CSc 553
Principles of Compilation

09. Profiling and Profile-Guided Code Optimizations

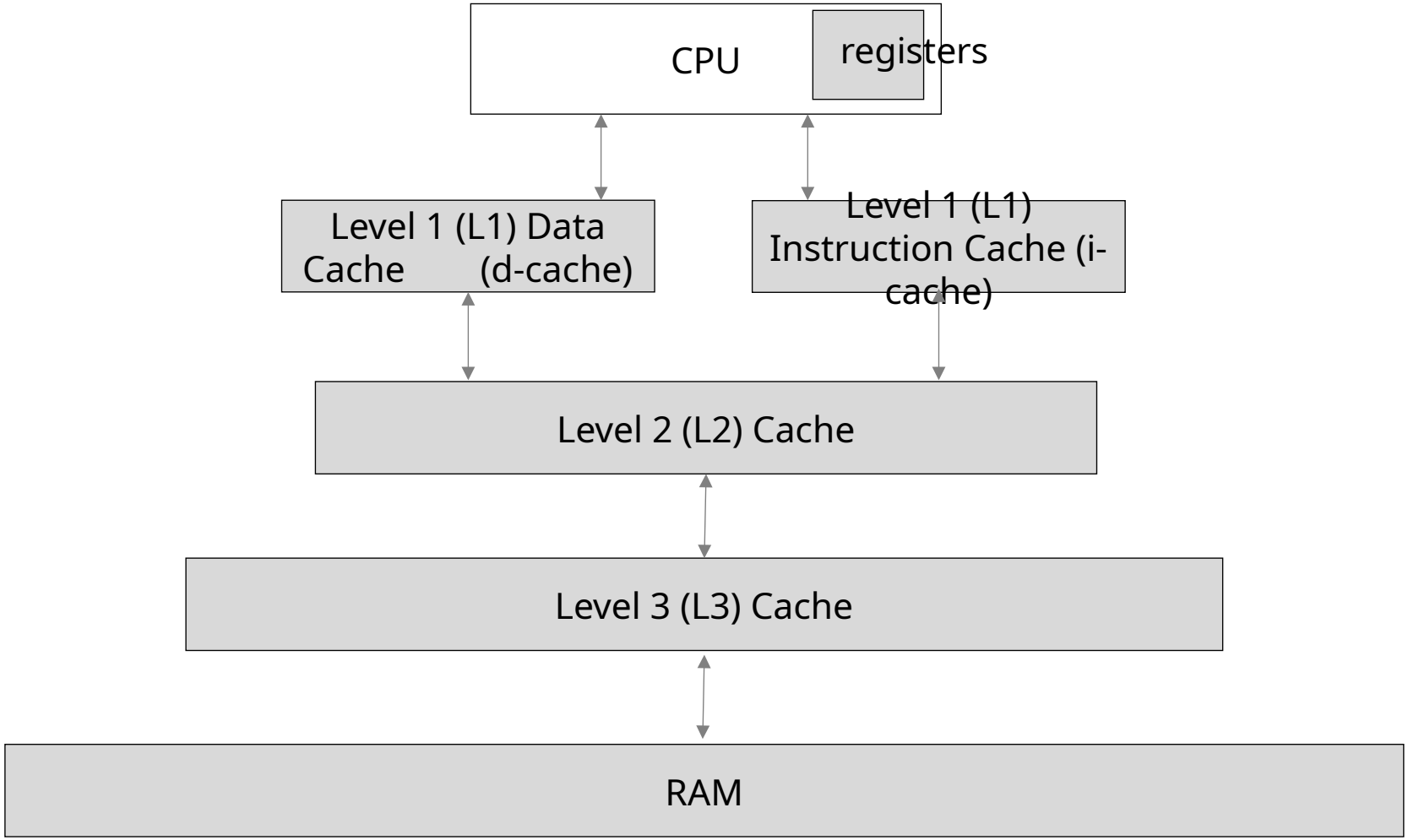
Saumya Debray

The University of Arizona

Tucson, AZ 85721

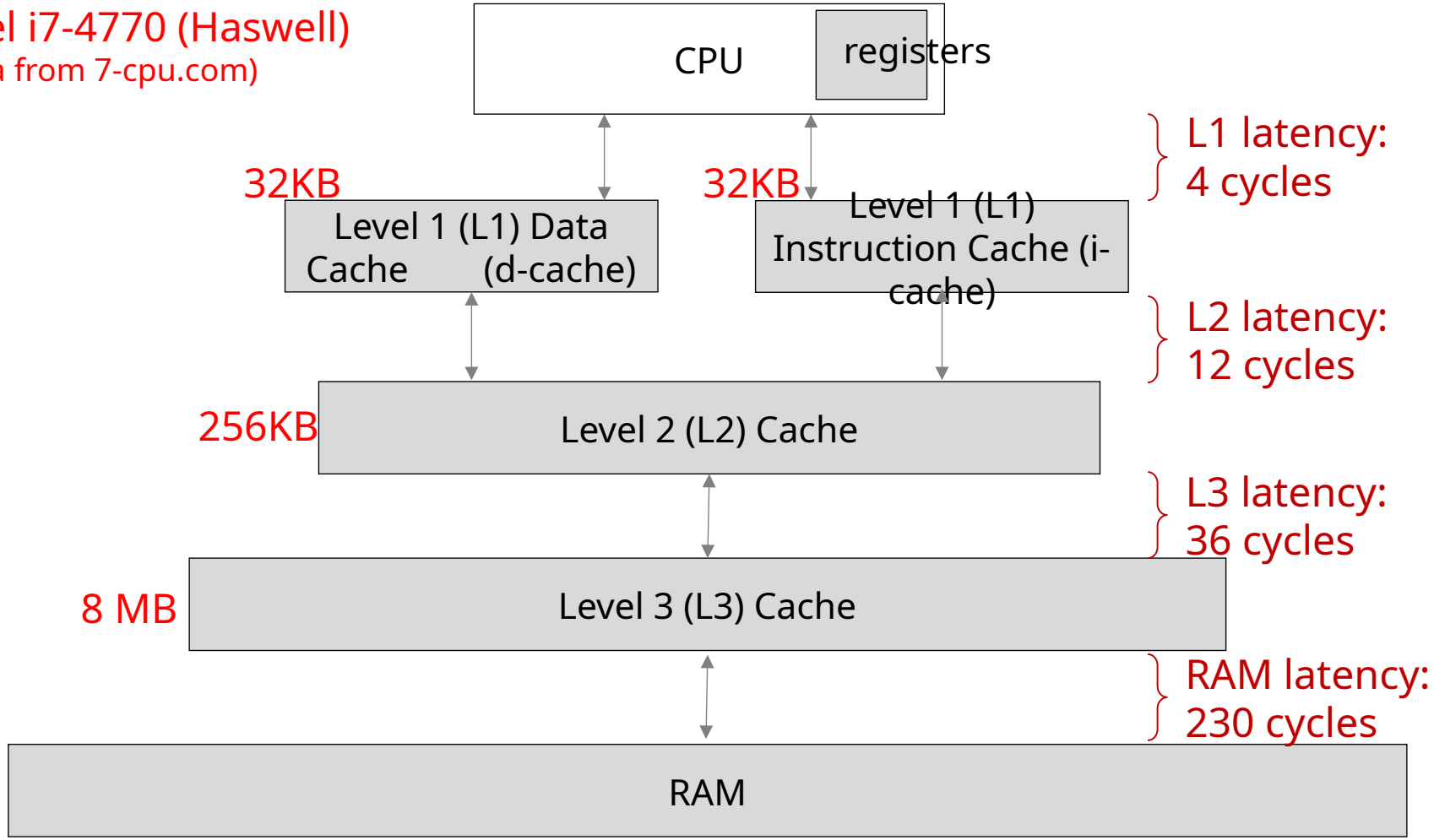
Background: caches

Typical memory hierarchy structure



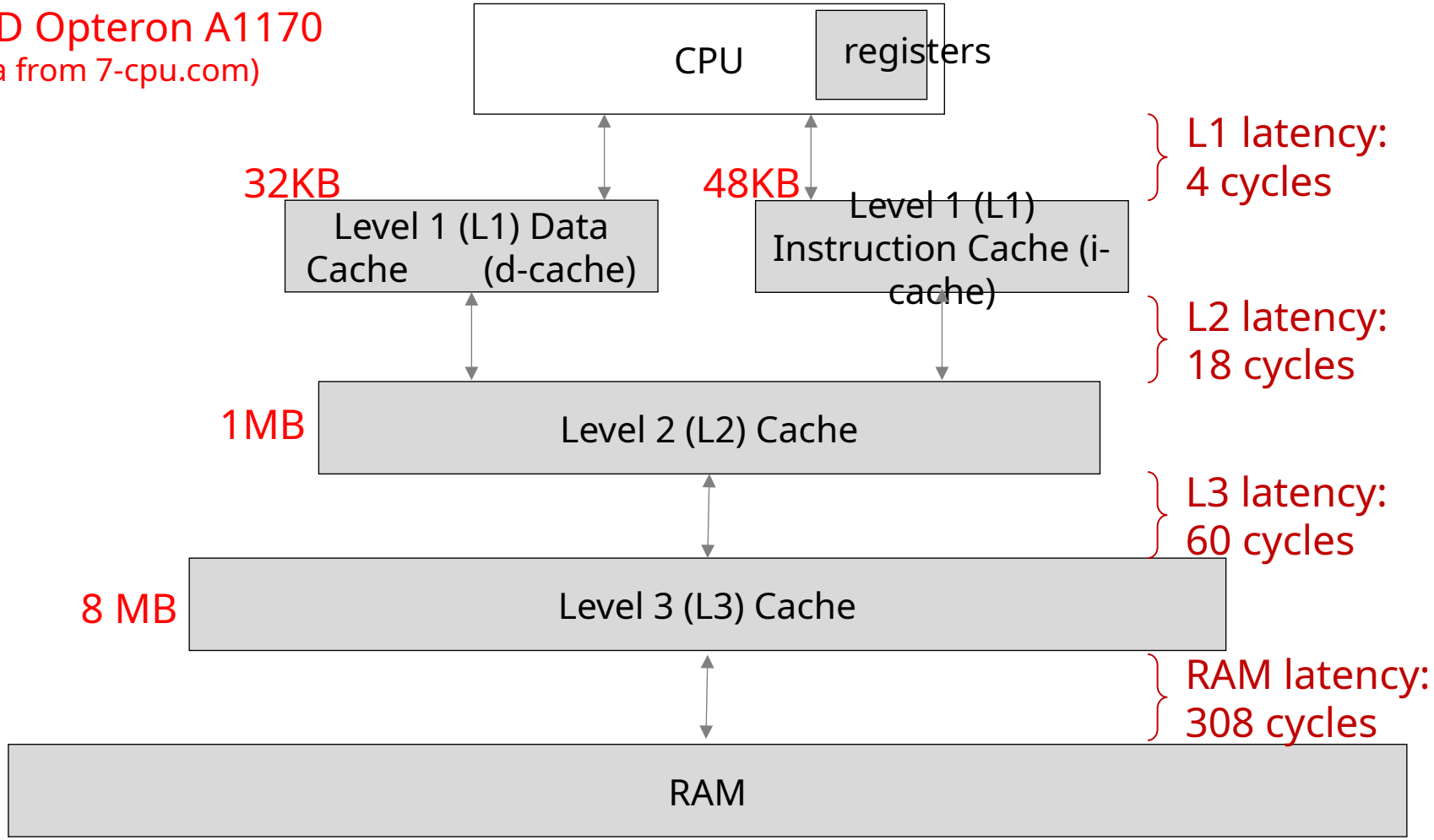
Typical memory hierarchy structure

Intel i7-4770 (Haswell)
(data from 7-cpu.com)



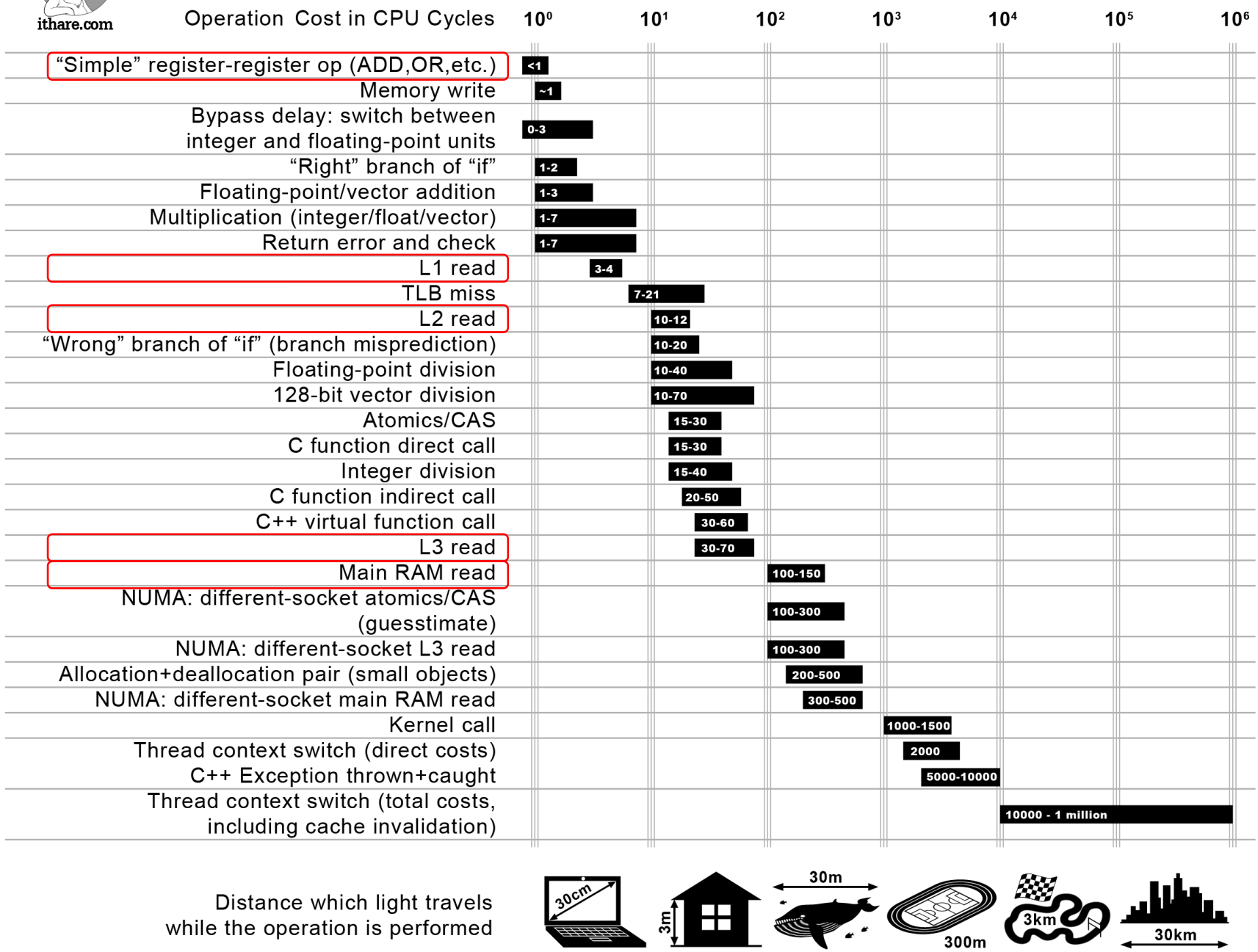
Typical memory hierarchy structure

AMD Opteron A1170
(data from 7-cpu.com)



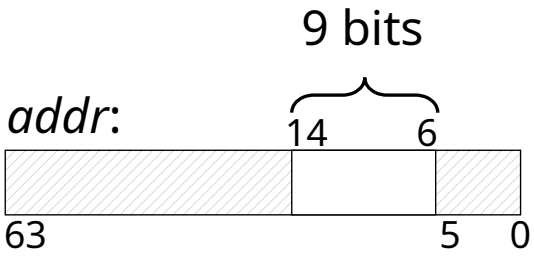
Not all CPU operations are created equal

ithare.com



L1 instruction cache

- Each cache access is to a "cache line" large enough to contain several instructions
- The cache line that a memory address *addr* maps to is determined by selected bits of *addr*
  - conceptually similar to hashing



- E.g., on Intel i7-4770 Haswell:
  - cache line size = 64 bytes
  - 32 KB cache ⇒ 32 KB/64 = 512 cache lines
  - use 9 bits from the address to determine cache line

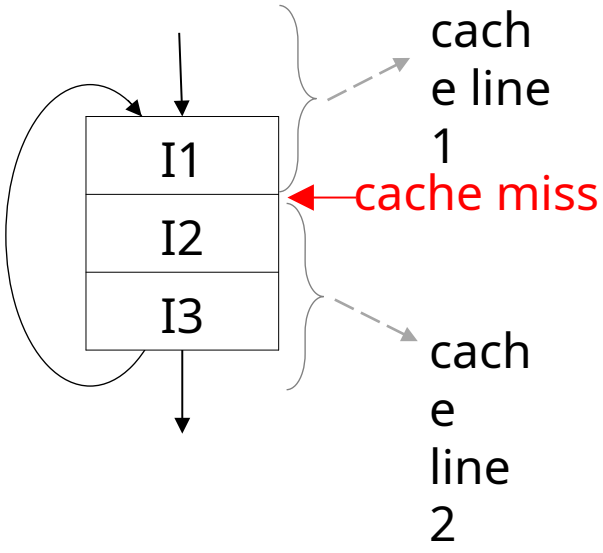
L1 instruction cache

When fetching an instruction at address *addr*:

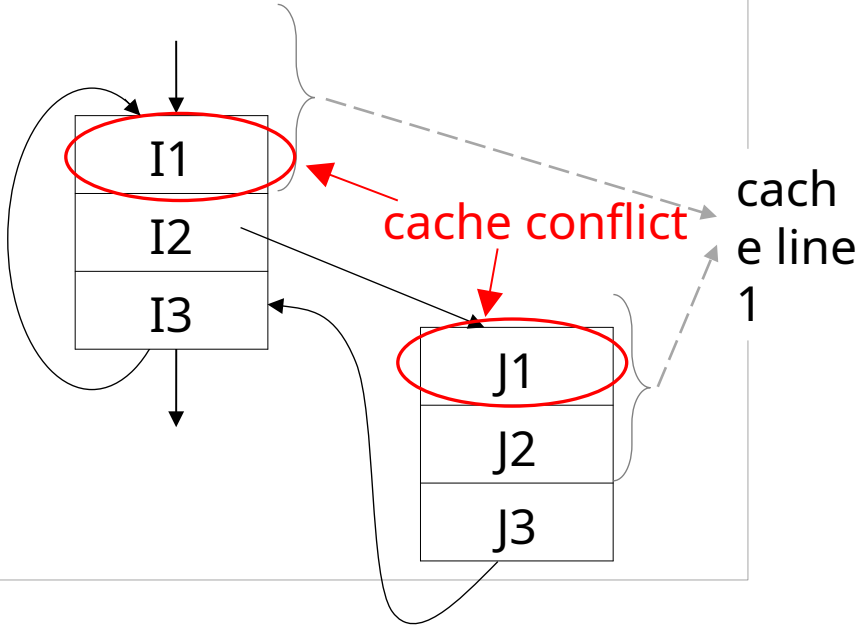
- if *addr* is currently in L1 i-cache: fetch the contents from i-cache and execute
- otherwise: cache miss at address *addr*:
  - a cache-line-sized block of memory containing address *addr* is loaded into the appropriate cache line
    - cost depends on where in the memory hierarchy that location is found
  - instruction at location *addr* brought into the CPU
    - the next few instructions will probably be found in the cache
  - the prior contents of that cache line are discarded
    - if any of the discarded instructions is executed again, that incurs another cache miss ("cache conflict")

Cache utilization issues

- Cache misses due to instructions mapping to different cache lines



- Cache conflicts due to instructions mapping into the same cache line



Profiling

Profiling

- Knowledge of (relative) execution counts can be very useful for guiding optimization.
  - Heuristics usually give only crude estimates; often inaccurate.
  - Direct measurement (*profiling*) gives better results.
- General approaches:
  - Sampling based: periodically samples the program counter (e.g., gprof).
  - Counter based: counts the number of occurrences of runtime events (e.g.: gcc –fprofile-generate/-fprofile-use)

For compiler optimization, counter-based profiling is more useful.

Examples of Counter-based Profiles

- Basic block profiles
  - Counts the number of times each basic block is executed.
- Edge profiles
  - Counts the number of times each control flow edge is taken.

Using Profiles in Compilation

1. Generate an instrumented executable.
   - Contains additional code to collect the profiles during execution, print them out at program exit.

gcc –fprofile-generate

2. Run the instrumented code on “typical” inputs.

3. Recompile the program, this time with the execution profile as an additional input.
   - The compiler uses the profile information to guide optimization decisions.

gcc –fprofile-use

Implementing Profiling

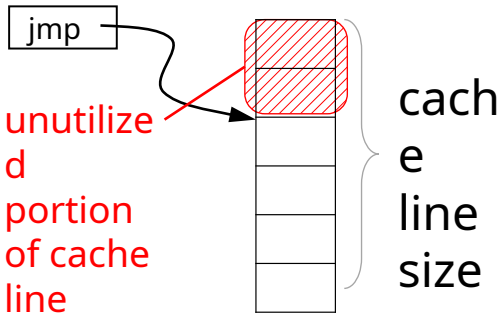
- Allocate a 0-initialized counter for each entity (basic block, edge, …) being profiled.
- Add code to update the appropriate counter when appropriate.
  - *Basic block profiling*: add code in the block to update the counter.
  - *Edge profiling*: “split” the edge by inserting a new basic block containing counter-update code.
- Add code to write out the profiles at end of execution.

Code generation: Improving cache utilization

Improving i-cache utilization

- Align the targets of frequently taken jumps and calls on cache line boundaries
  - add no-ops above the target if necessary

    gcc: -falign-functions, -falign-loops, -falign-jumps
- Put frequently-executed-together code close together in memory
  - puts groups of such instructions into the same cache line where possible
  - reduces the no. of cache misses



Profile-Guided Code Layout

Goal: improve memory hierarchy (esp. i-cache) performance by careful code layout.

Intuition: convert temporal locality to spatial locality.

- Code that is executed close together in time gets placed closed together in memory.
- This helps reduce cache conflicts and improves prefetching behavior.

Information needed: most likely outcome for each conditional branch

- execution frequency of edges in the control flow graph

Code Transformations for Layout

- Procedure positioning
  - the relative order of procedures in a program.
- Basic block positioning
  - The relative order of basic blocks within the code for a procedure.
- Procedure splitting
  - Place “hot code” within a procedure far away from the “cold code.”

Procedure Positioning

- *General idea*: if *p* calls *q* frequently, they should be placed close together in memory.
- *Benefits*:
  - The need for “long branch” instructions is reduced.
  - Assuming p and q end up on the same page, the amount of paging is reduced.
- *Data structure*: weighted call graph. This is an undirected graph where:
  - each vertex is a procedure in the program; and
  - There is an edge (*p*, *q*) with weight *k* if there are *k* calls between *p* and *q* (from the program’s profile).

Procedure Positioning: Cluster Graphs

Given a call graph *G* for a program *P*,
construct a *cluster graph* *G*' = (*V*', *E*'):

- Each vertex in G' is a set (cluster) of procedures of *P*.
- Initially:
  - each procedure is in a cluster containing only itself.
  - edges in G' are derived from the edges in the call graph G.

Procedure Positioning: Algorithm

Carry out *node merging* on the cluster graph:

while $E' \neq \varnothing$:

- Find a heaviest edge $e \equiv (a, b) \in E'$;
- merge clusters a and b, remembering the order of merges;
- delete e from E', update other edge weighs accordingly.

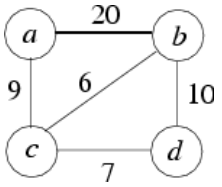
Procedure Positioning: Algorithm (cont'd)

Generate a procedure placement ordering:

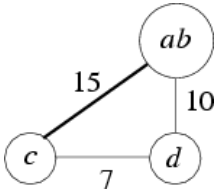
- Initially, *Order* = ε;
- Consider the vertices in the order in which they were added to the cluster.
  - For each vertex *v*, attach v to whichever end of *Order* the vertex *v* has a higher edge weight to (in the original call graph for the program).

Procedure Ordering: Example

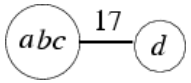
Initial cluster graph:



After merging {a}, {b}:



After merging {a,b}, {c}:



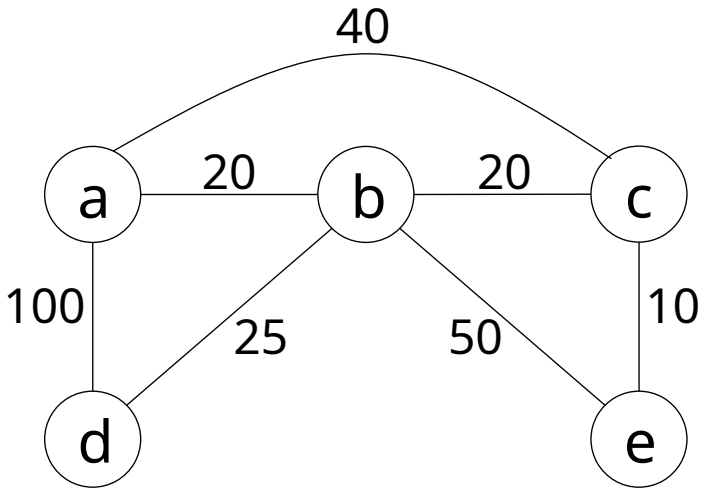
After merging {a,b,c}, {d}:

Final procedure layout: c, a, b, d

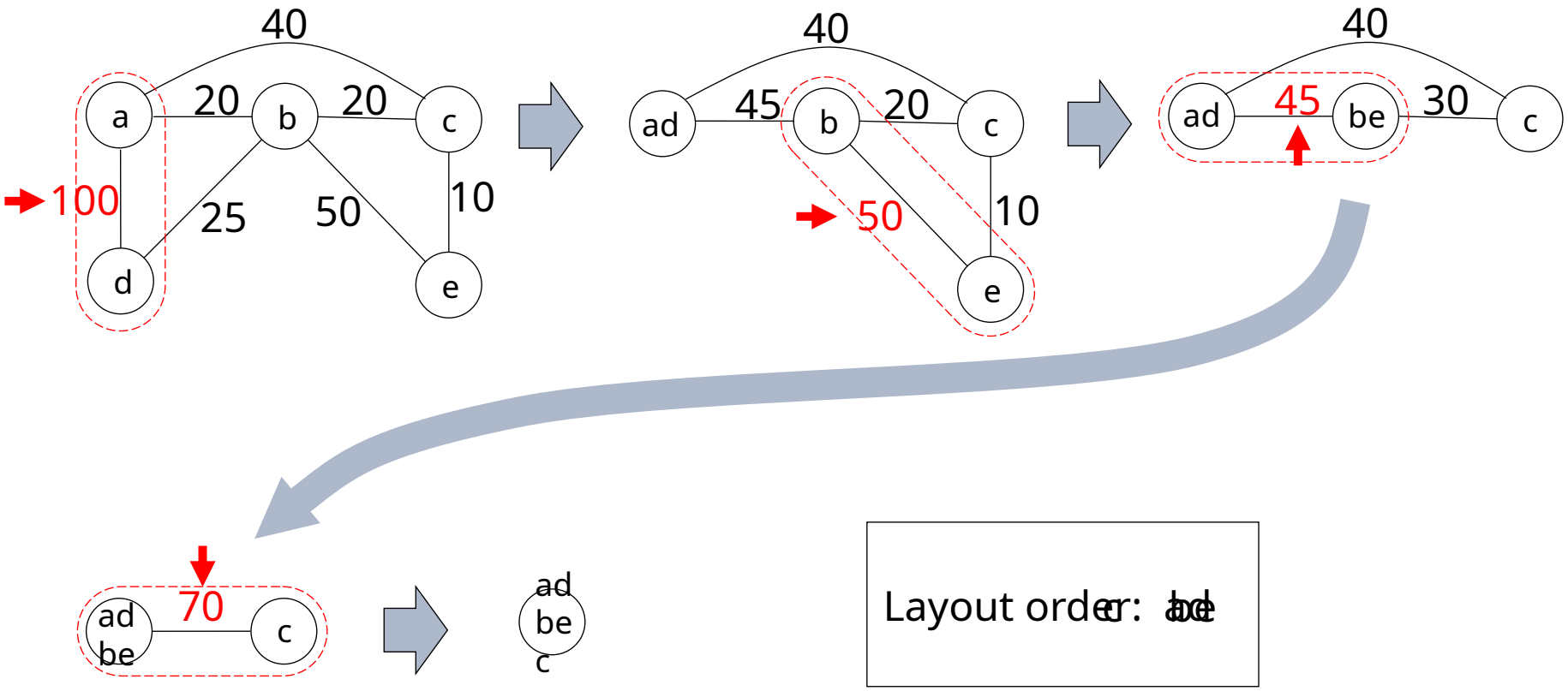
From the original call graph, c prefers to be closer to a than to b; d prefers to be near b.

EXERCISE

Work out a good procedure placement ordering given the following weighted call graph:

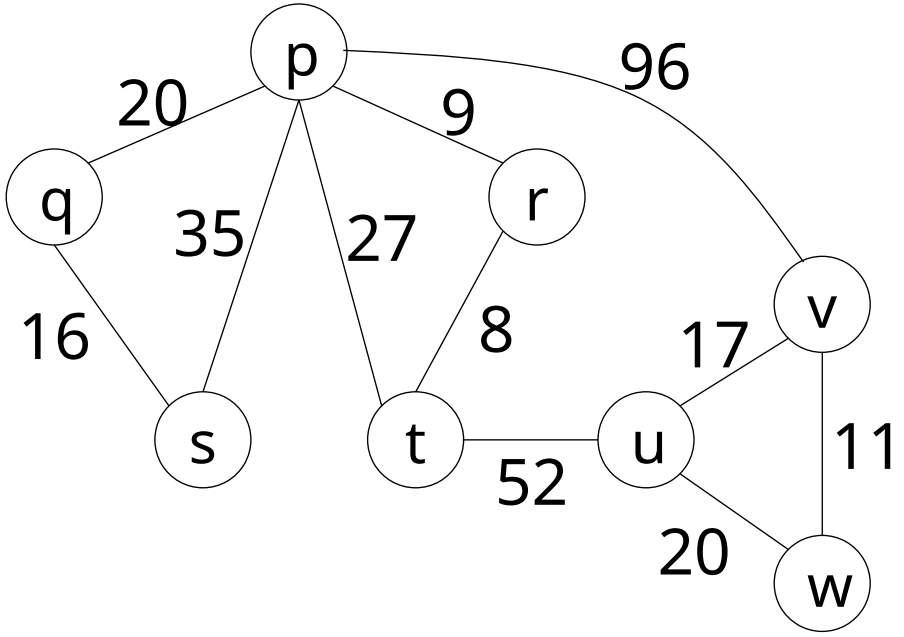


EXERCISE - solution



EXERCISE

Work out a good procedure placement ordering given the following weighted call graph:

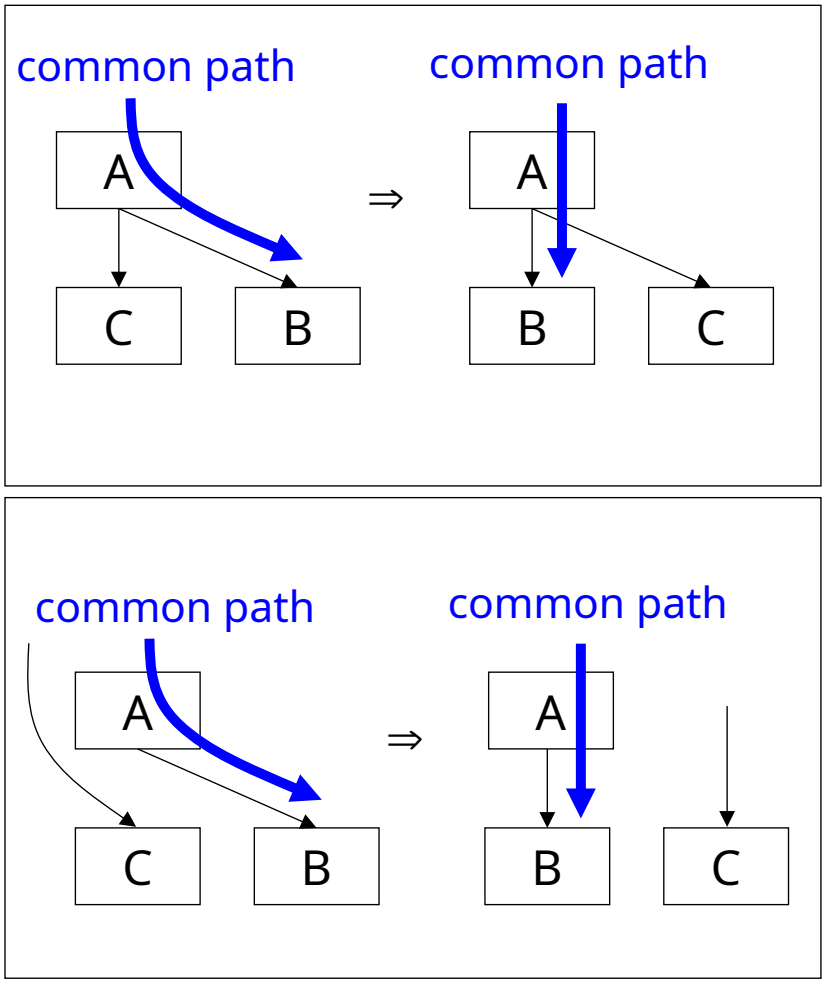


Procedure Positioning: Effects

- *Speed improvement*: up to ~10%.
- *Long branch instructions*:
  - *Static count*: increases significantly (~65–400%).
  - *Dynamic count*: the no. of long branch instructions executed decreases by 80–98%.
    - » Profile-guided code positioning improves the behavior of “hot” code even though it could adversely affect “cold” code.
- *Memory hierarchy*:
  - no data presented in paper
  - authors claim it improves paging, TLB, and i-cache performance.

Basic block ordering

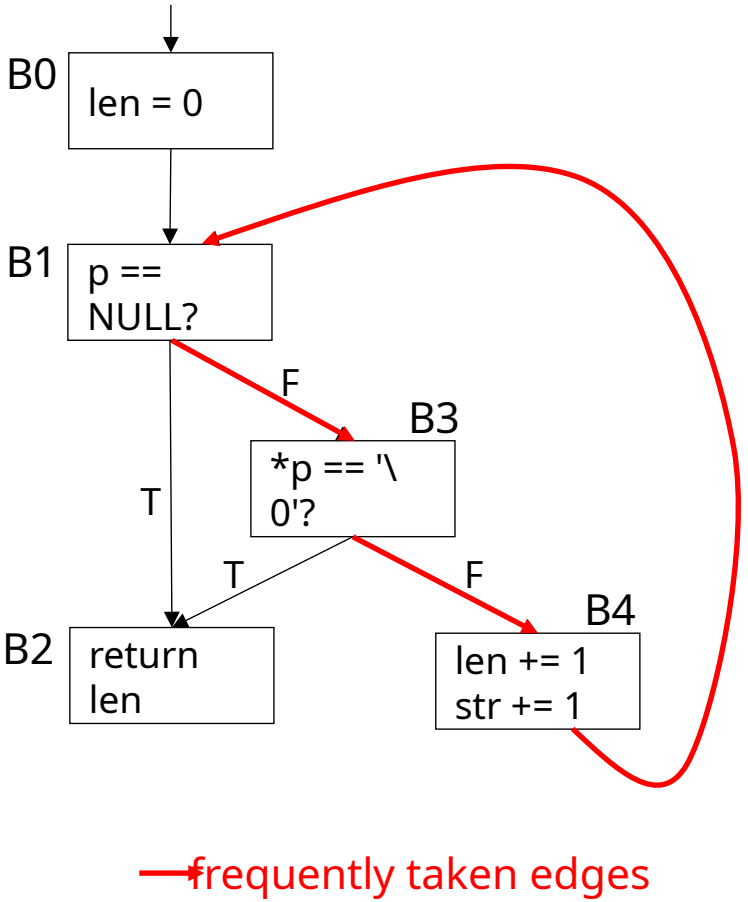
- *General idea*:
  - if block B is executed soon after block A, place B close to A in memory.
- *Equivalent formulation*:
  - orient branches to maximize the not-taken (I.e., fall through) case;
  - eliminate unconditional branches if possible.



Example

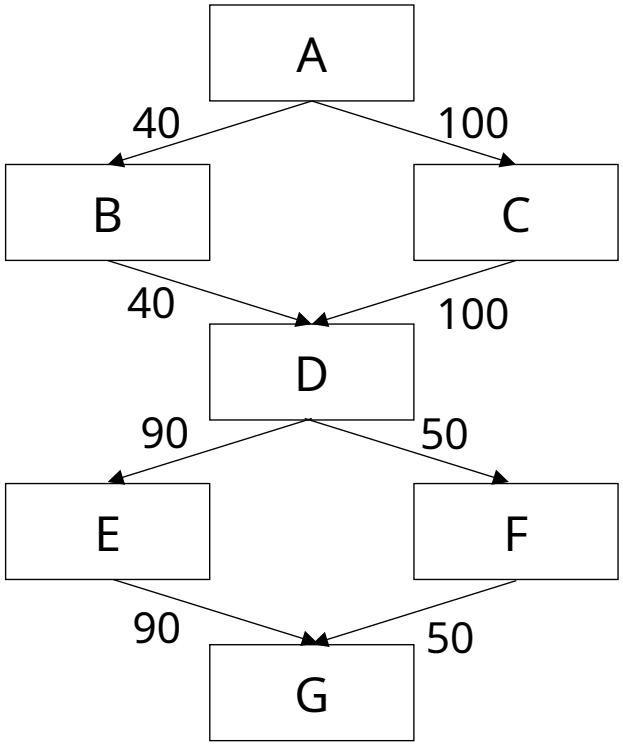
```
int strlen(char *str) {
   len = 0;
   while (str != NULL && *str != '\0')
{
      len += 1;
      str += 1;
   }
   return len;
}
```

Code Layout	
Unoptimized	Optimized
B0	B0
B1	B1
B2	B3
B3	B4
B4	B2



EXERCISE

What is a good basic block order for the following weighted control flow graph? (Edge weights represent execution counts.)



Basic Block Ordering: Algorithm

1. Compute an edge profile for the program.
2. Process the edges in descending order of execution count:
   - For each edge e, chain e if possible; else discard.

   This results in a collection of chains of basic blocks
3. Concatenate these chains together to get the final basic block order.

Basic Block Ordering: Chaining edges

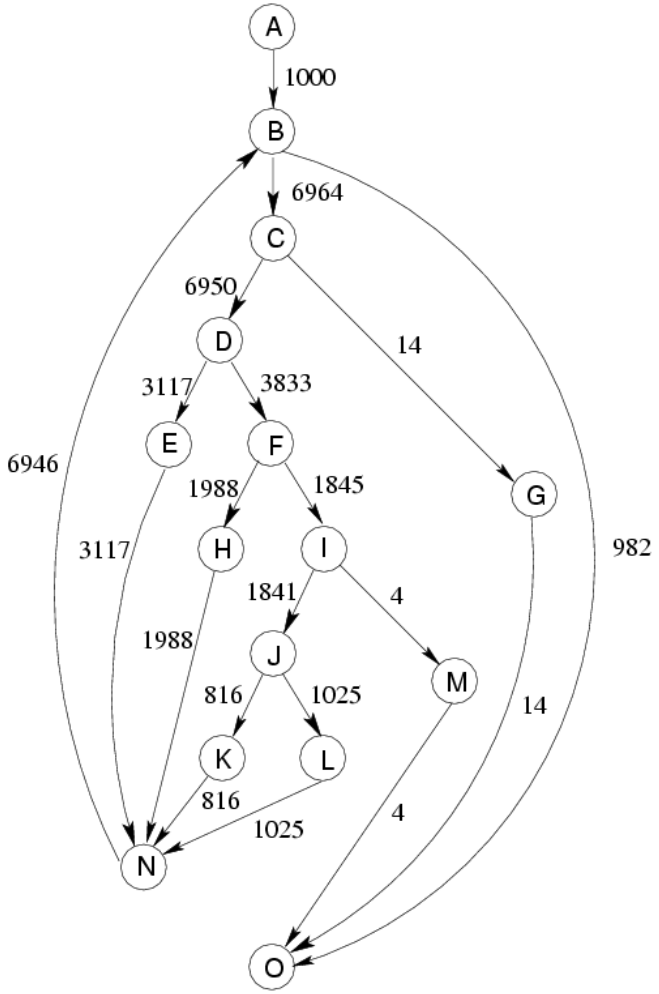
Given a set of chains **S**; an edge $e \equiv a \rightarrow b$ ($a \neq b$):

- if neither a nor b belongs to any chain in **S**:
  - Start a new chain containing only e; add this to **S**.
- else if an existing edge $C \in \mathbf{S}$ can be extended using e (i.e., C ends with a or begins with b), then extend C with e:
  - If C ≡ 'b → ...' then extended C ≡ 'a → b → ...'
  - If C ≡ '... → a' then extended C ≡ '... → a → b'
- else discard e.

Basic Block Ordering: Example

From: “Profile-Guided Code Positioning” by Pettis and Hansen, *Proc. PLDI 1990.*

Initial control flow graph



Basic Block Ordering: Example

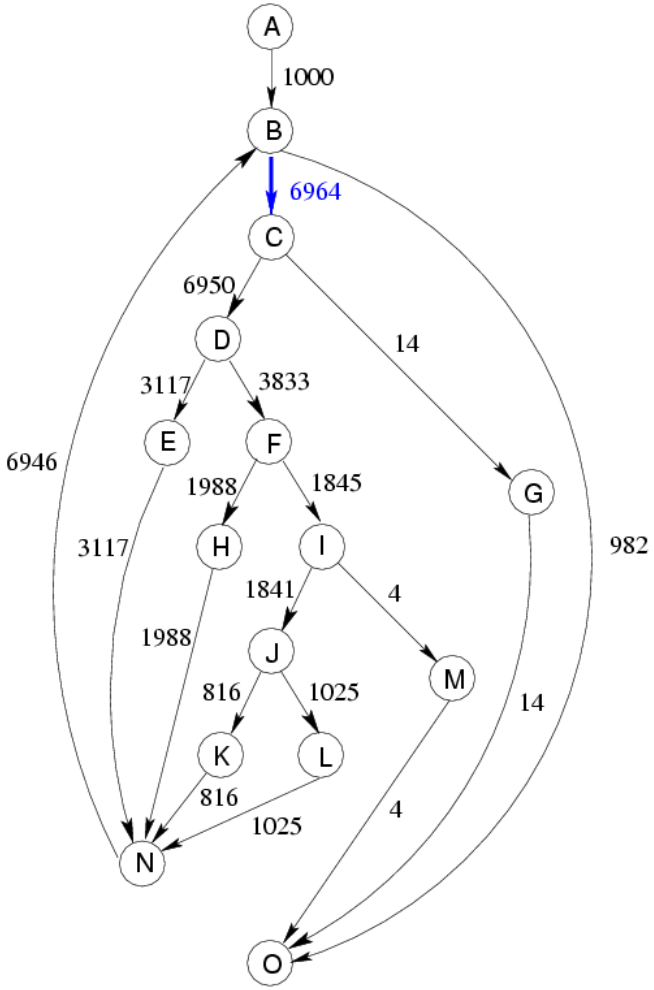
Step 1:

current chains:

- none

current edge: B → C

action: new chain



Basic Block Ordering: Example

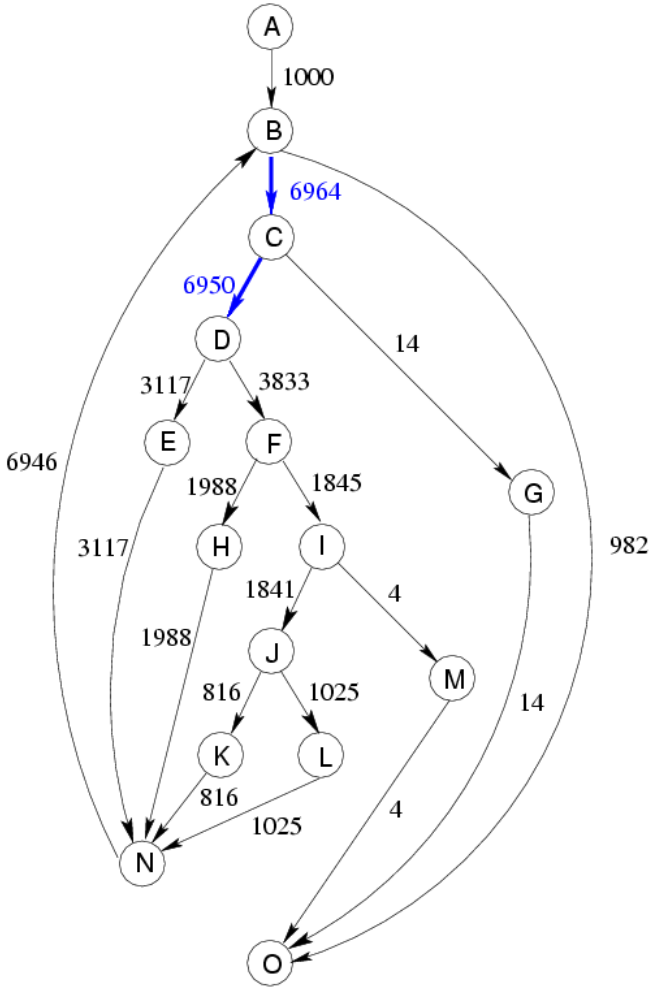
Step 2:

current chains:

- B → C

current edge: C → D

action: attach to chain



Basic Block Ordering: Example

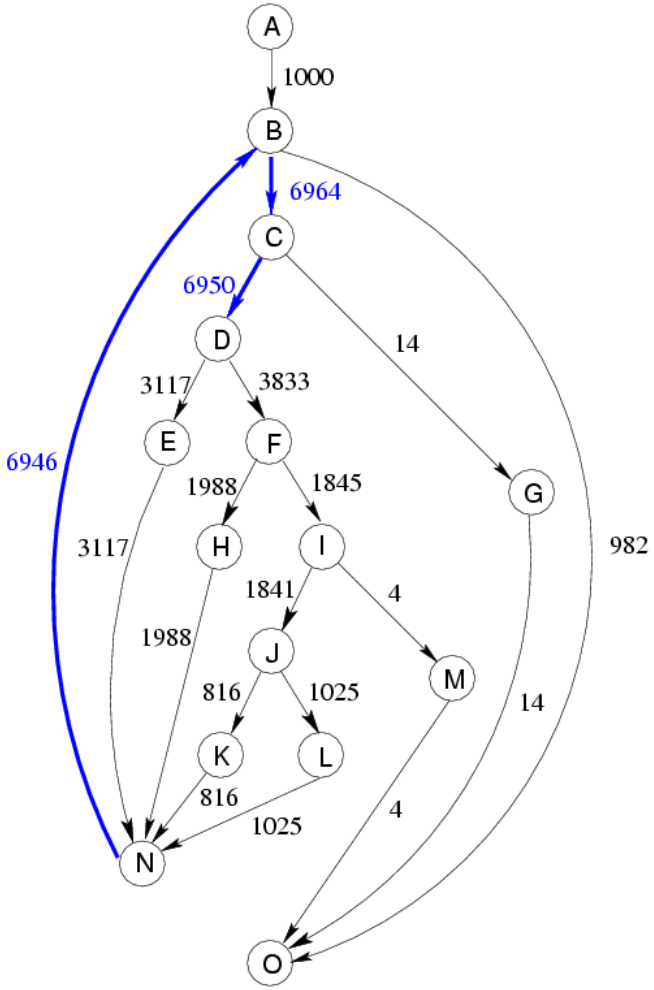
Step 3:

current chains:

- B → C → D

current edge: N → B

action: attach to chain



Basic Block Ordering: Example

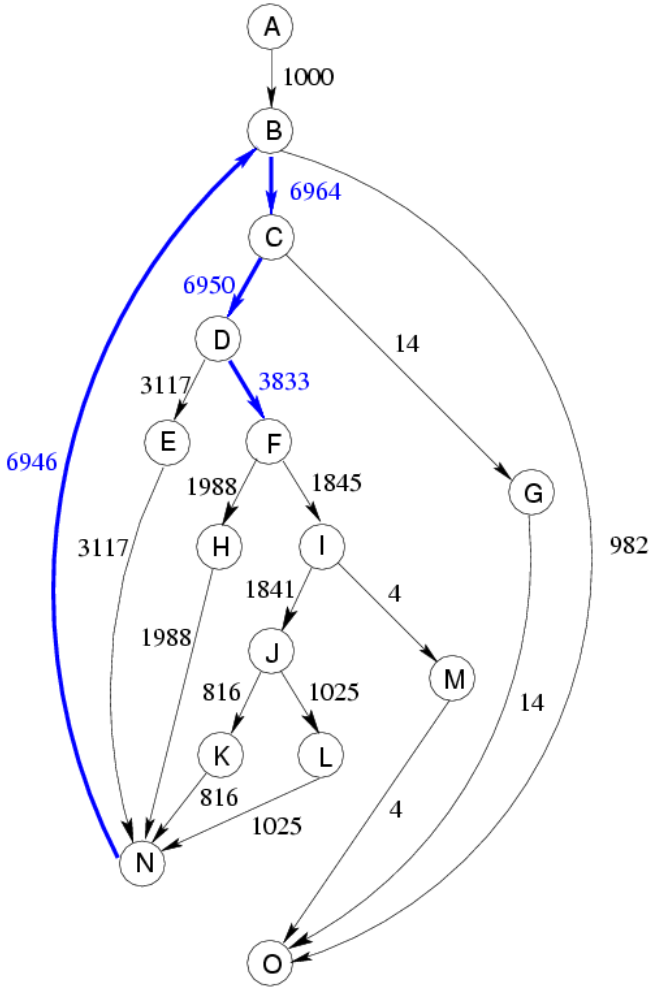
Step 4:

current chains:

- N → B → C → D

current edge: D → F

action: add to chain



Basic Block Ordering: Example

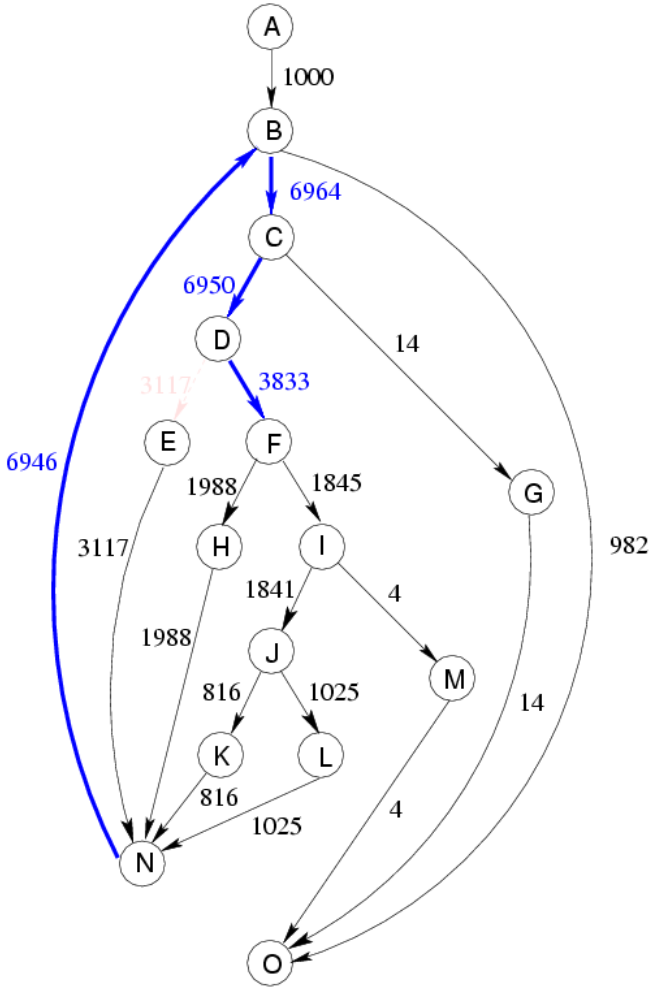
Step 5:

current chains:

- N → B → C → D → F

current edge: D → E

action: discard



Basic Block Ordering: Example

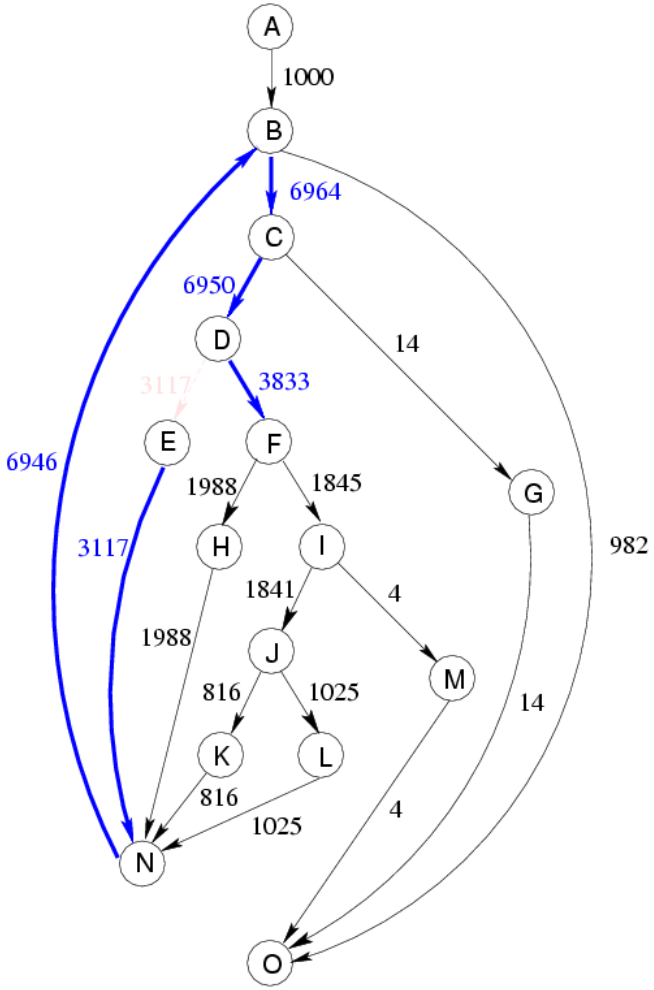
Step 6:

current chains:

- N → B → C → D → F

current edge: E → N

action: add to chain



Basic Block Ordering: Example

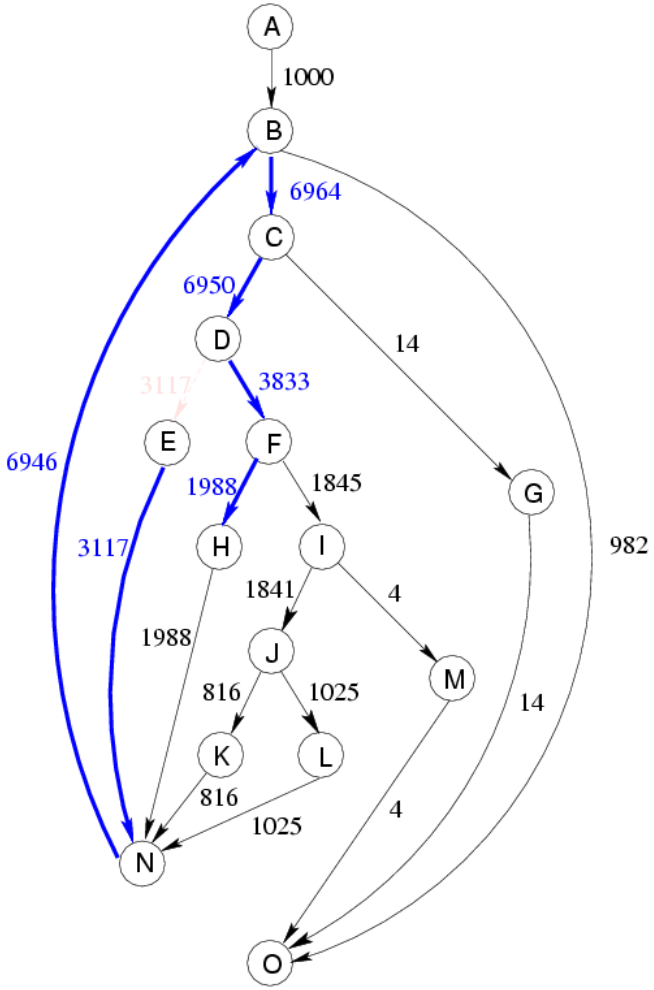
Step 7:

current chains:

- E → N → B → C → D → F

current edge: F → H

action: add to chain



Basic Block Ordering: Example

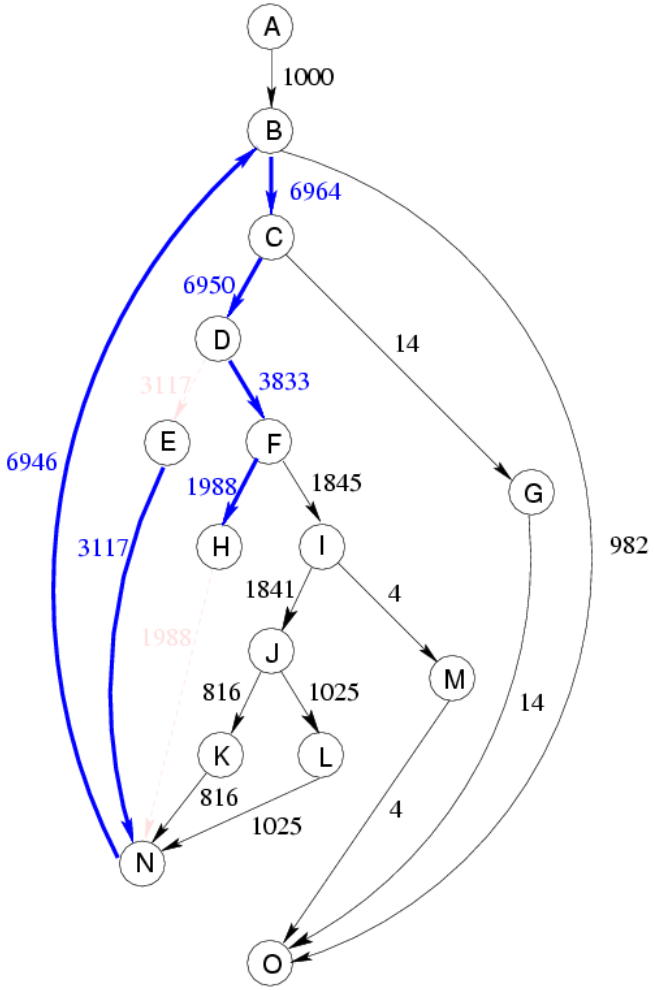
Step 8:

current chains:

- E → N → B → C → D → F → H

current edge: H → N

action: discard



Basic Block Ordering: Example

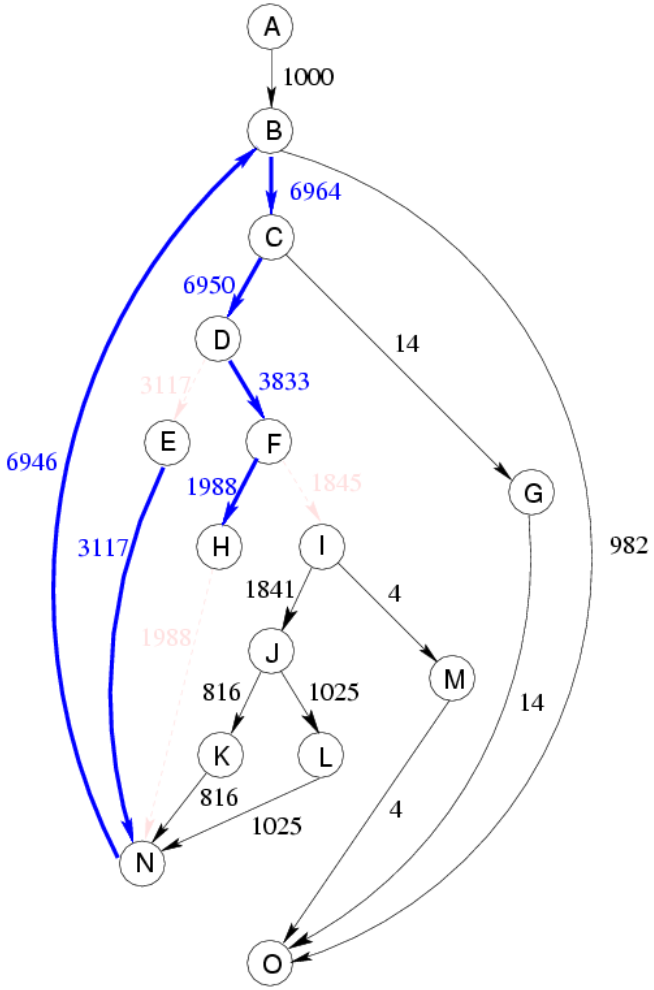
Step 9:

current chains:

- E → N → B → C → D → F → H

current edge: F → I

action: discard



Basic Block Ordering: Example

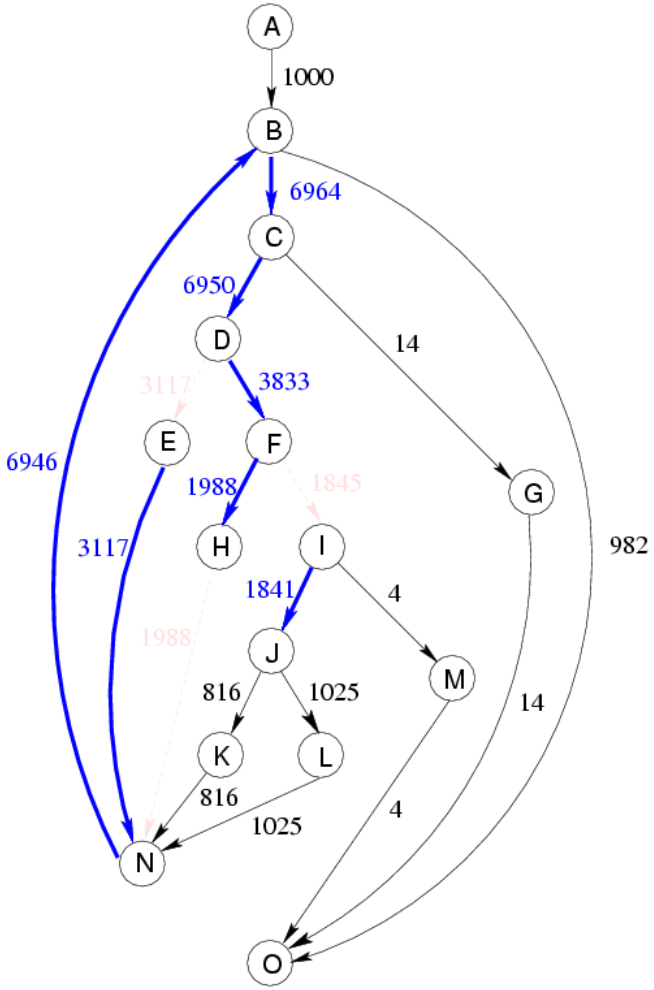
Step 10:

current chains:

- E → N → B → C → D → F → H

current edge: I → J

action: new chain



Basic Block Ordering: Example

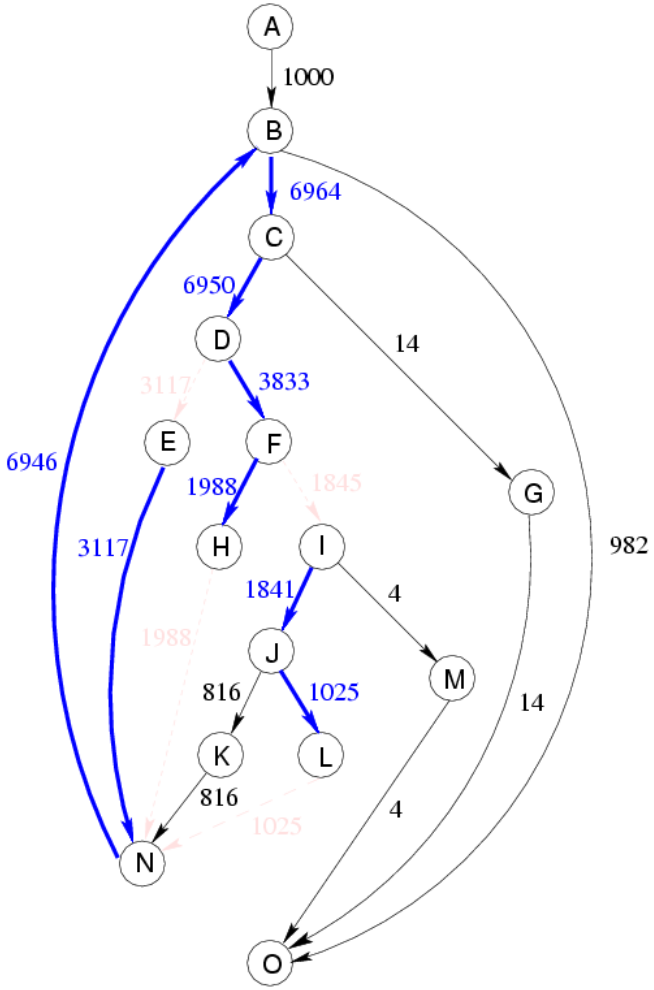
Step 11:

current chains:

- E → N → B → C → D → F → H
- I → J

current edge: J → L

action: attach to chain



Basic Block Ordering: Example

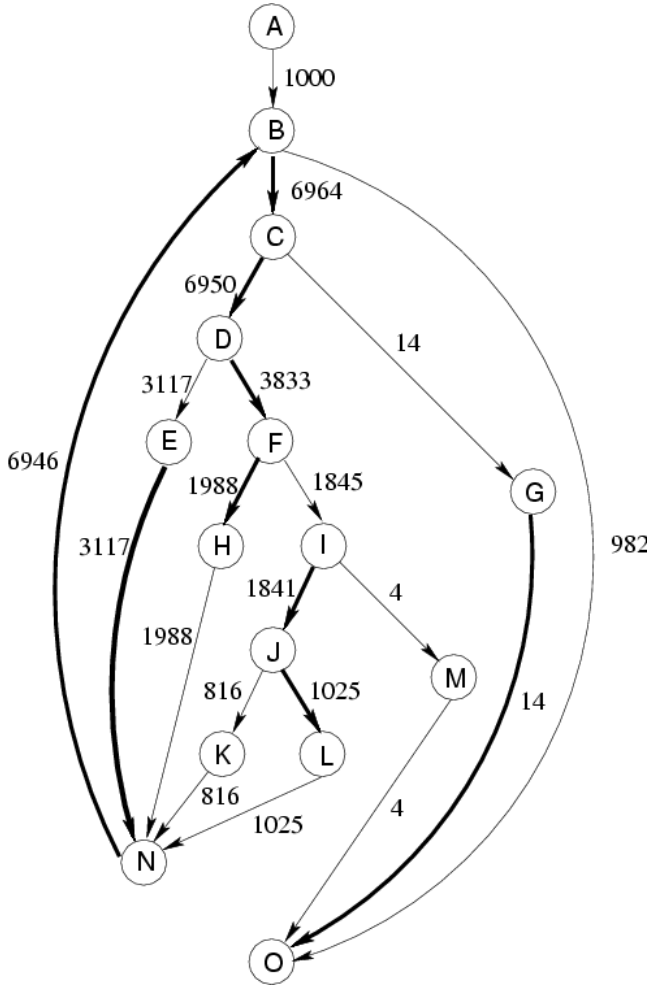
Final:

current chains:

- E → N → B → C → D → F → H
- I → J → L
- G → O

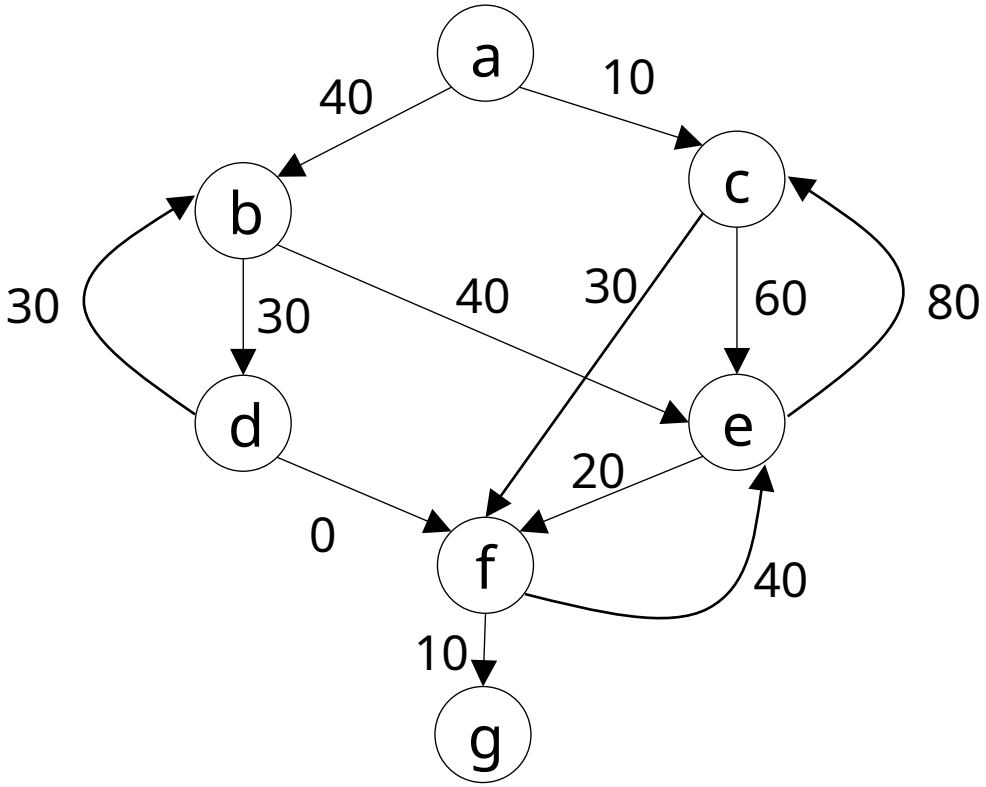
Final layout:

A, E-N-B-C-D-F-H, I-J-L, G-O, K, M



EXERCISE

Work out a good procedure placement ordering given the following weighted call graph:



Basic Block Ordering: Effects

[from Pettis-Hansen, PLDI ’90]:

- Speed improvement: 0.6% to 16.2%

  Together with procedure positioning: 2.1% to 20.3%

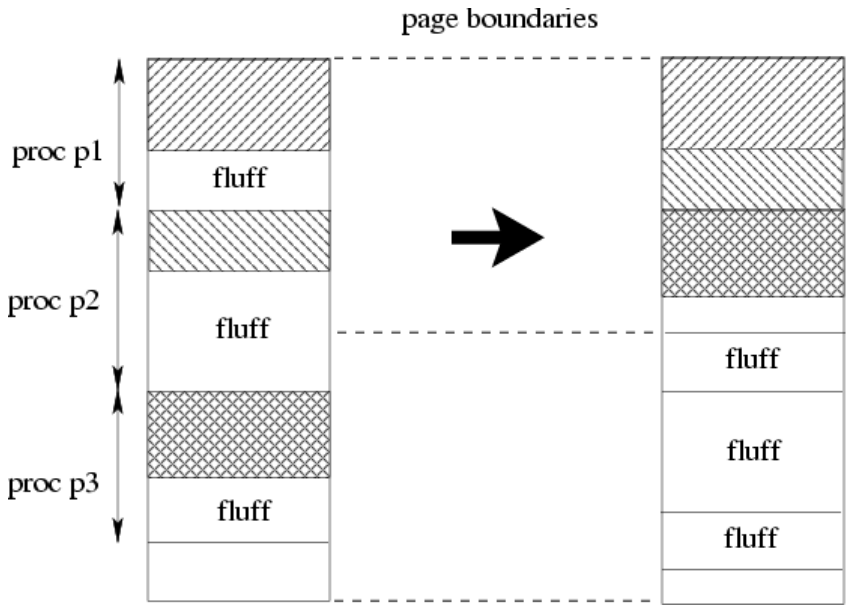
  Basic block positioning, by itself, can still incur cache conflicts between blocks from different functions. Combining it with procedure placement fixes this.

- Other improvements:
  - significant reduction in i-cache misses
  - significant decrease in branch misprediction penalty cycles
  - small reduction in the no. of instructions executed

    This happens because we don’t have to jump around cold code interspersed with the hot code.

Procedure Splitting

- Effects of basic block ordering for a function:
  - frequently executed blocks moved to the beginning;
  - infrequently executed code moved to the end.
- Procedure splitting:
  - moves blocks with execution count = 0 (“fluff blocks”) to a separate page.



Procedure Splitting: Issues and Effects

- Issue:
  - after procedure splitting, some blocks may be so far away that they cannot be reached with a “short branch” instruction.
  - Solution: place a stub block in the non-fluff code that uses a “long branch” to jump to the target.
- Effects:
  - magnifies the effects of basic block and procedure placement by further improving locality.
  - Speed improvements with all three transformations [Pettis & Hansen, PLDI ’90]: 2.1% to 26.0%.