18-747 Lecture 17: Advanced ILP Scheduling

James C. Hoe
Dept of ECE, CMU
October 31, 2001

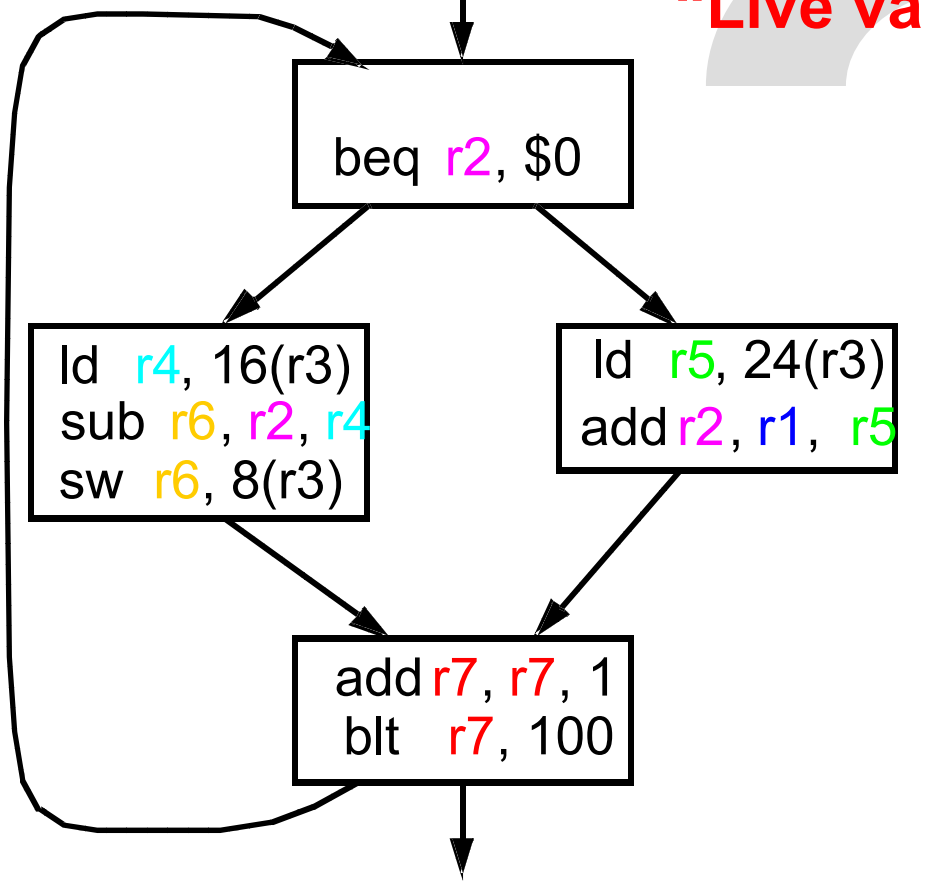
Reading Assignments: MJ Ch11

Announcements: Last chance for Quiz 1 re-grade requests

Handouts:

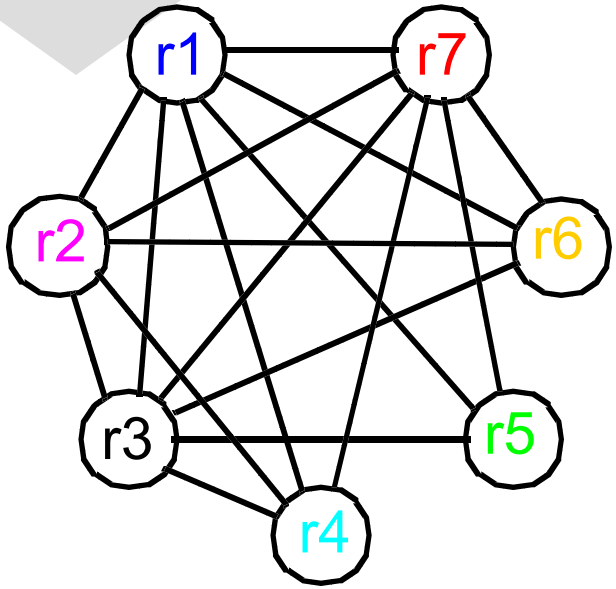
Interference Graph

r1, r2 & r3 are live-in



r1& r3 are live-out

"Live variable analysis"



Nodes: live ranges

Edges: interference

Register Interference & Allocation

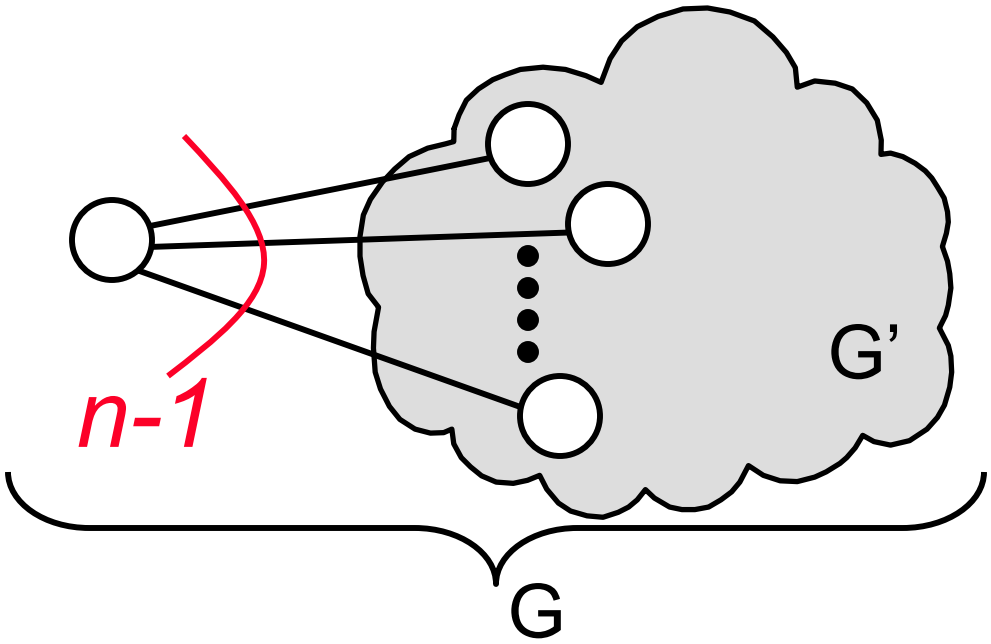
- Interference Graph: G = <E,V>
  - Nodes (V) = variables, (more specifically, their live ranges)
  - Edges (E) = interference between variable live ranges
- Graph Coloring (vertex coloring)
  - Given a graph, G=<E,V>, assign colors to nodes (V) so that **no** two adjacent (connected by an edge) nodes have the same color
  - A graph can be "*n*-colored" if no more than *n* colors are needed to color the graph.
  - The chromatic number of a graph is *min{n}* such that it can be *n*-colored
  - *n*-coloring is an NP-complete problem, therefore optimal solution can take a long time to compute

How is graph coloring related to register allocation?

Chaitin's Graph Coloring Theorem

- Key observation: If a graph G has a node X with degree less than *n* (i.e. having less than *n* edges connected to it), then G is *n*-colorable *IFF* the reduced graph G' obtained from G by deleting X and all its edges is *n*-colorable.

Proof:



Graph Coloring Algorithm *(Not Optimal)*

- Assume the register interference graph is *n*-colorable

  How do you choose n?

- Simplification
  - Remove all nodes with degree less than *n*
  - Repeat until the graph has *n* nodes left
- Assign each node a different color
- Add removed nodes back one-by-one and pick a legal color as each one is added (2 nodes connected by an edge get different colors)

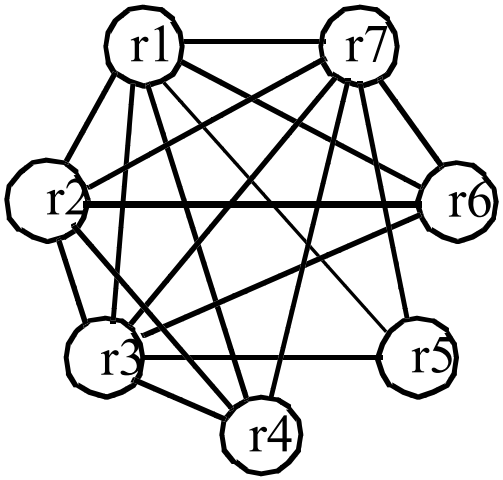
  Must be possible with less than n colors

- *Complications:* simplification can block if there are no nodes with less than *n* edges

  Choose one node to spill based on spilling heuristic

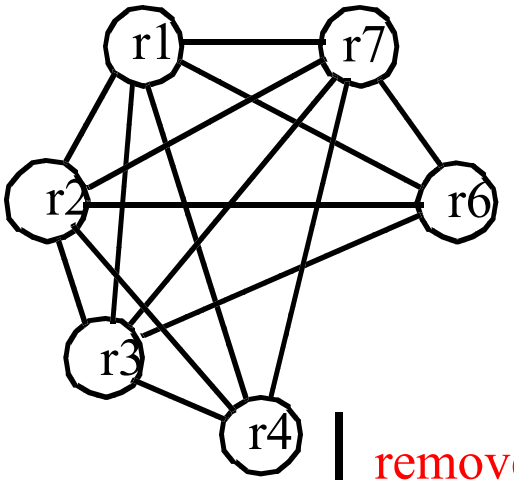
Example (N = 5)

COLOR stack = {}

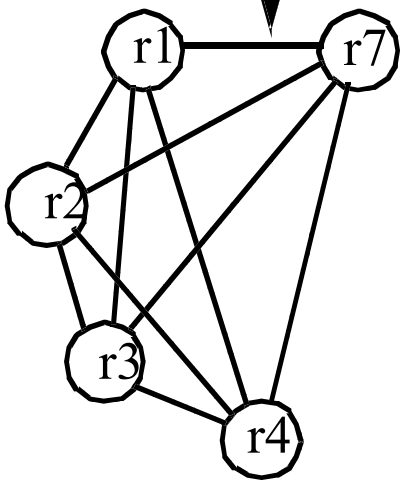


remove r5

COLOR stack = {r5}

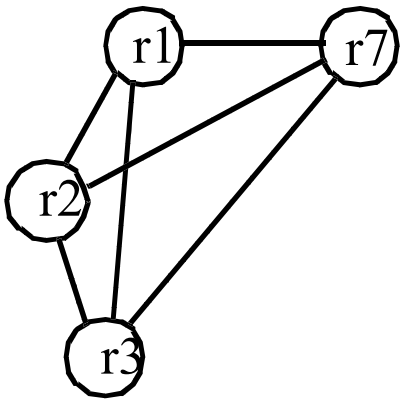


remove r6



COLOR stack = {r5, r6}

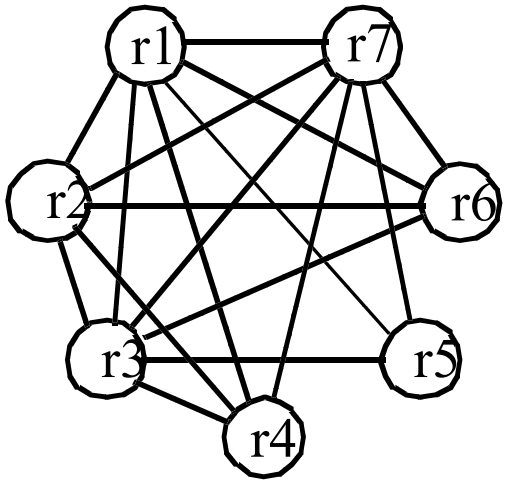
remove r4



COLOR stack = {r5, r6, r4}

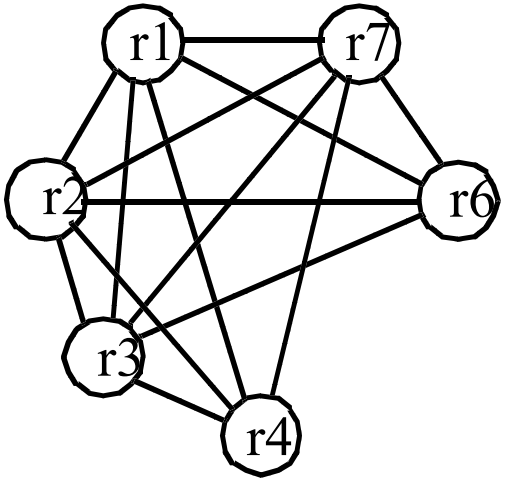
Example (N = 4)

COLOR stack = {}



remove r5

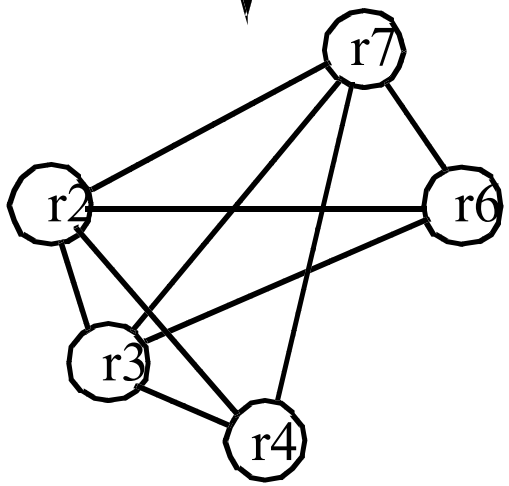
COLOR stack = {r5}



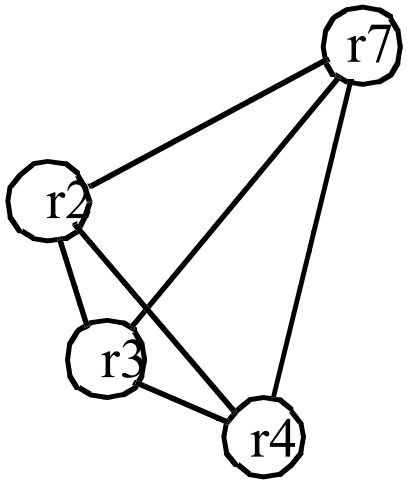
blocks

spill r1

Is this a good choice??



remove r6



COLOR stack = {r5, r6}

COLOR stack = {r5}

Register Spilling

- When simplification is blocked, pick a node to delete from the graph in order to unblock
- Deleting a node implies the variable it represents will not be kept in register (i.e. spilled into memory)
  - When constructing the interference graph, each node is assigned a value indicating the estimated cost to spill it.
  - The estimated cost can be a function of the total number of definitions and uses of that variable weighted by its estimated execution frequency.
  - When the coloring procedure is blocked, the node with the least spilling cost is picked for spilling.
- When a node is spilled, spill code is added into the original code to store a spilled variable at its definition and to reload it at each of its use
- After spill code is added, a new interference graph is rebuilt from the modified code, and n-coloring of this graph is again attempted

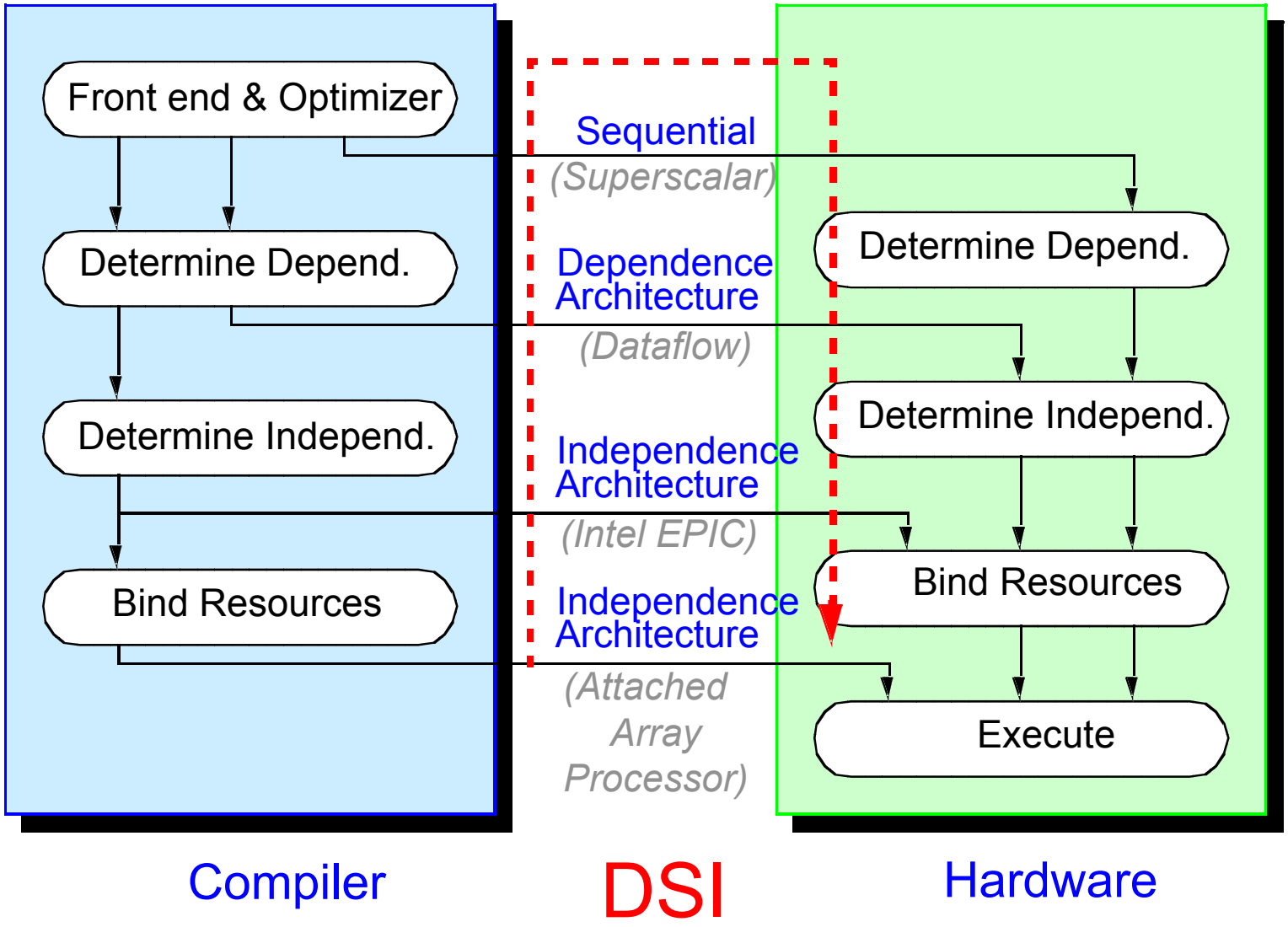
Phase Ordering Problem

- Register allocation prior to code scheduling
  - false dependencies induced due to register reuse
  - anti and output dependencies impose unnecessary constraints
  - code motion unnecessarily limited

- Code scheduling prior to register allocation
  - increase date live time (between creation and consumption)
  - overlap otherwise disjoint live ranges (increase register pressure)
  - may cause more live ranges to spill (run out of registers)
  - spill code produced will not have been scheduled

One option: do both prepass and postpass scheduling.

Compiler/Hardware Interactions



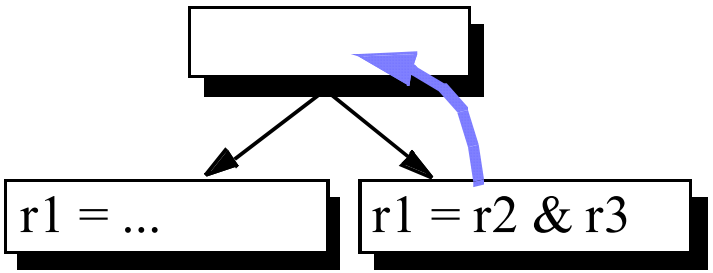
[B. Rau & J. Fisher, 1993]

Limitations of List Scheduling

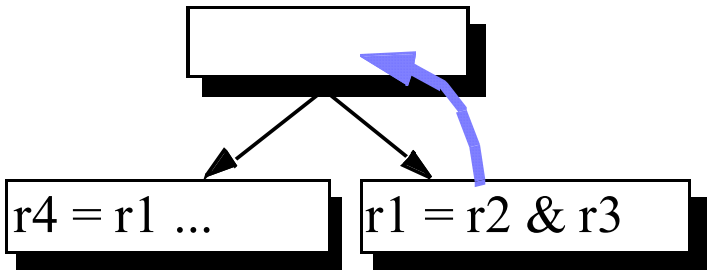
- Cannot move instructions past conditional branch instructions in the program (scheduling limited by basic block boundaries)
- Problem: Many programs have small numbers of instructions (4-5) in each basic block. Hence, not much code motion is possible
- Solution: Allow code motion across basic block boundaries.
- Speculative Code Motion: “jumping the gun”
  - Execute instructions before we know whether or not we need to
  - Utilize otherwise idle resources to perform work which we speculate will need to be done
- Relies on program profiling to make intelligent decisions about speculation

Types of Speculative Code Motion

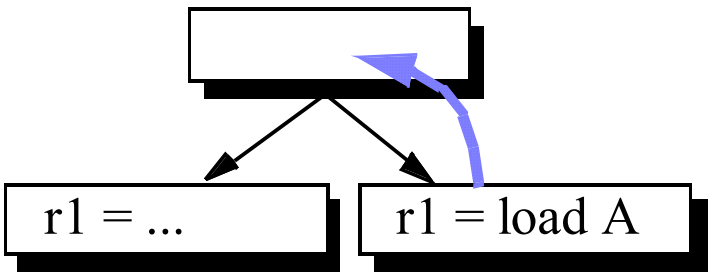
- Two characteristics of speculative code motion:
  - safety, which indicates whether or not spurious exceptions may occur
  - legality, which indicates correctness of results
- Four possible types of code motion:



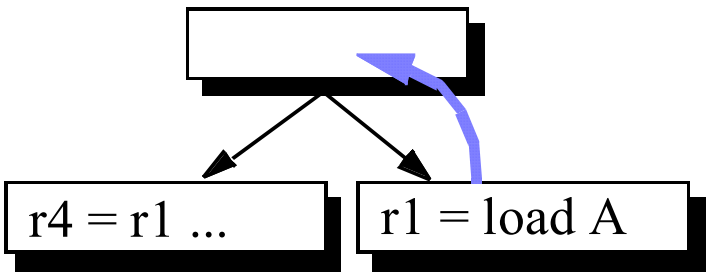
(a) safe and legal



(b) illegal



(c) unsafe



(d) unsafe and illegal

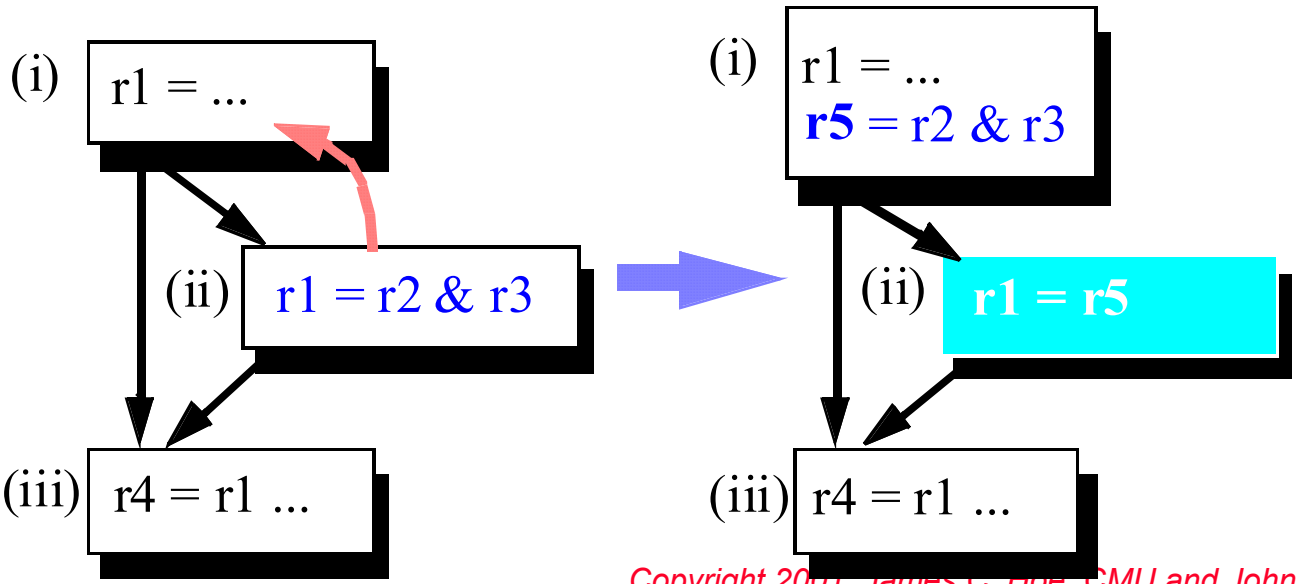
Register Renaming

- Prevents boosted instructions from overwriting register state needed on alternate execution path.
- Utilizes idle (non-live) registers (r6 in example below).

BB#	Original Code			Scheduled Code		
n		load	r4= ...		load	r4= ...
		load	r5= ...		load	r5= ...
		cmpi	c0,r4,10		cmpi	c0,r4,10
		add	r4=r4+r5		add	r4=r4+r5
		<stall>			sub	r3=r7-r4
		<stall>			and	**r6**=r3&r5
		bc	c0, A1		bc	c0, A1
n+1		st	... =r4		st	... =r4
n+2	A1:	sub	r3=r7-r4	A1:	st	... =**r6**
		and	r4=r3&r5			
		st	... =r4			

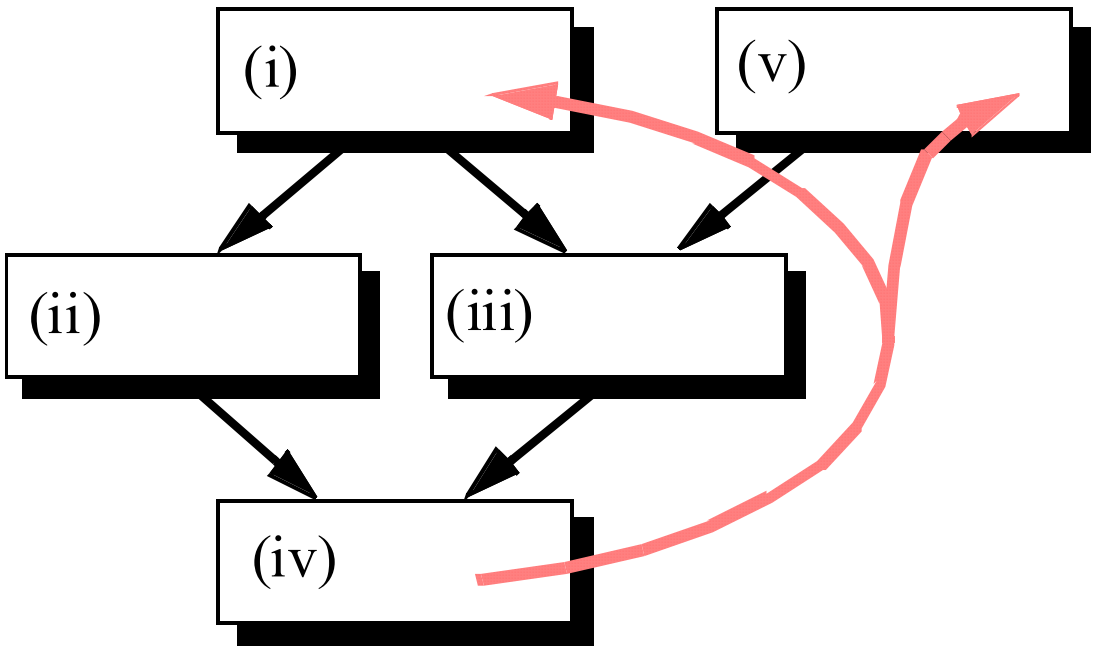
Copy Creation

- Register renaming causes a problem when there are multiple definitions of a register reaching a single use:
  - Below, definitions of r1 in both (i) and (ii) reach the use in (iii).
  - If the instruction in (ii) is boosted into (i), it must be renamed to preserve the first value of r1.
  - However, the boosted definition of r1 must reach the use in (iii) as well.
  - Hence, we insert a copy instruction in (ii).



Instruction Replication

- General case of upward code motion: crossing control flow joins.
- Instructions must be present on each control flow path to their original basic block
- Replicate set is computed for each basic block that is a source of instructions to be boosted



Profile Driven Optimizations

- Wrong optimization choices can be costly!

  How do you determine dynamic information during compilation?

- During initial compilation, "extra code" can be added to a program to generate profiling statistics when the program is executed
- Execution Profile, e.g.
  - how many times is a basic block executed
  - how often is a branch taken vs. not taken
- Recompile the program using the profile to guide optimization choices
- A profile is associated with a particular program input

  $\Rightarrow$ *may not work well on all executions*

Trace Scheduling *[Josh Fisher]*

- Generate multi-basic block traces based on profiling information
  - find the most often executed control path
- List schedule a trace at a time
  - optimize the execution of the trace *(common case)*
  - fix any problem with off-trace paths as necessary *(infrequently executed)*
- Good for very biased and predictable branching behavior
- Trace scheduling engendered the VLIW architecture innovation and was implemented in the Multiflow TRACE compiler, which provided the basis for superscalar compilation techniques now being used by Intel, HP, and DEC

Trace Scheduling Overview

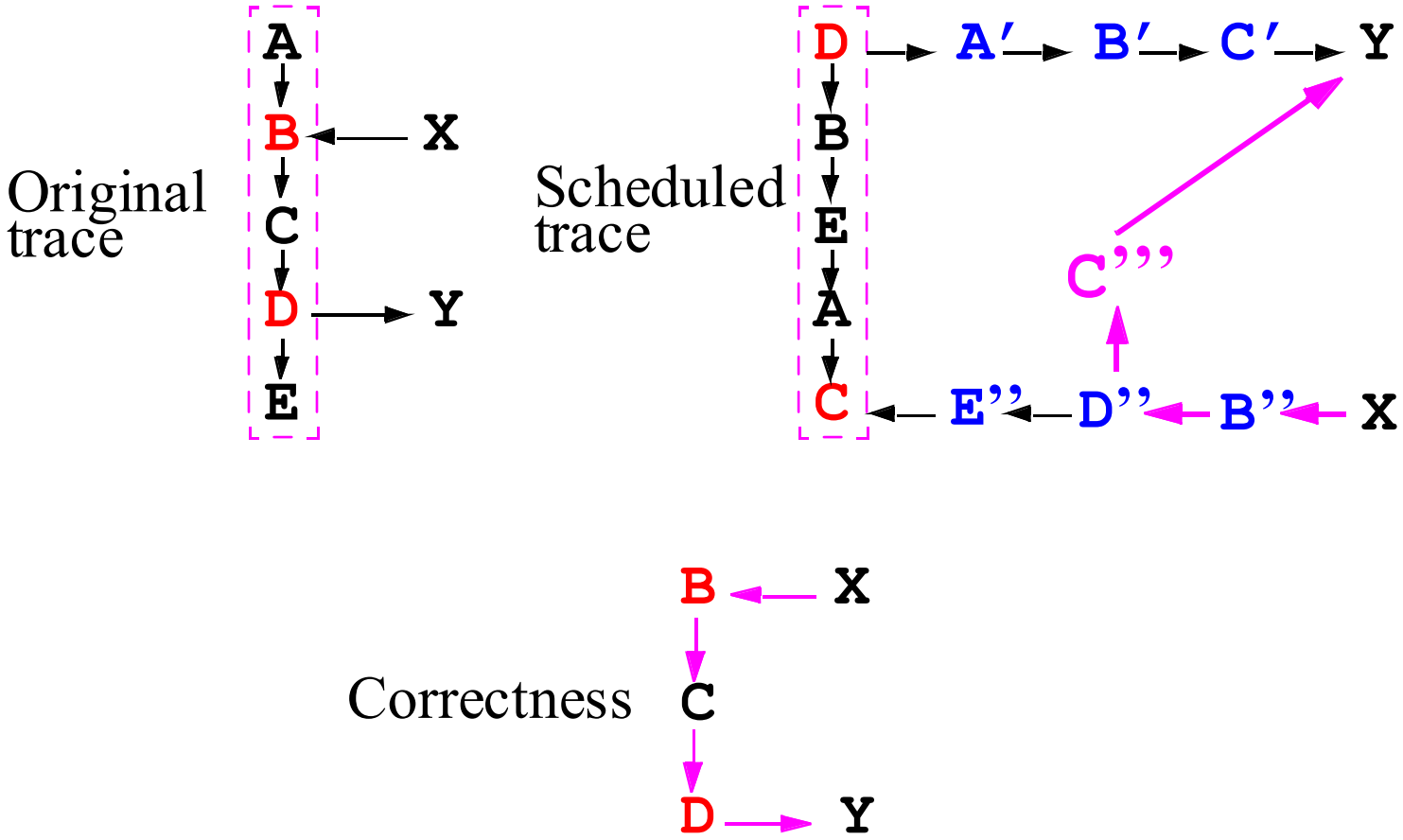
- Trace Selection
  - select seed (the highest frequency basic block)
  - extend trace (along the highest frequency edges)
    forward (successor of the last block of the trace)
    backward (predecessor of the first block of the trace)
  - don't cross loop back edge
  - bound max_trace_length heuristically
- Trace Scheduling
  - build data precedence graph for a whole trace
  - perform list scheduling and allocate registers
  - add compensation code to maintain semantic correctness
- Speculative Code Motion (upward)
  - Move an instruction above branches if safe

Compensation Code for Downward Motion

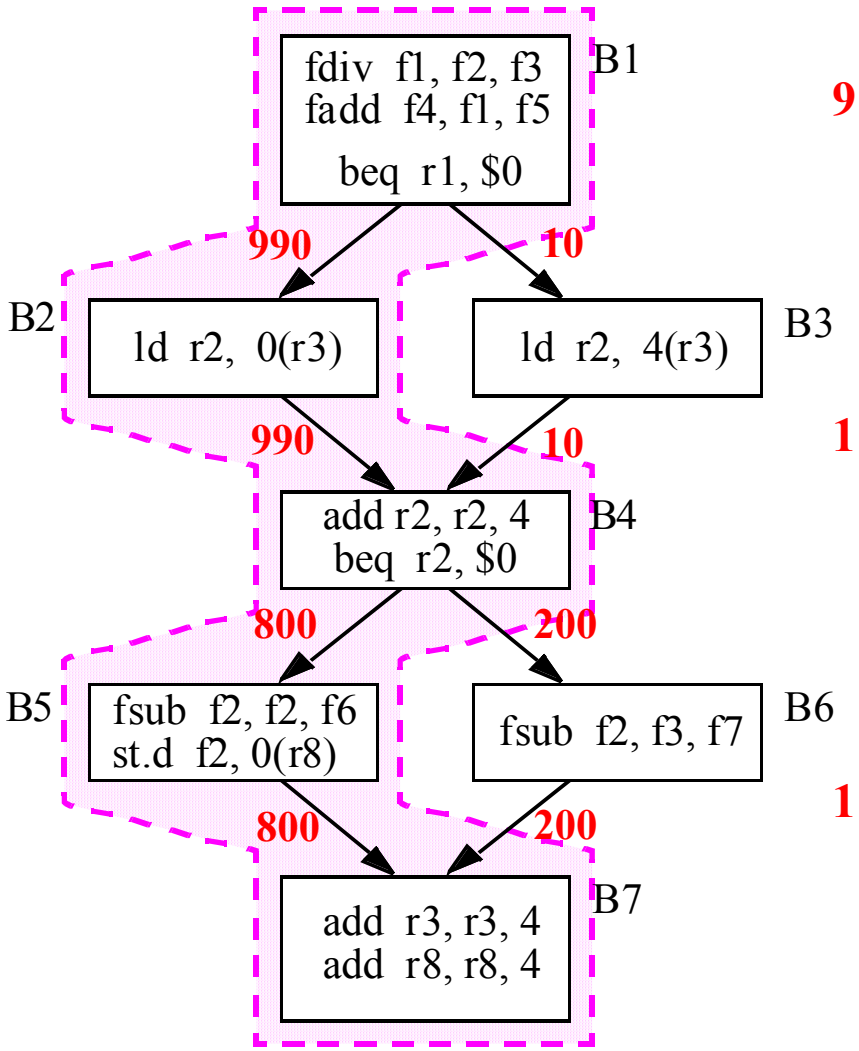
- Split Compensation Code:
  - Instruction with more than one successor (conditional branch)

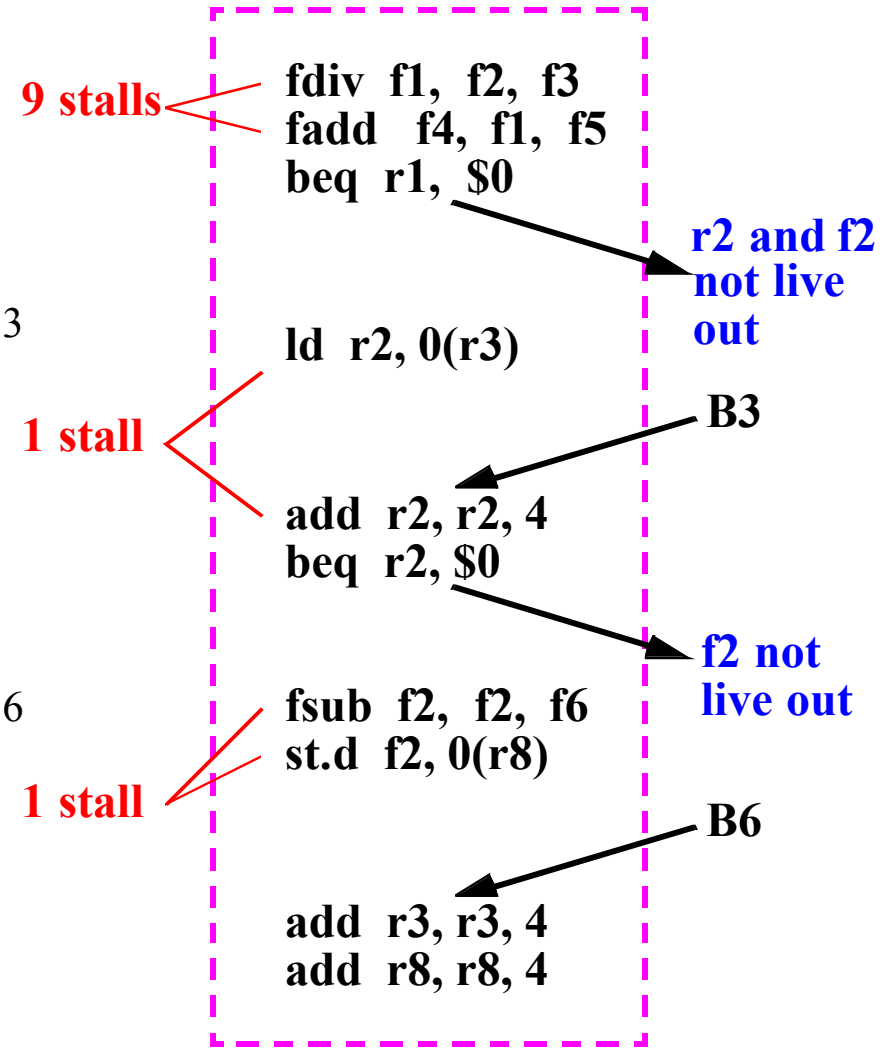
- Join Compensation Code:
  - Instruction with more than one predecessor

Copied Split Instruction

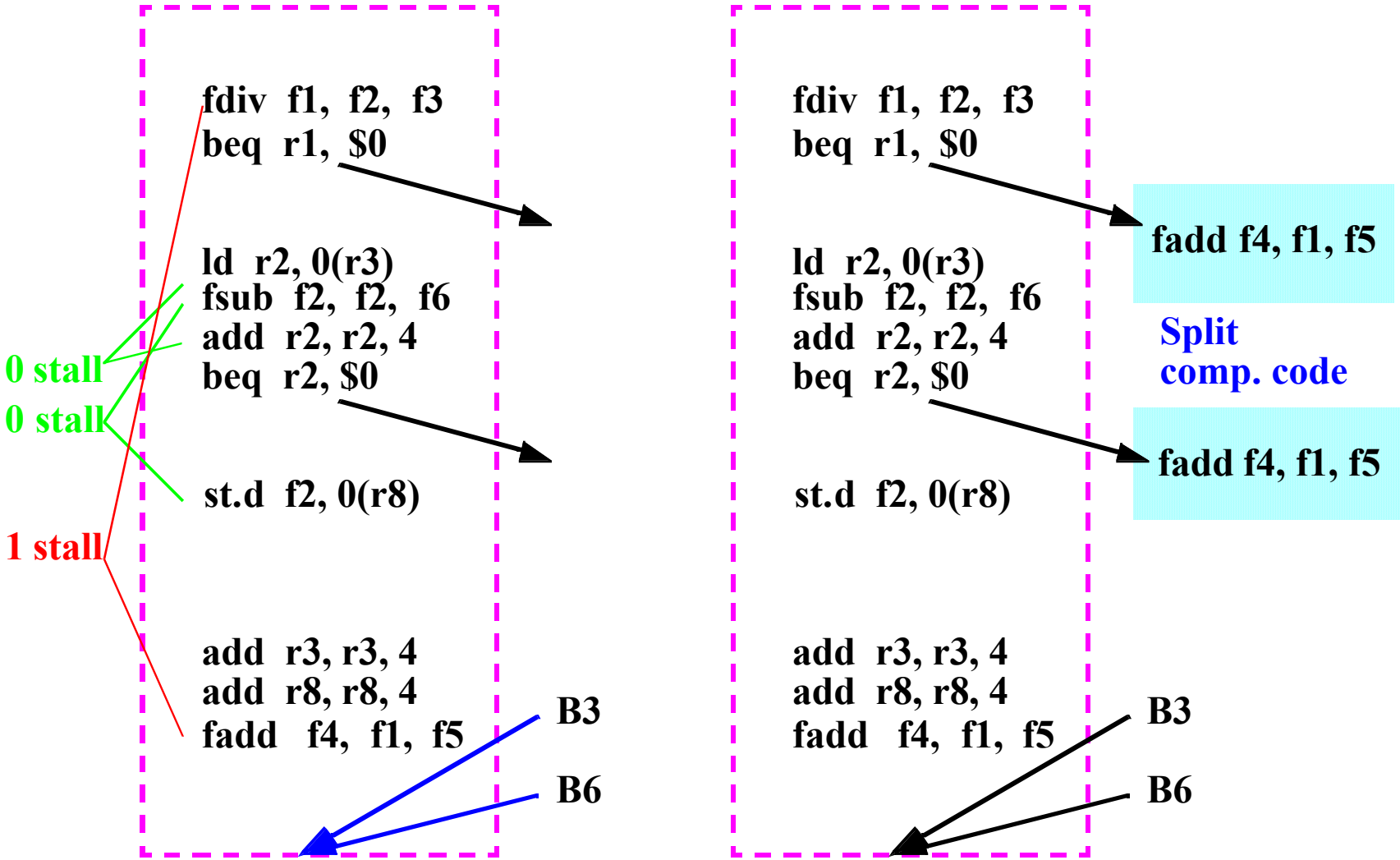


Trace Scheduling Example

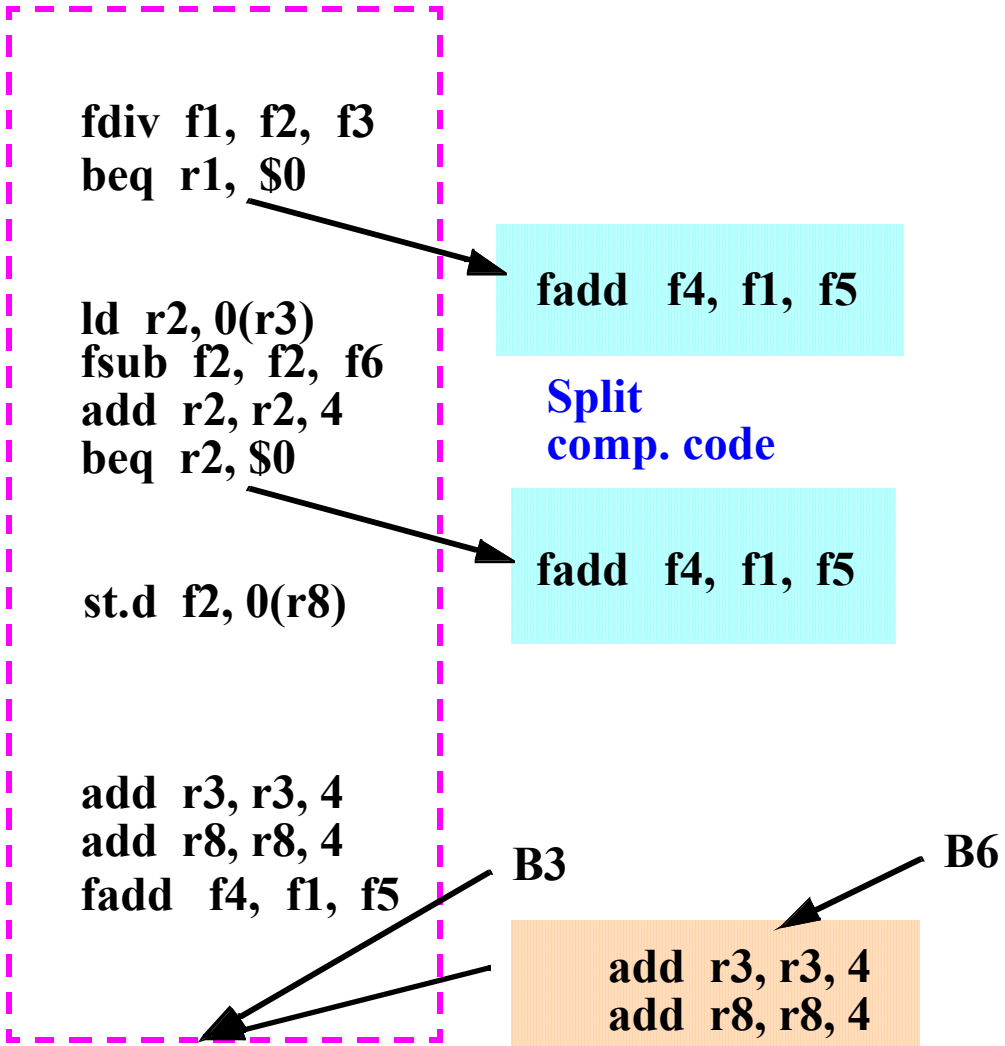




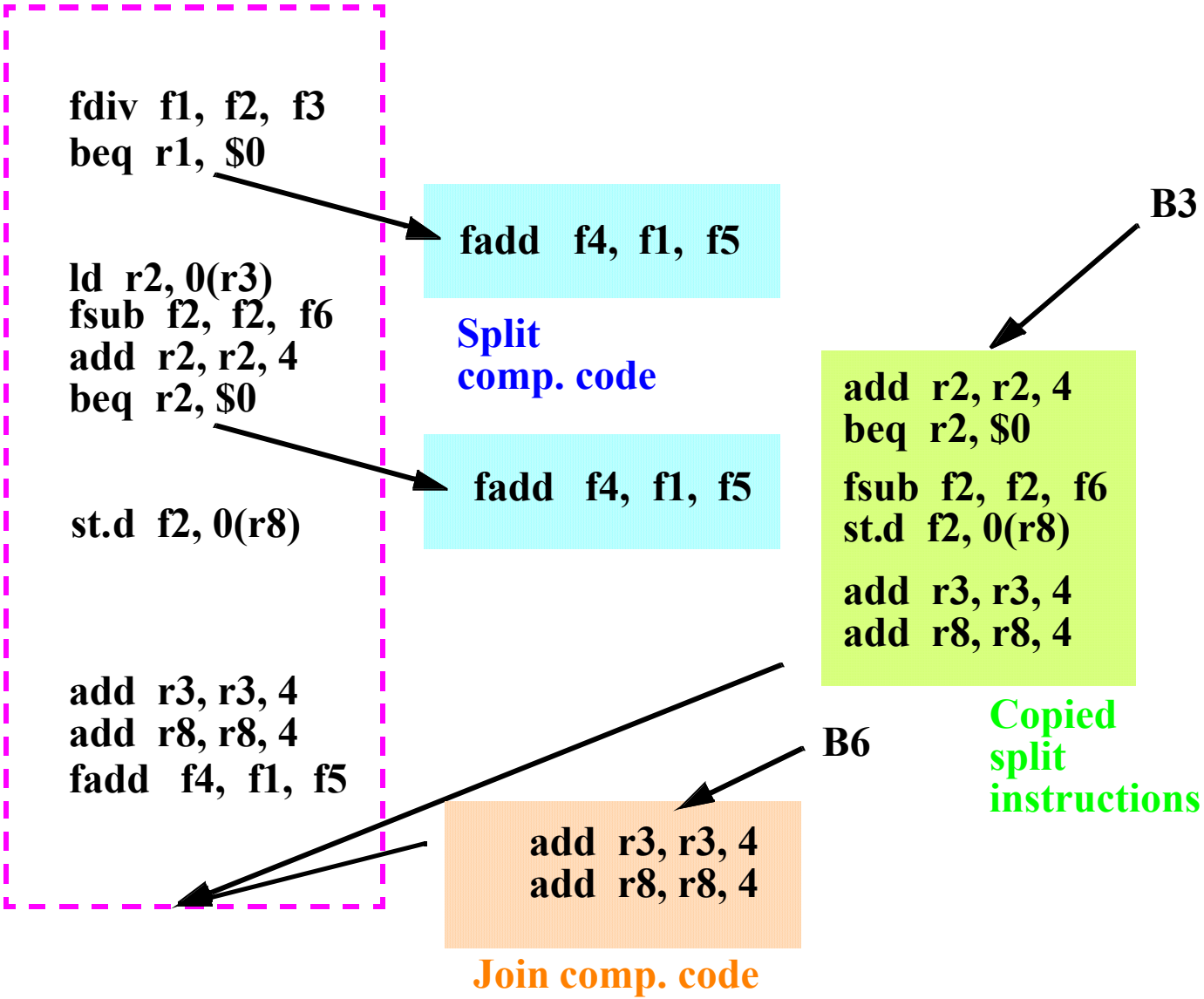
Compensation Code Example



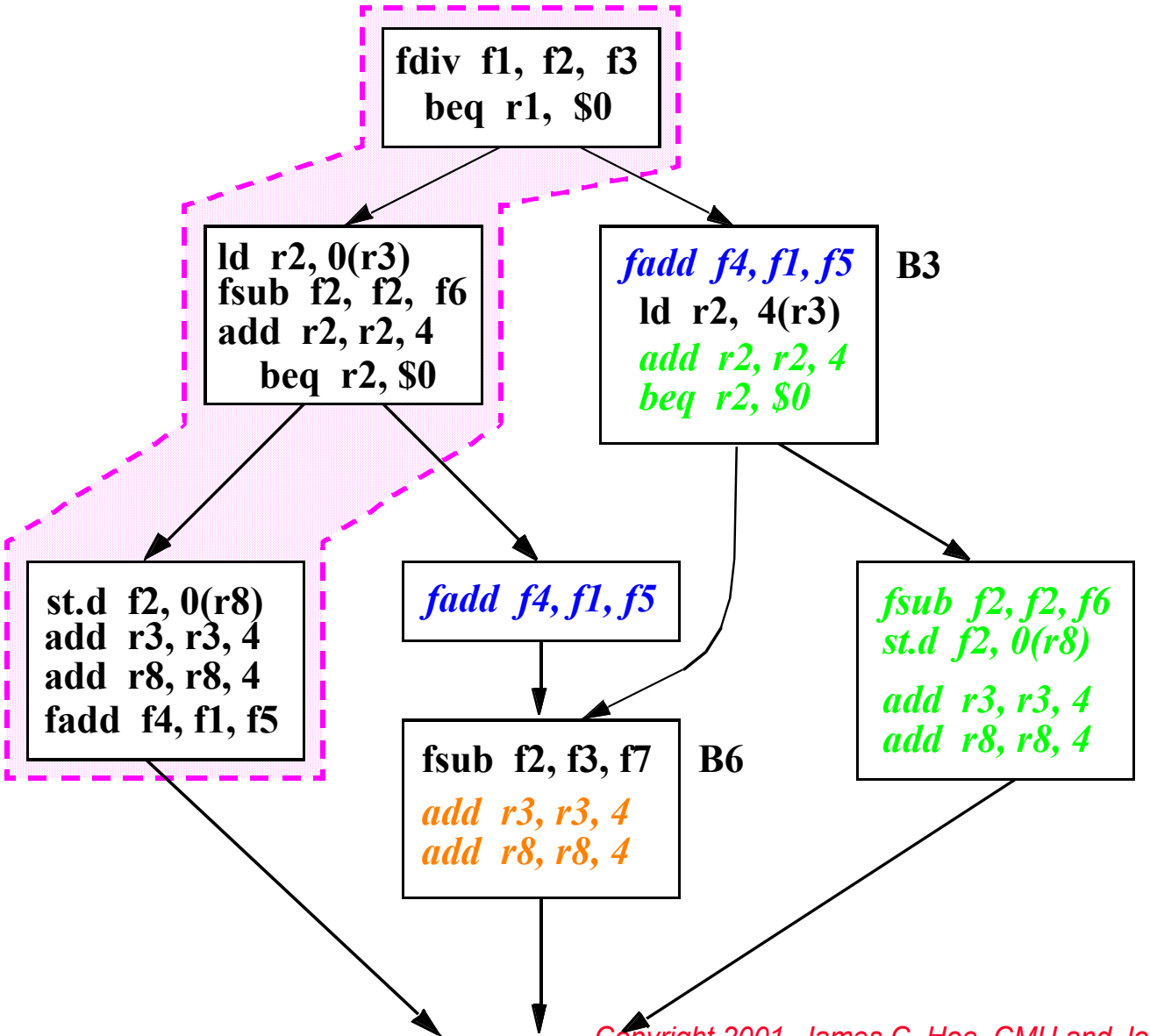
Compensation Code Example



Compensation Code Example

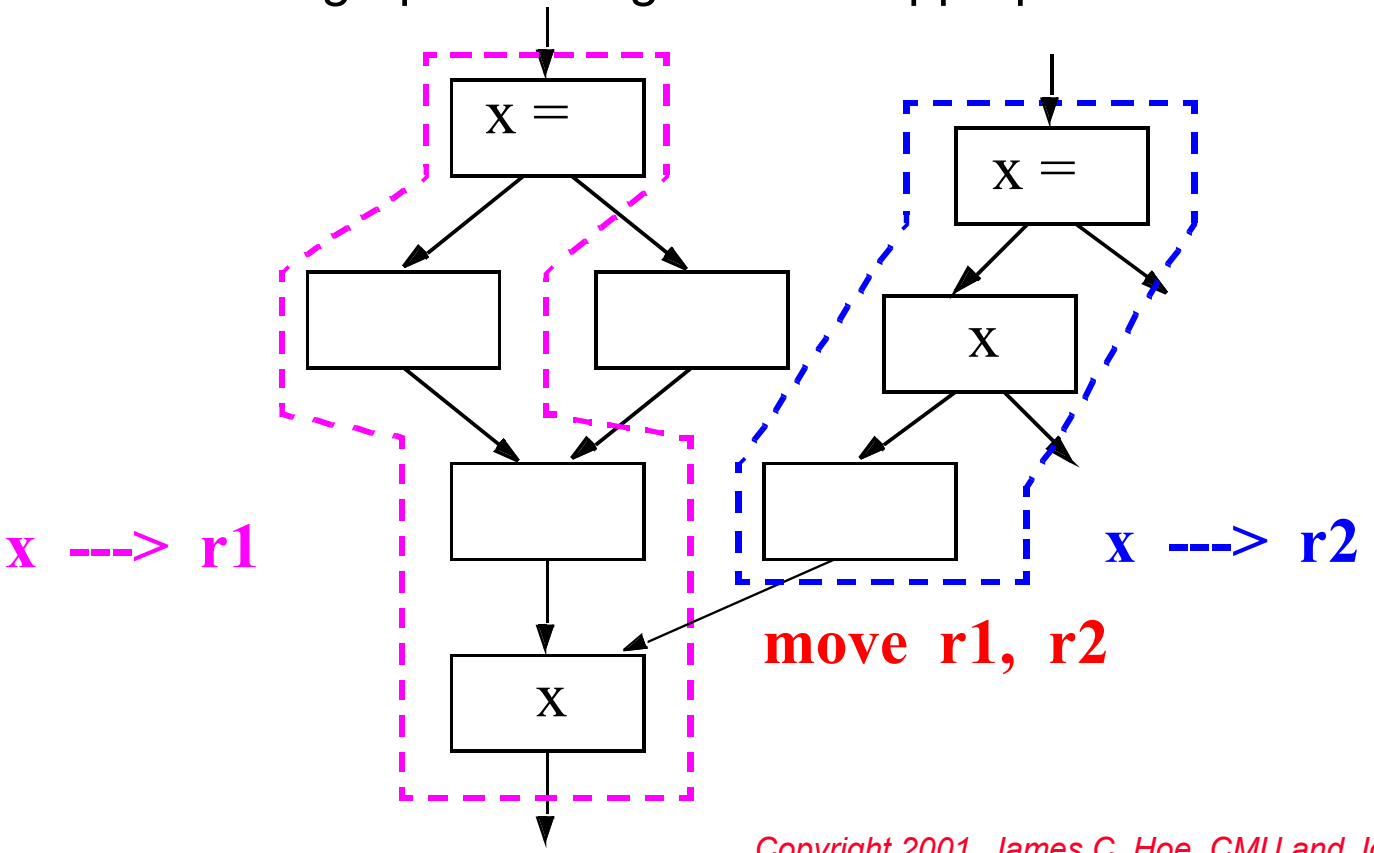


Compensation Code Illustration



Register Binding

- Perform register allocation for a trace
  - After scheduling a trace, do register allocation
  + Most frequently executed traces have maximum freedom of register usage
  - Do not use graph coloring due to inappropriate framework

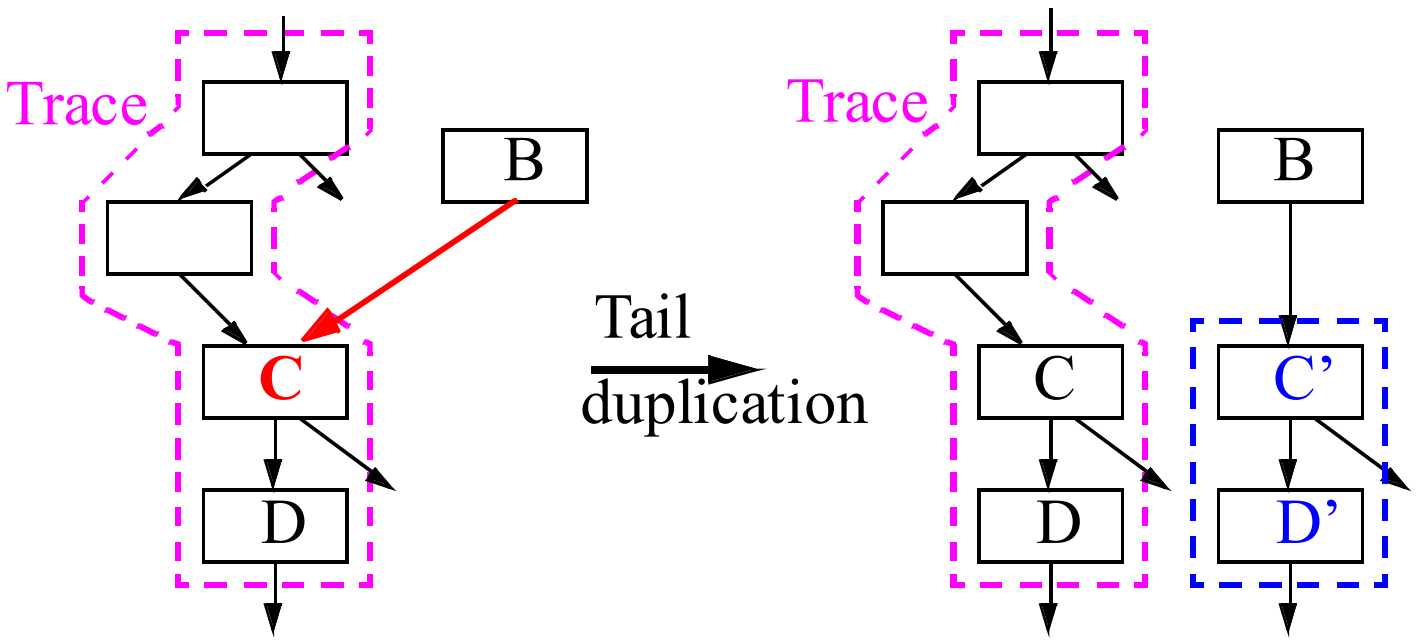


Superblock Scheduling

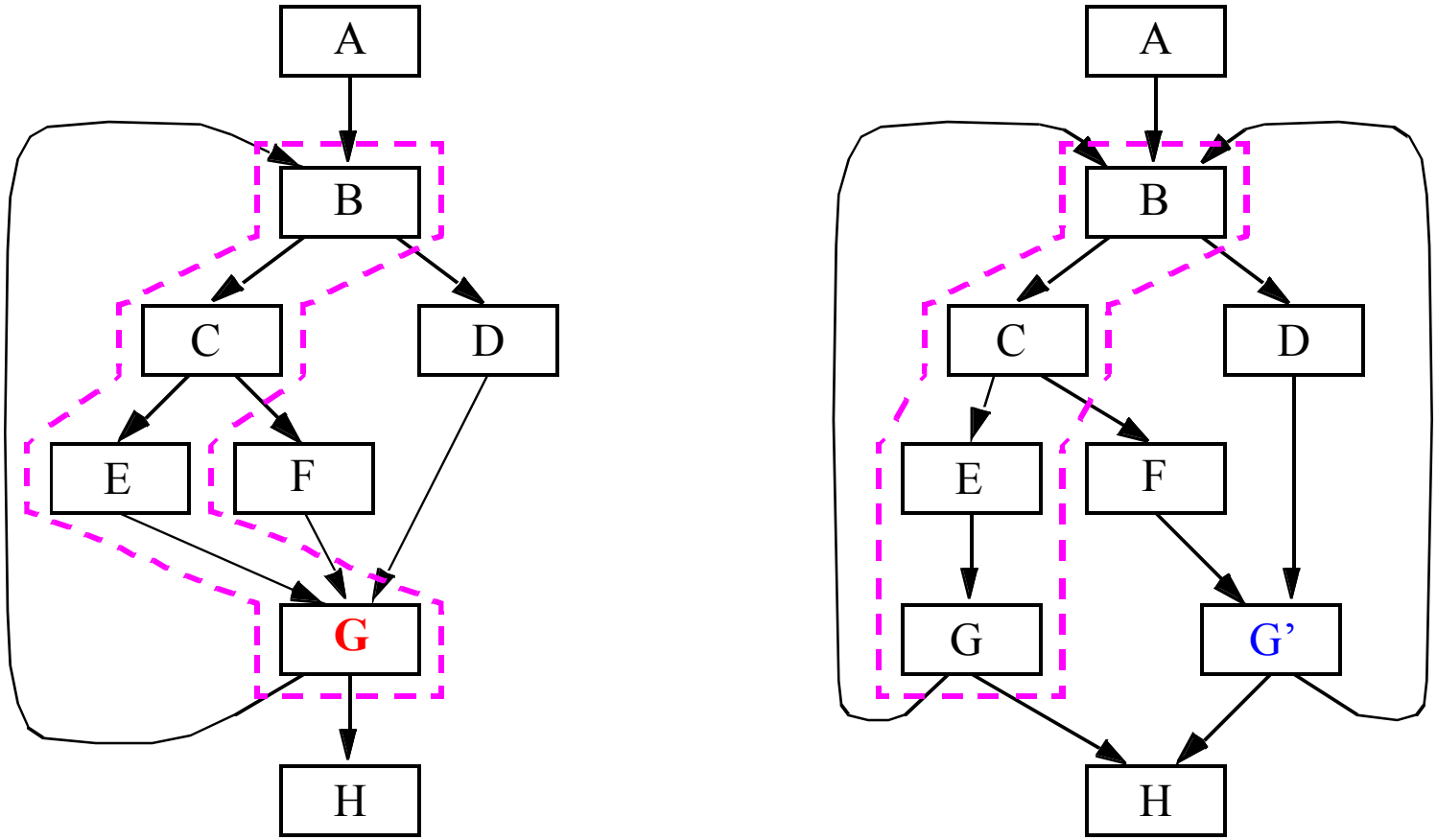
- Motivation
  - Trace scheduling is a good idea
  - Maintaining semantic correctness (compensation code) is a pain
- Superblock
  - Trace with one entry point (multiple entries create control flow joins)
  - May have multiple exits

Superblock Formation Example

- Identify traces using profiling information
- Use tail duplication to eliminate side entry points

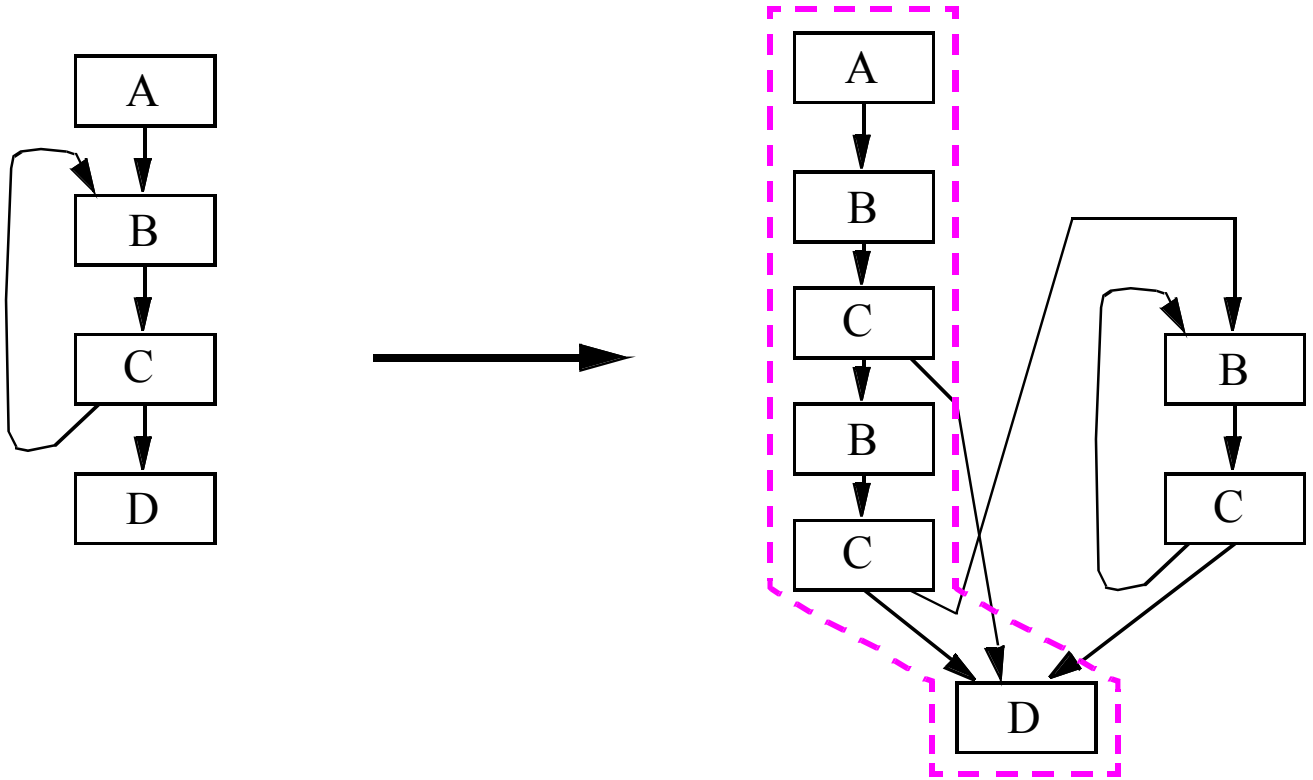


Superblock Formation Example



Superblock Enlarging

- Branch Target Expansion
  - Expand along likely-taken path
- Loop Unrolling & Loop Peeling



ILP Optimization

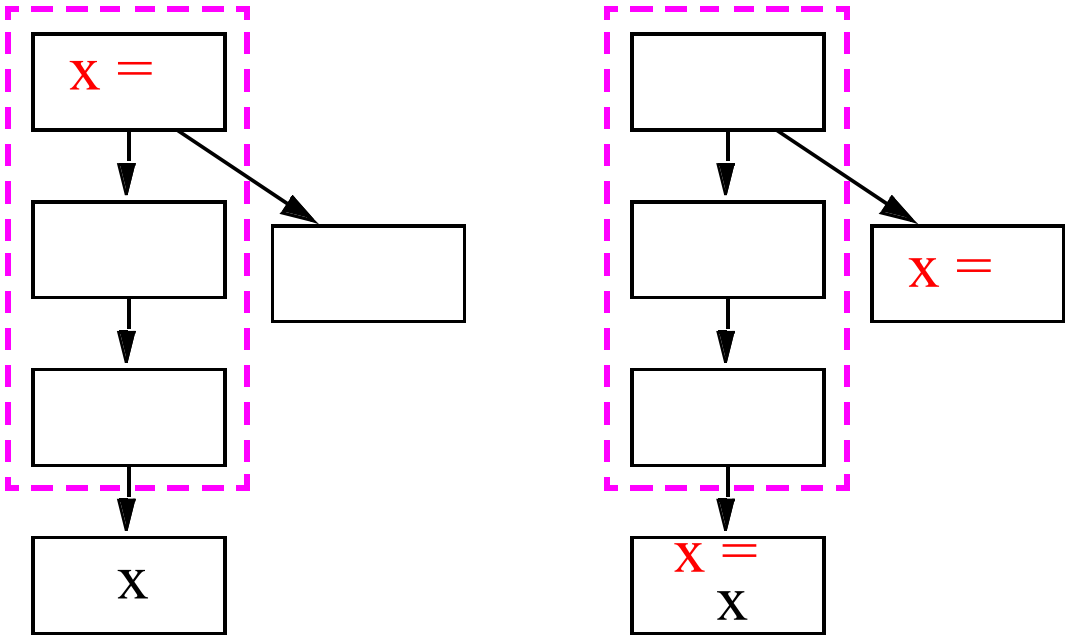
- Basic Block Size

	Average Block Size:
Basic block:	3 instructions
Superblock-original:	4 instructions
Superblock-formation:	10 instructions
Superblock-enlargement:	13 instructions

- Dependence Elimination
  - Code transformations to eliminate data dependencies
  - Give code scheduler more freedom to move instructions

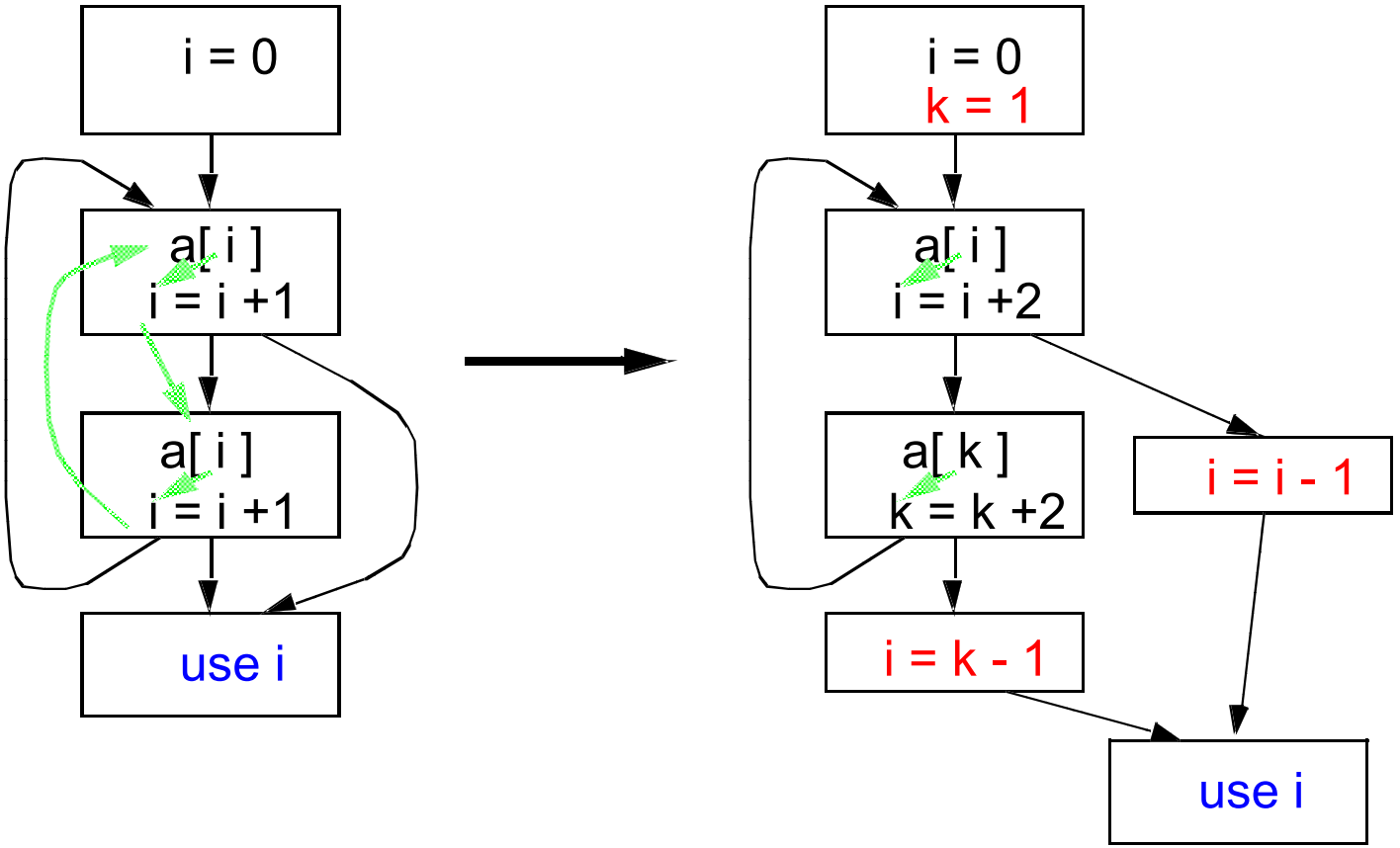
Operand Migration

- Move instructions whose results are not used within trace to less frequently executed paths



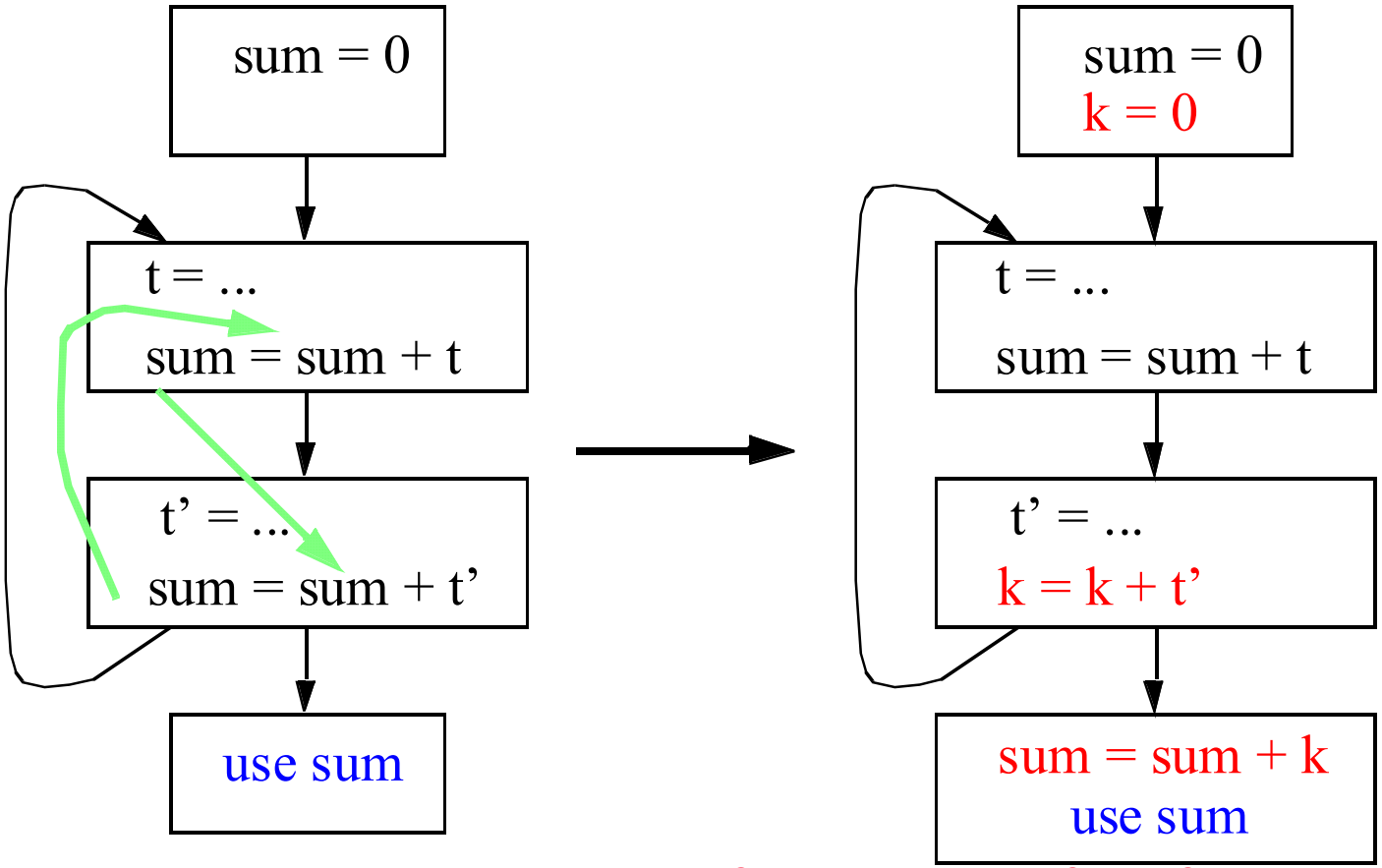
Induction Variable Expansion

- Eliminate redefinitions of induction variables within unrolled loops
- Insert code to maintain semantic correctness



Accumulator Variable Expansion

- Accumulate a sum or product in each iteration
- Insert code to maintain semantic correctness
- May not be safe for floating point



Superblock List Scheduling

- Restricted percolation
  - No architecture support
  - Instructions that could cause exceptions are not moved beyond branches
  - Memory load/store, integer divide and floating point instructions

- General percolation
  - Architecture support (non-trapping instructions)
  - Write garbage value when exceptions occur for non-trapping instructions

We will see this when we discuss Intel EPIC