Dynamic Binary Optimization

Contents

- Overview of Applying Optimization on VMs
- Dynamic Program Behavior
- Profiling
- Optimizing Translation Blocks

Classical Optimizations

```
addl    %edx, 4(%eax)
movl    4(%eax), %edx
```

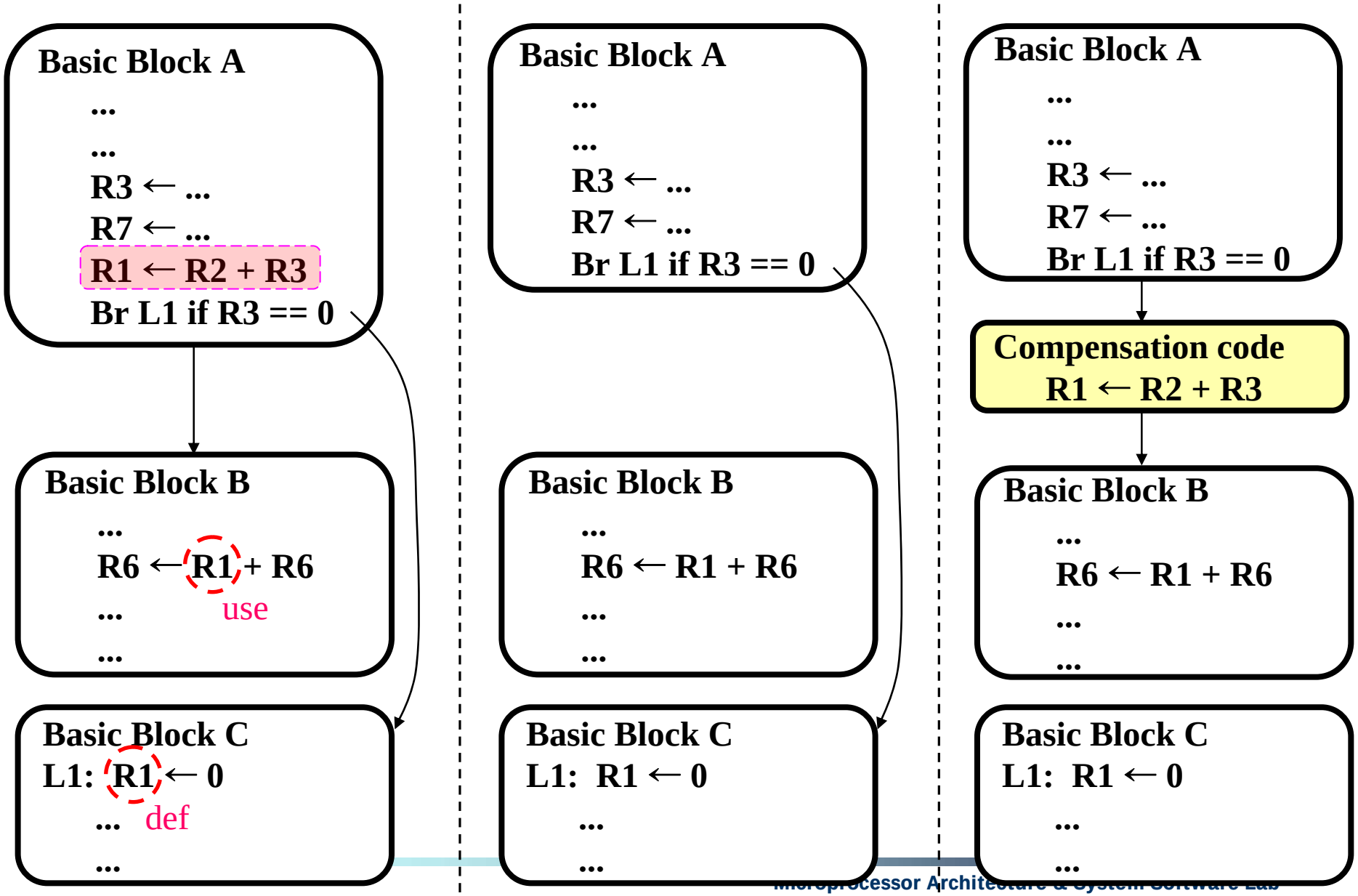
Translation from IA-32 to PowerPC code.

```
addi    r16, r4, 4          ; add 4 to %eax
lwzx    r17, r2, r16        ; load operand from memory
add     r7, r17, r7         ; perform add of %edx
addi    r16, r4, 4          ; add 4 to %eax
stwx    r7, r2, r16         ; store %edx value into memory
```

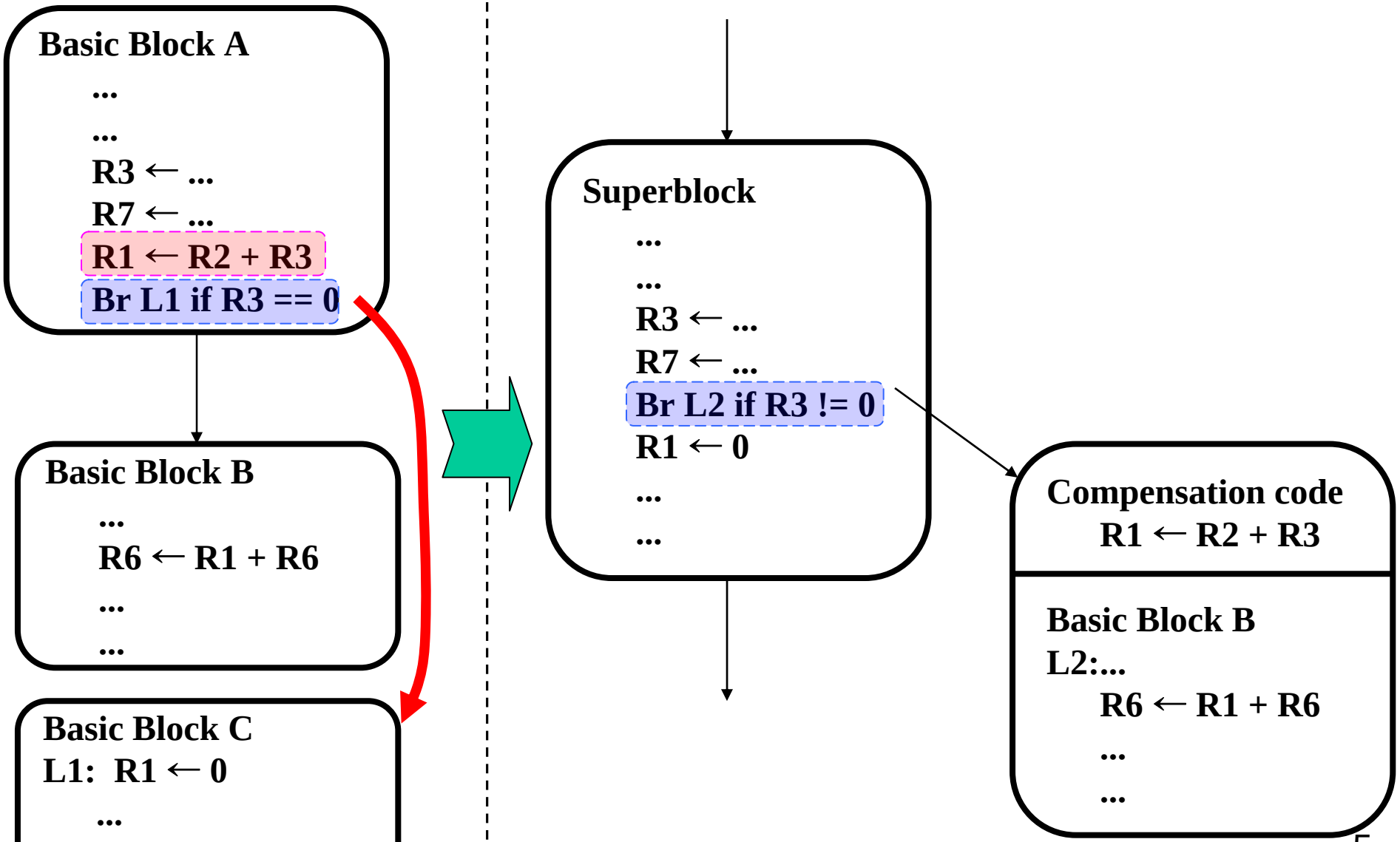
Common Subexpression Elimination (CSE)

```
addi    r16, r4, 4          ; add 4 to %eax
lwzx    r17, r2, r16        ; load operand from memory
add     r7, r17, r7         ; perform add of %edx
stwx    r7, r2, r16         ; store %edx value into memory
```

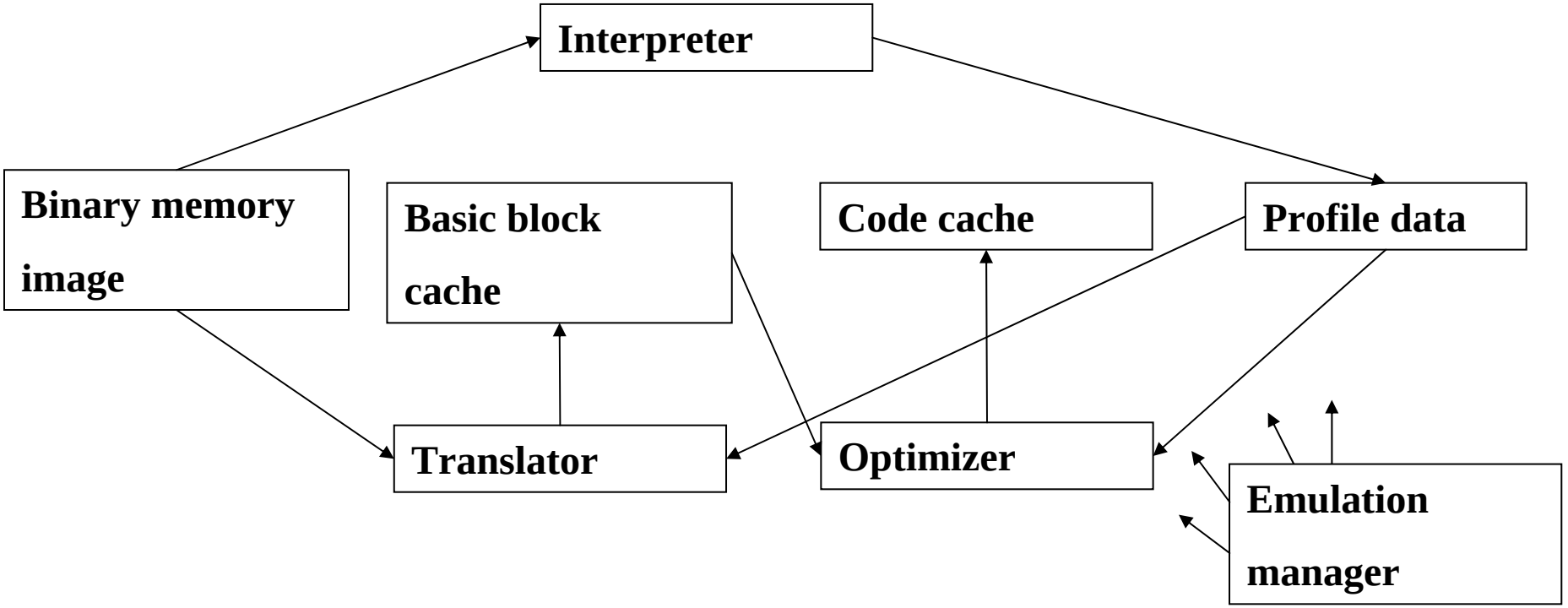
Optimization Based on Profiling



Optimization Based on Profiling



A staged optimization system



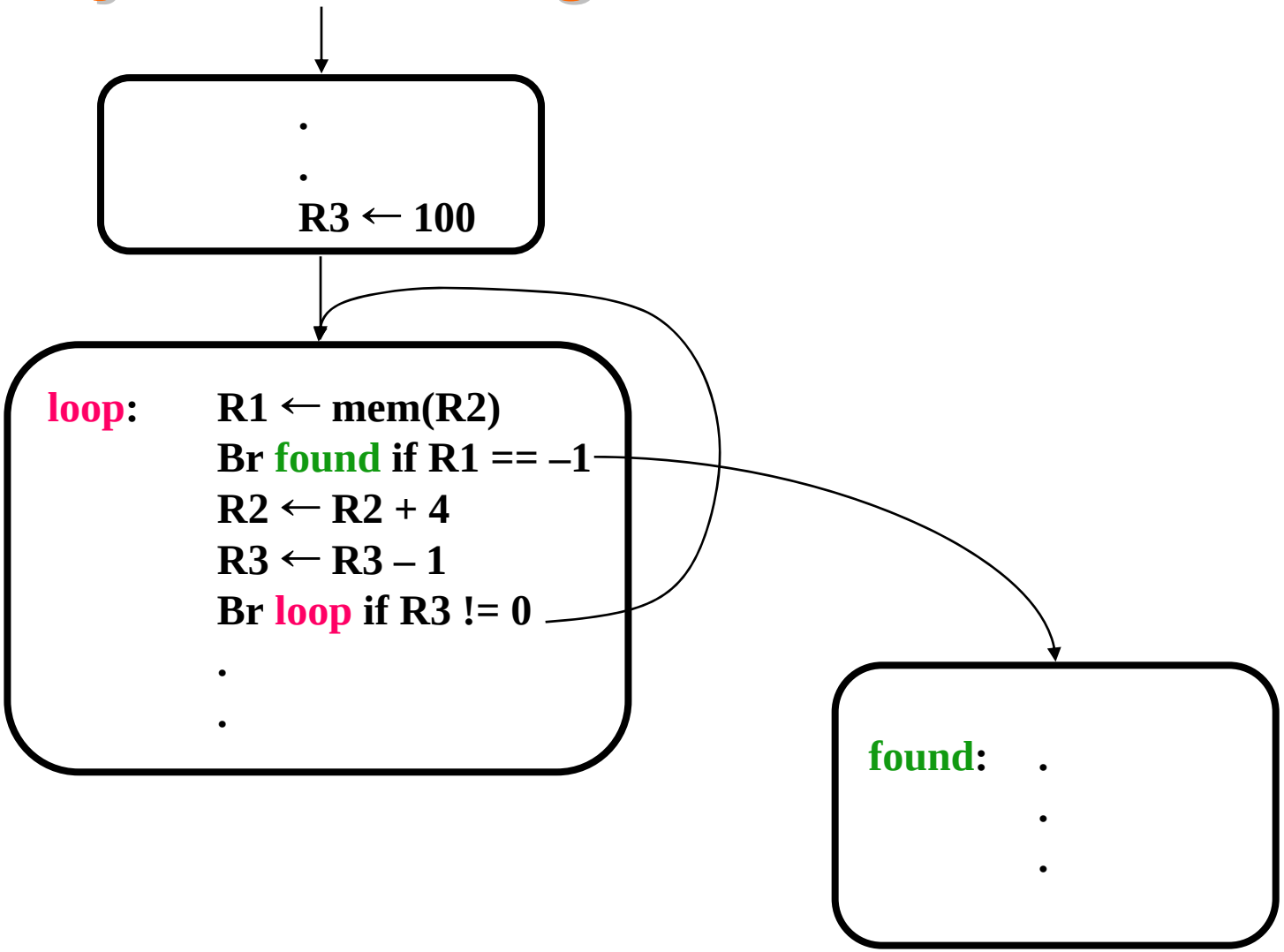
Stages: Interpret → Basic translation → Optimized block → Highly optimized blocks

Fast startup ⟷ Very slow startup

Slow steady state ⟷ Fast steady state

Simple profiling ⟷ Extensive profiling

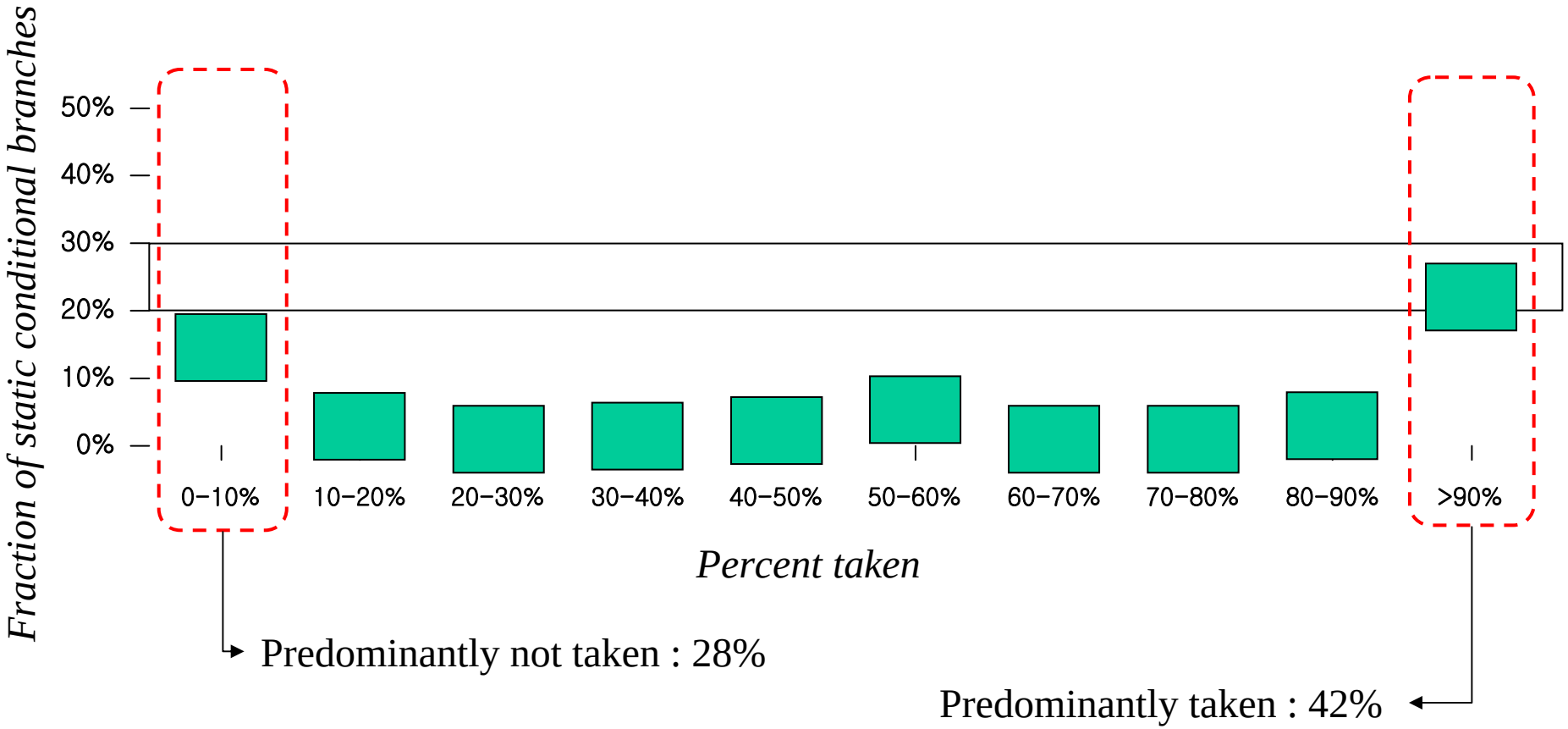
Dynamic Program Behavior



- Dynamic control flow is highly predictable

Dynamic Program Behavior

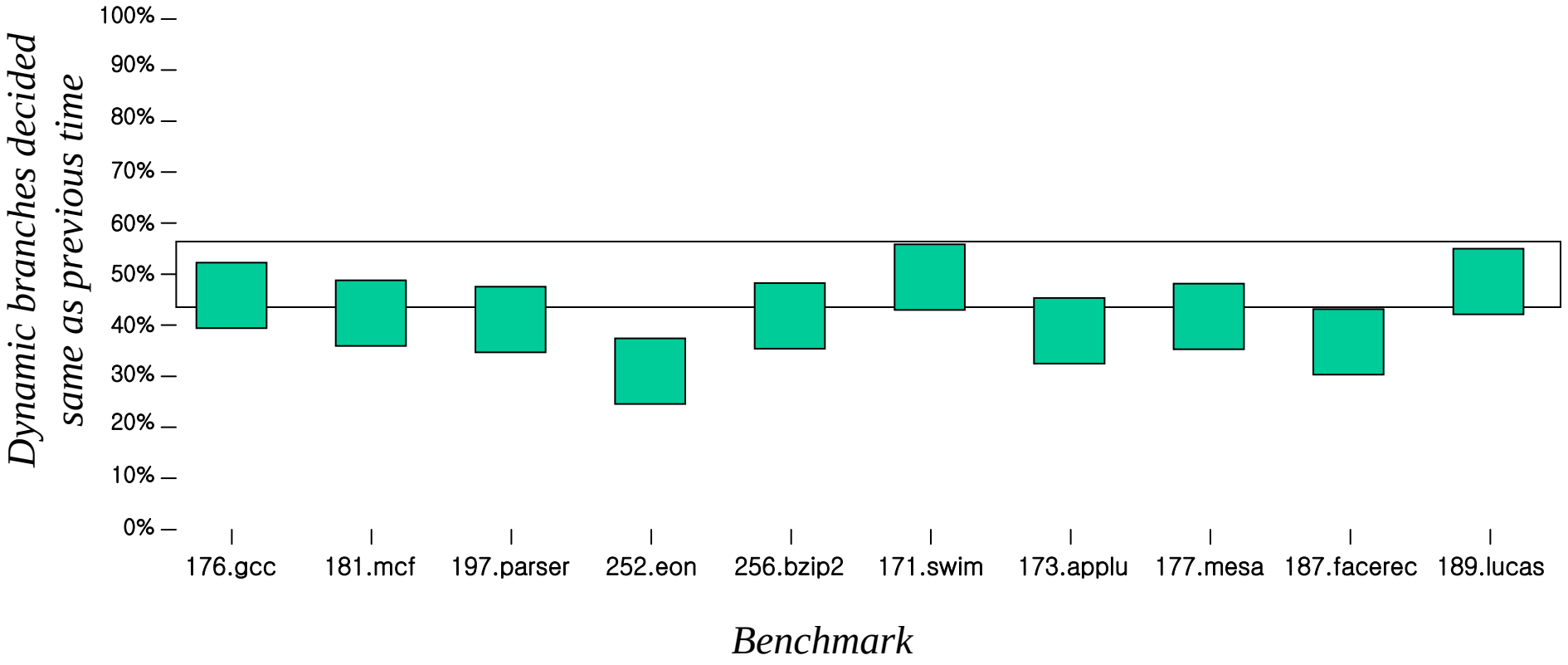
- Distribution of taken conditional branches



Back...

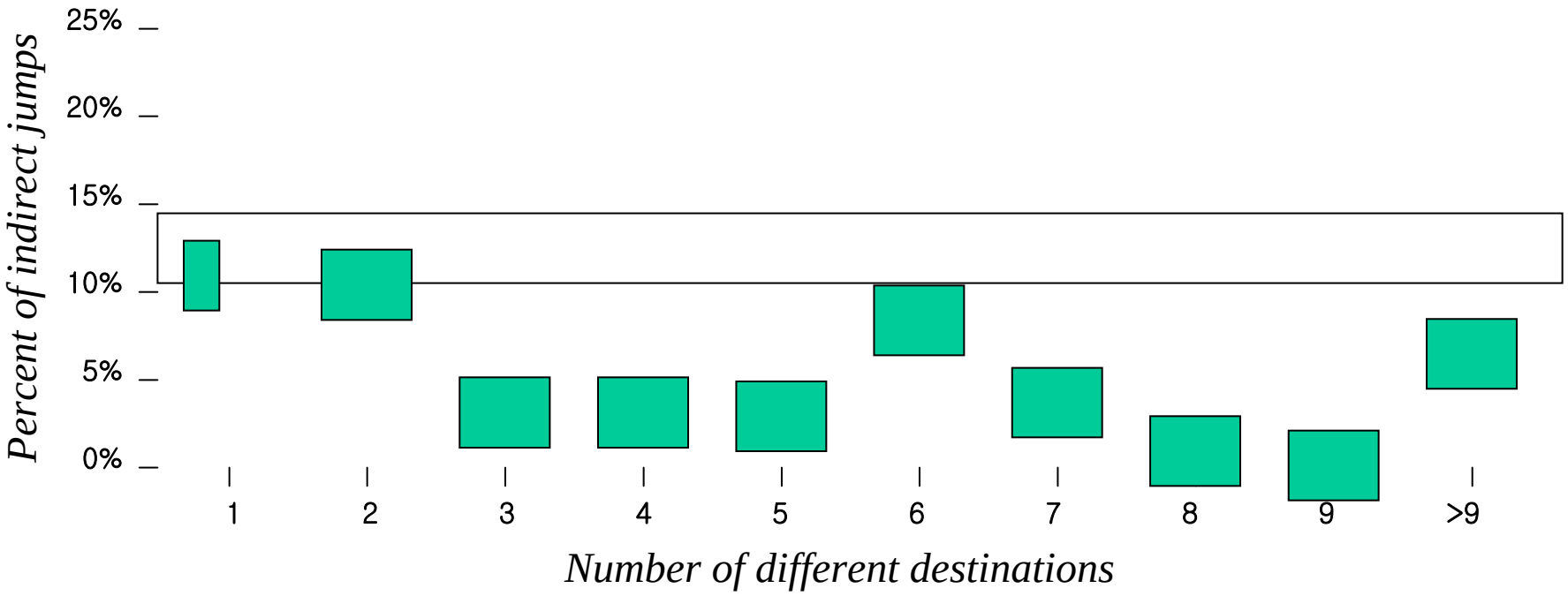
Dynamic Program Behavior

- Consistency of conditional branches
  - The high percentage consists of backward branches



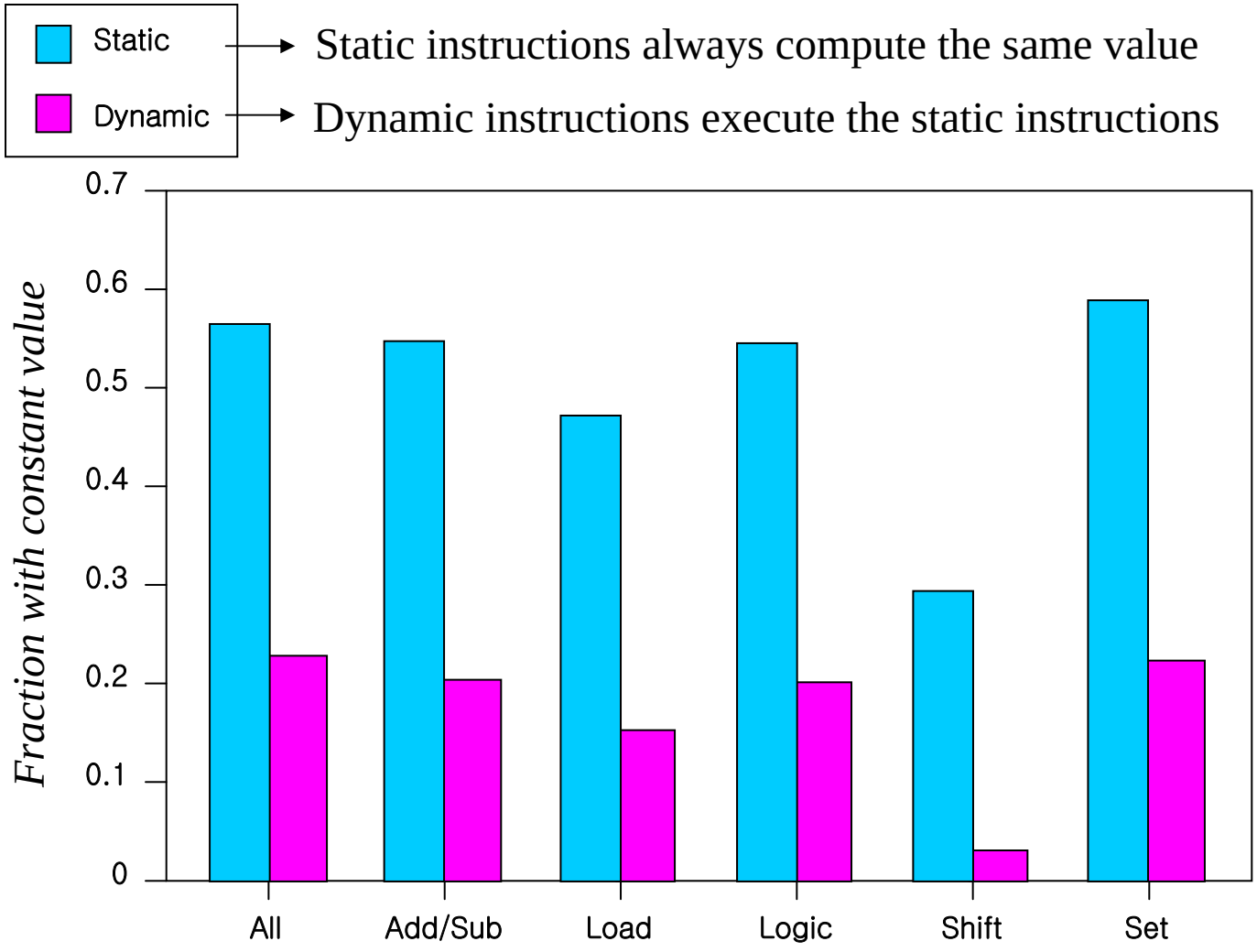
Dynamic Program Behavior

- The predictability of indirect jumps
  - Some jump destination addresses seldom change



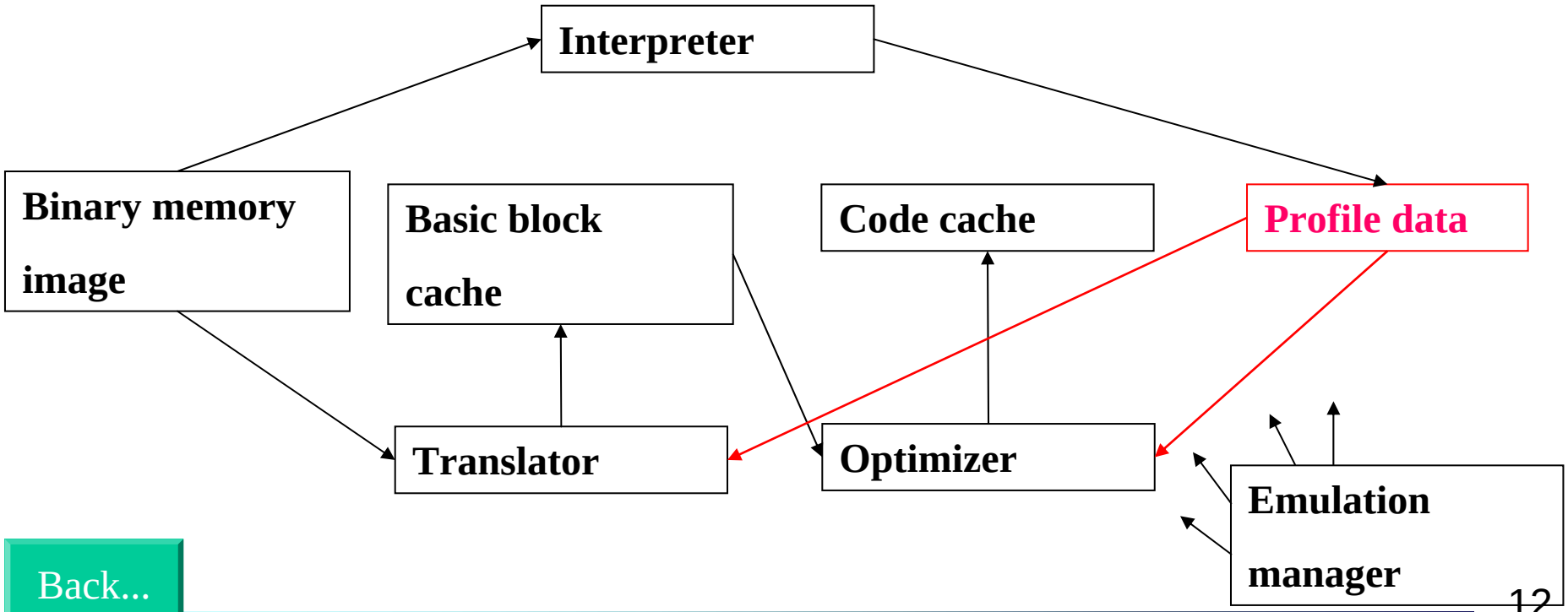
Dynamic Program Behavior

- The predictability of data value



Profiling

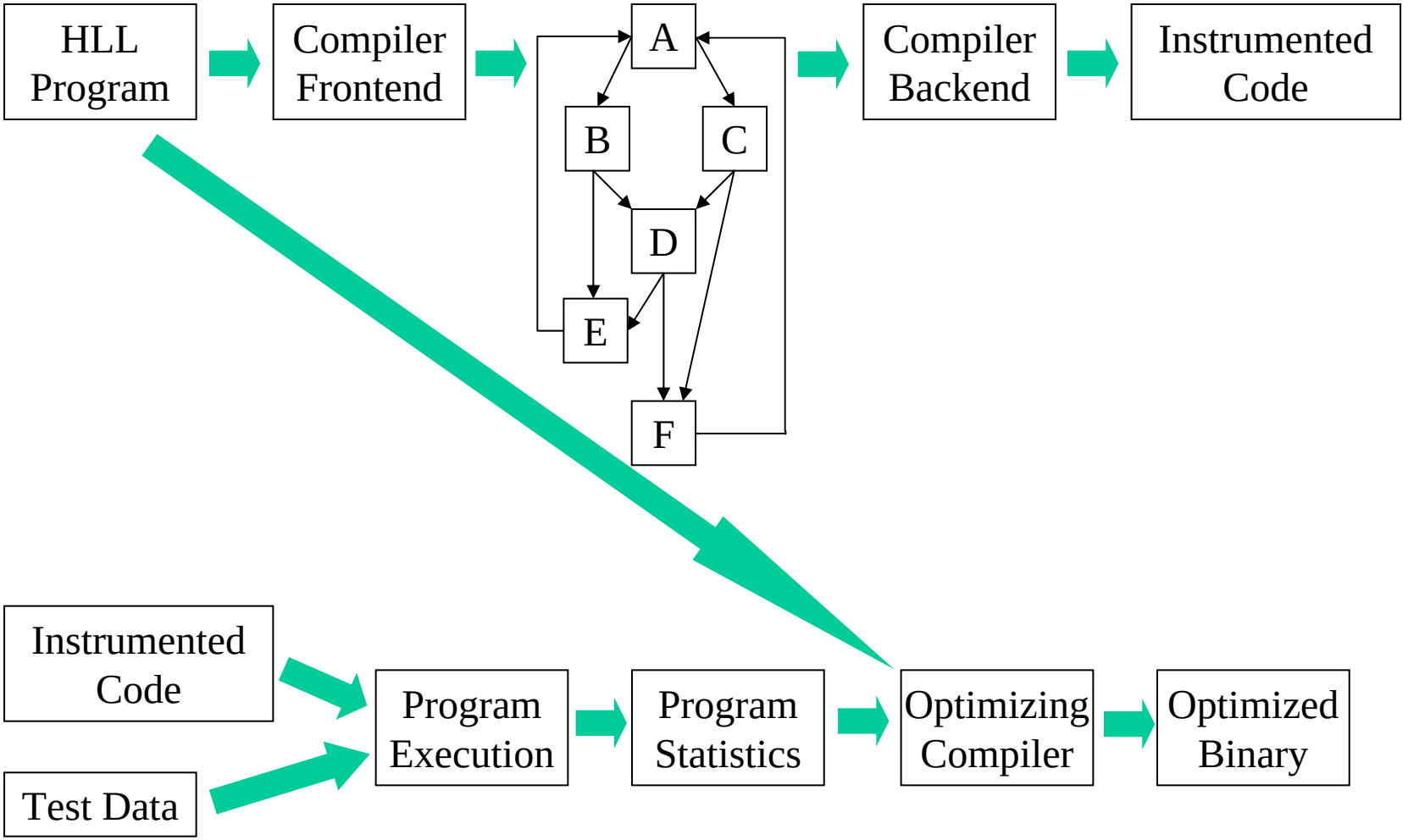
- The process of collecting instruction and data statistics for an executing program
- Optimization based on profiling work



Back...

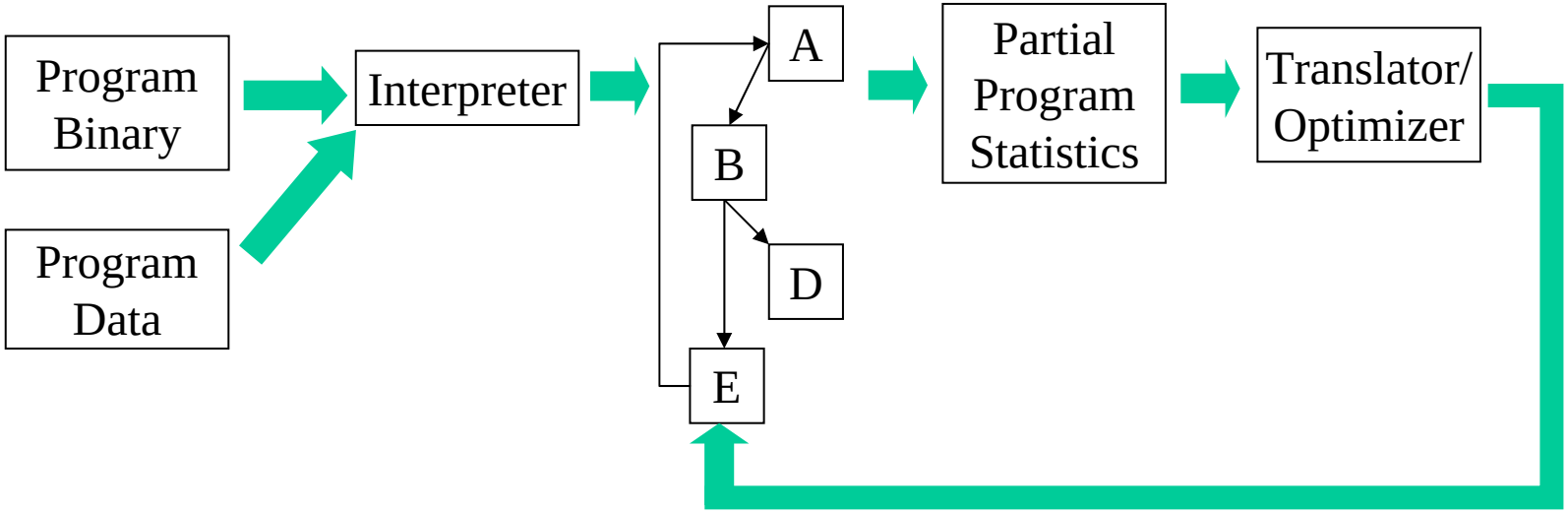
The Role of Profiling

- Traditional profiling



The Role of Profiling

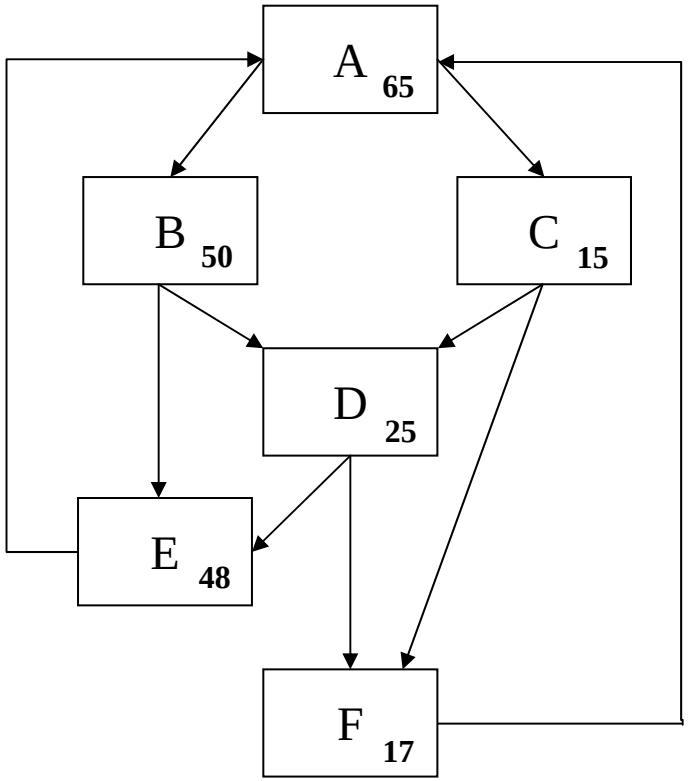
- On-the-fly profiling in a dynamic optimizing VM



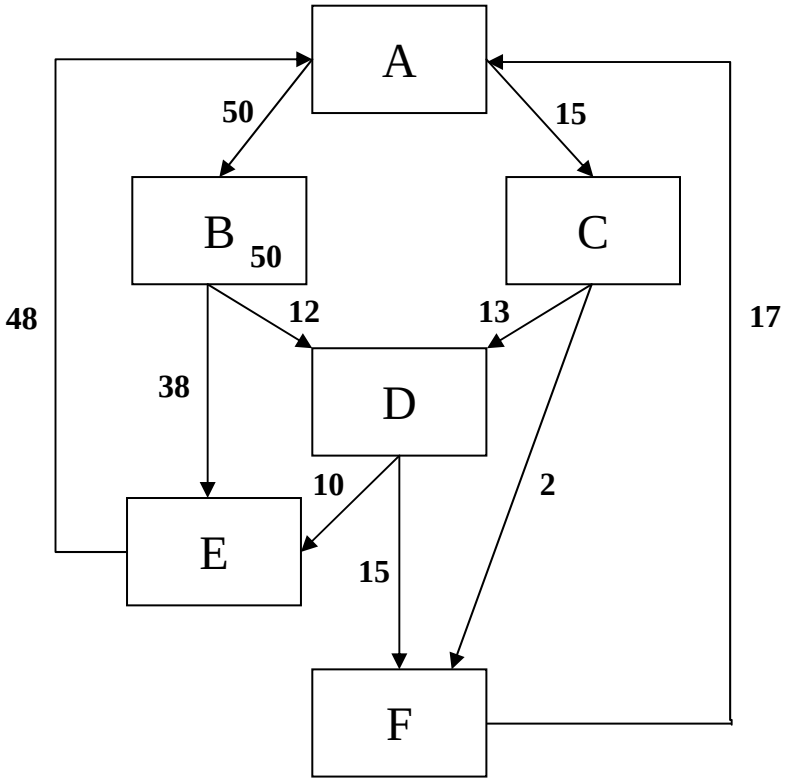
Types of Profiles

- Several types of profile data
  - How frequently code regions are being executed?
    - Can be used to decide the level of optimization
    - Often represented by basic block profile
  - Control flow statistics
    - May be used for rearranging basic blocks into superblocks, for example
    - Often represented by edge profile
  - Edge profile gives more information
  - Edge profile is more expensive, but there is an efficient algorithm [Ball & Larus 1994]
  - There is an efficient algorithm for a path profile

Types of Profiles



A basic block profile



A edge profile

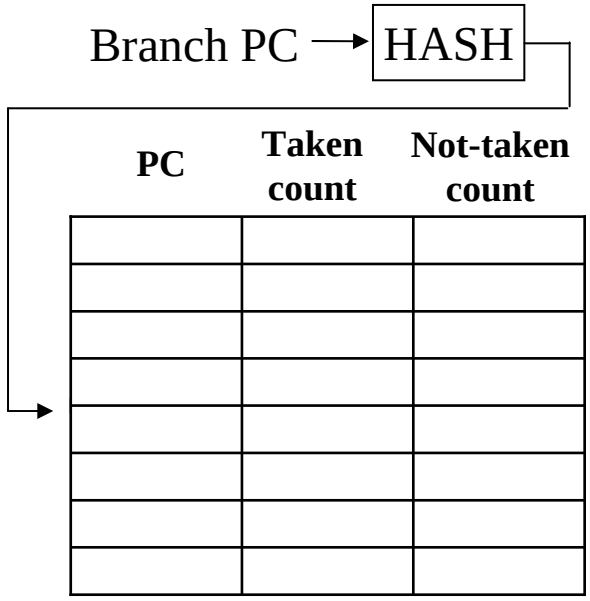
Collecting Profiles

- Instrumentation-based profiling
  - Specific program-related events and counts all instances of the events being profiled
  - Software-based Vs hardware-based
    - Speed, Support, Flexibility?
- Sampling-based profiling
  - Program runs in its unmodified form, the program is interrupted and event is captured
- Instrumentation compared to sampling
  - Can collect profile data in a shorter time
  - Slow down program more
    - Sampling in fact is also costly due to trap

Profiling During Interpretation

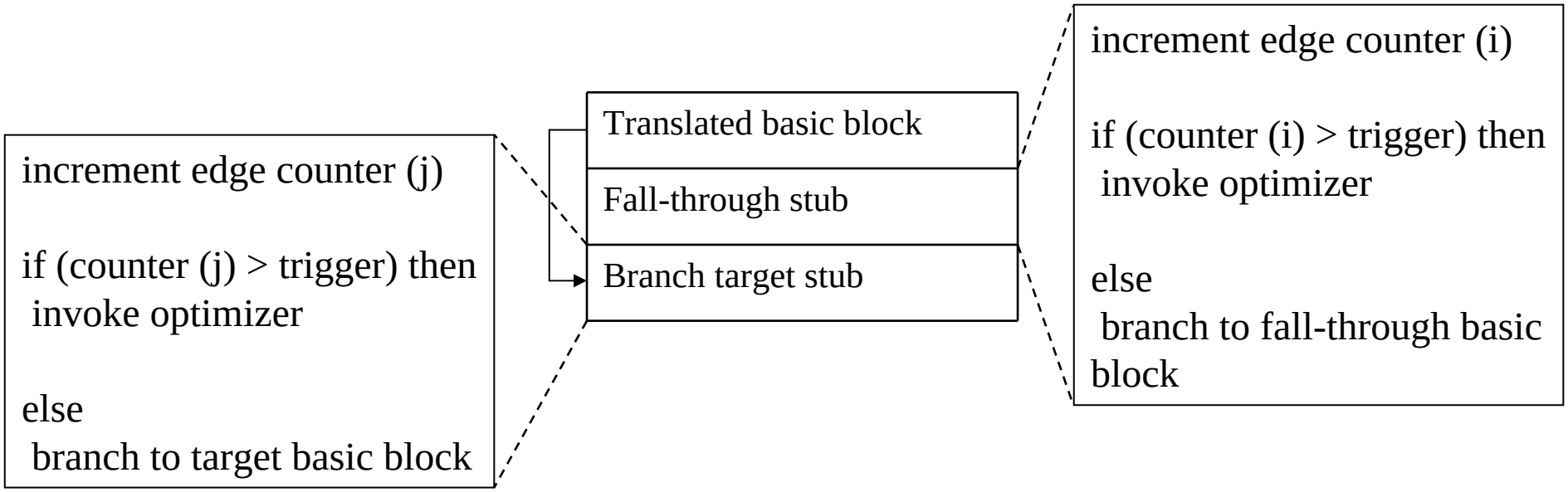
```
Instruction function list
.
.
branch_conditional(inst) {
        BO     = extract(inst, 25, 5);
        BI     = extract(inst, 20, 5);
        displacement = extract(inst, 15, 14) * 4;
        .
        .
        // code to compute whether branch should be taken
        .
        .
        profile_addr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

PowerPC Branch Conditional Interpreter Routine



Profile Table for Collecting an Edge Profile During Interpretation

Profiling Translated Code



Edge Profiling Code Inserted into Stubs of a Binary Translated Basic Block

Emulation Stages

Profiling Overhead

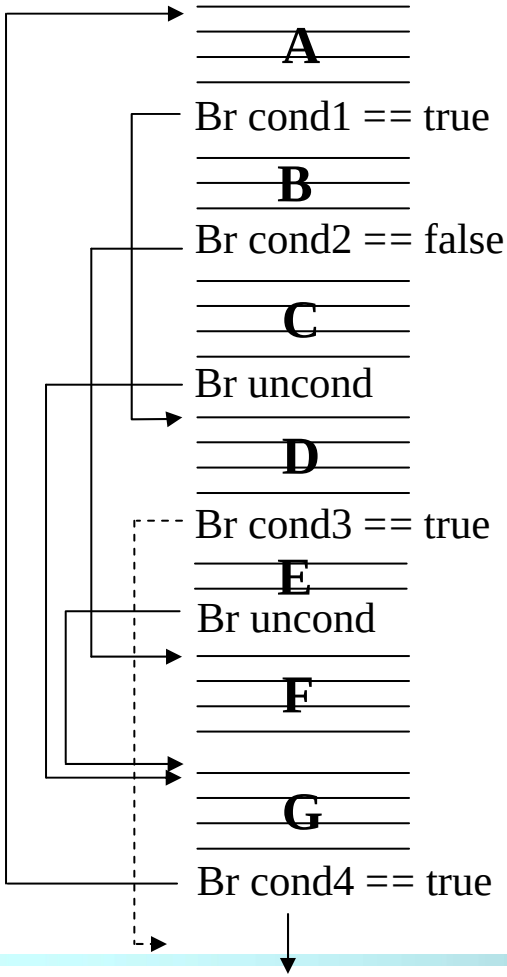
- For profiling during interpretation, occurring 10-20% overhead
- Less for translated code

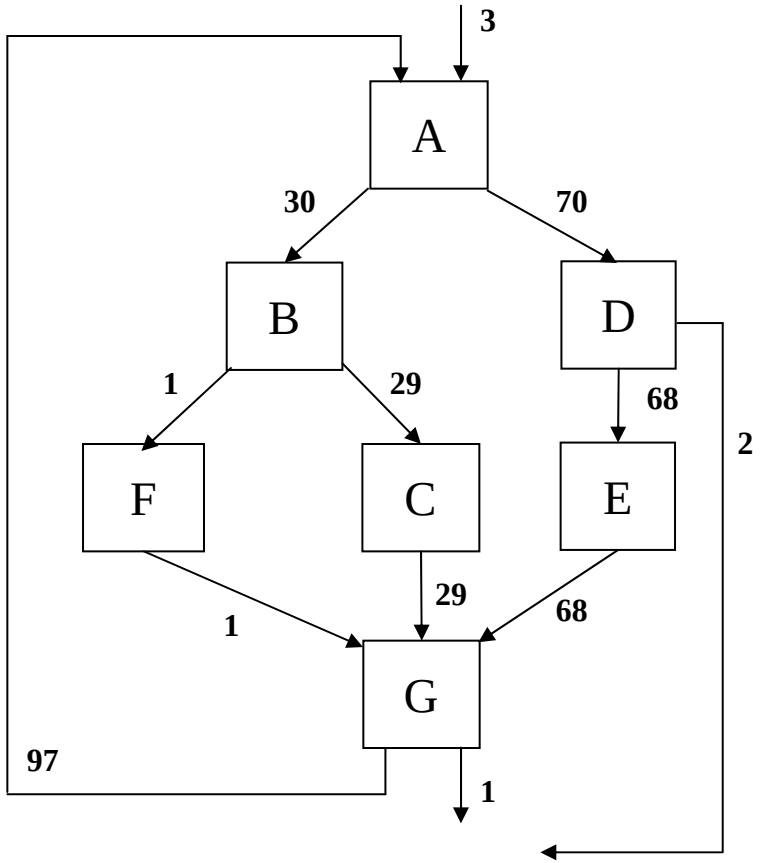
Optimizing Translation Blocks

- Two-part strategy for optimizations
  - Improving spatial locality
    - Rearrange the layout of blocks
    - Inlining
      - ✓ Which also removes call-return code
  - Optimizing enlarged translated blocks
    - Traces, superblocks, tree groups
    - Unit for code optimization

Improving Locality

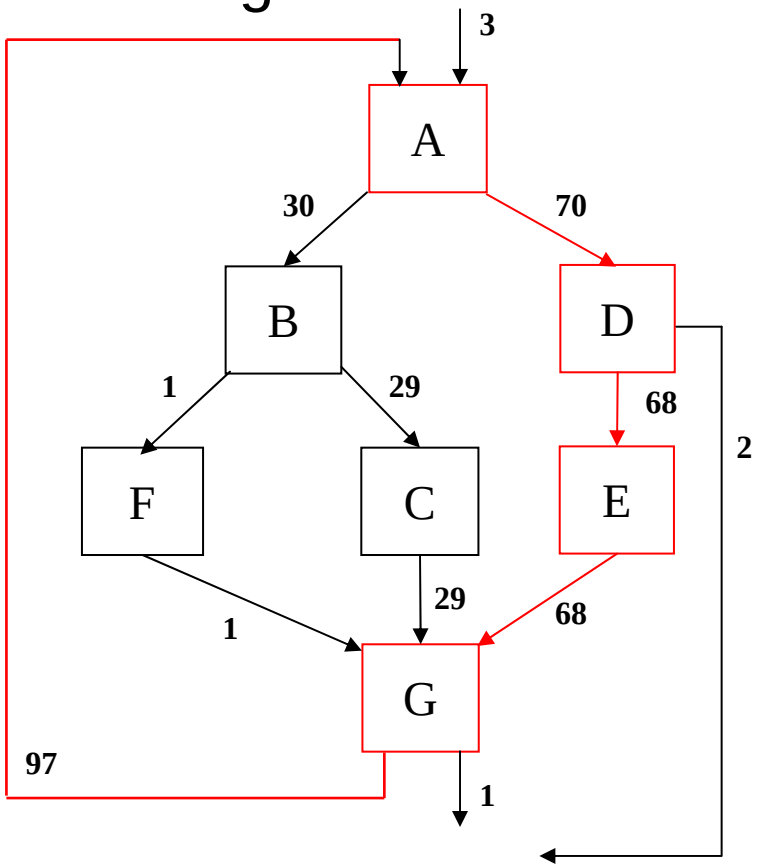
- Example code sequence

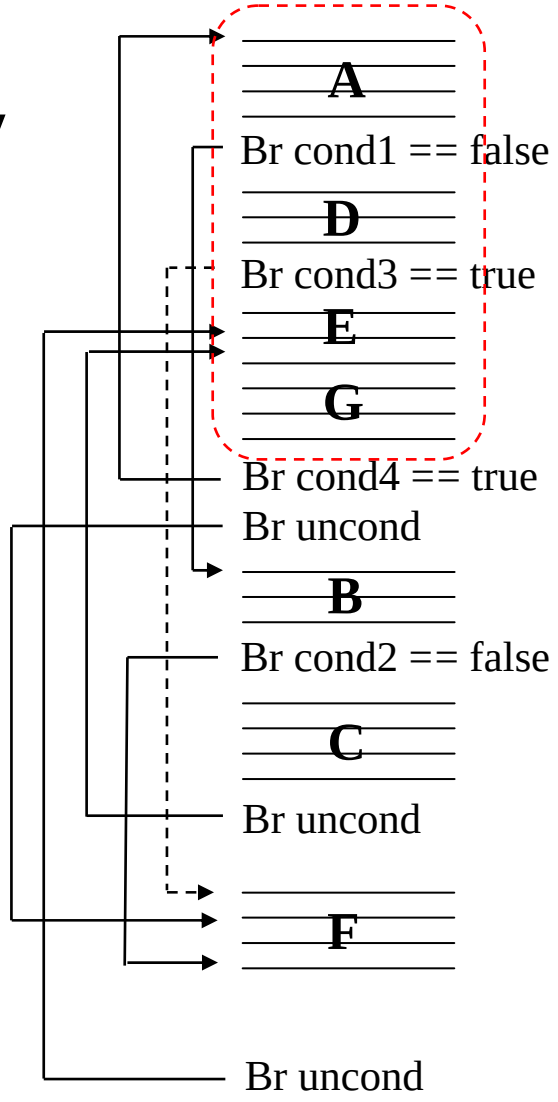




Improving Locality

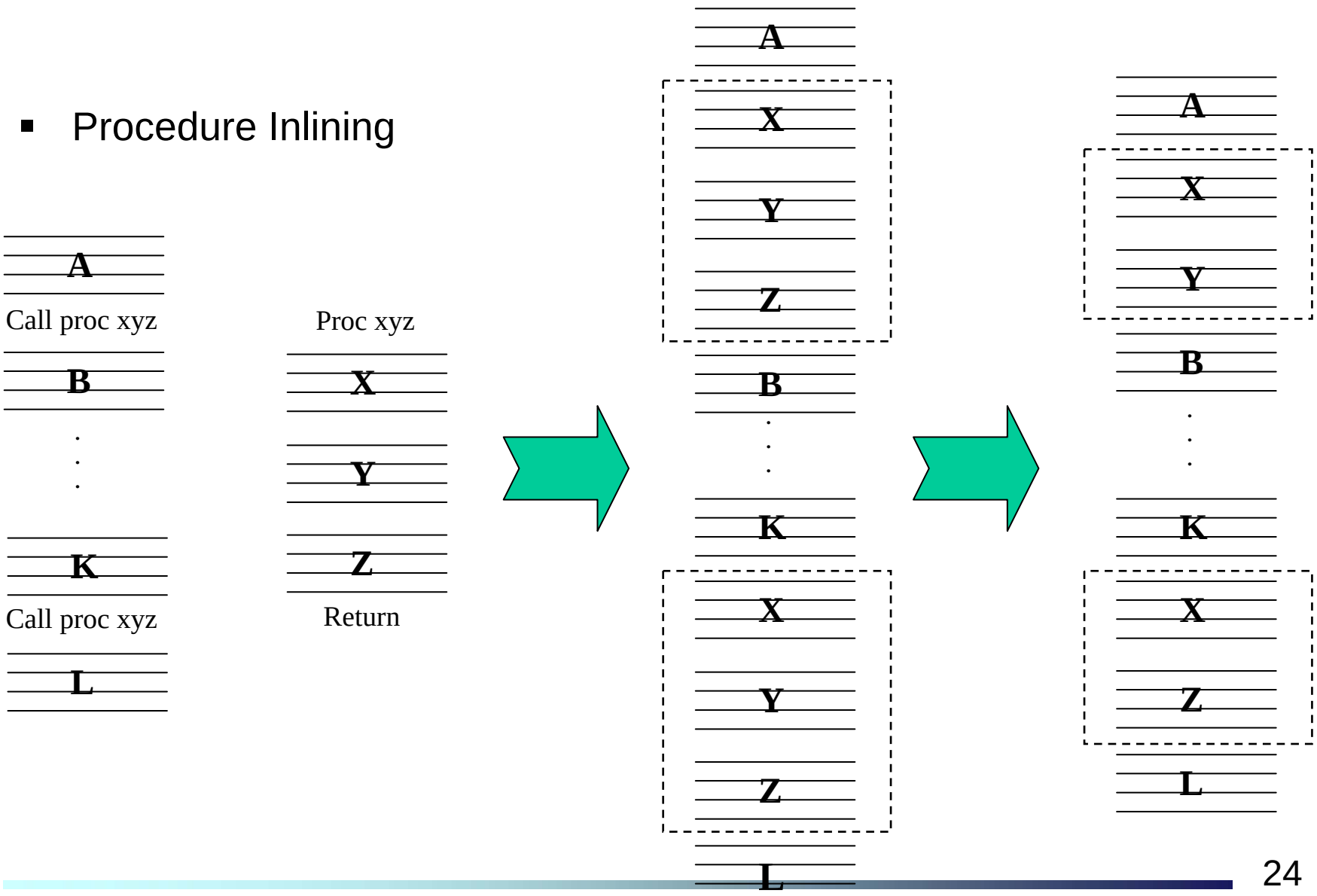
- Rearrange the blocks in memory





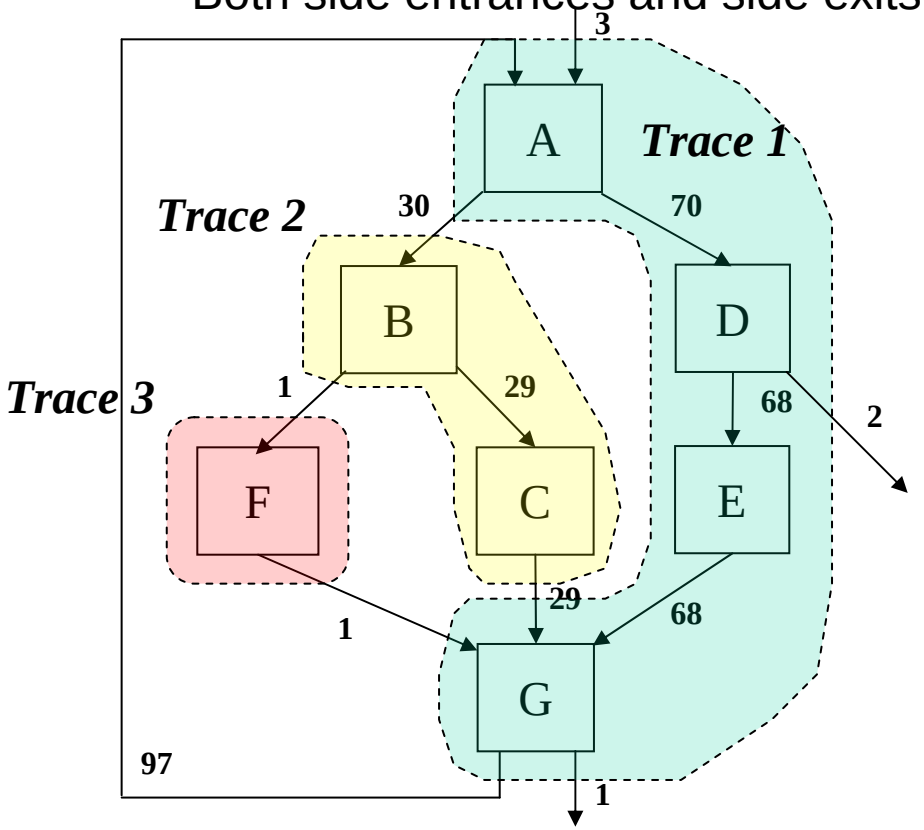
Improving Locality

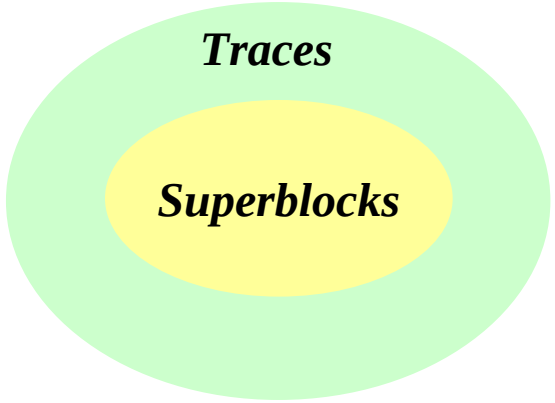
- Procedure Inlining



Traces

- Trace
  - A contiguous sequence
  - Both side entrances and side exits

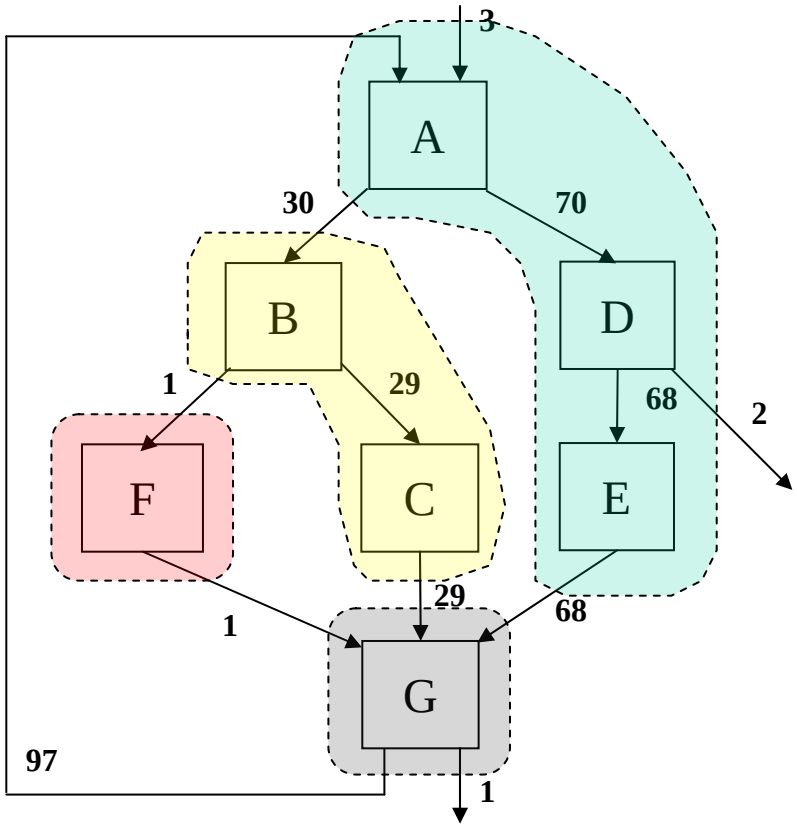


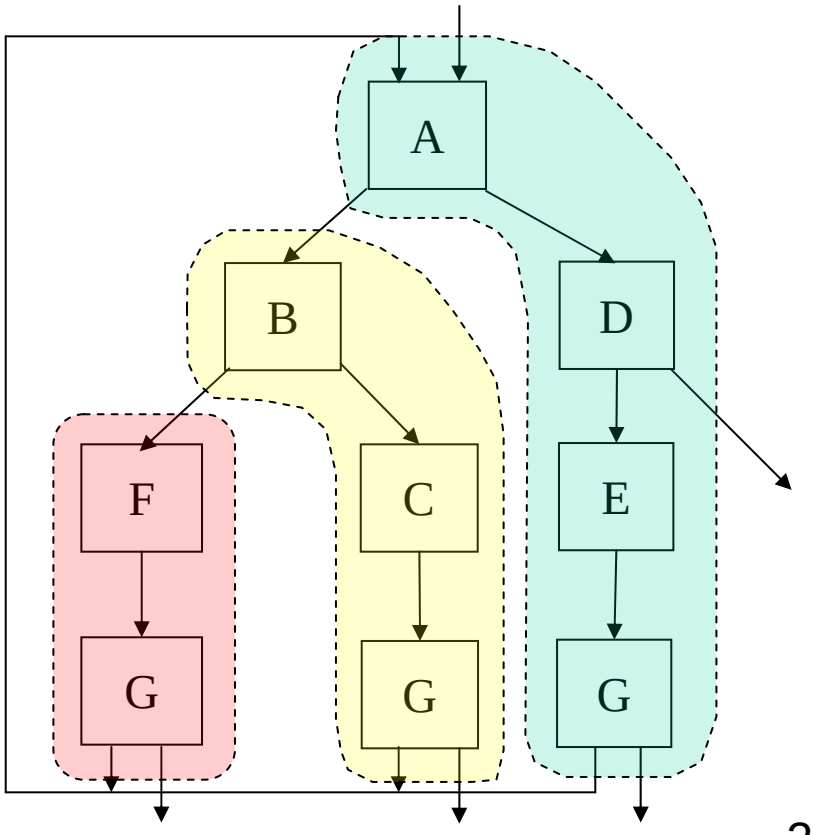


Relations between Superblocks and Traces

Superblocks

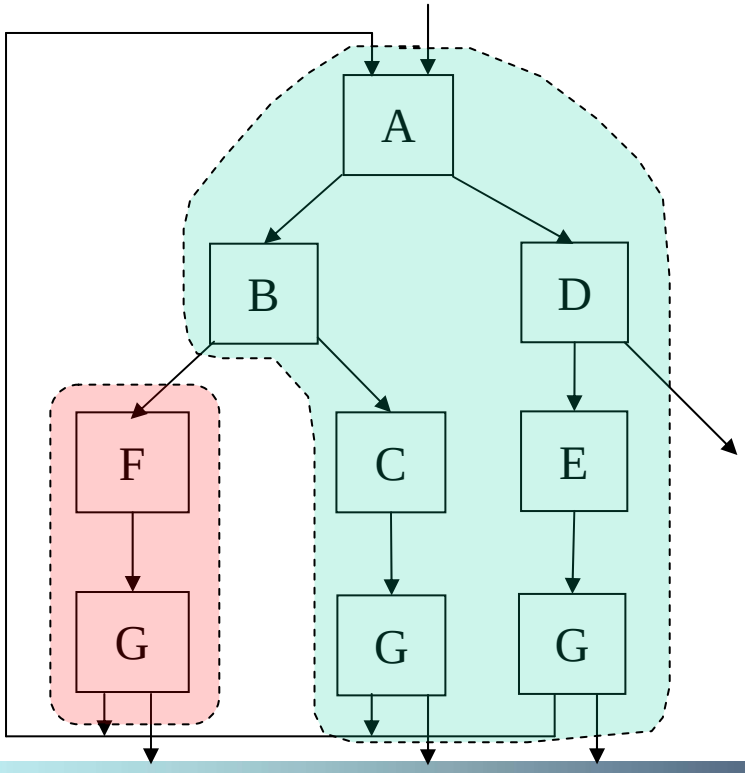
- Superblocks
  - Regions of code with only one entry and one or more exit points, often formed by tail duplication





Tree Groups

- Tree groups
  - Regions of code with only one entry and one or more exit points
  - Useful when branches arbitrarily

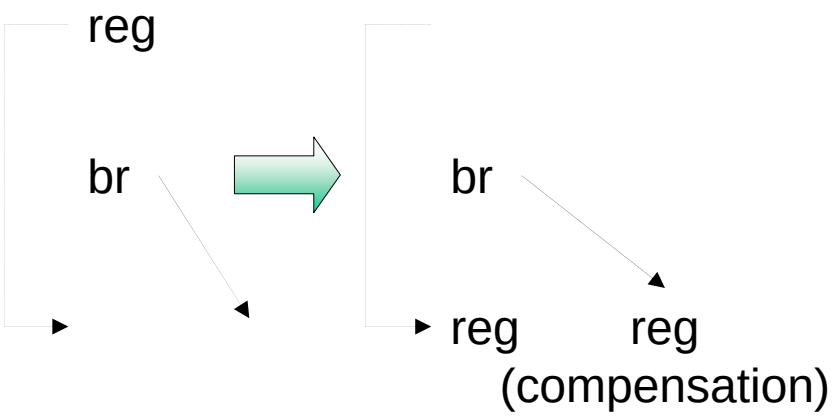


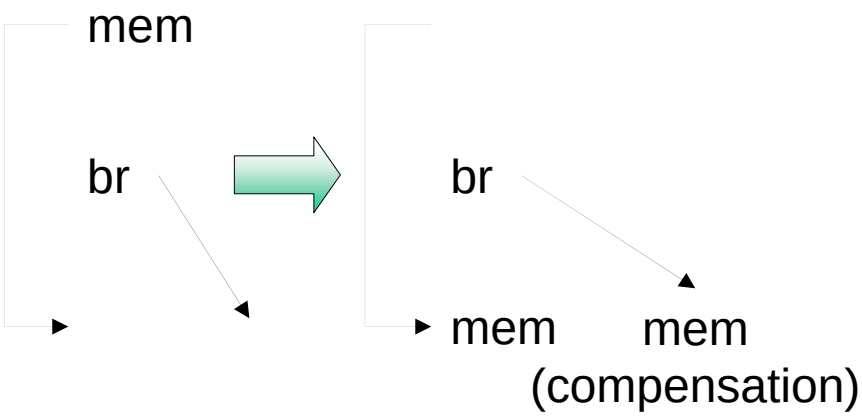
Code Reordering and Trap Handling

- For higher performance dynamic optimizers reorder instructions a lot
  - To exploit instruction-level parallelism
  - To reduce stalls
  - To optimize the code
- Reordering examples

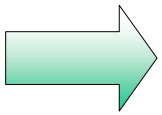
- Reordering raises an issue of trap compatibility and state compatibility

Move instruction around branch





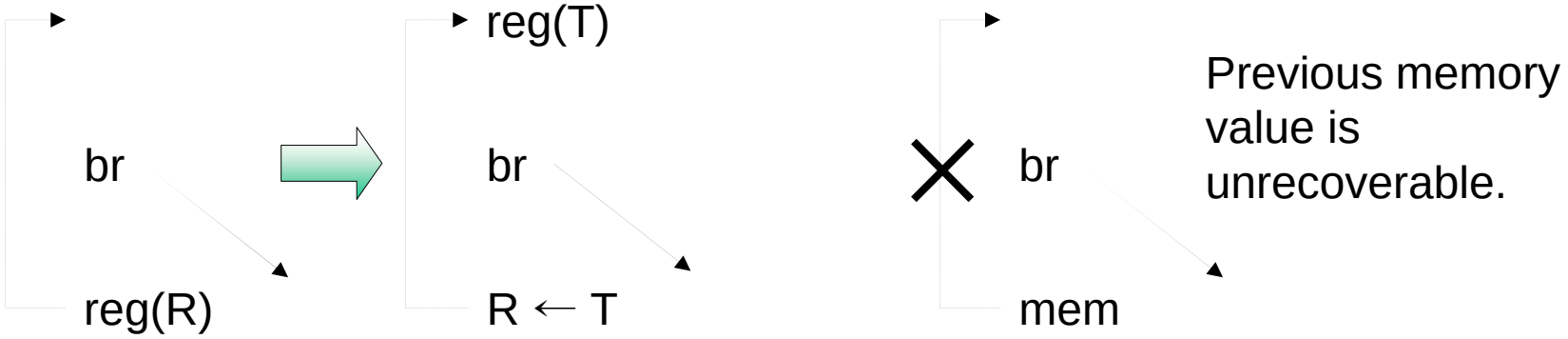

```
R1 ← mem(R6)
R2 ← mem(R6+4)
R3 ← R1 + 1
R4 ← R1 << 2
br exit if R7 == 0
R7 ← R7 + 1
mem(R6) ← R3
```



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

```
R4 ← R1 << 2
```

Speculative code motion above branch

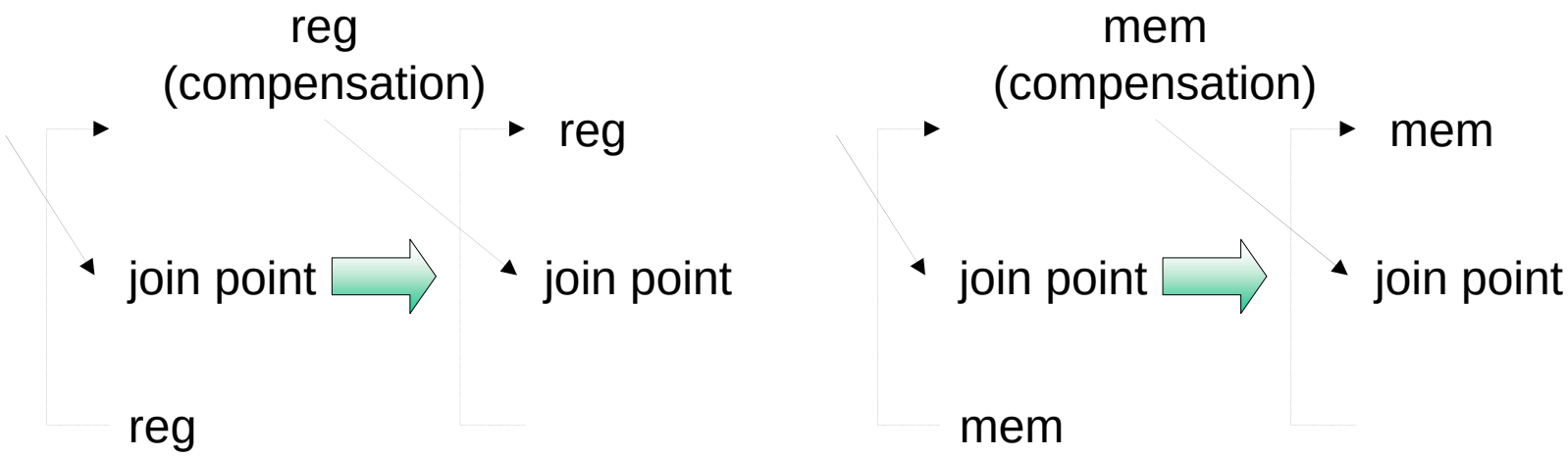


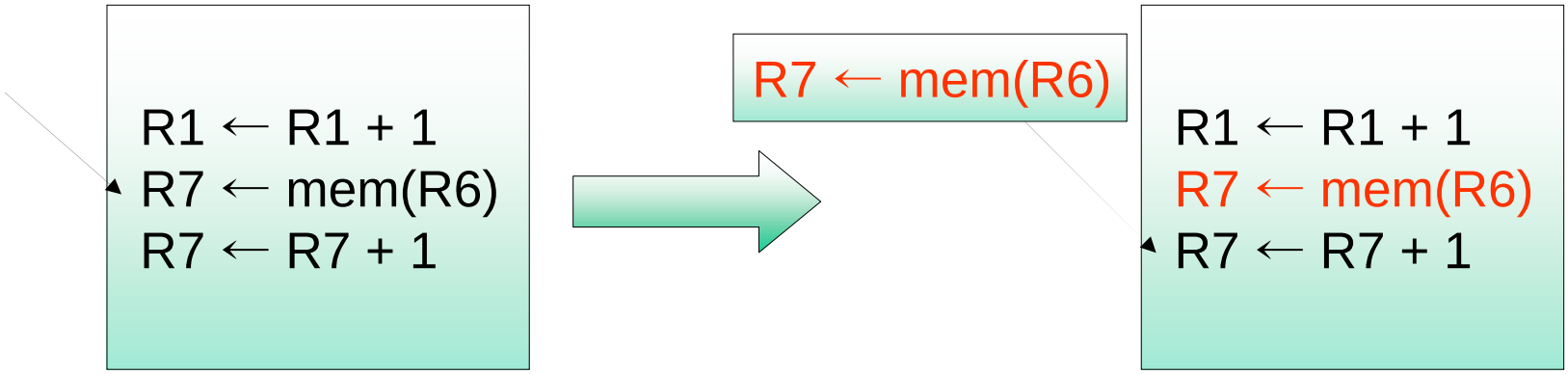

```
R2 ← R1 << 2
br exit if R8 == 0
R6 ← R7 * R2
mem(R6) ← R3
R6 ← R2 + 2
```

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

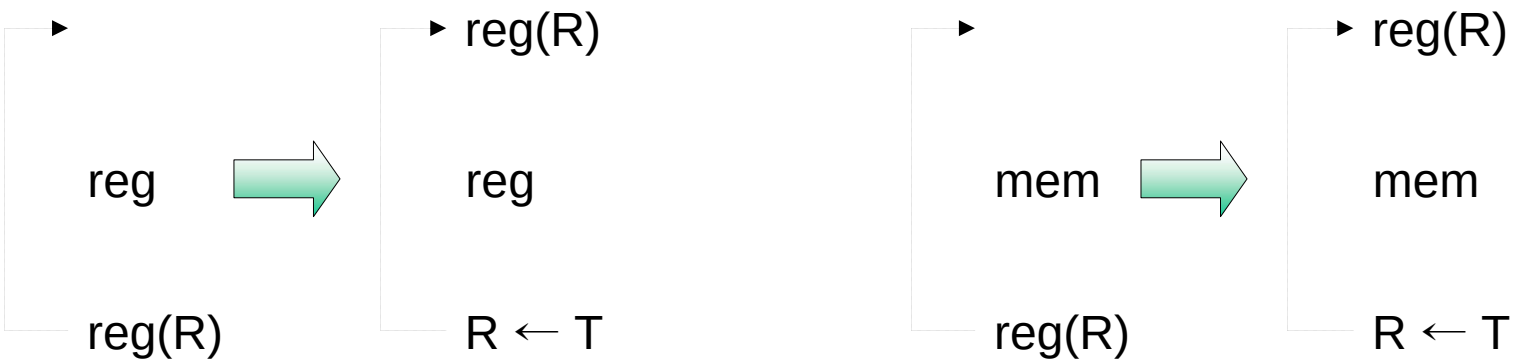
```
R2 ← R1 << 2
T1 ← R7 * R2
br exit if R8 == 0
mem(T1) ← R3
R6 ← R2 + 2
```

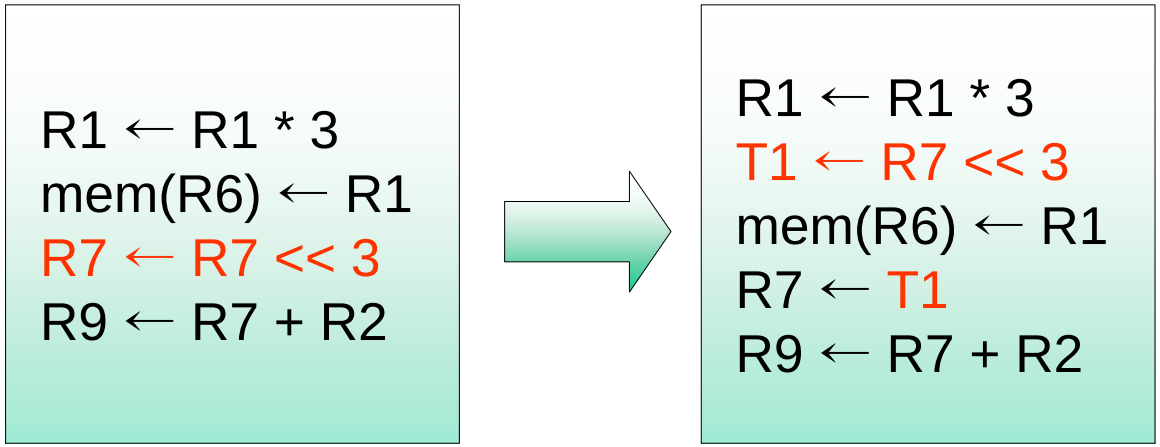
Move instructions above join point





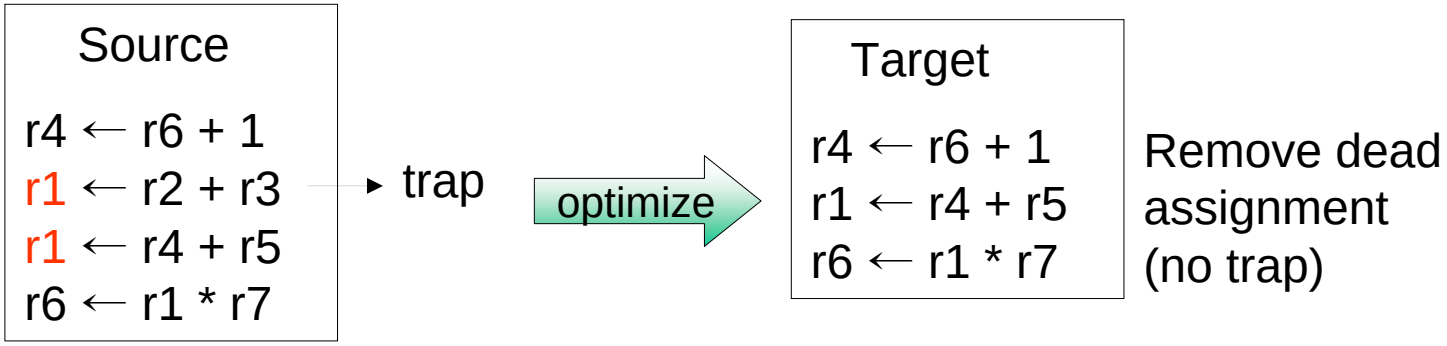
Move instructions in straight-line code





Compatibility Issues

- Trap compatibility
  - Any trap in native execution should be observed during emulation and any trap observed during emulation should also occur in native execution



Compatibility Issues

- Memory and state compatibility
  - At trap point, if runtime can reconstruct the state of memory and registers at the native platform