Computer Science 160
Translation of Programming Languages

Instructor: Christopher Kruegel

Code Optimization

Code Optimization

- What should we optimize?
  - improve running time
  - decrease space requirements
  - decrease power consumption

- Why does optimization work?
  - remove redundancies
  - no need for full generality
    - more specific instances of abstract constructs
  - leverage knowledge of target machine
    - pipelining, runtime of instructions, …

Program Analysis

- Scope of program analysis
  - within a basic block (local)
  - within a method (global or intra-procedural)
  - across methods (whole-program or inter-procedural)

- Analysis
  - control flow graph
    - dominators, loops, etc.
  - dataflow analysis
    - flow of values
  - static-single-assignment
    - transform programs such that each variable has a unique definition
  - alias analysis
    - pointer memory usage

Optimization Overview

- Classes of optimizations
  - machine independent or dependent

- Produce faster code
  - eliminate redundant (or useless) computation
    - common (sub)-expression elimination
    - constant folding
    - dead code elimination
  - move code
    - loop transformations
  - specialize code
  - instruction selection and scheduling
  - register allocation

Eliminate Redundant Computation

Original Block	Rewritten Block
a ← b + c	a ← b + c
b ← a - d	b ← a - d
c ← b + c	c ← b + c
d ← a - d	d ← b

Local Value Numbering

- Basic idea
  - assigns a distinct number to each value that the block computes
  - choose the numbers so that two expressions, *e1* and *e2*, have the same value number if and only *e1* and *e2* have provably equal values for all possible operands of the expressions

$$a^2 \leftarrow b^0 + c^1$$
$$b^4 \leftarrow a^2 - d^3$$
$$c^5 \leftarrow b^4 + c^1$$
$$d^4 \leftarrow a^2 - d^3$$

- Is value “4” still in the hash table?
- Yes, and it is associated with “b”
- Thus, can replace last operation with copy from “b”

Local Value Numbering


```
for i ← 0 to n-1, where the block has n operations   "Ti ← Li Opi Ri"
    1. get the value numbers for Li and Ri
    2. construct a hash key from Opi and the value numbers for Li and Ri
    3. if the hash key is already present in the table then
              replace operation i with a copy of the value into Ti and
              associate the value number with Ti
       else
              insert a new value number into the table at the hash key location
              record that new value number for Ti
```

Local Value Numbering

- Extended LVN algorithm
  - add support for commutative operations
  - add support for constant folding
  - add support for algebraic identities

- Algebraic identities
  - multiply variable with 0 or 1
  - add or subtract 0 from a variable
  - xor variable with itself
  - more possibilities …

Finding Uninitialized Variables

- Simple example of global data flow analysis
  - similar techniques used for other applications (e.g., finding unused/dead code)

- Approach based on liveness analysis
  - variable *v* is live at point *p* if and only if there exists a path in the CFG from *p* to a use of *v* along which *v* is not redefined

- LiveOut(*B*)
  - set that contains all the variables that are live on exit from block *B*
  - Given a LiveOut set for the CFG entry node n_0, each variable in LiveOut(n_0) has a potentially uninitialized use

Finding Uninitialized Variables

- Computing LiveOut set for block B
  - use the LiveOut sets of *B*'s successors in the CFG
  - use two sets UEVar(*B*) and VarKill(*B*) that encode facts how the code in B manipulates variables

- UEVar(B) - Upward Exposed Variable
  - this set contains all the variables that are **used** in B (without being defined before their uses)

- VarKill(B)
  - this set contains all the variables that are **defined** in B

- Since LiveOut(B) depends on LiveOut of other blocks that it is connected to, we can use an iterative fixed-point method

Finding Uninitialized Variables

LiveOut(n) for a block n based on successor nodes m

$$\text{LIVEOUT}(n) = \bigcup_{m \in succ(n)} (\text{UEVAR}(m) \cup (\text{LIVEOUT}(m) \cap \overline{\text{VARKILL}(m)}))$$

Variable *v* is live on entry to *m* under one of two conditions:

- it can be referenced in *m* before it is redefined in *m*
- it can be live on exit from *m* and pass unscathed through *m* because *m* does not redefine it

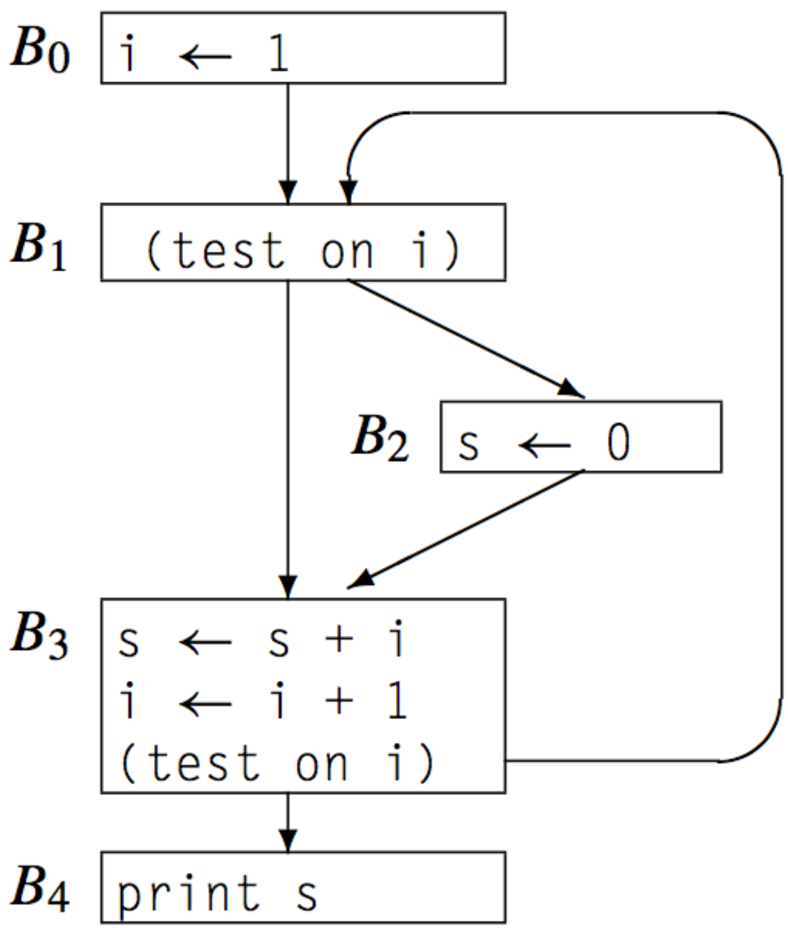
Finding Uninitialized Variables

- First, compute UEVar and VarKill for each block

- Second, apply iterative dataflow analysis

```
// assume CFG has N blocks
// numbered 0 to N-1

for i ← 0 to N-1
   LiveOut(i) ← Ø

changed ← true

while (changed)

   changed ← false

   for i ← 0 to N-1

      recompute LiveOut(i)

      if LiveOut(i) changed then
         changed ← true
```

Finding Uninitialized Variables



	UEVar	VarKill
B_0	∅	{i}
B_1	{i}	∅
B_2	∅	{s}
B_3	{s,i}	{s,i}
B_4	{s}	∅

Finding Uninitialized Variables

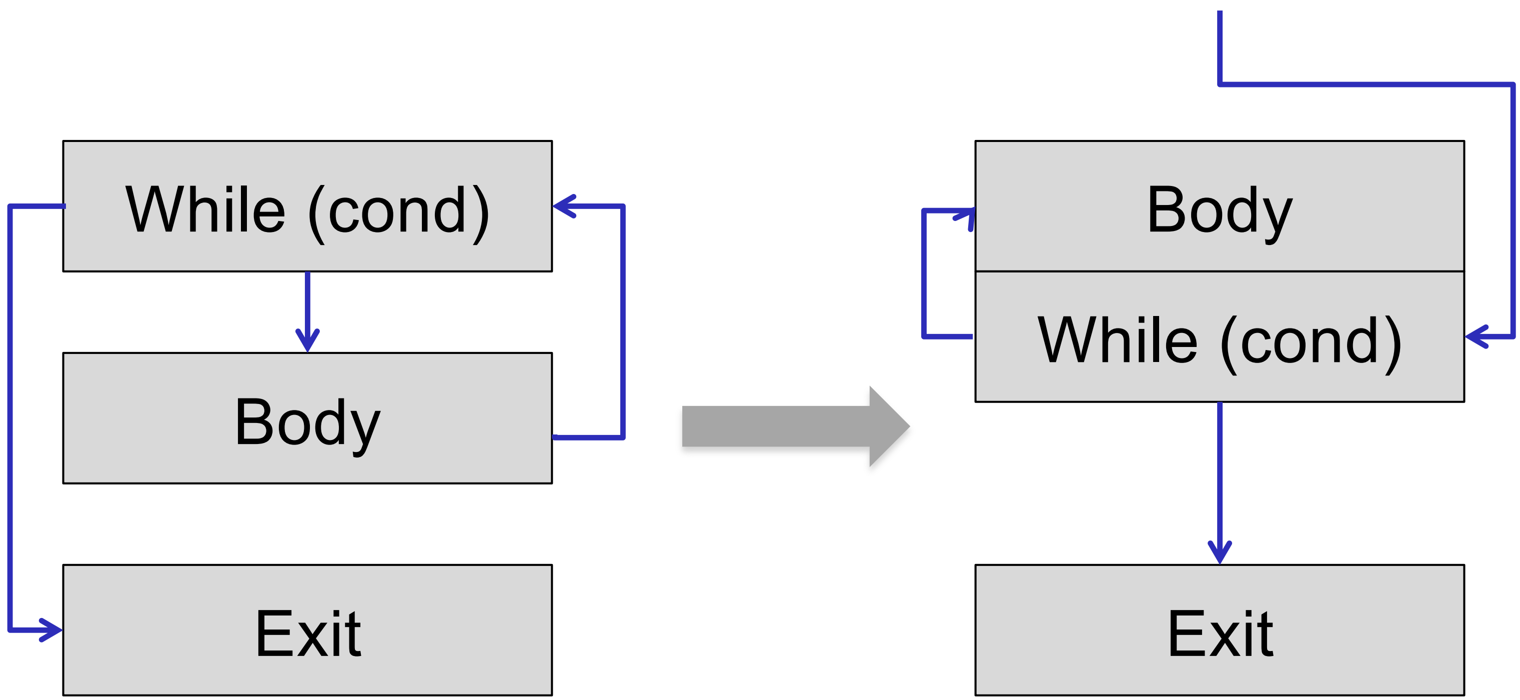
Iteration	LIVEOUT(*n*)				
	B_0	B_1	B_2	B_3	B_4
Initial	∅	∅	∅	∅	∅
1	{i}	{s,i}	{s,i}	{s,i}	∅
2	{s,i}	{s,i}	{s,i}	{s,i}	∅
3	{s,i}	{s,i}	{s,i}	{s,i}	∅

Moving Code

- Loop unrolling
  - take the body of a loop and make N consecutive copies
  - saves overhead of jumping back to loop head and evaluate loop condition
  - need to be careful to make sure that N copies of loop body need to be executed
  - short prologue loop that peels off enough iterations to ensure that the unrolled loop processes an integral multiple of N iterations
- Invariant code moving
  - invariant computations do not change with each loop iteration
  - compute value once outside of the loop, then use result inside loop body

Moving Code

- Code placement
  - change code layout to reduce number of jumps
  - convert frequently-taken edges into fall through operations



Moving Code

- Function inlining
  - procedure linkage creates overhead
  - function body might be very small (e.g., string copy)
  - copy function body into caller, save overhead

Instruction Selection

- Peephole optimization
  - use a small sliding window over sequence of instructions
  - replace individual instructions with faster alternatives
  - replace common sequences with faster alternatives

- Individual instructions
  - use shift instead of multiply (by power of 2)
  - use address computation logic instead of arithmetic

```
lea (%rdi,%rdi), %eax
```

instead of

```
shl $0x1, %edi
mov %edi, %eax
```

Instruction Selection

- Store followed by load

```
storeAI r1       ⇒ rarp,8        ⇒    storeAI r1 ⇒ rarp,8
loadAI  rarp,8   ⇒ r15                i2i     r1 ⇒ r15
```

$$\begin{array}{lll} \texttt{storeAI} & r_1 & \Rightarrow r_{arp},8 \\ \texttt{loadAI} & r_{arp},8 & \Rightarrow r_{15} \end{array} \quad \Rightarrow \quad \begin{array}{lll} \texttt{storeAI} & r_1 & \Rightarrow r_{arp},8 \\ \texttt{i2i} & r_1 & \Rightarrow r_{15} \end{array}$$

- Double jump

$$\begin{array}{rl} & \texttt{jumpI} \rightarrow l_{10} \\ l_{10}: & \texttt{jumpI} \rightarrow l_{11} \end{array} \quad \Rightarrow \quad \begin{array}{rl} & \texttt{jumpI} \rightarrow l_{11} \\ l_{10}: & \texttt{jumpI} \rightarrow l_{11} \end{array}$$

- More complex algorithms possible, which work on AST tree patters

Instruction Scheduling

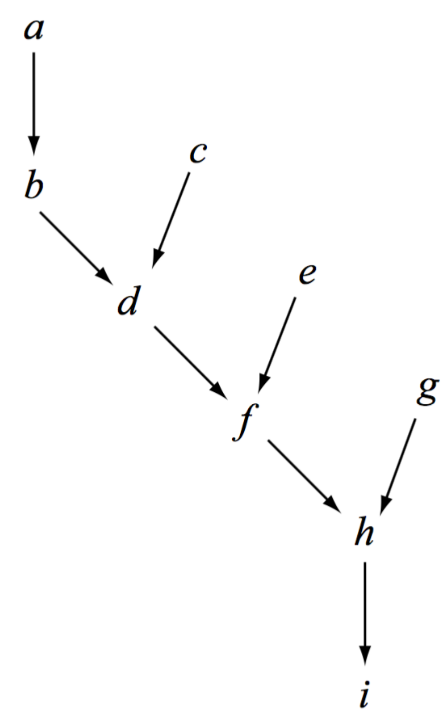
- Exploit multiple functional units of CPU
  - make sure that all units are busy at the same time
    - integer and floating point units
    - pipeline units
- Move *independent* instructions around
- Example
  - load/store = 3 cycles, multiply = 2 cycles, rest = 1 cycle

$$a \leftarrow a \times 2 \times b \times c \times d$$

Instruction Scheduling

a:	loadAI	r_{arp}, @a	⇒	r_1
b:	add	r_1, r_1	⇒	r_1
c:	loadAI	r_{arp}, @b	⇒	r_2
d:	mult	r_1, r_2	⇒	r_1
e:	loadAI	r_{arp}, @c	⇒	r_3
f:	mult	r_1, r_2	⇒	r_1
g:	loadAI	r_{arp}, @d	⇒	r_2
h:	mult	r_1, r_2	⇒	r_1
i:	storeAI	r_1	⇒	r_{arp}, @a

(a) Example Code



(b) Its Dependence Graph

Instruction Scheduling

Start	Operations			
1	loadAI	r_{arp}, @a	⇒	r_1
4	add	r_1, r_1	⇒	r_1
5	loadAI	r_{arp}, @b	⇒	r_2
8	mult	r_1, r_2	⇒	r_1
10	loadAI	r_{arp}, @c	⇒	r_2
13	mult	r_1, r_2	⇒	r_1
15	loadAI	r_{arp}, @d	⇒	r_2
18	mult	r_1, r_2	⇒	r_1
20	storeAI	r_1	⇒	r_{arp}, @a

(a) Original Code

Start	Operations			
1	loadAI	r_{arp}, @a	⇒	r_1
2	loadAI	r_{arp}, @b	⇒	r_2
3	loadAI	r_{arp}, @c	⇒	r_3
4	add	r_1, r_1	⇒	r_1
5	mult	r_1, r_2	⇒	r_1
6	loadAI	r_{arp}, @d	⇒	r_2
7	mult	r_1, r_3	⇒	r_1
9	mult	r_1, r_2	⇒	r_1
11	storeAI	r_1	⇒	r_{arp}, @a

(b) Scheduled Code