CS 2210: Optimization

Jonathan Misurda
jmisurda@cs.pitt.edu

A "Bad" Name

Optimization is the process by which we turn a program into a better one, for some definition of better.

This is impossible in the general case.

For instance, a *fully optimizing compiler* for size must be able to recognize all sequences of code that are infinite loops with no output, so that it can replace it with a one-instruction infinite loop.

This means we must solve the halting problem.

So, what can we do instead?

Optimization

An optimizing compiler transforms *P* into a program *P'* that always has the same input/output behavior as *P*, and might be smaller or faster.

Optimizations are code or data transformations which typically result in improved performance, memory usage, power consumption, etc.

Optimizations applied naively may sometimes result in code that performs *worse*.

We saw one potential optimization before, ***loop interchange***, where we decide to change the order of loop headers in order to get better cache locality. However, this may result in worse overall performance if the resulting code must do more work and the arrays were small enough to fit in the cache regardless of the order.

Register Allocation

Register allocation is also an optimization as we previously discussed.

On register-register machines, we avoid the cost of memory accesses anytime we can keep the result of one computation available in a register to be used as an operand to a subsequent instruction.

Good register allocators also do coalescing which eliminates move instructions, making the code smaller and faster.

Dataflow Analyses

Reaching Definitions

Does a particular value *t* directly affect the value of *t* at another point in the program?

Given an *unambiguous* definition *d*,

```
t ← a ⊕ b
```

or

```
t ← M[a]
```

we say that *d reaches* a statement *u* in the program if there is some path in the CFG from *d* to *u* that does not contain any unambiguous definition of *t*.

An *ambiguous* definition is a statement that might or might not assign a value to *t*, such as a call with pointer parameters or globals. Decaf will not register allocate these, and so we can ignore the issue.

Reaching Definitions

We label every move statement with a definition ID, and we manipulate sets of definition IDs.

We say that the statement

d_1: t ← x ⊕ y

generates the definition d_1, because no matter what other definitions reach the beginning of this statement, we know that d_1 reaches the end of it.

This statement **kills** any other definition of *t*, because no matter what other definitions of *t* reach the beginning of the statement, they cannot directly affect the value of *t* after this statement.

Reaching Definitions

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

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

This looks familiar, but is the reverse of our liveness calculations.

We solve it using iteration the same as with liveness.

Available Expressions

An expression:

x ⊕ y

is **available** at a node *n* in the flow graph if, on every path from the entry node of the graph to node *n*, `x ⊕ y` is computed at least once *and* there are no definitions of *x* or *y* since the most recent occurrence of `x ⊕ y` on that path.

Any node that computes `x ⊕ y` *generates* `{x ⊕ y}`, and any definition of *x* or *y kills* `{x ⊕ y}`.

A *store* instruction (`M[a] ← b`) might modify any memory location, so it kills any *fetch* expression (`M[x]`). If we were sure that *a* = *x*, we could be less conservative, and say that `M[a] ← b` does not kill `M[x]`. This is called **alias analysis.**

Available Expressions

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

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

Compute this by iteration.

Define the *in* set of the start node as empty, and initialize all other sets to *full* (the set of all expressions), not empty.

Intersection makes sets *smaller*, not bigger.

Reaching Expressions

We say that an expression:

t ← x ⊕ y

(in node *s* of the flow graph) reaches 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 computation of *x* ⊕ *y*.

Dataflow Optimizations

Dead-code Elimination

If there is an IR instruction

```
s : a ← b ⊕ c
```

or

```
s : a ← M[x]
```

such that *a* is not *live-out* of *s*, then the instruction can be deleted.

Some instructions have implicit side effects such as raising an exception on overflow or division by zero. The deletion of those instructions will change the behavior of the program.

The optimizer shouldn't always do this. Optimizations that eliminate even seemingly harmless runtime behavior cause unpredictable behavior of the program. A program debugged with optimizations on may fail with them disabled.

Dead Code Elim in SSA

SSA makes dead-code analysis quick and easy.

A variable is live at its site of definition iff there are uses of this variable:

- there can be no other definition of the same variable (SSA!), and
- the definition of a variable dominates every use – so there must be a path from definition to us

An iterative algorithm for deleting dead code:

```
while there is some variable v with no uses
      and the statement that defines v has no other side effects
          do delete the statement that defines v
```

Dead Code Elim in SSA

```
W ← a list of all variables in the SSA program
while W is not empty
    remove some variable v from W
    if v's list of uses is empty
        let S be v's statement of definition
        if S has no side effects other than the assignment to v
            delete S from the program
            for each variable x_i used by S
                delete S from the list of uses of x_i
                W ← W ∪ {x_i}
```

Constant Propagation

Suppose we have a statement d:

```
t ← c
```

where *c* is a constant,

and another statement *n* that uses *t*:

```
y ← t ⊕ x
```

We know that *t* is constant in *n* if *d* reaches *n*, and no other definitions of *t* reach *n*.

In this case, we can rewrite *n* as:

```
y ← c ⊕ x
```

Constant Propagation in SSA

Any *φ*-function of the form $v \leftarrow \varphi(c_1, c_2, \ldots, c_n)$, where all the c_i are equal, can be replaced by $v \leftarrow c$.

```
W ← a list of all statements in the SSA program
while W is not empty
    remove some statement S from W
    if S is v ← φ(c, c, . . . , c) for some constant c
        replace S by v ← c
    if S is v ← c for some constant c
        delete S from the program
        for each statement T that uses v
            substitute c for v in T
            W ← W ∪ {T}
```

Copy Propagation

This is like constant propagation, but instead of a constant *c* we have a variable *z*.

Suppose we have a statement:

```
d: t ← z
```

and another statement *n* that uses *t*, such as:

```
n: y ← t ⊕ x
```

If *d* reaches *n*, and no other definition of *t* reaches *n*, and there is no definition of *z* on any path from *d* to *n* (including a path that goes through *n* one or more times), then we can rewrite *n* as:

```
n: y ← z ⊕ x
```

Copy Propagation in SSA

A single-argument *φ*-function $x \leftarrow \varphi(y)$ or a copy assignment $x \leftarrow y$ can be deleted, and *y* substituted for every use of *x*.

Add this to our worklist algorithm.

Constant folding

Constant folding

If we have a statement

$$x \leftarrow a \oplus b$$

where *a* and *b* are constant, we can evaluate $c \leftarrow a \oplus b$ at compile time and replace the statement with

$$x \leftarrow c$$

Constant Conditions

A conditional branch:

if $a < b$ **goto** L_1 **else** L_2

where *a* and *b* are constant, can be replaced by either:

goto L_1 or **goto** L_2

depending on the (compile-time) value of $a < b$.

The control-flow edge from *L* must be deleted.

The number of predecessors of L_2 (or L_1) is reduced, and the *φ*-functions in that block must be adjusted by removing the appropriate argument.

Unreachable code

Deleting a predecessor may cause block L_2 to become unreachable.

In this case, all the statements in *L*2 can be deleted.

The variables that are used in these statements are now potentially unused.

The block itself should be deleted, reducing the number of predecessors of its successor blocks.

Common Subexpression Elim

Compute *reaching expressions*, that is, find statements of the form

```
n: v ← x ⊕ y
```

such that the path from *n* to *s* does not compute $x \oplus y$ or define *x* or *y*.

Choose a new temporary *w*, and for such *n*, rewrite as:

```
n: w ← x ⊕ y
n': v ← w
```

Finally, modify statement *s* to be:

```
s: t ← w
```

We will rely on copy propagation to remove some or all of the extra assignment quadruples.

Peephole Optimizations

Peephole Optimizations

Peephole optimizations examine a sliding window of target instructions (called the *peephole*) and replacing instruction sequences within the peephole by a shorter or faster sequence, whenever possible.

This can be done on IR or on machine code.

Redundant Loads and Stores

If we see the instruction sequence

```
mov dword ptr [esp + 8], eax
mov eax, dword ptr [esp + 8]
```

We can remove it if it is in the same basic block.

Algebraic Simplification

We can identify algebraic identities such as:

```
x + 0 = 0 + x = x
x * 1 = 1 * x = x
    x - 0 = x
     x/1 = x
```

to eliminate computations from a basic block.

Strength Reduction

Certain machine instructions are more expensive than others. One classic example is multiplication being 30ish cycles while addition and shifting are just 1 cycle.

Strength reduction replaces a more expensive operation with a cheaper one.

In the simplest case, we can do things such as replace multiplications with shifts (taking care to deal with the rounding of negative numbers correctly).

Expensive	Cheaper
x^2	`x * x`
`2 * x`	`x + x`
`x/2`	`x * 0.5`

Loop Optimizations

Loops

A **loop** in a CFG is a set of nodes *S* including a **header** node *h* with the following properties:

- From any node in *S* there is a path of directed edges leading to *h*.
- There is a path of directed edges from *h* to any node in *S*.
- There is no edge from any node outside *S* to any node in *S* other than *h*.

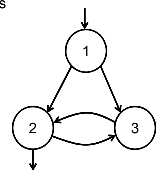
A *loop entry* node is one with some predecessor outside the loop.

A *loop exit* node is one with a successor outside the loop.

Natural Loops

A *back edge* in a CFG is an edge whose head dominates its tail.

In the graph shown here, since there are ways to enter into block 2 without going into block 3 first and vice versa, there is no dominance relationship. Thus, there is no back edge in this graph.

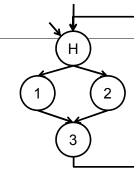


Given a back edge, m→ n, the **natural loop** of m→n is the subgraph consisting of:

1. The set of nodes containing *n* and
2. All of the nodes from which *m* can be reached in the CFG without passing through *n*
3. The edge set connecting all of the nodes in its node set.

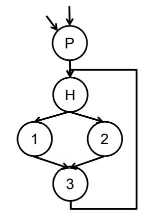
Node *n* is thus the loop header.

Loop Preheader



Loop-based optimizations often wish to move code out of a loop and place it just before the loop header.

To guarantee we have such a place available, we will introduce a new, initially empty block called the **loop preheader**.



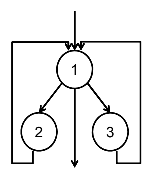
All edges from outside the loop whose target was the loop header will now go to the preheader.

The preheader has a single control flow edge from it to the header.

Nested Loops

As long as two natural loops do not share the same header, they are either:

- Disjoint,
- Nested, or
- Make up just one loop.



The CFG shown here could arise from two different situations, however:

```
B1: if( i >= 100 ) goto B4;
    else if( (i%10)==0 ) goto B3;
    else …

B2: …
    goto B1;
B3: …
    goto B1;
B4: …
```

The left loop is an inner loop.

```
B1: if( i<j ) goto B2;
    else if( i>j ) goto B3;
    else goto B4;

B2: …
    goto B1;
B3: …
    goto B1;
B4: …
```

They make up one loop together.

Strongly Connected Components

Natural loops are not the only looping structure we can observe. The most general looping structure that may occur in a CFG is a **strongly connected component**.

A strongly connected component (SCC) is a subgraph:

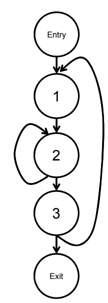
$$G_S = (N_S, E_S)$$

Such that every node in NS is reachable from every other node by a path that includes only edges in ES.

An SCC is *maximal* if every SCC containing it is the component itself.

In the CFG shown here, there are 2 SCCs:

1. ({1,2,3}, E) is maximal
2. ({2},{2 → 2}) is not maximal



Reducible Flow Graphs

Reducibility is an important but misnamed property.

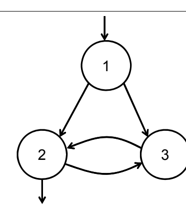
Reducible results from several kinds of transformations that can be applied to CFGs successively to reduce subgraphs into single nodes. If the resultant subgraph has a single node, it is considered **reducible**.

A flowgraph G=(N,E) is reducible iff E can be partitioned into disjoint sets E_F, the forward edge set, and E_B, the back edge set, such that (N,E_F) forms a DAG in which each node can be reached from the entry node, and the edges in E_B are all back edges.

Alternatively, if a CFG is reducible, all the loops in it are natural loops characterized by their back edges and vice versa: There are no jumps into the middle of loops. Each loop is entered via its loop header.

Reducible Flow Graphs

The graph shown here is not a natural loop. Either node in the strongly connected component *(2, 3)* can be reached without going through the other.



Regions of the source code, called improper regions, make CFGs irreducible. They cause multiple-entry strongly connected components to arise in the CFG.

Some languages restrict these improper regions from occuring, and even in languages that don't, most loops in programs are natural loops.

Advantages of Reducible CFGs

Many dataflow analyses can be done very efficiently on reducible flow graphs.

Instead of using fixed-point iteration, we can determine an order for computing the assignments, and calculate in advance how many assignments will be necessary There will never be a need to check to see if anything changed.

Loop-Invariant Computations

If a loop contains a statement

$$t \leftarrow a \oplus b$$

such that *a* has the same value each iteration, and *b* has the same value each iteration, then *t* will also have the same value each iteration.

We cannot always tell if *a* will have the same value every time, so we will conservatively approximate. The definition:

$$d : t \leftarrow a_1 \oplus a_2$$

is loop-invariant within loop *L* if, for each operand a_i,

1. a_i is a constant, or
2. all the definitions of a_i that reach *d* are outside the loop, or
3. only one definition of a_i reaches *d*, and that definition is loop-invariant.

This leads naturally to an iterative algorithm for finding loop-invariant definitions:

1. Find all the definitions whose operands are constant or from outside the loop
2. Repeatedly find definitions whose operands are loop-invariant.

Hoisting

Suppose `t ← a ⊕ b` is loop-invariant. Can we hoist it out of the loop and into the preheader?

L0:
  t ← 0
L1:
  i ← i + 1
  t ← a ⊕ b
  M[i] ← t
  if i<N goto L1
L2:
  x ← t
	L0:
	  t ← 0
	L1:
	  if i≥N goto L2
	  i ← i + 1
	  t ← a ⊕ b
	  M[i] ← t
	  goto L1
	L2:
	  x ← t
		L0:
		  t ←0
		L1:
		  i ← i + 1
		  t ← a ⊕ b
		  M[i] ← t
		  t ← 0
		  M[j] ← t
		  if i<N goto L1
		L2:
			L0:
			  t ←0
			L1:
			  M[j]← t
			  i ← i + 1
			  t ← a ⊕ b
			  M[i] ← t
			  if i<N goto L1
			L2:
			  x ←t

Hoist			

Hoisting

Suppose `t ← a ⊕ b` is loop-invariant. Can we hoist it out of the loop and into the preheader?

L0:
  t ← 0
L1:
  i ← i + 1
  t ← a ⊕ b
  M[i] ← t
  if i<N goto L1
L2:
  x ← t
	L0:
	  t ← 0
	L1:
	  if i≥N goto L2
	  i ← i + 1
	  t ← a ⊕ b
	  M[i] ← t
	  goto L1
	L2:
	  x ← t
		L0:
		  t ←0
		L1:
		  i ← i + 1
		  t ← a ⊕ b
		  M[i] ← t
		  t ← 0
		  M[j] ← t
		  if i<N goto L1
		L2:
			L0:
			  t ←0
			L1:
			  M[j]← t
			  i ← i + 1
			  t ← a ⊕ b
			  M[i] ← t
			  if i<N goto L1
			L2:
			  x ←t

Hoist			

Hoisting makes the program compute the same result faster.

Hoisting

Suppose `t ← a ⊕ b` is loop-invariant. Can we hoist it out of the loop and into the preheader?

L0:
  t ← 0
L1:
  i ← i + 1
  t ← a ⊕ b
  M[i] ← t
  if i<N goto L1
L2:
  x ← t
	L0:
	  t ← 0
	L1:
	  if i≥N goto L2
	  i ← i + 1
	  t ← a ⊕ b
	  M[i] ← t
	  goto L1
	L2:
	  x ← t
		L0:
		  t ←0
		L1:
		  i ← i + 1
		  t ← a ⊕ b
		  M[i] ← t
		  t ← 0
		  M[j] ← t
		  if i<N goto L1
		L2:
			L0:
			  t ←0
			L1:
			  M[j]← t
			  i ← i + 1
			  t ← a ⊕ b
			  M[i] ← t
			  if i<N goto L1
			L2:
			  x ←t

Hoist	**Don't**		

Hoisting makes the program faster but incorrect – the original program does not *always* execute $t \leftarrow a \oplus b$, but the transformed program does, producing an incorrect value for *x* if $i \geq N$ initially.

Hoisting

Suppose `t ← a ⊕ b` is loop-invariant. Can we hoist it out of the loop and into the preheader?

L0:
  t ← 0
L1:
  i ← i + 1
  t ← a ⊕ b
  M[i] ← t
  if i<N goto L1
L2:
  x ← t
	L0:
	  t ← 0
	L1:
	  if i≥N goto L2
	  i ← i + 1
	  t ← a ⊕ b
	  M[i] ← t
	  goto L1
	L2:
	  x ← t
		L0:
		  t ←0
		L1:
		  i ← i + 1
		  t ← a ⊕ b
		  M[i] ← t
		  t ← 0
		  M[j] ← t
		  if i<N goto L1
		L2:
			L0:
			  t ←0
			L1:
			  M[j]← t
			  i ← i + 1
			  t ← a ⊕ b
			  M[i] ← t
			  if i<N goto L1
			L2:
			  x ←t

Hoist	**Don't**	**Don't**	

The original loop had more than one definition of *t*, and the transformed program interleaves the assignments to *t* in a different way.

Hoisting

Suppose `t ← a ⊕ b` is loop-invariant. Can we hoist it out of the loop and into the preheader?

L0:	L0:	L0:	L0:
t ← 0	t ← 0	t ←0	t ←0
L1:	L1:	L1:	L1:
i ← i + 1	if i≥N goto L2	i ← i + 1	M[j]← t
t ← a ⊕ b	i ← i + 1	**t ← a ⊕ b**	i ← i + 1
M[i] ← t	**t ← a ⊕ b**	M[i] ← t	**t ← a ⊕ b**
if i<N goto L1	M[i] ← t	t ← 0	M[i] ← t
L2:	goto L1	M[j] ← t	if i<N goto L1
x ← t	L2:	if i<N goto L1	L2:
	x ← t	L2:	x ←t
Hoist	**Don't**	**Don't**	**Don't**

There is a use of *t* before the loop-invariant definition, so after hoisting, this use will have the wrong value on the first iteration of the loop.

Hoisting

With these pitfalls in mind, we can set the criteria for hoisting

`d: t ← a ⊕ b`

to the end of the loop preheader:

1. *d* dominates all loop exits at which *t* is *live-out*, and
2. there is only one definition of *t* in the loop, and
3. *t* is not *live-out* of the loop preheader.

Condition (1) tends to prevent many computations from being hoisted from **while** loops.

To solve this problem, we can transform the **while** loop into a **do…while** loop preceded by an **if** statement. The drawback is that this requires duplication of the statements that were in the header node, and will not work in cases where there is a **break** in the loop.

Induction Variables

Some loops have a variable *i* that is incremented or decremented, and a variable *j* that is set (in the loop) to $i \times c + d$, where *c* and *d* are loop-invariant.

```
j = i * c + d
```

Then we can calculate *j* 's value without reference to *i* : whenever *i* is incremented by *a* we can increment *j* by $c \times a$.

```
i++;
j+= c * a;
```

Detection Of Induction Variables

```
    sum ← 0
    i ← 0
L1: if i ≥ n goto L2
    j ← i * 4
    k ← j + a
    x ← M[k]
    sum ← sum + x
    i ← i + 1
    goto L1
L2:
```

We say that a variable such as *i* is a *basic induction variable*, and *j* and *k* are *derived induction variables in the family of i*.

Right after *j* is defined, we have $j = a_j + i \times b_j$, where $a_j = 0$ and $b_j = 4$.

We can completely characterize the value of *j* at its definition by *(i, a, b)*, where *i* is a basic induction variable and *a* and *b* are loop-invariant expressions.

Another derived induction variable is $k \leftarrow j + c_k$ (where c_k is loop-invariant). Thus *k* is also in the family of *i*. We can characterize *k* by the triple: $(i, a_j + c_k, b_j)$, that is, $k = aj + c_k + i \times b_j$.

Detection Of Induction Variables

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.

The variable *k* is a **derived induction variable** in loop *L* if:

1. There is only one definition of *k* within *L*, of the form:

   $k \leftarrow j \times c$ or $k \leftarrow j + d$

   where *j* is an induction variable and *c, d* are loop-invariant; and

2. if *j* is a derived induction variable in the family of *i*, then:
   a) the only definition of *j* that reaches *k* is the one in the loop, and
   b) there is no definition of *i* on any path between the definition of *j* and the definition of *k*.

Strength Reduction

For each derived induction variable *j* whose triple is *(i, a, b)*, make a new variable *j'* (although different derived induction variables with the same triple can share the same *j'* variable).

After each assignment $i \leftarrow i + c$, make an assignment $j' \leftarrow j' + c \times b$, where $c \times b$ is a loop-invariant expression that may be computed in the loop preheader. If *c* and *b* are both constant, then the multiplication may be done at compile time.

Replace the (unique) assigment to *j* with $j \leftarrow j'$.

Finally, it is necessary to initialize *j* at the end of the loop preheader, with:
$j' \leftarrow a + i \times b$.

Strength Reduction

```
    sum ← 0
    i ← 0
L1: if i ≥ n goto L2
    j ← i * 4
    k ← j + a
    x ← M[k]
    sum ← sum + x
    i ← i + 1
    goto L1
L2:
```

We find that *j* is a derived induction variable with triple *(i,* 0, 4*)*, and *k* has triple *(i, a,* 4*)*.

```
    sum ← 0
    i ← 0
    j' ← 0
    k' ← a
L1: if i ≥ n goto L2
    j ← j'
    k ← k'
    x ← M[k]
    sum ← sum + x
    i ← i + 1
    j' ← j' + 4
    k' ← k' + 4
    goto L1
L2:
```

Elimination

```
    sum ← 0
    i ← 0
    j' ← 0
    k' ← a
L1: if i ≥ n goto L2
    j ← j'
    k ← k'
    x ← M[k]
    sum ← sum + x
    i ← i + 1
    j' ← j' + 4
    k' ← k' + 4
    goto L1
L2:
```

We can perform *dead-code elimination* to remove the statement *j* ← *j'*.

We would also like to remove all the definitions of the *useless variable j'*, but technically it is not dead, since it is used in every iteration of the loop.

A variable is **useless** in a loop *L* if it is dead at all exits from *L*, and its only use is in a definition of itself. All definitions of a useless variable may be deleted.

After the removal of *j*, the variable *j'* is useless. We can delete *j'* ← *j'* + 4. This leaves a definition of *j'* in the preheader that can now be removed by dead-code elimination.

Rewriting Comparisons

```
    sum ← 0
    i ← 0
    k' ← a
L1: if i ≥ n goto L2
    k ← k'
    x ← M[k]
    sum ← sum + x
    i ← i + 1
    k' ← k' + 4
    goto L1
L2:
```

A variable *k* is **almost useless** if it is used only in comparisons against loop-invariant values and in definitions of itself, and there is some other induction variable in the same family that is not useless.

If we have $k < n$, where *j* and *k* are **coordinated** induction variables in the family of *i*, and *n* is loop-invariant, then we know that:

$$(j - a_j)/b_j = (k - a_k)/b_k$$

so therefore the comparison $k < n$ can be written as:

$$j < (b_j / b_k) \times (n - a_k) + a_j$$

If b_j/b_k is negative, use > instead.

Rewriting Comparisons

```
    sum ← 0
    i ← 0
    k' ← a
L1: if i ≥ n goto L2
    k ← k'
    x ← M[k]
    sum ← sum + x
    i ← i + 1
    k' ← k' + 4
    goto L1
L2:
```

The comparison $i < n$ can be replaced by $k < a + 4 \times n$. Then, $a + 4 \times n$ is loop-invariant and should be hoisted. Finally, i will be useless and may be deleted.

```
    sum ← 0
    k' ← a
    b ← n * 4
    c ← a + b
L1: if k' < c goto L2
    k ← k'
    x ← M[k]
    sum ← sum + x
    k' ← k' + 4
    goto L1
L2:
```

Elimination

```
    sum ← 0
    k' ← a
    b ← n * 4
    c ← a + b
L1: if k' < c goto L2
    k ← k'
    x ← M[k]
    sum ← sum + x
    k' ← k' + 4
    goto L1
L2:
```

Finally, copy propagation can eliminate:
k ←*k'*

```
    sum ← 0
    k' ← a
    b ← n * 4
    c ← a + b
L1: if k' < c goto L2
    x ← M[k']
    sum ← sum + x
    k' ← k' + 4
    goto L1
L2:
```

Spill Cost Model

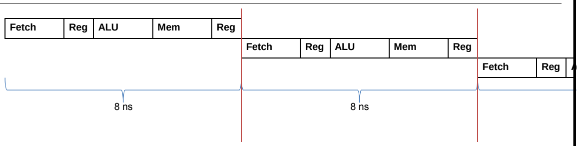
When doing register allocation, spills that must be reloaded each iteration will adversely affect the performance of the loop. We have the choice of any node in the interference graph of significant degree to spill, so our choice is arbitrary but important.

We can model the cost of spilling and derive a priority for spilling each remaining node when simply or freeze cannot proceed:

$$\text{Spill priority} = \frac{\text{(Uses+Defs outside loop)} + 10 \times \text{(Uses+Defs within loop)}}{\text{Degree}}$$

We divide by degree to slightly bias those nodes of highest degree as their spillage may result in fewer subsequent spills of the remaining nodes.

Single Cycle Implementation

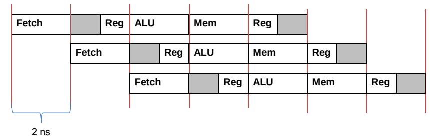


Each instruction can be done with a single 8 ns cycle

Time between the first and fourth instruction: 24 ns

For three Load instructions, it will take 3 * 8 = 24 ns

Pipelined Implementation



Each step takes 2 ns (even register file access) because the slowest step is 2 ns

Time between 1st and 4th instruction starting: 3 * 2 ns = 6 ns

Total time for the three instructions = 14 ns

Control Hazards

Control hazard: attempt to make a decision before condition is evaluated.

Branch instructions:

```
beq $1,$2,L0
add $4,$5,$6
...
L0: sub $7,$8,$9
```

Which instruction do we fetch next?

Make a **guess** that the branch is **not taken**. If we're right, there's no problem (no stalls). If we're wrong…?

What would have been stalls if we waited for our comparison are now "wrong" instructions. We need to cancel them out and make sure they have no effect. These are called **bubbles**.

Branch Prediction

Attempt to predict the outcome of the branch before doing the comparison.

- Predict branch taken (fetch branch target instruction)
- Predict branch not taken (fetch fall through)

If wrong, we'll need to squash the mispredicted instructions by setting their control signals to zero (no writes). This turns them into nops.

Times to do prediction:

- Static
  - Compiler inserts hints into the instruction stream
  - CPU predicts forward branches not taken and backwards branches taken
- Dynamic
  - Try to do something in the CPU to guess better

Loop Unrolling

A tight loop may perform better if it is unrolled: where multiple loop iterations are replaced by multiple copies of the body in a row.

```
int x;
for (x = 0; x < 100; x++)
{
    printf("%d\n", x);
}
```

```
int x;
for (x = 0; x < 100; x+=5)
{
     printf("%d\n", x);
     printf("%d\n", x+1);
     printf("%d\n", x+2);
     printf("%d\n", x+3);
     printf("%d\n", x+4);
 }
```

Loop Unrolling

Benefits:

- Reduce branches and thus potentially mispredictions
- More instruction-level parallelism

Drawbacks:

- Code size increase can cause instruction cache pressure
- Increased register usage may result in spilling

Duff’s Device

```
do {
    *to = *from++;
    /* Note that the 'to' pointer
is NOT incremented */
} while(--count > 0);
```

```
send(to, from, count)
register short *to, *from;
register count;
{
    register n = (count + 7) / 8;
    switch(count % 8) {
        case 0: do { *to=*from++;
        case 7:      *to=*from++;
        case 6:      *to=*from++;
        case 5:      *to=*from++;
        case 4:      *to=*from++;
        case 3:      *to=*from++;
        case 2:      *to=*from++;
        case 1:      *to=*from++;
                   } while(--n>0);
    }
}
```