Overview of Optimization

- An Example of Code Optimization
- Overview of Optimization Concepts

An Example of Code Optimization

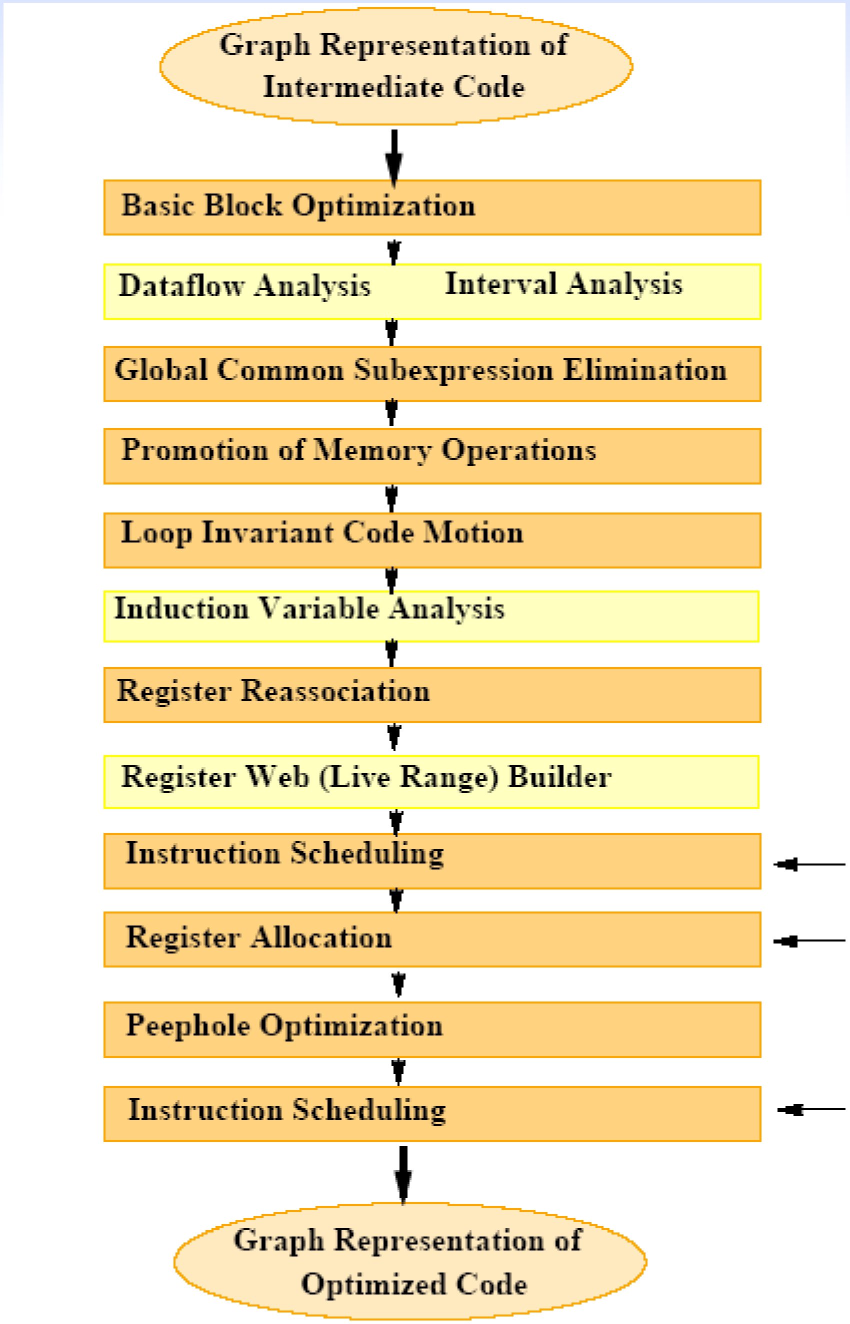
```
--------------- Source C Code ------------------
int a[25][25];
main()
{
      int i;
      for (i=0; i<25; i++)
            a[i][0] = 0;
}

------------- Optimized Assembly Code ---------------

            ADDIL   LR'a-$global$,%r27                  ;offset 0x0
            LD0     RR'a-$global$(%r1),%r31             ;offset 0x4
            LDI     -25,%r23                            ;offset 0x8
$00000003
            ADDIB,<          1,%r23,$00000003           ;offset 0xc
            STWM    %r0,100(%r31)                       ;offset 0x10
```

Un-optimized Code

```
          STW         0,-40(30)
          LDW         -40(30),206
          LDI         25,212
          IFNOT       206 < 212 GOTO $00000002
          NOP
$00000003
          LDW         -40(30),206
          ADDILG      LR'a-$global$,27,213
          LDO         RR'a-$global$(213),208
          MULTI       100,206,214
          ADD         208,214,215
          STWS        0,0(215)
          LDW         -40(30),206
          LDO         1(206),216
          STW         216,-40(30)
          LDW         -40(30),206
          LDI         25,212
          IF          206 < 212 GOTO $00000003
          NOP
$00000002
```

register notations
30: stack pointer
27: pointer to global data area

```
int a[25][25];
main()
{
  int i;
  for (i=0; i<25; i++)
      a[i][0] = 0;
}
```


Graph Representation of Intermediate Code
Basic Block Optimization
Dataflow Analysis
Interval Analysis
Global Common Subexpression Elimination
Promotion of Memory Operations
Loop Invariant Code Motion
Induction Variable Analysis
Register Reassociation
Register Web (Live Range) Builder
Instruction Scheduling
Register Allocation
Peephole Optimization
Instruction Scheduling
Graph Representation of Optimized Code

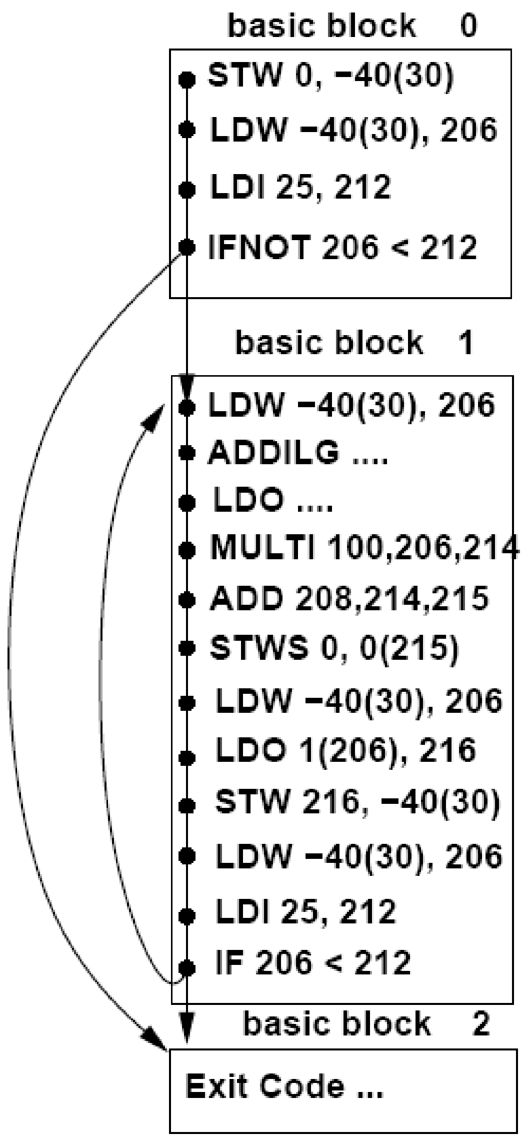
Representation: a Basic Block

- *Basic Block* = A Consecutive Sequence of Instructions (Statements)
  - A sequence of consecutive instructions in which flow of control enters at the beginning and leaves at the end without halt or possibility of branching except at the end
  - A *Basic Block Header*: Target instruction of a branch or a control join point

Optimizations within a basic block are *local* optimizations.

- How to build basic blocks?
  - First build a control flow graph of instructions, then identify basic block headers
  - Many optimizations work on a control flow graph of basic blocks

Building Basic Blocks for the Example Code



Local Optimizations

- Analysis and transformation performed within a basic block
  - No control flow information is considered
- Examples of local optimization
  - Load to copy optimization: when a store followed by a load
    - `store x @z; y=load @z -> store x @z; y=x`

  - Local common sub-expression elimination (CSE):
    - Analysis: some expression evaluated more than once in BB
    - Transformation: replace with single calculation (delete later ones if they have the same target register)
    - `x=y+z; …   w=y+z -> x=y+z; … w=x;`
    - `x=y+z; ..  x=y+z -> x=y+z; …`

Local Optimizations

- Examples of local optimization (continued)
  - Local constant folding or elimination
    - Analysis: expressions can be evaluated at compile-time
    - Transformation: replace by constant, compile-time value
    - `if (4>3)  -> if (true)`
  - Dead code elimination
    - When none uses the target of an instruction, it is dead code
    - `x=y+1; ….`
  - Copy propagation or constant propagation
    - `x=y;… z=x+100 -> x=y;… z=y+100`

One Thing to Note

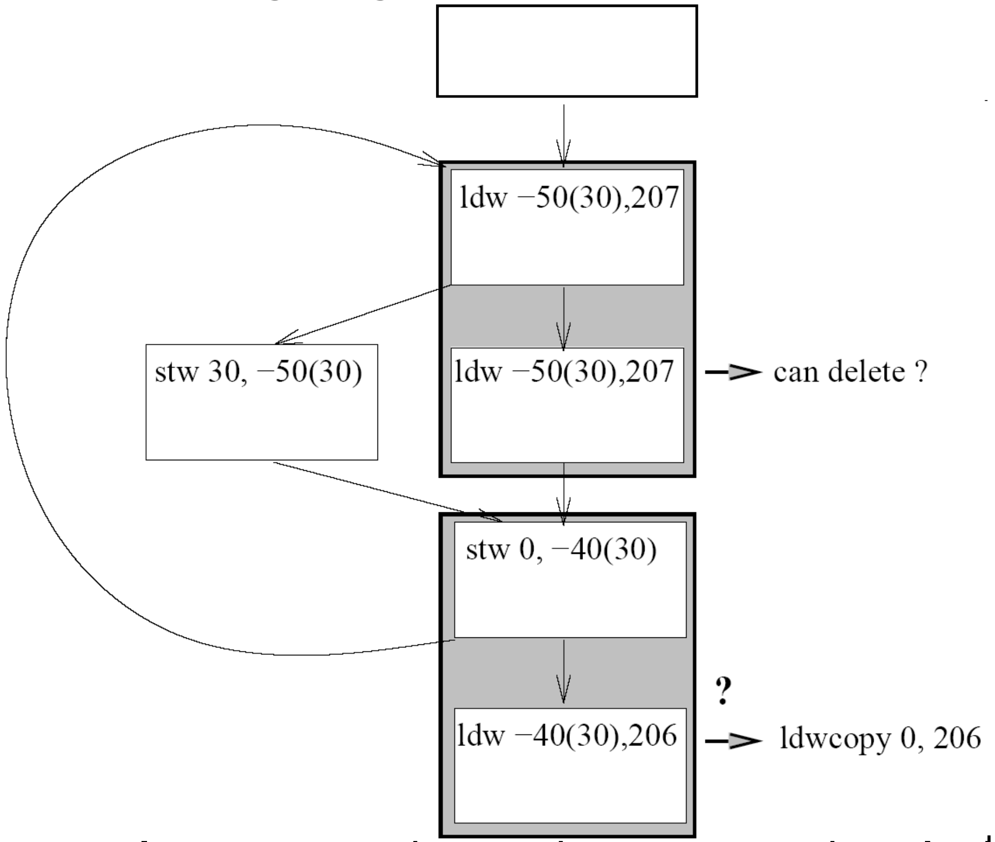
- Some of these optimizations do not seem to arise in practice if you "program very well", such as common subexpression elimination (CSE), dead code elimination, and copy propagation, constant propagation, constant folding, etc.
  - CSE: x=y+z; ... w=y+z -> x=y+z; ...w=x;
  - Copy propagation: x=y;... z=x+100 -> x=y;... z=y+100
  - Constant folding: if (4>3) -> if (true)
- The reality is that although you may be able to avoid explicit ones while you do your programming, the compiler still generate those opportunities
  - E.g., address computations
  - In our example, the same memory loads and computations are executed repeatedly

After Local Optimizations

```
          STW               0,-40(30)
          LDWCOPY           0,206                 :: <==      LDW        -40(30),206
          LDI               25,212                            load-copy opt.
$00000003
          LDW               -40(30),206
          ADDILG            LR'a-$global$,27,213
          LDO               RR'a-$global$(213),208
          MULTI             100,206,214
          ADD               208,214,215
          STWS              0,0(215)              :: Deleted  LDW        -40(30),206
          LDO               1(206),216                        CSE opt.
          STW               216,-40(30)
          LDWCOPY           216,206               :: <==      LDW        -40(30),206
          LDI               25,212                            load-copy opt.
          IF 206 < 212 GOTO $00000003
          NOP
$00000002
```

Extended Basic Block

- A chain of sequential basic blocks that has no incoming branches yet can have outgoing branches



- Can we apply same optimizations on extended basic blocks? Yes

Global Common Subexpression Elimination (CSE)

Redundant Definition Elimination

- Value numbering:
  - Hints for applying CSE: our code generator is expected to assign the same target pseudo register for the same right-hand-sides (RHS):
    - redundant expressions will always have same target register
- Compute available expressions across all paths
  - an expression x+y is available at a point *p* if every path to *p* evaluates x+y and after last evaluation prior to reaching *p*, there are no subsequent assignment to x or y
- Delete redundant expressions at *p*

After Global CSE Optimization

```
    STW       0,-40(30)        ; @i
    LDWCOPY   0,206            ; @i
    LDI       25,212
$00000003     :: what are available expressions here?
    ADDILG    LR'a-$global$,27,213  :: deleted LDW  -40(30),206
    LDO       RR'a-$global$(213),208
    MULTI     100,206,214
    ADD       208,214,215
    STWS      0,0(215)         ; @a[i][0]
    LDO       1(206),216
    STW       216,-40(30)      ; @i
    LDWCOPY   216,206          ; @i
    IF 206 < 212 GOTO $00000003  :: deleted LDI  25,212
    NOP
```

Definition, Use, and Live range

- Notation for accesses to locations
  - Definition means writing, Use means reading
    - E.g., there is one definition and two uses in x = y + z
- Live range: a set of definitions and uses
  - Which access the same location (register or memory)
  - For each use in the set, all definitions that might *reach* it should also be in the set
  - For each definition, all of its uses should also be in the set
  - $$\begin{tikzcd} \boxed{x = \text{load}} \arrow[dr] & & \boxed{x = x+1} \arrow[dl] \\ & \boxed{y = x/2} & \end{tikzcd}$$
  - There are two kinds of live ranges:
    - **Register live range** (web): register allocation unit
    - **Memory live range**: access the same memory location

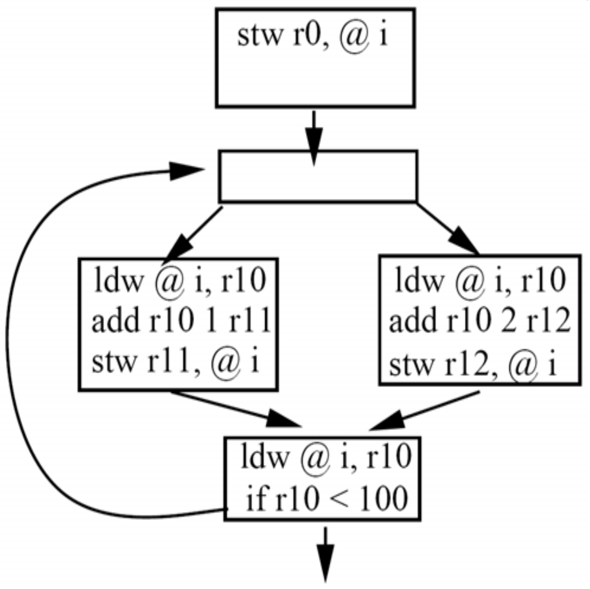
Promotion of Memory Operations

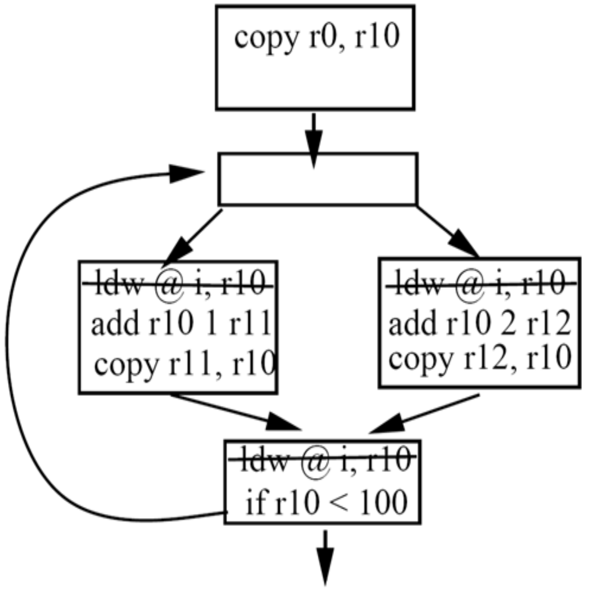
- Promote memory operation into register operations
  - E.g., `for (i=0; i<100; i++){}` : where would `i` be located?
- First build a memory live range, which is
  - a set of stores and loads accessing the same memory location
- A live range of memory can be promoted to register operations if certain conditions are met
  - Should be a singleton variable (no array, pointer, struct, etc)
  - Action: stores are promoted into copies and loads are deleted
  - The front-end provides some information on which loads and stores access the same location, or you can find it by yourself by analyzing the assembly code at this phase
    - By analyzing address derivations for memory instructions
    - `STW 0,-40(30):`
      - `Address derivation: -40 + SP (r30)`

Example

Example

```
int i =0;

Do {

   if (...)
      i = i + 1;
   else
      i = i + 2;
}  while (i < 100)
```





After Register Promotion

- There are two Memory Live Ranges

```
STW        0,-40(30)      ; @i    STW            216,-40(30)     ;@i
LDWCOPY 0,206             ; @i    LDWCOPY        216,206         ;@i
```

- After Optimization

```
        Copy        0,206
        LDI         25,212
        NOP
$00000003
        ADDILG      LR'a-$global$,27,213
        LDO         RR'a-$global$(213),208
        MULTI       100,206,214
        ADD         208,214,215
        STWS        0,0(215)
$00000001
        LDO         1(206),216
        COPY        216,206
        IF    206 < 212 GOTO $00000003
        NOP
```

Loop Transformation

Traditional loop optimizations

- Loop invariant code motion (LICM)
- Strength reduction
- Induction variable elimination

After Loop Optimization

```
        COPY        0,206
        LDI         25,212
        ADDILG      LR'a-$global$,27,213         :: ← loop invariant code
        LDO         RR'a-$global$(213),208       :: ← motion into loop header
        LDI         0,218                        :: ← newly inserted
        LDI         2500,219                     :: ← at the loop header
$00000003
        COPY        218,214                      :: ← MULTI       100,206,214
        ADD         208,214,215
        STWS        0,0(215)

        LDO         100(218),220                 :: ← LDO         1(206),216

        COPY        220,218                      :: ← COPY        216, 206
        IF 218 < 219 GOTO $00000003              :: ← IF 206 < 212 GOTO $00000003
```

Building Register Live Ranges (Webs)

- Live range of register definition and uses
- Unit for register allocation
- Helps for dead code elimination

After Building Register Live Ranges

- Two Dead Instructions

```
        COPY        0,206
        LDI         25,212
```

- After Optimization (each register number is replaced by web number)

```
        ADDILG      LR'a-$global$,27,66
        LDO         RR'a-$global$(66),65
        LDI         0,69
        LDI         2500,71
$00000003
        COPY        69,67
        ADD         65,67,68
        STWS        0,0(68)
        LDO         100(69),70
        COPY        70,69
        IF 71 < 69 GOTO $00000003
```

Instruction Scheduling

- Assume that the machine has two ALUs and can issue two instructions per clock cycle
  - We group independent instructions together

```
        ADDILG      LR'a-$global$,27,66
        LDO         RR'a-$global$(66),65
        LDI         0,69
        LDI         2500, 71
$00000003
        COPY        69,67
        LDO         100(69),70      :: ← code motion
        ADD         65,67,68
        COPY        70,69           :: ← code motion
        STWS        0,0(68)
        IF 71 < 69 GOTO $00000003
```

Register Allocation

- Allocate registers to live ranges (webs)
  - Two live ranges cannot be allocated to the same register if they *interfere* (e.g., `x` and `y` below)
  - `x = RHS1; y = RHS2;  z = x*10;  .. = y/31`
- Graph coloring register allocation
  - Interference graph
    - Nodes: live ranges, Edges: interference relationship
    - Color the graph with the given number of registers
    - NP-complete, so we use heuristics
  - If we do not have enough registers to keep all webs allocated, we need to spill some to memory
    - Add store right after defs , add load right before uses

Copy Elimination

- Two approaches
  - Copy propagation: make copy dead
    - `x=y; z=y+1 => x=y; z=x+1`
  - Copy coalescing: if the target live range and the source live range of a copy instruction does not interface, those live ranges are merged together and are allocated the same register; the copy is deleted since it has the form of “copy r1 r1”
    - Negative impact on the interference graph:
      - Merged nodes are hard to color due to more edges

After Register Allocation

- “COPY 69,67” is propagated and deleted
- “COPY 70,69” is coalesced as “COPY 69 69”
- Result after Copy Elimination (for non-scheduled version)

```
        ADDILG      LR’a-$global$,27,66
        LDO         RR’a-$global$(66),65
        LDI         0,69
        LDI         2500,71
$00000003
        ADD         65,69,68
        STWS        0,0(68)
        LDO         100(69),69
        IF 71 < 69 GOTO $00000003
```

- For scheduled version, coalescing is not possible

Wrap-up

- We skimmed thru some basic optimization techniques or optimization processes
- We will learn each optimization technique in detail throughout the semester
- Optimizations are possible after analyzing the code completely, though
- We will learn basic analysis techniques first