rightarrow (=:,\text{Oprnd},\text{Res})\)
Peephole Optimization: Useful Replacement Rules

- Combine Operations – many with 1
  - Load A,Ri; Load A+1,Ri+1 ⇒ DoubleLoad A,Ri
  - BranchZero L1,R1; Branch L1; L1: ⇒ BranchNotZero L2, R1
  - Subtract #1,R1; BranchZero L1,R1 ⇒ SubtractOneBranch L1,R1

- Algebraic Laws
  - (+,Lit,Oprnd,Res) ⇒ (+,Oprnd,Lit,Res)
  - (-,0,Oprnd,Res) ⇒ (negate,Oprnd,Res)
Peephole Optimization: Useful Replacement Rules

- **Combine Operations** – many with 1
  - Subtract #1, R1 ⇒ decrement R1
  - Add #1, R1 ⇒ increment R1
  - Load #0, R1; Store A, R1 ⇒ Clear A

- **Address Mode operations**
  - Load A, R1; Add 0(R1), R2 ⇒ Add @A, R2
    - @A denotes indirect addressing
  - Subtract #2, R1; Clear 0(R1) ⇒ Clear -(R1)
    - -(Ri) denotes auto decrement

- **Others**
  - (:=,A,A) ⇒
  - (:=,oprand1,A)(:=,oprnd2,A) ⇒ (:=,Oprnd2,A)
Global Optimizations vs Local Optimizations

- Consider
  if A = B then
    C := 1;
    D := 2;
  else
    E := 3;
  endif;
  A := 1;

- Data flow graph

- Local optimization
  - On a branch

- Global optimization
  - Between branches
Global Optimizations vs Local Optimizations

- Consider
  \[ A := B + C; \]
  \[ D := B + C; \]
  
  if \( A > 0 \) then
    \[ E := B + C; \]
  endif;

- 1st CSE detected with a local optimization
- The second requires a global one
Loop optimizations

- Invariant expressions within a loop
  ```
  while J > I loop
    C := 8 * I;
    A(J) := C;
    J := J - 2;
  end loop;
  ```

- Should c := 8 * I happen each iteration?
- Can we move it out of the loop?
  ```
  C := 8 * I;
  while J > I loop
    A(J) := C;
    J := J - 2;
  end loop;
  ```
Loop optimizations

- Invariant expressions within a loop
- Can we move it out of the loop?

C := 3; J = 1; I = 10;    /* Values before loop */

while J > I loop
  C := 8*I;
  A(J) := C;
  J := J - 2;
end loop;
R := C;

- What value is R with our “optimization”?  
- What value is R without it?
Loop optimizations

- Invariant expressions within a loop

  ```
  while J > I loop
      A(J) := 10/I;
      J := J - 2;
  end loop;
  ```

- Should 10/I happen each iteration?
- Can we move it out of the loop?
  - We are only moving a subexpression that isn't used elsewhere.
Loop optimizations

- Invariant expressions within a loop
  
  I := 0;
  J := -3;
  while J > I loop
    A(J) := 10/I;
    J := J - 2;
  end loop;

- Should 10/I happen each iteration?
- Can we move it out of the loop?
  - We are only moving a subexpression that isn't used elsewhere.
- What happens?
Loop optimizations

- Invariant expressions within a loop

```plaintext
loop I = 1 to 1,000,000
  while j > I loop
    SomeExpensiveCalculationThatdoesn'tusej
    J := J - 2;
  end loop;
end loop
```

- Should we move the expensive calculation?
Loop optimizations

- Invariant expressions within a loop
  
  ```
  j := 3;
  loop l = 1 to 1,000,000
      while j > l loop
         SomeExpensiveCalculationThatdoesn'tusej
         J := J - 2;
      end loop;
  end loop
  ```

- What happens if we move it?
Loop optimizations

- Invariant expressions within a loop

```
for (i = 0, i < N, i++)
    b = e * f;
    c(i) = b * i;
end if
```

- What is happening?

```
C(0) = b*0 = 0
C(1) = b*1 = 0+b
C(2) = b*2 = 0+b+b
C(3) = b*3 = 0+b+b+b ...
```

- Can change to an add, strength reduction
  - What is consequence of changing to an add if b were int or real?
The Truth About Global Optimization

- Global optimization is complex, expensive and sometimes unsafe
- Effect of calls must be considered when performing other optimizations
- Most programs don't need optimized
- Optimization can save 25-50% (speed & size)
  - A better algorithm is much more effective!
- Loops & function calls best place to apply
Loop Optimization

```plaintext
for i = 1..100 loop
    for j = 1..100 loop
        for k = 1..100 loop
            A(i,j,k) := i*j*k;
        end loop;
    end loop;
end loop;

3,000,000 subscripting
2,000,000 multiplies
```
Loop Optimization: Factor inner loop

for i = 1..100 loop
  for j = 1..100 loop
    for k = 1..100 loop
      A(i,j,k) := i*j*k;
    end loop;
  end loop;
end loop;

- 3,000,000 subscripting
- 2,000,000 multiplies

for i = 1..100 loop
  for j = 1..100 loop
    temp1 := adr(A(i)(j));
    temp2 := i*j;
    for k = 1..100 loop
      temp1(k) := temp2*k;
    end loop;
  end loop;
end loop;

- factor inner loop
  - 1,020,000 subscripts
  - 1,010,000 multiplies
Loop Optimization

for i = 1..100 loop
    for j = 1..100 loop
        temp1 := adr(A(i)(j));
        temp2 := i*j;
        for k = 1..100 loop
            temp1(k) := temp2*k;
        end loop;
    end loop;
end loop;

- factor inner loop
  - 1,020,000 subscripts
  - 1,010,000 mults

for i = 1..100 loop
    temp3 := adr(A(i))
    for j = 1..100 loop
        temp1 := adr(temp3(j));
        temp2 := i*j;
        for k = 1..100 loop
            temp1(k) := temp2*k;
        end loop;
    end loop;
end loop;

- factor 2nd loop
  - Add?
  - Subscripts? 1,010,100
  - Mults? 1,010,000
Loop Optimization: Strength Reduction

for i = 1..100 loop
    temp3 := adr(A(i))

for j = 1..100 loop
    temp1 := adr(temp3(j));
    temp2 := i*j;

for k = 1..100 loop
    temp1(k) := temp2*k;

end loop;

end loop;
Loop Optimization: Copy Propagation

for i = 1..100 loop
    temp3 := adr(A(i))
    temp4 := l; -- init i*j
    for j = 1..100 loop
        temp1 := adr(temp3(j));
        temp2 := temp4; -- t4 == i*j
        temp5 := temp2; -- init temp2*k
        for k = 1..100 loop
            temp1(k) := temp5;
            temp5 := temp5 + temp2;
        end loop;
        temp4 := temp4+i
    end loop;
end loop;

for i = 1..100 loop
    temp3 := adr(A(i))
    temp4 := l; -- initial value of i*j
    for j = 1..100 loop
        temp1 := adr(temp3(j));
        temp2 := temp4; -- t4 == i*j
        temp5 := temp4;
        for k = 1..100 loop
            temp1(k) := temp5 -- holds I*j*K;
            temp5 := temp5 + temp4;
        end loop;
        temp4 := temp4+i
    end loop;
end loop;

Add?  Subscripts?  Mults?
Loop Optimization: Expand Subscripting Code

for i = 1..100 loop
    temp3 := adr(A(i))
    temp4 := 1;  -- initial value of i*j
end loop;

for j = 1..100 loop
    temp1 := adr(temp3(j));
    -- temp2 := temp4;  -- t4 == i*j
    temp5 := temp4;
    for k = 1..100 loop
        temp1(k) := temp5;  -- holds l*j*K
        temp5 := temp5 + temp4;
    end loop;
    temp4 := temp4 + i
end loop;
end loop;

for i = 1..100 loop
    temp3 := A0+(10000*I)-10000;
    temp4 := 1;  -- initial value of i*j
end loop;

for j = 1..100 loop
    temp1 := temp3+(100*J)-100;
    -- temp4 holds i*j
    temp5 := temp4;
    for k = 1..100 loop
        (temp1+k-1) := temp5;  -- holds i*j*K
        temp5 := temp5 + temp4;
    end loop;
    temp4 := temp4 + i
end loop;
end loop;

Add? Subscripts? Mults?
Strength Reduction on Subscripting code

for i = 1..100 loop
    temp3 := A0+(10000*i)-10000;
    temp4 := i; -- initial value of i*j

    for j = 1..100 loop
        temp1 := temp3+(100*j)-100;
        -- temp4 holds i*j
        temp5 := temp4;
        for k = 1..100 loop
            (temp1+k-1) := temp5; -- holds i*j*k
            temp5 := temp5 + temp4;
        end loop;
        temp4 := temp4+i
    end loop;
end loop;

temp6:=A0;

for i = 1..100 loop
    temp3 := temp6;
    temp4 := i; -- initial value of i*j
    temp7 := temp3; -- initial value of Adr(A(i)(j))

    for j = 1..100 loop
        temp1 := temp7;
        temp5 := temp4;
        -- initial value of temp4*k
        temp8 := temp1; -- initial value of Adr(A(i)(j)(k))
        for k = 1..100 loop
            temp8 := temp5 -- temp5 holds i*j*k
            temp5 := temp5 + temp4;
            temp8 := temp8 + 1;
        end loop;
        temp4 := temp4+i
    end loop;
end loop;

temp6 := temp6+1000;

Add?

Subscripts?

Mults?
temp6:=A0;
for i = 1..100 loop
    temp3 := temp6;
    temp4 := i; -- initial value of i*j
    temp7 := temp3; -- initial value of Adr(A(i)(j))
    for j = 1..100 loop
        temp1 := temp7;
        temp5 := temp4; -- initial value of temp4*k
        temp8 := temp1 -- initial value of Adr(A(i)(j)(k))
        for k = 1..100 loop
            temp8 := temp5 -- temp5 holds i*j*k
            temp5 := temp5 + temp4;
            temp8 := temp8 + 1;
        end loop;
        temp4 := temp4+i
        temp7 := temp7+100;
    end loop;
    temp6 := temp6+1000;
end loop;

Add?
Subscripts?
Mults?
Loop Optimization

for i = 1..100 loop
    for j = 1..100 loop
        for k = 1..100 loop
            A(i,j,k) := i*j*k;
        end loop;
    end loop;
end loop;

- 3,000,000 subscripting
- 2,000,000 multiplies

Adds 0
Subscripts 3,000,000
Mults 2,000,000
Assigns 1,000,000

-- temp3 := temp6;
temp4 := 1; -- initial value of i*j
temp7 := temp6; -- initial value of Adr(A(i)(j))
for j = 1..100 loop
    -- temp1 := temp7;
temp5 := temp4; -- initial value of temp4*k
    temp8 := temp7 -- initial value of Adr(A(i)(j)(k)
    for k = 1..100 loop
        temp8 := temp5 -- temp5 holds i*j*k
        temp5 := temp5 + temp4;
        temp8 := temp8 + 1;
    end loop;
    temp4 := temp4 + i
    temp7 := temp7 + 100;
end loop;

Add 2,020,200
Subscripts 3,000,000
Mults 0
Assigns 3,040,301
Data Flow Analysis

- Determine which variables are live going in and out of a block
- These tools allow deeper analysis

A := D
if A = B then
  B := 1
else C := 1
end if;
A := A + B

if D = B then
  B := 1
else C := 1
end if;
A := D+B;

A only used twice between assignments, using copy propagation reduces to zero so we can remove the assignment
Data Flow Analysis

- Removing dead code
  - If (debug) printf(“....
  - This will never get executed if you have
    - Debug := false; prior to the if
Optimizations for Machine Code

- Filling load and branch delays
  - May be competing with HW scheduler
- For CISC/VLIW replace multiple instructions with more complex instructions
- Loop unrolling
- Taking advantage of memory accesses
- Understanding pipelines
- Multiple cores