15-740/18-740 Computer Architecture Lecture 21: Superscalar Processing

Prof. Onur Mutlu
Carnegie Mellon University

Announcements

- Project Milestone 2
  - Due November 10

- Homework 4
  - Out today
  - Due November 15

Last Two Lectures

- SRAM vs. DRAM
- Interleaving/Banking
- DRAM Microarchitecture
  - Memory controller
  - Memory buses
  - Banks, ranks, channels, DIMMs
  - Address mapping: software vs. hardware
  - DRAM refresh
- Memory scheduling policies
- Memory power/energy management
- Multi-core issues
  - Fairness, interference
  - Large DRAM capacity

Today

- Superscalar processing

Readings

- Required (New):
  - Patel et al., "Evaluation of design options for the trace cache fetch mechanism," IEEE TC 1999.
  - Palacharla et al., "Complexity Effective Superscalar Processors," ISCA 1997.

- Required (Old):
  - **Smith and Sohi, "The Microarchitecture of Superscalar Processors," Proc. IEEE, Dec. 1995.**
  - Stark, Brown, Patt, "On pipelining dynamic instruction scheduling logic," MICRO 2000.
  - Boggs et al., "The microarchitecture of the Pentium 4 processor," Intel Technology Journal, 2001.
  - Kessler, "The Alpha 21264 microprocessor," IEEE Micro, March-April 1999.

- Recommended:
  - Rotenberg et al., "Trace Cache: a Low Latency Approach to High Bandwidth Instruction Fetching," MICRO 1996.

Types of Parallelism

- Task level parallelism
  - Multitasking, multiprogramming” multiple different tasks need to be completed
    - e.g., multiple simulations, audio and email
  - Multiple tasks executed concurrently to exploit this

- Thread (instruction stream) level parallelism
  - Program divided into multiple threads that can execute in parallel. Each thread
    - can perform the same “task” on different data (e.g. zoom in on an image)
    - can perform different tasks on same/different data (e.g. database trans.)
  - Multiple threads executed concurrently to exploit this

- Instruction level parallelism
  - Processing of different instructions can be carried out independently
  - Multiple instructions executed concurrently to exploit this

Exploiting ILP via Pipelining

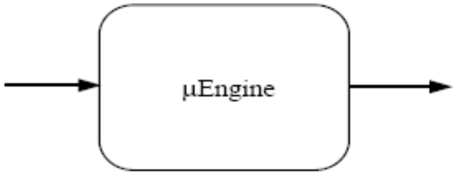
- Pipelining
  - Increases the number of instructions processed concurrently in the machine
  - Exploits parallelism within the "instruction processing cycle"
    - One instruction being fetched when another is executed
  - So far we have looked at only scalar pipelines

- Scalar execution
  - One instruction fetched, issued, retired per cycle (at most)
  - The best case CPI of a scalar pipeline is 1.

Reducing CPI beyond 1

- CPI vs. IPC
  - Inverse of each other
  - IPC more commonly used to denote retirement of multiple instructions

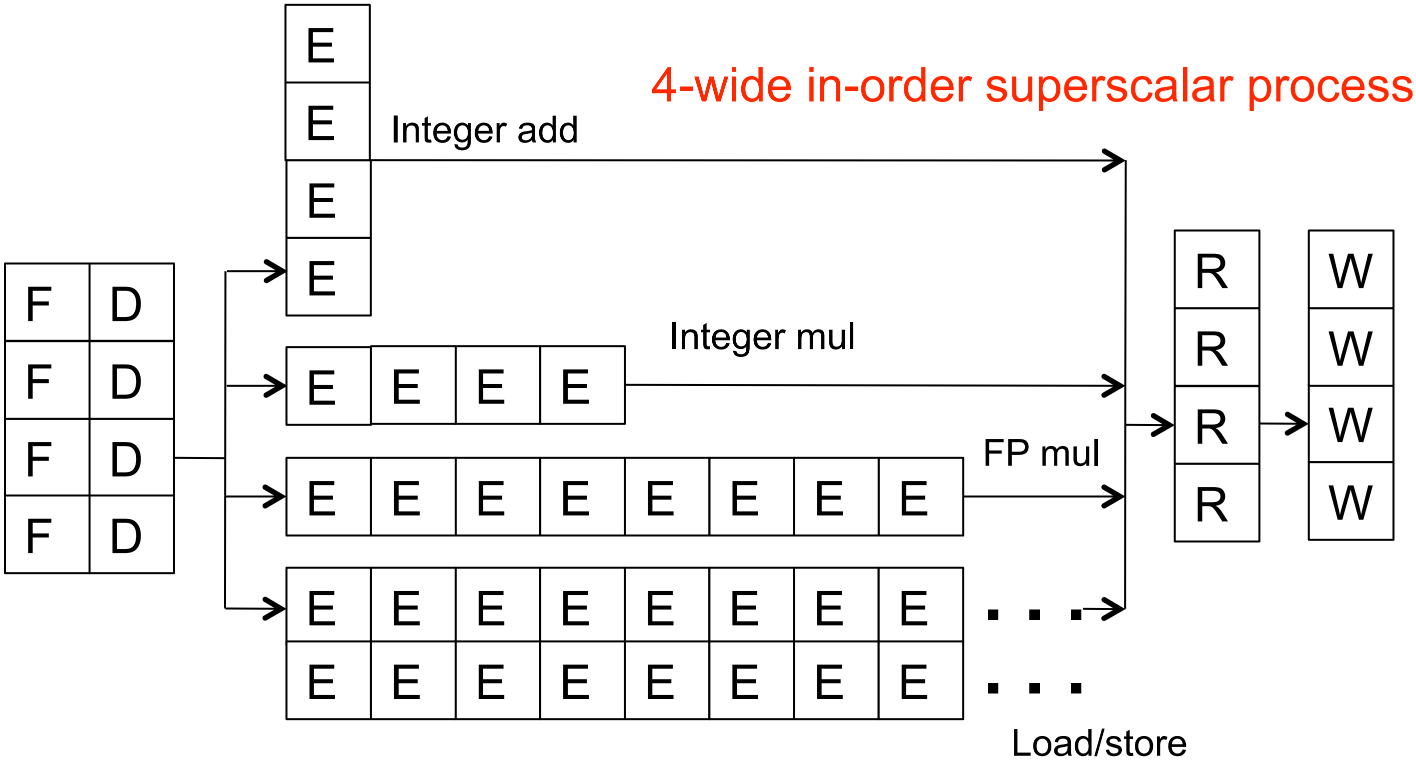
- Flynn's bottleneck
  - You cannot retire more than you fetch
  - If we want IPC > 1, we need to fetch > 1 instruction per cycle.



Can't get out more than you put in.

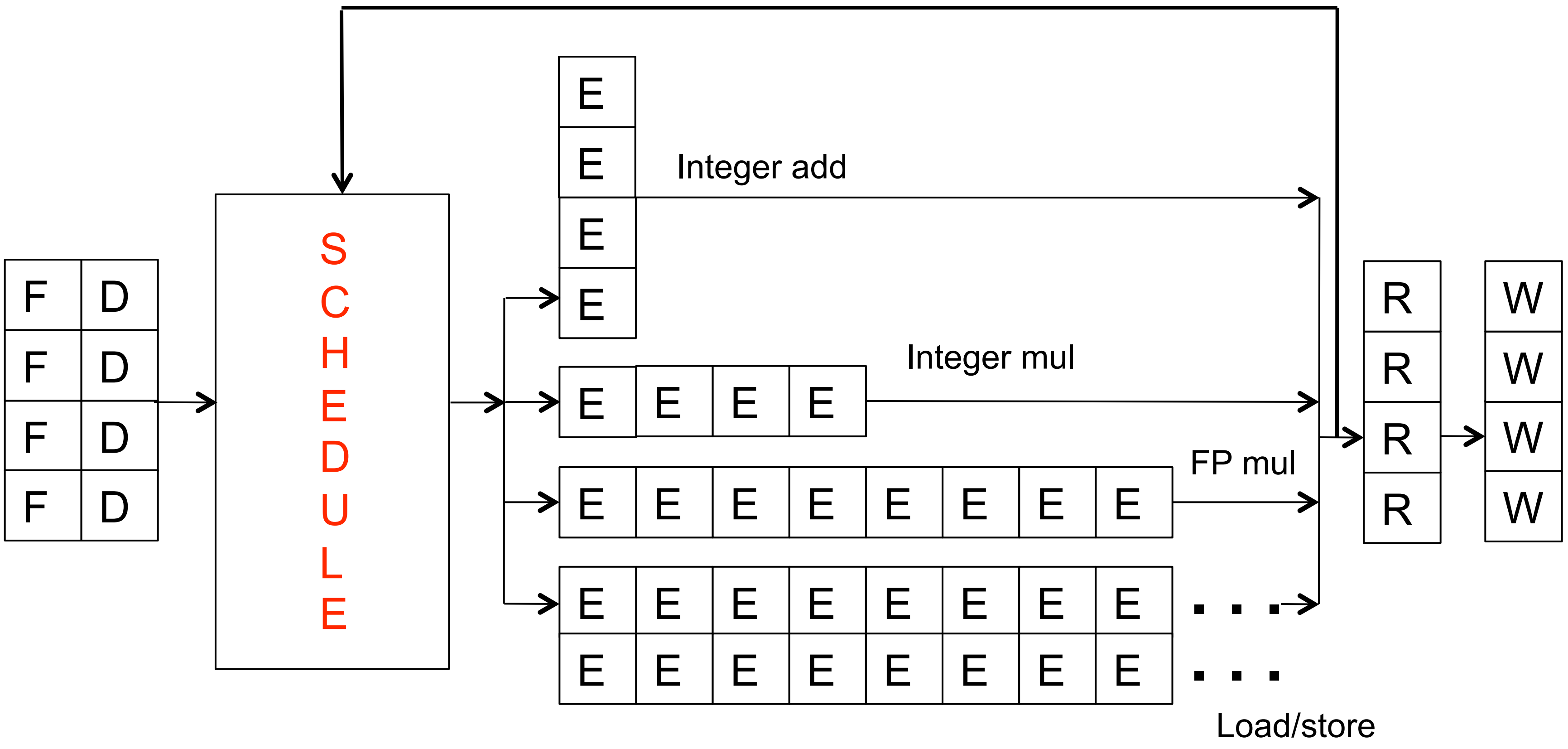
Superscalar Processing

- Idea: Have multiple pipelines to fetch, decode, execute, and retire multiple instructions per cycle
- Can be used with in-order or out-of-order execution
- Superscalar width: number of pipelines

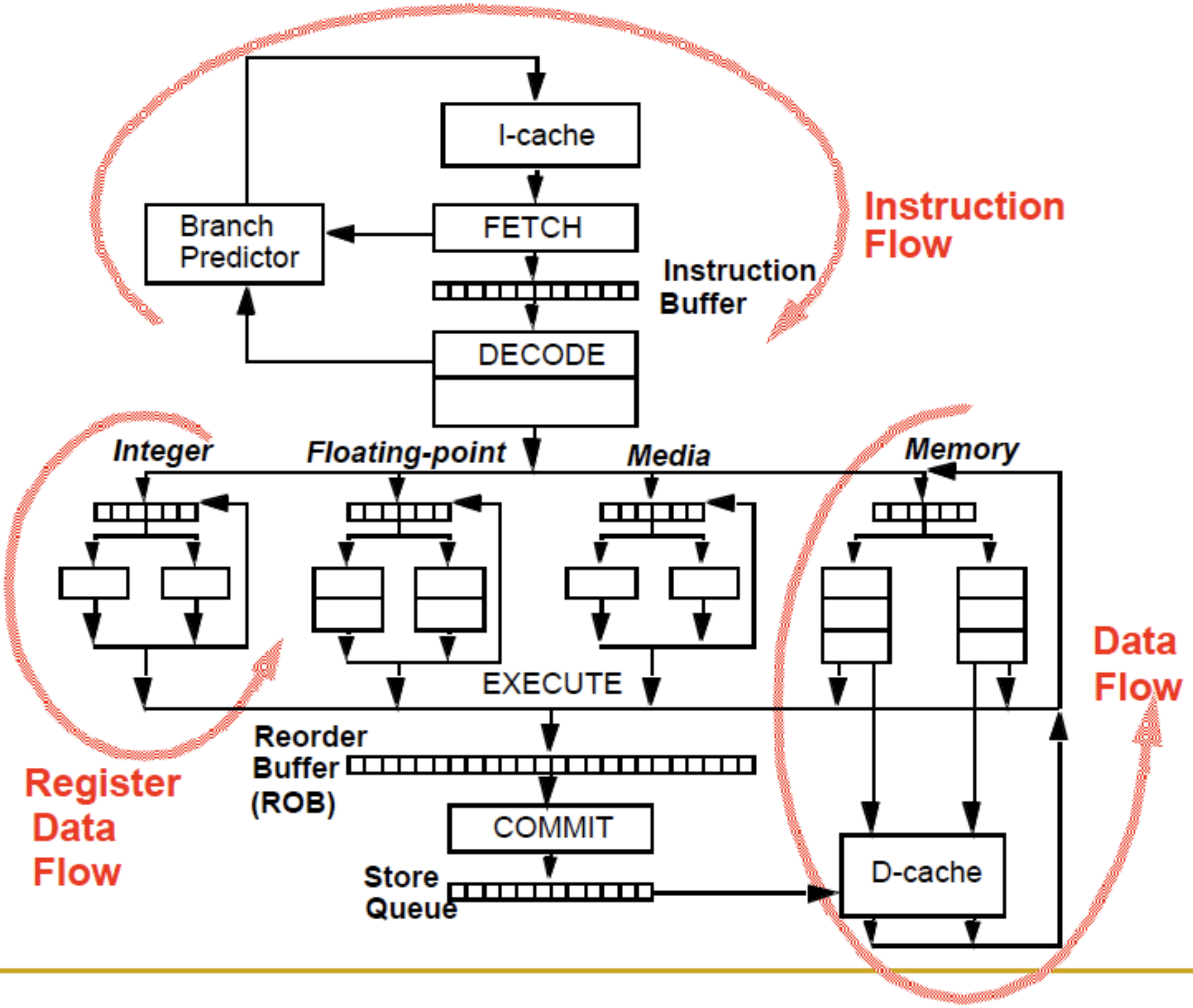


4-wide in-order superscalar processor

4-wide Superscalar Out-of-order Processor



A Superscalar Out-of-order Processor



Superscalar Processing

- Fetch (supply N instructions)
- Decode (generate control signals for N instructions)
- Rename (detect dependencies between N instructions)
- Dispatch (determine readiness and select N instructions to execute in-order or out-of-order)
- Execute (have enough functional units to execute N instructions + forwarding paths to forward results of N instructions)
- Write into Register File (have enough ports to write results of N instructions)
- Retire (N instructions)

Fetching Multiple Instructions Per Cycle

- Two problems

1. Alignment of instructions in I-cache
   - What if there are not enough (N) instructions in the cache line to supply the fetch width?

2. Fetch break: Branches present in the fetch block
   - Fetching sequential instructions in a single cycle is easy
   - What if there is a control flow instruction in the N instructions?
   - Problem: The direction of the branch is not known but we need to fetch more instructions

- These can cause effective fetch width < peak fetch width

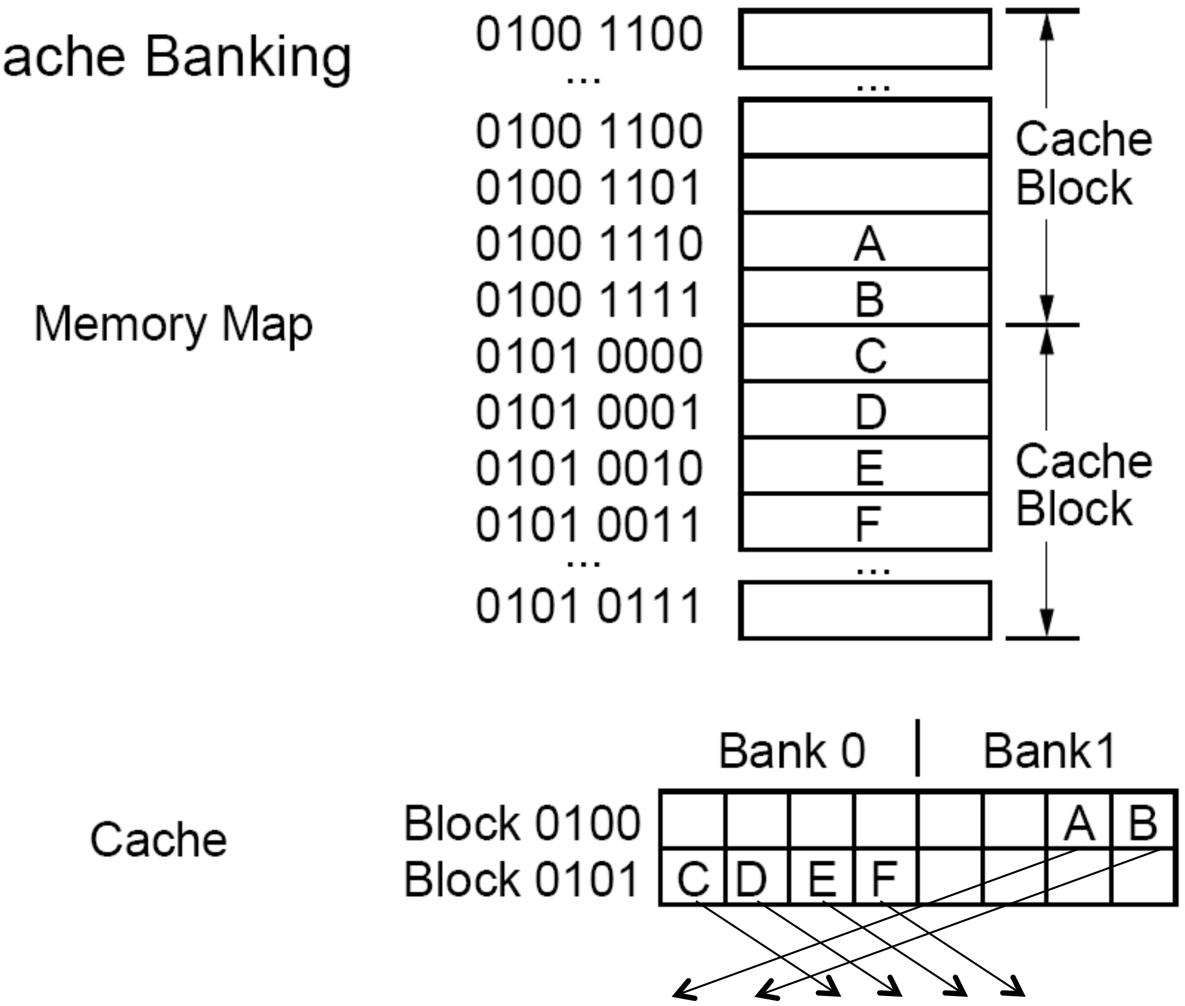
Wide Fetch Solutions: Alignment

- Large cache blocks: Hope N instructions contained in the block

- Split-line fetch: If address falls into second half of the cache block, fetch the first half of next cache block as well
  - Enabled by banking of the cache
  - Allows sequential fetch across cache blocks in one cycle
  - Pentium and AMD K5

Split Line Fetch

Cache Banking



Memory Map

0100 1100

...

0100 1100

0100 1101

0100 1110 A

0100 1111 B

0101 0000 C

0101 0001 D

0101 0010 E

0101 0011 F

...

0101 0111

Cache Block

Cache Block

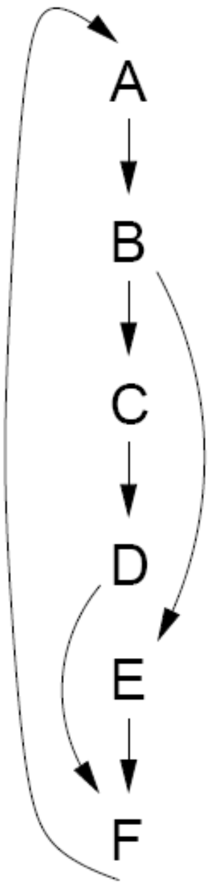
Cache

Bank 0 | Bank1

Block 0100							A	B
Block 0101	C	D	E	F				

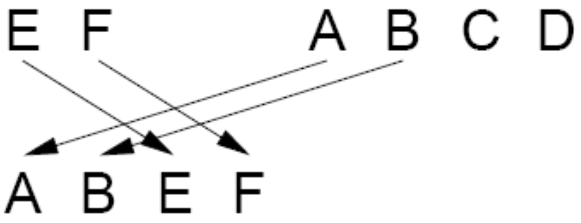
Need alignment logic:

Short Distance Predicted-Taken Branches

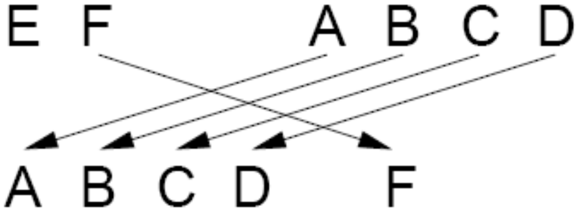


	Bank 0				Bank1			
Block 0100					A	B	C	D
Block 0101	E	F						

First Iteration (Branch B taken to E)



Second Iteration (Branch B fall through to C)



Techniques to Reduce Fetch Breaks

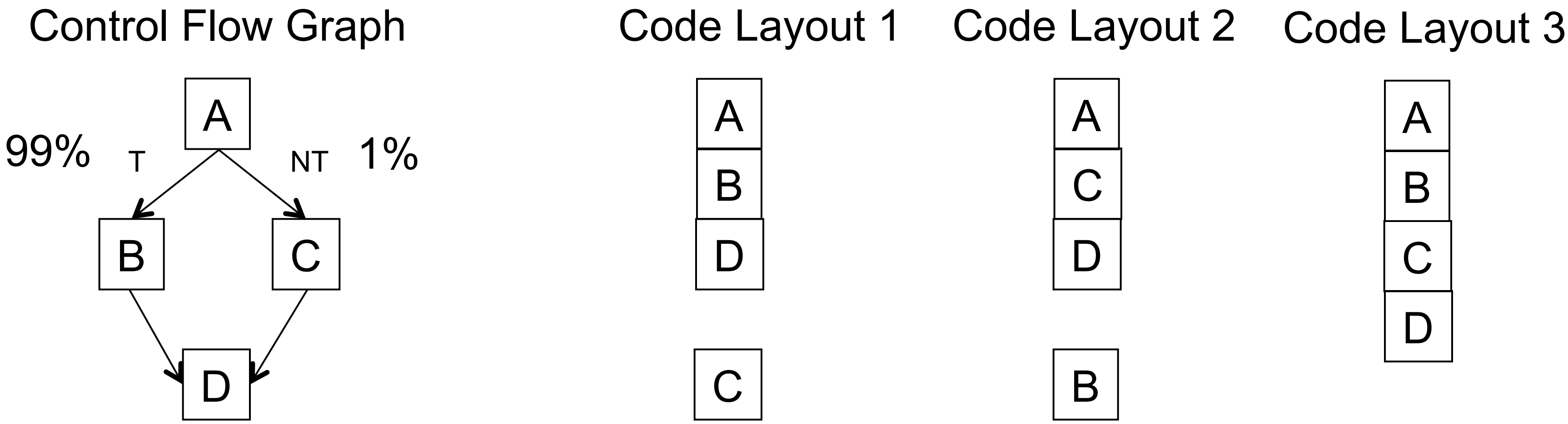
- Compiler
  - Code reordering (basic block reordering)
  - Superblock

- Hardware
  - Trace cache

- Hardware/software cooperative
  - Block structured ISA

Basic Block Reordering

- Not-taken control flow instructions not a problem: no fetch break: make the likely path the not-taken path
- Idea: Convert taken branches to not-taken ones
  - i.e., reorder basic blocks (after profiling)
  - Basic block: code with a single entry and single exit point



- Code Layout 1 leads to the fewest fetch breaks

Basic Block Reordering

- Pettis and Hansen, "Profile Guided Code Positioning," PLDI 1990.

- Advantages:

  + Reduced fetch breaks (assuming profile behavior matches runtime behavior of branches)

  + Increased I-cache hit rate

  + Reduced page faults

- Disadvantages:

  -- Dependent on compile-time profiling

  -- Does not help if branches are not biased

  -- Requires recompilation

Superblock

- Idea: Combine frequently executed basic blocks such that they form a single-entry multiple exit larger block, which is likely executed as straight-line code

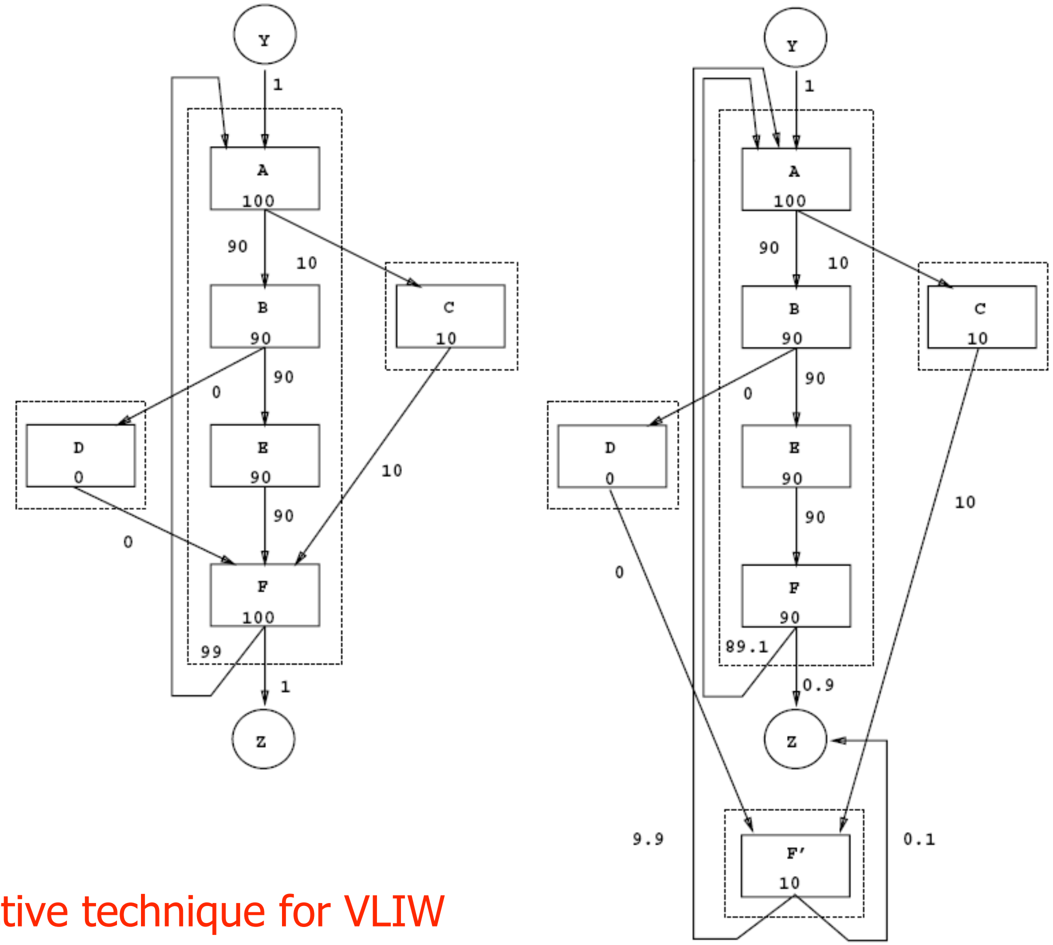
+ Helps wide fetch

+ Enables aggressive compiler optimizations and code reordering within the superblock

-- Increased code size

-- Profile dependent

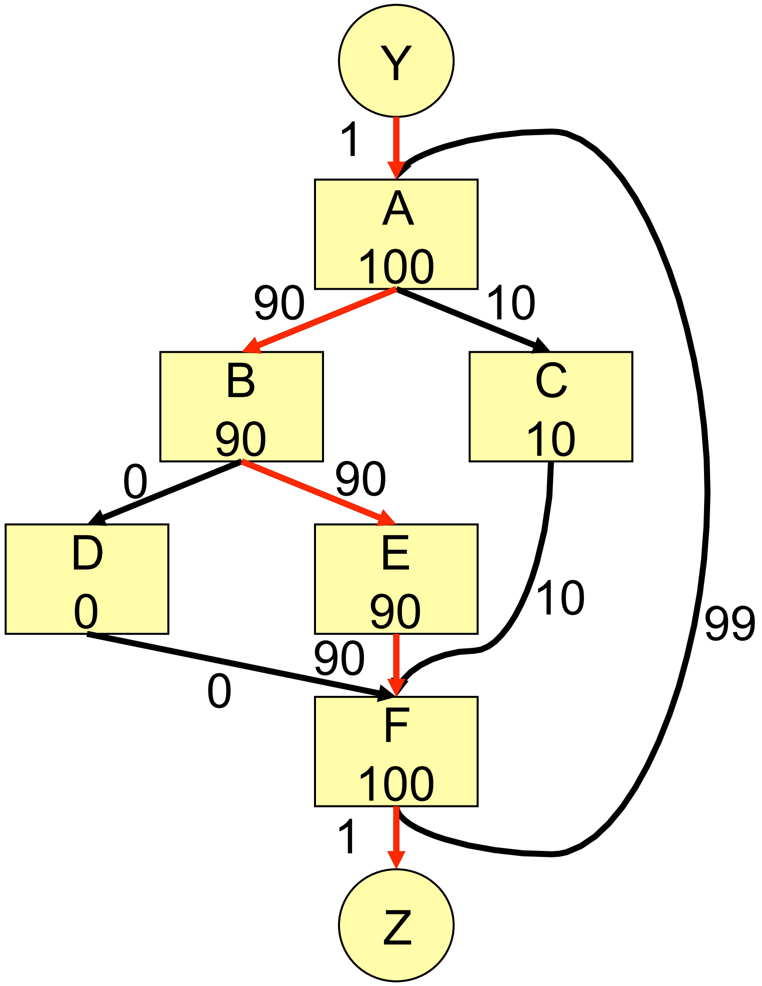
-- Requires recompilation

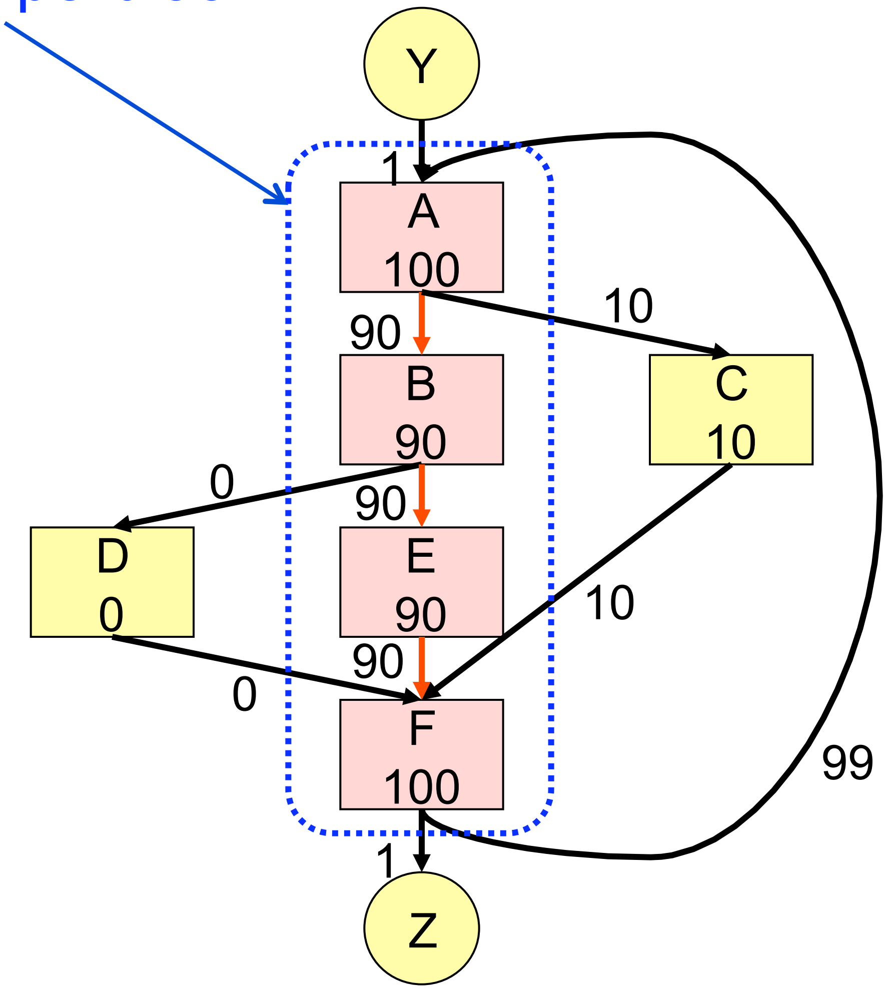


- Hwu et al. "The Superblock: An effective technique for VLIW and superscalar compilation," Journal of Supercomputing, 1993.

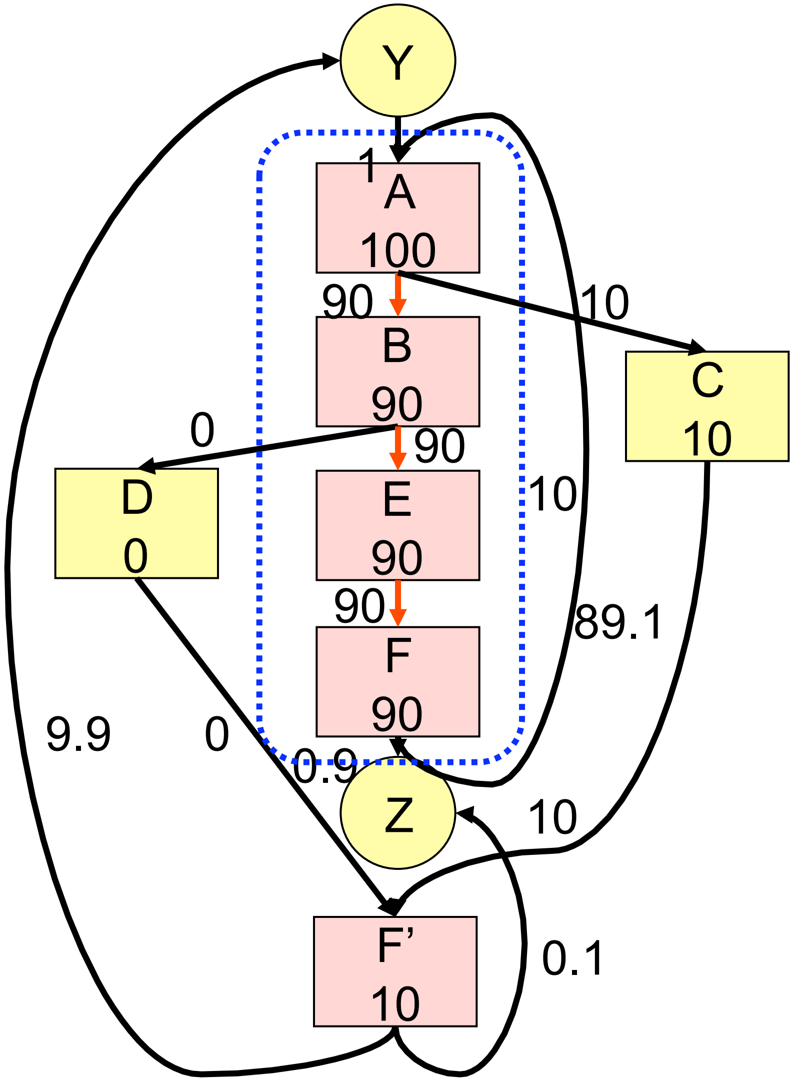
Superblock Formation (I)

Is this a superblock?



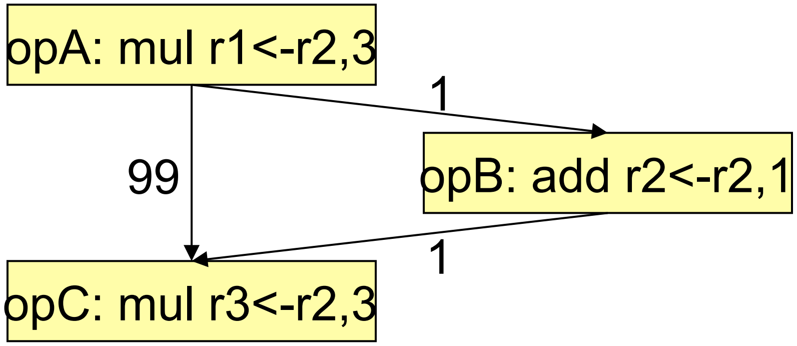


Superblock Formation (II)

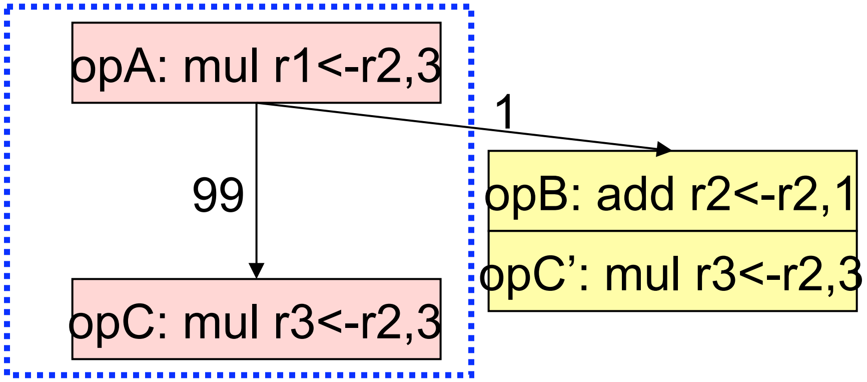


Tail duplication:
duplication of basic blocks after a side entrance to eliminate side entrances
→ transforms
a trace into a superblock.

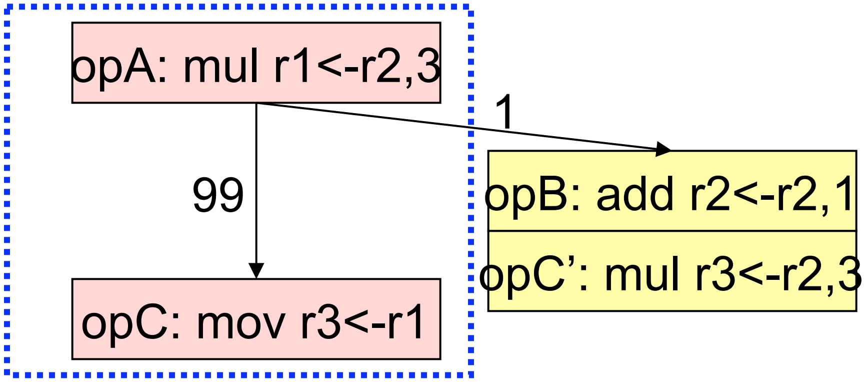
Superblock Code Optimization Example



Original Code



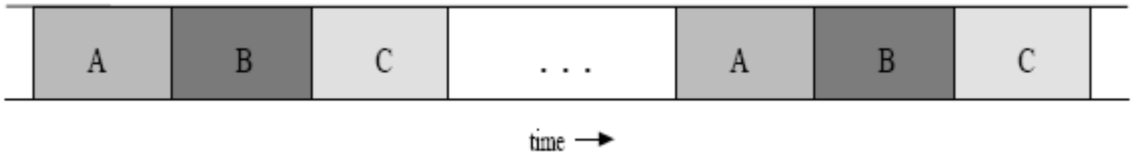
Code After Superblock Formation



Code After Common Subexpression Elimination

Reducing Fetch Breaks: Trace Cache

- Dynamically determine the basic blocks that are executed consecutively
- Trace: Consecutively executed basic blocks
- Idea: Store consecutively-executed basic blocks in physically-contiguous internal storage (called trace cache)



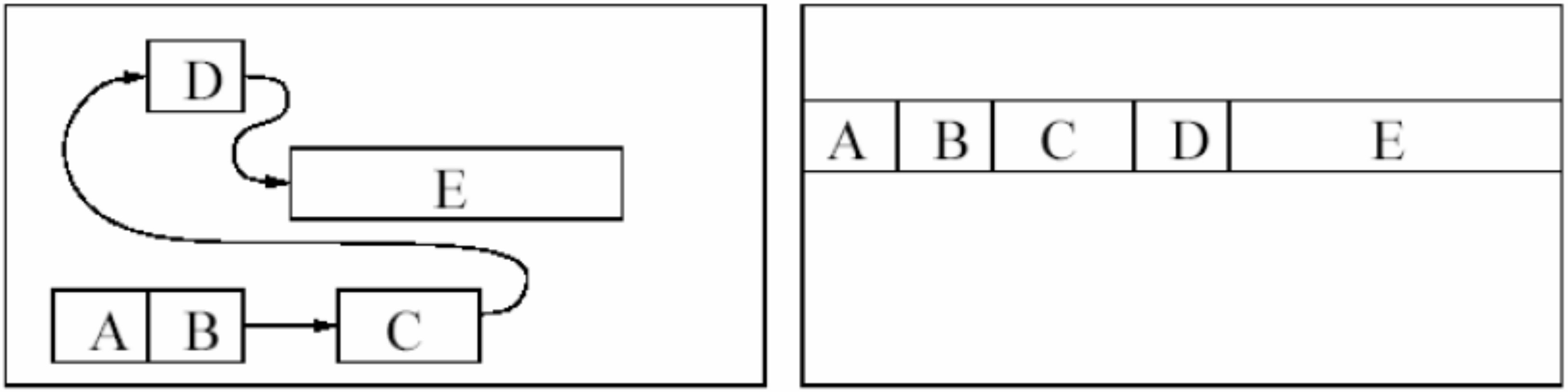
Dynamic Instruction Stream

- Basic trace cache operation:
  - Fetch from consecutively-stored basic blocks (predict next trace or branches)
  - Verify the executed branch directions with the stored ones
  - If mismatch, flush the remaining portion of the trace

- Rotenberg et al., "Trace Cache: a Low Latency Approach to High Bandwidth Instruction Fetching," MICRO 1996.
- Patel et al., "Critical Issues Regarding the Trace Cache Fetch Mechanism," Umich TR, 1997.

Trace Cache: Basic Idea

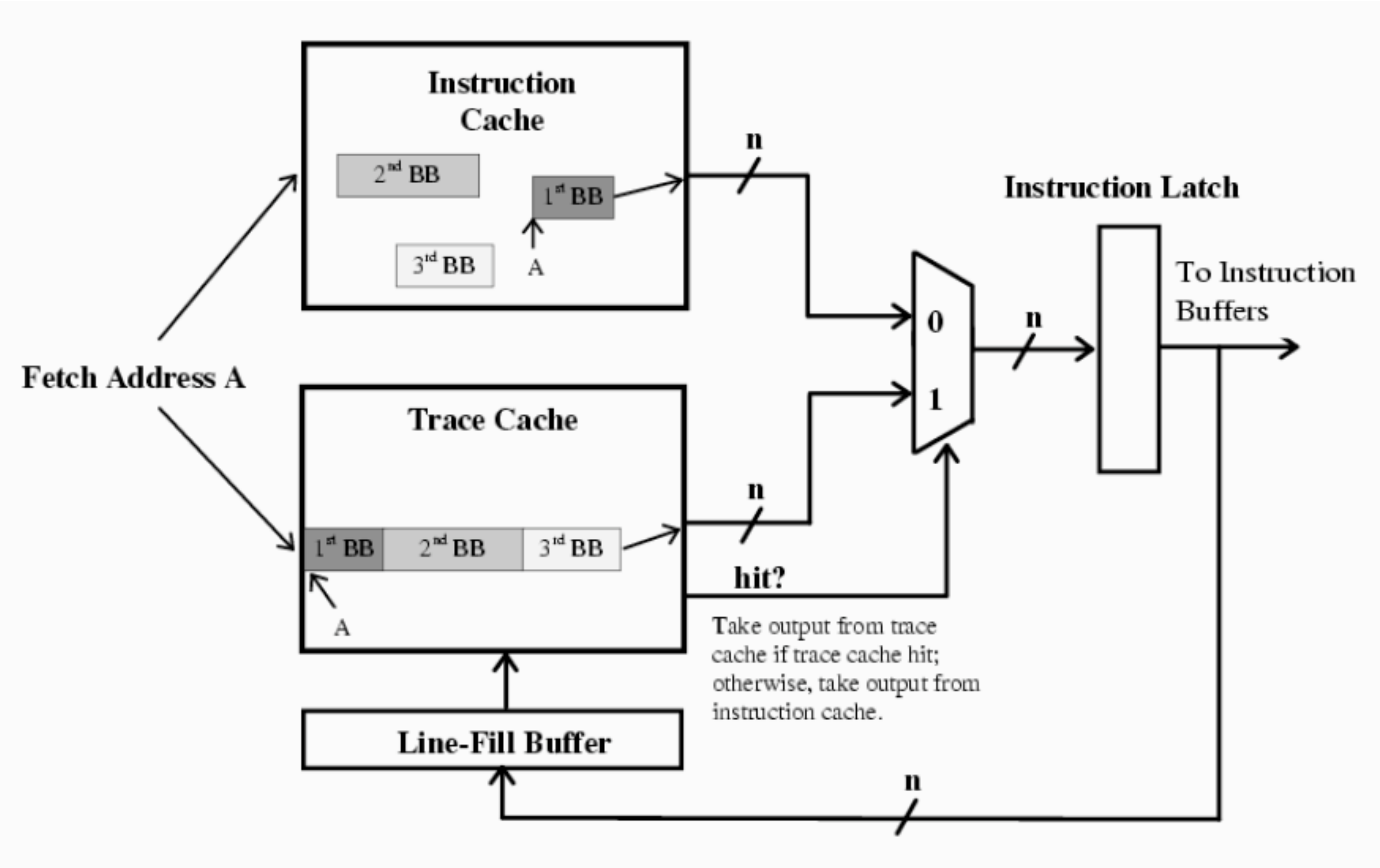
- A trace is a sequence of instructions starting at any point in a dynamic instruction stream.
- It is specified by a start address and the branch outcomes of control transfer instructions.



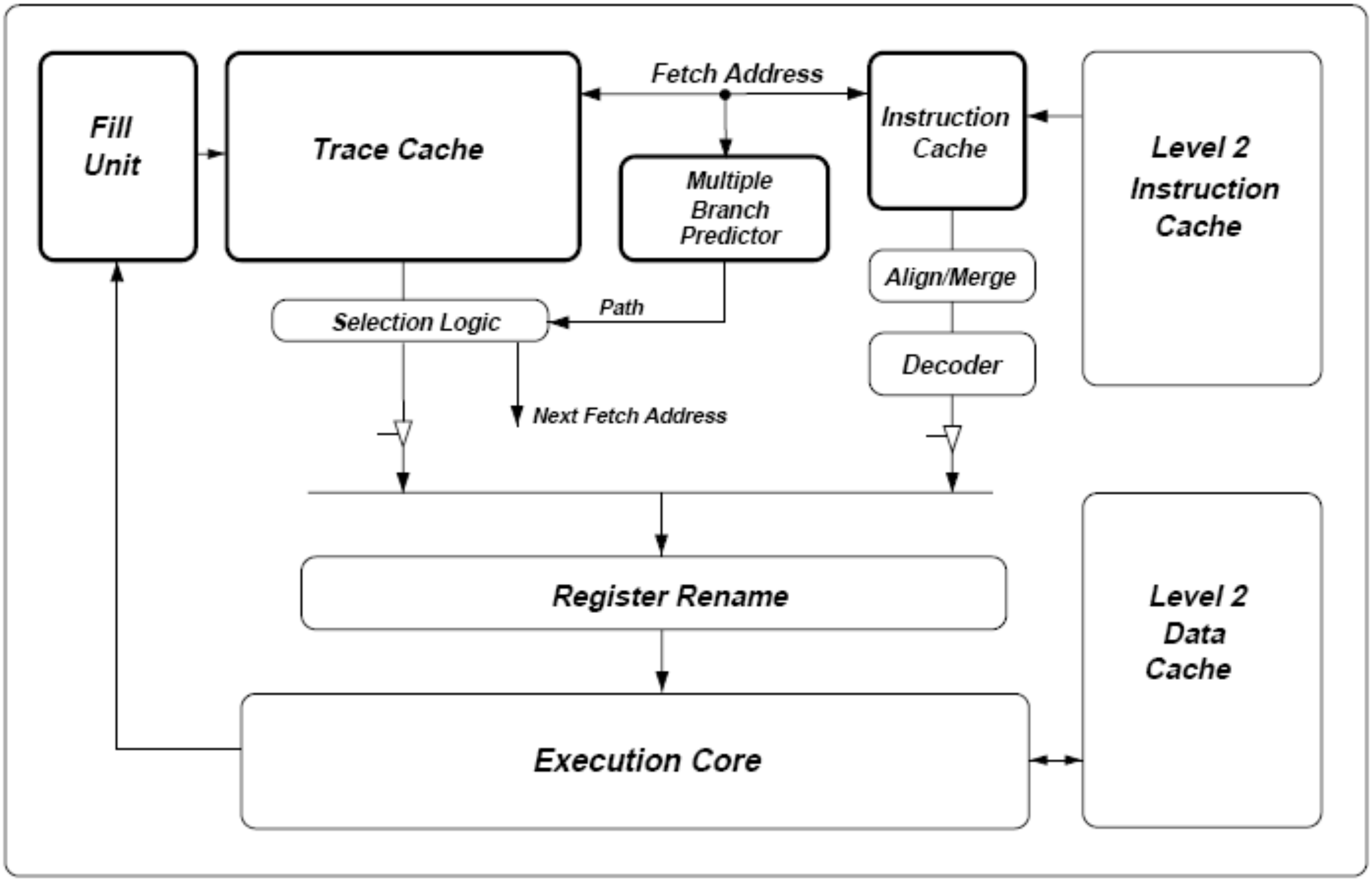
(a) Instruction cache.

(b) Trace cache.

Trace Cache: Example

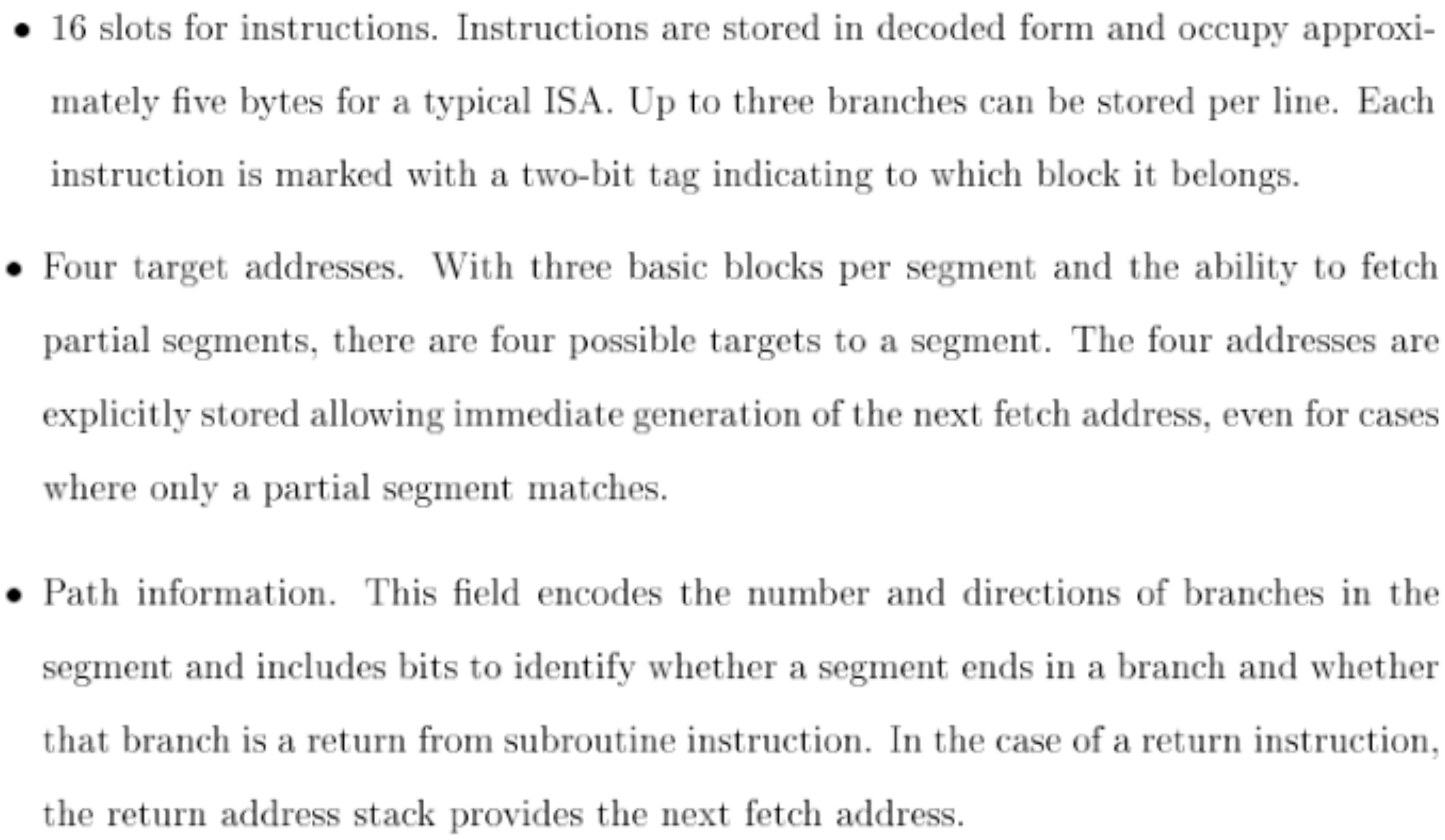


An Example Trace Cache Based Processor



- From Patel's PhD Thesis: "Trace Cache Design for Wide Issue Superscalar Processors," University of Michigan, 1999.

What Does A Trace Cache Line Store?



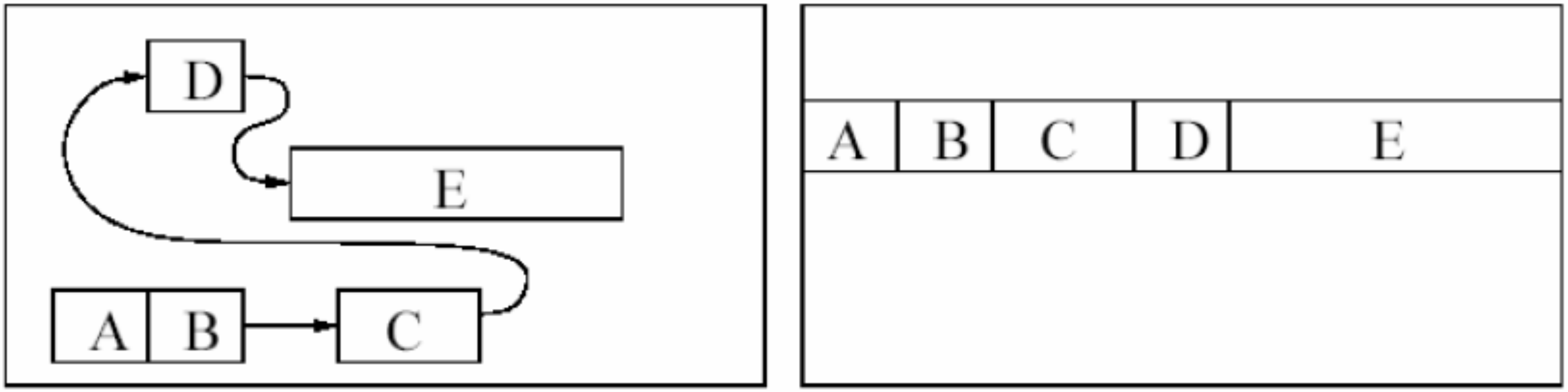
- 16 slots for instructions. Instructions are stored in decoded form and occupy approximately five bytes for a typical ISA. Up to three branches can be stored per line. Each instruction is marked with a two-bit tag indicating to which block it belongs.

- Four target addresses. With three basic blocks per segment and the ability to fetch partial segments, there are four possible targets to a segment. The four addresses are explicitly stored allowing immediate generation of the next fetch address, even for cases where only a partial segment matches.

- Path information. This field encodes the number and directions of branches in the segment and includes bits to identify whether a segment ends in a branch and whether that branch is a return from subroutine instruction. In the case of a return instruction, the return address stack provides the next fetch address.

- Patel et al., "Critical Issues Regarding the Trace Cache Fetch Mechanism," Umich TR, 1997.

Trace Cache: Advantages/Disadvantages



(a) Instruction cache.

(b) Trace cache.

+ Reduces fetch breaks (assuming branches are biased)

+ No need for decoding (instructions can be stored in decoded form)

+ Can enable dynamic optimizations within a trace

-- Requires hardware to form traces (more complexity) → called fill unit

-- Results in duplication of the same basic blocks in the cache

-- Can require the prediction of multiple branches per cycle

  -- If multiple cached traces have the same start address

  -- What if XYZ and XYT are both likely traces?

Trace Cache Design Issues (I)

- Granularity of prediction: Trace based versus branch based?

  + Trace based eliminates the need for multiple predictions/cycle

  -- Trace based can be less accurate

  -- Trace based: How do you distinguish traces with the same start address?

- When to form traces: Based on fetched or retired blocks?

  + Retired: Likely to be more accurate

  -- Retired: Formation of trace is delayed until blocks are committed

    -- Very tight loops with short trip count might not benefit

- When to terminate the formation of a trace
  - After N instructions, after B branches, at an indirect jump or return

Trace Cache Design Issues (II)

- Should entire "path" match for a trace cache hit?
- Partial matching: A piece of a trace is supplied based on branch prediction

+ Increases hit rate when there is not a full path match

-- Lengthens critical path (next fetch address dependent on the match)

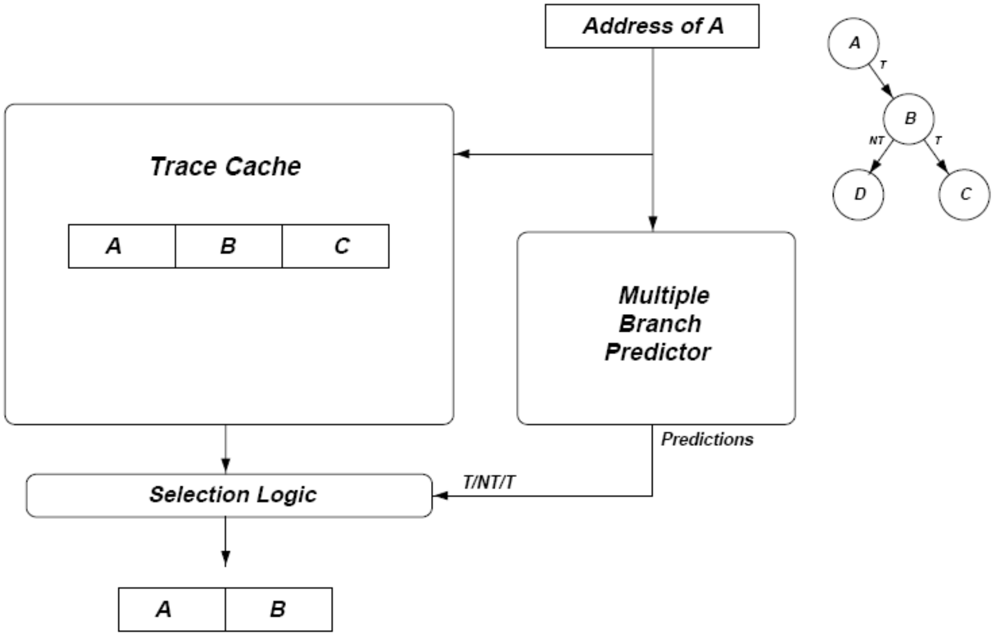
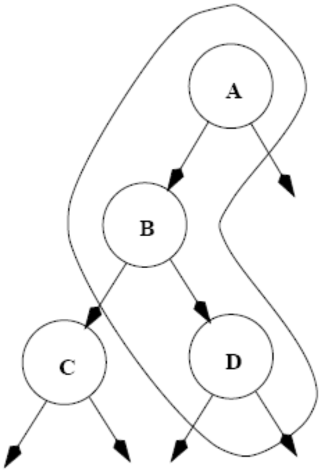


Figure 6.1: The trace cache and branch predictor are indexed with the address of block A. The inset figure shows the control flow from block A. The predictor selects the sequence ABD. The trace cache only contains ABC. AB is supplied.



Predicted path: ABC
Fetched segment: ABD

No partial matching: miss
Partial Matching: AB

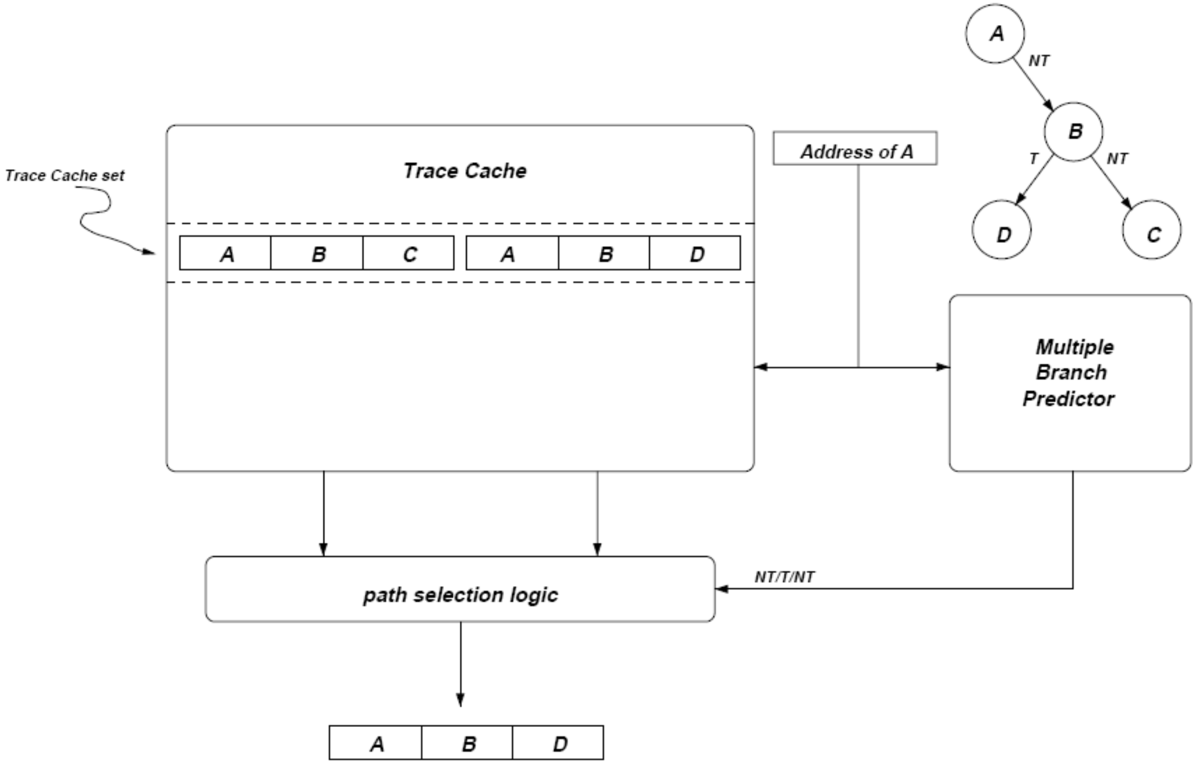
Trace Cache Design Issues (III)

- Path associativity: Multiple traces starting at the same address can be present in the cache at the same time.

+ Good for traces with unbiased branches (e.g., ping pong between C and D)

-- Need to determine longest matching path

-- Increased cache pressure


A
NT
B
T
NT
D
C
Trace Cache set
Trace Cache
A | B | C | A | B | D
Address of A
Multiple Branch Predictor
path selection logic
NT/T/NT
A | B | D