

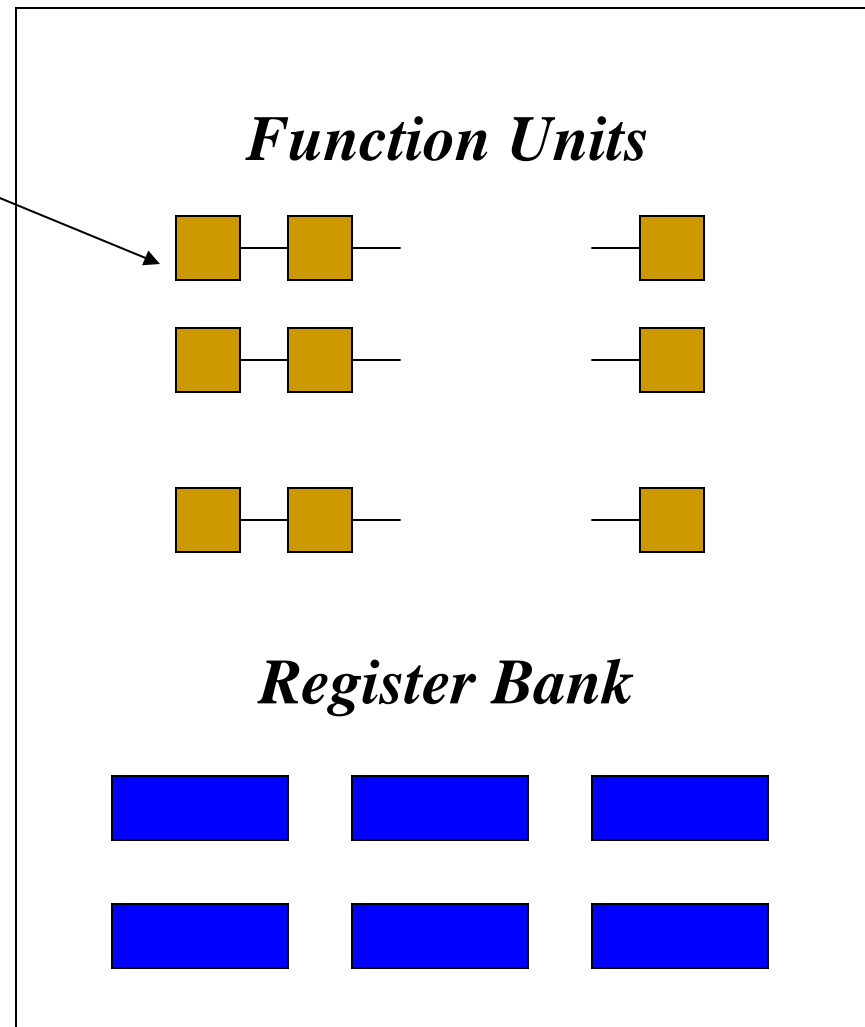


RICE

Instruction Scheduling

Superscalar (RISC) Processors

Pipelined
Fixed, Floating
Branch etc.



Canonical Instruction Set

- Register —Register Instructions (Single cycle).
- Special instructions for Load and Store to/from memory (multiple cycles).

A few notable exceptions of course.

Eg., Dec Alpha, HP-PA RISC, IBM Power & RS6K, Sun Sparc ...

Opportunity in Superscalars

- High degree of *Instruction Level Parallelism (ILP)* via multiple (possibly) *pipelined functional units (FUs)*.

Essential to harness promised performance.

- Clean simple model and Instruction Set makes *compile-time* optimizations feasible.
- Therefore, performance advantages can be harnessed automatically

Example of Instruction Level Parallelism

Processor components

- 5 functional units: 2 fixed point units, 2 floating point units and 1 branch unit.
- Pipeline depth: floating point unit is 2 deep, and the others are 1 deep.

Peak rates: 7 instructions being processed simultaneously in each cycle

Instruction Scheduling: The Optimization Goal

Given a source program P:

schedule the operations so as to minimize the overall execution time on the functional units in the target machine.

Alternatives for Embedded Systems:

- Minimize the amount of power consumed by functional units during execution of the program.
- Ensure operations are executed within given time constraints.

Cost Functions

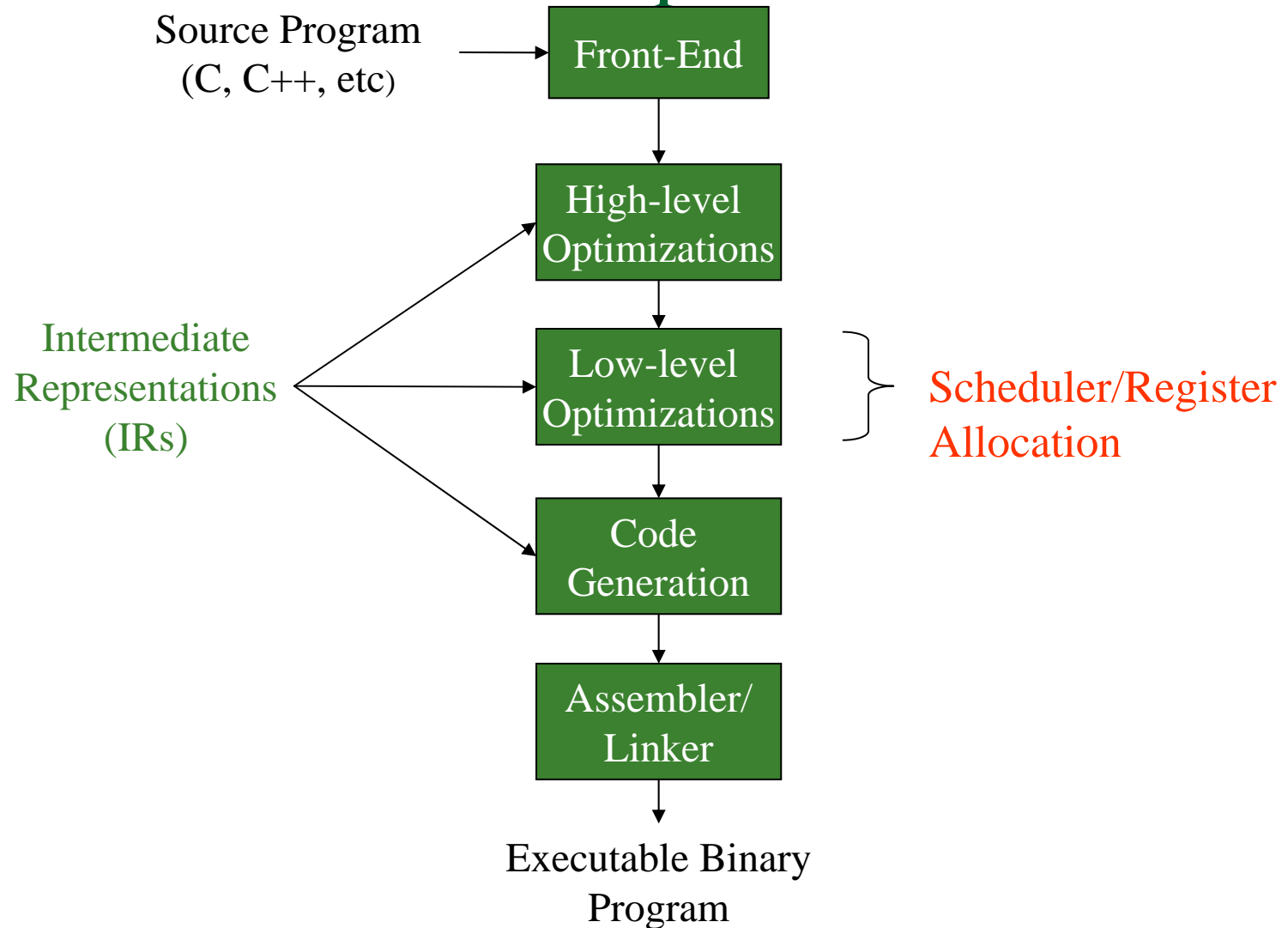
- Effectiveness of the Optimizations: *How well can we optimize our objective function?*

Impact on running time of the *compiled* code determined by the completion time.

- Efficiency of the optimization: *How fast can we optimize?*

Impact on the time it takes to compile or cost for gaining the benefit of code with fast running time.

Recap: Structure of an Optimizing Compiler



Instruction Scheduling: The Optimization Goal

Given a source program P:

schedule the operations so as to minimize the overall execution time on the functional units in the target machine.

Alternatives for Embedded Systems:

- Minimize the amount of power consumed by functional units during execution of the program.
- Ensure operations are executed within given time constraints.

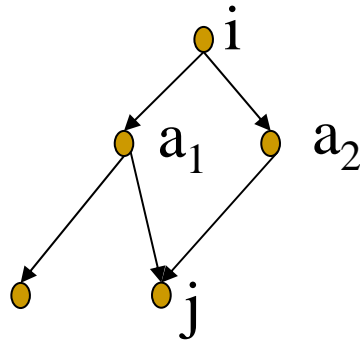
Cost Functions

- Effectiveness of the Optimizations: *How well can we optimize our objective function?*
 - Impact on running time of the *compiled* code determined by the completion time.
 - Impact on the power consumption of the compiled code during execution
 - Impact on the synchronization between operations
- Efficiency of the optimization: *How fast can we optimize?*
 - Impact of the optimization on the compile-time of the program

Graphs and Data Dependence

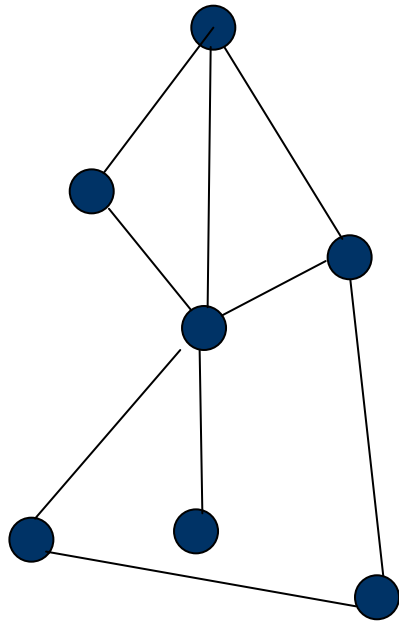
Introduction to DAGs

- A DAG is a Directed Acyclic Graph I.e. a graph with no cycles

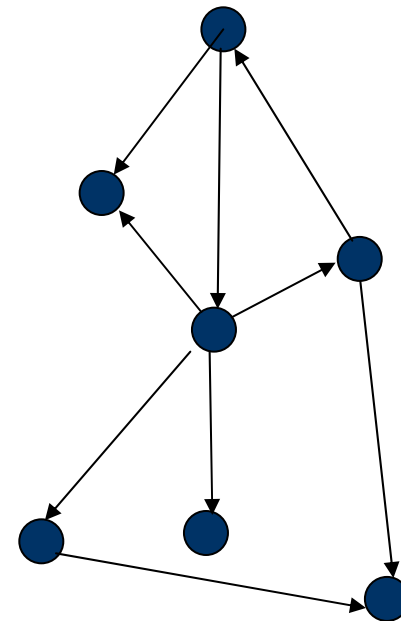


- $\text{Preds}(x) \equiv$ predecessors of node x ; e.g. $\text{preds}(j) = \{a_1, a_2\}$
- $\text{Succs}(x) \equiv$ successors of node x . e.g. $\text{succs}(i) = \{a_1, a_2\}$
- (i, a_2) represents the edge from node i to node a_2 ; note that $(i, a_2) \neq (a_2, i)$
- A path is a set of 1 or more edges that form a connection between two nodes e.g. $\text{path}(i, j) = \{(i, a_2); (a_2, j)\}$

Examples

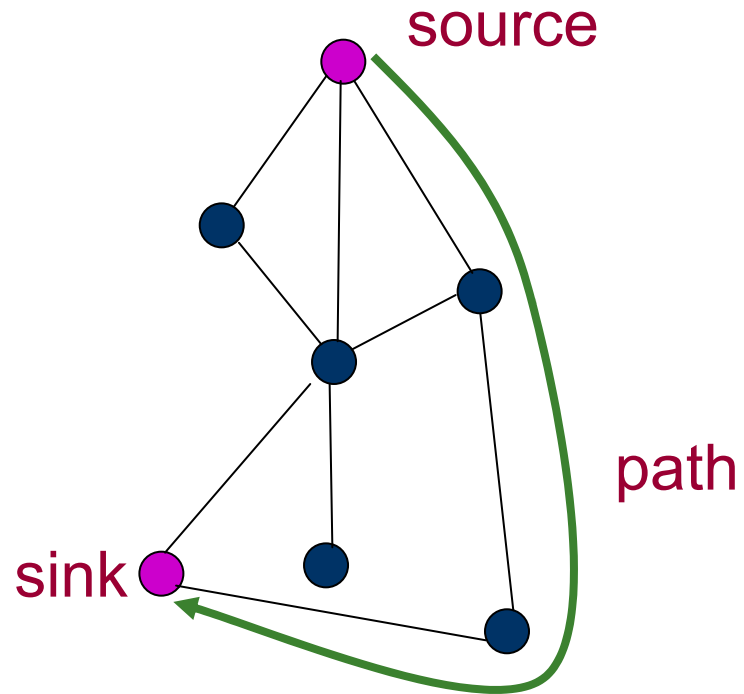


Undirected graph

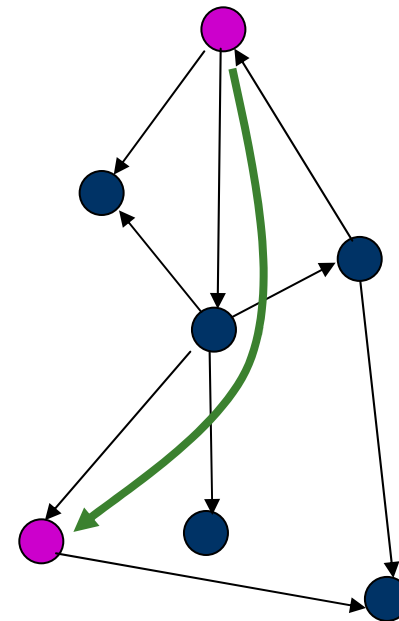


Directed graph

Paths

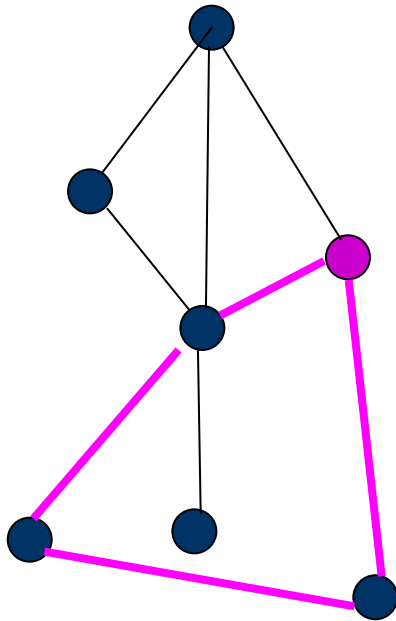


Undirected graph

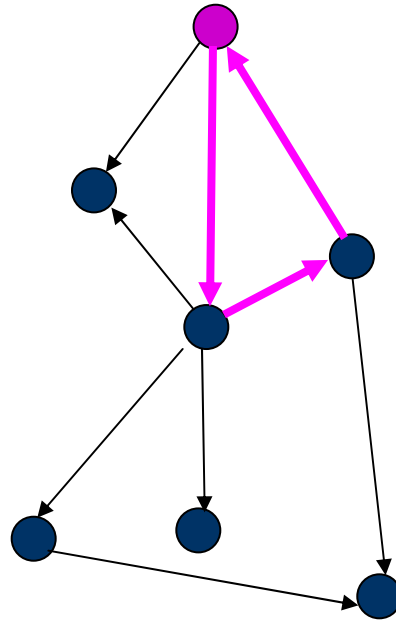


Directed graph

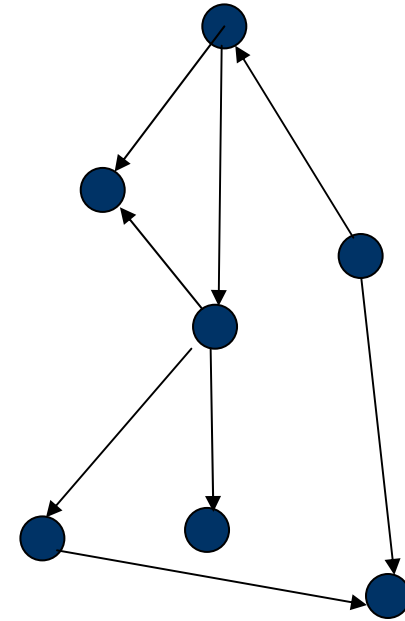
Cycles



Undirected graph

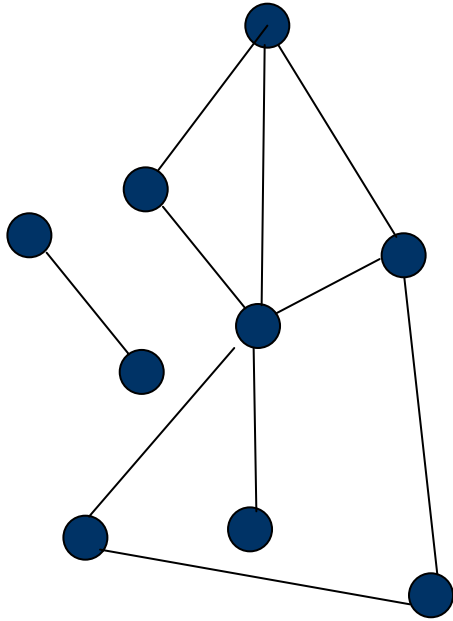


Directed graph

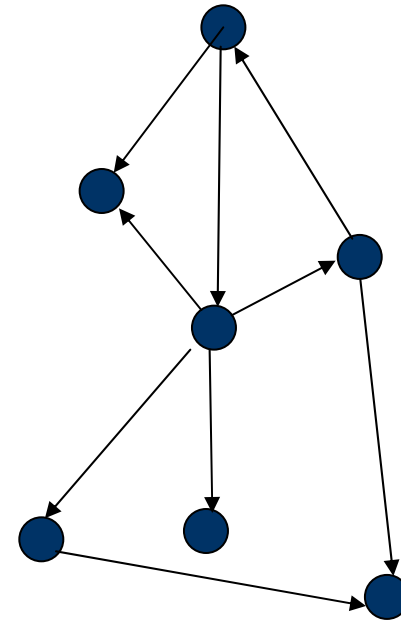


Acyclic
Directed
graph

Connected Graphs



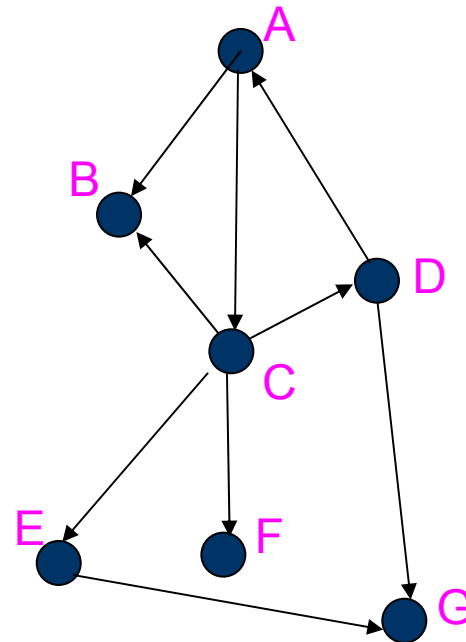
Unconnected graph



Connected
directed graph

Connectivity of Directed Graphs

- A strongly connected directed graph is one which has a path from each vertex to every other vertex
- Is this graph strongly connected?



Data Dependence Analysis

If two operations have potentially interfering data accesses, data dependence analysis is necessary for determining whether or not an interference actually exists. If there is no interference, it may be possible to reorder the operations or execute them concurrently.

The data accesses examined for data dependence analysis may arise from array variables, scalar variables, procedure parameters, pointer dereferences, etc. in the original source program.

Data dependence analysis is conservative, in that it may state that a data dependence exists between two statements, when actually none exists.

Data Dependence: Definition

A *data dependence*, $S_1 \rightarrow S_2$, exists between CFG nodes S_1 and S_2 with respect to variable X if and only if

1. there exists a path $P: S_1 \rightarrow S_2$ in *CFG*, with no intervening write to X , and
2. at least one of the following is true:
 - (a) **(flow)** X is written by S_1 and later read by S_2 , or
 - (b) **(anti)** X is read by S_1 and later is written by S_2 or
 - (c) **(output)** X is written by S_1 and later written by S_2



Instruction Scheduling Algorithms

Impact of Control Flow

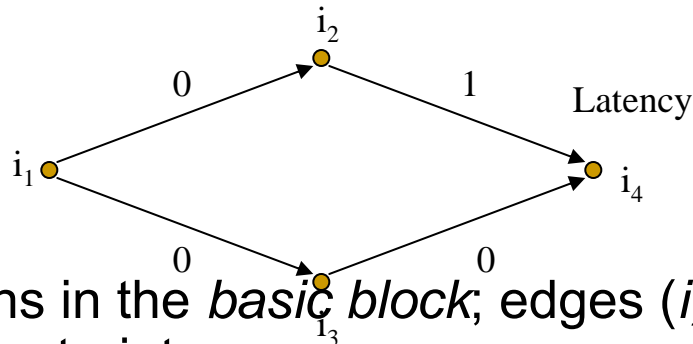
- *Acyclic* control flow is easier to deal with than *cyclic* control flow. Problems in dealing with cyclic flow:
 - A loop *implicitly* represent a large run-time program space compactly.
 - Not possible to open out the loops fully at compile-time.
 - Loop unrolling provides a partial solution.
 - Using the loop to optimize its dynamic behavior is a challenging problem.
 - Hard to optimize well without detailed knowledge of the range of the iteration.
 - In practice, profiling can offer limited help in estimating loop bounds

Acyclic Instruction Scheduling

- The acyclic case itself has two parts:
 - The simpler case that we will consider first has no branching and corresponds to *basic block* of code, eg., loop bodies.
 - The more complicated case of scheduling programs with acyclic control flow *with* branching will be considered next.
- Why basic blocks?
 - All instructions specified as part of the input must be executed.
 - Allows *deterministic* modeling of the input.
 - No “branch probabilities” to contend with; makes problem space easy to optimize using classical methods.

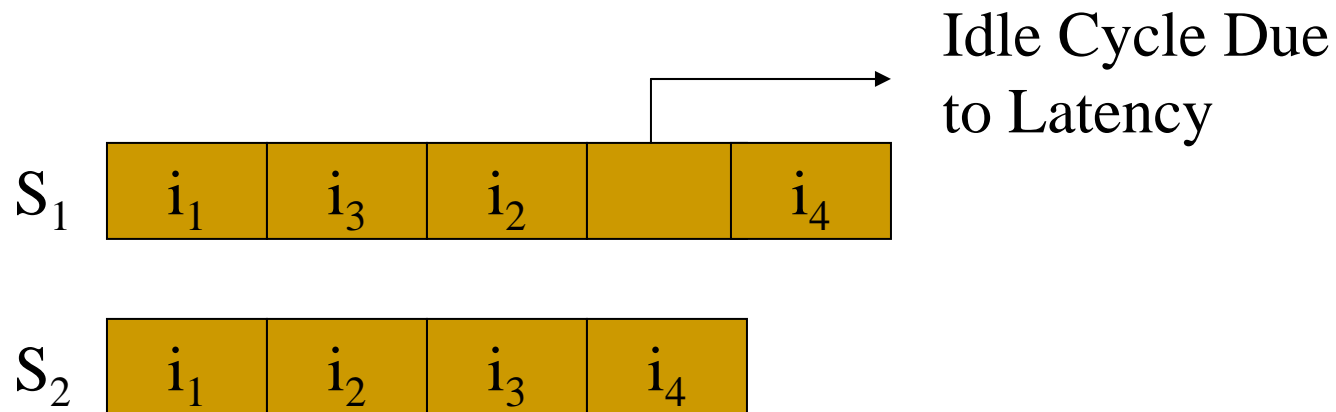
Example: Instruction Scheduling

Input: A basic block represented as a DAG



- $i_{\#}$ are instructions in the *basic block*; edges (i, j) represent dependence constraints
- i_2 is a load instruction.
- Latency of 1 on (i_2, i_4) means that i_4 cannot start for one cycle after i_2 completes.
- Assume 1 FU
- **What are the possible schedules?**

Example(cont): Possible Schedules



- Two possible schedules for the DAG
- The length of the schedule is the number of cycles required to execute the operations
 - $\text{Length}(S_1) > \text{Length}(S_2)$
- Which schedule is optimal?

Generalizing the Instruction Scheduling Problem

Input: DAG representing each basic block where:

1. Nodes encode *unit execution time* (single cycle) operations.
2. Each node requires a definite class of FU.
3. Additional time delays encoded as latencies on the edges.
4. Number of FUs of each type in the target machine.

more...

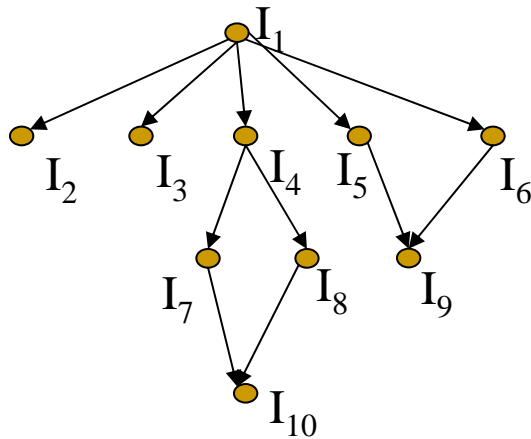
Generalizing the Instruction Scheduling Problem (Contd.)

Feasible Schedule: A specification of a *start time* for each instruction such that the following constraints are obeyed:

1. Resource: Number of instructions of a given type of any time $<$ corresponding number of FUs.
2. Precedence and Latency: For each predecessor j of an instruction i in the DAG, i is started only δ cycles after j finishes where δ is the latency labeling the edge (j,i) ,

Output: A schedule with the minimum *overall completion time (makespan)*.

Scheduling with infinite FUs

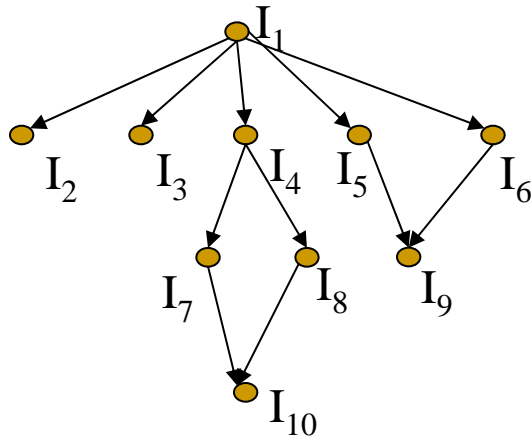


Input: DAG

Cycle	Ops
1	I ₁
2	I ₂ , I ₃ , I ₄ , I ₅ , I ₆
3	I ₇ , I ₈ , I ₉
4	I ₁₀

- Infinite FUs implies only the #2 constraint holds
 - Minimal length for a correct schedule can be obtained

Scheduling with finite FUs



Input: DAG

Cycle	Ops
1	I ₁ , <empty>
2	??

- Assuming 2 FUs
- **What happens in Cycle #2?** Must Choose from {I₂, I₃, I₄, I₅, I₆}
 - How does an algorithm decide which ops to choose?
 - What factors may influence this choice?
 - Fanout
 - Height
 - Resources available

Addressing Scheduling Questions

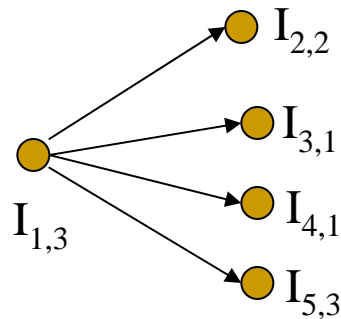
- **Greediness** helps in making sure that idle cycles don't remain if there are available instructions further "down stream."
 - If an instruction is available for a slot, then fill the slot
- **Ranks** help prioritize nodes such that choices made early on favor instructions with greater enabling power, so that there is no unforced idle cycle.
 - Ranks are an encoding for a scheduling heuristic
 - Ranks are based on characteristics of the operations, and allow the algorithm to compare operations

A Canonical Greedy List Scheduling Algorithm

1. Assign a *Rank (priority)* to each instruction (or node).
2. **Sort** and build a priority *list* \mathcal{L} of the instructions in non-decreasing order of Rank.
 - Nodes with smaller ranks occur earlier in this list
 - Smaller ranks imply higher priority
3. **Greedy list-schedule** \mathcal{L} .
 - **An instruction is ready** provided it has not been chosen earlier and all of its predecessors have been chosen and the appropriate latencies have elapsed.
 - Scan \mathcal{L} iteratively and on each scan, choose the largest number of “ready” instructions from the front of the list subject to resource (FU) constraints.

Applying the Canonical Greedy List Algorithm

Example: Consider the DAG shown below, where nodes are labeled (id, rank)



Sorting by ranks gives a list $\mathcal{L} = \langle i_{3,1}, i_{4,1}, i_{2,2}, i_{1,3}, i_{5,3} \rangle$

- The following slides apply the algorithm assuming 2 FUs.
more...

Applying the Canonical Greedy List Algorithm (cont.)

1. On the first scan
 1. $i_{1,3}$ is added to the schedule.
 2. No other ops can be scheduled, one empty slot
2. On the second and third scans
 1. $i_{3,1}$ and $i_{4,1}$ are added to the schedule
 2. All slots are filled, both FUs are busy
3. On the fourth and fifth scans
 1. $i_{2,2}$ and $i_{5,3}$ are added to the schedule
 2. All slots are filled, both FUs are busy
4. All ops have been scheduled

How Good is Greedy?

Approximation: For any pipeline depth $k \geq 1$ and any number m of pipelines,

$$S_{\text{greedy}}/S_{\text{opt}} \leq \left(2 - \frac{1}{mk}\right).$$

- For example, with one pipeline ($m=1$) and the latencies k grow as 2,3,4,..., the approximate schedule is guaranteed to have a completion time no more 66%, 75%, and 80% over the optimal completion time.
- This theoretical guarantee shows that greedy scheduling is not bad, but the bounds are worst-case; practical experience tends to be much better.

more...

How Good is Greedy? (Contd.)

Running Time of Greedy List Scheduling: Linear in the size of the DAG.

“Scheduling Time-Critical Instructions on RISC Machines,” K. Palem and B. Simons, *ACM Transactions on Programming Languages and Systems*, 632-658, Vol. 15, 1993.



RICE

A Critical Choice: The Rank
Function for Prioritizing Nodes

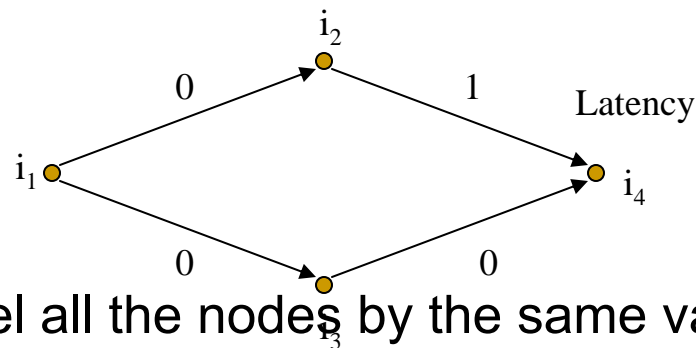
Rank Functions

1. “Postpass Code Optimization of Pipelined Constraints”, J. Hennessey and T. Gross, *ACM Transactions on Programming Languages and Systems*, vol. 5, 422-448, 1983.
2. “Scheduling Expressions on a Pipelined Processor with a Maximal Delay of One Cycle,” D. Bernstein and I. Gertner, *ACM Transactions on Programming Languages and Systems*, vol. 11 no. 1, 57-66, Jan 1989.
3. “Scheduling Time-Critical Instructions on RISC Machines,” K. Palem and B. Simons, *ACM Transactions on Programming Languages and Systems*, 632-658, vol. 15, 1993

Optimality: 2 and 3 produce optimal schedules for RISC processors

An Example Rank Function

The example DAG



1. Initially label all the nodes by the same value, say α
2. Compute new labels from old starting with nodes at level zero (i_4) and working towards higher levels:
 - (a) All nodes at level zero get a rank of α .

more...

An Example Rank Function (Contd.)

- (b) For a node at level 1, construct a new label which is the concentration of all its successors connected by a latency 1 edge.

Edge i_2 to i_4 in this case.

- (c) The empty symbol \emptyset is associated with latency zero edges.

Edges i_3 to i_4 for example.

An Example Rank Function

(d) The result is that i_2 and i_3 respectively get new labels and hence ranks $\alpha' = \alpha > \alpha'' = \emptyset$.

Note that $\alpha' = \alpha > \alpha'' = \emptyset$ i.e., labels are drawn from a totally ordered alphabet.

(e) Rank of i_1 is the concentration of the ranks of its immediate successors i_2 and i_3 i.e., it is $\alpha''' = \alpha' | \alpha''$.

3. The resulting sorted list is (optimum) i_1, i_2, i_3, i_4 .



Control Flow Graphs

Control Flow Graphs

- **Motivation:** language-independent and machine-independent representation of control flow in programs used in high-level and low-level code optimizers. The flow graph data structure lends itself to use of several important algorithms from graph theory.

Control Flow Graph: Definition

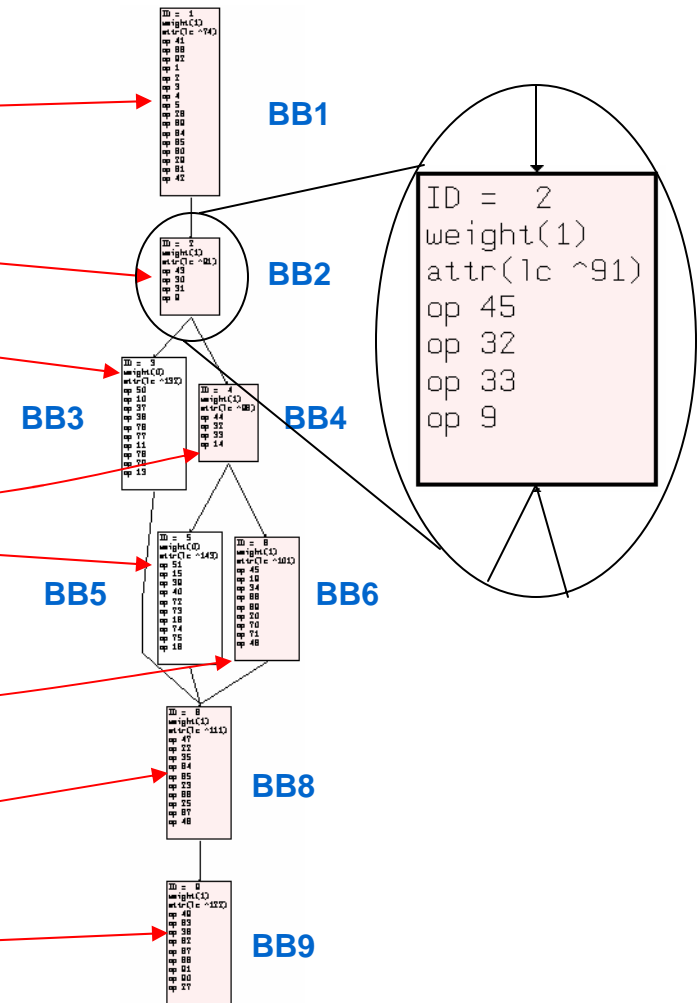
A control flow graph $CFG = (N_c; E_c; T_c)$ consists of

- N_c , a set of nodes. A node represents a straight-line sequence of operations with no intervening control flow i.e. a **basic block**.
- $E_c \subseteq N_c \times N_c \times Labels$, a set of *labeled* edges.
- T_c , a node type mapping. $T_c(n)$ identifies the type of node n as one of: *START*, *STOP*, *OTHER*.

We assume that CFG contains a unique *START* node and a unique *STOP* node, and that for any node N in CFG , there exist directed paths from *START* to N and from N to *STOP*.

Example CFG

```
main(int argc, char *argv[ ]  
{  
    if (argc == 1) {  
        printf("1");  
    } else {  
        if (argc == 2) {  
            printf("2");  
        } else {  
            printf("others");  
        }  
    }  
    printf("done");  
}
```



Control Dependence Analysis

We want to capture two related ideas with control dependence analysis of a CFG:

1. Node Y should be control dependent on node X if node X evaluates a predicate (conditional branch) which can control whether node Y will subsequently be executed or not. This idea is useful for determining whether node Y needs to wait for node X to complete, even though they have no data dependences.

Control Dependence Analysis (contd.)

2. Two nodes, Y and Z , should be identified as having identical control conditions if in every run of the program, node Y is executed if and only if node Z is executed. This idea is useful for determining whether nodes Y and Z can be made adjacent and executed concurrently, even though they may be far apart in the CFG.

Program Dependence Graph

- The **Program Dependence Graph** (PDG) is the intermediate (abstract) representation of a program designed for use in optimizations
- It consists of two important graphs:
 - **Control Dependence Graph** captures control flow and control dependence
 - **Data Dependence Graph** captures data dependences

Data and Control Dependences

Motivation: identify only the essential control and data dependences which need to be obeyed by transformations for code optimization.

Program Dependence Graph (PDG) consists of

1. Set of nodes, as in the CFG
2. Control dependence edges
3. Data dependence edges

Together, the control and data dependence edges dictate whether or not a proposed code transformation is legal.



RICE

The More General Case
Scheduling Acyclic

Control Flow Graphs

Significant Jump in Compilation Cost

What is the problem when compared to basic-blocks?

- Conditional and unconditional branching is permitted.
- The problem being optimized is no longer deterministically and completely known at compile-time.
- Depending on the sequence of branches taken, the problem structure of the graph being executed can vary
- Impractical to optimize all possible combinations of branches and have a schedule for each case, since a sequence of k branches can lead to 2^k possibilities -- a *combinatorial explosion* in cost of compiling.

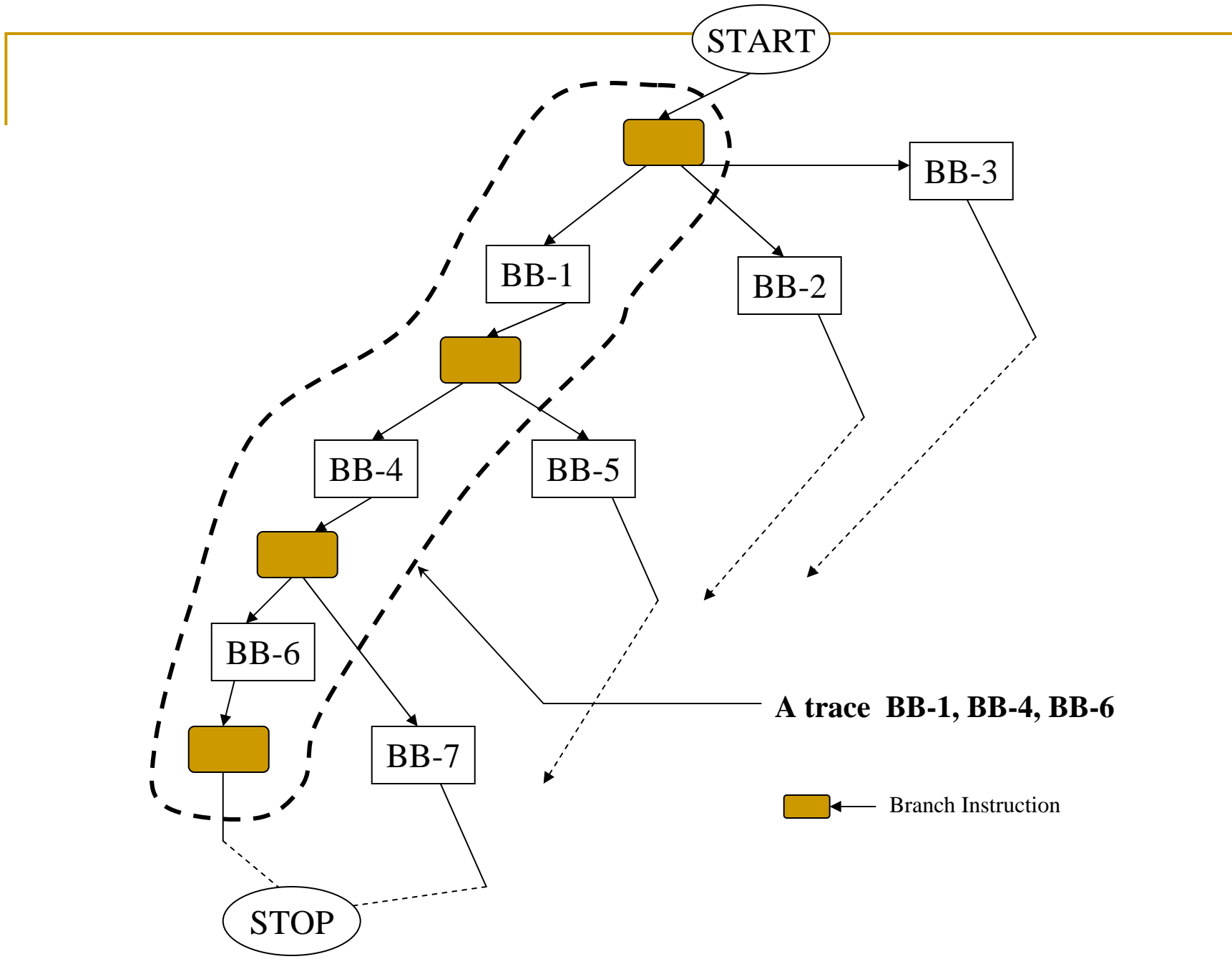
Trace Scheduling

A well known classical approach is to consider *traces* through the (acyclic) *control flow graph*. An example is presented in the next slide.


“Trace Scheduling: A Technique for Global Microcode Compaction,” J.A. Fisher, *IEEE Transactions on Computers*, Vol. C-30, 1981.

Main Ideas:

- Choose a program segment that has no cyclic dependences.
- Choose *one* of the paths out of each branch that is encountered.



A trace BB-1, BB-4, BB-6

 ← Branch Instruction

Trace Scheduling (Contd.)

- Use statistical knowledge based on (estimated) program behavior to bias the choices to favor the more frequently taken branches.
- This information is gained through profiling the program or via static analysis.
- The resulting sequence of basic blocks including the branch instructions is referred to as a trace.

Trace Scheduling

High Level Algorithm:

1. Choose a (maximal) segment s of the program with acyclic control flow.

The instructions in s have associated “frequencies” derived via statistical knowledge of the program’s behavior.

2. Construct a trace τ through s :
 - (a) Start with the instruction in s , say i , with the highest frequency.

more...

Trace Scheduling (Contd.)

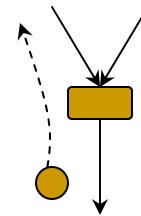
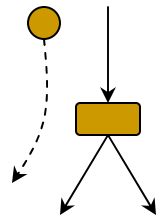
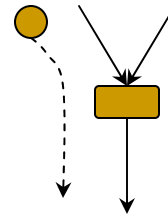
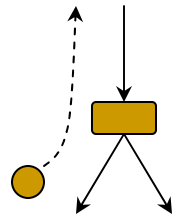
- (b) Grow a path out from instruction i in both directions, choosing the path to the instruction with the higher frequency whenever there is

Frequencies can be viewed as a way of prioritizing the path to choose and subsequently optimize.


3. Rank the instructions in τ using a rank function of choice.
4. Sort and construct a list \mathcal{L} of the instructions using the ranks as priorities.
5. Greedily list schedule and produce a schedule using the list \mathcal{L} as the priority list.

The Four Elementary but Significant Side-effects

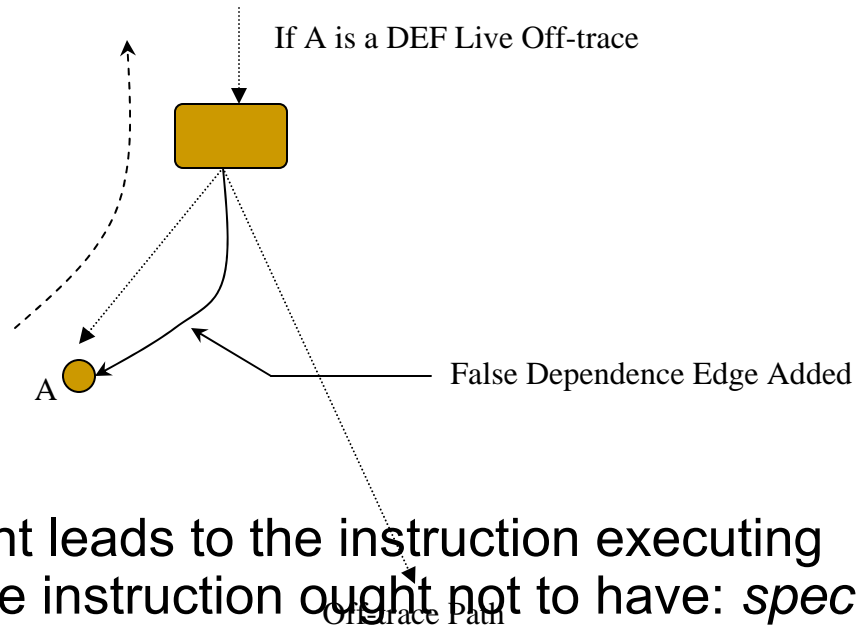
Consider a single instruction moving past a conditional branch:



 ← Branch Instruction

 ← Instruction being moved

The First Case

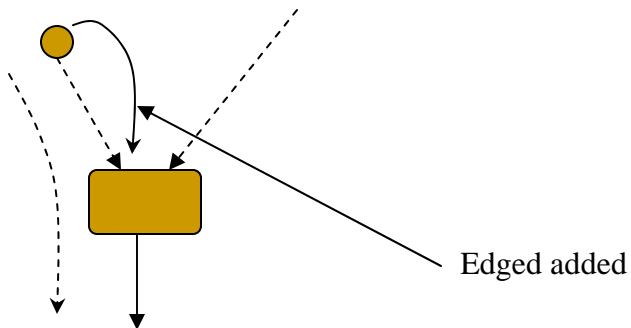


- This code movement leads to the instruction executing sometimes when the instruction ought not to have: *speculatively.*
more...

The First Case (Contd.)

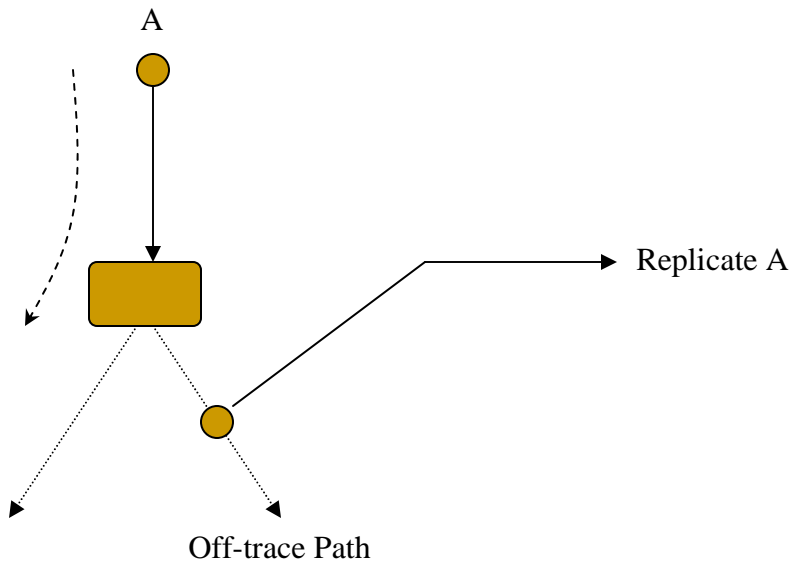
- If A is a *write* of the form $a := \dots$, then, the variable (virtual register) a must not be live on the off-trace path.
- In this case, an additional pseudo edge is added from the branch instruction to instruction A to prevent this motion.

The Second Case



- Identical to previous case except the pseudo-dependence edge is *from A* to the *join* instruction whenever A is a “write” or a *def*.
- A more general solution is to permit the code motion but undo the effect of the speculated definition by adding repair code
An expensive proposition in terms of compilation cost.

The Third Case



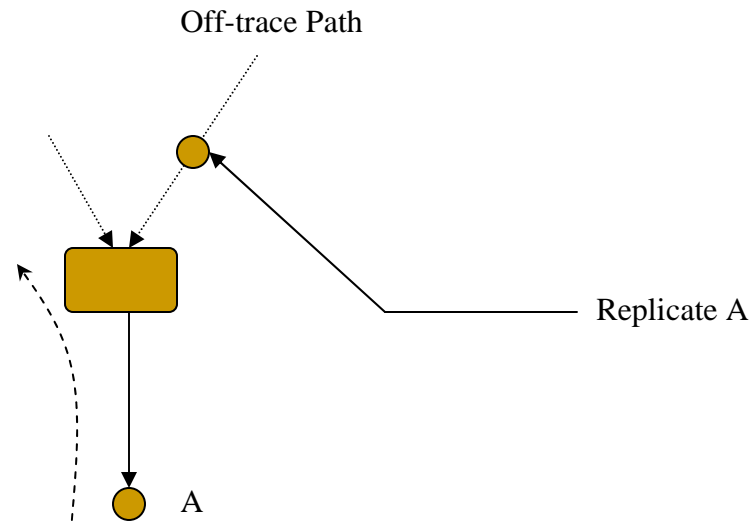
- Instruction *A* will *not* be executed if the off-trace path is taken.
- To avoid mistakes, it is *replicated*.

more...

The Third Case (Contd.)

- This is true in the case of read and write instructions.
- Replication causes A to be executed independent of the path being taken to preserve the original semantics.
- If (non-)liveness information is available, replication can be done more conservatively.

The Fourth Case



- Similar to Case 3 except for the direction of the replication as shown in the figure above.

At a Conceptual Level: Two Situations

- **Speculations:** Code that is executed “sometimes” when a branch is executed is now executed “always” due to code motion as in Cases 1 and 2.
 - *Legal* speculations wherein data-dependences are not violated.
 - *Safe* speculation wherein control-dependences on exceptions-causing instructions are not violated.

more...

At a Conceptual Level: Two Situations (Contd.)

- *Unsafe speculation* where there is no restriction and hence exceptions can occur.

This type of speculation is currently playing a role in “production quality” compilers.

- **Replication:** Code that is “always” executed is duplicated as in Cases 3 and 4.

Comparison to Basic Block Scheduling

- Instruction scheduler needs to handle speculation and replication.
- Otherwise the framework and strategy is identical.

Significant Comments

- We pretend as if the trace is always taken and executed and hence schedule it in steps 3-5 using the same framework as for a basic-block.
- The important difference is that conditionals branches are there on the path, and moving code past these conditionals can lead to side-effects.
- These side effects are not a problem in the case of basic-blocks since there, every instruction is executed all the time.
- This is not true in the present more general case when an outgoing or incoming off-trace branch is taken however infrequently: we will study these issues next.

Fisher's Trace Scheduling Algorithm

Description:

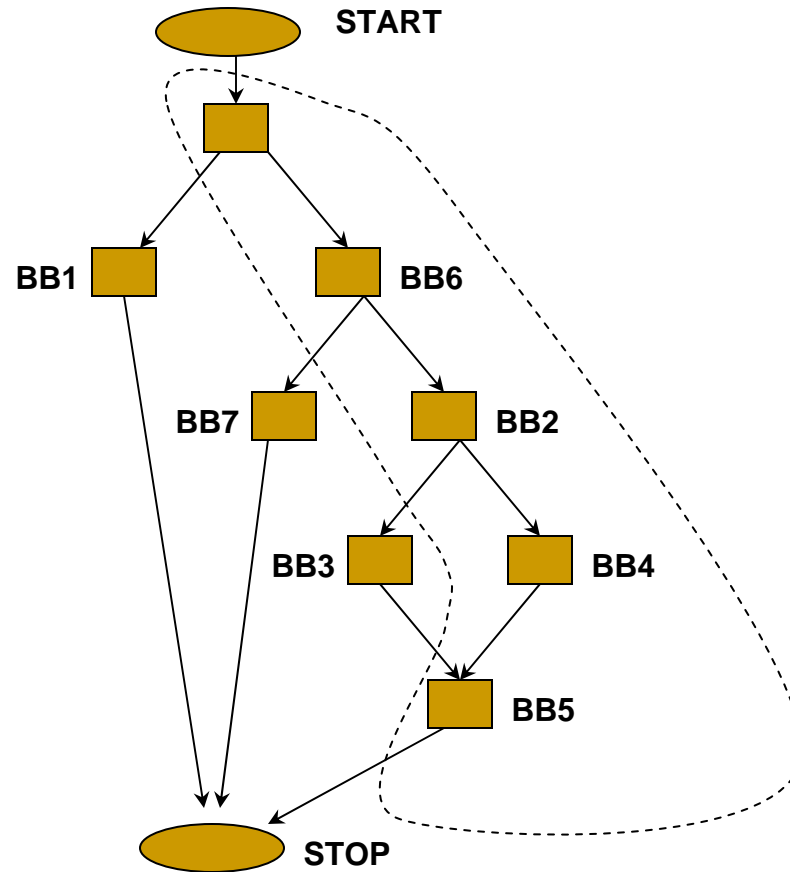
1. Choose a (maximal) region s of the program that has acyclic control flow.
2. Construct a trace τ through s .
3. Add additional dependence edges to the DAG to limit speculative execution.
Note that this is Fisher's solution.

more...

Fisher's Trace Scheduling Algorithm (Contd.)

4. Rank the instructions in τ using a rank function of choice.
5. Sort and construct a list \mathcal{L} of the instructions using the ranks as priorities.
6. Greedily list schedule and produce a schedule using the list \mathcal{L} as the priority list.
7. Add replicated code whenever necessary on all the off-trace paths.

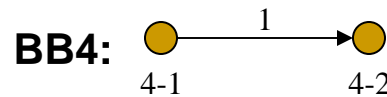
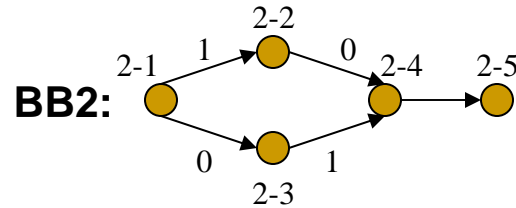
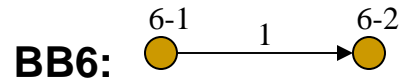
Example applying Fisher's Algorithm



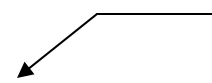
BBi  *Basic-block*

Example (Contd.)

TRACE: BB6, BB2, BB4, BB5



Concentration of Local Schedules



Obvious advantages of global code motion are that the idle cycles have disappeared.

Feasible Schedule: 6-1 X 6-2 2-1 X 2-2 2-3 X 2-4 2-5 4-1 X 4-2 5-1

Global Improvements 6-1 2-2 6-2 2-2 2-3 X 2-4 2-5 4-1 X 4-2 5-1:

6-1 2-1 6-2 2-3 2-2 2-4 2-5 4-1 X 4-2 5-1

6-1 2-1 6-2 2-3 2-2 2-4 2-5 4-1 5-1 4-2

X:Denotes Idle Cycle

Limitations of This Approach

- Optimizations depends on the traces being the dominant paths in the program's control-flow
- Therefore, the following two things should be true:
 - Programs should demonstrate the behavior of being skewed in the branches taken at run-time, for typical mixes of input data.
 - We should have access to this information at compile time. Not so easy.

Hyperblocks

- Single entry/ multiple exit set of predicated basic block (**if conversion**)
- Two conditions for hyperblocks:
 - **Condition 1:** There exist no incoming control flow arcs from outside basic blocks to the selected blocks other than the entry block I.e. no entry edges
 - **Condition 2:** There exist no nested inner loops inside the selected blocks

Hyper block formation procedure

- Tail duplication
 - Remove side entries
 - Code expansion must be monitored
 - Loop Peeling
 - Create bigger region for nested loop
-
- Node Splitting
 - Eliminate dependencies created by control path merge
 - Large code expansion must be monitored
 - After above three transformations, perform **if-conversion**
- } Hyperblock Pruning
- } Reducing Ctrl Flow Complexity

Criteria for Selecting BBs

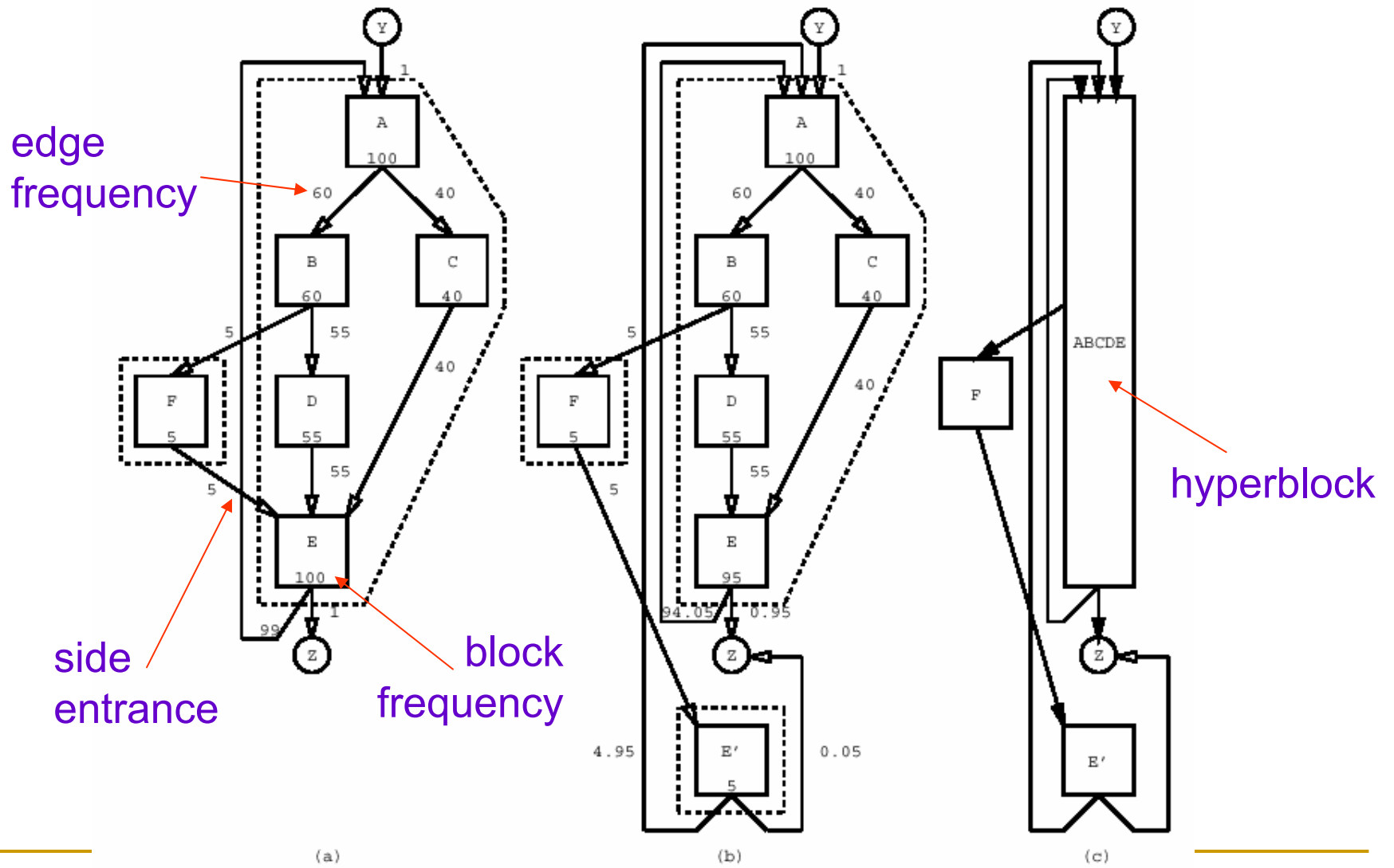
- To form hyperblocks, we must consider
 - Execution Frequency
 - Exclude paths that are not frequently executed
 - Basic Block Size
 - Include smaller blocks in favor of larger blocks.
 - Larger blocks use many machine resources, having an adverse affect on the performance of smaller blocks.
 - instruction characteristics
 - Basic blocks containing hazardous instructions are less likely to be included
 - Hazardous instructions are procedure calls, unresolvable memory accesses, etc. (l.e. any ambiguous operations)

The Formulated Selection Heuristic

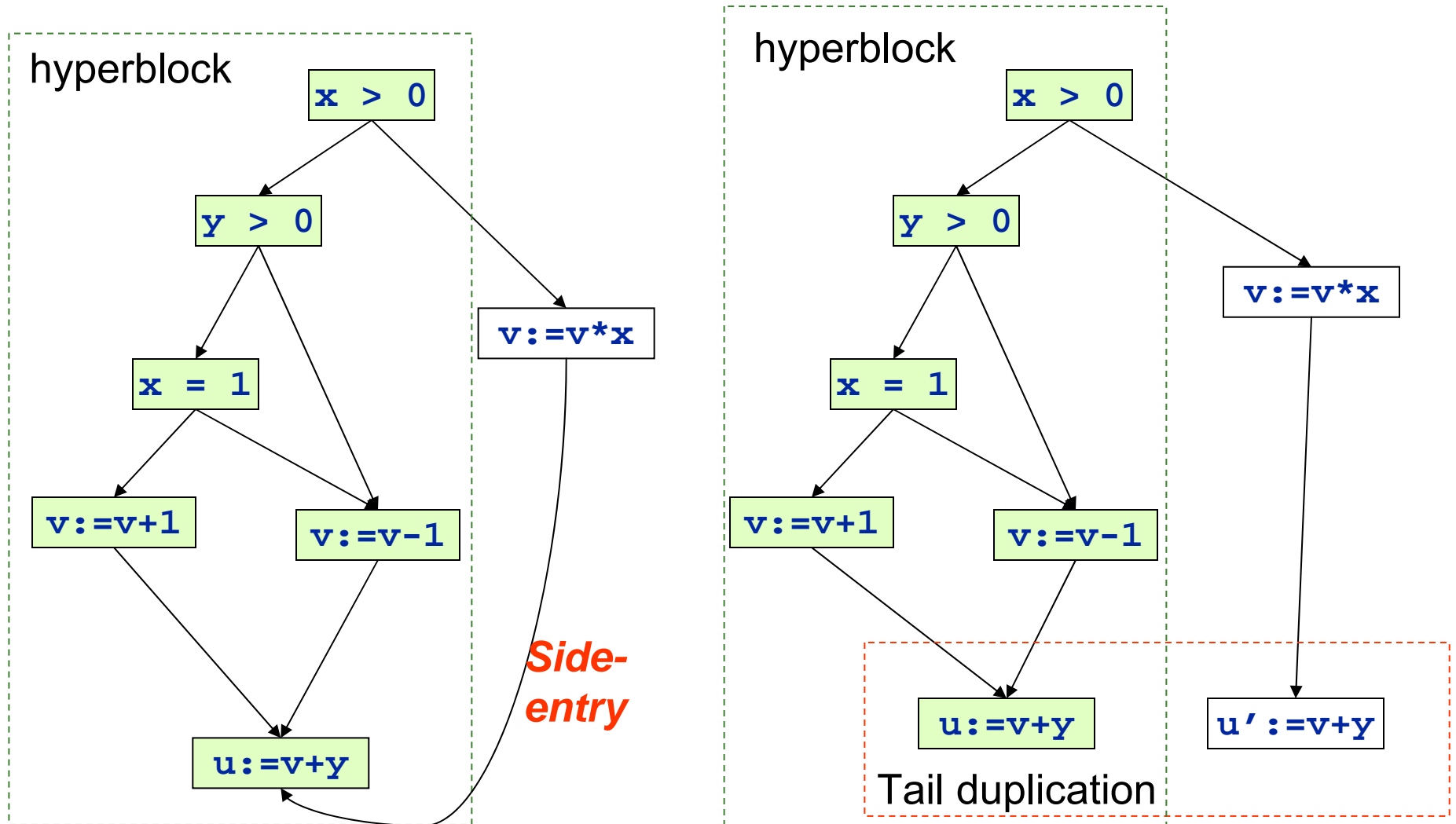
$$BSV_i = (K \times \frac{weight_bb_i}{size_bb_i} \times \frac{size_main_path_1}{weight_main_path_1} \times bb_char_i)$$

- *BSV* : Block Selection Value
- *K*: machine parameter to represent the issue rate of the processor
- *weight_bb*: execution frequency of the block
- *Size_bb*: number of instructions per block
- “*main path*” is the most likely executed control path through the region of blocks considered for inclusion in the hyperblock
- *bb_char_i* is a “characteristic value”; lower for blocks containing hazardous instructions; always less than 1
- large blocks have a lower probability of selection

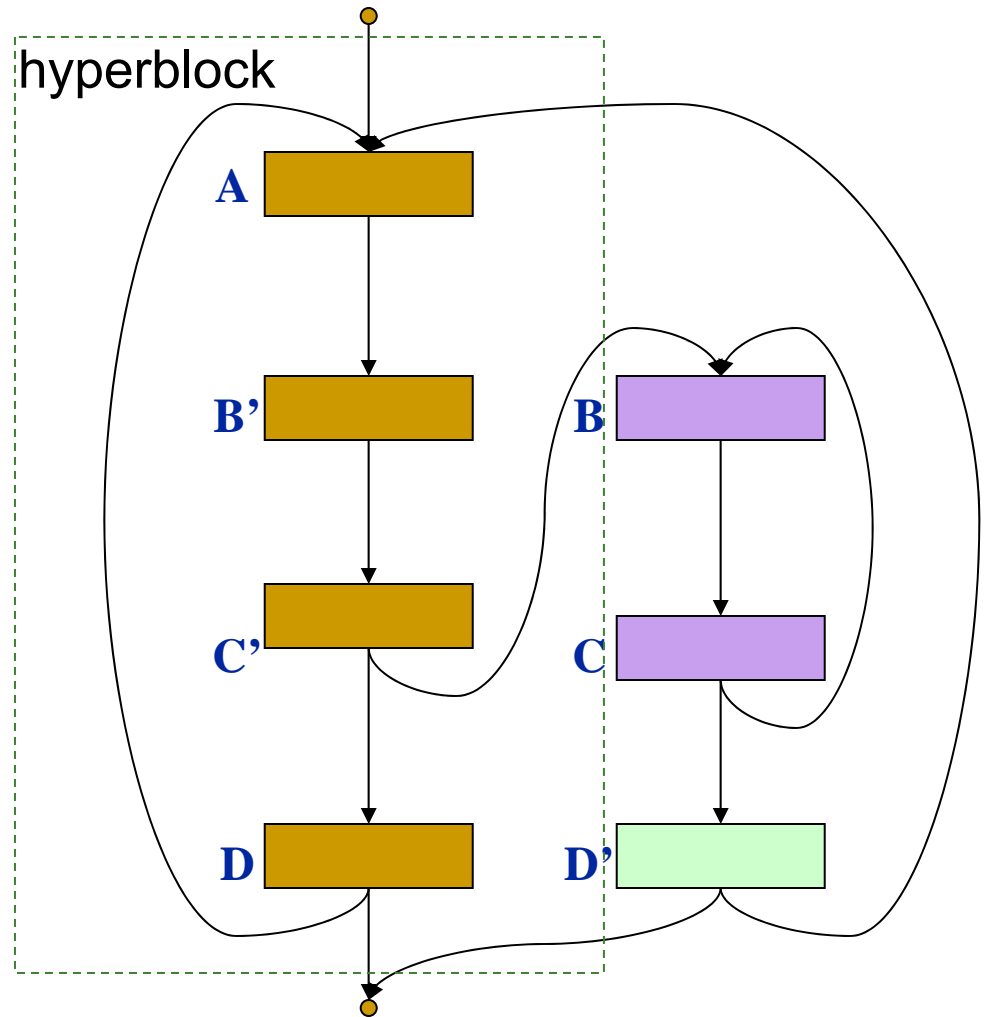
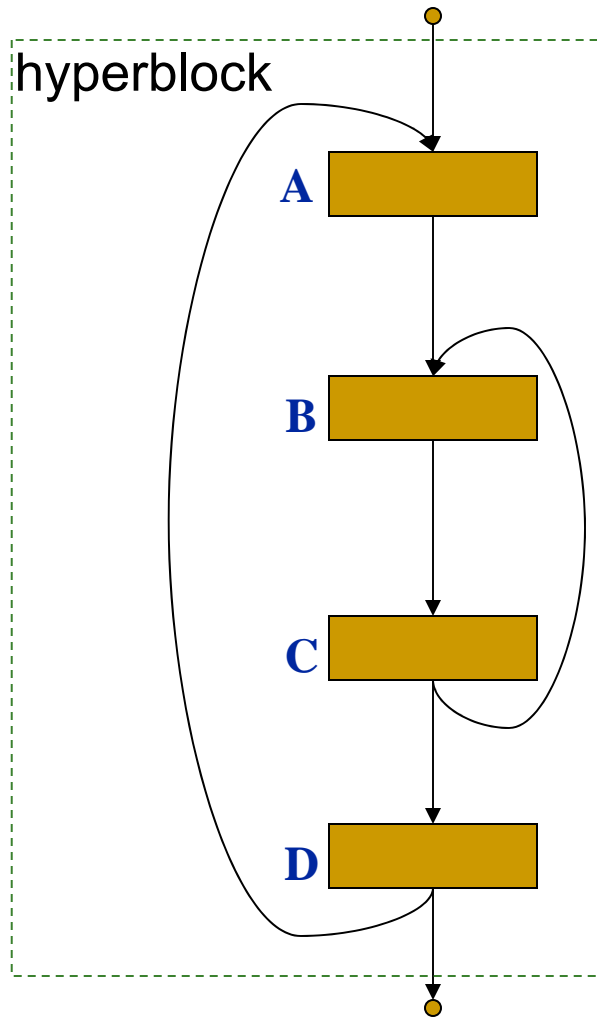
An Example



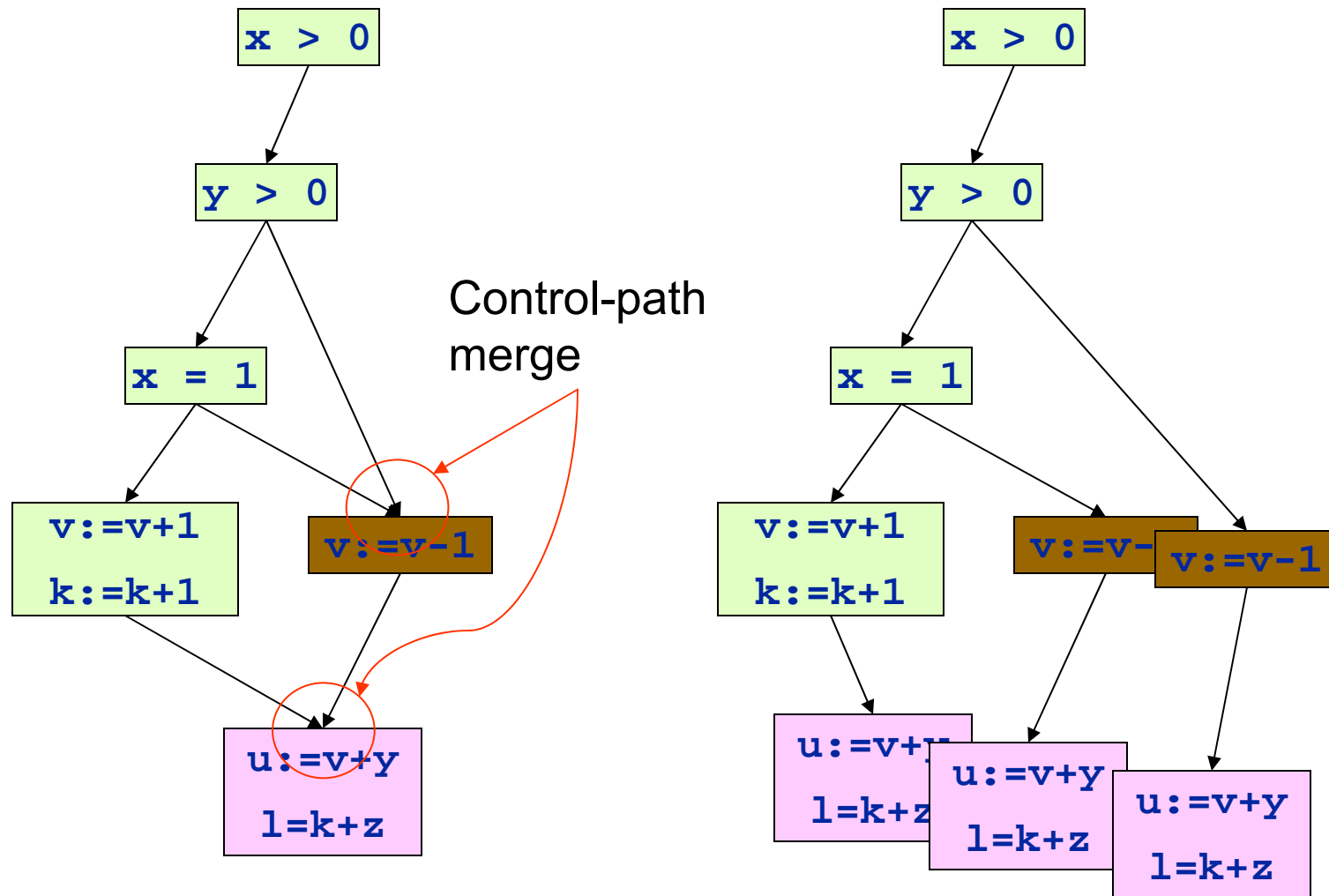
Tail Duplication: Removes Side Entries



Loop Peeling: Removes Inner loop



Node Splitting: Eliminate Dependencies due to Merge



Managing Node Splitting

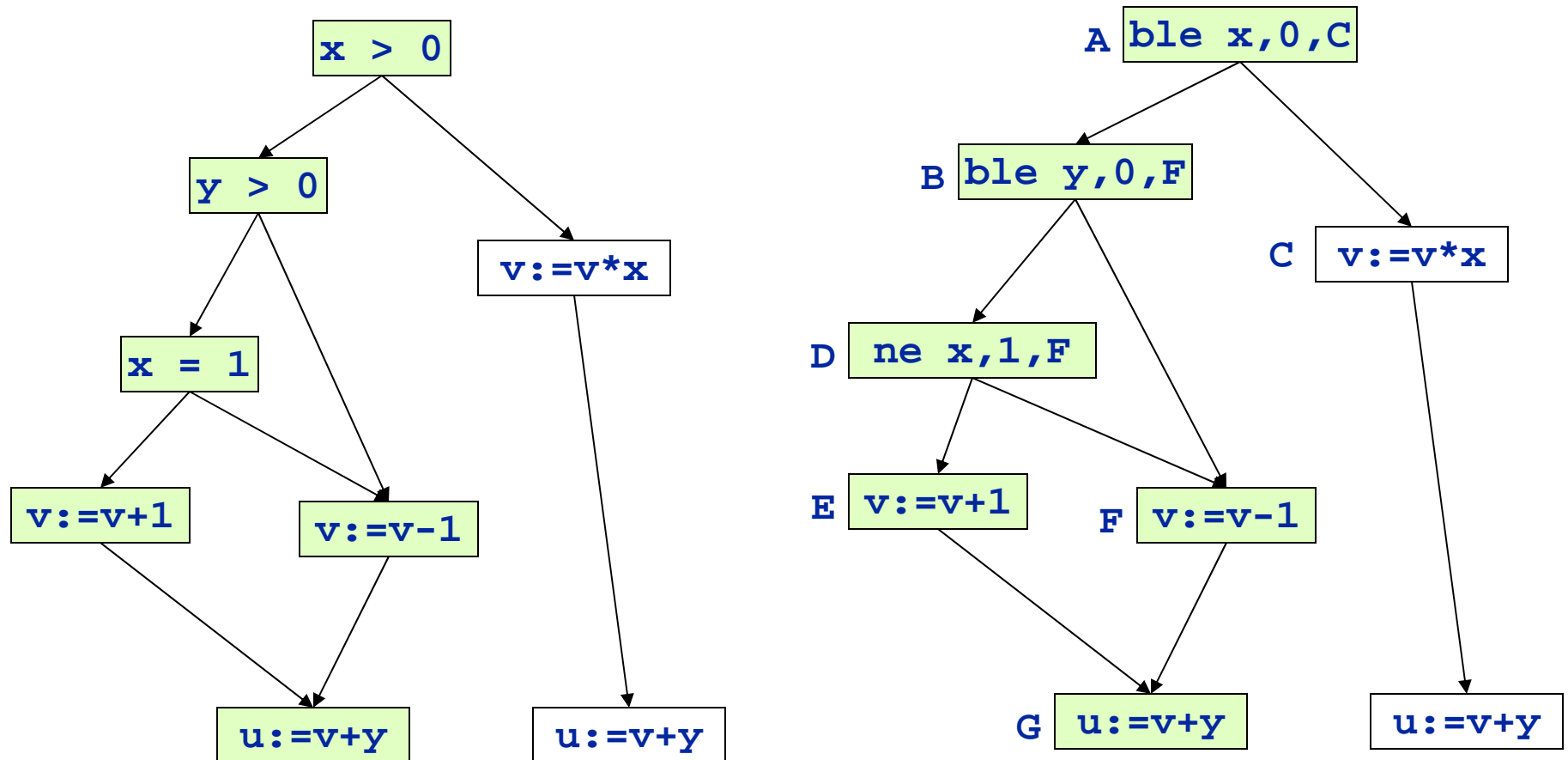
- Excessive node splitting can lead to code explosion
- Use the following heuristics, the Flow Selection Value, which is computed for each control flow edge in the blocks selected for the hyperblock that contain two or more incoming edges

$$FSV_i = \left(K \times \frac{weight_flow_i}{size_flow_i} \times \frac{size_main_path_1}{weight_main_path_1} \times bb_char_i \right),$$

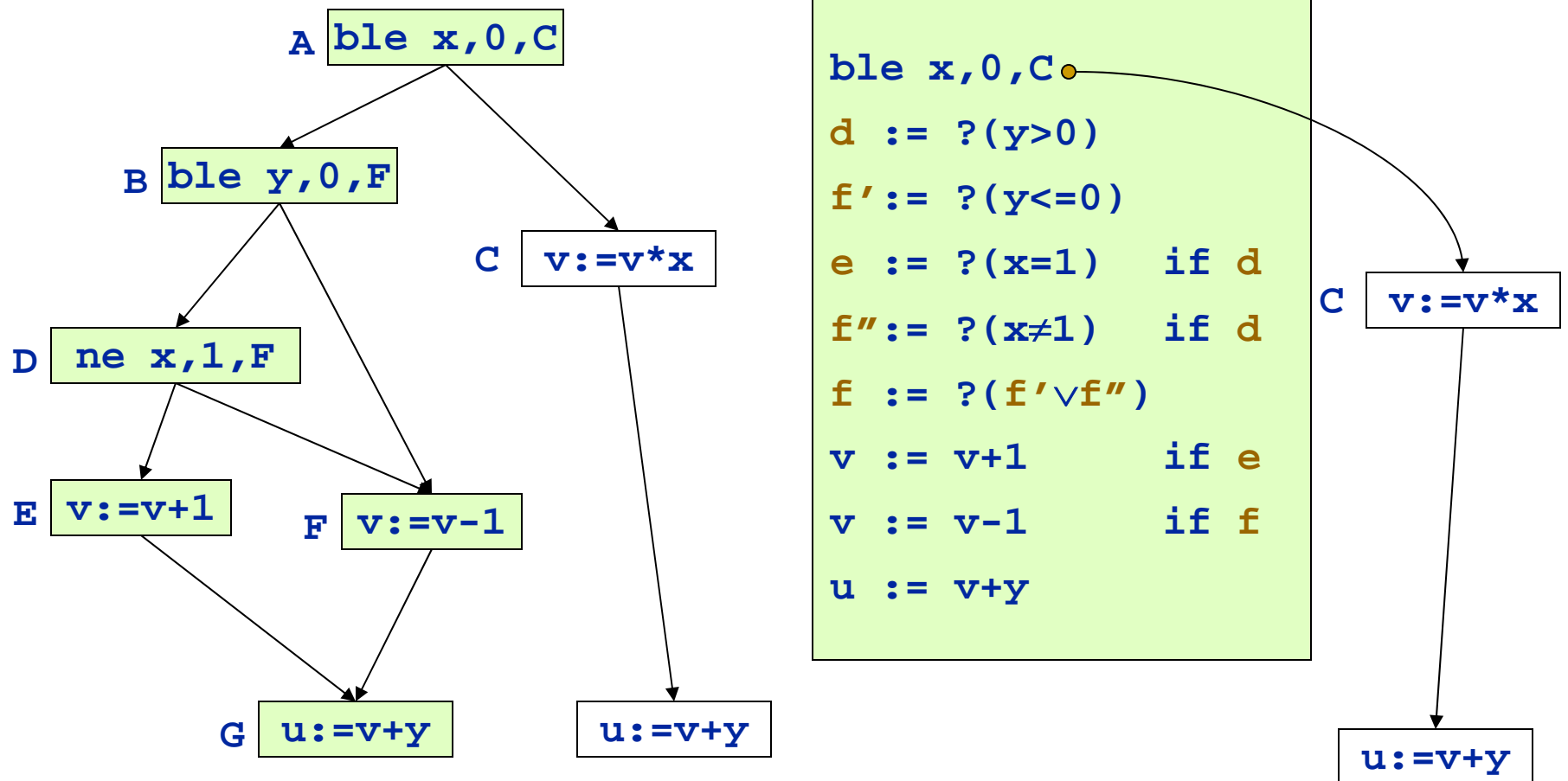
the point of the flow edge

- Large differences in FSV \Rightarrow unbalance control flow \rightarrow split those first

If-conversion Example: Assembly Code



If conversion



Region Size Control

- Experiments show that 85% of the execution time was contained in regions with fewer than 250 operations, when region size is not limited.
- There are some regions formed with more than 10000 operations. (May need limit)
- How can I decide the size limit?
 - Open Issue

Additional references

- Region Based Compilation: An Introduction and Motivation, *Richard Hank, Wen-mei Hwu, Bob Rau*, Micro-28, 1995
- Effective compiler support for predicated execution using the hyperblock, Scott Mahlke, David Lin, William Chen, Richard Hank, Roger Bringmann, Micro-25, 1992

Supplemental Readings

1. “All Shortest Routes from a Fixed Origin in a Graph”, G. Dantzig, W. Blattner and M. Rao, *Proceedings of the Conference on Theory of Graphs*, 85-90, July 1967.
2. “The Program Dependence Graph and its use in optimization,” J. Ferrante, K.J. Ottenstein and J.D. Warren, *ACM TOPLAS*, vol. 9, no. 3, 319-349, Jul. 1987.
3. “ The VLIW Machine: A Multiprocessor for Compiling Scientific Code”, J. Fisher, *IEEE Computer*, vol.7, 45-53, 1984.
4. The Superblock: An Effective Technique for VLIW and Superscalar Compilation”, W. Hwu, S. Mahlke, W. Chen, P. Chang, N. Warter, R. Bringmann, R. Ouellette, R. Hank, T. Kiyohara, G. Habb, J. Holm and D. Lavery, *Journal of Supercomputing*, 7(1,2), March 1993.
5. “Data Flow and Dependence Analysis for Instruction Level Parallelism”, B. Rau, *Proceedings of the Fourth Workshop on Language and Compilers for Parallel Computing*, August 1991.
6. “Some Scheduling Techniques and an Easily Schedulable Horizontal Architecture for High-Performance Scientific Computing”, B. Rau and C. Glaeser, *Proceedings of the 14th Annual Workshop on Microprogramming*, 183-198, 1981.



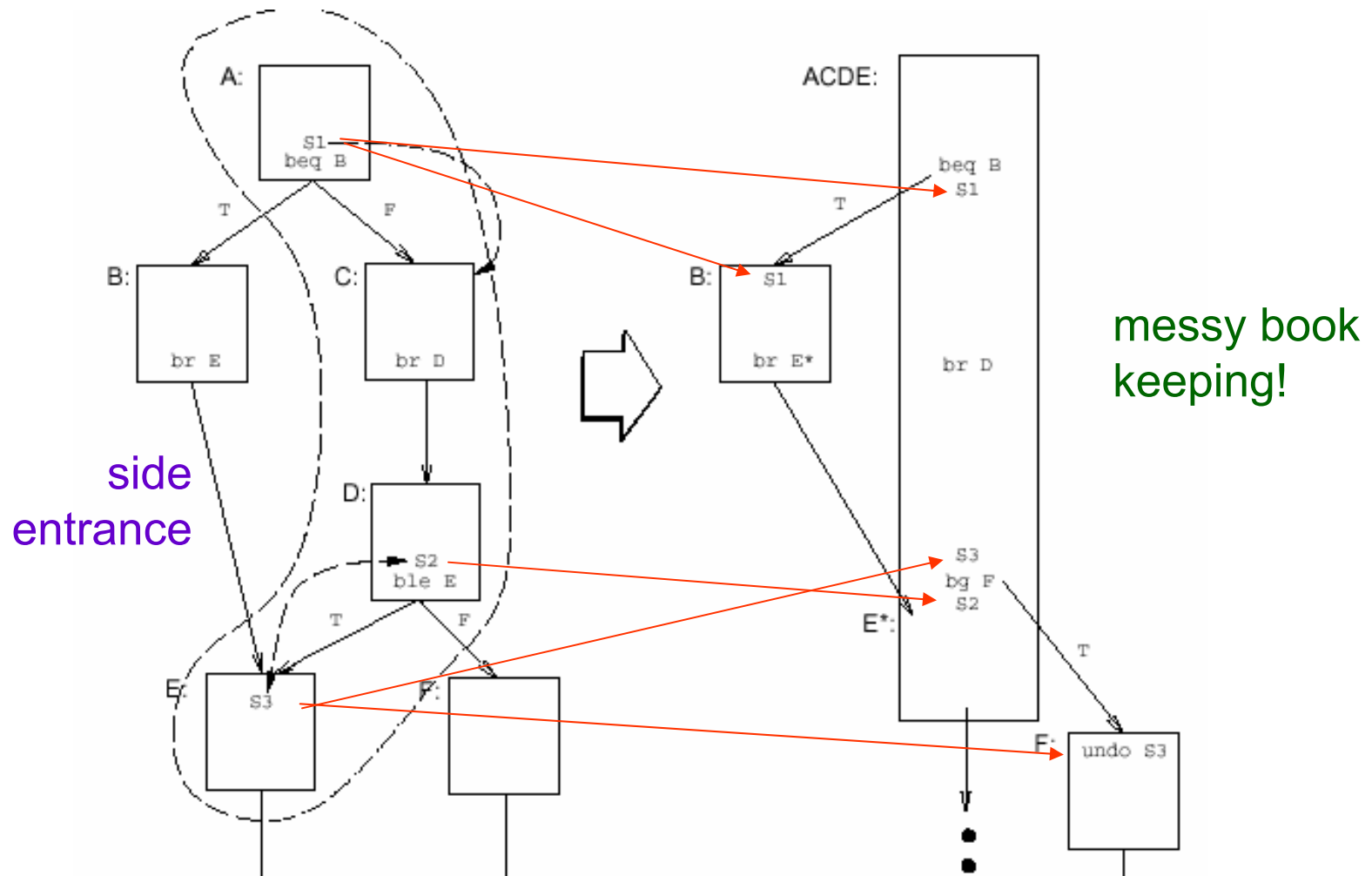
RICE

Appendix A: Superblock Formation

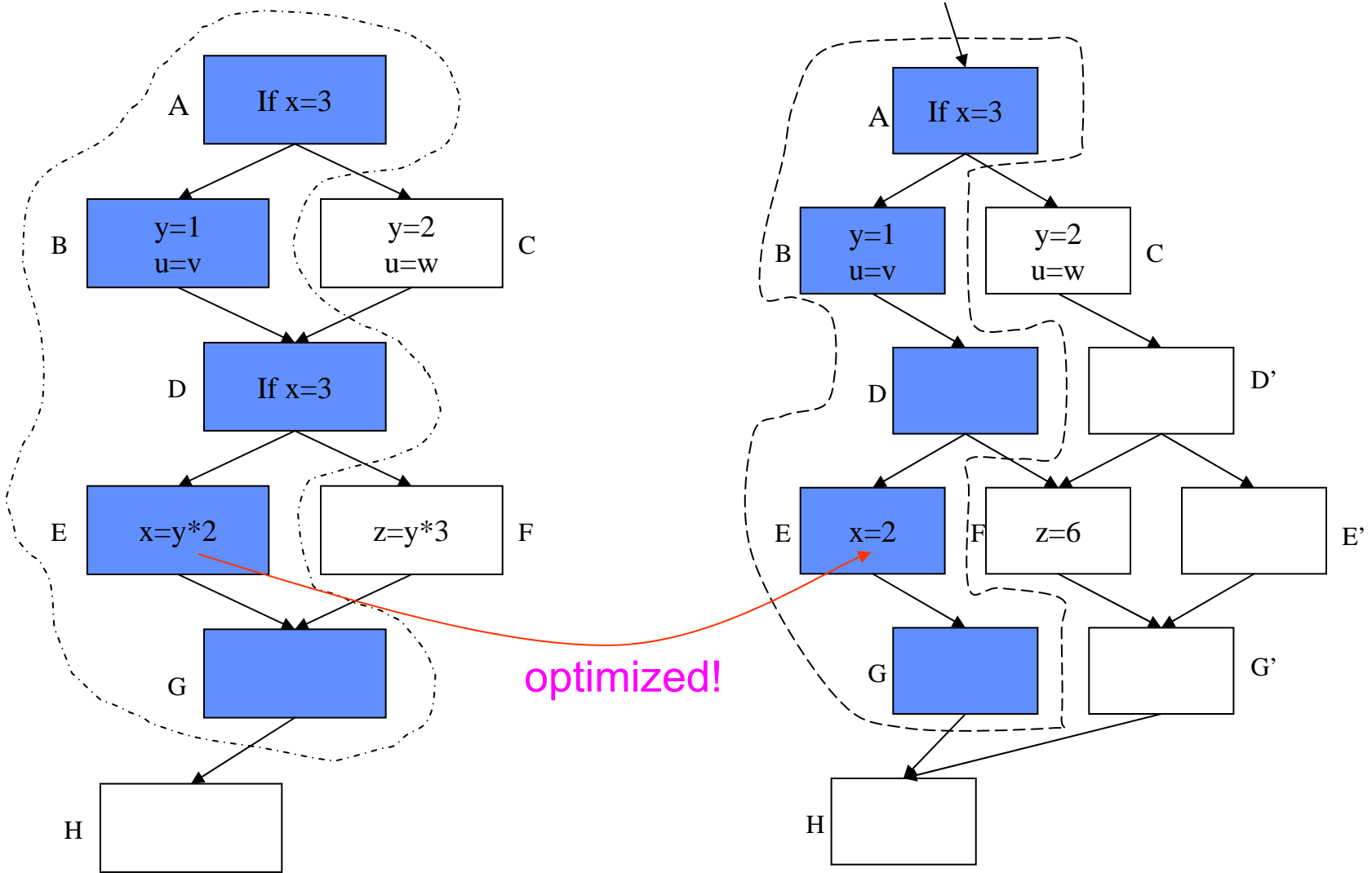
Building on Traces: Super Block Formation

- A trace with a single entry but potentially many exits
- Two step formation
 - Trace picking
 - Tail duplication - eliminates side entrances

The Problem with Side Entrance



Super Block Formation and Tail Duplication



Super Block: Implications to Code Motion

- Simplifies code motion during scheduling
 - Upward movements past a side exit within a block are pure speculation
 - Downward movements past a side exit within a block are pure replication
 - Downward movements past a side entry must be predicated, and replicated.
 - Eliminated via tail duplication
 - Upward movements past a side entry must be replicated and speculated.
 - Eliminated via tail duplication
- Super Blocks eliminate the more complex cases.