CprE 488 – Embedded Systems Design

Lecture 6 – Software Optimization

Joseph Zambreno
Electrical and Computer Engineering
Iowa State University

www.ece.iastate.edu/~zambreno
rcl.ece.iastate.edu

If you lie to the compiler, it will get its revenge. – Henry Spencer

A Motivating Example

- Any performance guesses?

	Code	-O0	-O3

a)	`for (i=0; i<N; i++)`
	`for (j=0; j<N; j++)`
	`A[i][j] = 0;`
		~1.40s	~0.84s

b)	`for (i=0; i<N; i++)`
	`for (j=0; j<N; j++)`
	`A[j][i] = 0;`
		~21.8s	~21.8s

c)	`p = &A[0][0];`
	`for (i=0; i<N*N; i++)`
	`*p++ = 0;`
		~1.59s	~0.83s

d)	`memset((void*)&A[0][0], 0, N*N*sizeof(int));`	~0.83s	~0.80s

- Assumptions:
  - N = 20000 (so 400,000,000 integers)
  - gcc 4.9.2 running on an Intel Core i7-6600U CPU @ 2.6 GHz

Compilers and Abstraction

- Compilers make abstraction affordable:
  - Cost of executing code should reflect the underlying work rather than the way the programmer chose to write it
  - Change in expression should bring small performance change

```
struct point {
  int x; int y;
}

void Padd(struct point p, struct point q, struct point *r) {
  r->x = p.x + q.x;
  r->y = p.y + q.y;
}

int main( int argc, char *argv[] ) {
  struct point p1, p2, p3;

  p1.x = 1; p1.y = 1;
  p2.x = 2; p2.y = 2;

  Padd(p1, p2, &p3);

  printf("Result is <%d,%d>.\n", p3.x, p3.y);
}
```

Example © Keith Cooper, Rice University

Compilers and Abstraction (cont.)

```
_main:
L5:
        popl    %ebx
        movl    $1, -16(%ebp)
        movl    $1, -12(%ebp)
        movl    $2, -24(%ebp)
        movl    $2, -20(%ebp)
        leal    -32(%ebp), %eax
        movl    %eax, 16(%esp)
        movl    -24(%ebp), %eax
        movl    -20(%ebp), %edx
        movl    %eax, 8(%esp)
        movl    %edx, 12(%esp)
        movl    -16(%ebp), %eax
        movl    -12(%ebp), %edx
        movl    %eax, (%esp)
        movl    %edx, 4(%esp)
        call    _PAdd
        movl    -28(%ebp), %eax
        movl    -32(%ebp), %edx
        movl    %eax, 8(%esp)
        movl    %edx, 4(%esp)
        leal    LC0-"L00000000001$pb"(%ebx), %eax
        movl    %eax, (%esp)
        call    L_printf$stub
        addl    $68, %esp
        popl    %ebx
        leave
        ret
```

Assignments to p1 and p2

Setup for call to PAdd

Setup for call to printf

Address calculation for format string in printf call

Compilers and Abstraction (cont.)

```
_PAdd:
        pushl   %ebp
        movl    %esp, %ebp
        subl    $8, %esp
        movl    8(%ebp), %edx
        movl    16(%ebp), %eax
        addl    %eax, %edx
        movl    24(%ebp), %eax
        movl    %edx, (%eax)
        movl    12(%ebp), %edx
        movl    20(%ebp), %eax
        addl    %eax, %edx
        movl    24(%ebp), %eax
        movl    %edx, 4(%eax)
        leave
        ret
```

Actual work

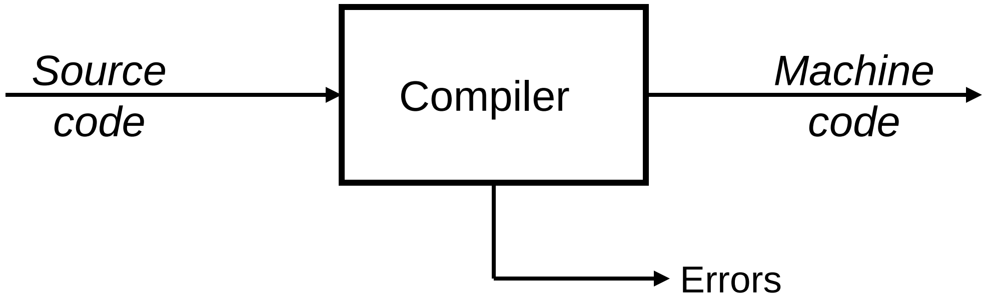
- The code does a lot of work to execute two add instructions (factor of 10 in overhead)
- Code optimization (careful compile-time reasoning & transformation) can make matters better

This Week's Topic

- The compiler's role in software optimization:
  - Early optimizations
  - Redundancy elimination
  - Loop restructuring
  - Instruction scheduling
  - Low-level optimizations
- Data representation
- Case study: MP-2 color space conversion

- Reading:
  - Wolf chapter 5

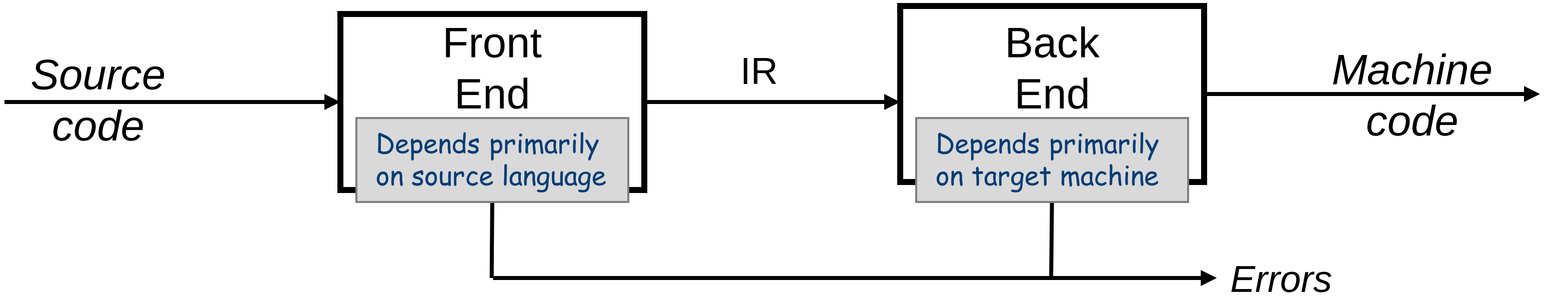
High-Level View of a Compiler



Implications

- Must recognize legal (and illegal) programs
- Must generate correct code
- Must manage storage of all variables (and code)
- Must agree with OS & linker on format for object code

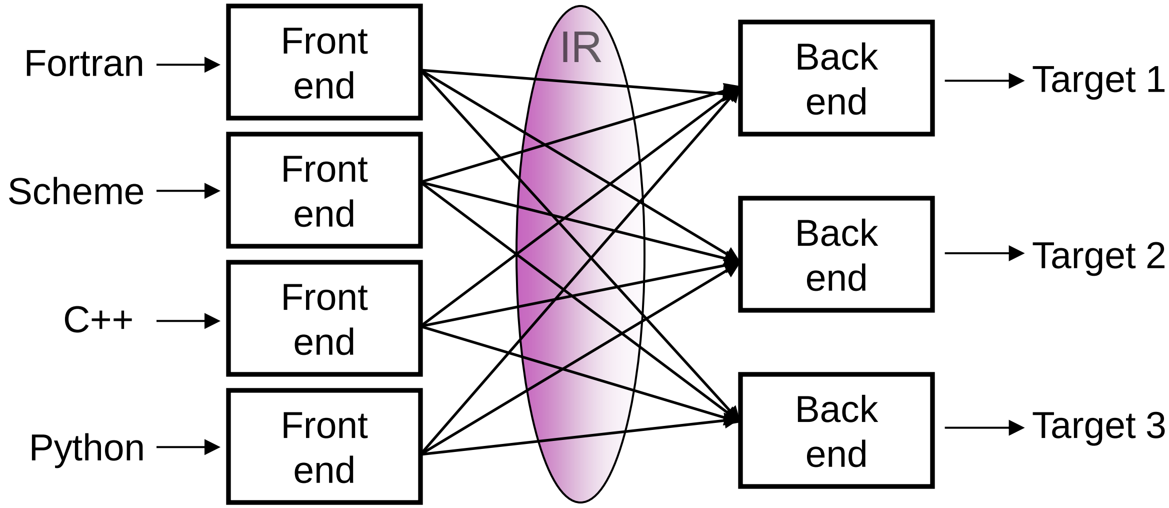
Traditional Two-Pass Compiler



Implications

- Use an intermediate representation (IR)
- Front end maps legal source code into IR
- Back end maps IR into target machine code
- Potentially multiple front ends & multiple passes

A Common Fallacy

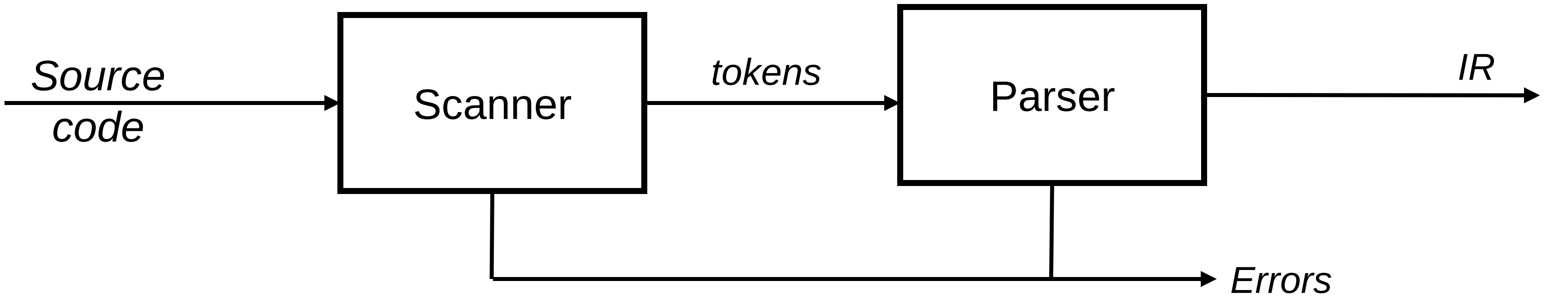


Can we build *n x m* compilers with *n + m* components?

- Must encode all language specific knowledge in each front end
- Must encode all features in a single IR (e.g. gcc rtl or llvm ir)
- Must encode all target specific knowledge in each back end

- Successful in systems with assembly level (or lower) IRs

The Front End

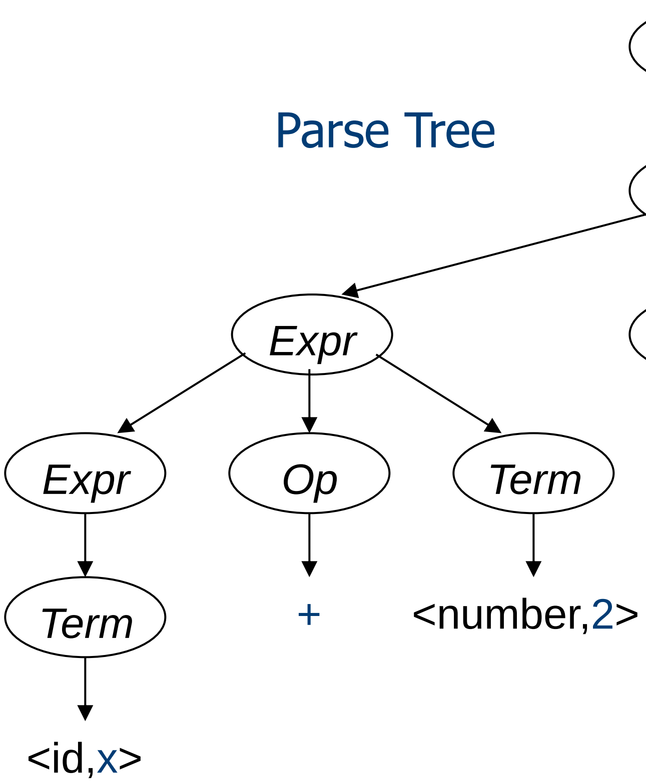


Responsibilities

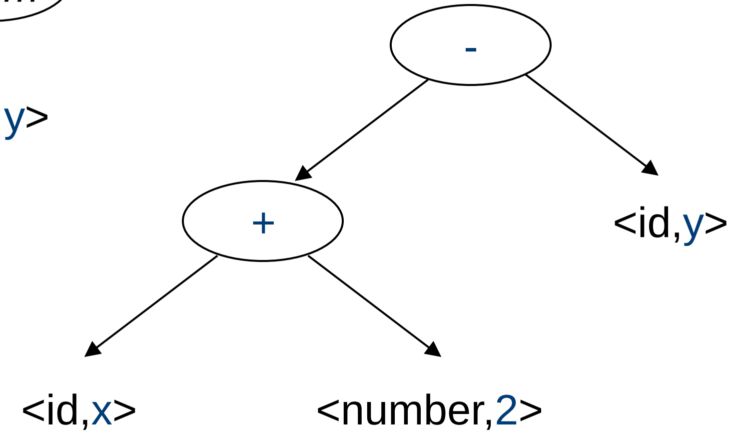
- Recognize legal (and illegal) programs
- Report errors in a useful way
- Produce IR and preliminary storage map
- Shape the code for the rest of the compiler
- Much of front end construction can be automated

The Front End (cont.)

- The parser output can be represented by a parse tree or an abstract syntax tree
  - Both trees represent expression: `x + 2 - y`

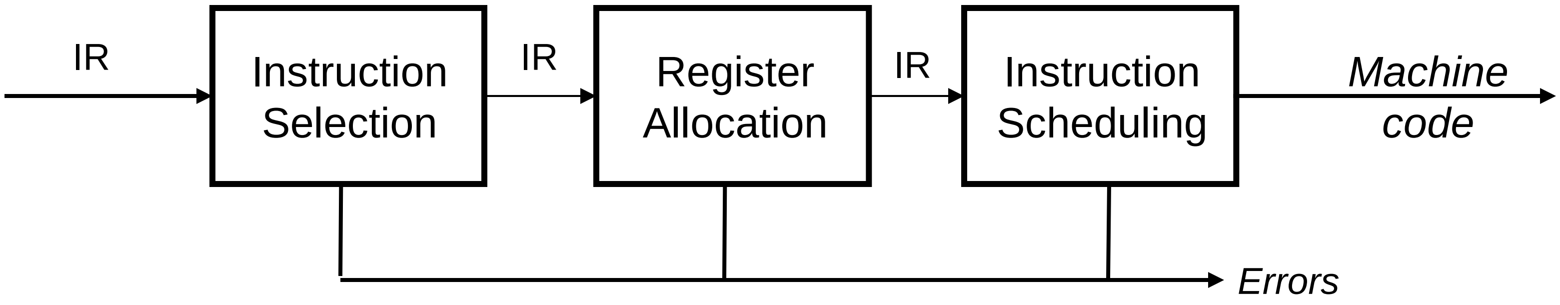


Parse Tree



Abstract Syntax Tree

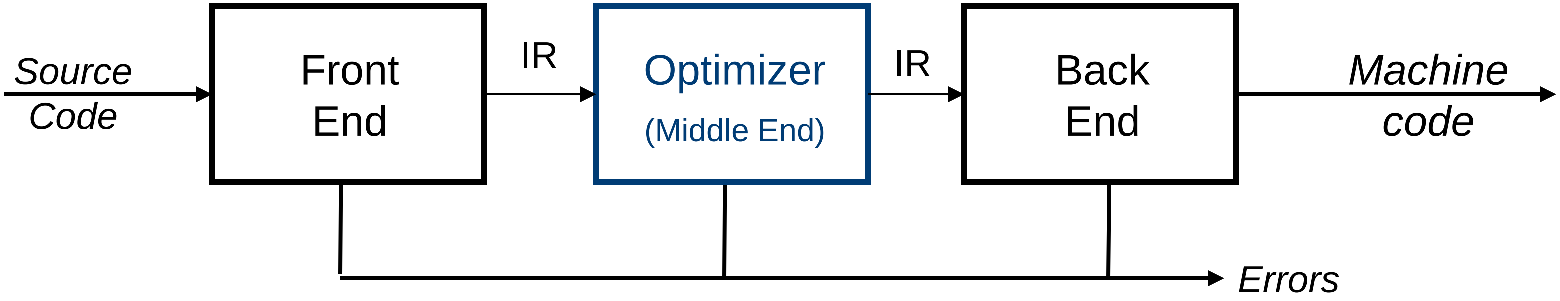
The Back End



Responsibilities

- Translate IR into target machine code
- Choose instructions to implement each IR operation
- Decide which values to keep in registers
- Ensure conformance with system interfaces

Traditional Three-Part Compiler

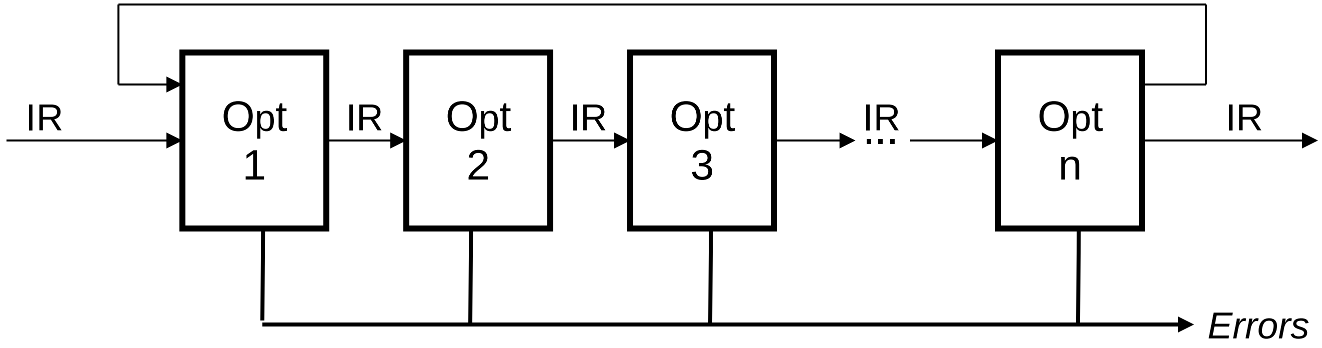


Code Improvement (or *Optimization*)

- Analyzes IR and rewrites (or *transforms*) IR
- Primary goal is to reduce running time of the compiled code
  - May also improve space, power consumption, …
- Must preserve "meaning" of the code
  - Measured by values of named variables

- Note that "optimization" is a misnomer – optimizations generally improve performance, although this is not typically guaranteed

The Optimizer



Modern optimizers are structured as a series of passes

Typical Transformations

- Discover & propagate some constant value
- Move a computation to a less frequently executed place
- Specialize some computation based on context
- Discover a redundant computation & remove it
- Remove useless or unreachable code
- Encode an idiom in some particularly efficient form

Types of (Classical) Optimizations

- Operation-level – 1 operation in isolation
  - Constant folding, strength reduction
  - Dead code elimination (global, but 1 op at a time)
- Local – pairs of operations in same basic block
- Global – again pairs of operations
  - But, operations in different basic blocks
  - More advanced dataflow analysis necessary here
- Loop – body of a loop
- Interprocedural – look across multiple function calls

Constant Folding

- Also known as constant-expression evaluation

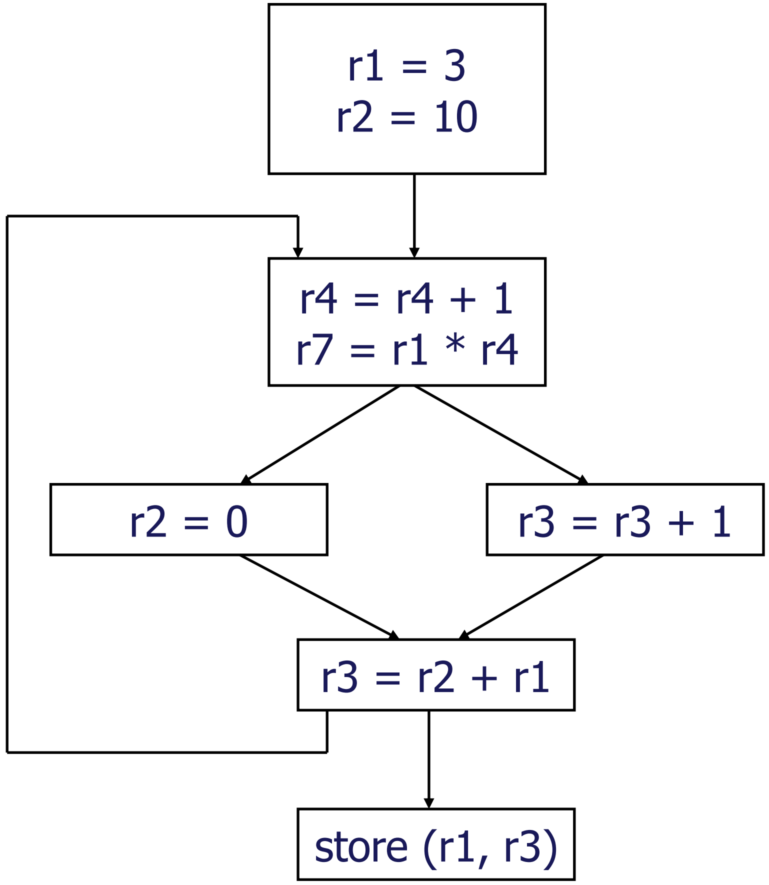
- Simplify operation based on values of source operands
  - Constant propagation creates opportunities for this
- All constant operands
  - Evaluate the op, replace with a move
    - r1 = 3 * 4 → r1 = 12
    - r1 = 3 / 0 → ??? Don't evaluate excepting ops !, what about FP?
  - Evaluate conditional branch, replace with branch or nop
    - if (1 < 2) goto BB2 → branch BB2
    - if (1 > 2) goto BB2 → convert to a nop
- Algebraic identities
  - r1 = r2 + 0, r2 – 0, r2 | 0, r2 ^ 0, r2 << 0, r2 >> 0 → r1 = r2
  - r1 = 0 * r2, 0 / r2, 0 & r2 → r1 = 0
  - r1 = r2 * 1, r2 / 1 → r1 = r2

Strength Reduction

- Replace expensive ops with cheaper ones
  - Constant propagation creates opportunities for this
- Power of 2 constants
  - Mult by power of 2: r1 = r2 * 8 → r1 = r2 << 3
  - Div by power of 2: r1 = r2 / 4 → r1 = r2 >> 2
  - Rem by power of 2: r1 = r2 REM 16 → r1 = r2 & 15
- More exotic
  - Replace multiply by constant by sequence of shift and adds/subs
    - r1 = r2 * 6
      - r100 = r2 << 2; r101 = r2 << 1; r1 = r100 + r101
    - r1 = r2 * 7
      - r100 = r2 << 3; r1 = r100 – r2
- Can be ISA dependent (remember ARM examples)

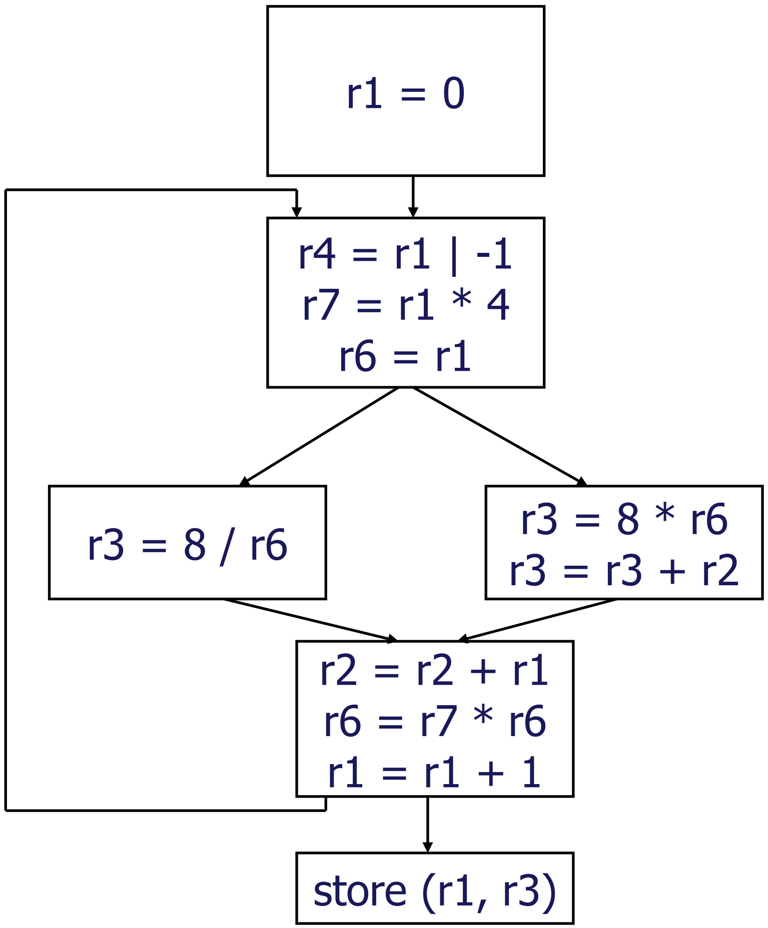
Dead Code Elimination

- Remove any operation whose result is never consumed
- Rules
  - X can be deleted
    - no stores or branches
  - DU chain empty or dest not live
- This misses some dead code!!
  - Especially in loops
  - Critical operation
    - store or branch operation
  - Any operation that does not directly or indirectly feed a critical operation is dead
  - Trace UD chains backwards from critical operations
  - Any op not visited is dead



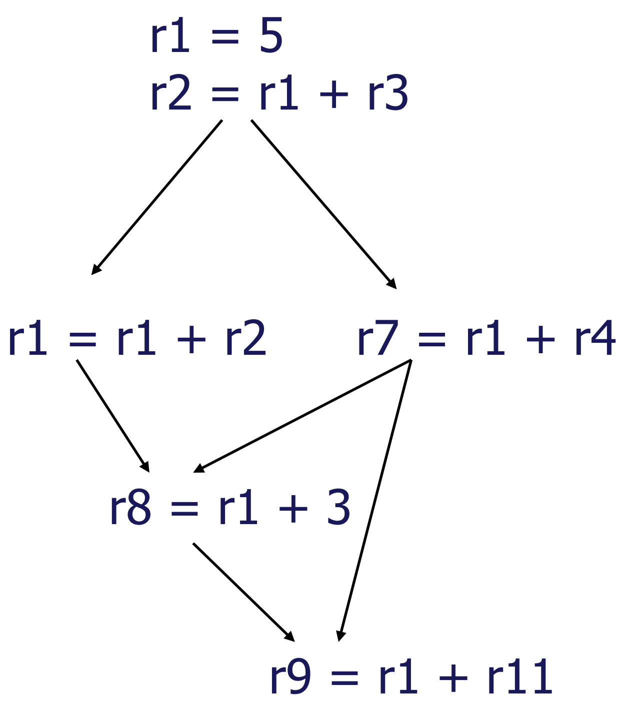
EX-06.1: Early Optimizations

- Optimize this block of code, using:
  - Constant folding
  - Strength reduction
  - Dead code elimination



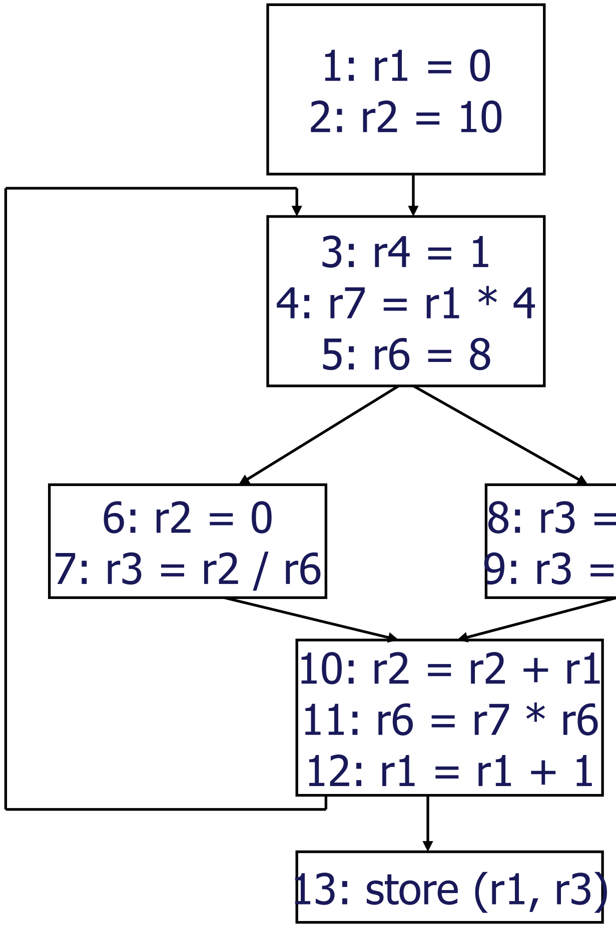
Constant Propagation

- Forward propagation of moves of the form
  - rx = L (where L is a literal)
  - Maximally propagate
  - Assume no instruction encoding restrictions
- When is it legal?
  - SRC: Literal is a hard coded constant, so never a problem
  - DEST: Must be available
    - Guaranteed to reach
    - May reach not good enough



EX-06.2: Constant Propagation

- Optimize this block of code, using:
  - Constant propagation
  - Constant folding
  - Strength reduction
  - Dead code elimination



Local Common Subexpression Elimination

- Eliminate recomputation of an expression
  - X: r1 = r2 * r3
  - → r100 = r1
  - …
  - Y: r4 = r2 * r3 → r4 = r100
- Benefits
  - Reduce work
  - Moves can get copy propagated
- Rules (ops X and Y)
  - X and Y have the same opcode
  - src(X) = src(Y), for all srcs
  - for all srcs(X) no defs of srci in [X … Y)
  - if X is a load, then there is no store that may write to address(X) between X and Y

```
r1 = r2 + r3
r4 = r4 +1
r1 = 6
r6 = r2 + r3
r2 = r1 -1
r6 = r4 + 1
r7 = r2 + r3
```

EX-06.3: Subexpression Elimination

- Optimize this block of code, using:
  - Constant propagation
  - Constant folding
  - Strength reduction
  - Dead code elimination
  - Common subexpression elimination

```
r1 = 9
r4 = 4
r5 = 0
r6 = 16
r2 = r3 * r4
r8 = r2 + r5
r9 = r3
r7 = load(r2)
r5 = r9 * r4
r3 = load(r2)
r10 = r3 / r6
store (r8, r7)
r11 = r2
r12 = load(r11)
store(r12, r3)
```

Loop Optimizations

- Arguably the most important set of optimizations (why?)
- Many optimizations are possible
  - Loop invariant code motion
  - Global variable migration
  - Induction variable optimizations
  - Loop restructuring (unrolling, tiling, etc.)

Loop Unswitching

- Removes loop independent conditionals from a loop

```
for i=1 to N do
    for j=2 to N do
        if T[i] > 0 then
            A[i,j] = A[i, j-1]*T[i] + B[i]
        else
            A[i,j] = 0.0
        endif
    endfor
endfor
```

```
for i=1 to N do
    if T[i] > 0 then
        for j=2 to N do
            A[i,j] = A[i, j-1]*T[i] + B[i]
        endfor
    else
        for j=2 to N do
            A[i,j] = 0.0
        endfor
    endif
endfor
```

- Advantage: reduces the frequency of execution of the conditional statement
- Disadvantages: Loop structure is more complex, code size expansion

Loop Peeling

- Separates the first (or last) iteration of the loop

```
for i=1 to N do
    A[i] = (X+Y)*B[i]
endfor
```

```
if N >= 1 then
    A[1] = (X+Y)*B[1]
    for j=2 to N do
        A[j] = (X+Y)*B[j]
    endfor
endif
```

- Advantage: Used to enable loop fusion or remove conditionals on the index variable from inside the loop. Allows execution of loop invariant code only in the first iteration
- Disadvantages: Code size expansion

Index Set Splitting

- Divides the index into two portions

```
for i=1 to 100 do
    A[i] = B[i] + C[i]
    if i > 10 then
        D[i] = A[i] + A[i-10]
    endif
endfor
```

```
for i=1 to 10 do
    A[i] = B[i] + C[i]
endfor
for i=11 to 100 do
    A[i] = B[i] + C[i]
    D[i] = A[i] + A[i-10]
endfor
```

- Advantage: Used to enable loop fusion or remove conditionals on the index variable from inside the loop. Can remove conditionals that test index variables.
- Disadvantages: Code size expansion

Scalar Expansion

- Breaks anti-dependence relations by expanding, or promoting a scalar into an array

```
for i=1 to N do
    T = A[i] + B[i]
    C[i] = T + 1/T
endfor
```

```
if N >= 1 then
    allocate Tx(1:N)
    for i=1 to N do
        Tx[i] = A[i] + B[i]
        C[i] = Tx[i] + 1/Tx[i]
    endfor
    T = Tx[N]
endif
```

- Advantage: Eliminates anti-dependences and output dependences
- Disadvantages: In nested loops the size of the array might be prohibitive

Loop Fusion

- Takes two adjacent loops and generates a single loop

```
(1) for i=1 to N do
(2)    A[i] = B[i] + 1
(3) endfor
(4) for i=1 to N do
(5)     C[i] = A[i] / 2
(6) endfor
(7) for i=1 to N do
(8)     D[i] = 1 / C[i+1]
(9) endfor
```

```
(1) for i=1 to N do
(2)     A[i] = B[i] + 1
(5)     C[i] = A[i] / 2
(8)     D[i] = 1 / C[i+1]
(9) endfor
```

- Advantage: Eliminates loop iteration code
- Disadvantages: Potential locality implications, anything else????

Loop Fusion (cont.)

- To be legal, a loop transformation must preserve all the data dependencies of the original loop(s)

```
(1) for i=1 to N do
(2)    A[i] = B[i] + 1
(3) endfor
(4) for i=1 to N do
(5)     C[i] = A[i] / 2
(6) endfor
(7) for i=1 to N do
(8)     D[i] = 1 / C[i+1]
(9) endfor
```

The original loop has the flow dependencies:

$S_2 \; \delta^f \; S_5$

$S_5 \; \delta^f \; S_8$

What are the dependences in the fused loop?

```
(1) for i=1 to N do
(2)     A[i] = B[i] + 1
(5)     C[i] = A[i] / 2
(8)     D[i] = 1 / C[i+1]
(9) endfor
```

Loop Fission (Loop Distribution)

- Breaks a loop into multiple smaller loops

```
(1) for i=1 to N do
(2)    A[i] = A[i] + B[i-1]
(3)    B[i] = C[i-1]*X + Z
(4)    C[i] = 1/B[i]
(5)    D[i] = sqrt(C[i])
(6) endfor
```

```
(1) for ib=0 to N-1 do
(3)    B[ib+1] = C[ib]*X + Z
(4)    C[ib+1] = 1/B[ib+1]
(6) endfor
(1) for ib=0 to N-1 do
(2)    A[ib+1] = A[ib+1] + B[ib]
(6) endfor
(1) for ib=0 to N-1 do
(5)    D[ib+1] = sqrt(C[ib+1])
(6) endfor
(1) i = N+1
```

- Advantage: can improve cache use in machines with very small caches. Can be required for other transformations, such as loop interchanging.
- Disadvantages: Code size increase

Loop Interchange

- Reverses the order of nested loops

```
(1) for j=2 to M do
(2)     for i=1 to N do
(3)         A[i,j] = A[i,j-1] + B[i,j]
(4)     endfor
(5) endfor
```

```
(1) for i=1 to N do
(2)     for j=2 to M do
(3)         A[i,j] = A[i,j-1] + B[i,j]
(4)     endfor
(5) endfor
```

- Advantage: can reduce the startup cost of the inner-most loop. Can enable vectorization
- Disadvantages: can change the locality of memory references

Loop Unrolling

- Replicates the loop body
- Benefits:
  - Reduces loop overhead
  - Increased ILP (esp. VLIW)
  - Improved locality (consecutive elements)

```
do i = 2, n-1
    a[i] = a[i] + a[i-1] * a[i+1]
end do
```

```
do i = 1, n-2, 2
    a[i] = a[i] + a[i-1] * a[i+1]
    a[i+1] = a[i+1] + a[i] * a[i+2]
end do
if (mod(n-2,2) = 1) then
    a[n-1] = a[n-1] + a[n-2] * a[n]
end if
```

Induction Variable Elimination

- Frees the register used by the variable, reduces the number of operations in the loop framework

```
for(i = 0; i < n; i++) {
      a[i] = a[i] + c;
}
```

```
A = &a;
T = &a + n;
while(A < T){
      *A = *A + c;
      A++;
}
```

Loop Invariant Code Motion

- A specific case of code hoisting
- Needs a register to hold the invariant value
  - Ex: multi-dim. indices, pointers, structures

```
do i = 1, n
    a[i] = a[i] + sqrt(x)
end do
```

```
if (n > 0) C = sqrt(x)
do i = 1, n
    a[i] = a[i] + C
end do
```

Strip Mining

- Adjusts the granularity of an operation
  - usually for vectorization
  - also controlling array size, grouping operations
- Often requires other transforms first

```
do i = 1, n
    a[i] = a[i] + c
end do
```

```
TN = (n/64)*64
do TI = 1, TN, 64
    a[TI:TI+63] = a[TI:TI+63] + c
end do
do i= TN+1, n
    a[i] = a[i] + c
end do
```

Loop Tiling

- Multidimensional specialization of strip mining
- Goal: to improve cache reuse
- Adjacent loops can be tiled if they can be interchanged

```
do i = 1, n
    do j = 1, n
        a[i,j] = b[j,i]
    end do
end do
```

```
do TI = 1, n, 64
    do TJ = 1, n, 64
        do i = TI, min(TI+63, n)
            do j = TJ, min(TJ+63, n)
                a[i,j] = b[j,i]
            end do
        end do
    end do
end do
```

Fixed Point Representation

– Insert implicit “binary point” between two bits

– Bits to left of point have value ≥ 1

– Bits to right of point have value < 1

binary point

2^9	2^8	2^7	2^6	2^5	2^4	2^3	2^2	2^1	2^0	2^{-1}	2^{-2}	2^{-3}	2^{-4}	2^{-5}	2^{-6}
(512)	(256)	(128)	(64)	(32)	(16)	(8)	(4)	(2)	(1)	(0.5)	(0.25)	(0.125)	(0.0625)	(0.03125)	(0.015625)
0	1	0	0	0	0	1	1	0	1.	0	1	1	1	1	0

269.46875

Converting to Fixed point

1. Take fractional part and multiply by 2
2. If the result is > 1, then answer is 1, if 0 then answer is 0
3. Start again with the remaining decimal part, until you get an answer of 0

- E.g.

  Convert 0.75 to fixed point

0.75 * 2	= 1.5	Use 1
0.5 * 2	= 1.0	Use 1

  Ans: 0.75 in Decimal = 0.11 in binary

Fixed Point Pros and Cons

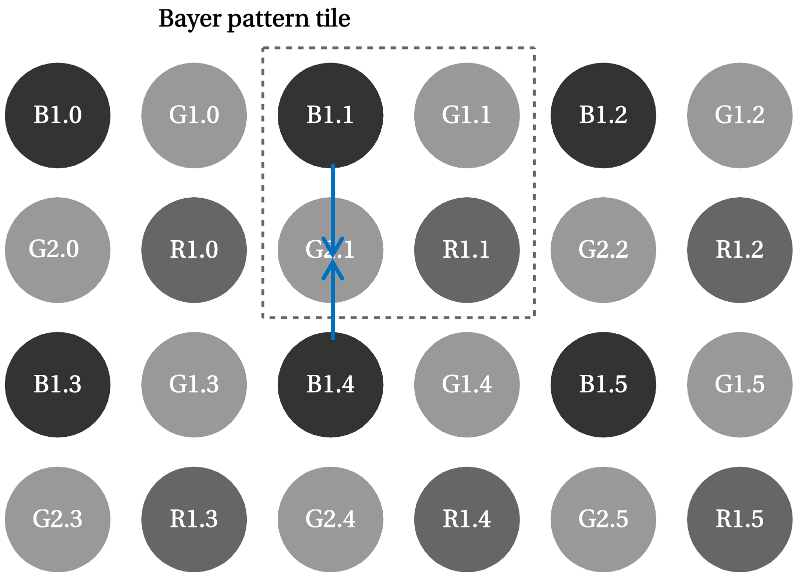
- Pros – simplicity:
  - The same hardware that does integer arithmetic can do fixed point arithmetic
  - In fact, the programmer can use ints with an implicit fixed point (ints are just fixed point numbers with the binary point to the right of b_0)

- Cons – there is no good way to pick where the fixed point should be
  - Sometimes you need range, sometimes you need precision. The more you have of one, the less of the other
  - Can only exactly represent numbers of the form $x/2^k$
    - Other rational numbers have repeating bit representations

Value	Representation
1/3	$0.0101010101[01]..._2$
1/5	$0.001100110011[0011]..._2$
1/10	$0.0001100110011[0011]..._2$

Putting it All Together: MP-2 Optimization

- Color filter array:



- Color space conversion:

$$[Y \quad Cb \quad Cr] = \begin{bmatrix} 0.183 & 0.614 & 0.062 \\ -0.101 & -0.338 & 0.439 \\ 0.439 & -0.399 & -0.040 \end{bmatrix} \cdot \begin{bmatrix} R \\ G \\ B \end{bmatrix} + \begin{bmatrix} 16 \\ 128 \\ 128 \end{bmatrix}$$

- Chroma resampling:
  – Output pattern – Cb-Y, Cr-Y, Cb-Y, Cr-Y, …

Acknowledgments

- These slides are inspired in part by material developed and copyright by:
  - Marilyn Wolf (Georgia Tech)
  - Keith Cooper (Rice University)
  - Scott Mahlke (University of Michigan)
  - José Amaral (University of Alberta)