



Toronto Portable Optimizer

Second-Order Predictive Commoning

Arie Tal

CASCON 2004

Outline

■ Predictive Commoning

- Reusing computations across loop iterations
 - Detecting indexing sequences
 - Unrolling to avoid register copying

■ Second-Order Predictive Commoning

- Reusing complex computations
- Re-associating expressions
 - “Parallel sequences”
 - Expression equivalence classes
 - The transformation
 - Example of a “real” Second-Order Predictive Commoning transformation.

Reusing Computations Across Iterations

- **Accessing multiple consecutive array elements in a loop**

```
for (i=0; i < n; i++)  
    a[i+2]=a[i]+a[i+1];
```

- **Requires 2 load operations (one of them of a stored value)**
- **Performance can be improved by applying Predictive Commoning**

```
p0=a[0];  
p1=a[1];  
for (i=0; i < n; i++) {  
    a[i+2]=p2=p0+p1;  
    p0=p1;  
    p1=p2;  
}
```

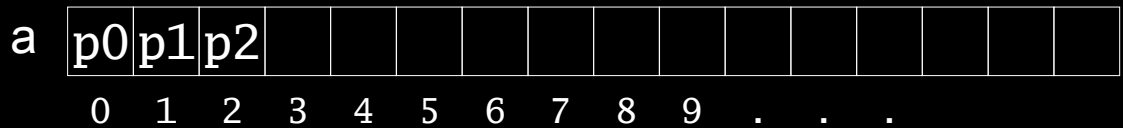
- **Requires zero load operations and two register copies**

Reusing Computations Across Iterations (cont.)

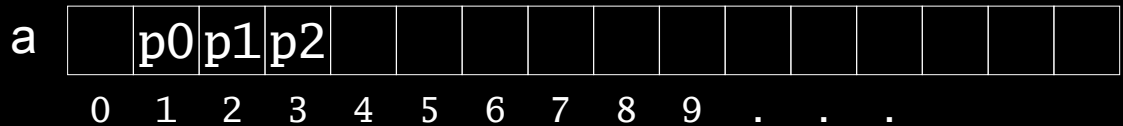
- Transfer values between iterations using registers

```

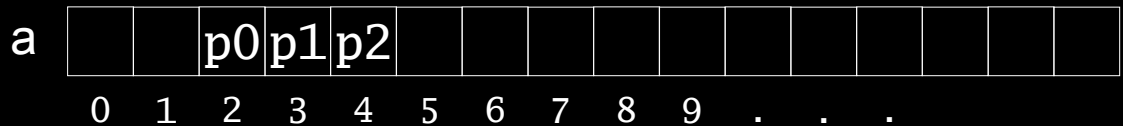
p0=a[0];
p1=a[1];
for (i=0; i < n; i++) {
    a[i+2]=p2=p0+p1;
    p0=p1;
    p1=p2;
}
    
```



$p0 \leftarrow p1 \leftarrow p2$



$p0 \leftarrow p1 \leftarrow p2$

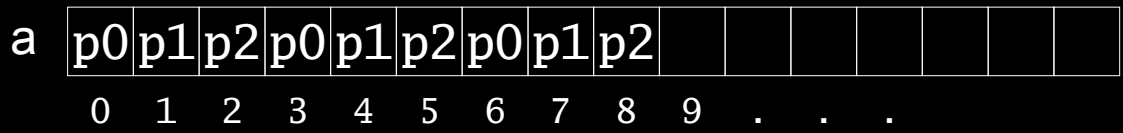


Reusing Computations Across Iterations (cont.)

- **Unrolling and renaming registers can be applied to avoid copying registers**

```
p0=a[0];
p1=a[1];
for (i=0; i < n; i++) {
    a[i+2]=p2=p0+p1;
    p0=p1;
    p1=p2;
}
```

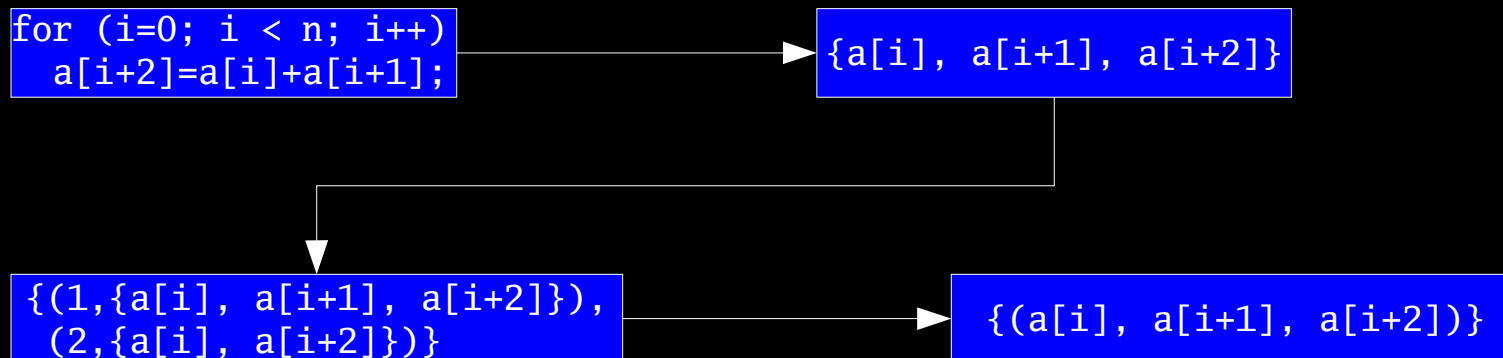
```
p0=a[0];
p1=a[1];
for (i=0; i < n-n%3; i+=3) {
    a[i+2]=p2=p0+p1;
    a[i+3]=p0=p1+p2;
    a[i+4]=p1=p2+p0;
}
for (;i<n;i++)
    a[i+2]=a[i]+a[i+1];
```



Overview of the algorithm

■ Indexing Sequences

- Identify indexed expressions
- Compute strides between expressions
 - Note: We handle sequences with stride greater than 1
- Group expressions into sequences
- Filter out “unwanted” sequences



- Need to be careful with stride distances
 - $a[2*i+1]$ and $a[2*i+2]$ have a stride distance of zero

Overview of the algorithm (cont.)

- **Determine unroll factor**

- Multiple sequences of varying lengths and strides
 - $\text{LCM}(\text{length}_1 * \text{stride}_1, \text{length}_2 * \text{stride}_2, \text{length}_3 * \text{stride}_3, \dots)$

- **Assign temporary variables to sequence elements**

- $a[i] \rightarrow p_0 \quad a[i+1] \rightarrow p_1 \quad a[i+2] \rightarrow p_2$

- **Generate initial variable initializations**

- **Unroll loop**

- Generate “feeder” loads or stores (if needed)
- Replace expressions with temporary variables
- Rotate temporary variable assignments for next i
 - $a[i] \rightarrow p_1 \quad a[i+1] \rightarrow p_2 \quad a[i+2] \rightarrow p_0$
 - $a[i] \rightarrow p_2 \quad a[i+1] \rightarrow p_0 \quad a[i+2] \rightarrow p_1$

```
p0=a[0];
p1=a[1];
for (i=0; i < n-n%3; i+=3) {
    a[i+2]=p2=p0+p1;
    a[i+3]=p0=p1+p2;
    a[i+4]=p1=p2+p0;
}
for (;i<n;i++)
    a[i+2]=a[i]+a[i+1];
```


Reusing Complex Computations

- **Repeating computations between consecutive iterations**

```
for (i=0; i < n; i++)  
    f[i] = a[i]*exp(i+1)+a[i+1]*exp(i+2);
```

- **With Predictive Commoning we can eliminate a function call, a load and a multiply in each iteration**

```
p0=a[0]*exp(1);  
for (i=0; i < n-n%2; i+=2) {  
    p1=a[i+1]*exp(i+2);  
    f[i] = p0+p1;  
    p0=a[i+2]*exp(i+3);  
    f[i+1] = p1+p0;  
}  
for (; i < n; i++)  
    f[i] = a[i]*exp(i+1)+a[i+1]*exp(i+2);
```


Re-association

- **Repeating re-associable computations**

```
for (i=0; i < n; i++)  
  f[i] = a[i]*(b[i]+c[i])+  
         a[i+1]*(b[i+1]+d[i]+c[i+1]);
```

- **With single-operator re-association**

```
p0=a[0];  
r0=b[0]+c[0];  
for (i=0; i < n-n%2; i+=2) {  
  p1=a[i+1]; r1=b[i+1]+c[i+1];  
  f[i] = p0*r0 + p1*(r1+d[i]);  
  p0=a[i+2]; r0=b[i+2]+c[i+2];  
  f[i+1] = p1*r1 + p0*(r0+d[i+1]);  
}  
for (i=0; i < n; i++)  
  f[i] = a[i]*(b[i]+c[i])+  
         a[i+1]*(b[i+1]+d[i]+c[i+1]);
```

Re-association

- **Repeating re-associable computations**

```
for (i=0; i < n; i++)  
  f[i] = a[i]*(b[i]+c[i])+  
        a[i+1]*(b[i+1]+d[i]+c[i+1]);
```

- **With multi-operator re-association**

```
p0=a[0];  
r0=a[0]*(b[0]+c[0]);  
for (i=0; i < n-n%2; i+=2) {  
  p1=a[i+1]; r1=p1*(b[i+1]+c[i+1]);  
  f[i] = r0 + r1 + p1*d[i];  
  p0=a[i+2]; r0=p0*(b[i+2]+c[i+2]);  
  f[i+1] = r1 + r0 + p0*d[i+1];  
}  
for (i=0; i < n; i++)  
  f[i] = a[i]*(b[i]+c[i])+  
        a[i+1]*(b[i+1]+d[i]+c[i+1]);
```

Re-association algorithm - overview

■ Single-operator re-association

- Identify “parallel” sequences
 - Same stride and length
- Maximize expressions
 - Partition parallel sequences into equivalence classes
 - Two sequences are equivalent if all their parallel elements (firsts, seconds, etc.) appear together in all expressions with the same operator (tree)
 - Choose a “representative” sequence in each partition and convert the expressions containing its elements to temporary variables
 - Replace all references to other sequences in the partition with an operator-neutral value
 - will “go away” after simplification

Re-association algorithm - example

■ Parallel sequences

- $\{(a[i],a[i+1]), (b[i],b[i+1]), (c[i],c[i+1])\}$

■ Partition parallel sequences

- $\{(a[i],a[i+1])\} \{(b[i],b[i+1]),(c[i],c[i+1])\}$

■ Assign temporary variables to elements of representative sequences

- $a[i] \rightarrow p0 \quad a[i+1] \rightarrow p1$
- $b[i] \rightarrow r0 \quad b[i+1] \rightarrow r1$

■ Replace all references to other sequences in the partition with an operator-neutral value

- All references of $c[i]$ and $c[i+1]$ will be replaced with zero

```
for (i=0; i < n; i++)
  f[i] = a[i]*(b[i]+c[i])+
        a[i+1]*(b[i+1]+d[i]+c[i+1]);
```

```
p0=a[0];
r0=b[0]+c[0];
for (i=0; i < n-n%2; i+=2) {
  p1=a[i+1]; r1=b[i+1]+c[i+1];
  f[i] = p0*r0 + p1*(r1+d[i]);
  p0=a[i+2]; r0=b[i+2]+c[i+2];
  f[i+1] = p1*r1 + p0*(r0+d[i+1]);
}
for (i=0; i < n; i++)
  f[i] = a[i]*(b[i]+c[i])+
        a[i+1]*(b[i+1]+d[i]+c[i+1]);
        a[i+1]*(b[i+1]+d[i]+c[i+1]);
```

mgrid (spec2000) – from subroutine psinv

```
DO I3=2,N-1
  DO I2=2,N-1
    DO I1=2,N-1
      U(I1,I2,I3)=U(I1,I2,I3)
      +C(0)*( R(I1, I2, I3 ) )
      +C(1)*( R(I1-1,I2, I3 ) + R(I1+1,I2, I3 )
      + R(I1, I2-1,I3 ) + R(I1, I2+1,I3 )
      + R(I1, I2, I3-1) + R(I1, I2, I3+1) )
      +C(2)*( R(I1-1,I2-1,I3 ) + R(I1+1,I2-1,I3 )
      + R(I1-1,I2+1,I3 ) + R(I1+1,I2+1,I3 )
      + R(I1, I2-1,I3-1) + R(I1, I2+1,I3-1)
      + R(I1, I2-1,I3+1) + R(I1, I2+1,I3+1)
      + R(I1-1,I2, I3-1) + R(I1-1,I2, I3+1)
      + R(I1+1,I2, I3-1) + R(I1+1,I2, I3+1) )
      +C(3)*( R(I1-1,I2-1,I3-1) + R(I1+1,I2-1,I3-1)
      + R(I1-1,I2+1,I3-1) + R(I1+1,I2+1,I3-1)
      + R(I1-1,I2-1,I3+1) + R(I1+1,I2-1,I3+1)
      + R(I1-1,I2+1,I3+1) + R(I1+1,I2+1,I3+1) )
    END DO
  END DO
END DO
```

mgrid (spec2000) – from subroutine psinv

```

DO I3=2,N-1
  DO I2=2,N-1
    R0=R(1,I2,I3)
    R1=R(2,I2,I3)
    R3=R(1,I2-1,I3-1)+R(1,I2+1,I3-1)+R(1,I2-1,I3+1)+R(1,I2+1,I3+1)
    R4=R(2,I2-1,I3-1)+R(2,I2+1,I3-1)+R(2,I2-1,I3+1)+R(2,I2+1,I3+1)
    R6=R(1,I2-1,I3)+R(1,I2+1,I3)+R(1,I2,I3-1)+R(1,I2,I3+1)
    R7=R(2,I2-1,I3)+R(2,I2+1,I3)+R(2,I2,I3-1)+R(2,I2,I3+1)
    DO I1=2,N-1-MOD(N-2,3),3
      R2=R(I1+1,I2,I3)
      R5=R(I1+1,I2-1,I3-1)+R(I1+1,I2+1,I3-1)+R(I1+1,I2-1,I3+1)+R(I1+1,I2+1,I3+1)
      R8=R(I1+1,I2-1,I3)+R(I1+1,I2+1,I3)+R(I1+1,I2,I3-1)+R(I1+1,I2,I3+1)
      U(I1,I2,I3)=U(I1,I2,I3)+C(0)*R1+C(1)*(R0+R2+R4)+C(2)*(R3+R5+R7)+ C(3)*(R6+R8)

      R0=R(I1+2,I2,I3)
      R3=R(I1+2,I2-1,I3-1)+R(I1+2,I2+1,I3-1)+R(I1+2,I2-1,I3+1)+R(I1+2,I2+1,I3+1)
      R6=R(I1+2,I2-1,I3)+R(I1+2,I2+1,I3)+R(I1+2,I2,I3-1)+R(I1+2,I2,I3+1)
      U(I1,I2,I3)=U(I1,I2,I3)+C(0)*R2+C(1)*(R1+R0+R5)+C(2)*(R4+R3+R8)+ C(3)*(R7+R6)

      R1=R(I1+3,I2,I3)
      R4=R(I1+3,I2-1,I3-1)+R(I1+3,I2+1,I3-1)+R(I1+3,I2-1,I3+1)+R(I1+3,I2+1,I3+1)
      R7=R(I1+3,I2-1,I3)+R(I1+3,I2+1,I3)+R(I1+3,I2,I3-1)+R(I1+3,I2,I3+1)
      U(I1,I2,I3)=U(I1,I2,I3)+C(0)*R0+C(1)*(R2+R1+R3)+C(2)*(R5+R4+R6)+C(3)*(R8+R7)
    END DO
    DO I1=MAX(2,N-1-MOD(N-2,3)),N-1
      < original loop body >
    END DO
  END DO
END DO

```

mgrid (spec2000) – from subroutine psinv

- **For each iteration**
 - Before transformation
 - 28 load operations
 - 27 add operations
 - After Second-Order Predictive Commoning
 - 10 load operations
 - 14 add operations
- **Overall improvement for mgrid**
 - Transformation applied to psinv and resid
 - 1.6x overall improvement of mgrid on Power4+

