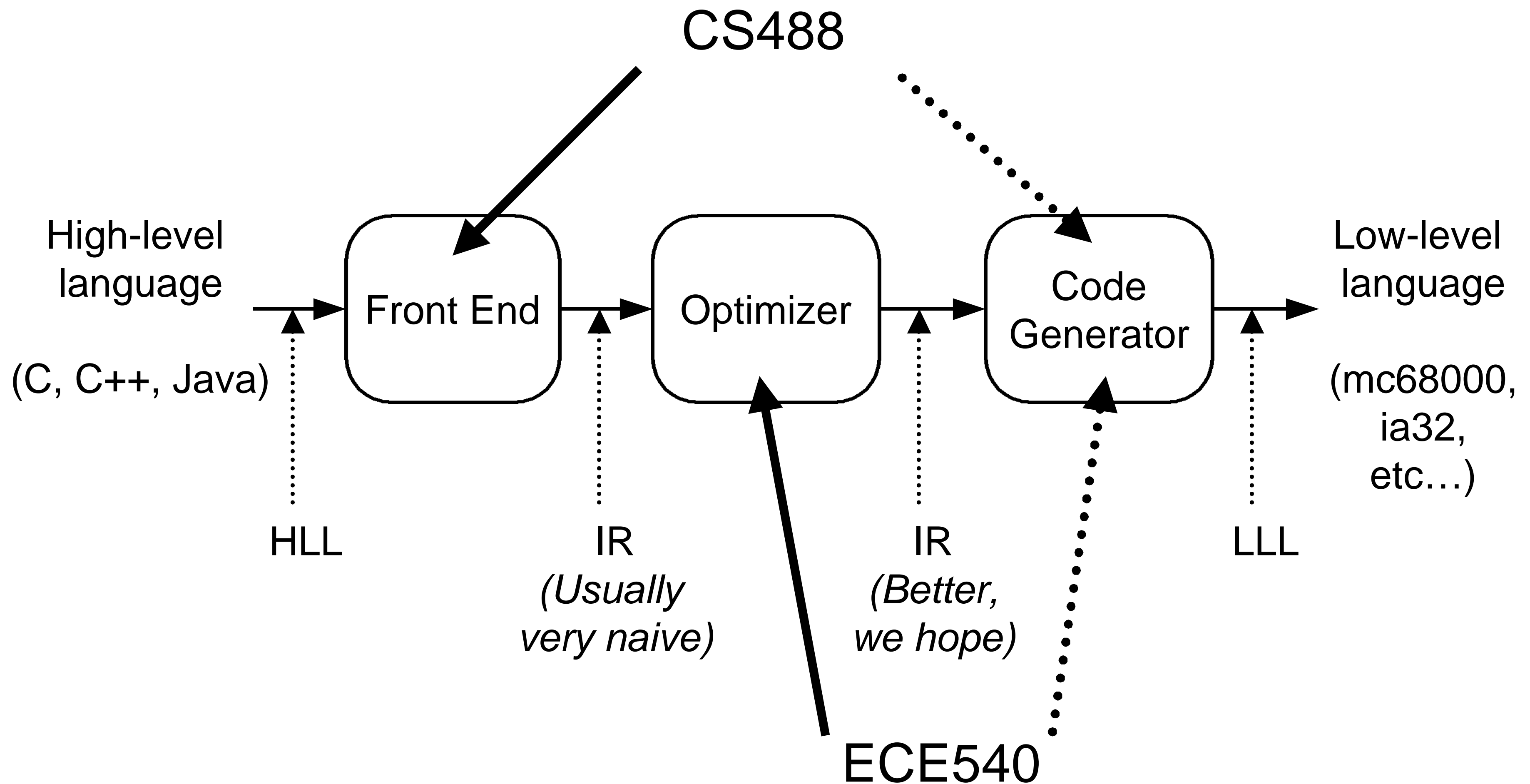
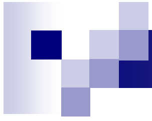
What's in an optimizing compiler?



What are compiler optimizations?

Optimization: the transformation of a program P into a program P', that has the same input/output behavior, but is somehow "better".

- "better" means:
  - faster
  - or smaller
  - or uses less power



Optimizations

Simple Optimizations

- Constant Folding
- Algebraic Simplifications

Redundancy optimizations

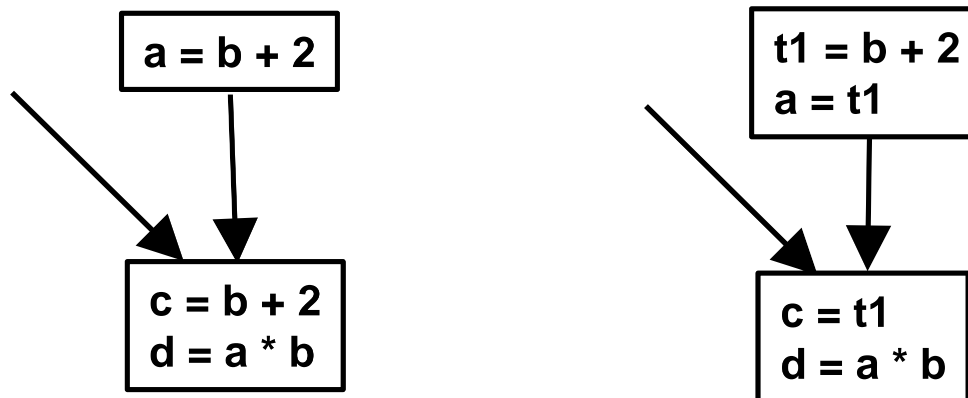
- Value numbering
- Common subexpression elimination
- Forward substitution (reverse of CSE)
- Copy propagation
- Loop-invariant code motion
- Code Hoisting

Common Subexpression Elimination

- An occurrence of an expression is a ***common subexpression*** if there is another occurrence that always precedes it in execution order and whose operands remain unchanged between these evaluations.
  - i.e. the expression has been already computed and the result is still valid.
- *Common Subexpression Elimination* replaces the recomputations with a saved value.
  - reduces the number of computations

Forward Substitution

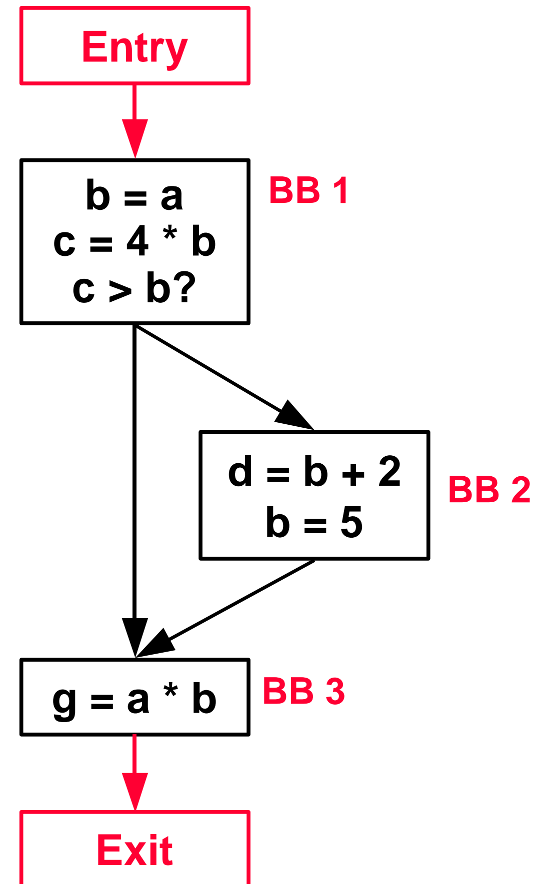
- Replace a copy by reevaluation of the expression
- Why?
  - perhaps holds a register too long, causes spills



- See that you have a store of an expression to a temporary followed by an assignment to a variable. If the expression operands are not changed to point of substitution replace with expression.

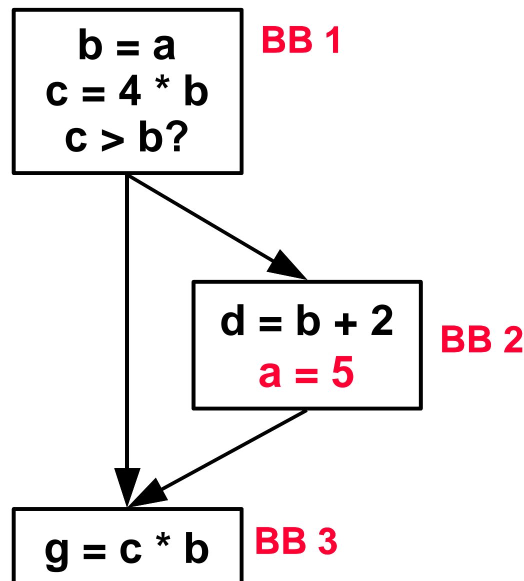
Copy Propagation

- A ***copy instruction*** is an instruction in the form x = y.
- ***Copy propagation*** replaces later uses of x with uses of y provided intervening instructions do not change the value of either x or y.
- Benefit: saves computations, reduces space; enables other transformations.



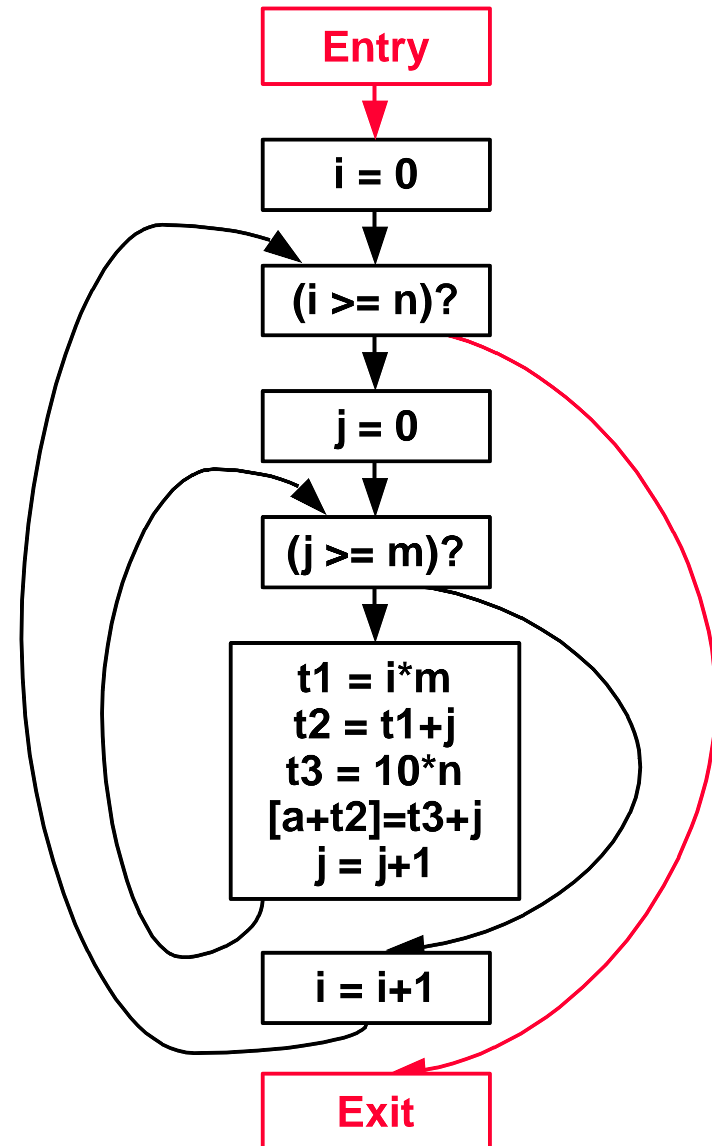
Copy Propagation (cont…)

- To propagate a copy statement in the form s: x = y, we must:
  - Determine all places where this definition of x is used.
  - For each such use, u:
    - s must be the only definition of x reaching u; and
    - on every path from s to u, there are no assignments to y.



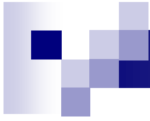
Loop-Invariant Code Motion

- A computation inside a loop is said to be loop-invariant if its execution produces the same value as long as control stays within the loop.
- Loop-invariant code motion moves such computations outside the loop (into the loop pre-header).
- Benefit: eliminates redundant computations.



Code Hoisting

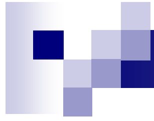
- *Code Hoisting* finds expressions that are always evaluated following a point p and moves them to the latest point beyond which they are always evaluated.
  - reduces space occupied by the program
  - may impact execution time positively, negatively or not at all
- Uses Very Busy Expressions



Removing Deadcode

Deadcode Elimination

- A *variable* is *dead* if it is not used on any path from the location in the code where it is defined to the exit point of the routine in question.
- An *instruction* is dead if it computes a dead variable.
- A local variable is dead if it is not used before the procedure exit
- A variable with wider visibility may require interprocedural analysis unless it is reassigned on every possible path to the procedure exit.



Loop Optimizations

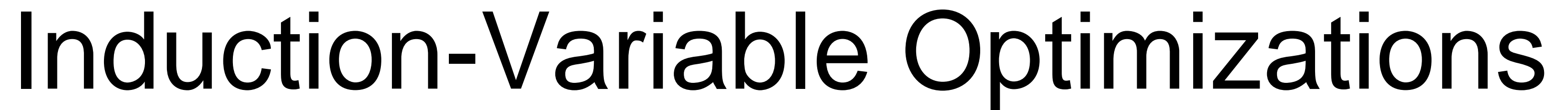
Well-behaved loops

- Fortran and Pascal have all *well-behaved* loops
- For C, only a subset are well-behaved, defined as

for (*exp1*; *exp2*; *exp3*)
    stmt

where: exp1 assigns a value to an integer variable ***i***
exp2 compares ***i*** to a loop constant
exp3 increments or decrements ***i*** by a loop constant

Similar if-goto loops can also be considered well-behaved



Induction-Variable Optimizations

- *induction-variables* are variables whose successive values form an arithmetic progression over some part of a program, usually a loop.
  - A loop's iterations are usually counted by an integer variable that increases/decreases by a constant amount each iteration
  - Other variables, e.g. subscripts, often follow patterns similar to the loop-control variable's.

Induction Variables: Example

```
INTEGER A(100)
DO I = 1,100
  A(I) = 202 - 2*I
ENDDO
```

```
INTEGER A(100)
T1 = 202
DO I = 1,100
  T1 = T1 - 2
  A(I) = T1
ENDD
```

- `I` has an initial value 1, increments by 1, and ends as 100
- `A(I)` is initially assigned 200, decreases by 2, and ends as 2
- The address of `A(I)` is initially addr a, increases by 4 each iteration, and ends as (addr a) + 396
  - addr a(i) = (addr a) + 4 * i - 4

Induction Variables: Example

```
     t1 = 202
     i = 1
L1:  t2 = i > 100
     if t2 goto L2
     t1 = t1 - 2
     t3 = addr a
     t4 = t3 - 4
     t5 = 4 * i
     t6 = t4 + t5
     [t6] = t1
     i = i + 1
     GOTO L1
L2:
```

```
     t1 = 202
     t3 = addr a
     t4 = t3 - 4
     t5 = 4
     t6 = t4
     t7 = t3 + 396
L1:  t2 = t6 > t7
     if t2 goto L2
     t1 = t1 - 2
     t6 = t4 + t5
     [t6] = t1
     t5 = t5 + 4
     GOTO L1
L2:
```

- `i` is used to count iterations and calculate `A(I)`
- Induction variable optimizations improve if preceded by constant propagation

Other important optimizations

- Instruction scheduling
  - what types of ops to use on an architecture and how to order them in a BB
- Parallelization (Dependence Analysis)
- Locality Optimizations
  - change ordering of loops and instructions to benefit cache behavior…