Assignment statement optimizations

Constant folding

Scalar replacement of aggregates

Algebraic simplification and Reassociation

Common subexpression elimination

Copy propagation
Constant folding

- Constant-expressions evaluation or constant folding, refers to the evaluation at compile time of expressions whose operands are known to be constant.

- Interprocedural constant propagation is particularly important when procedures or macros are passed constant parameters.

- Although it is apparently a relatively simple transformation, compilers do not do a perfect job at recognizing all constant expressions as can be seen in the next three examples from the Sparc Fortran compiler.

- In fact, constant propagation is undecidable.
```c
#include <stdio.h>
int pp( )
{
    int ia =1;
    int ib =2;
    int result;

    result = ia +ib;
    return result;
}
```

**pp.c**

```assembly
.globl pp
pp:
    .type   pp,2
    .size   pp,(.-pp)
```

**cc -O3 -S pp.c**

- .global pp
- pp:
  - /* 000000 */ retl ! Result = %o0
  - /* 0x0004 */ or %g0,3,%o0
  - /* 0x0008 */ 0 */ .type pp,2
  - /* 0x0008 */ */ .size pp,(-pp)
```
```c
#include <stdio.h>
int pp()
{
    int ia =1;
    int ib =2;
    int result;

    result = ia +ib;
    return result;
}
```

```c
cc -O3 -S pp.c
```

```assembly
.globl pp
pp:
    .type   pp,2
    .size   pp,(.-pp)
```
```c
int pp(int id){
    int ic, ia, ib;
    if (id == 1) {
        ia =1;
        ib =2; }
    else {
        ia =2;
        ib =1; }
    ic = ia + ib;
    return ic;
}
```

```assembly
cc -O3 -S pp1.c
! 3           !int pp(int id){
! 4           !  int ic, ia, ib;
! 5           !  if (id == 1) {
    /* 000000 */ 0 /* cmp %o0,1
    /* 0x0004 */ 0 /* bne .L77000003
    /* 0x0008 */ 0 /* or %g0,1,%g1
      .L77000002:
    ! 6           !    ia =1;
    ! 7           !    ib =2; }
    /* 0x00c */ 7 /* or %g0,2,%g2
    /* 0x010 */ 0 /* retl ! Result = %o0
    /* 0x014 */ 0 /* add %g1,%g2,%o0
    .L77000003:
    ! 8           ! else {
    ! 9           !    ia =2;
    /* 0x018 */ 9 /* or %g0,2,%g1
    ! 10          !    ib =1;}
    /* 0x01c */ 10 /* or %g0,1,%g2
    /* 0x020 */ 0 /* retl ! Result = %o0
    /* 0x024 */ 0 /* add %g1,%g2,%o0
    /* 0x028 */ 0 /* .type pp,2
    /* 0x028 */ 0 /* .size pp,(.-pp)
```
```c
int pp() {
    int ic, ia, ib;
    int id = 1;
    if (id == 1) {
        ia = 1;
        ib = 2;
    } else {
        ia = 2;
        ib = 1;
    }
    ic = ia + ib;
    return ic;
}
```
Scalar replacement of aggregates.

- Replaces aggregates such as structures and arrays with scalars to facilitate other optimizations such as register allocation, constant and copy propagation.
DO I = 1, N
    DO J = 1, M
        A(I) = A(I) + B(J)
    ENDDO
ENDDO

DO I = 1, N
    T = A(I)
    DO J = 1, M
        T = T + B(J)
    ENDDO
    A(I) = T
ENDDO

- A(I) can be left in a register throughout the inner loop
- Register allocation fails to recognize this
- All loads and stores to A in the inner loop have been saved
- High chance of T being allocated a register by the coloring algorithm
Algebraic simplification and Reassociation

- Algebraic simplification uses algebraic properties of operators or particular operand combinations to simplify expressions.
- Reassociation refers to using associativity, commutativity, and distributivity to divide an expressions into parts that are constant, loop invariant and variable.
For integers:

Expression simplification such as
\[
\begin{align*}
    i+0 & \rightarrow i \\
    i \, ^2 & \rightarrow i \, * \, i \\
    i*5 & \rightarrow t := i \, \text{shl} \, 3; \, t=t-I
\end{align*}
\]

Associativity and distributivity can be applied to improve parallelism (reduce the height of expression trees).

Algebraic simplifications for floating point operations are seldom applied.

The reason is that floating point numbers do not have the same algebraic properties as real numbers.
For example, the in the code

```cpp
eps := 1.0
while eps + 1.0 > 1.0
    oldeps := eps
    eps := 0.5 * eps
```

Replacing `eps + 1.0 > 1.0` with `eps > 0.0` would change the result significantly. The original form computes the smallest number such that `1 + x = x` while the optimized form computes the maximal `x` such that `x/2` rounds to 0.
The goal is to reduce height of expression tree to reduce execution time.

In a parallel environment:

```
+  
|  
|   a
+  
|  
|   +
|   |
|   b
+  
|  
|   +
|   |
|   |
|   c
d \  
|   
+  
|  
|   +
e
```

Transformed to:

```
+  
|  
|   a
+  
|  
|   +
|   |
|   |
|   b
+  
|  
|   +
|   |
|   |
|   c
d \  
|   
+  
|  
|   +
e
```
Common subexpression elimination

- Transform the program so that the value of a (usually scalar) expression is saved to avoid having to compute the same expression later in the program.

For example:

\begin{align*}
x &= e^3 + 1 \\
&<\text{statement sequence}> \\
y &= e^3 \\
\end{align*}

- is replaced (assuming that $e$ is not reassigned in $<\text{statement sequence}>$) with

\begin{align*}
t &= e^3 \\
x &= t + 1 \\
&<\text{statement sequence}> \\
y &= t \\
\end{align*}

- There are local (to the basic block), global, and interprocedural versions of cse.
Copy propagation

Eliminates unnecessary copy operations.

For example:

```
x = y
<statement sequence>
t = x + 1
```

Is replaced (assuming that neither \(x\) nor \(y\) are reassigned in \(<\text{statement sequence}>\) with

```
<statement sequence>
t = y + 1
```

Copy propagation is useful after common subexpression elimination. For example.

```
x = a+b
<statement sequence>
y = a+b
```
Is replaced by CSE into the following code
\[
\begin{align*}
\text{t} &= \text{a} + \text{b} \\
\text{x} &= \text{t} \\
&\langle \text{statement sequence} \rangle \\
\text{z} &= \text{x} \\
\text{y} &= \text{a} + \text{b}
\end{align*}
\]

Here \( \text{x} = \text{t} \) can be eliminated by copy propagation.
Loop body optimizations

Loop invariant code motion

Induction variable detection

Strength reduction
Recognizes computations in loops that produce the same value on every iteration of the loop and moves them out of the loop.

An important application is in the computation of subscript expressions:

```fortran
do i=1,n
   do j=1,n
      ...a(j,i)....
```

Here \( a(j,i) \) must be transformed into something like \( a((i-1) \times M+j-1) \) where \( (i-1) \times M \) is a loop invariant expression that could be computed outside the \( j \) loop.
ppl()
{
    float a[100][100];
    int i, j;
    for (i=1; i++ ; i <= 50)
        for (j=1; j++ ; j <= 50)
            a[i][j] = 0;
}

optimized
cc -O3 -S pp1.c

a[i][j] = 0;
...

.L900000109:
    or %g0, %o0, %g2
    add %o3, 4, %o3
    add %o0, 1, %o0
    cmp %g2, 0
    bne, a.L900000109
    st %f0, [%o3] ! volatile

.L95:
    8     a[i][j]=0;
    sethi hi(.L_cseg0), %o0
    ld [%o0+%lo(.L_cseg0)], %f2
    sethi39, %o0
    xor %o0, -68, %o0
    add %fp, %o0, %o3
    sethi39, %o0
    xor %o0, -72, %o0
    ld [%fp+%o0], %o2
    sll %o2, 4, %o1
    sll %o2, 7, %o0
    add %o1, %o0, %o1
    sll %o2, 8, %o0
    add %o1, %o0, %o0
    add %o3, %o0, %o1
    sethi39, %o0
    xor %o0, -76, %o0
    ld [%fp+%o0], %o0
    sll %o0, 2, %o0
    st %f2, [%o1+%o0]
    sethi39, %o0
    xor %o0, -76, %o0
    ld [%fp+%o0], %o0
    mov %o0, %o2
    sethi39, %o0
    xor %o0, -76, %o0
    ld [%fp+%o0], %o0
    add %o0, 1, %o1
    sethi39, %o0
    xor %o0, -76, %o0
    cmp %o2, %g0
    bne.L95
    st %o1, [%fp+%o0]
Induction variable detection

- Induction variables are variables whose successive values form an arithmetic progression over some part of the program.
- Their identification can be used for several purposes:
  - Strength reduction (see below).
  - Elimination of redundant counters.

```
do  i=1,n
  j=j+2
  a(j)=
end
```

```
do  i=1,n
  a(j+2*i)=
end
```
• Elimination of interactions between iterations to enable parallelization.

-- The following loop cannot be transform as is into parallel form

    do i=1,n
        k=k+3
        a(k) = b(k)+1
    end do

-- The reason is that induction variable \( k \) is both read and written on all iterations. However, the collision can be easily removed as follows

    do i=1,n
        a(3*i) = b(3*i) +1
    end do

-- Notice that removing induction variables usually has the opposite effect of strength reduction.
Strength reduction


In real compiler probably only multiplication to addition is the only optimization performed.

Candidates for strength reduction include:
1. Multiplication by a constant
   ```
   loop
   n=i*a
   ...
   i=i+b
   ```
after strength reduction

```
loop
  n=t1
  ...
  i=i+b
  t1=t1+a*b
```

after loop invariant removal

```
c = a * b
t1 = i*a
loop
  n=t1
  ...
  i=i+b
  t1=t1+c
```
2. Two induction variables multiplied by a constant and added

\[
\text{loop} \\
\quad n = i \times a + j \times b \\
\quad \ldots \\
\quad i = i + c \\
\quad \ldots \\
\quad j = j + d
\]

after strength reduction

\[
\text{loop} \\
\quad n = t1 \\
\quad \ldots \\
\quad i = i + c \\
\quad t1 = t1 + a \times c \\
\quad j = j + d \\
\quad t1 = t1 + b \times d
\]
3. Trigonometric functions

\[
\text{loop} \\
y = \sin(x) \\
\ldots \\
x = x + \Delta x
\]

After strength reduction

\[
\text{loop} \\
\ldots \\
x = x + \Delta x \\
t\sin x = t\sin x \cdot t\cos \Delta x + t\cos x \cdot t\sin \Delta x \\
t\cos x = t\sin x \cdot t\sin \Delta x + t\cos x \cdot t\cos \Delta x
\]
By propagating assertions it is possible to avoid unnecessary bound checks

For example, bound checks are not needed in:

```plaintext
real a(1000)
   do i=1,100
      ... a(i)...
   end do
```

And they are not needed either in

```plaintext
if i > 1 and i < 1000 then
   ... a(i)...
end if
```

A related transformation is predicting the maximum value subscripts will take in a region to do pre-allocation for languages (like MATLAB) where arrays grow dynamically.
Procedure optimizations

Tail recursion elimination

Procedure integration

Leaf routine optimization

Tail recursion elimination
Tail recursion elimination

- Converts tail recursive procedures into iterative form

```c
void make_node(p, n)
    struct node *p;
    int n;
    { struct node *q;
        q = malloc(sizeof(struct node));
        q->next = nil;
        q->value = n;
        p->next = q;
    }

void insert_node(n, l)
    int n;
    struct node *l;
    { if (n > l->value)
        if (l->next == nil) make_node(l, n);
        else insert_node(n, l->next);
    }

void insert_node(n, l)
    int n;
    struct node *l;
    { loop:
        if (n > l->value)
            if (l->next == nil) make_node(l, n);
        else
            { l := l->next;
            goto loop;
            }
    }
```
Procedure integration

- Expands inline the procedure.
- This gives the compiler more flexibility in the type of optimizations it can apply.
- Can simplify the body of the routine using parameter constant values.
- Procedure cloning can be used for this last purpose also.
- If done blindly, it can lead to long source files and incredibly long compilation times.
subroutine sgefa(a,lda,n,ipvt,info)
  integer lda,n,ipvt(i),info
  real a(lda,1)
  real t
  integer isamax,j,kp1,l,nml

    do 30 j = kp1, n
      t = a(1,j)
      if (t .eq. k) go to 20
      a(l,j) = a(k,j)
      a(k,j) = t
    20    continue
    call saxpy(n-k,t,a(k+1,k),1,a(k+1,j),1)
  30    continue

subroutine saxpy(n,da,dx,incy,dy,incy)
  real dx(1),dy(1),da
  integer i,incx,incy,ix,iy,ix,incy,ix,iy
  if (n .le. 0) return
  if (da .eq. ZER0) return
  if (incx .eq. 1 .and. incy .eq. 1) go to 20
  ix = 1
  iy = 1
  if (incx .lt. 0) ix = (-n+1)*incx + 1
  if (incy .lt. 0) iy = (-n+1)*incy + 1
  do 10 i = 1,n
    dy(iy) = dy(iy) + da*dx(ix)
    ix = ix + incx
    iy = iy + incy
  10    continue
    return
  20    continue
  do 30 i = 1,n
    dy(i) = dy(i) + da*dx(i)
  30    continue
    return
  end
Leaf routine optimization

- Leaf routines tend to be the majority (leafs of binary trees are one more than the number of interior nodes).

- Save instructions that prepare for further calls (set up the stack/display registers, save/restore registers)
Register allocation

- Objective is to assign registers to scalar operands in order to minimize the number of memory transfers.

- An NP-complete problem for general programs. So need heuristics. Graph coloring-based algorithm has become the standard.

- Register allocation algorithms will be discussed in more detail later in the semester.
Instruction scheduling

- Objective is to minimize execution time by reordering executions.
- Scheduling is an NP-complete problem.

<table>
<thead>
<tr>
<th></th>
<th>Issue latency</th>
<th>Result latency</th>
</tr>
</thead>
<tbody>
<tr>
<td>L: ldf [r1], f0</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>fadd s0, f1, f2</td>
<td>1</td>
<td>7</td>
</tr>
<tr>
<td>stf f2, [r1]</td>
<td>6</td>
<td>3</td>
</tr>
<tr>
<td>sub r1, 4, r1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>cmp r1, c</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>bg L</td>
<td>1</td>
<td>2</td>
</tr>
<tr>
<td>nop</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>

**FIG. 17.15** A simple SPARC loop with assumed issue and result latencies.

**FIG. 17.16** Pipeline diagram for the loop body in Figure 17.15.
Control-flow optimizations

Unreachable code elimination

• Unreachable code is code that cannot be executed regardless of the input.

• Eliminating it save space

Straightening

• It applies to pairs of basic blocks so that the first has no successors other than the second and the second has no predecessors other than the first.
If simplification

- Applies to conditional constructs one of both of whose arms are empty

Loop inversion

- Transforms a while loop into a repeat loop.
- Fewer loop bookkeeping operations are needed

Unswitching

- Moves loop-invariant conditional branches out of loops

Dead code elimination

- Eliminates code that do not affect the outcome of the program.
**Stripmining**

Transforms a loop into a doubly nested loop.

```plaintext
for (i=1; i<=100, i++)
    {...}

for (K=1; K<=100; k+=20)
    for (i=K; i<=K+19; i++)
        {...}
```

**Loop fusion**

Joins two loops into a single one.
**Loop fission**

Breaks a loop into two loops

\[
\text{for } (i=1; i<=n; i++) \{ \alpha; \beta; \chi; \delta; \varepsilon \}
\]

\[
\text{for } (i=1; i<=n; i++) \{ \alpha; \beta; \chi \}
\]

\[
\text{for } (i=1; i<=n; i++) \{ \delta; \varepsilon \}
\]

**Loop interchange**

Changes the order of loop headers

\[
\text{for } (i=1; i<=n; i++)
\]

\[
\quad \text{for } (j=1; j<=m; j++) \{ ... \}
\]

\[
\quad \text{for } (j=1; j<=m; j++)
\]

\[
\text{for } (i=1; i<=n; i++) \{ ... \}
\]
Loop tiling

This is a combination of strip mining followed by interchange that changes traversal order of a multiply nested loop so that the iteration space is traversed on a tile-by-tile basis.

```c
for (i=0; i<N; i++)
    for (j=0; j<N; j++)
        c[i] = a[i,j]*b[i];

for (i=0; i<N; i+=2)
    for (j=0; j<N; j+=2)
        for (ii=i; ii<min(i+2,N); ii++)
            for (jj=j; jj<min(j+2,N); jj++)
                c[ii] = a[ii,jj]*b[ii];
```
Iteration space and loop transformations
DEPENDENCES

The correctness of many loop transformations can be decided using dependences.

Still a good introduction to the notion of dependence and its applications can be found in D. Kuck, R. Kuhn, D. Padua, B. Leasure, M. Wolfe: Dependence Graphs and Compiler Optimizations. POPL 1981.

Flow Dependence (True Dependence)

S1  X=A+B
S2  C=X+1

Anti Dependence

S1  A=X+B
S2  X=C+D

Output Dependence

S1  X=A+B
    . . .
S2  X=C+D

The notion of dependence applies to sequential programs.
Transformations are sequential to sequential or sequential to parallel.
Dependences and parallelization

\[ S_1; S_2; S_3 \text{ can execute in parallel with } S_4; S_5; S_6 \]

\[ S_8; S_9 \quad " \quad " \quad " \quad " \quad " \quad S_{10}; S_{11} \]
Renaming

\[ S_1 \quad A = X + B \]
\[ S_2 \quad X = Y + 1 \]
\[ S_3 \quad C = X + B \]
\[ S_4 \quad X = Z + B \]
\[ S_5 \quad D = X + 1 \]

\[ S_1 \quad A' = X' + B \]
\[ S_2 \quad X' = Y' + 1 \]
\[ S_3 \quad C' = X' + B \]
\[ S_4 \quad X' = Z' + B \]
\[ S_5 \quad D' = X' + 1 \]
Scalar expansion

\[
\begin{align*}
\text{DO} & \quad I=1,N \\
S1: & \quad A=B(I)+1 \\
S2: & \quad C(I)=A+D(I) \\
\text{END DO}
\end{align*}
\]

\[
\begin{align*}
\text{DO} & \quad I=1,N \\
S1: & \quad A1(I)=B(I)+1 \\
S2: & \quad C(I)=A1(I)+D(I) \\
\text{END DO} \\
A & =A1(N)
\end{align*}
\]
The dependence information determines whether or not the loop headers can be interchanged.

For example, the following loop headers can be interchanged:

\[
\begin{align*}
d & \text{do } i=1,n \\
& \text{do } j=1,n \\
& \quad a(i,j) = a(i,j-1) + a(i-1,j) \\
& \text{end do} \\
& \text{end do}
\end{align*}
\]
• However, the headers in the following loop cannot be interchanged

```fortran
do i=1,n
    do j=1,n
        a(i,j) = a(i,j-1) + a(i-1,j+1)
    end do
end do
```

Invalid!
Loop unrolling

Replicates the body of the loop and reduces the number of iterations

```
for (int x = 0; x < 100; x++)
{
    delete(x);
}
```

```
for (int x = 0; x < 100; x += 5)
{
    delete(x);
    delete(x+1);
    delete(x+2);
    delete(x+3);
    delete(x+4);
}
```
Dependences in loops must be represented in summary form by “collapsing” the unrolled graph as shown next.

```
    do I=1 to N
    S1    A=B(I)+1
    S2    C(I)=A+2
```
Definition of dependence in loops

\[ \text{do I=1 to N} \]

\[ S1 \text{ X}(I(I)) = B(I)+1 \]

\[ S2 \text{ A}(I) = X(I(I))+2 \]

We say that \( S_1 \iff \exists I_1 \leq I_2 \text{ and } I_1, I_2 \in [1,N] \)

such that \( I(I_1) = I(I_2) \)

We say that \( S_2 \iff \exists I_1 < I_2 \text{ and } I_1, I_2 \in [1,N] \)

such that \( I(I_2) = I(I_1) \)
do I=1 to N
S1 \( A(I) = X(J(I)) + 1 \)
S2 \( X(J(I)) = B(I) + 2 \)

We say that \( S_2 \) iff \( \exists \ I_1 < I_2 \) and \( I_1, I_2 \in [1, N] \)
such that \( J(I_1) = J(I_2) \)
Testing the conditions above could be very expensive in general. However, most subscript expressions are quite simple and can be easily analyzed.

The approach that has traditionally been used is to try to break a dependence, that is to try to prove that the dependence does not exist.

Practical tests are usually conservative. That is, they may not break a dependence that does not exist. Assuming a dependence that exists is conservative but will not lead to incorrect transformations for the cases discussed in this course.
Banerjee’s test

Assume that \( f(I) = A_1 I + A_0 \) and \( g(I) = B_1 I + B_0 \)

Banerjee’s test proceeds by finding an upper bound \( U \) and a lower bound \( L \) of \( A_1 I_1 - B_1 I_2 \) under the constrains that \( 1 \leq I_1 \leq I_2 \leq N \).

If either \( L > A_0 - B_0 \) or \( U < A_0 - B_0 \), then the functions do not intersect, and therefore we know there is no flow dependence.

For example, consider the loop

\[
\text{do I=1 to 5} \\
S1: \quad X(I+5) = B(I) + 1 \\
S2: \quad A(I) = X(I) + 2
\]

The upper limit of \( A_1 I_1 - B_1 I_2 = I_1 - I_2 \) is 4, which is \( < A_0 - B_0 = 5 \) and therefore the dependence would be broken by Banerjee’s test.