

EECS 583 – Class 9

ILP Optimization

University of Michigan

September 29, 2021

Announcements & Reading Material

- ❖ Hopefully everyone is making some progress on HW 2
- ❖ 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 others are fine
 - » Groups of 2-4 people (1 or 5 persons is possible in some cases)
 - » Extend existing research paper or go out on your own
- ❖ Topic areas (Not in any priority order)
 - » Automatic parallelization/SIMDization
 - » High level synthesis/FPGAs
 - » Approximate computing
 - » Memory system optimization
 - » Reliability
 - » Energy
 - » Security
 - » Dynamic optimization
 - » Optimizing for GPUs

Course Projects – Timetable

- ❖ Now
 - » Start thinking about potential topics, identify group members
- ❖ Oct 18-25: Project discussions, No class Oct 18 (Mon, Fall Break), Oct 20 (Wed), Oct 25 (Mon)
 - » GSIs and I will meet with each group virtually for 10 mins, slot signups the week before Oct 11-15
 - » Ideas/proposal discussed at meeting
 - » Short written proposal (a paragraph plus some references) due Wed, Oct 27 from each group, submit via email
- ❖ Nov 8 – End of semester: Research presentations
 - » Each group presents a research paper related to their project (15 mins) – details later
- ❖ Mid-Late Nov - Optional quick discussion with groups on progress
- ❖ Dec 9-16: 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

❖ Control/Dataflow optimization

- » Superblock formation
- » Make an LLVM optimization smarter with profile data
- » Implement optimization not in LLVM

❖ Reliability

- » AVF profiling, vulnerability analysis
- » Selective code duplication for soft error protection
- » Low-cost fault detection and/or recovery
- » Efficient soft error protection on GPUs/SIMD

Sample Project Ideas (Traditional cont)

❖ Energy

- » Minimizing instruction bit flips
- » Deactivate parts of processor (FUs, registers, cache)
- » Use different processors (e.g., big.LITTLE)

❖ 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

❖ Dealing with pointers

- » Memory dependence analysis – try to improve on LLVM
- » Using dependence speculation for optimization or code reordering

Sample Project Ideas (Parallelism)

❖ Optimizing for GPUs

- » Dumb OpenCL/CUDA → smart OpenCL/CUDA – selection of threads/blocks and managing on-chip memory
- » Reducing uncoalesced memory accesses – measurement of uncoalesced accesses, code restructuring to reduce these
- » Matlab → CUDA/OpenCL
- » Kernel partitioning across multiple GPUs

❖ Parallelization/SIMDization

- » DOALL loop parallelization, dependence breaking transformations
- » DSWP parallelization
- » Access-execute program decomposition

More Project Ideas

- ❖ Dynamic optimization (Dynamo, LLVM, Dalvik VM)
 - » Run-time DOALL loop parallelization
 - » Run-time program analysis for reliability/security
 - » Run-time profiling tools (cache, memory dependence, etc.)
- ❖ Binary optimizer
 - » Arm binary to LLVM IR, de-register allocation
- ❖ 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)

And Yet a Few More

- ❖ Approximate computing
 - » New approximation optimizations (lookup tables, loop perforation, tiling)
 - » Impact of local approximation on global program outcome
 - » Program distillation - create a subset program with equivalent memory/branch behavior
- ❖ Machine learning
 - » Using ML/search to guide optimizations (e.g., unroll factors)
 - » Using ML/search to guide optimization choices (which optis/order)
- ❖ Remember, don't be constrained by my suggestions, you can pick other topics!

From Last Time: 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

Class Problem From Last Time - Solution

Optimize this applying

1. constant propagation

2. constant folding

BB1

1. r1 = 0
2. r2 = 10
3. r3 = 0

BB2

4. r4 = 1
5. r7 = r1 * 4
6. r6 = 8
7. if (r3 > 0)

BB3

8. r2 = 0
9. r6 = r6 * r7
10. r3 = r2 / r6

BB4

11. r3 = r4
12. r3 = r3 + r2
13. r1 = r6

BB5

14. r2 = r2 + 1
15. r1 = r1 + 1
16. if (r1 < 100)

BB6

17. store (r1, r3)



BB1

1. r1 = 0
2. r2 = 10
3. r3 = 0

BB2

4. r4 = 1
5. r7 = r1 * 4
6. r6 = 8
7. if (r3 > 0)

BB3

8. r2 = 0
9. r6 = r7 << 3
10. r3 = 0

BB4

11. r3 = 1
12. r3 = 1 + r2
13. r1 = 8

BB5

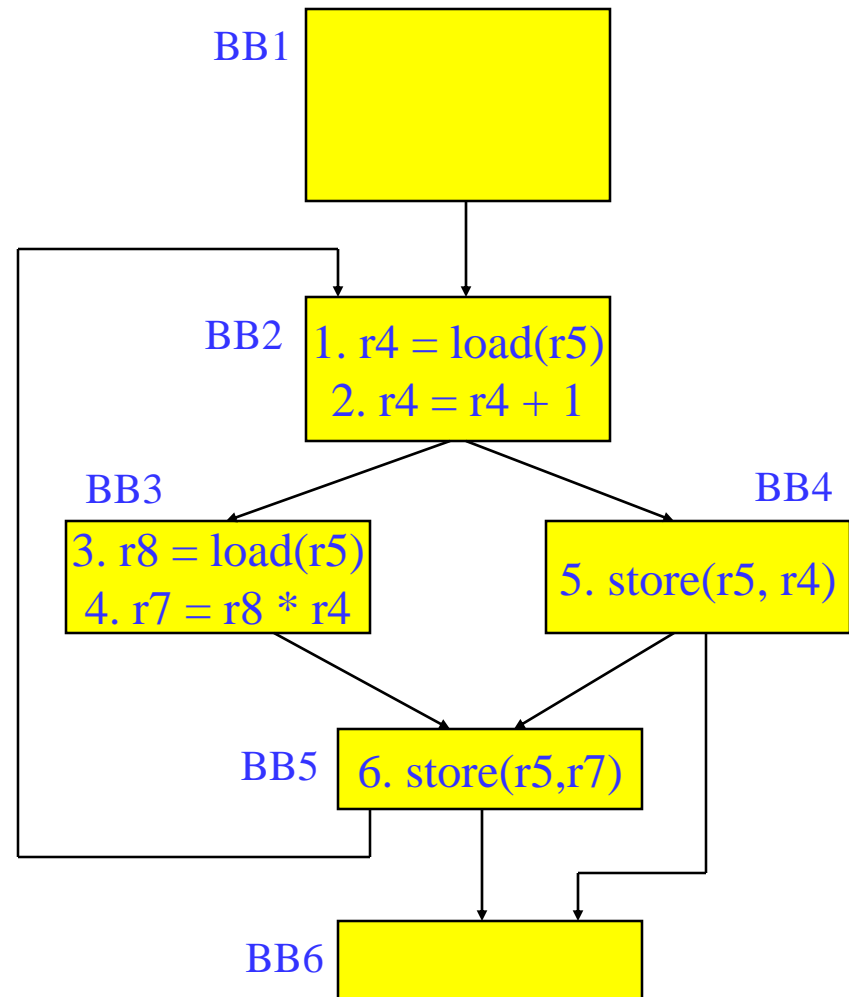
14. r2 = r2 + 1
15. r1 = r1 + 1
16. if (r1 < 100)

BB6

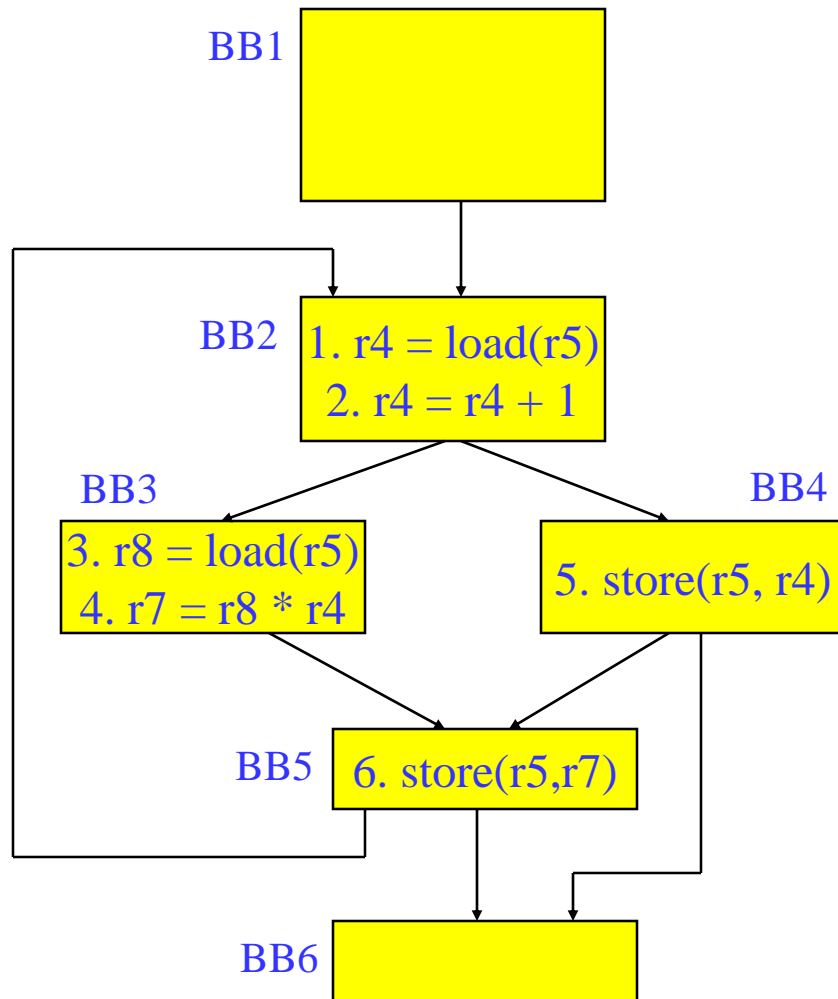
17. store (r1, r3)

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

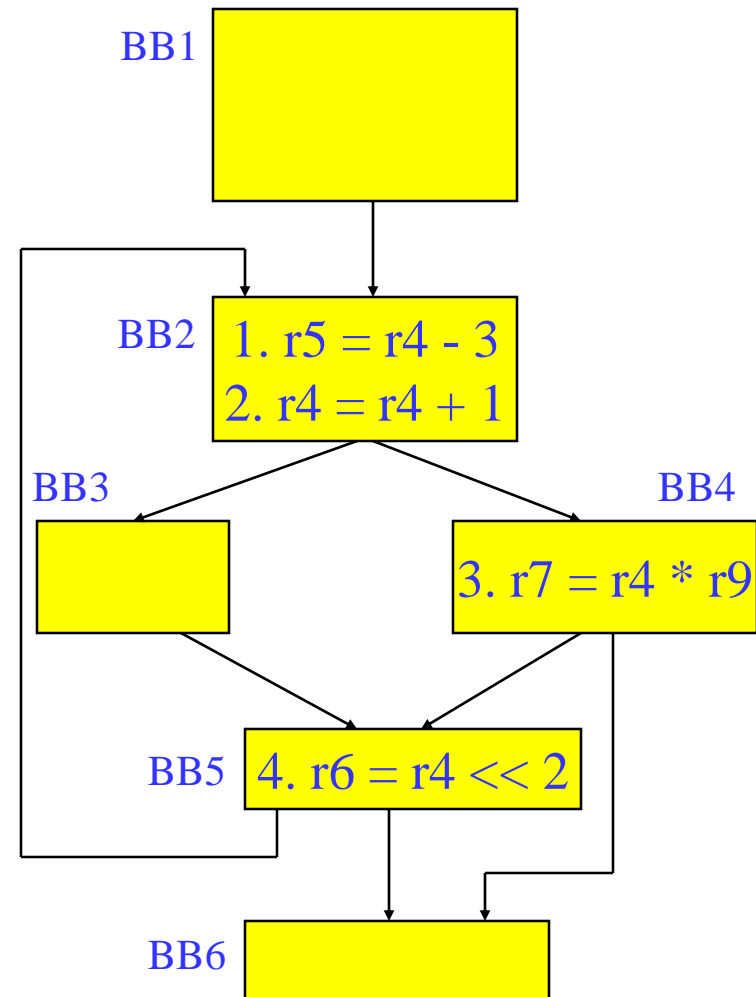


Global Variable Migration Example



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



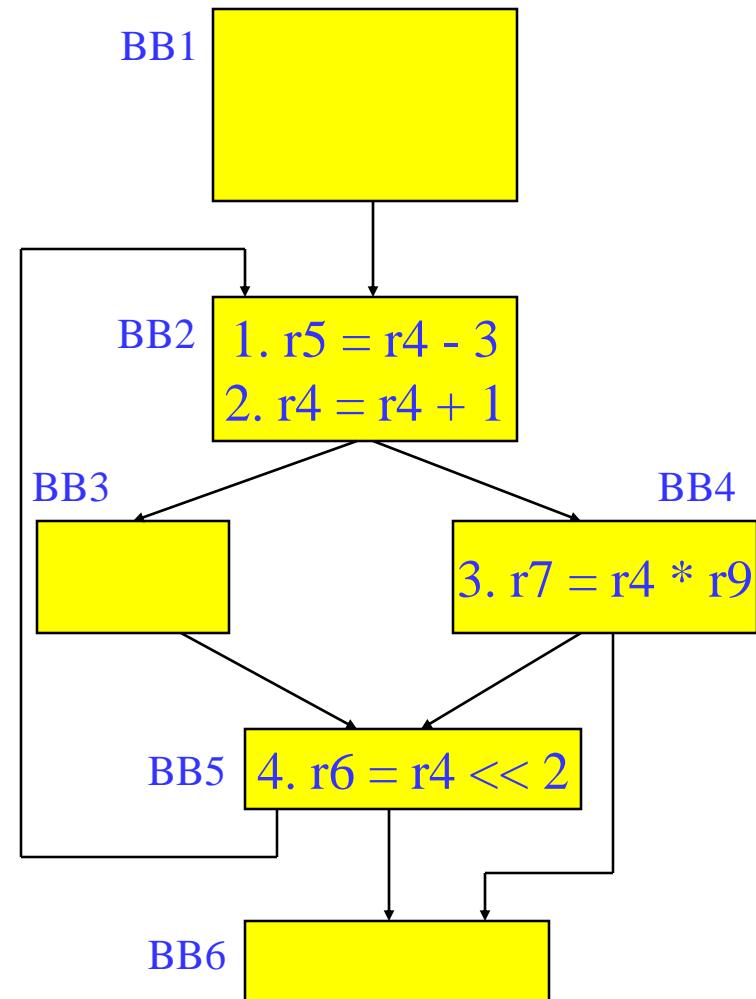
Induction Variable Strength Reduction (2)

❖ Rules

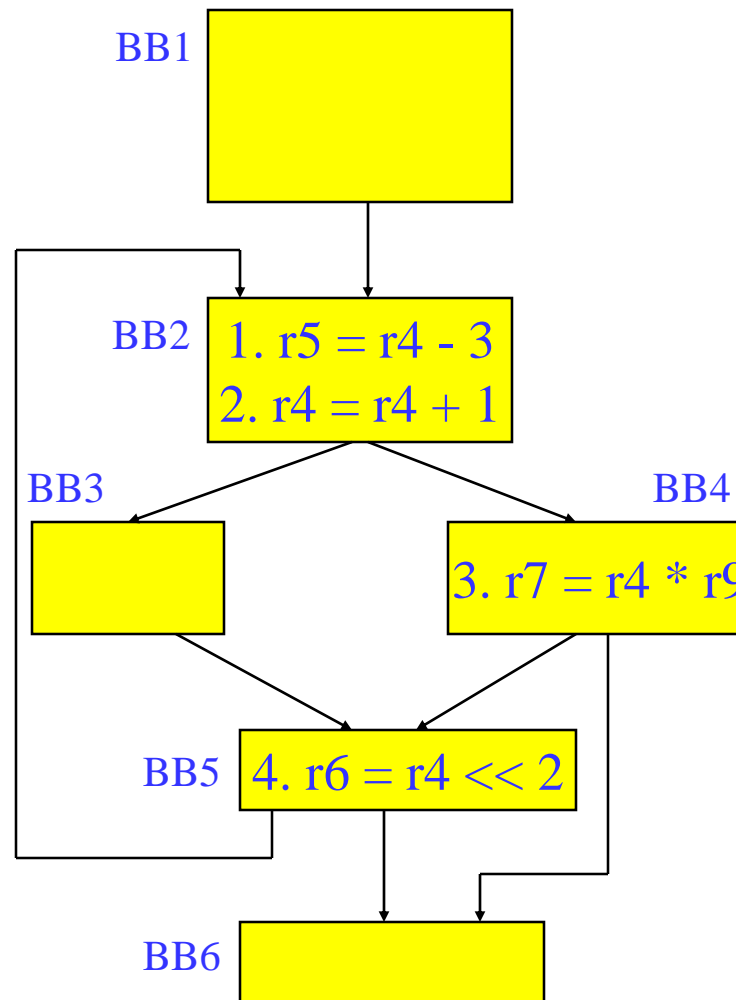
- » X is a *, <<, + or - operation
- » $\text{src1}(X)$ is a basic ind var
- » $\text{src2}(X)$ is invariant
- » No other ops modify $\text{dest}(X)$
- » $\text{dest}(X) \neq \text{src}(X)$ for all srcs
- » $\text{dest}(X)$ is a register

❖ Transformation

- » Insert the following into the preheader
 - $\text{new_reg} = \text{RHS}(X)$
- » If $\text{opcode}(X)$ is not add/sub, insert to the bottom of the preheader
 - $\text{new_inc} = \text{inc}(\text{src1}(X)) \text{ opcode}(X) \text{ src2}(X)$
- » else
 - $\text{new_inc} = \text{inc}(\text{src1}(X))$
- » Insert the following at each update of $\text{src1}(X)$
 - $\text{new_reg} += \text{new_inc}$
- » Change $X \rightarrow \text{dest}(X) = \text{new_reg}$



Induction Variable Strength Reduction - Example



Class Problem

Optimize this applying
induction var str
reduction

BB1

1. r1 = 0
2. r2 = 0

BB2

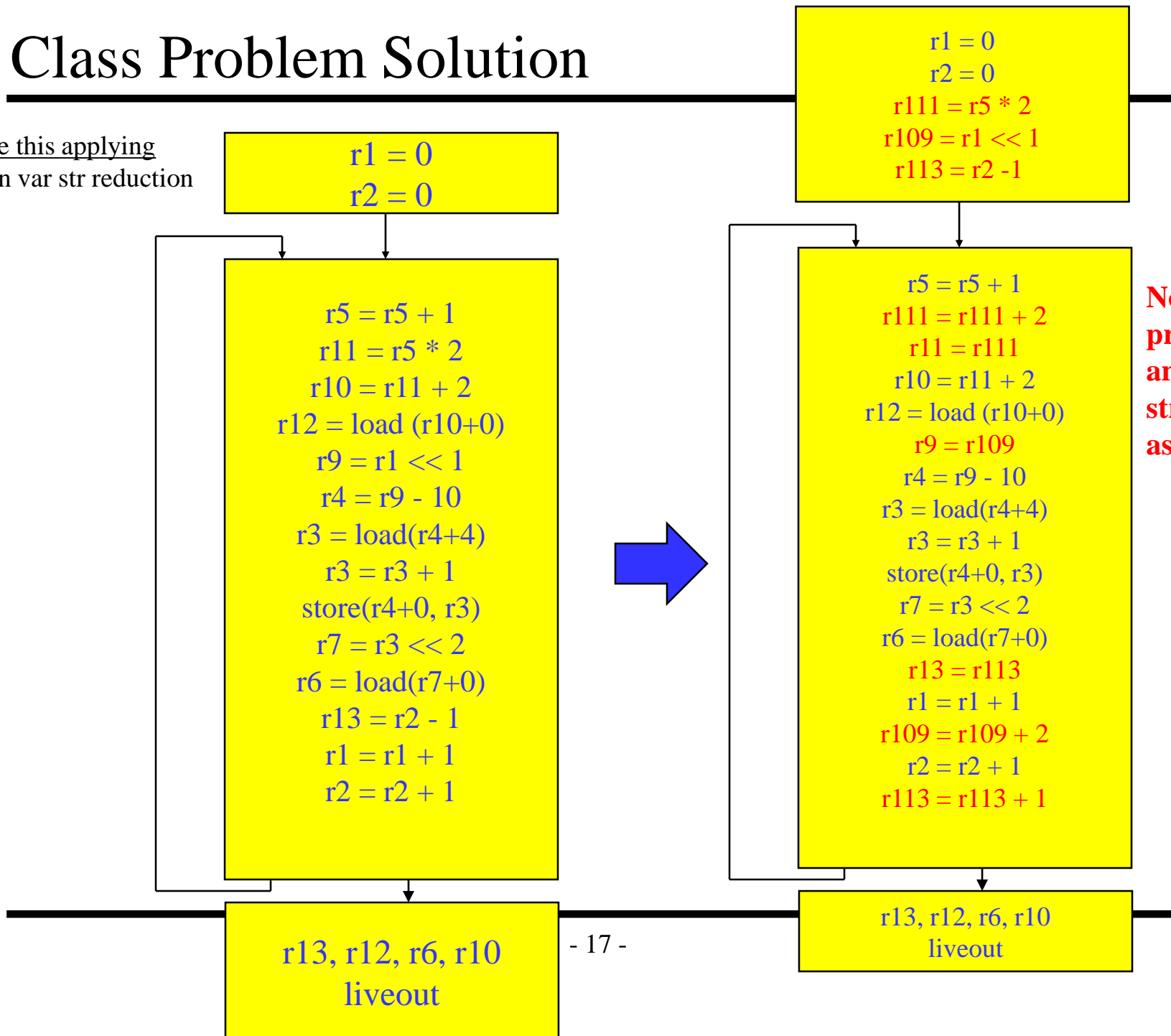
3. r5 = r5 + 1
4. r11 = r5 * 2
5. r10 = r11 + 2
6. r12 = load(r10+0)
7. r9 = r1 << 1
8. r4 = r9 - 10
9. r3 = load(r4+4)
10. r3 = r3 + 1
11. store(r4+0, r3)
12. r7 = r3 << 2
13. r6 = load(r7+0)
14. r13 = r2 - 1
15. r1 = r1 + 1
16. r2 = r2 + 1

BB3

r13, r12, r6, r10
liveout

Class Problem Solution

Optimize this applying
induction var str reduction



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

$r13 = r1 + r2 + r3 - r4 + r5 - r6$

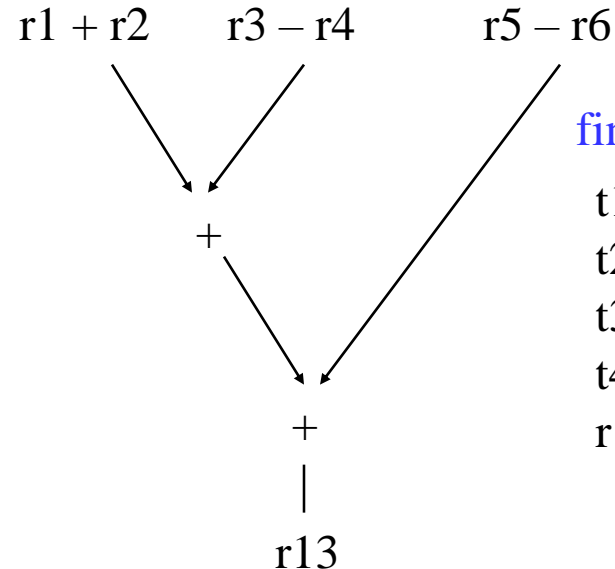
Tree Height Reduction

- ❖ 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)
 - $\text{ceil}(\log_2(n))$
 - » 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 substitute

Re-express in tree-height reduced form

Account for latency and arrival times

Loop Unrolling

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

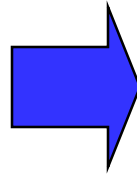


```
loop:  r1 = load(r2)  
       r3 = load(r4)  
       r5 = r1 * r3  
       r6 = r6 + r5  
       r2 = r2 + 4  
       r4 = r4 + 4  
       if (r4 < 400) goto loop
```

Unroll = replicate loop body
n-1 times.

Hope to enable overlap of
operation execution from
different iterations

unroll 3 times



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

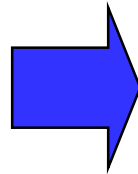
Smarter Loop Unrolling with Known Trip Count

Want to remove early exit branches

Trip count = $400/4 = 100$

```
loop:  r4 = 0
      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

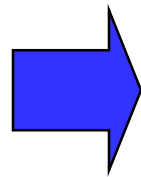


```
loop:  r1 = load(r2)
      r3 = load(r4)
      r5 = r1 * r3
      r6 = r6 + r5
      r2 = r2 + 4
      r4 = r4 + 4
      -----
      r1 = load(r2)
      r3 = load(r4)
      r5 = r1 * r3
      r6 = r6 + r5
      r2 = r2 + 4
      r4 = r4 + 4
      -----
      r1 = load(r2)
      r3 = load(r4)
      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
exit:
```


What if the Trip Count is not Statically Known?

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

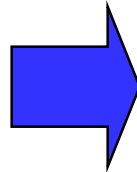
Create a preloop to
ensure trip count of
unrolled loop is a multiple
of the unroll factor



```
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)  
iter3    r5 = r1 * r3  
        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)
iter2  r15 = r11 * r13
      r16 = r16 + r15
      r12 = r12 + 12
      r14 = r14 + 12
      -----
      r21 = load(r22)
      r23 = load(r24)
iter3  r25 = r21 * r23
      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 , $\text{init} + \text{step}$, $\text{init} + 2 * \text{step}$, etc.
- ❖ Step increased to $n * \text{original step}$
- ❖ 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)
iter2  r15 = r11 * r13
      r16 = r16 + r15

-----
      r21 = load(r2+8)
      r23 = load(r4+8)
iter3  r25 = r21 * r23
      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:

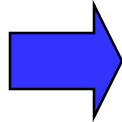
r1 = load(r2)

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