Software Programmable DSP Platform Analysis

Episode 6, Wednesday 4 May 2005, Ingredients

Datafbw Analysis

Reaching Definitions
Constant Propagation, Copy Propagation
Available Expressions, Reaching Expressions
Common Subexpression Elimination
Dead Code Elimination

Loop Optimizations

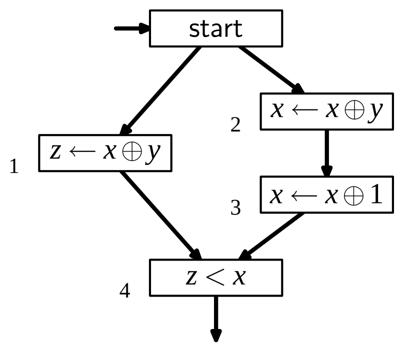
What is a loop? Loop Dominators.
Loop Invariants and Hoisting
Induction Variables. Strength Reduction
Loop Unrolling

Reaching Definitions

- An unambigous definition d of t is an assignment $t \leftarrow a \oplus b$ or $t \leftarrow M[a]$.
- A definition d reaches a statement u if there is a path of control edges leading from d to u that does not pass through any other definitions of t.

Reaching Definitions

Example



- Definition 1 reaches 4
- Definition 2 does **not** reach 4, because all paths from 2 to 4 path through 3 that kills 2.

Reaching Definitions: gen/kill sets

Defs(t): set of all definitions of temporary t.

statement s	$gen[s]$	$kill[s]$
$d : t \leftarrow b \oplus c$	$\{d\}$	$defs(t) - \{d\}$
$d : t \leftarrow M[b]$	$\{d\}$	$defs(t) - \{d\}$
$M[a] \leftarrow b$	$\{\}$	$\{\}$
if $a\ R\ b$ goto L_1 else goto L_2	$\{\}$	$\{\}$
goto L	$\{\}$	$\{\}$
$L :$	$\{\}$	$\{\}$
$f(a_1, \ldots, a_n)$	$\{\}$	$\{\}$
$d : t \leftarrow f(a_1, \ldots a_n)$	$\{d\}$	$defs(t) - \{d\}$

Calculating Reaching Definitions

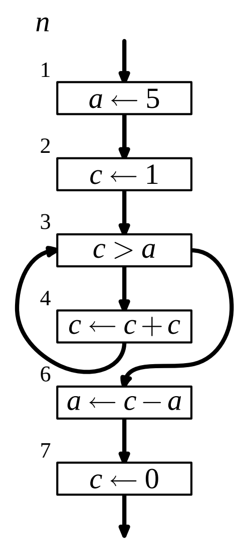
Initialize *in*[n] and *out*[n] to be empty sets.

Apply following equations until a fixpoint is reached:

$$in[n] = \bigcup_{p \in pred[n]} out[p]$$

$$out[n] = gen[n] \cup (in[n] - kill[n])$$

Gen and *kill* sets are defined on previous slide.



$a \leftarrow 5$

$c \leftarrow 1$

$L_1:$ if $c > 0$ goto L_2

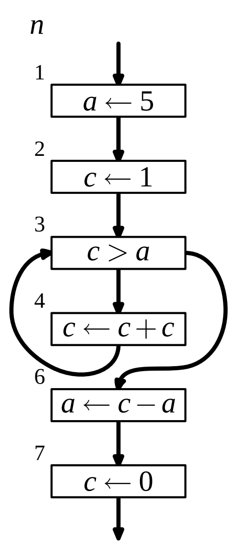
$c \leftarrow c + c$

goto L_1

$L_2:$ $a \leftarrow c - a$

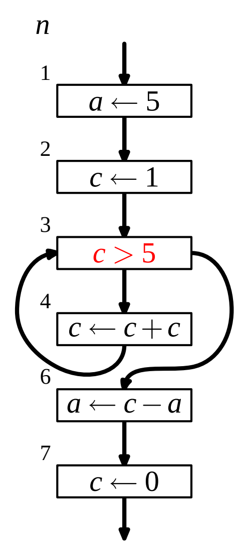
$c \leftarrow 0$

(source: Appel Program 17.3 p.389)



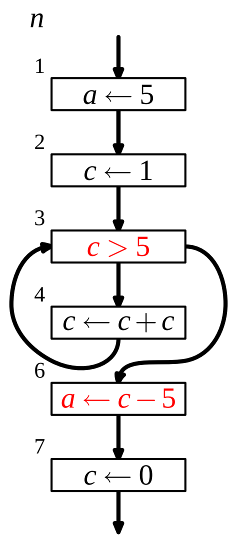
$gen[n]$	$kill[n]$	$in[n]$	$out[n]$
1	6	$\emptyset$	1
2	4, 7	1	1, 2
$\emptyset$	$\emptyset$	1, 2, 4	1, 2, 4
4	2, 7	1, 2, 4	1, 4
6	1	1, 2, 4	2, 4, 6
7	2, 4	2, 4, 6	6, 7

$in[n] = \bigcup_{p \in pred[n]} out[p]$ $\quad$ $out[n] = gen[n] \cup (in[n] - kill[n])$



$gen[n]$	$kill[n]$	$in[n]$	$out[n]$
1	6	$\emptyset$	1
2	4, 7	1	1, 2
$\emptyset$	$\emptyset$	1, 2, 4	1, 2, 4
4	2, 7	1, 2, 4	1, 4
6	1	1, 2, 4	2, 4, 6
7	2, 4	2, 4, 6	6, 7

The only def of a reaching 3 is 1, so can rewrite 3 to $c > 5$



$gen[n]$	$kill[n]$	$in[n]$	$out[n]$
1	6	$\emptyset$	1
2	4,7	1	1,2
$\emptyset$	$\emptyset$	1,2,4	1,2,4
4	2,7	1,2,4	1,4
6	1	1,2,4	2,4,6
7	2,4	2,4,6	6,7

Similarly with 6: $a \leftarrow c - 5$, but 4 cannot be rewritten.

Constant Propagation

- Let d be a statement: $t \leftarrow c$, where c is constant.
- Let n be another statement such as $y \leftarrow t \oplus x$.
- If d is the only definition of t reaching n,
- It is safe to rewrite n as $y \leftarrow c \oplus x$.

Copy Propagation

- Copy propagation is like constant propagation, but instead of constant c a variable is used.
- Let $d : t \leftarrow z$ be a statement.
- Let $n : y \leftarrow t \oplus x$ be a statement using t.
- If d is the only definition of t reaching n and there is no definition of z on **any** path from d to n then we can rewrite: $n : y \leftarrow z \oplus x$.
- This may remove t entirely from the program.
- Mind the “any” requirement: this includes paths that cross n more than once (for example loops), so the redefinition after n can also prevent copy propagation.

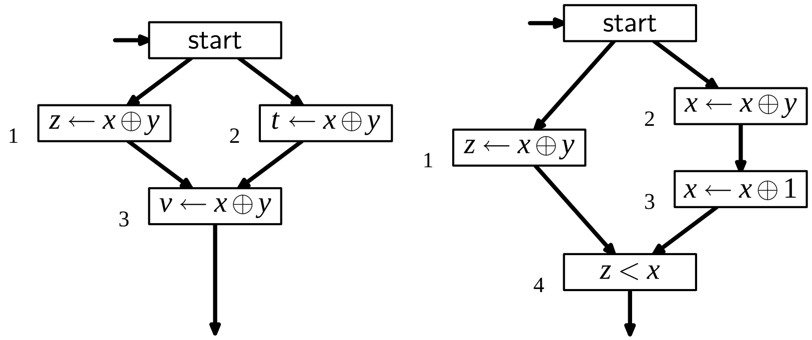
Available Expressions

An expression $x \oplus y$ is available at a node n in the flow graph if:

- on every path from the entry node to n, $x \oplus y$ is computed at least once,
- and there are no definitions of x or y since the most recent occurrence of $x \oplus y$ on that path.

Available Expressions

Example



$x \oplus y$ is available in 3 $\quad$ $x \oplus y$ is not available in 4

Computing Available Expressions

statement s	$gen[s]$	$kill[s]$
$d : t \leftarrow b \oplus c$	$\{b \oplus c\} - kill[s]$	all containing t
$d : t \leftarrow M[b]$	$\{M[b] - kill[s]\}$	all containing t
$M[a] \leftarrow b$	$\{\}$	all $M[x]$
if $a\ R\ b$ goto L_1 else goto L_2	$\{\}$	$\{\}$

$$in[n] = \bigcap_{p \in pred[n]} out[p] \quad \text{if } n \text{ is not entry}$$

$$out[n] = gen[n] \cup (in[n] - kill[n])$$

Initialize $in[\text{entry}]$ to empty set, initialize all other sets to contain all expressions of the program. Iterate until (the greatest) fixpoint is reached.

Reaching Expressions

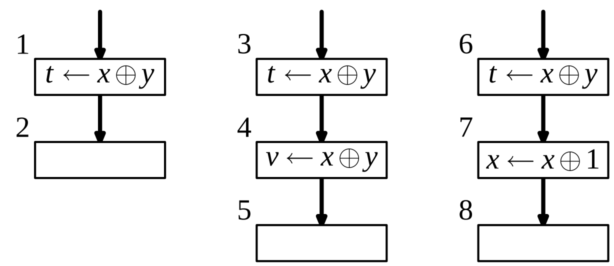
Reaching expressions, are much like reaching definitions. Expression $t \leftarrow x \oplus y$ in node s reaches a node n if:

- there is a path from s to n that
- does not go through any assignment to x or y,
- or through any other computation of $x \oplus y$.

Reaching expressions are characterized by their own *gen*, *kill* and *in*, *out* equations as for previous flow analyses. They are computed very much like previous examples.

Reaching Expressions

Example



- $1 : x \oplus y$ is reaching 2.
- $3 : x \oplus y$ is not reaching 5, as 4 recomputes it.
- But 4 is reaching 5.
- $6 : x \oplus y$ is not reaching 8, as 7 kills x.

Common Subexpression Elimination

If expression $x \oplus y$ is **available** at $s : t \leftarrow x \oplus y$ then the computation of $x \oplus y$ within s can be eliminated:

- Compute expressions $x \oplus y$ **reaching** s.
- Introduce a new (fresh) temporary w.
- For each such reaching node $n : v \leftarrow x \oplus y$ rewrite n to be:

$$n : w \leftarrow x \oplus y$$
$$n' : v \leftarrow w$$

- Modify s to use w: $s : t \leftarrow w$

Common Subexpression Elimination

Example

```
x = a + b + c;
y = a + b + d;
```

compiles to	CSE	copy prop.	reg. alloc.
$x \leftarrow a+b$	$w \leftarrow a+b$	$w \leftarrow a+b$	$y \leftarrow a+b$
$x \leftarrow x+c$	$x \leftarrow w$	$x \leftarrow w+c$	$x \leftarrow y+c$
$y \leftarrow a+b$	$x \leftarrow x+c$	$y \leftarrow w+d$	$y \leftarrow y+d$
$y \leftarrow y+d$	$y \leftarrow w$		
	$y \leftarrow y+d$		

Dead Code Elimination

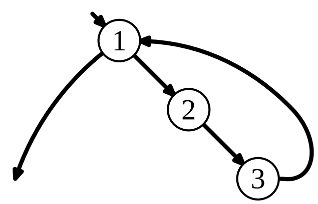
If $s : a \leftarrow b \oplus c$ (or $s : a \leftarrow M[x]$) and a is **not** live-out of s then the instruction can be eliminated.

```
x = a + b + c;
y = a + b + d;
return y;
```

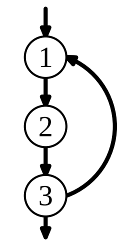
compiles to	*live-in*[s]	*live-out*[s]	DCE
$x \leftarrow a+b$	a,b	a,b,c,x	$x \leftarrow a+b$
$x \leftarrow x+c$	a,b,c,x	a,b	
$y \leftarrow a+b$	a,b	d,y	$y \leftarrow a+b$
$y \leftarrow y+d$	d,y	y	$y \leftarrow y+d$

Loops

Examples



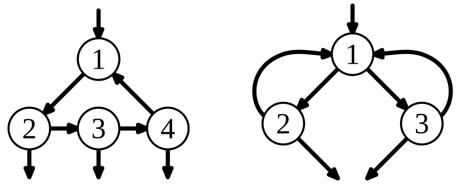
a *while-do* loop



a *do-while* loop
(also known as *repeat-until*)

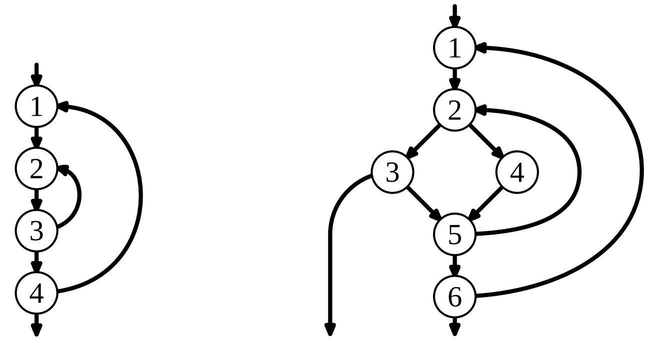
Loops with Multiple Exit Points

Examples



Nested Loops

Examples



nested “do-while” loops nested loops with “break”

Many loop structures cry for an abstract definition.

Loops Precisely Defined

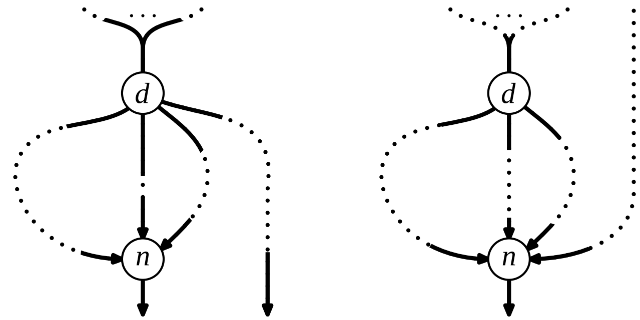
A set of nodes S constitutes a loop if:

- S contains a header node h such that
- from any node in S there is a path leading to h.
- There are not any edges from nodes outside S to nodes in S other than h.

All loops on previous slides are loops according to this definition.

Loop Dominator

Node d dominates node n if every path of directed edges from s_0 to n must go through d. Every node dominates itself.



d dominates n d does not dominate n

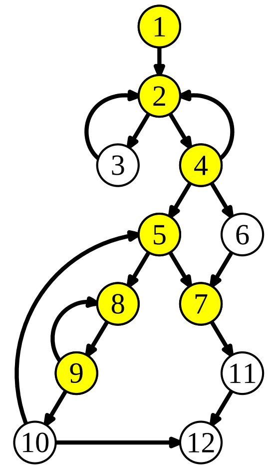
Computing Dominators

Dominators are computed by iterating the following equations over the nodes of the flow graph:

$$D[s_0] = \{s_0\}$$
$$D[n] = \{n\} \cup \Big(\bigcap_{p \in pred[n]} D[p] \Big) \text{ for } n \neq s_0$$

Initially each $D[n]$ should contain all nodes of the graph (except $D[n_0]$).

Computing Dominators
Example



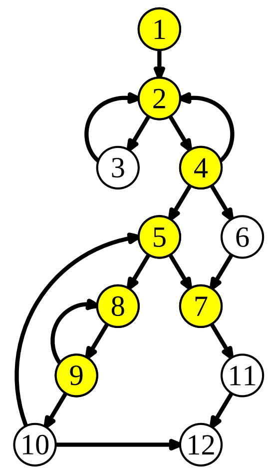
$D[1]$	1	immediate dominators
$D[2]$	1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12	
$D[3]$	1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12	
$D[4]$	1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12	
$D[5]$	1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12	
$D[6]$	1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12	
$D[7]$	1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12	
$D[8]$	1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12	
$D[9]$	1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12	
$D[10]$	1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12	
$D[11]$	1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12	
$D[12]$	1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12	

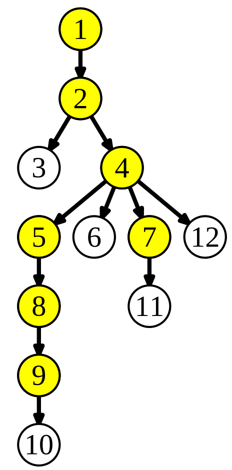
Dominator Tree

Every node n has at most one *immediate dominator* $idom[n]$ such that:

- $idom(n)$ is not the same node as n.
- $idom(n)$ dominates n.
- $idom(n)$ does not dominate any other dominator of n.

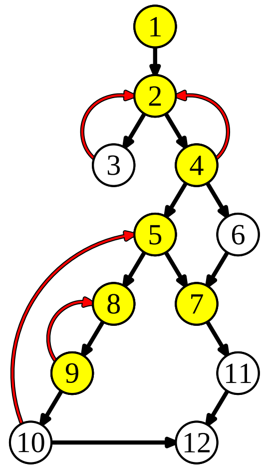
Dominator Tree
Example





Back Edge

An edge from node n to h, where h dominates n.

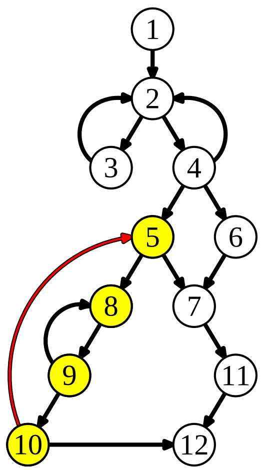


Loop Definition Revisitted

- The natural loop of a back-edge $n \rightarrow h$ is the set of nodes x, such that h dominates x and there is a path from x to n not containing h.
- Node h is the header of the loop.
- This definition allows automatic detection of loops.

Natural Loop

Example



- Natural loop of back edge $10 \rightarrow 5$
- Includes nodes: $5, 8, 9, 10$
- Contains the loop $8, 9$ nested.

Loop Invariant

The definition $d : t \leftarrow a_1 \oplus a_2$ is a loop invariant within loop L if $d \in L$ and for each operand a_i:

- a_i is constant,
- or all the definitions of a_i reaching d are outside the loop,
- or only one definition of a_i reaches d and that definition is loop invariant.

Loop invariant computations can sometimes be moved out (*hoisted*) out of the loop, speeding up the execution.

Can We Hoist $t \rightarrow a \oplus b$?

$L_0 : t \leftarrow 0$
$L_1 : i \leftarrow i+1$
$\quad t \leftarrow a \oplus b$
$\quad M[i] \leftarrow t$
$\quad$ if $i < N$ goto L_1
$L_2 : x \leftarrow t$

- $t \leftarrow a \oplus b$ is loop invariant.
- Moving it before the loop would not change the behaviour of our program.
- It would make the program faster.
- So the answer is: **YES!**

Hoisted.

$L_0 : t \leftarrow 0$
$\quad t \leftarrow a \oplus b$
$L_1 : i \leftarrow i+1$
$\quad M[i] \leftarrow t$
$\quad$ if $i < N$ goto L_1
$L_2 : x \leftarrow t$

- $t \leftarrow a \oplus b$ is loop invariant.
- Moving it before the loop would not change the behaviour of our program.
- It would make the program faster.
- So the answer is: **YES!**

Can We Hoist $t \rightarrow a \oplus b$?

$L_0 : t \leftarrow 0$
$L_1 :$ if $i \geq N$ goto L_2
$\quad i \leftarrow i+1$
$\quad t \leftarrow a \oplus b$
$\quad M[i] \leftarrow t$
$\quad$ goto L_1
$L_2 : x \leftarrow t$

- The original program does not always execute $t \leftarrow a \oplus b$.
- Hoisting would execute it unconditionally always at least once.
- Leading to a wrong value of x if no loop iterations are executed.
- So the answer is: **NO!**

Minimum trip count pragma might help though...

Can We Hoist $t \rightarrow a \oplus b$?

$L_0 : t \leftarrow 0$
$L_1 : i \leftarrow i+1$
$\quad t \leftarrow a \oplus b$
$\quad M[i] \leftarrow t$
$\quad t \leftarrow 0$
$\quad M[j] \leftarrow t$
$\quad$ if $i < N$ goto L_1
$L_2 :$

- The original program has more than one def of t.
- Hoisting would change the interleaving of the assignments.
- So the answer is: **NO!**

Can We Hoist $t \rightarrow a \oplus b$?

$L_0 : t \leftarrow 0$
$L_1 : M[j] \leftarrow t$
$\quad i \leftarrow i + 1$
$\quad t \leftarrow a \oplus b$
$\quad M[i] \leftarrow t$
$\quad$ if $i < N$ goto L_1
$L_2 : x \leftarrow t$

- t is used before the loop invariant definition.
- So the answer is: **NO!**

Sufficient Conditions for Hoisting

Loop invariant computation $d : t \leftarrow a \oplus b$ can be hoisted if:

- d dominates all loop exits at which t is live-out.
- There is only one def of t in the loop.
- t is not live-out of the loop preheader.

Basic Induction Variable

$\quad s \leftarrow 0$
$\quad i \leftarrow 0$
$L_1 :$ if $i \geq n$ goto L_2
$\quad j \leftarrow i \cdot 4$
$\quad k \leftarrow j + a$
$\quad x \leftarrow M[k]$
$\quad s \leftarrow s + x$
$\quad i + 1$
$\quad$ goto L_1
$L_2 :$

The variable i is a basic induction variable in a loop L with header node h if the only definitions of i within L are of the form $i \leftarrow i + c$ or $i \leftarrow i - c$, where c is loop invariant.

Derived Induction Variable

$\quad s \leftarrow 0$
$\quad i \leftarrow 0$
$L_1 :$ if $i \geq n$ goto L_2
$\quad j \leftarrow i \cdot 4$
$\quad k \leftarrow j + a$
$\quad x \leftarrow M[k]$
$\quad s \leftarrow s + x$
$\quad i \leftarrow i + 1$
$\quad$ goto L_1
$L_2 :$

k is a derived induction variable if L contains only one definition of k, $k \leftarrow j \cdot c$ or $k \leftarrow j + d$, where j is an induction variable and c, d are invariant.

If j is an induction variable derived from i then the only def of j that reaches k is the one in the loop, and there is no def of i between the def of j and the def of k.

Strength Reduction

- On many machines multiplication is more expensive than addition (including C67xx).
- a definition of derived variable like $j \leftarrow i \cdot c$ can be replaced with addition.

$$
\begin{array}{ll}
 & s \leftarrow 0 \\
 & i \leftarrow 0 \\
L_1 & \text{:if } i \geq n \text{ goto } L_2 \\
 & j \leftarrow i \cdot 4 \\
 & k \leftarrow j + a \\
 & x \leftarrow M[k] \\
 & s \leftarrow s + x \\
 & i \leftarrow i + 1 \\
 & \text{goto } L_1 \\
L_2 & :
\end{array}
$$

$$
\begin{array}{ll}
 & s \leftarrow 0 \\
 & i \leftarrow 0 \\
 & j' \leftarrow 0 \;\; k' \leftarrow a \\
L_1 & \text{:if } i \geq n \text{ goto } L_2 \\
 & j \leftarrow j' \\
 & k \leftarrow k' \\
 & x \leftarrow M[k] \\
 & s \leftarrow s + x \\
 & i \leftarrow i + 1 \;\; j' \leftarrow j' + 4 \;\; k' \leftarrow k' + 4 \\
 & \text{goto } L_1 \\
L_2 & :
\end{array}
$$

Dead code elimination will remove $j \leftarrow j'$.
Elimination of useless variables (Appel p.424) eliminates $j' \leftarrow j' + 4$ too.

Loop Unrolling

Some loops have such a small body that most of the time is psent incrementing the loop counter variable and testing the loop-exit condition.

We can make these loops more efficient by unrolling them, putting two or more copies of the loop body in a row.

Let loop L have header h and back edges $s : s_i \rightarrow h$.
We unroll L as follows:

- Copy the nodes to make a loop L' with header h' and back edges $s_i' \rightarrow h'$.
- Change all the back edges in L from $s_i \rightarrow h$ to $s_i \rightarrow h'$.
- Change all the back edges in L' from $s_i' \rightarrow h'$ to $s_i' \rightarrow h$.

Useless Loop Unrolling

Example

$$\begin{aligned}
L_1 :& x \leftarrow M[i] \\
& s \leftarrow s + x \\
& i \leftarrow i + 4 \\
& \text{if } i < n \text{ goto } L_1 \text{ else } L_2 \\
L_2 :&
\end{aligned}$$

$$\begin{aligned}
L_1 :& x \leftarrow M[i] \\
& s \leftarrow s + x \\
& i \leftarrow i + 4 \\
& \text{if } i < n \text{ goto } L_1' \text{ else } L_2 \\
L_1' :& x \leftarrow M[i] \\
& s \leftarrow s + x \\
& i \leftarrow i + 4 \\
& \text{if } i < n \text{ goto } L_1 \text{ else } L_2
\end{aligned}$$

Useful Loop Unrolling

Example

Use information about induction vars to combine increments. This works for even number of iterations:

$$\begin{aligned}
L_1 :& x \leftarrow M[i] \\
& s \leftarrow s + x \\
& i \leftarrow i + 4 \\
& \text{if } i < n \text{ goto } L_1 \text{ else } L_2 \\
L_2 :&
\end{aligned}$$

$$\begin{aligned}
L_1 :& x \leftarrow M[i] \\
& s \leftarrow s + x \\
& x \leftarrow M[i+4] \\
& s \leftarrow s + x \\
& i \leftarrow i + 8 \\
& \text{if } i < n \text{ goto } L_1 \text{ else } L_2 \\
L_2 :&
\end{aligned}$$

General version in Appel p.430.

Some Optimizations of `cl6x`

- **-O0** register allocation, loop rotation, dead code elimination, keyword driven inlining
- **-O1** copy/constant propagation, useless variable elimination, common subexpression elimination
- **-O2** software pipelining, loop optimizations, global common subexpression elimination, global useless variable elimination, strength reduction with arrays and pointers, loop unrolling,
- **-O3** unsued function elimination, automatic inlining, (limited) partial evaluation,

We have now covered most of these optimizations!