EECS 583 – Class 9
Classic + ILP Optimization

University of Michigan

September 24, 2025

Announcements & Reading Material

- Hopefully everyone is making some progress on HW 2
  - Due Oct 8
- Today's class
  - "Compiler Code Transformations for Superscalar-Based High-Performance Systems," S. Mahlke, W. Chen, J. Gyllenhaal, W. Hwu, P, Chang, and T. Kiyohara, *Proceedings of Supercomputing '92*, Nov. 1992, pp. 808-817
- Next class (code generation)
  - "Machine Description Driven Compilers for EPIC Processors", B. Rau, V. Kathail, and S. Aditya, HP Technical Report, HPL-98-40, 1998. (long paper but informative)

Course Project – Time to Start Thinking About This

- Mission statement: Design and implement something "interesting" in a compiler
  - LLVM preferred, but other compilers are fine
  - Groups of 3-5 people (other group sizes are possible in some cases)
  - Extend existing research paper or go out on your own
- Group size
  - Higher expectations for larger groups, but not strictly proportional to group size
    - 3-4 seems like a sweet spot
  - Everyone needs to have done something (rough work distribution)

Course Projects – Timetable

- Now - Start thinking about potential topics, identify group members
  - Use piazza to recruit group members
- Oct 20-24: Project proposal discussions, No class Oct 20/22, Regular class resumes Mon Oct 27
  - Naveen, Rishika, and I will meet with each group virtually for 5-10 mins, slot signups the week before, Oct 13-17
  - Ideas/proposal discussed at meeting – don't come into the meeting with too many ideas (1-2 only)
  - Short written proposal (a paragraph plus 1-2 references) due Mon, Oct 27 from each group, submit via email
- Nov 3 – End of semester: Research presentations (details later)
  - Each group presents a research paper related to their project (15 mins)
- Mid Nov - Optional quick discussion with groups on progress
- Dec 5 - 12: Project demos
  - Each group, 15 min slot - Presentation/Demo/whatever you like
  - Turn in short report on your project

Sample Project Ideas (Traditional)

- Memory system
  - Cache profiler for LLVM IR – miss rates, stride determination
  - Data cache prefetching, cache bypassing, scratch pad memories
  - Data layout for improved cache behavior
  - Advanced loads – move up to hide latency
- Reliability
  - Selective code duplication for soft error protection
  - Low-cost fault detection and/or recovery
  - Efficient soft error protection on GPUs/SIMD
- Energy
  - Minimizing instruction bit flips
  - Deactivate parts of processor (FUs, registers, cache)
  - Use different processors (e.g., big.LITTLE)

Sample Project Ideas (Traditional cont)

- Security/Safety
  - Efficient taint/information flow tracking
  - Automatic mitigation methods – obfuscation for side channels
  - Preventing control flow exploits
  - Rule compliance checking (driving rules for AV software)
  - Run-time safety verification
- Parallelization/SIMDization
  - DOALL loop parallelization, dependence breaking transformations
  - DSWP parallelization
  - Access-execute program decomposition
  - Automatic SIMDization, Superword level parallelism

More Project Ideas

- Dynamic optimization (Dynamo, LLVM)
  - Run-time optimization of frequent paths
  - Run-time program analysis for reliability/security
  - Run-time memory optimization (cache, memory dependence, etc.)
- High level synthesis
  - Custom instructions - finding most common instruction patterns, constrained by inputs/outputs
  - Int/FP precision analysis, Float to fixed point
  - Custom data path synthesis
  - Customized memory systems (e.g., sparse data structs)
- Approximate computing
  - New approximation optimizations (lookup tables, loop perforation)
  - Impact of local approximation on global program outcome
  - Program distillation - create a subset program with equivalent behavior

Sample Project Ideas (ML)

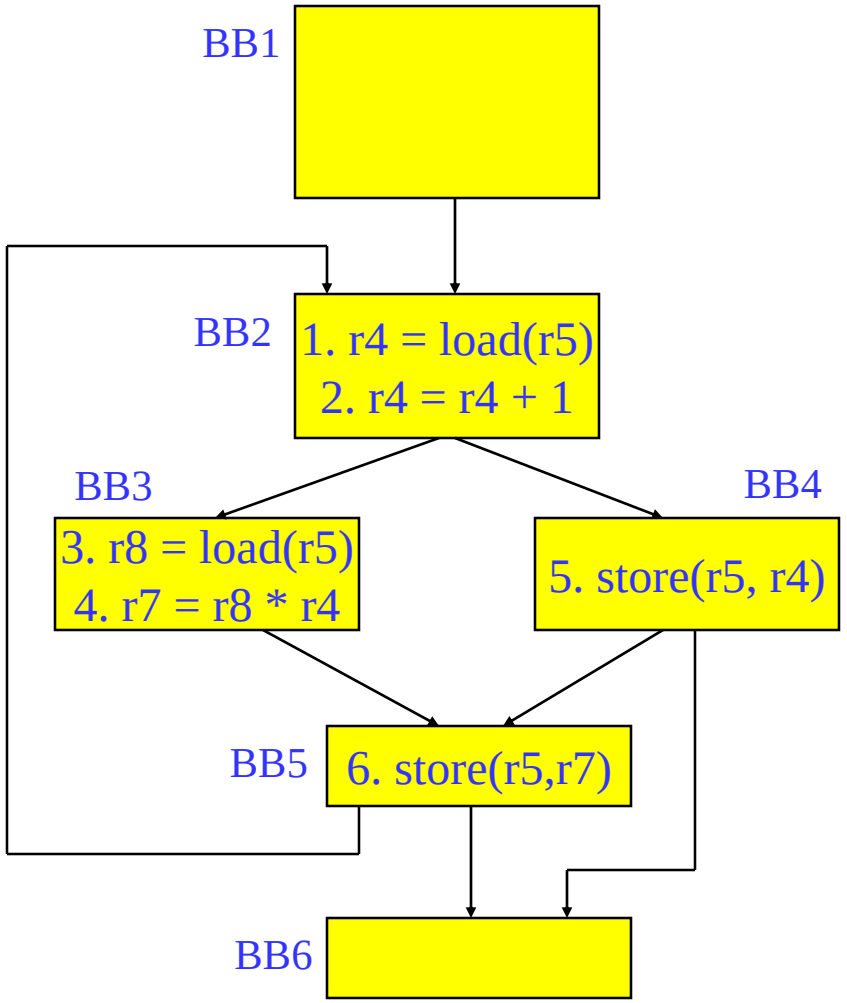
- ML for compilers
  - Using ML/search to guide optimizations (no unroll factors!!)
  - Using ML/search to guide optimization choices (which optis/order)
  - ML to infer profile data
  - ML to guide loop restructuring.
  - Be careful of low compiler content!!
- Compilers for ML/GPUs
  - Kernel fusion/restructuring
  - Kernel optimization (unrolling, tiling, etc.)
  - Reducing uncoalesced memory accesses – measurement of uncoalesced accesses, code restructuring to reduce these
  - Data restructuring
- Remember, this isn't a restaurant menu, don't just pick one of my suggestions, you can pick other topics!

Back to Code Optimization

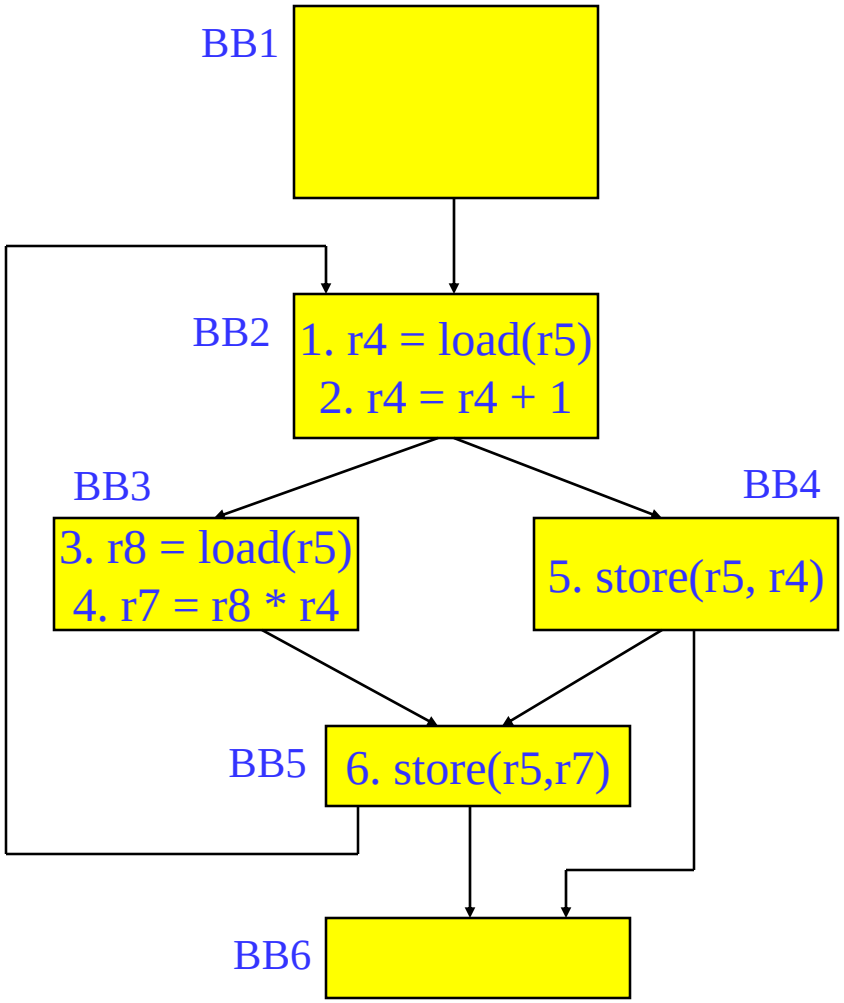
- Classical (machine independent, done at IR level)
  - Reducing operation count (redundancy elimination)
  - Simplifying operations
  - Generally good for any kind of machine
- We went through
  - Dead code elimination
  - Constant propagation
  - Constant folding
  - Copy propagation
  - CSE
  - LICM

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

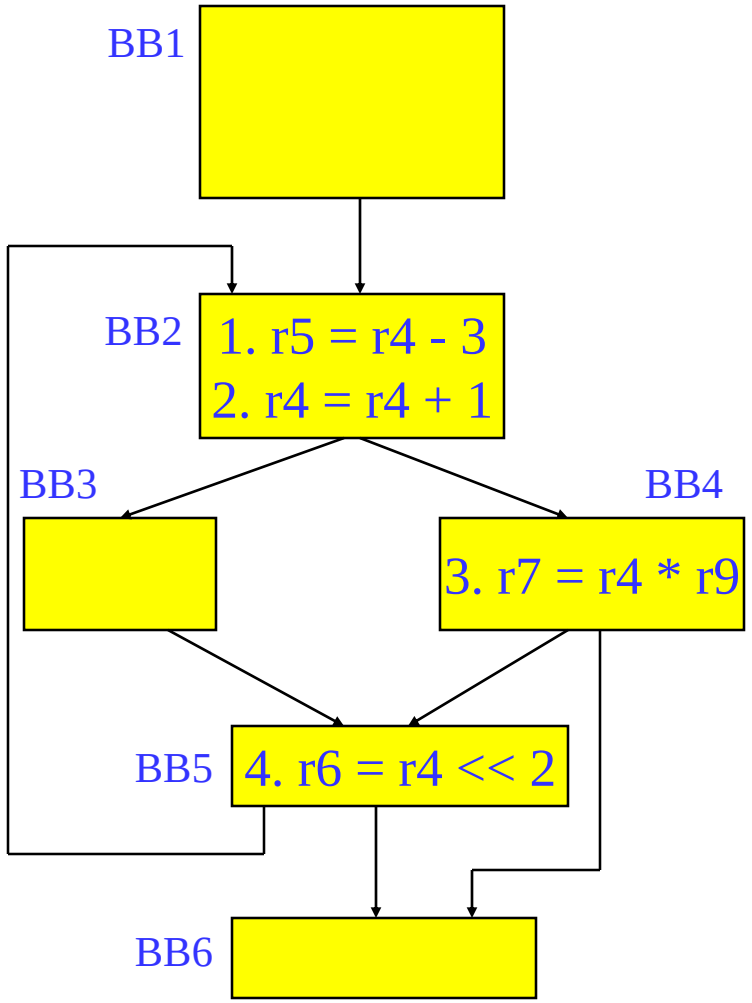


Global Variable Migration Example



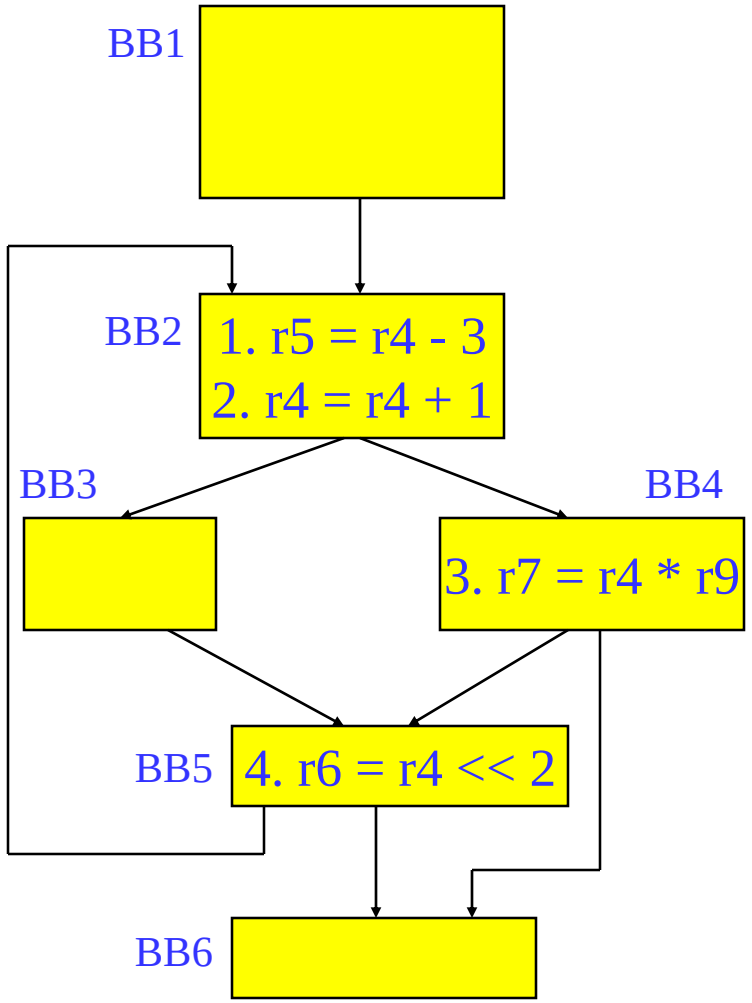
Induction Variable Strength Reduction

- Create basic induction variables from derived induction variables
- Induction variable
  - BIV (i++)
    - 0,1,2,3,4,...
  - DIV (j = i * 4)
    - 0, 4, 8, 12, 16, ...
  - DIV can be converted into a BIV that is incremented by 4
- Issues
  - Initial and increment vals
  - Where to place increments

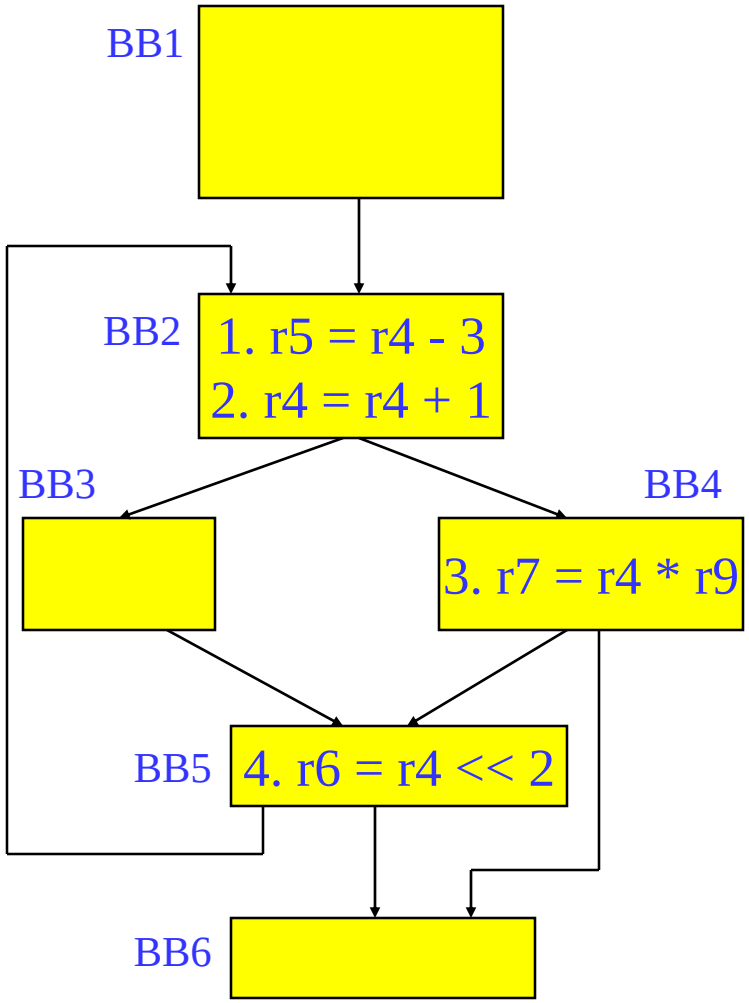


Induction Variable Strength Reduction (2)

- 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
- Transformation
  - Insert the following into the preheader
    - new_reg = RHS(X)
  - If opcode(X) is not add/sub, insert to the bottom of the preheader
    - new_inc = inc(src1(X)) opcode(X) src2(X)
  - else
    - new_inc = inc(src1(X))
  - Insert the following at each update of src1(X)
    - new_reg += new_inc
  - Change X → dest(X) = new_reg

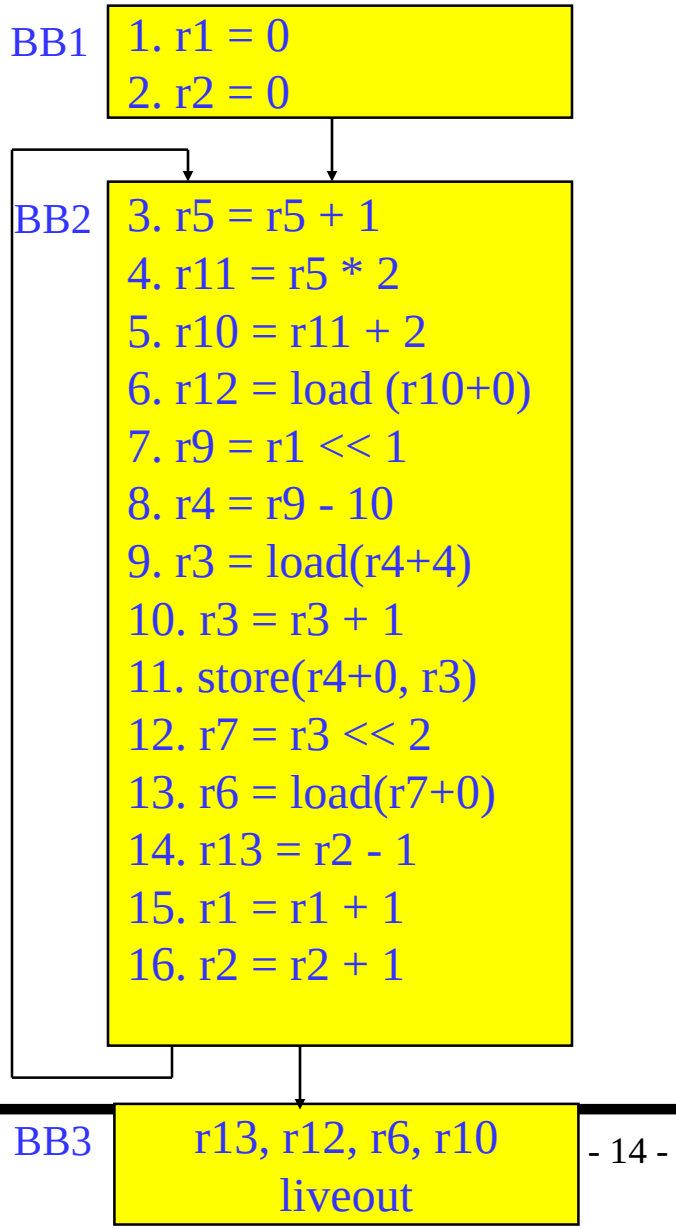


Induction Variable Strength Reduction - Example



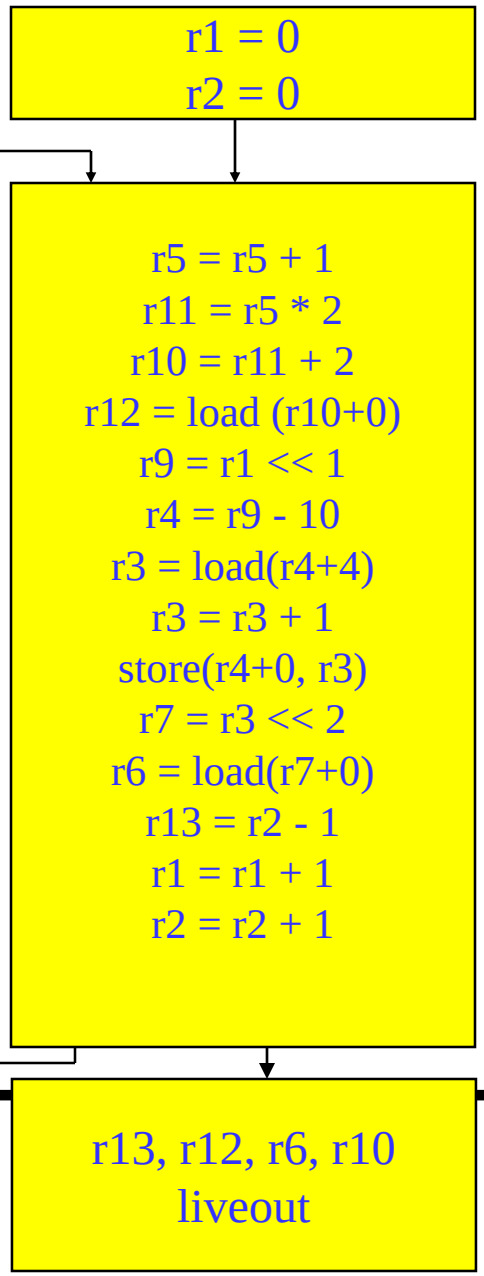
Class Problem

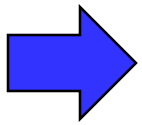
Optimize this applying induction var str reduction

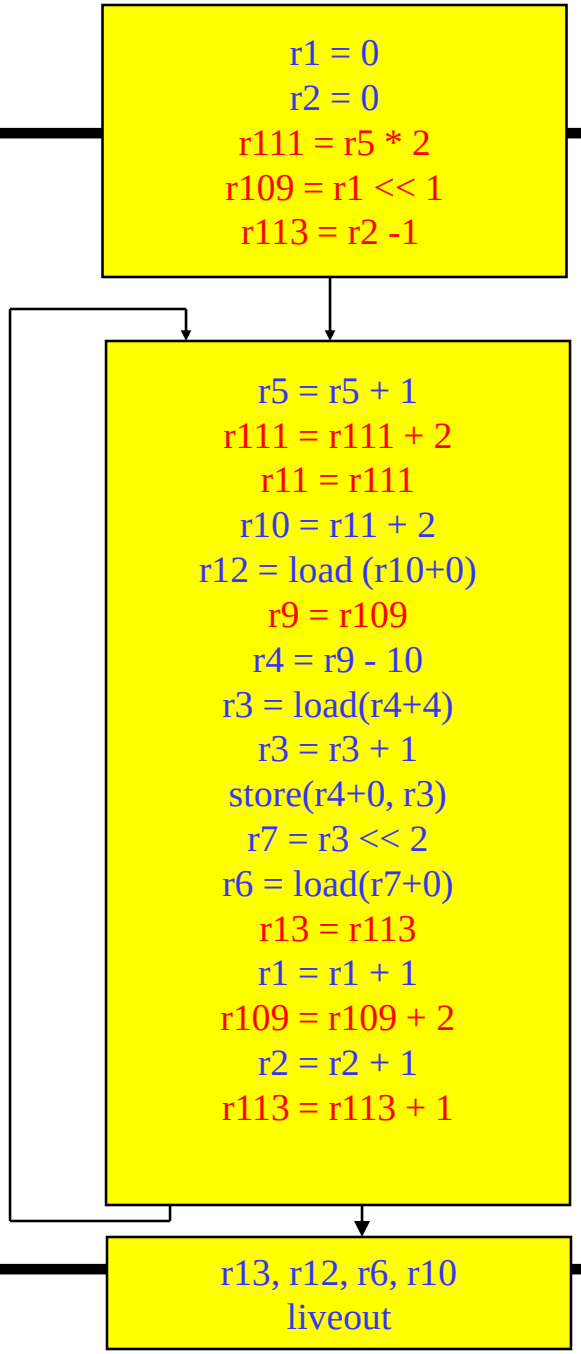


Class Problem Solution

Optimize this applying
induction var str reduction







Note, after copy propagation, r10 and r4 can be strength reduced as well.

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

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;

1. r9 = r1 + r2
2. r10 = r9 + r3
3. r11 = r10 - r4
4. r12 = r11 + r5
5. 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

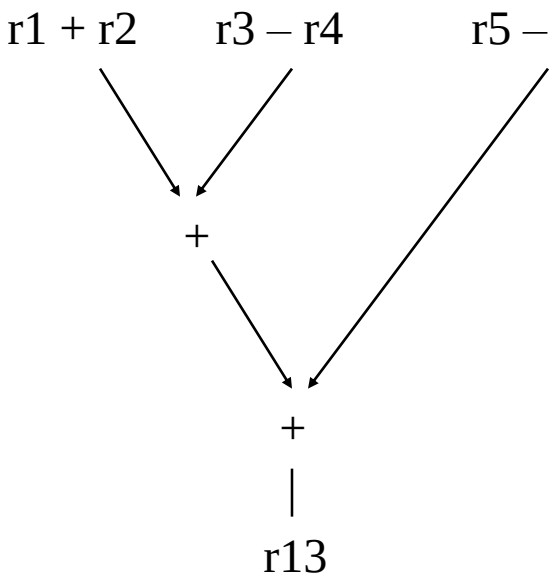
- Re-compute expression as a balanced binary tree
  - Obey precedence rules
  - Essentially re-parenthesize
  - Combine literals if possible
- Effects
  - Height reduced (n terms)
    - n-1 (assuming unit latency)
    - ceil(log2(n))
  - Number of operations remains constant
  - Cost
    - Temporary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

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
```

Class Problem

Assume: + = 1, * = 3

operand	0	0	0	1	2	0
arrival times	r1	r2	r3	r4	r5	r6

1. r10 = r1 * r2
2. r11 = r10 + r3
3. r12 = r11 + r4
4. r13 = r12 – r5
5. r14 = r13 + r6

Back susbstitute

Re-express in tree-height reduced form

Account for latency and arrival times

Class Problem Solution

Assume: + = 1, * = 3

operand	0	0	0	1	2	0
arrival times	r1	r2	r3	r4	r5	r6

1. r10 = r1 * r2
2. r11 = r10 + r3
3. r12 = r11 + r4
4. r13 = r12 – r5
5. r14 = r13 + r6

Back susbstitute

Re-express in tree-height reduced form

Account for latency and arrival times

Expression after back substitution
r14 = r1 * r2 + r3 + r4 - r5 + r6

Want to perform operations on r1,r2,r3,r6 first due to operand arrival times

t1 = r1 * r2
t2 = r3 + r6

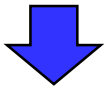
The multiply will take 3 cycles, so combine t2 with r4 and then r5, and then finally t1

t3 = t2 + r4
t4 = t3 – r5
r14 = t1 + t4

Equivalently, the fully parenthesized expression
r14 = ((r1 * r2) + (((r3 + r6) + r4) - r5))

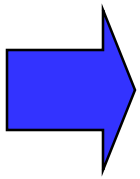
Loop Unrolling

```
for (i=x; i< 100; i++) {
   sum += a[i]*b[i];
}
```



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

unroll 3 times



```
loop:  r1 = load(r2)
       r3 = load(r4)
       r5 = r1 * r3
       r6 = r6 + r5
iter1  r2 = r2 + 4
       r4 = r4 + 4
       if (r4 >= 400) goto exit
       r1 = load(r2)
       r3 = load(r4)
       r5 = r1 * r3
iter2  r6 = r6 + r5
       r2 = r2 + 4
       r4 = r4 + 4
       if (r4 >= 400) goto exit
       r1 = load(r2)
       r3 = load(r4)
iter3  r5 = r1 * r3
       r6 = r6 + r5
       r2 = r2 + 4
       r4 = r4 + 4
       if (r4 < 400) goto loop
exit:
```

Unroll = replicate loop body n-1 times.

Hope to enable overlap of operation execution from different iterations

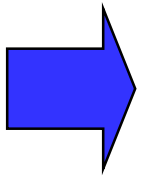
Smarter Loop Unrolling with Known Trip Count

Want to remove early exit branches

Trip count = 400/4 = 100

```
        r4 = 0
loop:   r1 = load(r2)
        r3 = load(r4)
        r5 = r1 * r3
        r6 = r6 + r5
        r2 = r2 + 4
        r4 = r4 + 4
        if (r4 < 400) goto loop
```

unroll multiple of trip count

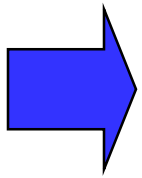


```
loop:   r1 = load(r2)
        r3 = load(r4)
        r5 = r1 * r3
iter1   r6 = r6 + r5
        r2 = r2 + 4
        r4 = r4 + 4
        ----------
        r1 = load(r2)
        r3 = load(r4)
        r5 = r1 * r3
iter2   r6 = r6 + r5
        r2 = r2 + 4
        r4 = r4 + 4
        ----------
        r1 = load(r2)
        r3 = load(r4)
        r5 = r1 * r3
iter3   r6 = r6 + r5
        r2 = r2 + 4
        r4 = r4 + 4
        ----------
        r1 = load(r2)
        r3 = load(r4)
        r5 = r1 * r3
iter4   r6 = r6 + r5
        r2 = r2 + 4
        r4 = r4 + 4
        if (r4 < 400) goto loop
exit:
```

What if the Trip Count is not Statically Known?

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

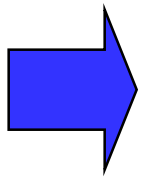
Create a preloop to ensure trip count of unrolled loop is a multiple of the unroll factor



```
preloop  for (i=0; i< ((400-r4)/4)%3; i++) {
           sum += a[i]*b[i];
         }
loop:    r1 = load(r2)
         r3 = load(r4)
iter1    r5 = r1 * r3
         r6 = r6 + r5
         r2 = r2 + 4
         r4 = r4 + 4
         ----------------------
         r1 = load(r2)
         r3 = load(r4)
iter2    r5 = r1 * r3
         r6 = r6 + r5
         r2 = r2 + 4
         r4 = r4 + 4
         ----------------------
         r1 = load(r2)
         r3 = load(r4)
         r5 = r1 * r3
iter3    r6 = r6 + r5
         r2 = r2 + 4
         r4 = r4 + 4
         if (r4 < 400) goto loop
exit:
```

Unrolling Not Enough for Overlapping Iterations: Register Renaming

```
loop:  r1 = load(r2)
       r3 = load(r4)
       r5 = r1 * r3
iter1  r6 = r6 + r5
       r2 = r2 + 4
       r4 = r4 + 4
       --------------------
       r1 = load(r2)
       r3 = load(r4)
       r5 = r1 * r3
iter2  r6 = r6 + r5
       r2 = r2 + 4
       r4 = r4 + 4
       --------------------
       r1 = load(r2)
       r3 = load(r4)
       r5 = r1 * r3
iter3  r6 = r6 + r5
       r2 = r2 + 4
       r4 = r4 + 4
       if (r4 < 400) goto loop
```



```
loop:  r1 = load(r2)
       r3 = load(r4)
       r5 = r1 * r3
iter1  r6 = r6 + r5
       r2 = r2 + 4
       r4 = r4 + 4
       --------------------
       r11 = load(r2)
       r13 = load(r4)
       r15 = r11 * r13
iter2  r6 = r6 + r15
       r2 = r2 + 4
       r4 = r4 + 4
       --------------------
       r21 = load(r2)
       r23 = load(r4)
       r25 = r21 * r23
iter3  r6 = r6 + r25
       r2 = r2 + 4
       r4 = r4 + 4
       if (r4 < 400) goto loop
```

Register Renaming is Not Enough!

```
loop:  r1 = load(r2)
       r3 = load(r4)
       r5 = r1 * r3
iter1  r6 = r6 + r5
       r2 = r2 + 4
       r4 = r4 + 4
       --------------------
       r11 = load(r2)
       r13 = load(r4)
       r15 = r11 * r13
iter2  r6 = r6 + r15
       r2 = r2 + 4
       r4 = r4 + 4
       --------------------
       r21 = load(r2)
       r23 = load(r4)
       r25 = r21 * r23
iter3  r6 = r6 + r25
       r2 = r2 + 4
       r4 = r4 + 4
       if (r4 < 400) goto loop
```

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

Accumulator Variable Expansion

```
        r16 = r26 = 0
loop:   r1 = load(r2)
        r3 = load(r4)
        r5 = r1 * r3
iter1   r6 = r6 + r5
        r2 = r2 + 4
        r4 = r4 + 4
        --------------------
        r11 = load(r2)
        r13 = load(r4)
        r15 = r11 * r13
iter2   r16 = r16 + r15
        r2 = r2 + 4
        r4 = r4 + 4
        --------------------
        r21 = load(r2)
        r23 = load(r4)
        r25 = r21 * r23
iter3   r26 = r26 + r25
        r2 = r2 + 4
        r4 = r4 + 4
        if (r4 < 400) goto loop
exit:   r6 = r6 + r16 + r26
```

- 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

```
        r12 = r2 + 4, r22 = r2 + 8
        r14 = r4 + 4, r24 = r4 + 8
        r16 = r26 = 0
loop:   r1 = load(r2)
        r3 = load(r4)
        r5 = r1 * r3
iter1   r6 = r6 + r5
        r2 = r2 + 12
        r4 = r4 + 12
        --------------------
        r11 = load(r12)
        r13 = load(r14)
        r15 = r11 * r13
iter2   r16 = r16 + r15
        r12 = r12 + 12
        r14 = r14 + 12
        --------------------
        r21 = load(r22)
        r23 = load(r24)
        r25 = r21 * r23
iter3   r26 = r26 + r25
        r22 = r22 + 12
        r24 = r24 + 12
        if (r4 < 400) goto loop
exit:   r6 = r6 + r16 + r26
```

- 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 !!

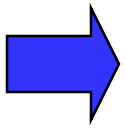
Better Induction Variable Expansion

```
        r16 = r26 = 0
loop:   r1 = load(r2)
        r3 = load(r4)
        r5 = r1 * r3
iter1   r6 = r6 + r5

        --------------------
        r11 = load(r2+4)
        r13 = load(r4+4)
        r15 = r11 * r13
iter2   r16 = r16 + r15

        --------------------
        r21 = load(r2+8)
        r23 = load(r4+8)
        r25 = r21 * r23
iter3   r26 = r26 + r25
        r2 = r2 + 12
        r4 = r4 + 12
        if (r4 < 400) goto loop
exit:   r6 = r6 + r16 + r26
```

- 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

Homework Problem

```
loop:
r1 = load(r2)
r5 = r6 + 3
r6 = r5 + r1
r2 = r2 + 4
if (r2 < 400) goto loop
```



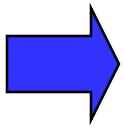
```
loop:
r1 = load(r2)
r5 = r6 + 3
r6 = r5 + r1
r2 = r2 + 4
r1 = load(r2)
r5 = r6 + 3
r6 = r5 + r1
r2 = r2 + 4
r1 = load(r2)
r5 = r6 + 3
r6 = r5 + r1
r2 = r2 + 4
if (r2 < 400) goto loop
```

Optimize the unrolled loop

Renaming
Tree height reduction
Ind/Acc expansion

Homework Problem - Answer

```
loop:
r1 = load(r2)
r5 = r6 + 3
r6 = r5 + r1
r2 = r2 + 4
if (r2 < 400) goto loop
```



```
loop:
r1 = load(r2)
r5 = r6 + 3
r6 = r5 + r1
r2 = r2 + 4
r1 = load(r2)
r5 = r6 + 3
r6 = r5 + r1
r2 = r2 + 4
r1 = load(r2)
r5 = r6 + 3
r6 = r5 + r1
r2 = r2 + 4
if (r2 < 400)
   goto loop
```

Optimize the unrolled loop

Renaming
Tree height reduction
Ind/Acc expansion

```
loop:
  r1 = load(r2)
  r5 = r1 + 3
  r6 = r6 + r5
  r2 = r2 + 4
  r11 = load(r2)
  r15 = r11 + 3
  r6 = r6 + r15
  r2 = r2 + 4
  r21 = load(r2)
  r25 = r21 + 3
  r6 = r6 + r25
  r2 = r2 + 4
  if (r2 < 400)
     goto loop
```

after renaming and tree height reduction

```
r16 = r26 = 0
loop:
  r1 = load(r2)
  r5 = r1 + 3
  r6 = r6 + r5
  r11 = load(r2+4)
  r15 = r11 + 3
  r16 = r16 + r15
  r21 = load(r2+8)
  r25 = r21 + 3
  r26 = r26 + r25
  r2 = r2 + 12
  if (r2 < 400)
     goto loop
r6 = r6 + r16
r6 = r6 + r26
```

after acc and ind expansion