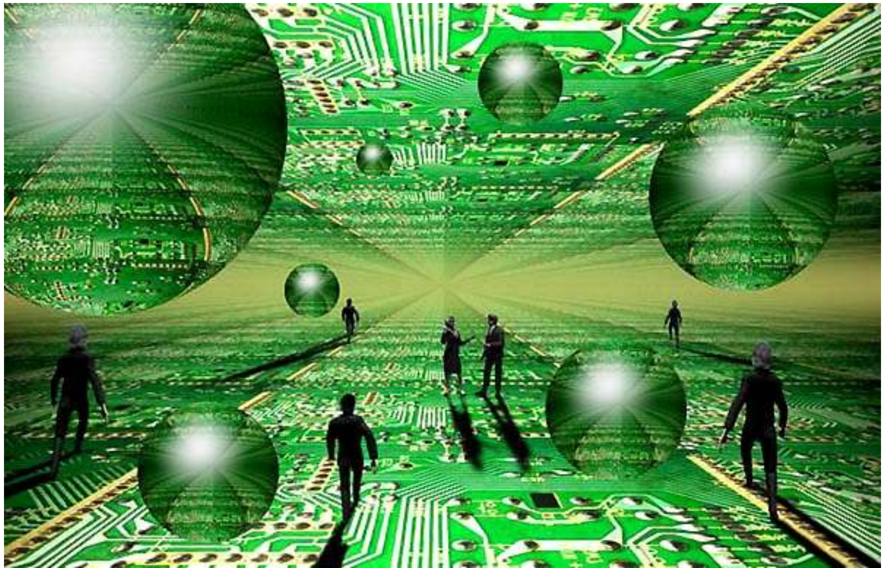
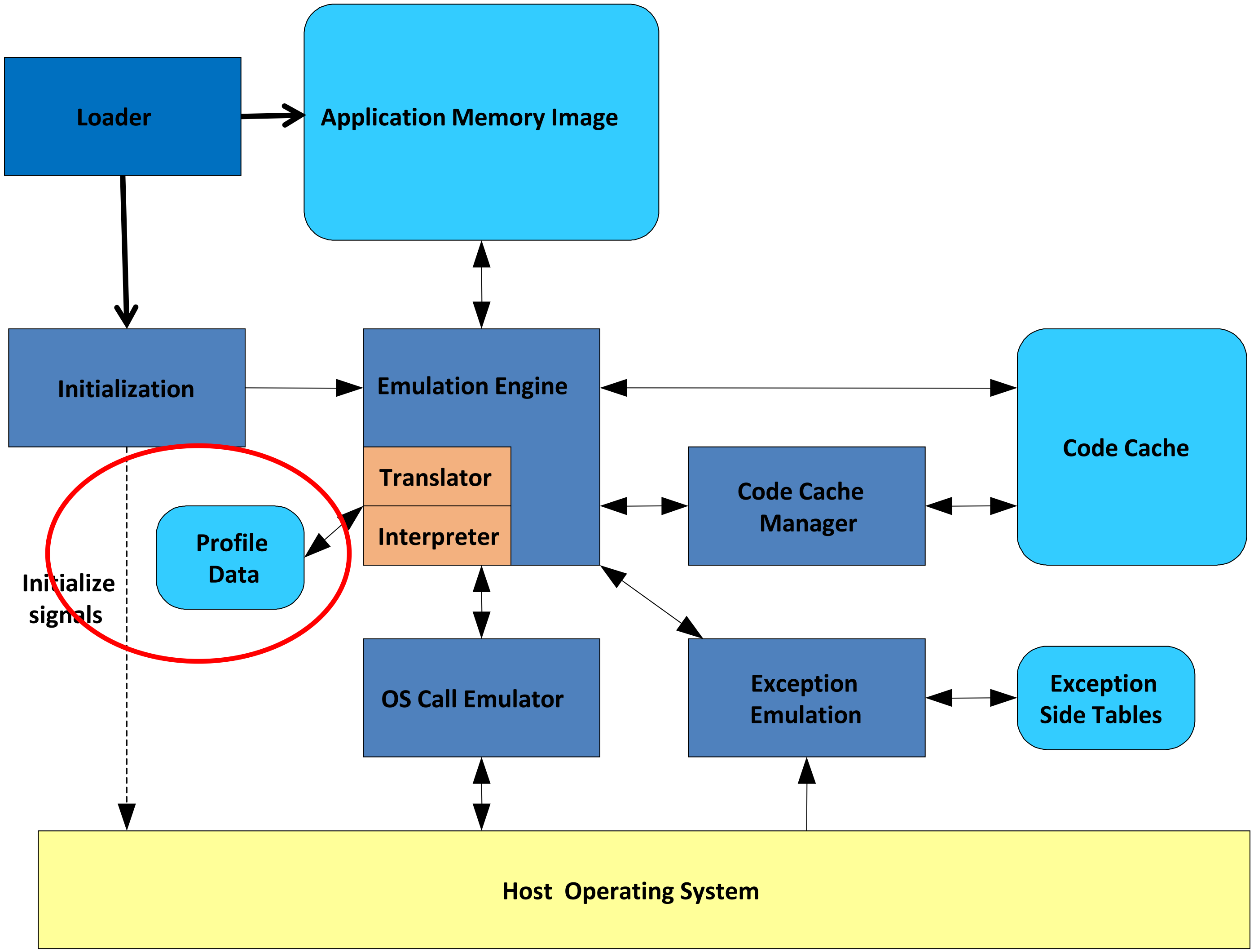
CSCI-GA.3033-015

Virtual Machines: Concepts & Applications

Lecture 4: Process VM - II

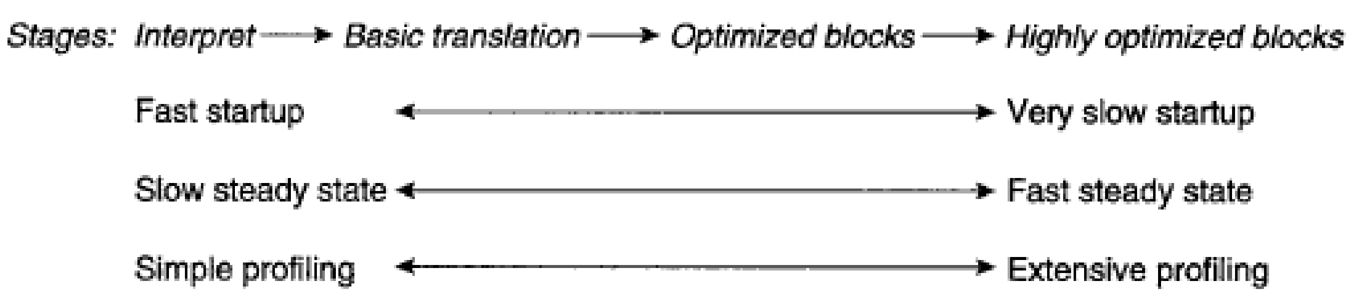
Mohamed Zahran (aka Z)
mzahran@cs.nyu.edu
http://www.mzahran.com



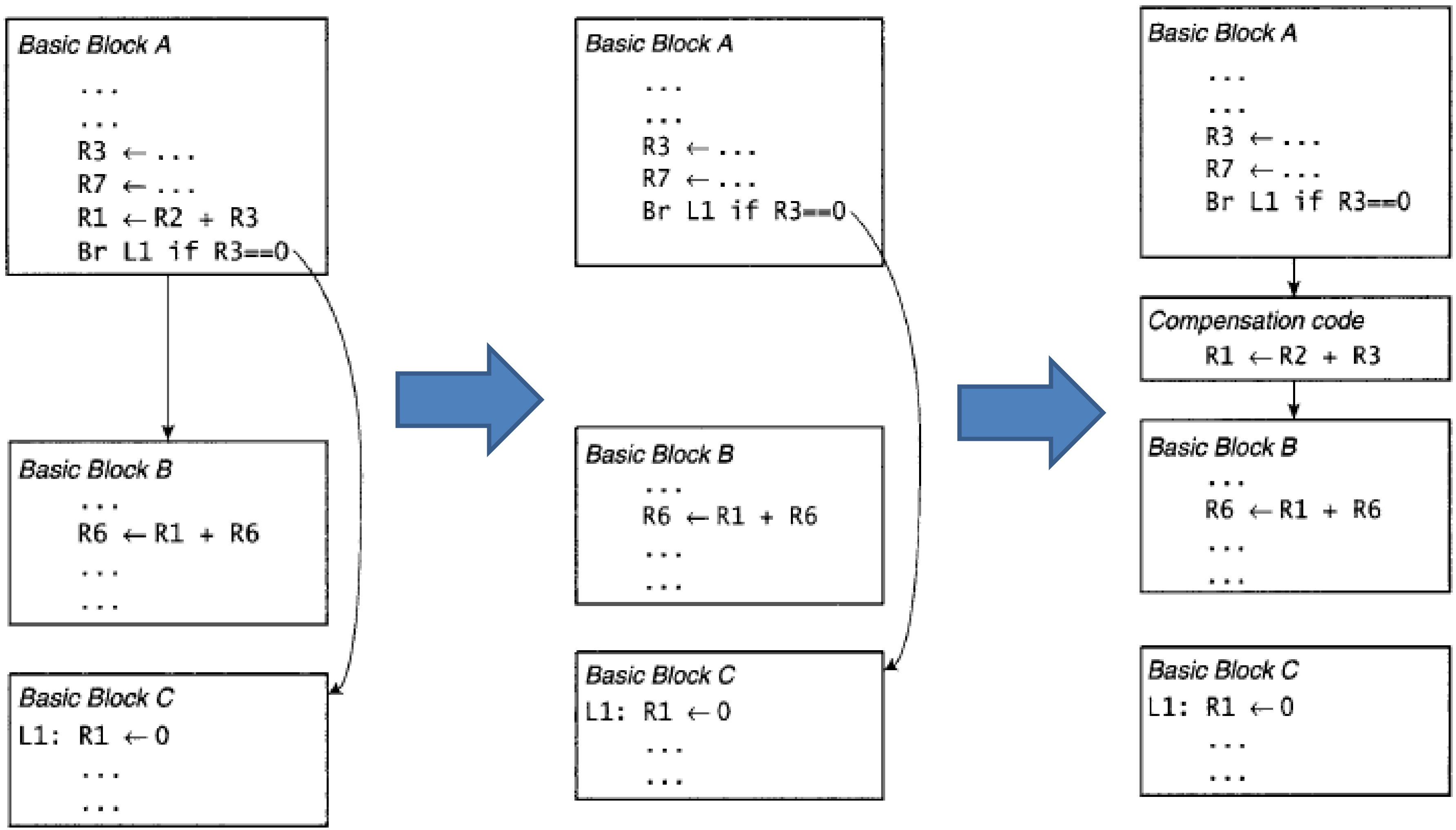

Loader
Application Memory Image
Initialization
Emulation Engine
Code Cache
Translator
Interpreter
Code Cache Manager
Profile Data
Initialize signals
OS Call Emulator
Exception Emulation
Exception Side Tables
Host Operating System

Profiling Directed Optimization

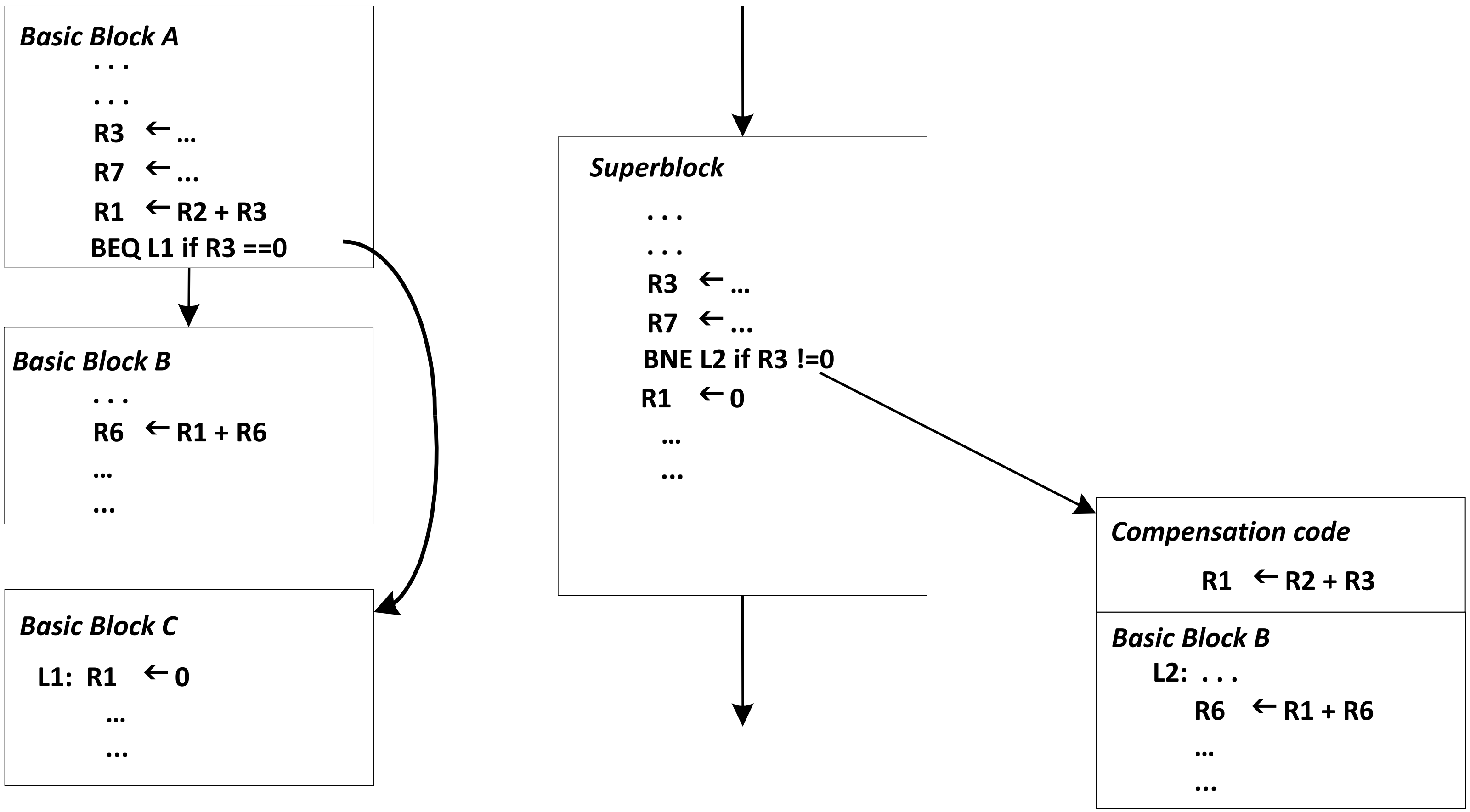
- Identify frequently executed *hot* code regions
  - Basic blocks
  - Paths
    - Better because it indicates control flow
  - Edges
    - Preferred approximation to paths
- Dynamic Profiling
  - Counts execution frequencies
  - Software implemented
  - Hardware implemented
  - Hybrids



Optimization Example



Another Optimization Example



Superblock

Program Behavior

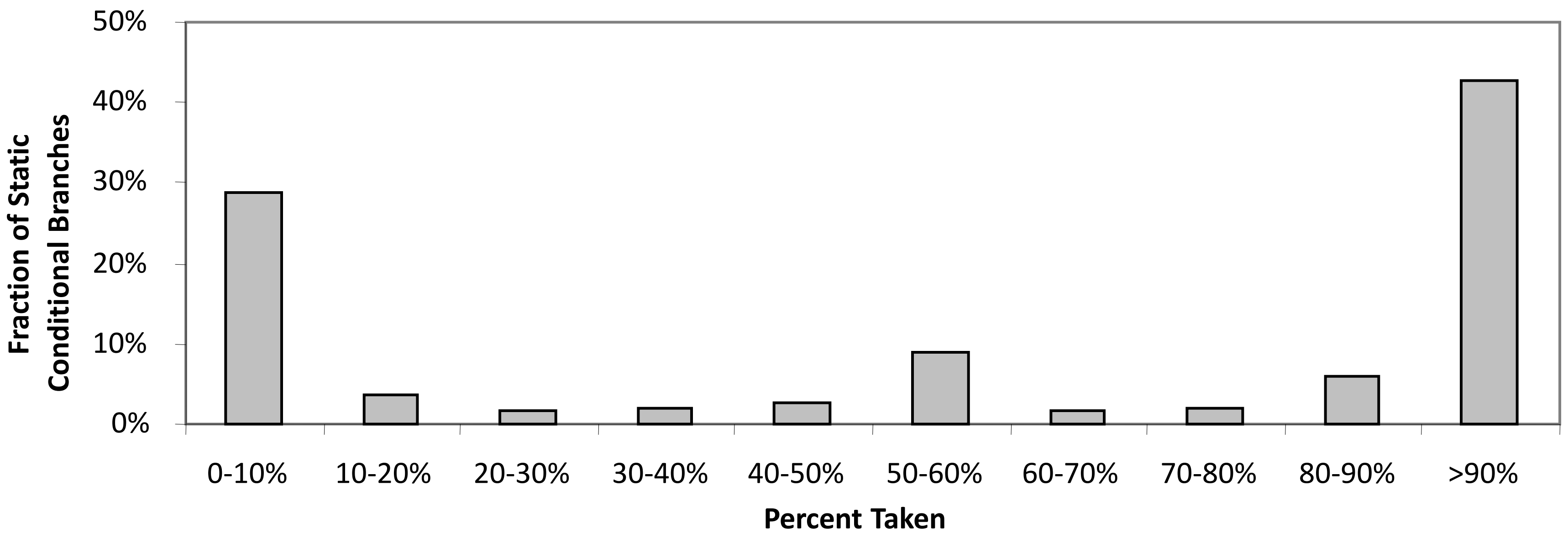
- Many aspects of program behavior are predictable
  - Based on history

```
R3 ← 100
loop:   R1 ← mem(R2)           ; load from memory
        Br found if R1 == -1   ; look for -1
        R2 ← R2 + 4
        R3 ← R3 -1
        Br loop if R3 != 0     ; loop closing branch
   .
found:
```

- Test for -1 primarily not taken
- Loop closing branch primarily taken

Branch Behavior

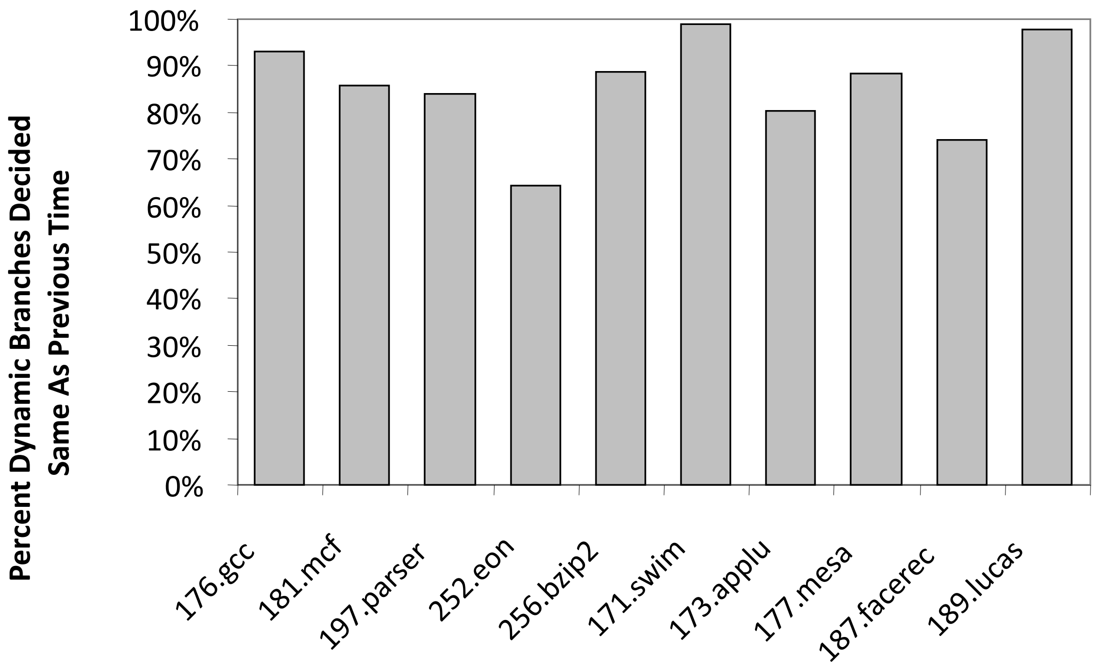
- A Conditional Branch is predominantly decided one way
  - Either taken or not taken



For SPEC benchmark suite

Branch Behavior

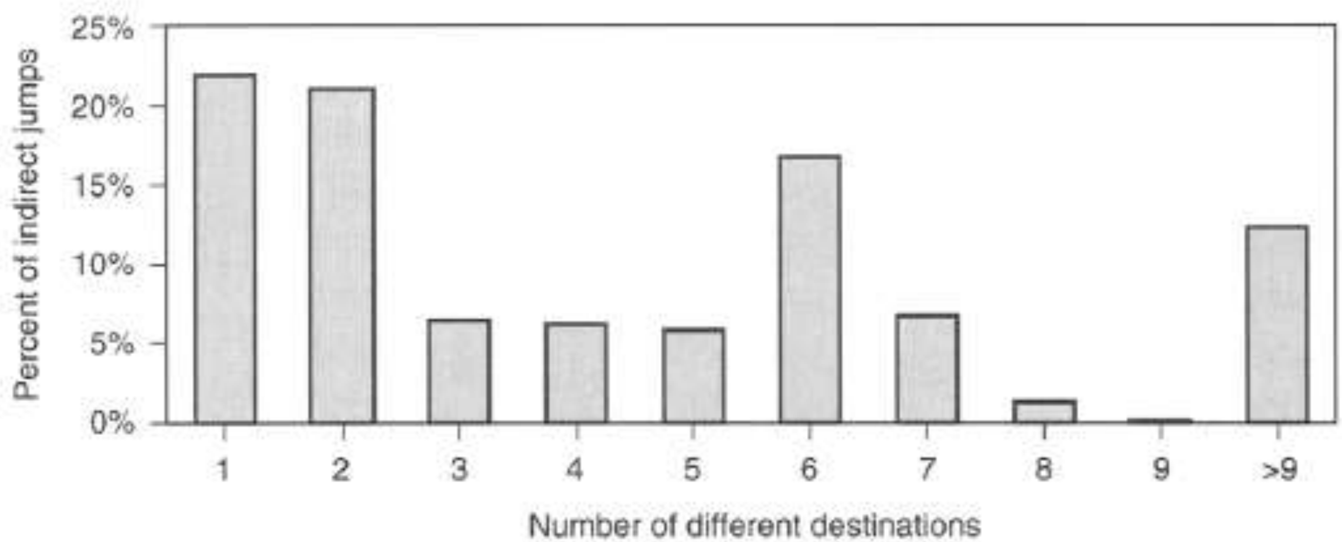
- Most branches are decided the same way as on previous execution
- Backward conditional branches are mostly taken
  - Forward conditional branches taken less often



For SPEC benchmark suite

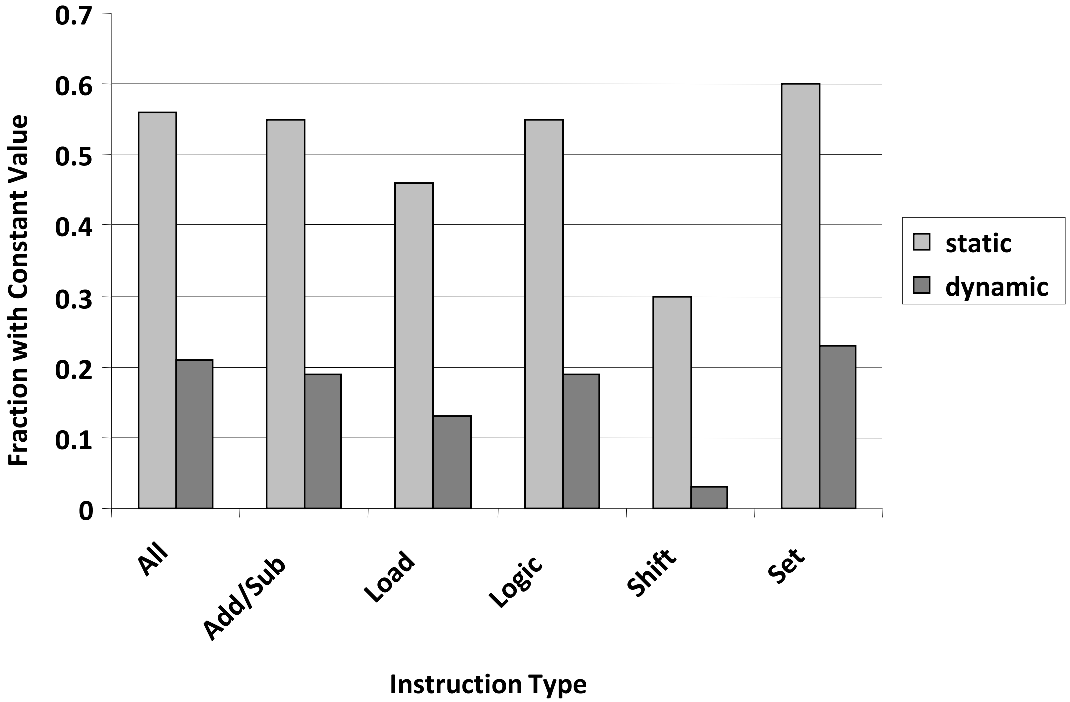
Program Behavior

- Some indirect jumps (i.e. target is stored in register) have a single target
  - Others have several targets (e.g. returns)



Program Behavior

- Predictability extends to data values
  - Many instructions always produce the same result

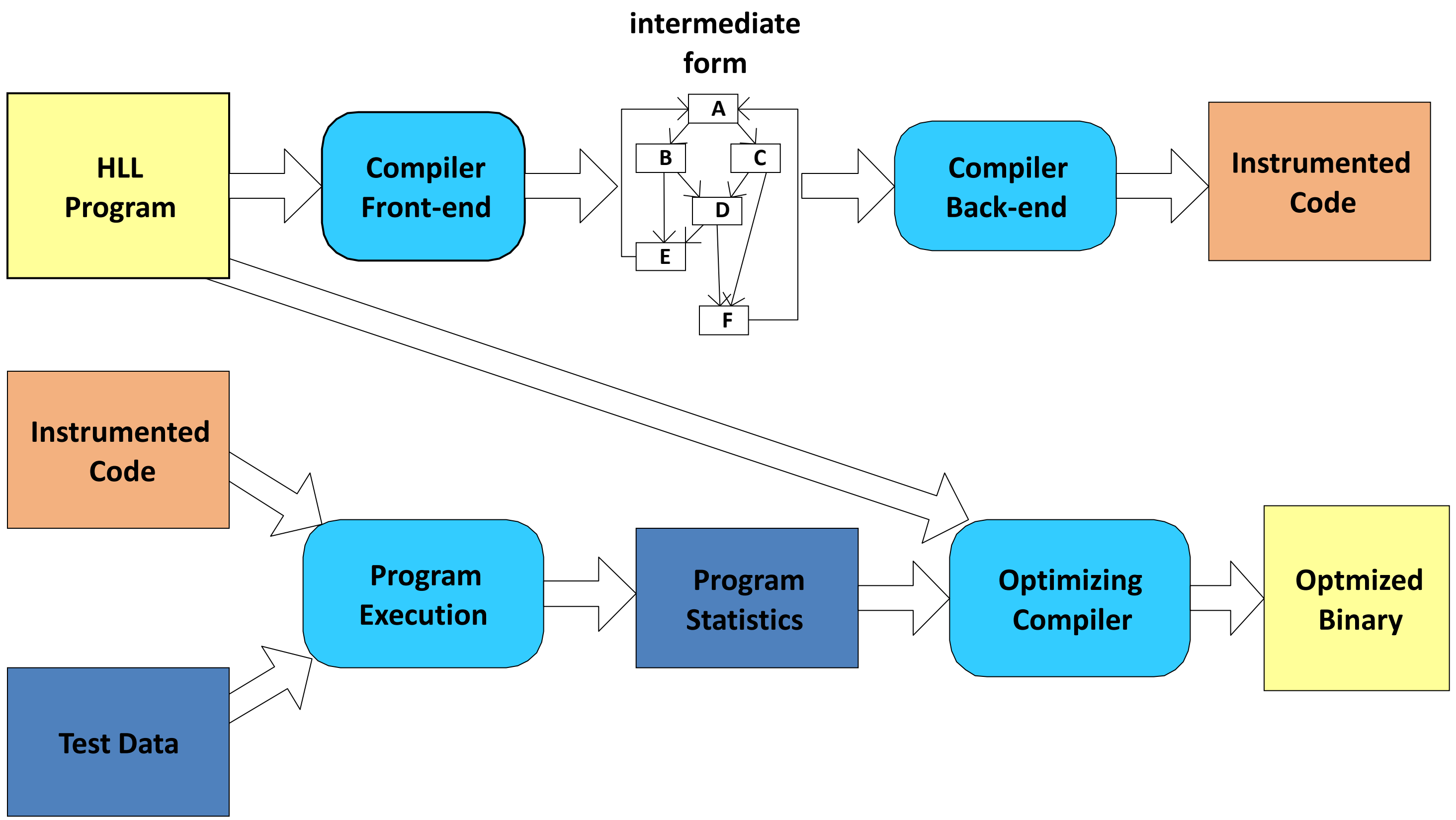


Profiling

- Collect statistics about a program as it runs
  - Branches (taken, not taken)
  - Jump targets
  - Data values
- Predictability allows these statistics to be used for optimizations to be used in the future
- Profiling in a VM differs from traditional profiling used for compiler feedback

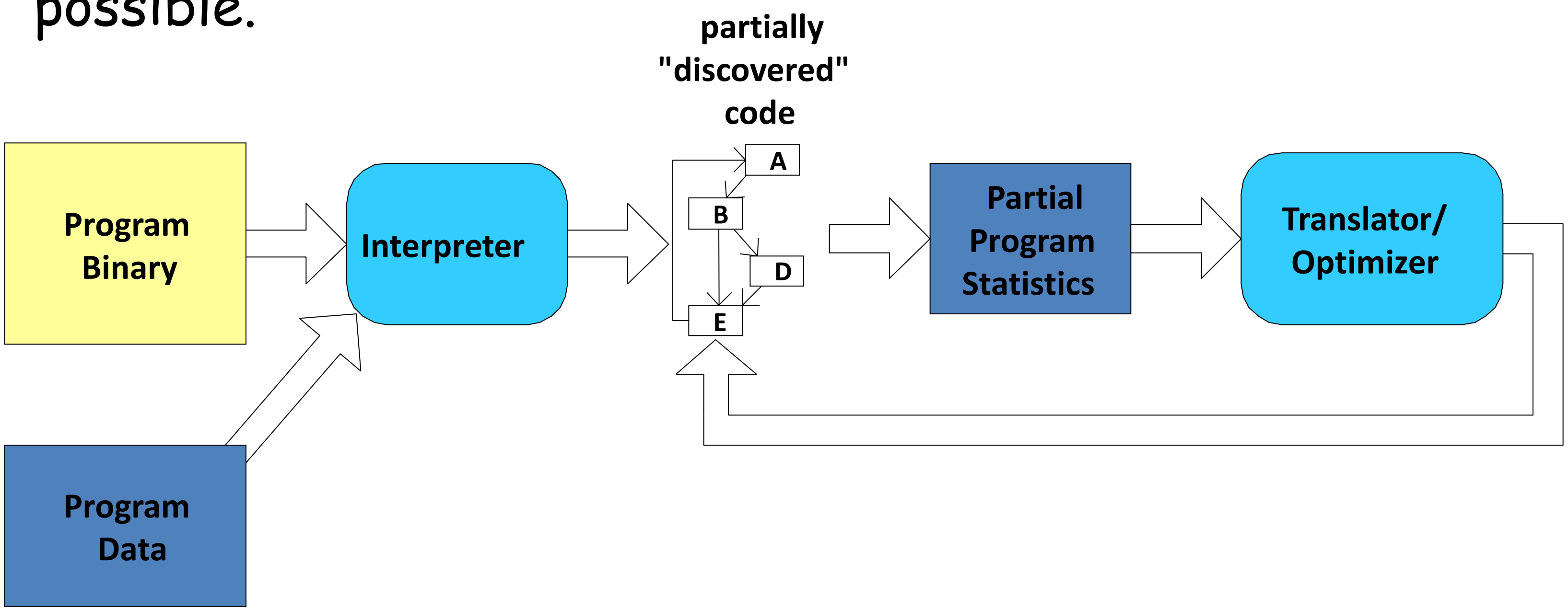
Conventional Profiling

- Multiple passes through compiler
- Done at program development time
  - Profile overhead is a small issue
- Can be based on global analysis



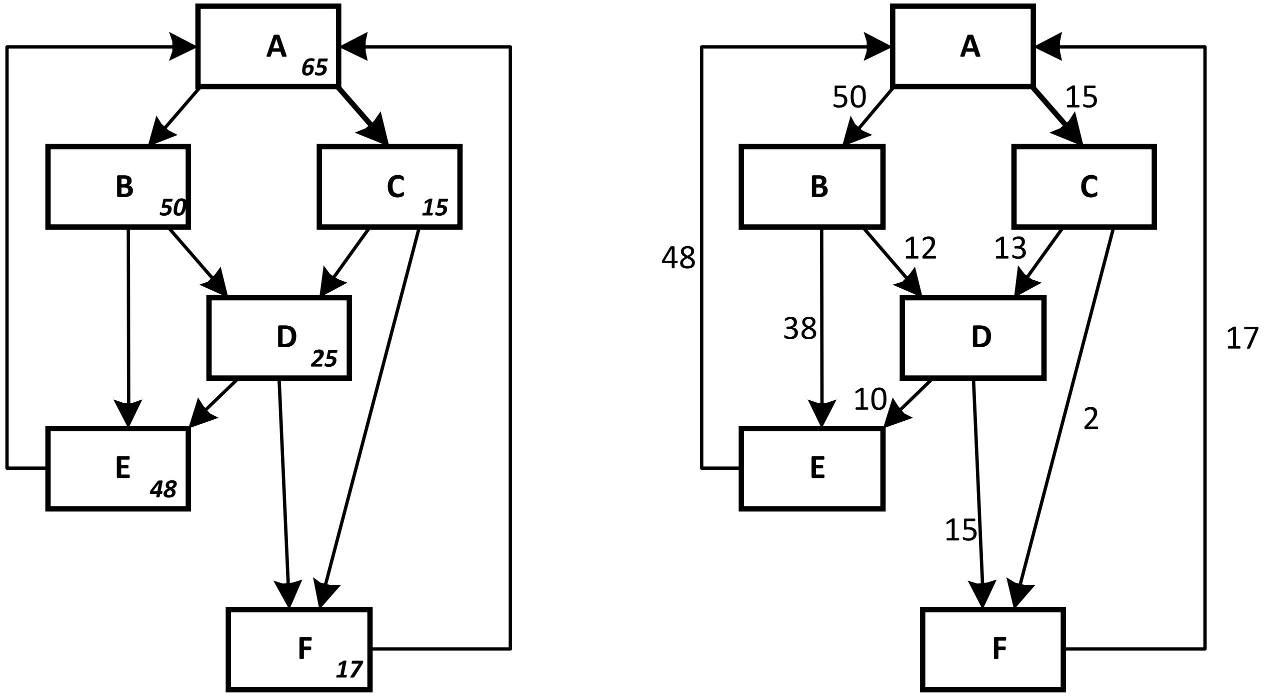
VM-Based Profiling

- Profile overhead is very important
  - Profile time comes out of execution time
- Limited view of program (no a priori global view)
  - Profile probes cannot be carefully placed
- Program characteristics must be determined as early as possible.



Types of Profiles

- Block or node profiles
  - Identify "hot" code blocks
  - Fewer nodes than edges
- Edge profiles
  - Give a more precise idea of program flow
  - Block profile can be derived from edge profile (not vice versa)

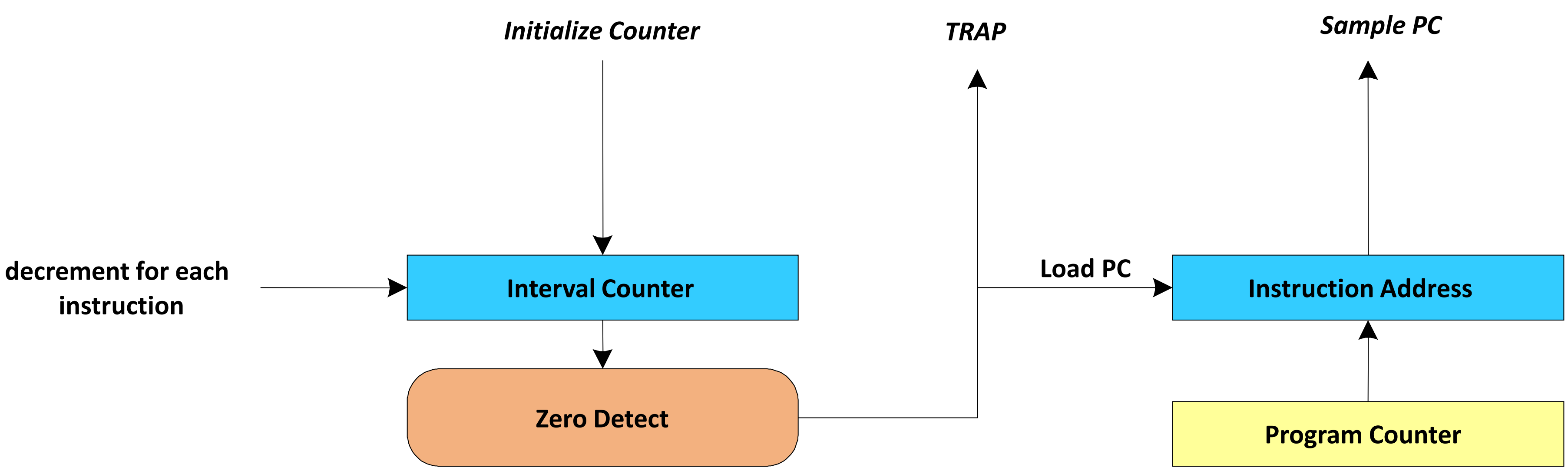


Collecting Profiles

- Instrumentation-based
  - Software probes
    - Slows down program more
    - Requires less total time
  - Hardware probes
    - Less overhead than software
    - Less well-supported in processors
    - Typically event counters
- Sampling based
  - Interrupt at random intervals and take sample
    - Slows down program less
    - Requires longer time to get same amount of data
  - Not useful during interpretation

Sampling

- Set interval counter
- Interrupt when counter hits zero
- Sample PC at that point
- Gives block profile
- Could be modified to give edge profile



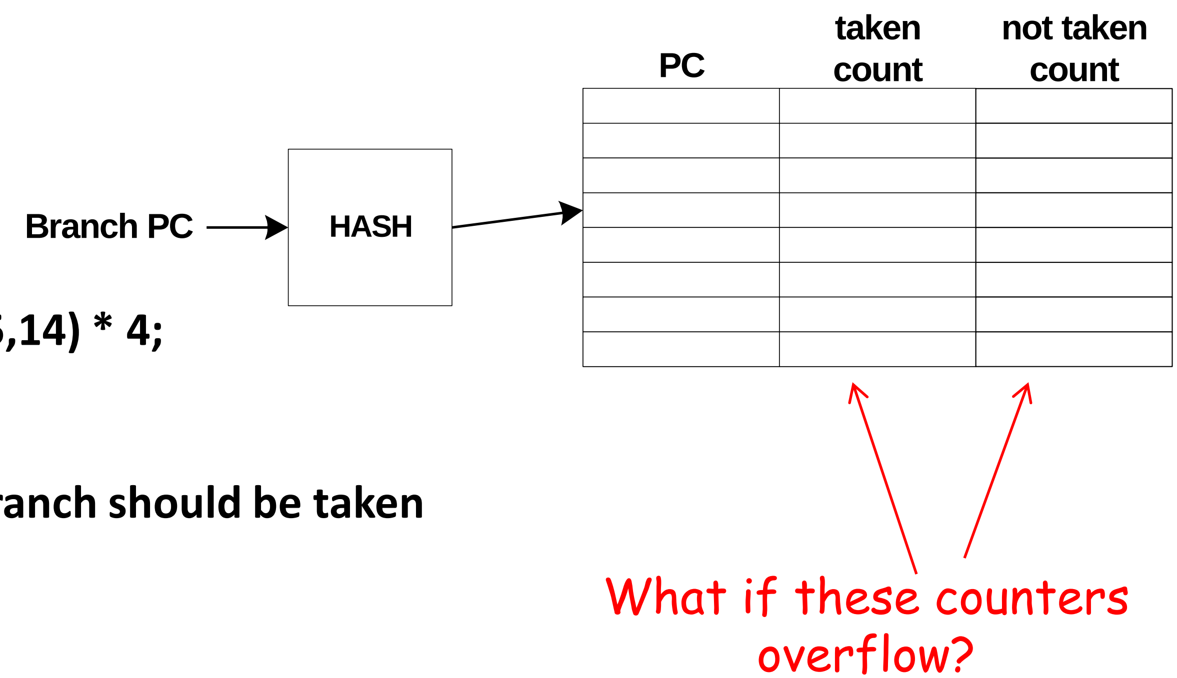
Profiling During Interpretation

- Source instructions are accessed as data.
- Interpreter routines are the code that is being executed.
- So: profiling code must be added to the interpreter routines.

Profiling During Interpretation

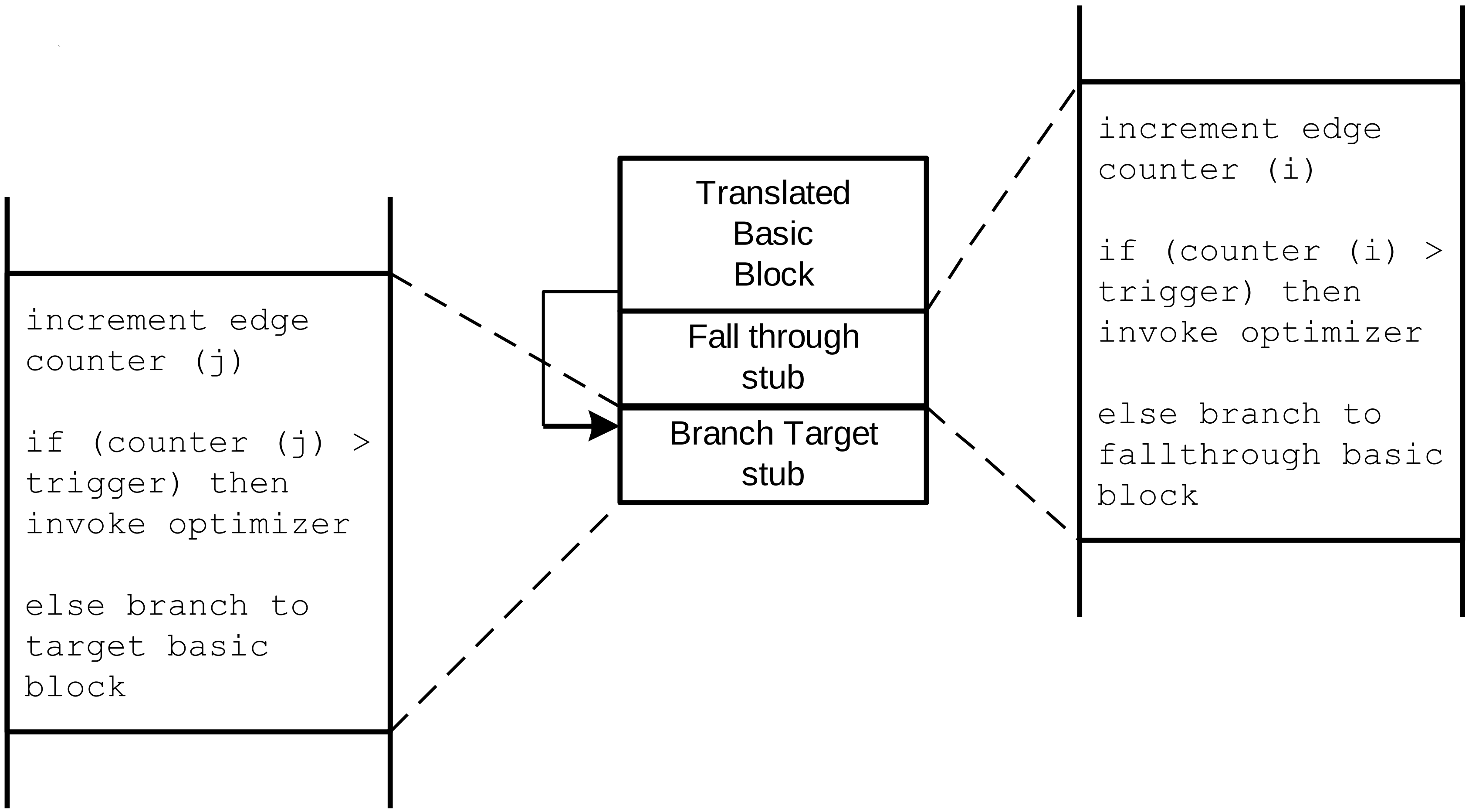
Instruction function list

```
.
branch_conditional(inst) {
  BO = extract(inst,25,5);
  BI = extract(inst,20,5);
  displacement = extract(inst,15,14) * 4;
  .
  .
// code to compute whether branch should be taken
  .
  .
  profile_addr = lookup(PC);
  if (branch_taken)
          profile_cnt(profile_addr, taken)++;
          PC = PC + displacement;
  Else
          profile_cnt(profile_addr,   nottaken)++;
                  PC = PC + 4;
}
```



Profiling Translated Code

- **Software Instrumentation in Stub Code**

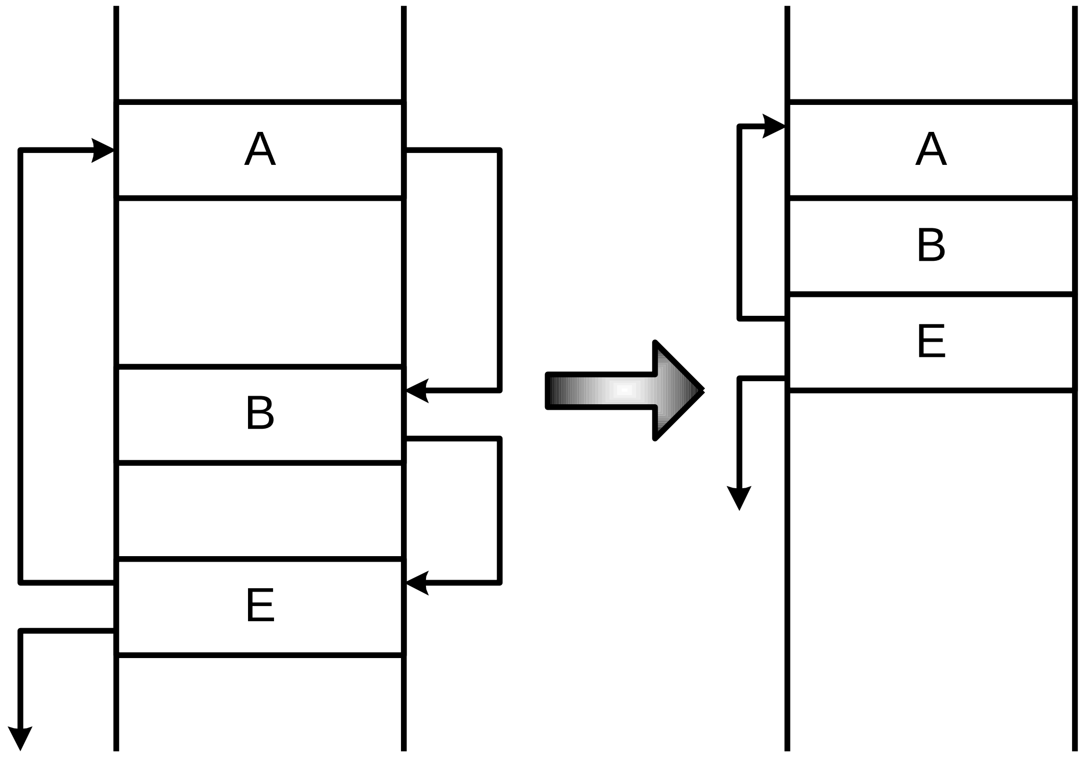


Now that we have profiling data, what can we do with it?

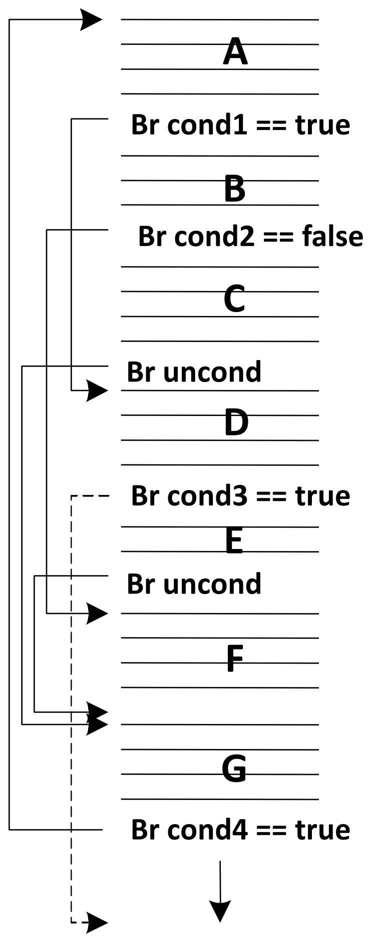
Strategies

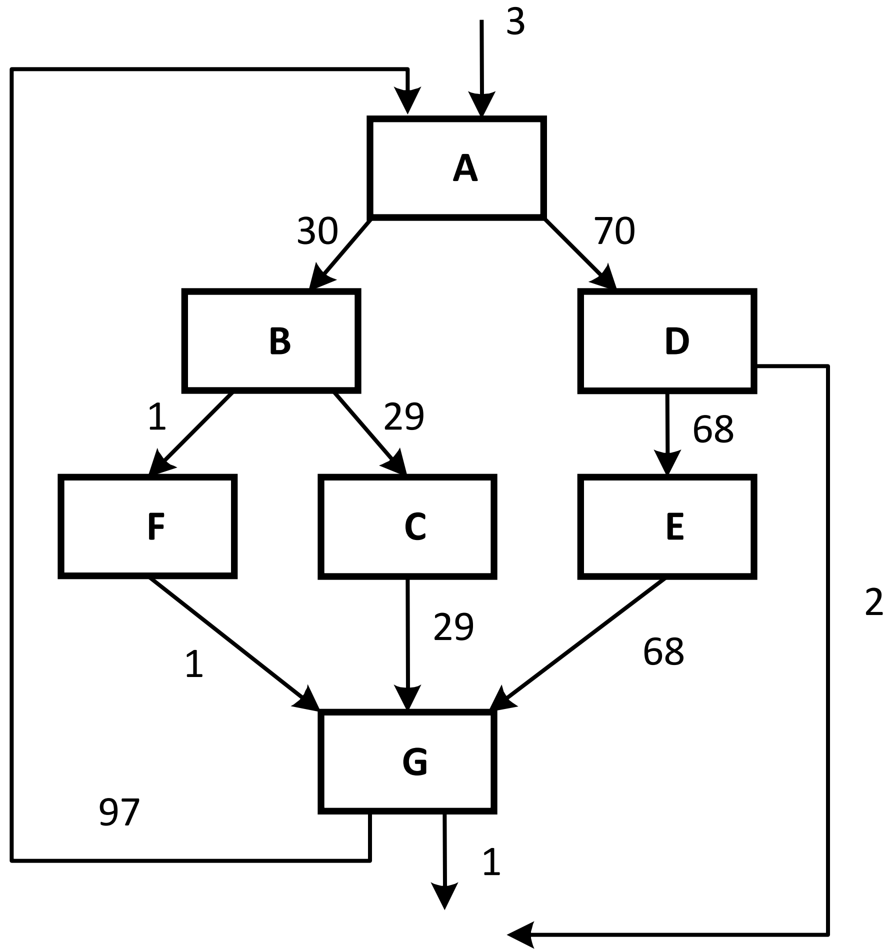
- Use our knowledge of control flow to put frequently followed sequences of basic blocks in contiguous memory locations to increase locality.
- Aggregate basic blocks into superblocks/traces/tree groups and optimize them.

Optimization: Improving Locality



Improving Locality: Example



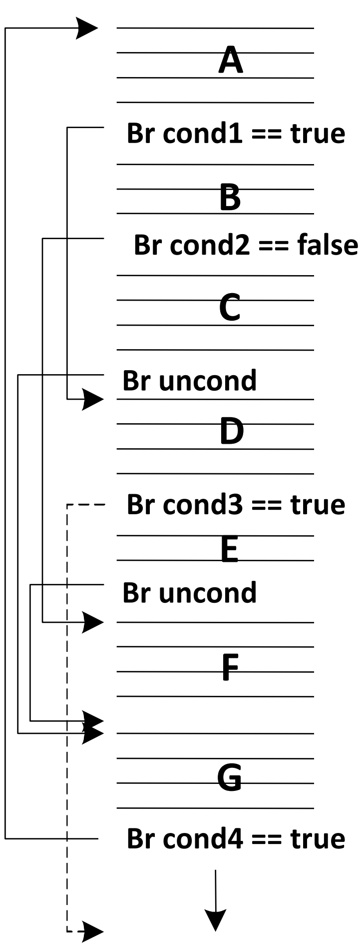


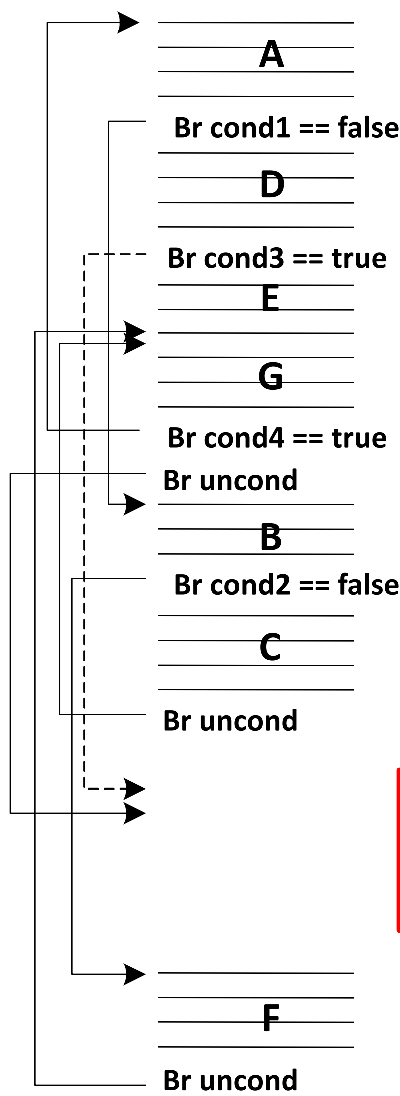
Improving Locality: Example

- Little locality (spatial or temporal) in cache line that spans blocks E and F
- F seldom used
  - Wasted I-cache space
  - Wasted I-fetch bandwidth
- Heavily used discontiguous code blocks
  - E.g., C and D
  - Still wastes I-fetch bandwidth

E Br uncond	F	F	F

Improving Locality: Rearrange Code

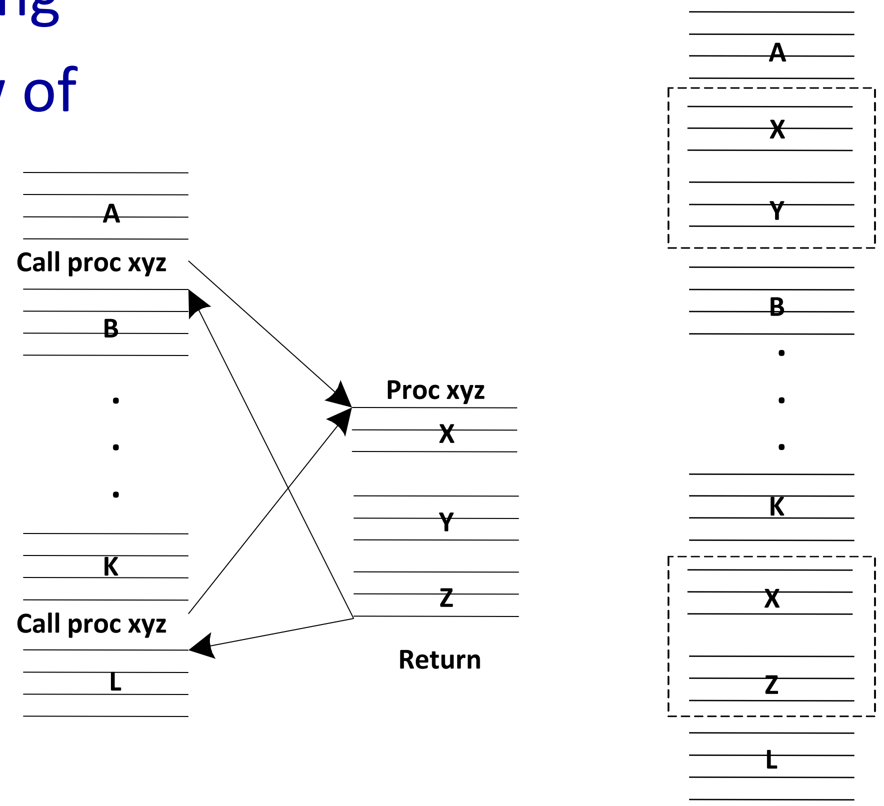




1. Decide on blocks arrangement
2. Update branches accordingly

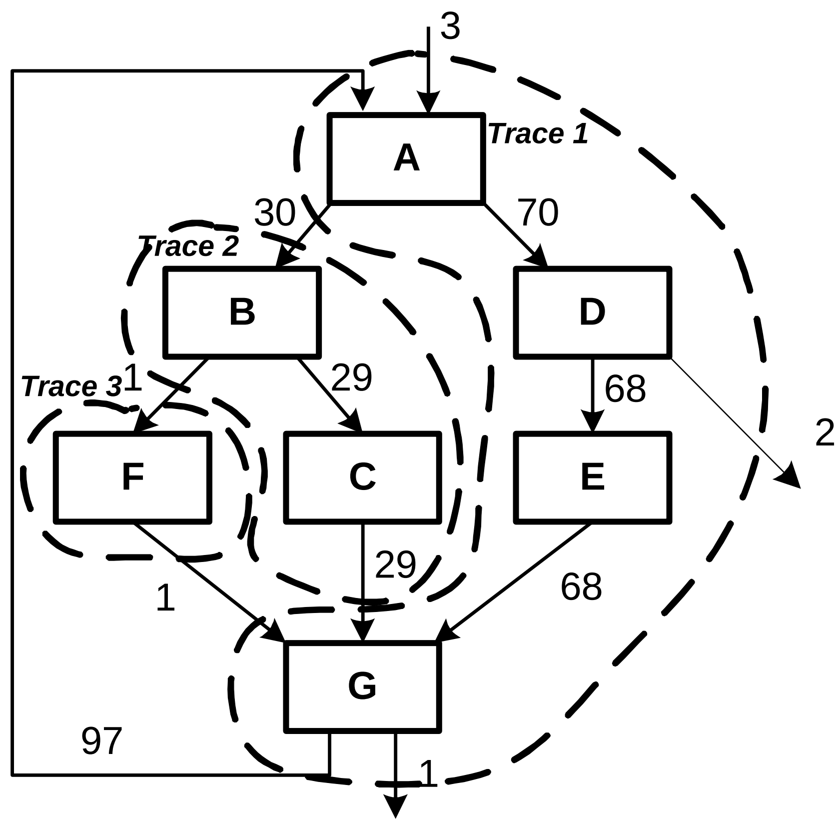
Improving Locality: Procedure Inlining

- **Partial inlining**
  - Unlike static full inlining
  - Follow dominant flow of control



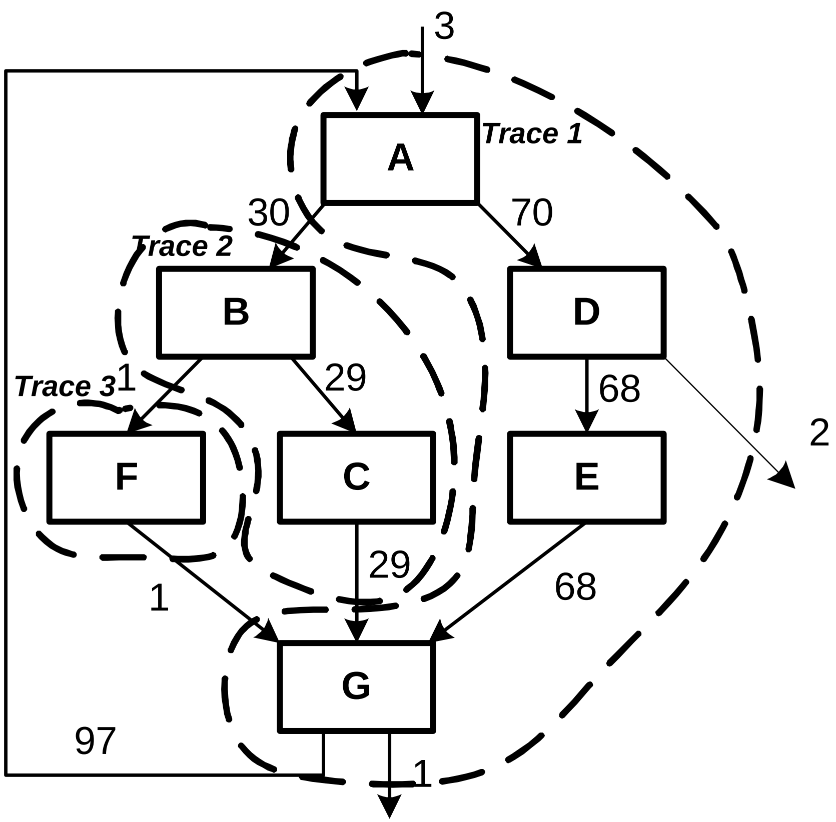
Improving Locality: Traces

- Proposed by Fisher (Multiflow)
  - Used overall profile/analysis
- Greedy Method
  - Suitable for on-the-fly
  - Start at hottest block not yet in a trace
  - follow hottest edges
  - Stop when trace reaches a certain size
  - Stop when a block already in a trace is reached



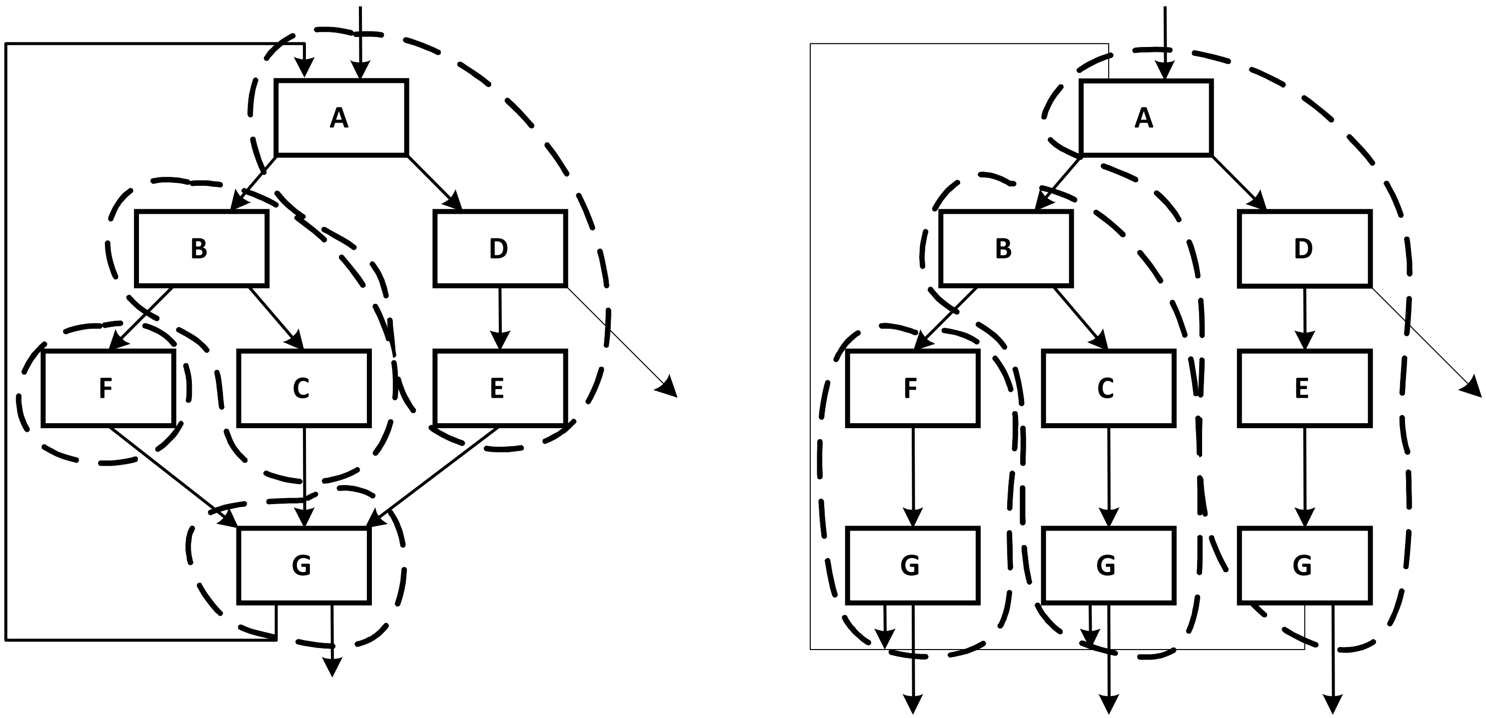
Traces, contd.

- No redundancy
  - Good for spatial locality
  - Not good for temporal locality
- Typically not used in optimizing VMs

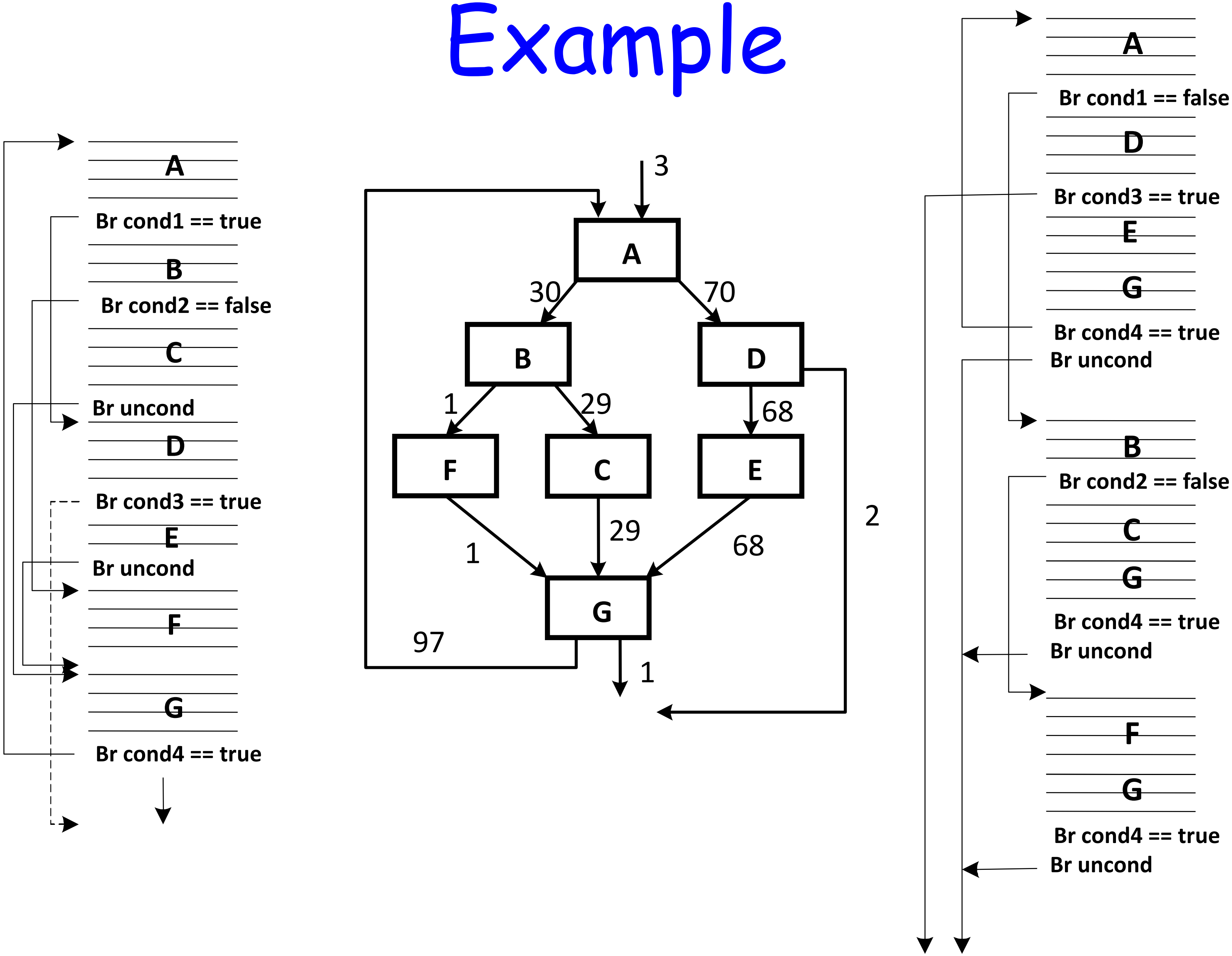


Improving Locality: Superblocks

- One entry multiple exits
- May contain redundant blocks (tail duplication)
- More commonly used by dynamic optimizers than traces



Example

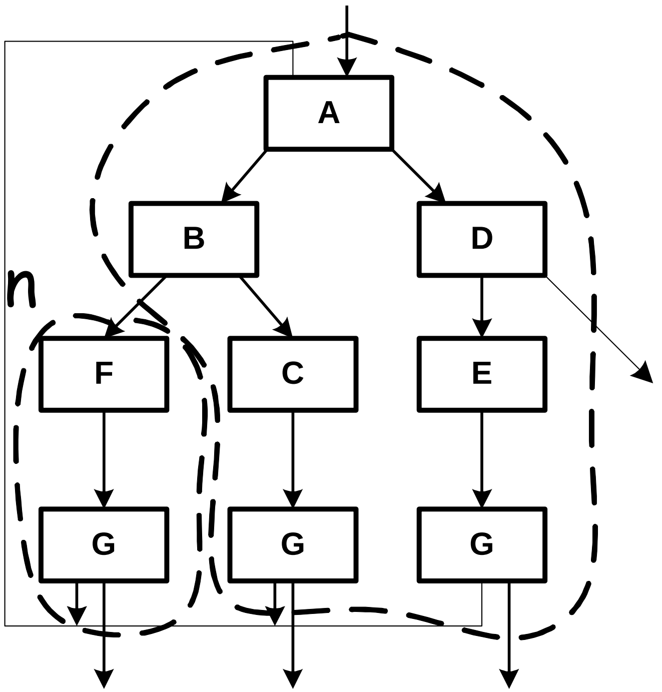

A
Br cond1 == true
B
Br cond2 == false
C
Br uncond
D
Br cond3 == true
E
Br uncond
F
G
Br cond4 == true
3
A
30
70
B
D
1
29
68
F
C
E
1
29
68
2
G
97
1
A
Br cond1 == false
D
Br cond3 == true
E
G
Br cond4 == true
Br uncond
B
Br cond2 == false
C
G
Br cond4 == true
Br uncond
F
G
Br cond4 == true
Br uncond

Superblock Formation

- Start Points
  - When block use reaches a threshold
  - Profile all blocks
  - Profile selected blocks
    - Profile only targets of backward branches (close loops)
    - Profile exits from existing superblocks
- Continuation
  - Use hottest edges above a (second) threshold
  - Follow current control path (most recent edge)
- End Points
  - Start point of this superblock
  - Start point of some other superblock
  - When a maximum size is reached
  - When no edge above threshold can be found
  - When an indirect jump is reached (depends on whether inlining is enabled)

Tree Groups (Tree Regions)

- Generalization of Superblocks
  - One entrance
  - Several exits
  - Several flows of control
- Good when one branch direction is not dominant
- Larger scope for optimization
- Good for predication
  - Merge alternate paths

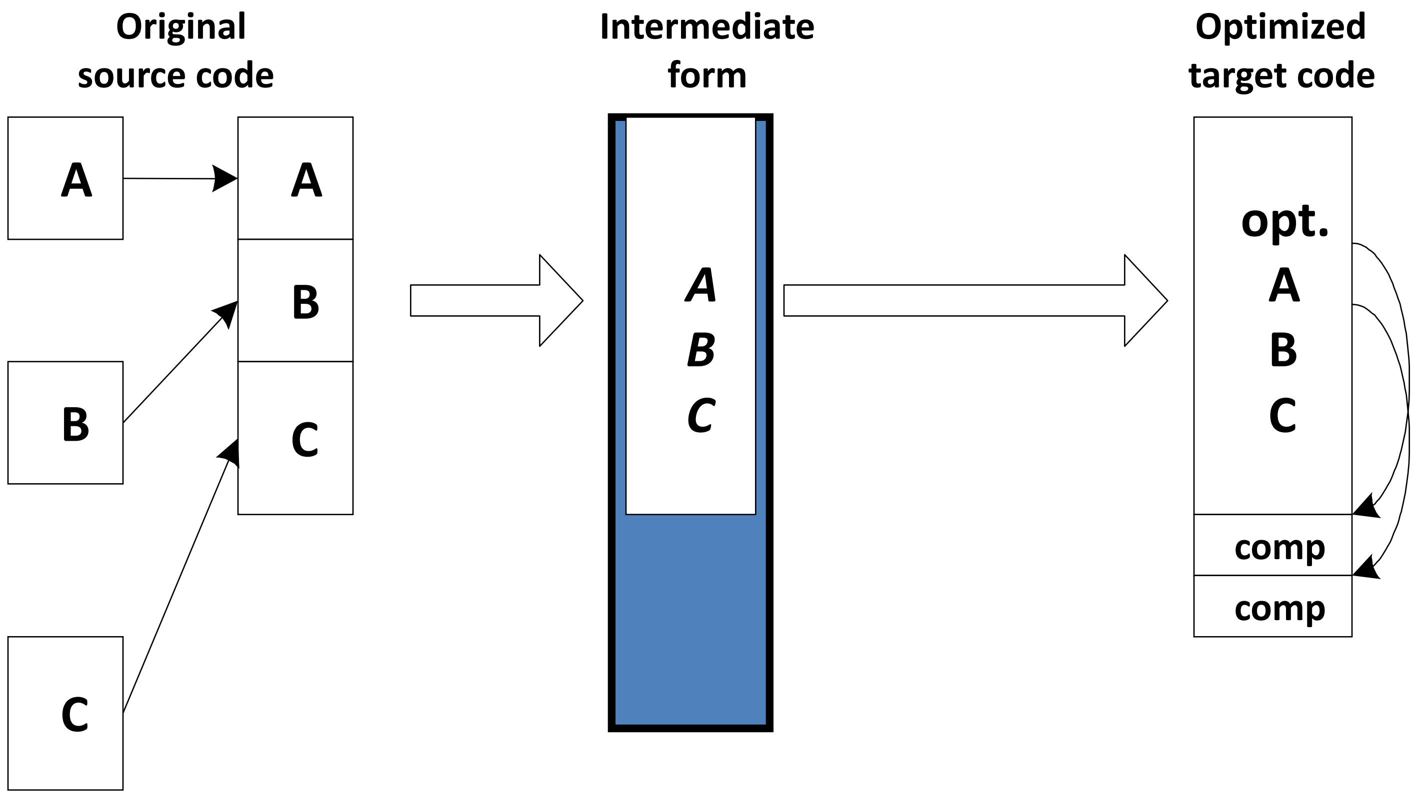


Now that we have superblocks, tree groups, etc. What do we do with them?

Static versus Dynamic Optimization

- With Static Optimization
  - More time for analysis (done offline)
    - Profiling/Opt. overhead does not add to total execution time
    - Can place profile probes more carefully
    - Can analyze results more carefully
- With Dynamic Optimization
  - Often use simpler, less optimal methods

Dynamic Optimization Overview



Collect basic blocks using profile information

Convert to intermediate form; place in buffer

Schedule and optimize

Generate target code

Add compensation code; place in code cache

Code Scheduling

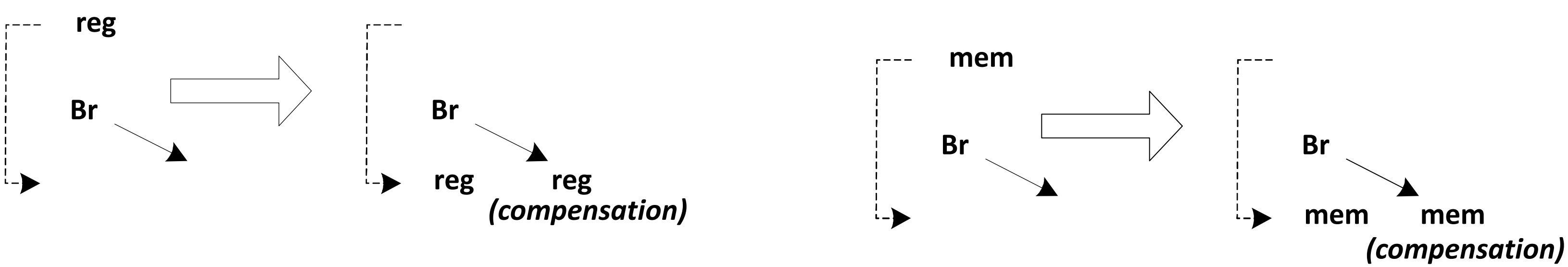
- Order code for better performance
- An important optimization in many VMs
  - Especially if host platform is in-order issue or VLIW
- We first will consider scheduling at a “micro” level
  - Consider code movement of specific instruction types
  - Instruction Types:
    - REG: Register updates
    - includes loads
    - later we separate trapping and non-trapping
    - MEM: Memory updates and volatile load/stores
    - BR: Branches and Jumps
    - JOIN: Join points

“Micro” Code Scheduling

- Example Code Sequence

. . .	
R1 ←mem(R6)	reg
R2 ← mem(R6 +4)	reg
R3 ← R1 + 1	reg
R4 ← R1 << 2	reg
Br exit; if R7 == 0	br
R7 ← R7 + 1	reg
mem (R6) ← R3	mem

Moving Code Below Branches

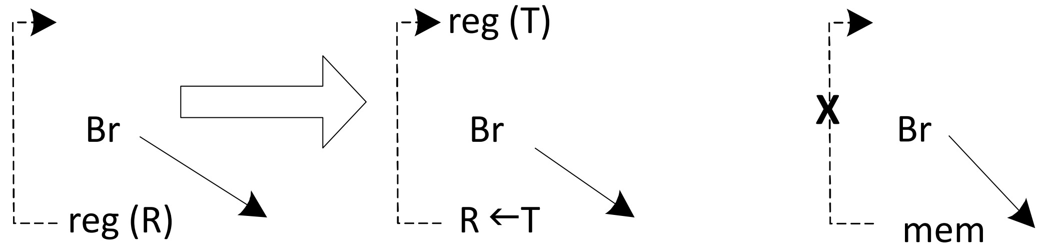



```
. . .
R1  ← mem(R6)
R2  ← mem(R6 +4)
R3  ← R1 + 1
R4  ← R1 << 2
Br exit; if R7 == 0
R7  ← R7 + 1
mem (R6)  ← R3
```

```
. . .
R1  ← mem(R6)
R2  ← mem(R6 +4)
R3  ← R1 + 1
Br exit; if R7 == 0  →  R4  ← R1 << 2
R4  ← R1 << 2
R7  ← R7 + 1
mem (R6)  ← R3
```

- Generally straightforward
- Compensation is via duplication

Moving Code Above Branches




```
. . .
R2  ← R1 << 2
Br exit; if R8 == 0
R6  ← R7 * R2
mem (R6)  ← R3
R6  ← R2 + 2
```

```
. . .
R2  ← R1 << 2
T1  ← R7 * R2
Br exit; if R8 == 0
R6  ← T1
mem (T1)  ← R3
R6  ← R2 + 2
```

```
. . .
R2  ← R1 << 2
T1  ← R7 * R2
Br exit; if R8 == 0
mem (T1)  ← R3
R6  ← R2 + 2
```

Moving Code Above Branches

- For reg instructions, "checkpoint"
  - Keep old value live in a temporary register
  - If exit branch is taken, mapped register does not get modified
  - If instruction traps, backup and interpret forward
- Moving store breaks memory state compatibility
  - E.g. what if exit branch is taken?

Conclusions

- Profiling is crucial to ensure acceptable performance for VMs
- The profiling data we gather depends on the type of optimizing we want to do.
- Usually we follow the following steps:
  1. Gather profiling data
  2. Form superblocks
  3. Optimize
- One of the most commonly used form of optimization is code reordering.
- Remember that till now we were working at the ISA level