CSC 488S/CSC 2107S Lecture Notes

Reading Assignment

```
Fischer, Cytron and LeBlanc
Section 14.1
```


```
References on Optimization

S. Muchnick, Advanced Compiler Design
   & Implementation, Morgan-Kaufmann, 1997

R. Morgan, Building an Optimizing Compiler,
Butterworth-Heinemann, 1998

Randy Allen & Ken Kennedy
Optimizing Compilers for Modern Architectures
Morgan-Kaufmann, 2001
```


Optimization

- Goals for Optimization
  - Make the object program faster and/or smaller, and/or use less power
    without changing the meaning of the program!!
  - To achieve object code as good as that produced by the best assembly language programmers. (Optimizing compilers do better than assembly language programmers on modern multi-issue RISC machines).
  - To produce object code that is *optimal*, i.e. minimum execution time or minimum space. Usually an NP-complete problem.
- Why optimization is needed
  - To improve on inefficient implementation of programming language constructs.
  - To mitigate the mismatch between high level programming language constructs and low level machine instructions.
  - To compensate for sloppy programming practices.
  - To improve programs in ways that can't be expressed in high level programming languages.
- Optimization **is not** a substitute for good algorithm selection.

Scope of optimization

peephole Optimize over a few machine instructions. (Slides 455 , 456)

local Optimize over a few statements.

Intraprocedural Optimize over the body of one routine.

Interprocedural Optimize over some collection of routines, e.g. all the member functions in a Class.

global Optimize over an entire program.

Why Optimization is Hard

- Optimization interacts with Programming Languages Features
  - Non compact data structures, e.g. subarrays.
  - Exception handling.
  - Overly specific language standards.
  - Symbolic debugging.
  - Aliasing of variables
  - Concurrency.
- Optimization interacts with target Hardware
  - Cache memory.
  - Virtual memory.
  - Register windows.
  - Instruction pipelining.
  - Super scalar, multi-issue instruction processors.
  - Multiple functional units.

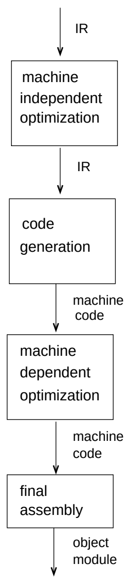
When is Optimization Worthwhile ?

- Most expressions in typical programs are very simple.
  For example the average expression in Fortran contains less than 1 operator.
- Big optimization gains come from optimizing loops and subscript calculation.[a]
- Generating locally good code to begin with always wins.
- Most interesting optimization problems are NP-complete.
  There are useful heuristics in many cases.
- Optimizations are usually divided into two categories
  - Machine Independent Optimizations
  - Machine Dependent Optimizations
- Historically, optimization is notorious for introducing bugs into a compiler.

[a]Review the discussion of array subscript polynomials in Slides 257 .. 263

Classical Optimizations

- Machine Independent Optimizations
  - Constant Folding
  - Common Subexpression Elimination
  - Algebraic Simplification
  - Code Motion (forward/backward)
  - Strength Reduction and Test Replacement
  - Dead Code Elimination
  - Loop Unrolling, Loop Fusion
  - Procedure/Function Integration
- Machine Dependent Optimizations
  - Register Allocation, allocating variables in registers
  - Use of hardware idioms
  - Peephole optimization
  - Instruction scheduling
  - Cache-sensitive code generation



- An optimizer will typically iterate over the machine independent optimizations using some heuristic strategy[a]
- Examples:
  - Constant folding may create more common subexpressions.
  - Algebraic simplification may create more common subexpressions.
  - Loop unrolling will create more common subexpressions (See Slide 484).
  - Procedure/Function integration may create more opportunities for constant folding and/or common subexpression elimination

[a]W. Wulf, *et.al.*,*The Design of an Optimizing Compiler*, American Elsevier, 1975 is a classic study into the development of these strategies. Also see R. Morgan, *Building and Optimizing Compiler* cited earlier

Optimization Inhibitors

The programming artifacts listed below affect the ability of the compiler to perform optimization. If these effects cannot be ruled out then the compiler must optimize very pessimistically.

- *Side Effects* Any construct that changes the value of a variable has a side effect on that variable. An optimizer needs to know when the value of a variable might change in order to optimize the use of variables. Example:

int I = 0 ;	
int F(**void**) {	
return ++I	J = F( ) + I + F( ) ;
}	

- *Exception Handler* An exception handling mechanism can be invoked asynchronously during the execution of a program and can in general cause side effects on variables. The occurrence of exception-caused side effects is extremely difficult to predict.

- *Aliasing* An alias exists when two identifiers refer to the same storage location. An optimizer needs to know the effect of every assignment statement. An alias causes an unexpected side effect. Example:

int I, J , K , A[100] ;	// Alias on I
void P(**int &** X , **int &** Y) {	P(I, J) ;
// X and Y are passed	
// by reference (address)	// Alias on A[47]
A[X] = A[Y] ;	P(A[47] , A[47]) ;
X = Y + A[Y] ;	
Y = I ;	// Alias on A[J] iff J == K
}	P(J , K) ;

Notation for Optimization Examples

- We use the operator @ as a C-style pointer dereferencing operator in these examples to avoid confusion with ∗ which is always multiply.
- We use a C-style assignment operator = that expects an l-value as its left operand and an r-value as its right operand
- & X is used for *address of* X
- Variables with names ending in P (e.g. AP, BP) are *byte pointer variables*.
- Most data items are assumed to be 4 bytes wide.

For most optimizations we will preferentially force address items into the form @ P + constant because this form fits well with the most common register + displacement addressing mode on many machines.
In most cases the + constant part can be absorbed into the displacement thus avoiding a run time addition.

Constant Folding

- If an expression involves only constants or other values known to the compiler, calculate the value of the expression at compile time.
- Issues
  - Compiler must contain a calculator for expressions.
    For cross compiler must calculate in the arithmetic of the target machine.
  - May want to extend to common builtin functions, e.g. max, sqrt.
  - Beware of arithmetic faults during constant expression evaluation, e.g. overflow, underflow, divide by zero.
  - May need to reorder expressions to facilitate constant folding, e.g. 3 + A + 4
- Example

#define aSize 100	
int A[aSize] ;	
A[aSize-1] = aSize ∗ aSize ;	@ (&A[0] + 4∗ (100-1)) = 100 ∗ 100
	@ (&A[0] + 396) = 10000

Common Subexpression Elimination

- If the same expression is calculated more than once, save the value of the first calculation and use it place of the repeated calculations.
- Issues
  - Detection of common subexpressions. Use heuristics, canonical ordering.
  - Must be able to detect when the values of the variables in an expression could be changed. The presence of functions with side effects, aliasing and exception handling make this more difficult.
- Example

A[J] = A[J] + 1 ;	@ (&A[0] + 4 ∗ J) = @ (&A[0] + 4 ∗ J) + 1
A[J + 1] = A[J] ;	@ (&A[0] + 4 ∗ J + 4) = @ (&A[0] + 4 ∗ J)
	AP = (&A[0] + 4 ∗ J)
	@ AP = @ AP + 1
	@ (AP+4) = @ AP

Example of CSE Optimization in Tuples

A[J] = A[J] + 1 ;
A[J + 1] = A[J] ;

Before					After				
201	mul	J	=4	R_{157}	201	mul	J	=4	R_{157}
202	add	&A[0]	R_{157}	R_{158}	202	add	&A[0]	R_{157}	R_{158}
203	mul	J	=4	R_{159}					
204	add	&A[0]	R_{159}	R_{160}					
205	load	R_{160}		R_{161}	205	load	R_{158}		R_{161}
206	add	R_{161}	=1	R_{162}	206	add	R_{161}	=1	R_{162}
207	store	R_{162}		R_{158}	207	store	R_{162}		R_{158}
208	add	J	=1	R_{163}	208	add	R_{158}	=4	R_{163}
209	mul	R_{163}	=4	R_{164}					
210	add	&A[0]	R_{164}	R_{165}					
211	mul	J	=4	R_{166}					
212	add	&A[0]	R_{166}	R_{167}					
213	load	R_{167}		R_{168}	213	load	R_{158}		R_{168}
214	store	R_{168}		R_{165}	214	store	R_{168}		R_{163}

Algebraic Simplification

- Use algebraic identities to simplify or reorder expressions. Use commutativity and associativity of operators. Often used to facilitate constant folding and common subexpression elimination.
- Issues
  - Must be careful not to change the meaning of the program.
- Examples

X + 0 ⇒ X	X - 0 ⇒ X	Y ∗ 1 ⇒ Y
Y / 1 ⇒ Y	X ∗ 0 ⇒ 0	-X + -Y ⇒ - (X + Y)
P **and** true ⇒ P	P **and** false ⇒ false	
P **or** true ⇒ true	P **or** false ⇒ P	

Code Motion

- Move code (statements, expressions or fragments thereof) out of loops (forward or backward) to places where they are executed less frequently.
- Code that is moved must be *loop invariant*, i.e. independent of the loop indices.
- Issues
  - Correctness (safety) of code motion.
  - Influenced by side effects, aliasing and exception handling.
  - Must preserve semantics of loops that never execute.
  - may need fixup code after the loop.
- Some optimizers do code motion by copy insertion followed by common subexpression elimination.

Code Motion Example

```
int J, K ;
float A[100][100], B[100] ;
for( J = 0 ; J < 100 ; J++ )
    for( K = 0 ; K < 100 ; K++ )
        A[J][K] = B[J] ;
```

```
for( J = 0 ; J < 100 ; J++ )
    for( K = 0 ; K < 100 ; K++ )
        @ (&A[0][0] + 400 ∗ J + 4 ∗ K)
            = @ (&B[0] + 4 ∗ J)
```

```
for( J = 0 ; J < 100 ; J++ ) {
    AP = &A[0][0] + 400 ∗ J
    BV = @ (&B[0] + 4 ∗ J)
    for( K = 0 ; K < 100 ; K++ )
        @ (AP + 4 ∗ K) = BV
}
```


Strength Reduction and Test Replacement

- Strength reduction is the replacement of "slow" operations (e.g. multiply and divide) with "faster" operations (e.g. add and subtract)
- Test Replacement occurs when (perhaps due to other optimizations) the body of a loop no longer contains any use of the loop index variable. Replace the loop index variable and the loop termination test.
- Issues
  - Deciding which operations are sufficiently "slow" as to warrant this optimization. Strength reduction was more important when multiply and divide were *really* slow relative to add and subtract.
  - Need to be careful not to change the meaning of the program.
- Strength reduction is often used to eliminate multiplications in array subscript polynomials. Often a precursor to machine dependent optimizations.

Strength Reduction Example

```
int J, K ;
float A[100][100], B[100] ;
for( J = 0 ; J < 100 ; J++ )
    for( K = 0 ; K < 100 ; K++ )
        A[J][K] = B[J] ;
```

```
for( J = 0 ; J < 100 ; J++ )
    for( K = 0 ; K < 100 ; K++ )
        @ (&A[0][0] + 400 ∗ J + 4 ∗ K)
            = @ (&B[0] + 4 ∗ J)
```

```
for( J4 = 0 , J400 = 0 , J = 0 ; J < 100 ;
                J++ , J4 += 4, J400 += 400 ) {
    AP = &A[0][0] + J400
    BV = @ (&B[0] + J4 )
    for( K = 0 , K4 = 4 ; K < 100
                ; K++, K4 += 4 )
        @ (AP + K4 ) = BV
}
```


Test Replacement Example

```
int J, K ;
float A[100][100], B[100] ;
for( J = 0 ; J < 100 ; J++ )
    for( K = 0 ; K < 100 ; K++ )
        A[J][K] = B[J] ;
```

```
for( J = 0 ; J < 100 ; J++ )
    for( K = 0 ; K < 100 ; K++ )
        @ (&A[0][0] + 400 ∗ J + 4 ∗ K)
            = @ (&B[0] + 4 ∗ J)
```

```
for( J4 = 0 , J400 = 0 ; J4 < 400 ;
                J4 += 4, J400 += 400 ) {
    AP = &A[0][0] + J400
    BV = @ (&B[0] + J4)
    for( K4 = 0 ; K4 < 400 ; K4 += 4 )
        @ (AP + K4) = BV
}
J = 100
K = 100
```

Dead Code and Useless Computation Elimination

- Compiler detects and eliminates code that can never be executed (dead code) and computations of values that are never subsequently used (useless computations).
- Issues
  - Detection of dead code requires analysis of control flow graph.
  - Useless calculations may result from programmer error or from other optimizations. Test replacement is a form of useless calculation elimination.
- Examples

if(true)	
X = Y ;	X = Y ;
else	
X = Z ;	
return ;	return ;
X = Y ;	

Variable Folding

- If a variable is assigned a "computationally simple" value, replace subsequent uses of the variable by the value. Creates new opportunities for constant folding and common subexpression elimination.
- Issues: aliasing, side effects.
- Examples

J = 13 ;	
A[J] = B[J] ∗ C[J] ;	A[13] = B[13] ∗ C[13] ;
K = J + 1 ;	
A[K] = B[K] ;	A[J + 1] = B[J + 1] ;

Loop Unrolling

- Replicate the body of a loop and adjust loop index. Reduces loop overhead relative to loop body. Enables common subexpression and constant folding optimizations.
- Issues:
  - detecting unrollable loops
  - aliasing & side effects
  - loop carried dependencies
  - handling loop remainder
- Example

for(J = 0 ; J < 200 ; J++)	for(J = 0 ; J < 200 ; J + = 4) {
A[J] = B[J] ∗ C[J] ;	A[J] = B[J] ∗ C[J] ;
	A[J + 1] = B[J + 1] ∗ C[J + 1] ;
	A[J + 2] = B[J + 2] ∗ C[J + 2] ;
	A[J + 3] = B[J + 3] ∗ C[J + 3] ;
	}

Optimizations After Loop Unrolling

A[J] = B[J] ∗ C[J] ;	@ (&A[0]+4*J) =@ (&B[0]+4*J) + @ (&C[0] + 4*J)
A[J + 1] = B[J + 1] ∗ C[J + 1] ;	@ (&A[0]+4*J+4) =@ (&B[0]+4*J+4) + @ (&C[0] + 4*J+4)
A[J + 2] = B[J + 2] ∗ C[J + 2] ;	@ (&A[0]+4*J+8) =@ (&B[0]+4*J+8) + @ (&C[0] + 4*J+8)
A[J + 3] = B[J + 3] ∗ C[J + 3] ;	@ (&A[0]+4*J+12) =@ (&B[0]+4*J+12) + @ (&C[0] + 4*J+12)
	J4 = 4 * J
	AJ4P = &A[0] + J4
	BJ4P = &B[0] + J4
	CJ4P = &C[0] + J4
	@ AJ4P = @ BJ4P + @ CJ4P
	@ (AJ4P + 4) = @ (BJ4P + 4) + @ (CJ4P + 4)
	@ (AJ8P + 8) = @ (BJ8P + 8) + @ (CJ8P + 8)
	@ (AJ12P + 12) = @ (BJ12P + 12) + @ (CJ12P + 12)

Loop Fusion (Jamming)

- Combine and simplify two or more loops that share a common index or range. Leads to common subexpression optimizations.
- Issues:
  - detection of fusible loops
  - side effects
  - aliasing
- Example

for(J = 0 ; J < 100 ; J++)
    for(K = 0 ; K < 100 ; K++)
        A[J][K] = 0 ;
for(J = 0 ; J < 100 ; J++)
    A[J][J] = 1 ;
	for(J = 0 ; J < 100 ; J++) {
	    for(K = 0 ; K < 100 ; K++)
	        A[J][K] = 0 ;
	    A[J][J] = 1 ;
	}

Routine Inlining

- Replace the call of a procedure or function with an inline expansion of the routine body with actual parameters substituted for the formal parameters.
- Eliminates routine call and return overhead.
  Exposes code in the expanded body to further optimizations.
- Issues
  - Aliasing and side effects. Recursive procedures.
  - Extreme care must be taken to preserve the semantics of parameter passing.
  - Often only done on leaf routines (i.e. routines that don't call any other routines. Candidate routines are those with no (or very little) local storage.
  - Must systematically rename any variables local to the routine to avoid accidental synonyms.
- This is an important optimization for Object Oriented languages where Objects export a lot of very small Object access routines (e.g. set and get functions in Java).

Routine Inlining Example

void swap(int * A , int * B) {
    int T = *A ;
    *A = *B ;
        *B = T ;
        return ;
}
...
int T , U ;
swap(&T , &U) ;
	
	{ int __T0023 = @ (&T) ;
	    @ (&T) = @ (& U) ;
	    @ (&U) = __T0023 ;
	}

Tail Recursion Elimination

- If the body of a recursive routine ends with a recursive call to the routine, replace the call with a setting of parameters followed by a branch to the start of the routine body.
- Saves routine call and return overhead.
  Saves activation record allocation/deallocation overhead *and a huge amount of stack space*.
- Issues:
  - Needs extreme care to preserve parameter passing semantics.
  - Often done where parameters are passed by value.
- This is a *really* important optimization in functional languages, e.g. Lisp, Scheme, ML

Tail Recursion Elimination Example

```
int Func( int A , int B ) {

    ...

    return Func( A - 1 , B + 1 );
}
```

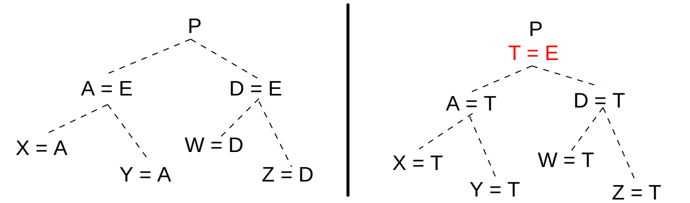
```
int Func( int A , int B ) {
    top:

    ...

    A = A - 1 ;
    B = B + 1 ;
    goto top ;
}
```


Code Hoisting

- If some expression is calculated on all paths leading *from* some point P in the programs control flow graph *and* the expression satisfies the conditions for common subexpression elimination then code for the expression can be moved (hoisted) to point P.
- Issues
  - Safety, aliasing, side effects.
  - Computationally expensive to detect hoisting opportunities.
  - Most useful if code is hoisted out of loops.
- Example



Machine Dependent Optimization

- Recognize specific programming language constructs that can be more efficiently implemented using specialized machine instructions
- Issues
  - Must maintain programming language semantics.
  - Be careful about interactions with other instructions, i.e. setting of condition codes.
- Examples

I = I + 1 ;	INC I	PDP-11, VAX
J = 0 ;	CLR J	PDP-11
A[$*$] = B[$*$] ;	MVCL A,B	System/370
for(I = 1 ; I <= 100 ; I++)	BXLE	System/370
X $* 2^n$	SLA rX,n	System/370
C = 'A' ;	MVI 'A',C	System/370

Matrix Multiplication Example
Initial Program

- Matrix multiplication: $C_{mn} = A_{mp} \times B_{pn}$

$$c_{i,j} = \sum_{k=1}^{p} a_{i,k} \star b_{k,j}$$

- Naive program - PL/I

```
DECLARE ( I, J, K ) FIXED BINARY(31,0) ;
DECLARE ( A( 20 , 10 ) , B( 10 , 20 ) , C( 20 , 20 ) ) FLOAT BINARY(31);
...
DO I = 1 TO 20 ;
   DO J = 1 TO 20 ;
      C( I , J ) = 0.0 ;
      DO K = 1 TO 10 ;
         C( I , J ) = C( I , J ) + A( I , K ) * B( K , J ) ;
      END ;
   END ;
END ;
```

Matrix Multiplication Example
Make Subscripts Explicit[a]

```
@ (&C(1 , 1) + 4 ∗ (20 ∗ (I - 1) + (J - 1) ) ) = 0.0 ;
DO K = 1 TO 10 ;
   @ (&C(1,1) + 4 ∗ (20 ∗ (I-1) + (J-1))) =
      @ (&C(1,1) + 4 ∗ (20 ∗ (I-1) + (J-1))) +
         @ (&A(1,1) + 4 ∗ (10 ∗ (I-1) + (K-1)))
            ∗ @ (&B(1,1) + 4 ∗ (20 ∗ (K-1) +(J-1))) ;
END ;
```

[a] &X[1 , 1] is the base address of array X.
PL/I arrays are 1-origin and FLOAT BINARY(31) is 4 bytes wide.
Review the discussion of array subscript polynomials in Slides 257 .. 263

Matrix Multiplication Example
Fold and Propagate Constants

```
@ (&C(1 , 1) + 80 ∗ I + 4 ∗ J - 84 ) ) = 0.0 ;
DO K = 1 TO 10 ;
   @ (&C(1 , 1) + 80 ∗ I + 4 ∗ J - 84 ) = @ (&C(1 , 1) + 80 ∗ I + 4 ∗ J - 84 ) +
   @ (&A(1 , 1) + 40 ∗ I + 4 ∗ K - 44) ∗ @ (&B(1 , 1) + 80 ∗ K + 4 ∗ J - 84 ) ;
END ;
```


Matrix Multiplication Example
Eliminate Common Subexpression

```
J484 = 4 ∗ J - 84 ;
@ (&C(1 , 1) + 80 ∗ I + J484 ) ) = 0.0 ;
DO K = 1 TO 10 ;
   @ (&C(1 , 1) + 80 ∗ I + J484 ) = @ (&C(1 , 1) + 80 ∗ I + J484 ) +
      @ (&A(1 , 1) + 40 ∗ I + 4 ∗ K - 44) ∗ @ (&B(1 , 1) + 80 ∗ K + J484 ) ;
END ;
```


Matrix Multiplication Example
Move Loop Independent Code Out of Loop

```
J484 = 4 ∗ J - 84 ;
AP = &A( 1 , 1 ) + 40 ∗ I - 44 ;
BP = &B( 1 , 1 ) + J484 ;
CP = &C( 1 , 1 ) + 80 ∗ I + J484 ;
@ CP = 0.0 ;
DO K = 1 TO 10 ;
   @ CP = @ CP + @ (AP + 4 ∗ K ) ∗ @ (BP + 80 ∗ K ) ;
END ;
```

Matrix Multiplication Example
Apply Strength Reduction

```
J484 = 4 ∗ J - 84 ;
AP = &A( 1 , 1 ) + 40 ∗ I - 44 ;
BP = &B( 1 , 1 ) + J484 ;
CP = &C( 1 , 1 ) + 80 ∗ I + J484 ;
@ CP = 0.0 ;
K4 = 4 ;
K80 = 80 ;
DO K = 1 TO 10 ;
    @ CP = @ CP + @ (AP + K4 ) ∗ @ (BP + K80 ) ;
    K4 = K4 + 4 ;
    K80 = K80 + 80 ;
END ;
```


Matrix Multiplication Example
Replace Test

```
J484 = 4 ∗ J - 84 ;
AP = &A( 1 , 1 ) + 40 ∗ I - 44 ;
BP = &B( 1 , 1 ) + J484 ;
CP = &C( 1 , 1 ) + 80 ∗ I + J484 ;
@ CP = 0.0 ;
K80 = 80 ;
DO K4 = 4 TO 40 BY 4 ;
    @ CP = @ CP + @ (AP + K4 ) ∗ @ (BP + K80 ) ;
    K80 = K80 + 80 ;
END ;
```


Matrix Multiplication Example
Machine Dependent Optimizations - System/370

- Accumulate sum in a register
  Use register-register instructions, loop variable in registers.
  Use BXLE for loop control, base+index addressing.
- Assume register assignments

R1 ← AP	R8 ← 4	R2 ← BP	R9 ← 40
R3 ← CP	R10 ← 80	R6 ← K80	F0 ← sum
F0 ← sum	F2 ← product		

- Use LM instruction to initialize all general registers in one instruction.
- Similar optimizations can be performed on the outer loops.

Matrix Multiplication Example

- System/370 Assembly Code for Loop

	SDR	F0,F0	F0 ← 0.0
KLOOP	LE	F2,0(R1,R7)	F2 ← @ (AP+K4)
	ME	F2,0(R2,R6)	F2 ← F2 ∗ @ (BP+K80)
	AER	F0,F2	F0 ← F0 + F2
	AR	R6,R10	K80 ← K80 + 80
	BXLE	R6,R8,R9,KLOOP	K4 ← K4 + 4 ; if(K4 <= 40) goto KLOOP
	STE	F0,0(,R3)	@ CP ← F0

- Optimized loop is 7 instructions, 22 bytes. 5 instruction inner loop.
  Unoptimized loop is approximately 37 instructions, 146 bytes.
- PL/I and Fortran optimizing compilers for the IBM System/370 achieve this level of optimization starting from the naive program.

C Optimization Example

```
float  A[ M ][ P ], B[ P ][ N ], C[ M ][ N ] ;
int  i, j, k ;
/* Naive Algorithm for      C = A x B */
for( i = 0 ; i < M ; i++ )
   for( j = 0 ; j < N ; j++ ) {
        C[i][j] = 0 ;
        for( k = 0 ; k < P ; k++ )
            C[i][j] = C[i][j] + A[i][k] * B[k][j] ;
   }
```

Sun Solaris Results

cc	cc -O	gcc	gcc -O2
100 instructions	25 instructions	94 instructions	18 instructions
74 inner loop	20 inner loop	90 inner loop	12 inner loop

gcc i686 unoptimized

```
.L7: movl    -8(%ebp),%ecx
     movl    -12(%ebp),%ebx
     movl    -8(%ebp),%eax
     movl    -12(%ebp),%edx
     imull    $200,%eax,%eax
     addl    %edx,%eax
     flds    C(,%eax,4)
     movl    -8(%ebp),%eax
     movl    -16(%ebp),%edx
     imull    $100,%eax,%eax
     addl    %edx,%eax
     flds    A(,%eax,4)
     movl    -16(%ebp),%eax
     movl    -12(%ebp),%edx
     imull    $200,%eax,%eax
     addl    %edx,%eax
     flds    B(,%eax,4)
     fmulp    %st,%st(1)
     faddp    %st,%st(1)
     imull    $200,%ecx,%eax
     addl    %ebx,%eax
     fstps    C(,%eax,4)
     addl    $1,-16(%ebp)
.L6: cmpl    $99,-16(%ebp)
     jle     .L7
```

gcc i686 -O3

```
.L4: flds    (%edx)
     addl    $1,%ecx
     addl    $4,%edx
     fmuls    (%eax)
     addl    $800,%eax
     cmpl    $100,%ecx
     faddp    %st,%st(1)
     jne    .L4
```


C Optimization Example

```
/*  C Hacking Algorithm (Don't Do This at Home)*/
float  A[ M ][ P ], B[ P ][ N ], C[ M ][ N ] ;
register float *ap, *bp, *cp, *aStart, *aEnd, *bStart, *bEnd, ;
register float  T , *aStop   ;
for( aStart = &A[0][0] , aEnd = &A[M][0] , cp =&C[0][0] ;
            aStart < aEnd  ;  aStart += P  ) {
    aStop = aStart + P ;
    for( bStart = &B[0][0] , bEnd = &B[0][N] ;
           bStart < bEnd  ;  bStart++ ) {
        for( ap = aStart, bp = bStart , T = 0.0 ,
              ; ap < aStop ; bp += N )
              T += *ap++ * *bp ;
        *cp++ = T ;
    }
}
```

Sun Solaris Results

cc	cc -O	gcc	gcc -O2
48 instructions	16 instructions	25 instructions	14 instructions
26 inner loop	12 inner loop	18 inner loop	8 inner loop

gcc i686 unoptimized

```
.L7: movl    -36(%ebp),%eax
     flds    (%eax)
     movl    -32(%ebp),%eax
     flds    (%eax)
     fmulp    %st,%st(1)
     flds    -8(%ebp)
     faddp    %st,%st(1)
     fstps    -8(%ebp)
     addl    $4,-36(%ebp)
     addl    $800,-32(%ebp)
.L6: movl    -4(%ebp),%eax
     cmpl    %eax,-36(%ebp)
     jb      .L7
```

gcc i686 -O3

```
.L8: flds    (%eax)
     addl    $4,%eax
     fmuls    (%edx)
     addl    $800,%edx
     cmpl    %ecx,%eax
     faddp    %st,%st(1)
     jne     .L8
```

gcc i686 Results

Naive Program		Hacking Program	
gcc	gcc -O3	gcc	gcc -O3
47 instructions	38 instructions	48 instructions	40 instructions
25 inner loop	8 inner loop	13 inner loop	7 inner loop

Value Numbering

- Value numbering is a technique for doing constant and variable folding and common subexpression elimination in a basic block.
- Value Numbering technique
  - Each distinct value created or used within a basic block is assigned a unique value number.
    Two values have the same value number if and only if they are provably identical.
  - Use a hash-coded table of available expressions to detect common subexpressions. Use value number plus operator as the hash key.
  - Constants are handled by special flags in the symbol table and the tuples
  - Start at the beginning of a basic block, Consider each quadruple in the IR in order.
  - Look the tuple up in the hash table to see the same value has already been computed.
  - If a tuple has already been computed it is a common subexpression and can be replaced by the previous computation.

Value Numbering Example

- Basic Block Source

  A = 4 ;
  K = I $*$ J + 5 ;
  L = 5 $*$ A $*$ K ;
  M = I ;
  B = M $*$ J + I $*$ A ;

- Quadruples

1	(assign	, =4	,	, A)
2	(mult	, I	, J	, R_1)
3	(add	, R_1	, =5	, R_2)
4	(assign	, R_2	,	, K)
5	(mult	, =5	, A	, R_3)
6	(mult	, R_3	, K	, R_4)
7	(assign	, R_4	,	, L)
8	(assign	, I	,	, M)
9	(mult	, M	, J	, R_5)
10	(mult	, I	, A	, R_6)
11	(add	, R_5	, R_6	, R_7)
12	(assign	, R_7	,	, B)

Value Numbering Data Structures

- *Symbol Table*
  Name, Value#, Constant Flag
- *Tuple Extension*
  Result Value#, Constant Flag
- *Available Expression Table*
  Left Value#, Operator, Right Value#
  Result Value#, Tuple#
- *Constant Value Table*
  Value#, Constant Value
- The following slides show two intermediate snapshots during processing of the basic block in Slide 506.

Value Numbering up to Tuple T_5

Symbol

Nm	Val#	Con?
=4	1	Yes
A	1	Yes
I	2	No
J	3	No
=5	5	Yes
K	6	No

Tuple Extensions

Tup	Res#	Con?
T_1	1	Yes
T_2	4	No
T_3	6	No
T_4	6	No

Available Expressions

Left#	Op	Right#	Res#	Tuple
2	$*$	3	4	T_2
4	+	5	6	T_3

Constants

Val#	Value
1	4
5	5

1	(assign	, =4	,	, A)
2	(mult	, I	, J	, R_1)
3	(add	, R_1	, =5	, R_2)
4	(assign	, R_2	,	, K)
5	(mult	, =5	, A	, R_3)
6	(mult	, R_3	, K	, R_4)

Action at T_5: Discover A and =5 are both constants
Add =20 to Constant Value Table. Delete T_5.
Modify T_6 to use new constant instead of R_3.

4	(assign	, R_2	,	, K)
6	(mult	, =20	, K	, R_4)

Value Numbering up to Tuple T_9

Symbol			Tuple Extensions			Available Expressions					Constants	
Nm	Val#	Con?	Tup	Res#	Con?	Left#	Op	Right#	Res#	Tuple	Val#	Value
=4	1	Yes	T_1	1	Yes	2	$*$	3	4	T_2	1	=4
A	1	Yes	T_2	4	No	4	+	5	6	T_3	5	=5
I	2	No	T_3	6	No	7	$*$	6	8	T_6	7	=20
J	3	No	T_4	6	No							
=5	5	Yes	T_5	7	Yes							
K	6	No	T_6	8	No							
=20	7	Yes	T_7	8	No							
L	8	No	T_8	2	No							
M	2	No										

7	(assign ,	R_4 ,	, L	)
8	(assign ,	I ,	, M	)
9	(mult ,	M ,	J ,	R_5)
10	(mult ,	I ,	A ,	R_6)
11	(add ,	R_5 ,	R_6 ,	R_7)

Action at T_9: Note that M has the same Value# as I.

Look up T_9 in Available Expressions table. Discover same value at T_2

Delete T_9, Replace reference to R_5 in T_{11} by R_1 .

11 (add , R_1 , R_6 , R_7)

RISC Machines

- Very simple instructions. Fairly uniform instruction size and timing.
  Only simple addressing modes. Often only register+displacement.
  Array subscripting requires explicit calculations.
  Often a Load/Store architecture. All arithmetic in registers.
- Many fast registers. Register window for parameter passing.
  Memory access is *very slow* relative to register access.
  Optimize usage of cache for maximum performance.
  Load and Store may have several instruction latency.
  Branching may involve several instruction latency and possibility of executing the instruction after the branch.
- Newer RISC machines have deep instruction pipelines.
  Code generation needs to try to keep pipeline full and busy.
  Branches tend to cause pipeline flushes.
  Newer RISC machines are *multi-issue*.
  Can issue (start) more than one instruction on each machine cycle.
  Usually complicated resource rules on which instructions can start together.

Optimizations for RISC Architectures

- Branch optimizations
  Eliminate branches wherever possible
  Missed branch predictions can stall/flush the pipeline
  Reorder/invert conditional branches for better branch prediction Replicate code to avoid branching.
- Optimize pipeline fill
  Schedule instructions to minimize operand delays
  Reorder instructions to maximize functional unit use.
- Optimize cache usage
  Try to organize code so that most code and data is in the first level cache.
  Split large code into blocks to reduce cache footprint
  Optimize use of cache lines to avoid cache thrashing.
  Reorder loads to reduce memory latency effects.

Optimizing for Pipelines

- Assume variable time instructions and a 4 stage pipeline
  Instruction fetch, operand fetch, execute, result store
- Goal is to schedule instructions so that the execution unit is always busy.
- Constraints on instruction scheduling
  - Operand wait (R_1)
    $R_1 \leftarrow R_x \; op_a \; R_y$
    $R_2 \leftarrow R_1 \; op_b \; R_z$
  - Delay store of result (R_1)
    $R_x \leftarrow R_y \; op_a \; R_1$
    $R_1 \leftarrow R_w \; op_b \; R_z$
  - Maintain order of writes (R_1)
    $R_1 \leftarrow R_w \; op_a \; R_x$
    $R_1 \leftarrow R_y \; op_b \; R_z$
  - Limits on available functional units (op_a)
    $R_u \leftarrow R_w \; op_a \; R_w$
    $R_x \leftarrow R_y \; op_a \; R_z$

Pipeline Execution Example

$$Y = A * X ** 2 + B * X + C$$

LOAD R1,X	R1 <- X
LOAD R2,A	R2 <- A
MR R0,R1,R1	R0 <- X**2
MR R6,R0,R2	R6 <- A*X**2
LOAD R3,B	R3 <- B
LOAD R4,C	R4 <- C
MR R3,R3,R1	R3 <- B*X
AR R5,R6,R3	R5 <- A*X**2+B*X
AR R7,R5,R4	R7 <- A*X**2+B*X+C
STORE R7,Y	Y <- R7