Where are we?

- Analysis
  - Control Flow/Predicate
  - Dataflow
  - SSA
- Optimization

Optimization

- Make the code run faster on the target processor
  - My favorite topic!!
  - Anything goes
    - Look at benchmark kernels, what's the bottleneck??
    - Invent your own optimizations (easier and harder than you think)
- Classes of optimization
  - 1. Classical (machine independent)
    - Reducing operation count (redundancy elimination)
    - Simplifying operations
    - Generally good for any kind of machine
  - 2. Machine specific
    - Peephole optimizations
    - Take advantage of specialized hardware features
  - 3. ILP enhancing
    - Increasing parallelism
    - Possibly increase instructions
Classical Optimizations

- Operation-level – 1 operation in isolation
  - Constant folding, strength reduction
- Dead code elimination (global, but 1 op at a time)
- Local/Global – Pairs of operations
  - Constant propagation
  - Forward copy propagation
  - Backward copy propagation
  - CSE
  - Constant combining
  - Operation folding
- Loop – Body of a loop
  - Invariant code removal
  - Global variable migration
  - Induction variable strength reduction
  - Induction variable elimination

Caveat

- Traditional compiler class
  - Sophisticated implementations of optimizations, efficient algorithms
  - Spend entire class on 1 optimization
- For this class – Go over concepts of each optimization
  - What it is
  - When can it be applied (set of conditions that must be satisfied)

Static Single Assignment (SSA)

Static Single Assignment Advantages:

- Less space required to represent def-use chains. For each variable, space is proportional to uses + defs.
- Eliminates unnecessary relationships:

\[
\begin{align*}
  &\text{for } i = 1 \text{ to } N \text{ do } A[i] = 0 \\
  &\text{for } i = 1 \text{ to } M \text{ do } B[i] = 1
\end{align*}
\]

- No reason why both loops should be forced to use same register to hold index register.
- SSA renames second \( i \) to new register which may lead to better register allocation.
- SSA form make certain optimizations quick and easy — dominance property.
- Variables have only one definition - no ambiguity.
- Dominator information is encoded in the assignments.
Dominance property of SSA form: definitions dominate uses

- If \( x \) is the \( i \)th argument of \( \phi \)-function in node \( n \), then definition of \( x \) dominates \( i \)th predecessor of \( n \).
- If \( x \) is used in non-\( \phi \) statement in node \( n \), then definition of \( x \) dominates \( n \).

Dead Code Elimination

Given \( d: t = x \odot y \)

- \( t \) is live at end of node \( d \) if there exists path from end of \( d \) to use of \( t \) that does not go through definition of \( t \).
- if program not in SSA form, need to perform liveness analysis to determine if \( t \) live at end of \( d \).
- if program is in SSA form:
  - cannot be another definition of \( t \)
  - if there exists use of \( t \), then path from end of \( d \) to use exists, since definitions dominate uses.
  - every use has a unique definition
  - \( t \) is live at end of node \( d \) if \( t \) is used at least once

Algorithm:

WHILE (for each temporary \( t \) with no uses &\& statement defining \( t \) has no other side-effects) DO

1: \( r1 = 5 \)
2: \( r2 = 10 \)
3: \( \text{branch } r3 > r2 \)
4: \( r2' = r2 + 15 \)
5: \( r4 = r3 + X \)
6: \( r2'' = \emptyset (r2', r2) \)
7: \( M[r4] = r2'' \)
Dead Code Elimination

- Remove any operation who’s result is never consumed
- Rules
  - X can be deleted
    - no stores or branches
  - DU chain empty or dest register not live
- This misses some dead code!!
  - Especially in loops
  - Critical operation
    - store or branch operation
  - Any operation that does not directly or indirectly feed a critical operation is dead
  - Trace UD chains backwards from critical operations
  - Any op not visited is dead

\[
\begin{align*}
  r_1 &= 3 \\
  r_2 &= 10 \\
  r_4 &= r_4 + 1 \\
  r_7 &= r_1 \times r_4 \\
  r_3 &= r_3 + 1 \\
  r_5 &= r_2 + r_1 \\
  r_2 &= 0 \\
  \text{store (r1, r3)}
\end{align*}
\]

Constant Folding

- Simplify 1 operation based on values of src operands
  - Constant propagation creates opportunities for this
- All constant operands
  - Evaluate the op, replace with a move
    - \( r_1 = 3 \times 4 \rightarrow r_1 = 12 \)
    - \( r_1 = 3 / 0 \rightarrow ?? \) Don’t evaluate excepting ops !, what about floating-point?
  - Evaluate conditional branch, replace with BRU or noop
    - if (1 < 2) goto BB2 \( \rightarrow \) BRU BB2
    - if (1 > 2) goto BB2 \( \rightarrow \) convert to a noop
- Algebraic identities
  - \( r_1 = r_2 + 0, r_2 - 0, r_2 | 0, r_2 ^ 0, r_2 << 0, r_2 >> 0 \)
  - \( r_1 = r_2 \)
  - \( r_1 = 0 * r_2, 0 / r_2, 0 & r_2 \)
  - \( r_1 = 0 \)
  - \( r_1 = r_2 * 1, r_2 / 1 \)
  - \( r_1 = r_2 \)

Strength Reduction

- Replace expensive ops with cheaper ones
  - Constant propagation creates opportunities for this
- Power of 2 constants
  - Multiply by power of 2, replace with left shift
    - \( r_1 = r_2 \times 8 \rightarrow r_1 = r_2 << 3 \)
  - Divide by power of 2, replace with right shift
    - \( r_1 = r_2 / 4 \rightarrow r_1 = r_2 >> 2 \)
  - Remainder by power of 2, replace with logical and
    - \( r_1 = r_2 \text{ REM 16} \rightarrow r_1 = r_2 & 15 \)
- More exotic
  - Replace multiply by constant by sequence of shift and adds/subs
    - \( r_1 = r_2 \times 6 \)
      - \( r_100 = r_2 << 2; r_101 = r_2 << 1; r_1 = r_100 + r_101 \)
    - \( r_1 = r_2 \times 7 \)
      - \( r_100 = r_2 << 3; r_1 = r_100 - r_2 \)
Class Problem

Optimize this applying
1. constant folding
2. strength reduction
3. dead code elimination

Constant Propagation

- Forward propagation of moves of the form
  - $rx = L$ (where $L$ is a literal)
  - Maximally propagate
  - Assume no instruction encoding restrictions
- When is it legal?
  - SRC: Literal is a hard coded constant, so never a problem
  - DEST: Must be available
    - Guaranteed to reach
    - May reach not good enough

Simple Constant Propagation

Given $d: t = c, c$ is constant Given $u: x = t \ \text{op} \ \ b$

- if program not in SSA form:
  - need to perform reaching definition analysis
  - use of $t$ in $u$ may be replaced by $c$ if $d$ reaches $u$ and no other definition of $t$ reaches $u$

- if program is in SSA form:
  - $d$ reaches $u$, since definitions dominate uses, and no other definition of $t$ exists on path from $d$ to $u$
  - $d$ is only definition of $t$ that reaches $u$, since it is the only definition of $t$
    * any use of $t$ can be replaced by $c$
    * any $\text{op}$-function of form $\text{op} = \phi(c_1, c_2, ..., c_n)$, where $c_i = c$, can be replaced by $\text{op} = c$
Local Constant Propagation

- Consider 2 ops, X and Y in a BB, X is before Y
  - 1. X is a move
  - 2. src1(X) is a literal
  - 3. Y consumes dest(X)
  - 4. There is no definition of dest(X) between X and Y
  - 5. No danger betw X and Y
    - When dest(X) is a Macro reg, BRL destroys the value

\[
\begin{align*}
\text{r1} &= 5 \\
\text{r2} &= \_x \\
\text{r3} &= 7 \\
\text{r4} &= \text{r4 + r1} \\
\text{r5} &= \text{r1 + r2} \\
\text{r6} &= \text{r1 + 1} \\
\text{r7} &= \text{r2 + r1} \\
\text{r8} &= \text{r1 + r2} \\
\text{r9} &= \text{r3 + r5} \\
\text{r10} &= \text{r3 - r1}
\end{align*}
\]

Global Constant Propagation

- Consider 2 ops, X and Y in different BBs
  - 1. X is a move
  - 2. src1(X) is a literal
  - 3. Y consumes dest(X)
  - 4. X is in \text{a_in}(BB(Y))
  - 5. Dest(x) is not modified between the top of BB(Y) and Y
  - 6. No danger betw X and Y
    - When dest(X) is a Macro reg, BRL destroys the value

\[
\begin{align*}
r1 &= \text{r1 + r2} \\
r7 &= \text{r1 - r2} \\
r8 &= \text{r1 * r2} \\
r9 &= \text{r1 + r2}
\end{align*}
\]

Class Problem

Optimize this applying
1. constant propagation
2. constant folding
3. strength reduction
4. dead code elimination
Forward Copy Propagation

- Forward propagation of the RHS of moves
  - \( r_1 = r_2 \)
  - \( \ldots \)
  - \( r_4 = r_1 + 1 \rightarrow r_4 = r_2 + 1 \)

- Benefits
  - Reduce chain of dependences
  - Eliminate the move

- Rules (ops X and Y)
  - X is a move
  - src1(X) is a register
  - Y consumes dest(X)
  - X.dest is an available def at Y
  - X.src1 is an available expr at Y

- \( r_2 = 0 \)
- \( r_5 = r_2 + r_3 \)
- \( r_6 = r_3 + 1 \)

Backward Copy Propagation

- Backward propagation of the LHS of moves
  - \( r_1 = r_2 + r_3 \rightarrow r_4 = r_2 + r_3 \)
  - \( \ldots \)
  - \( r_5 = r_1 + r_6 \rightarrow r_5 = r_4 + r_6 \)
  - \( \ldots \)
  - \( r_4 = r_1 \rightarrow \text{noop} \)

- Rules (ops X and Y in same BB)
  - dest(X) is a register
  - dest(X) not live out of BB(X)
  - Y is a move
  - dest(Y) is a register
  - Y consumes dest(X)
  - dest(Y) not consumed in (X...Y)
  - dest(Y) not defined in (X...Y)
  - There are no uses of dest(X) after the first redefinition of dest(Y)

- \( r_1 = r_8 + r_9 \)
- \( r_2 = r_9 + r_1 \)
- \( r_4 = r_2 \)
- \( r_6 = r_2 + 1 \)
- \( r_9 = r_1 \)
- \( r_{10} = r_6 \)
- \( r_5 = r_6 + 1 \)
- \( r_4 = 0 \)
- \( r_8 = r_2 + r_7 \)

CSE – Common Subexpression Elimination

- Eliminate recomputation of an expression by reusing the previous result
  - \( r_1 = r_2 \cdot r_3 \)
  - \( \rightarrow r_{100} = r_1 \)
  - \( \ldots \)
  - \( r_4 = r_2 \cdot r_3 \rightarrow r_4 = r_{100} \)

- Benefits
  - Reduce work
  - Moves can get copy propagated

- Rules (ops X and Y)
  - X and Y have the same opcode
  - src(X) = src(Y), for all srcs
  - expr(X) is available at Y
  - if X is a load, then there is no store that may write to address(X) along any path between X and Y

- if op is a load, call it redundant
- Load elimination rather than CSE

- \( r_2 = r_2 + 1 \)
- \( r_5 = r_2 \cdot r_6 \)
- \( r_6 = r_3 \cdot 7 \)
- \( r_8 = r_4 \cdot r_7 \)
- \( r_9 = r_3 \cdot 7 \)
Optimize this applying
1. constant propagation
2. constant folding
3. strength reduction
4. dead code elimination
5. forward copy propagation
6. backward copy propagation
7. CSE

Constant Combining

- Combine 2 dependent ops into 1 by combining the literals
  - \( r1 = r2 + 4 \)
  - \( \ldots \)
  - \( r5 = r1 - 9 \rightarrow r5 = r2 - 5 \)
- First op often becomes dead
- Rules (ops X and Y in same BB)
  - X is of the form \( rx + K \)
  - dest(X) \(!=\) src1(X)
  - Y is of the form \( ry + K \)
    (comparison also ok)
  - Y consumes dest(X)
  - src1(X) not modified in (X...Y)

Operation Folding

- Combine 2 dependent ops into 1 complex op
  - Classic example is MPYADD
    - \( r1 = r2 \times r3 \)
  - \( \ldots \)
    - \( r5 = r1 + r4 \rightarrow r5 = r2 \times r3 + r4 \)
- First op often becomes dead
- Borders on machine dependent opti (often it is \(!!\) )
- Rules (ops X and Y in same BB)
  - X is an arithmetic operation
  - dest(X) \(!=\) any src(X)
  - Y is an arithmetic operation
  - Y consumes dest(X)
  - X and Y can be merged
  - src(X) not modified in (X...Y)
Constant Combining

- Combine 2 dependent ops into 1 by combining the literals
  - \( r_1 = r_2 + 4 \)
  - ...
  - \( r_5 = r_1 - 9 \rightarrow r_5 = r_2 - 5 \)
- First op often becomes dead
- Rules (ops X and Y in same BB)
  - X is of the form \( \text{rx} \rightarrow \text{K} \)
  - dest(X) != src1(X)
  - Y is of the form \( \text{ry} \rightarrow \text{K} \) (comparison also ok)
  - Y consumes dest(X)
  - src1(X) not modified in (X...Y)

\[
\begin{align*}
\text{r}_1 & = \text{r}_2 + 4 \\
\text{r}_3 & = \text{r}_1 < 0 \\
\text{r}_2 & = \text{r}_3 + 6 \\
\text{r}_7 & = \text{r}_1 - 3 \\
\text{r}_8 & = \text{r}_7 + 5 \\
\end{align*}
\]

Operation Folding

- Combine 2 dependent ops into 1 complex op
  - Classic example is MPYADD
  - \( r_1 = r_2 \times r_3 \)
  - ...
  - \( r_5 = r_1 + r_4 \rightarrow r_5 = r_2 \times r_3 + r_4 \)
- First op often becomes dead
- Borders on machine dependent opti (often it is !! )
- Rules (ops X and Y in same BB)
  - X is an arithmetic operation
  - dest(X) != any src(X)
  - Y is an arithmetic operation
  - Y consumes dest(X)
  - X and Y can be merged
  - src(X) not modified in (X...Y)

\[
\begin{align*}
\text{r}_1 & = \text{r}_2 \& 4 \\
\text{r}_3 & = \text{r}_1 ^ -1 \\
\text{r}_2 & = \text{r}_3 < 6 \\
\text{r}_4 & = \text{r}_2 = = 0 \\
\text{r}_5 & = \text{r}_6 < < 1 \\
\text{r}_7 & = \text{r}_5 + r_8 \\
\end{align*}
\]

Loop Optimizations

- The most important set of optimizations
  - Because programs spend so much time in loops
- Optis
  - Invariant code removal
  - Global variable migration
  - Induction variable strength reduction
  - Induction variable elimination
Recall Loop Terminology

- \( r_1, r_4 \) are basic induction variables
- \( r_7 \) is a derived induction variable

\[
\begin{align*}
  r_1 &= 3 \\
  r_2 &= 10 \\
  r_4 &= r_4 + 1 \\
  r_7 &= r_4 \times 3 \\
  r_2 &= 0 \\
  r_3 &= r_2 + 1 \\
  r_1 &= r_1 + 2 \\
  &\text{store (}r_1, r_3\text{)}
\end{align*}
\]

临港 BB

exit BB

\hspace{1cm}

Global Variable Migration

- Assign a global variable temporarily to a register for the duration of the loop
  - Load in preheader
  - Store at exit points
- Rules
  - \( X \) is a load or store
  - address\((X)\) not modified in the loop
  - if \( X \) not executed on every iteration, then \( X \) must provably not cause an exception
  - All memory ops in loop whose address can equal address\((X)\) must always have the same address as \( X \)
Induction Variable Strength Reduction

- Create basic induction variables from derived induction variables

- Rules
  - X is a *, <<, + or operation
  - src1(X) is a basic ind var
  - src2(X) is invariant
  - No other ops modify dest(X)
  - dest(X) != src(X) for all srcs
  - dest(X) is a register

Induction Variable Elimination

- Remove unnecessary basic induction variables from the loop by substituting uses with another BIV
- Rules (same init val, same inc)
  - Find 2 basic induction vars x, y
  - x, y in same family
    - incremented in same places
  - increments equal
  - initial values equal
  - x not live when you exit loop
  - for each BB where x is defined, there are no uses of x between first/last defn of x and last/first defn of y

Optimize this applying
1. loop invariant removal
2. global variable migration
+ other optis
Induction Variable Elimination (2)

- 5 variants discussed in Mahlke thesis
  - 1. Trivial – induction variable that is never used except by the increments themselves, not live at loop exit
  - 2. Same increment, same initial value
  - 3. Same increment, initial values are a known constant offset from one another
  - 4. Same increment, know nothing about relation of initial values
  - 5. Different increments, know nothing about initial values

- The higher the number, the more complex the elimination
  - Also, the more expensive it is
  - 1,2 are basically free, so always should be done
  - 3-5 require preheader operations

Class Problem

Optimize this applying
Induction var strength red
Induction variable elim

ILP Optimization

- Traditional optimizations
  - Redundancy elimination
  - Reducing operation count

- ILP (instruction-level parallelism) optimizations
  - Increase the amount of parallelism and the ability to overlap operations
  - Operation count is secondary, often trade parallelism for extra instructions (avoid code explosion)

- ILP increased by breaking dependences
  - True or flow = read after write dependence
  - False or (anti/output) = write after read, write after write
Register Renaming

- Remove dependences caused by variable re-use
  - Re-use of source variables
  - Re-use of temporaries
  - Anti, output dependences
- Create a new variable to hold each unique life time
- Very simple transformation with straight-line code
  - Make each def a unique register
  - Substitute new name into subsequent uses

\[
\begin{align*}
  a: & \quad r_1 = r_2 + r_3 \\
  b: & \quad r_3 = r_4 + r_5 \\
  c: & \quad r_1 = r_7 * r_8 \\
  d: & \quad r_7 = r_1 + r_5 \\
  e: & \quad r_1 = r_3 + 4 \\
  f: & \quad r_4 = r_7 + 4
\end{align*}
\]

Global Register Renaming

- Straight-line code strategy does not work
  - A single use may have multiple reaching defs
- Web = Collection of defs/uses which have possible value flow between them
  - Identify webs
    - Take a def, add all uses
    - Take all uses, add all reaching defs
    - Take all defs, add all uses
    - repeat until stable soln
  - Each web renamed if name is the same as another web

Rename with Copy

- Renaming within a web
  - The worst case is a web spans all defs/uses
  - Want to enable some of the defs within the web to be reordered or executed in parallel
- Xform
  - Rename def
  - Rename uses for which def is the only reaching def
  - Insert copy
    - \texttt{orig\_dest = new\_dest}
Predicate Promotion

- Predicate promotion or predicate speculation
  - Remove dependence between
    CMPP and predicated operation
  - Modify predicate of an operation to
    an ancestor predicate
  - Operation executes more often
    than it should, "speculated"
- \( x = \ldots \) if \( p_1 \to \) if \( p_2 \)
- Where \( p_2 \) is an ancestor of \( p_1 \)
- Legal if \( x \) not live on \( p_2 \to p_1 \)
- And, \( op \) will not cause a spurious exception

\[
\begin{align*}
r_1 &= r_2 + r_3 \\
r_7 &= 0 \\
p_1, p_2 &= \text{CMPP} . \text{UN} . \text{UC}(r_1 < r_5) \\
r_4 &= r_5 \times r_6 \text{ if } p_1 \\
r_7 &= r_8 + r_9 \text{ if } p_2 \\
r_{10} &= r_4 + 4 \text{ if } p_1 \\
r_{11} &= r_7 + 1 \text{ if } T
\end{align*}
\]

Promote with Copy

- Similar to rename with copy
  - Promotion alone not legal
    because a live value destroyed
  - Rename destination, can
    promote to any ancestor
  - Might as well choose True
  - Substitute uses for which def is
    the only reaching def
  - Insert copy of old_dest =
    new_dest if original_ped
  - Again, must ensure operation
    will not cause a spurious exception

\[
\begin{align*}
r_7 &= 0 \\
p_1, p_2 &= \text{CMPP} . \text{UN} . \text{UC}(r_1 < r_5) \\
r_7 &= \text{load}(r_8) \text{ if } p_2 \\
r_{12} &= r_7 + 1 \text{ if } p_2 \\
r_1 &= r_7 + 1 \text{ if } T
\end{align*}
\]

Class Problem

1. Promote everything to its highest predicate w/o renaming
2. Promote any defs of \( r_1 \), \( r_2 \) that remain predicated to True
   using promotion with renaming

\[
\begin{align*}
r_1 &= 0 \text{ if } T \\
p_1 &= \text{CMPP} . \text{UN}(r_3 < r_4) \text{ if } T \\
r_2 &= r_6 + 3 \text{ if } p_1 \\
p_2, p_3 &= \text{CMPP} . \text{UN} . \text{UC}(r_5 < r_6) \text{ if } p_1 \\
r_1 &= r_5 + 1 \text{ if } p_2 \\
r_{10} &= r_2 + r_3 \text{ if } p_2 \\
r_1 &= r_3 \times 3 \text{ if } p_3 \\
r_{11} &= \text{load}(r_1) \text{ if } p_3 \\
\text{store} (r_1, r_{10}) &= \text{if } T \\
\text{store} (r_3, r_{11}) &= \text{if } T
\end{align*}
\]
Back Substitution

- Generation of expressions by compiler frontends is very sequential
  - Account for operator precedence
  - Apply left-to-right within same precedence
- Back substitution
  - Create larger expressions
    - Iteratively substitute RHS expression for LHS variable
  - Note – may correspond to multiple source statements
  - Enable subsequent optis
- Optimization
  - Re-compute expression in a more favorable manner

$y = a + b + c - d + e - f$;

$r9 = r1 + r2$
$r10 = r9 + r3$
$r11 = r10 - r4$
$r12 = r11 + r5$
$r13 = r12 - r6$

Subs r12:
$r13 = r11 + r5 - r6$

Subs r11:
$r13 = r10 - r4 + r5 - r6$

Subs r10:
$r13 = r9 + r3 - r4 + r5 - r6$

Subs r9:
$r13 = r1 + r2 + r3 - r4 + r5 - r6$

Tree Height Reduction

- Re-compute expression as a balanced binary tree
  - Obey precedence rules
  - Essentially re-parenthesize
- Effects
  - Height reduced (n terms)
    - n-1 (assuming unit latency)
    - ceil(log2(n))
  - Number of operations remains constant
- Cost
  - Temporary registers “live” longer
- Watch out for
  - Always ok for integer arithmetic
  - Floating-point – may not be!

Fancier Tree Height Reduction

- Take advantage of literals
  - Reassociate to maximize opportunities for combining literals at compile time
  - Reduces amount of computation

original:
$r9 = r1 + r2$
$r10 = r9 + r3$
$r11 = r10 - r4$
$r12 = r11 + r5$
$r13 = r12 - r6$

after back subs:
$r13 = r1 + r2 + r3 - r4 + r5 - r6$

final code:
$t1 = r1 + r2$
$t2 = r3 - r4$
$t3 = r5 - r6$
$t4 = t1 + t2$
$r13 = t4 + t3$

r1 + r2

+  

r3 - r4

+  

r13

after back subs:
$r13 = r1 + 4 + r2 - 3 + r3 - 6$

reassociate:
$r13 = r1 + r2 + r3 + (4 - 3 - 6)$

simplify:
$r13 = r1 + r2 + r3 - 5$

balance:
$r1 + r2$

+  

r3 - 5

+  

r13
Class Problem

Assume: + = 1, * = 3

<table>
<thead>
<tr>
<th>operand</th>
<th>0</th>
<th>0</th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>arrival times</td>
<td>r1</td>
<td>r2</td>
<td>r3</td>
<td>r4</td>
<td>r5</td>
<td>r6</td>
</tr>
</tbody>
</table>

- r10 = r1 * r2
- r11 = r10 + r3
- r12 = r11 + r4
- r13 = r12 - r5
- r14 = r13 + r6

Back substitute
Re-express in tree-height reduced form
Account for latency and arrival times

Optimizing Unrolled Loops

Unroll = replicate loop body n-1 times.

Hope to enable overlap of operation execution from different iterations

Not possible!
Register Renaming on Unrolled Loop

```
loop:  r1 = load(r2)
       r3 = load(r4)
       r5 = r1 + r3
       r6 = r5 + r5
       r2 = r2 + 4
       r4 = 4 + 4
       r1 = load(r2)
       r3 = load(r4)
       r5 = r1 + r3
       r6 = r6 + r5
       r2 = r2 + 4
       r4 = r4 + 4
       if (r4 < 400) goto loop

iter1
r1 = r1 + 4
r2 = r2 + 4
r3 = r3 + 4
r4 = r4 + 4
r5 = r5 + 4
r6 = r6 + r5

iter2
r1 = r1 + r2
r3 = r3 + 4
r4 = r4 + 4
r5 = r5 + 4
r6 = r6 + r15
r2 = r2 + 4
r4 = r4 + 4
r5 = r5 + 4
if (r4 < 400) goto loop

iter3
r1 = r1 + 4
r2 = r2 + 4
r3 = r3 + 4
r4 = r4 + 4
r5 = r5 + 4
r6 = r6 + r15

```

Register Renaming is Not Enough!

- Still not much overlap possible
- Problems
  - r2, r4, r6 sequentialize the iterations
  - Need to rename these
- 2 specialized renaming opts
  - Accumulator variable expansion (r6)
  - Induction variable expansion (r2, r4)

Accumulator Variable Expansion

```
loop:  r1 = load(r2)
       r3 = load(r4)
       r5 = r1 + r3
       r6 = r6 + r5
       r2 = r2 + 4
       r4 = r4 + 4
       r1 = load(r2)
       r3 = load(r4)
       r5 = r1 + r3
       r6 = r6 + r5
       r2 = r2 + 4
       r4 = r4 + 4
       if (r4 < 400) goto loop

iter1
r1 = r1 + 4
r2 = r2 + 4
r3 = r3 + 4
r4 = r4 + 4
r5 = r5 + 4
r6 = r6 + r5

iter2
r1 = r1 + r2
r3 = r3 + 4
r4 = r4 + 4
r5 = r5 + 4
r6 = r6 + r15
r2 = r2 + 4
r4 = r4 + 4
r5 = r5 + 4
if (r4 < 400) goto loop

iter3
r1 = r1 + 4
r2 = r2 + 4
r3 = r3 + 4
r4 = r4 + 4
r5 = r5 + 4
r6 = r6 + r15

```

- Accumulator variable
  - x = x + y or x = x − y
  - where y is loop variant!!
- Create n-1 temporary accumulators
- Each iteration targets a different accumulator
- Sum up the accumulator variables at the end
- May not be safe for floating-point values
Induction Variable Expansion

- Induction variable
  - $x = x + y$ or $x = x - y$
  - where $y$ is loop invariant!
- Create n-1 additional induction variables
- Each iteration uses and modifies a different induction variable
- Initialize induction variables to init, init+step, init+2*step, etc.
- Step increased to n*original step
- Now iterations are completely independent!!

Better Induction Variable Expansion

With base+displacement addressing, often don’t need additional induction variables
- Just change offsets in each iterations to reflect step
- Change final increments to n * original step

Class Problem

loop:
- $r_{16} = r_{26} = 0$
- $\text{loop:}$
- $r_1 = \text{load}(r2)$
- $r_3 = \text{load}(r4)$
- $r_5 = r_1 + r_3$
- $r_6 = r_6 + r_5$
- $r_2 = r_2 + 12$
- $r_4 = r_4 + 12$
- $r_{11} = \text{load}(r12)$
- $r_{13} = r_{11} + r_{13}$
- $r_{15} = r_{16} + r_{15}$
- $r_{12} = r_{12} + 12$
- $r_{14} = r_{14} + 12$
- $r_{21} = \text{load}(r22)$
- $r_{23} = \text{load}(r24)$
- $r_{25} = r_{21} * r_{23}$
- $r_{26} = r_{26} + r_{25}$
- $r_{22} = r_{22} + 12$
- $r_{24} = r_{24} + 12$
- if $(r_4 < 400)$ goto loop
- $r_6 = r_6 + r_{16} + r_{26}$