15-740/18-740 Computer Architecture Lecture 22: Superscalar Processing (II)

Prof. Onur Mutlu
Carnegie Mellon University

Announcements

- Project Milestone 2
    - Due Today
- Homework 4
    - Out today
    - Due November 15
- Midterm II
    - November 22
- Project Poster Session
    - December 9 or 10

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.

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)

Last Lecture

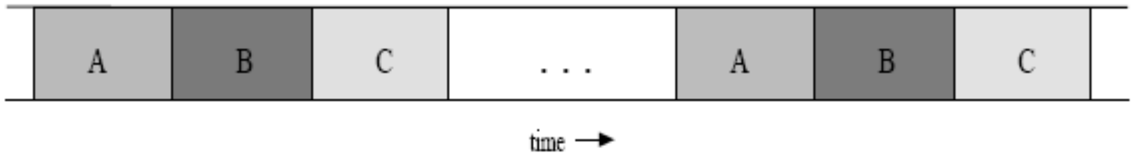
- Superscalar processing
  - Fetch issues: alignment and fetch breaks
  - Solutions to fetching N instructions at a time
    - Split line fetch
    - Basic block reordering
    - Superblock
    - Trace cache

Techniques to Reduce Fetch Breaks

- Compiler
  - Code reordering (basic block reordering)
  - Superblock
- Hardware
  - Trace cache
- Hardware/software cooperative
  - Block structured ISA

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)

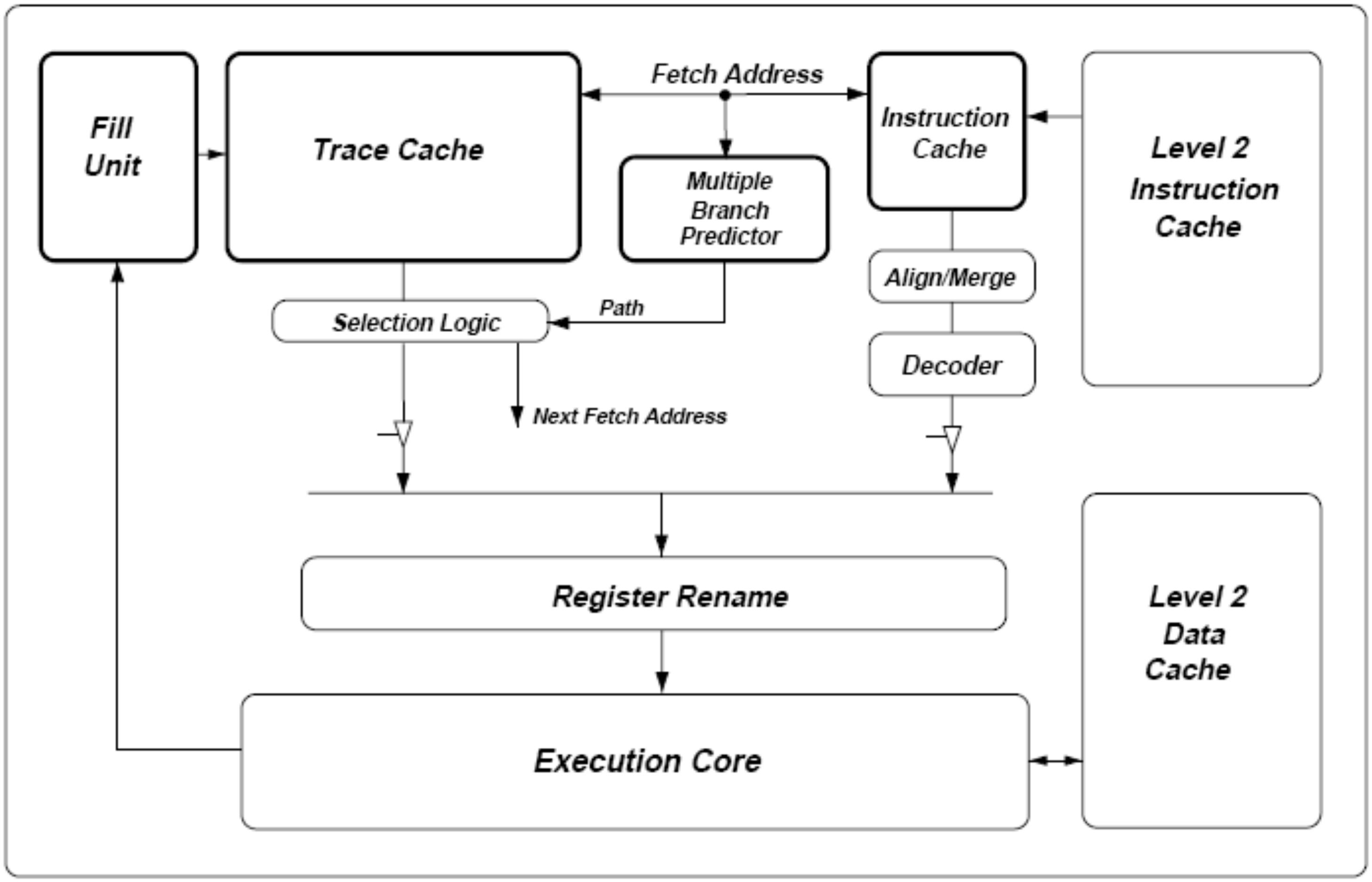


Dynamic Instruction Stream

- 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.

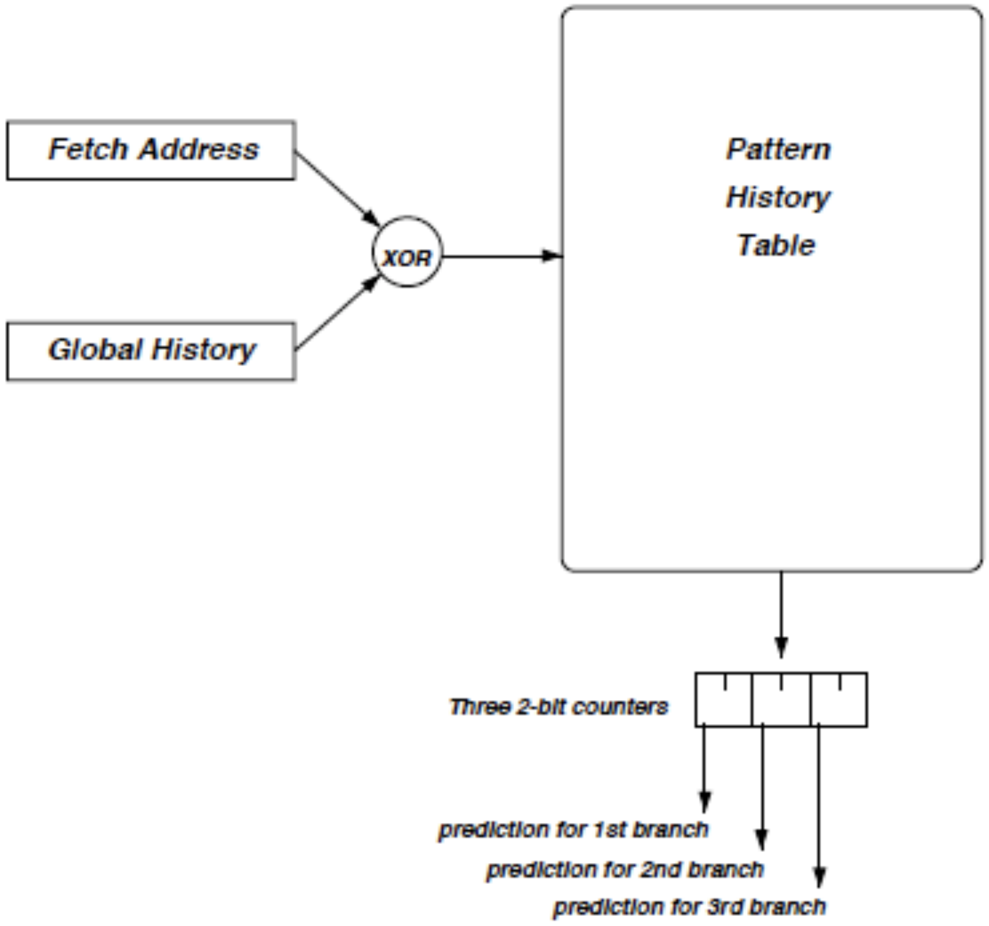
An Example Trace Cache Based Processor



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

Multiple Branch Predictor

- S. Patel, "Trace Cache Design for Wide Issue Superscalar Processors," PhD Thesis, University of Michigan, 1999.


Fetch Address
Global History
XOR
Pattern History Table
Three 2-bit counters
prediction for 1st branch
prediction for 2nd branch
prediction for 3rd branch

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)

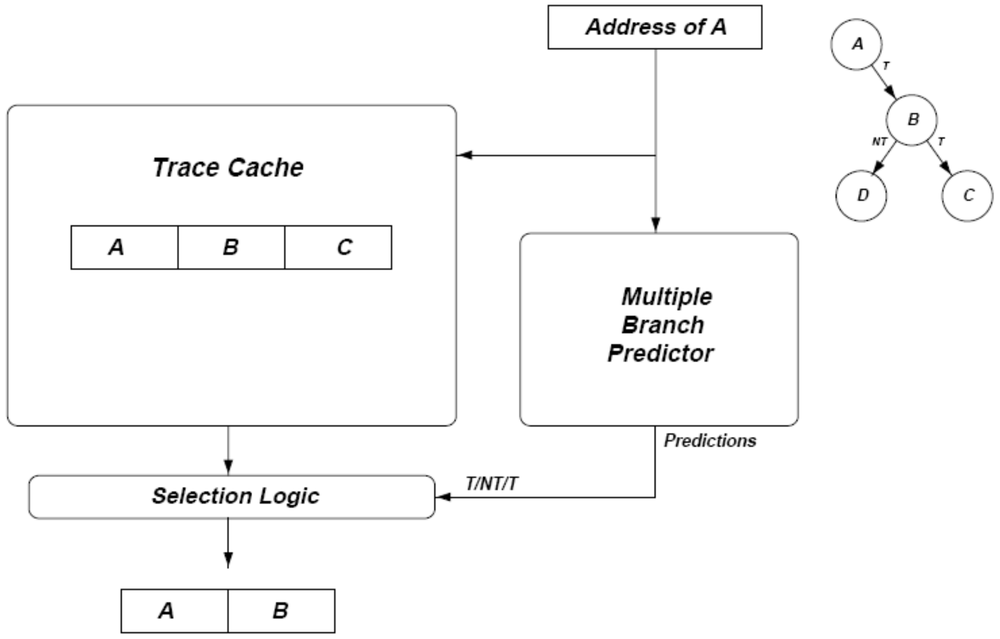
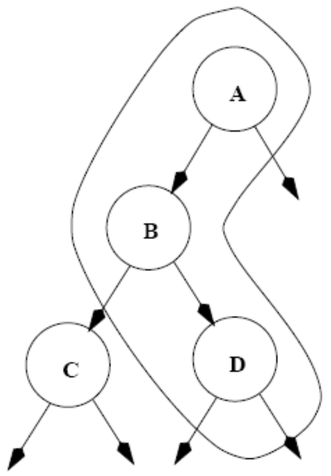


Figure 6.1: The trace cache and branch predictor are indexed with the address of block A. The inset figure shows the control flow from block A. The predictor selects the sequence ABD. The trace cache only contains ABC. AB is supplied.



Predicted path: ABC
Fetched segment: ABD

No partial matching: miss
Partial Matching: AB

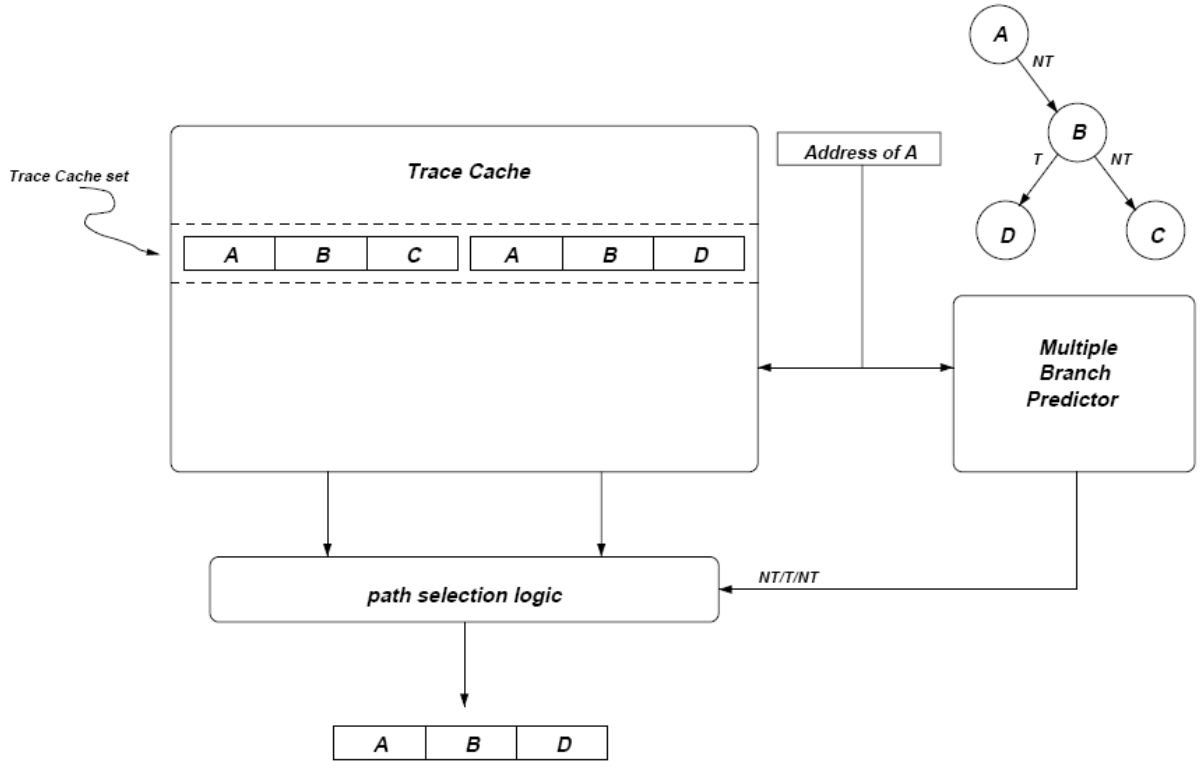
Trace Cache Design Issues (III)

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

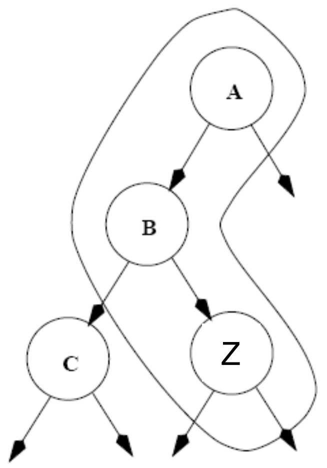
+ Good for traces with unbiased branches (e.g., ping pong between C and D)

-- Need to determine longest matching path

-- Increased cache pressure



Trace Cache Design Issues (IV)



Predicted path: ABC
Fetched segment: ABI Z

No partial matching: miss
Partial matching: AB
Inactive Issue: AB (active) Z ıctive)

- Inactive issue: All blocks within a trace cache line are issued even if they do not match the predicted path

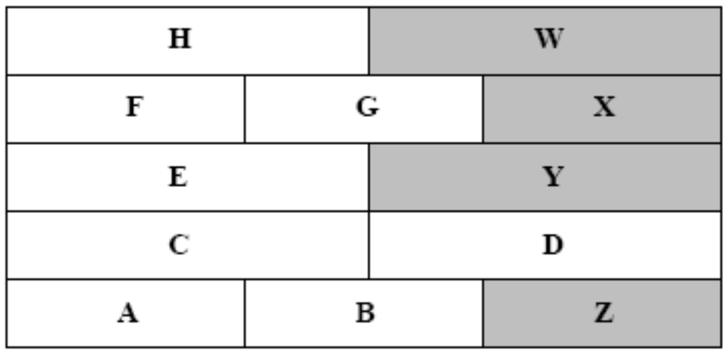
+ Reduces impact of branch mispredictions

+ Reduces basic block duplication in trace cache

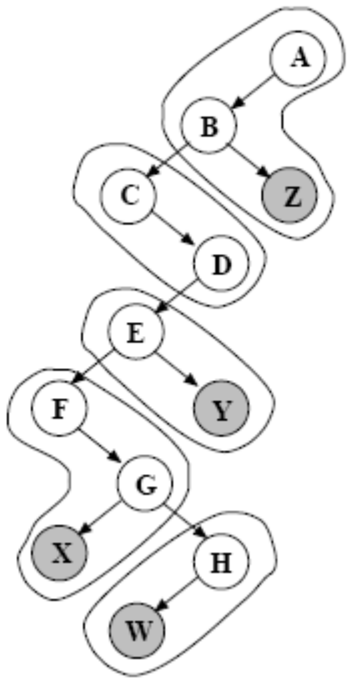
-- Slightly more complex scheduling/branch resolution

-- Some instructions not dispatched/flushed

Instruction Window



H		W
F	G	X
E		Y
C		D
A	B	Z



Trace Cache Design Issues (V)

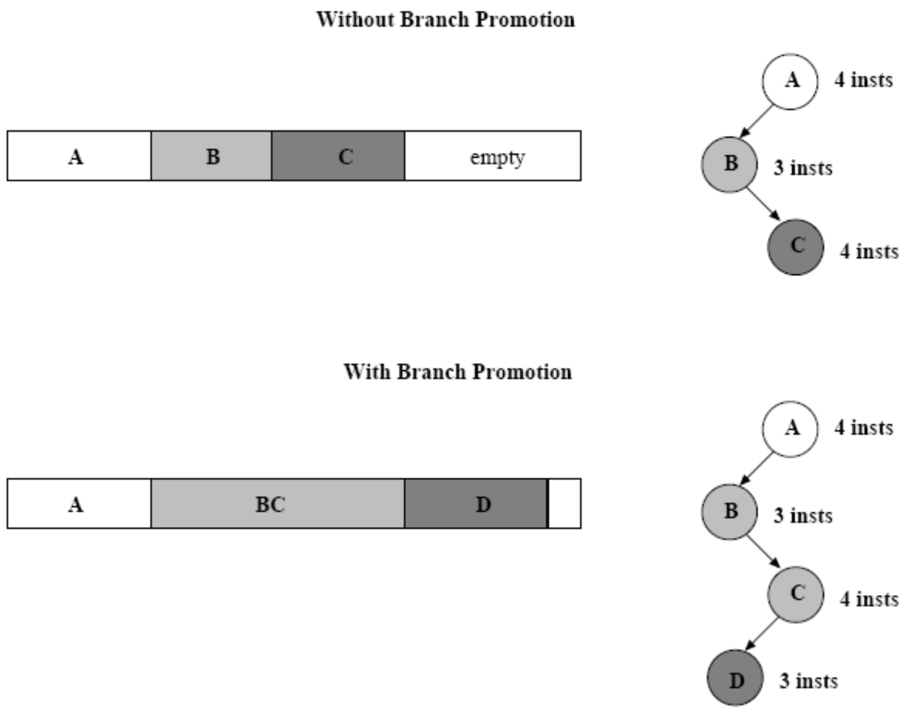
- Branch promotion: promote highly-biased branches to branches with static prediction

  + Larger traces

  + No need for consuming branch predictor BW

  + Can enable optimizations within trace

  -- Requires hardware to determine highly-biased branches



How to Determine Biased Branches

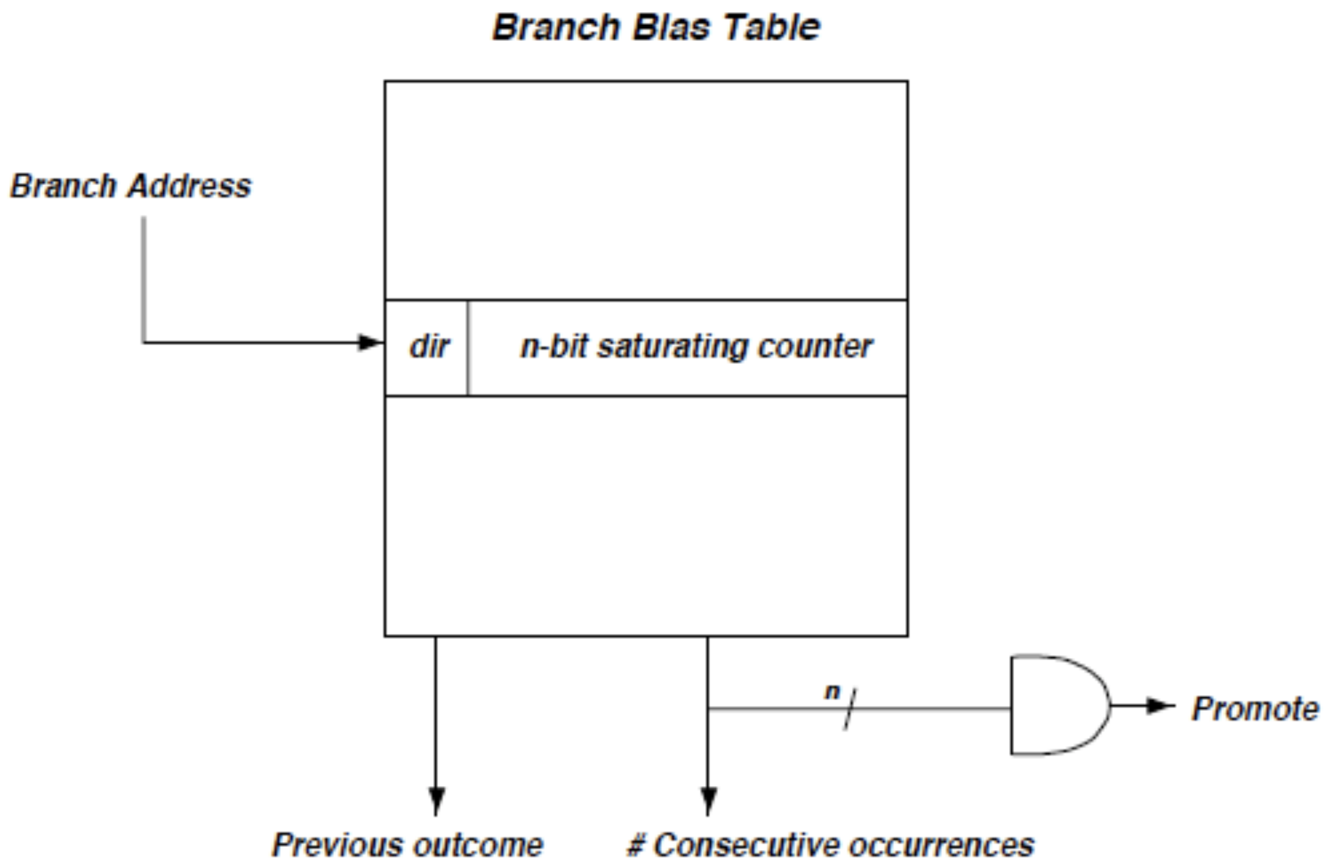
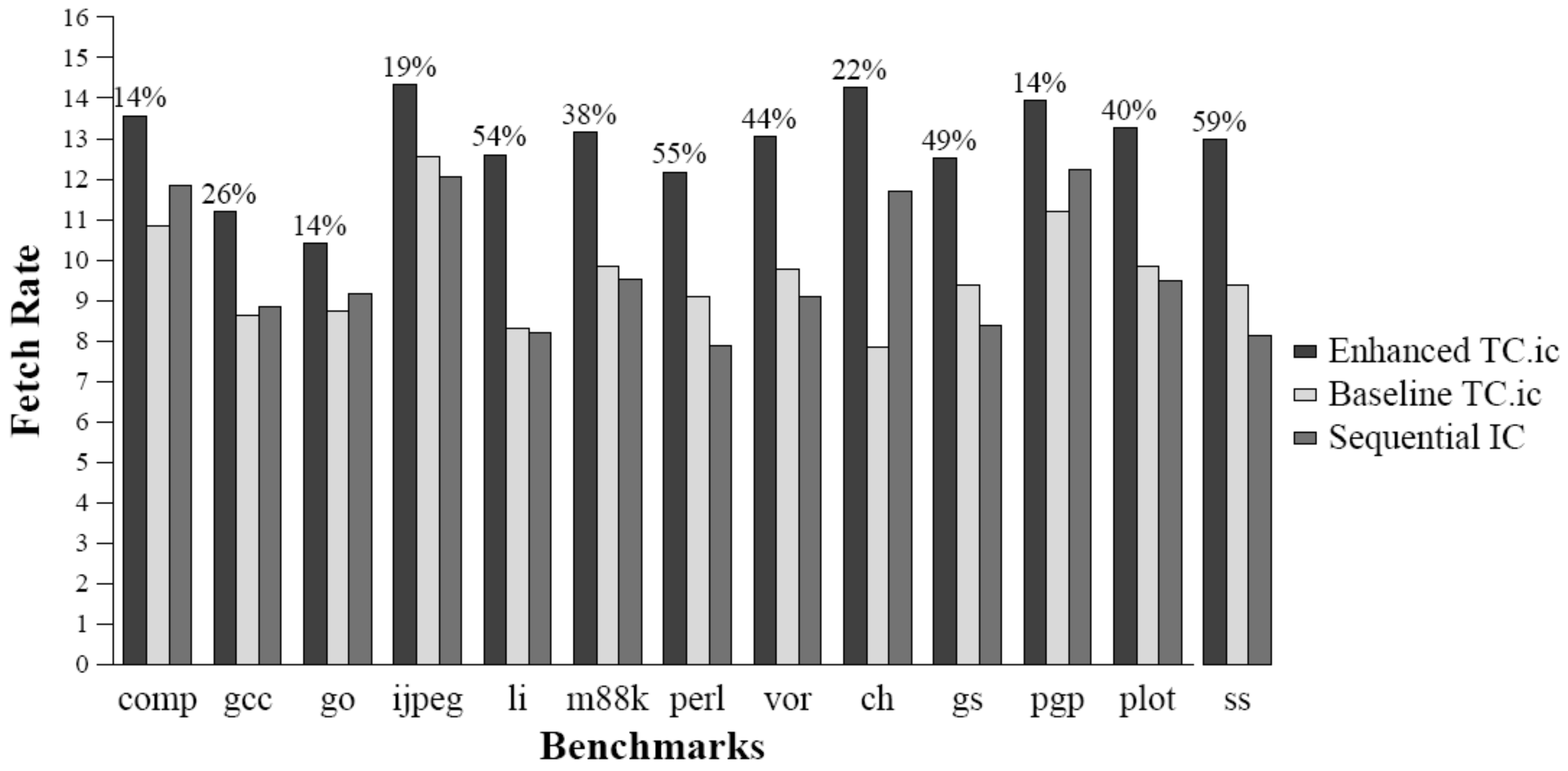
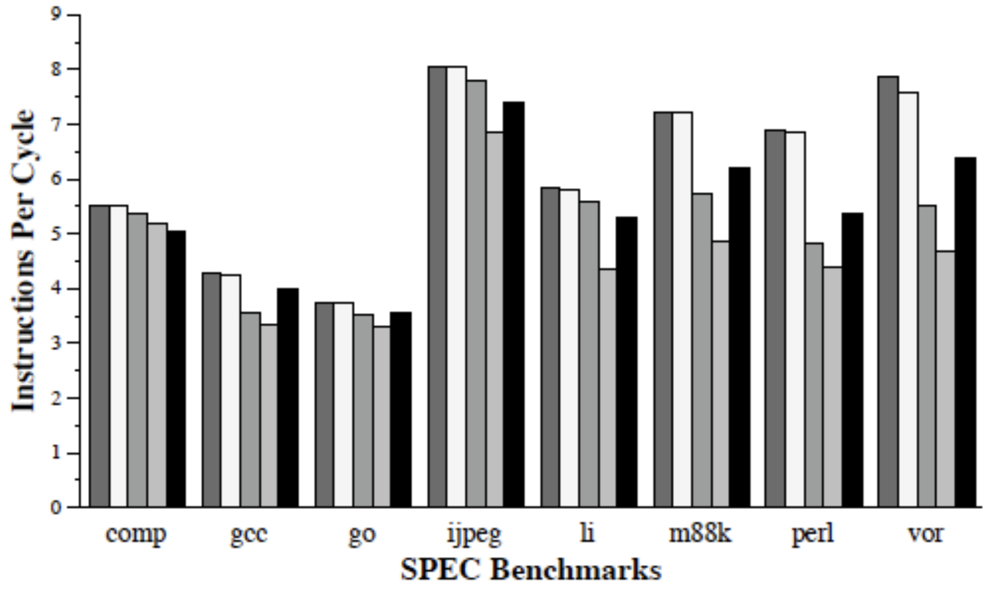


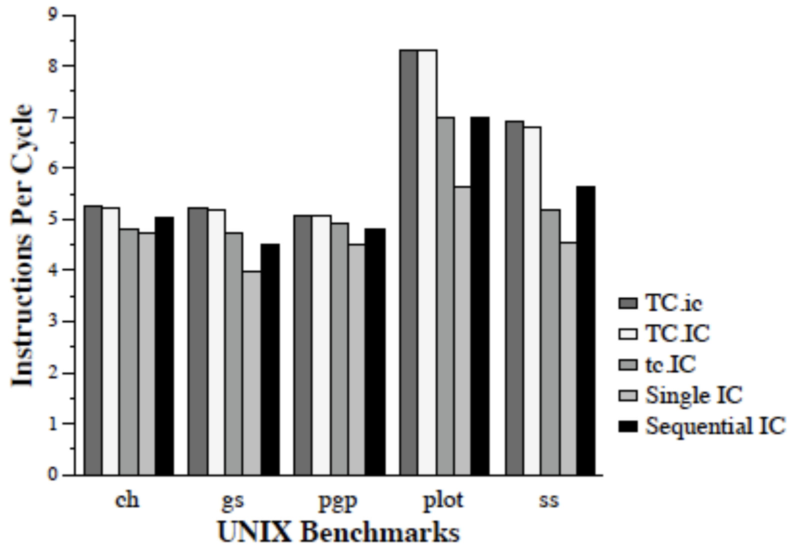
Figure 6.19: Diagram of the branch bias table.

Effect on Fetch Rate



Effect on IPC (16-wide superscalar)





Configuration Name	TCache Size	ICache Size	Blocks per Fetch	Br Pred Type	BTB Size
TC.ic	128KB	4KB	3	Multiple	1KB
TC.IC	64KB	64KB	3	Multiple	8KB
tc.IC	4KB	128KB	3	Multiple	16KB
Single	–	128KB	1	Hybrid	20KB
Sequential	–	128KB	3	Multiple	16KB

- ~15% IPC increase over “sequential I-cache” that breaks fetch on a predicted-taken branch

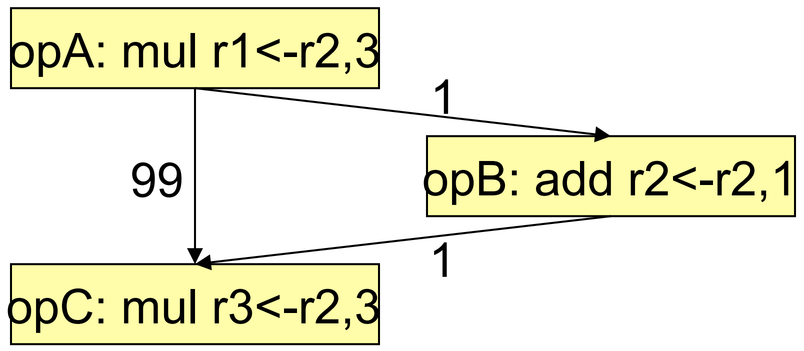
Fill Unit Optimizations

- Fill unit constructs traces out of decoded instructions
- Can perform optimizations across basic blocks
  - Branch promotion: promote highly-biased branches to branches with static prediction
  - Can treat the whole trace as an atomic execution unit
    - All or none of the trace is retired (based on branch directions in trace)
    - Enables many optimizations across blocks
  - Dead code elimination
  - Instruction reordering
  - Reassociation

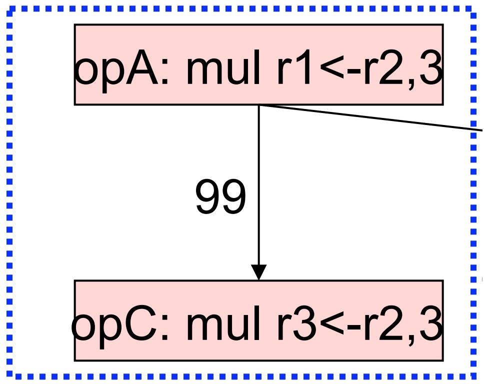
$$\begin{array}{l} ADDI\ Rx \leftarrow Ry + 4 \\ ADDI\ Rz \leftarrow Rx + 4 \end{array} \longrightarrow \begin{array}{l} ADDI\ Rx \leftarrow Ry + 4 \\ ADDI\ Rz \leftarrow Ry + 8 \end{array}$$

- Friendly et al., "Putting the Fill Unit to Work: Dynamic Optimizations for Trace Cache Microprocessors," MICRO 1998.

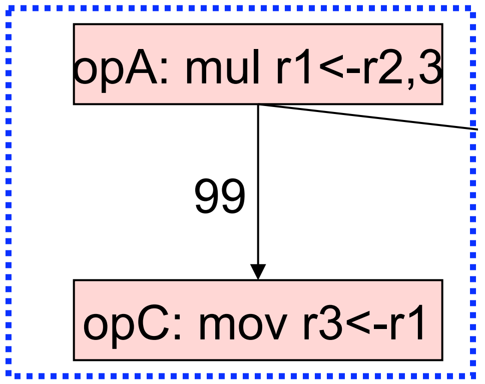
Remember This Optimization?



Original Code



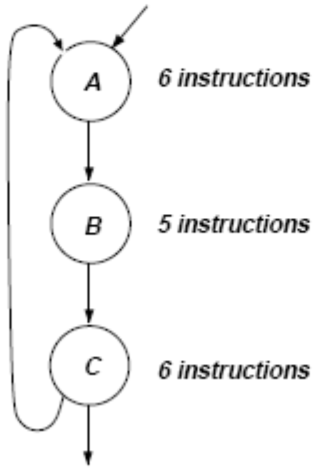
Part of Trace in Fill Unit



Optimized Trace

Redundancy in the Trace Cache

- ABC, BCA, CAB can all be in the trace cache
- Leads to contention and reduced hit rate



- One possible solution: Block based trace cache
- Idea: Decouple storage of basic blocks from their "names"
  - Store traces of pointers to basic blocks rather than traces of basic blocks themselves
  - Basic blocks stored in a separate "block table"

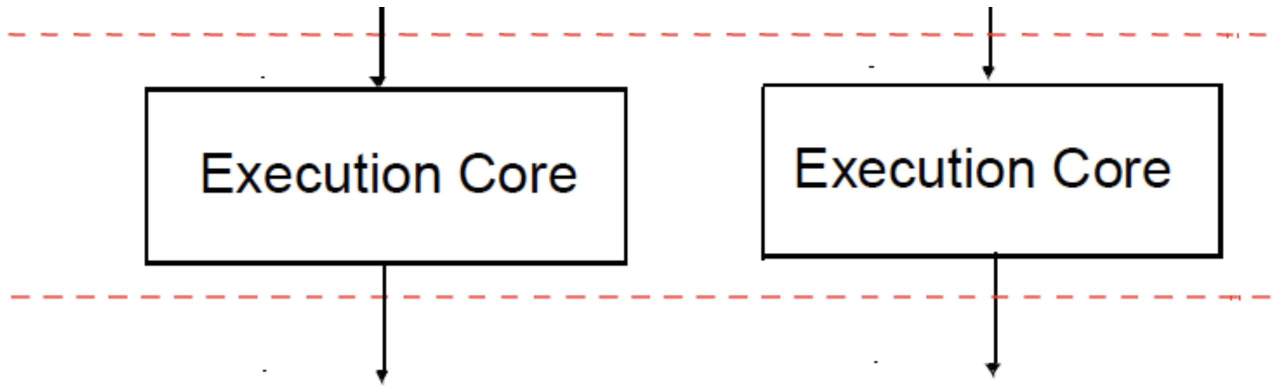
+ Reduces redundancy of basic blocks

-- Lengthens fetch cycle (indirection needed to access blocks)

-- Block table needs to be multiported to obtain multiple blocks per cycle

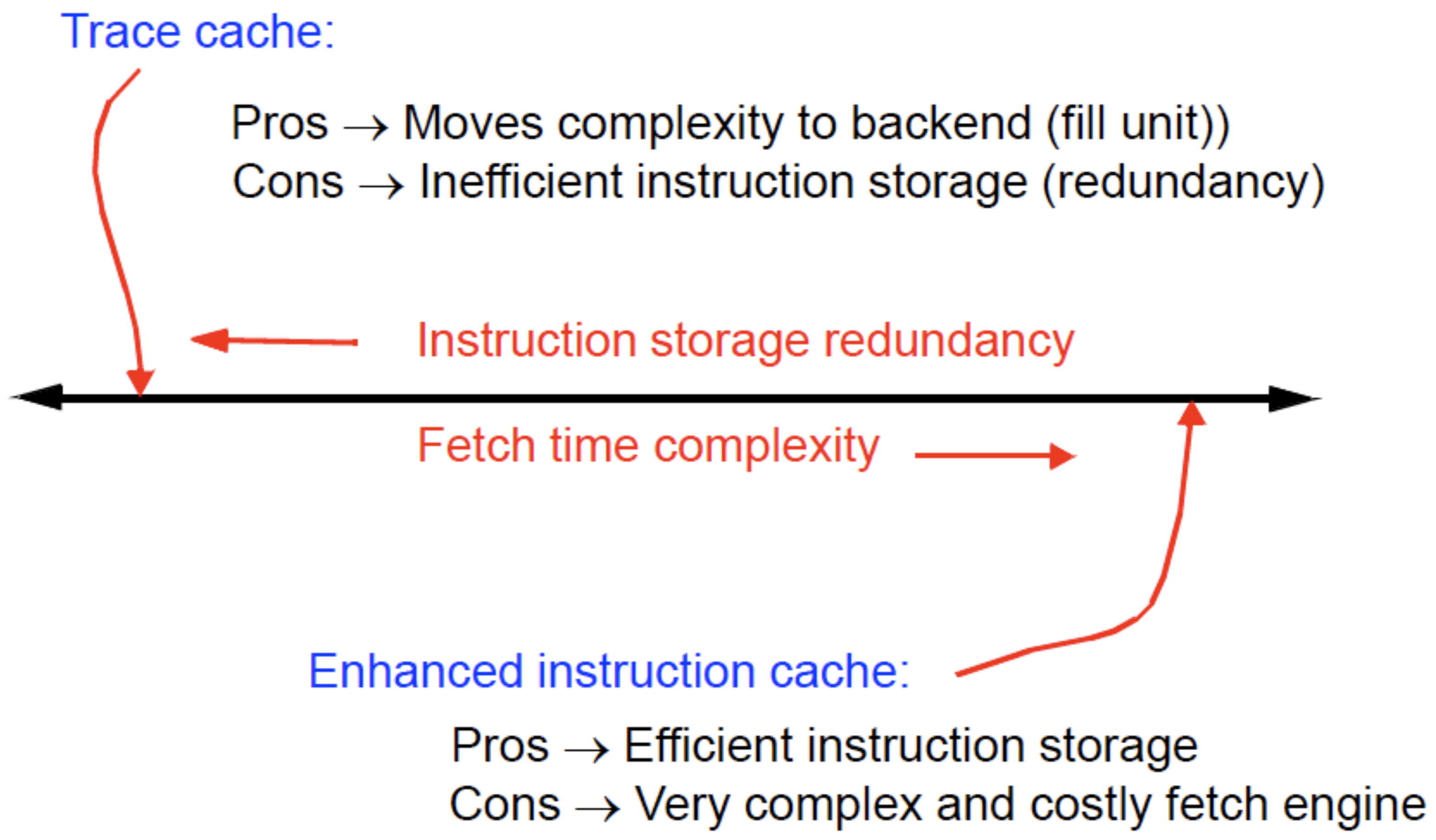
Enhanced I-Cache vs. Trace Cache (I)

	Enhanced Instruction Cache	Trace Cache
Fetch	1. Multiple-branch prediction 2. Instruction cache fetch from multiple blocks (N ports) 3. Instruction alignment & collapsing	1. Next trace prediction 2. Trace cache fetch



	Enhanced Instruction Cache	Trace Cache
Completion	1. Multiple-branch predictor update	1. Trace construction and fill 2. Trace predictor update

Enhanced I-Cache vs. Trace Cache (II)


Trace cache:
Pros → Moves complexity to backend (fill unit))
Cons → Inefficient instruction storage (redundancy)
Instruction storage redundancy
Fetch time complexity
Enhanced instruction cache:
Pros → Efficient instruction storage
Cons → Very complex and costly fetch engine

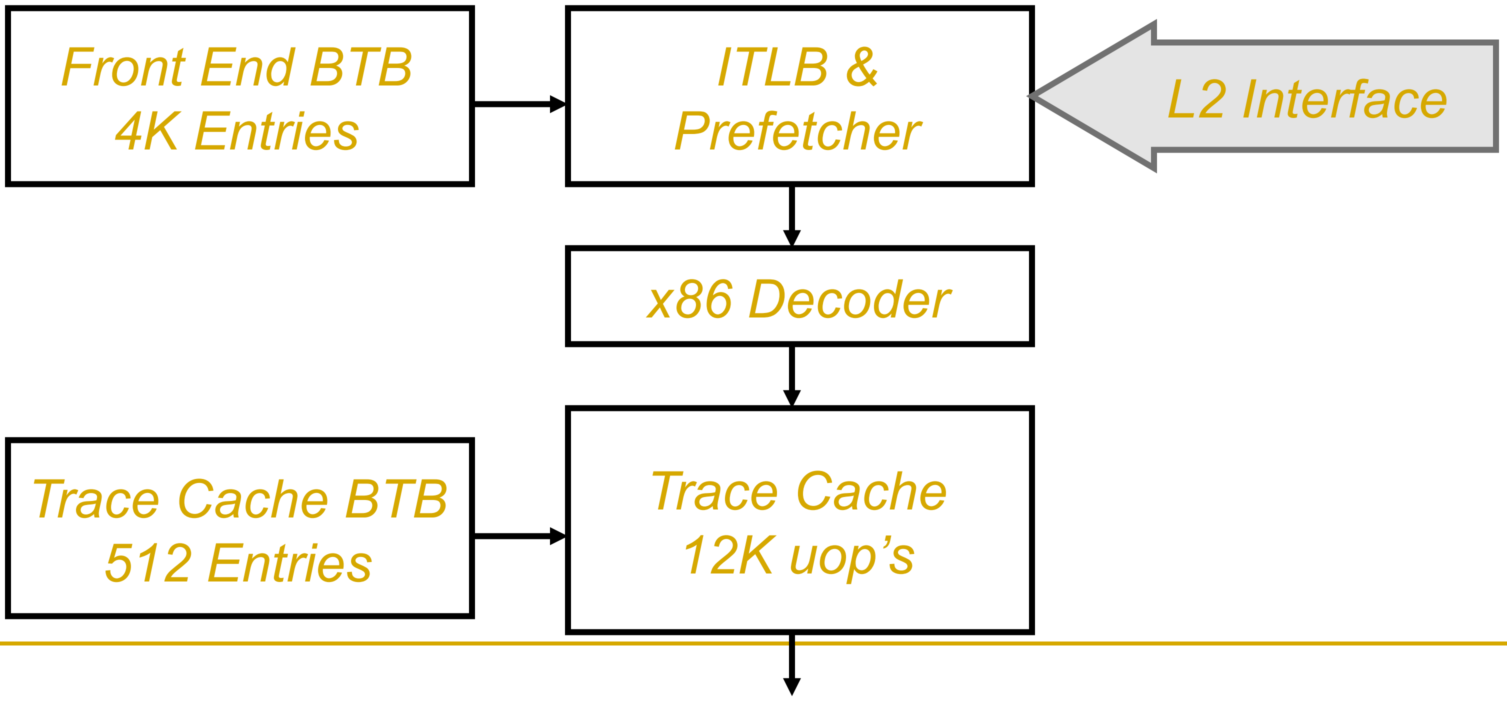
Frontend vs. Backend Complexity

- Backend is not on the critical path of instruction execution
  - Easier to increase its latency without affecting performance

- Frontend is on the critical path
  - Increased latency fetch directly increases
    - Branch misprediction penalty
  - Increased complexity can affect cycle time

Pentium 4 Trace Cache

- A 12K-uop trace cache replaces the L1 I-cache
- Trace cache stores decoded and cracked instructions
  - Micro-operations (uops): returns 6 uops every other cycle
- x86 decoder can be simpler and slower
- A. Peleg, U. Weiser; "Dynamic Flow Instruction Cache Memory Organized Around Trace Segments Independent of Virtual Address Line", United States Patent No. 5,381,533, Jan 10, 1995

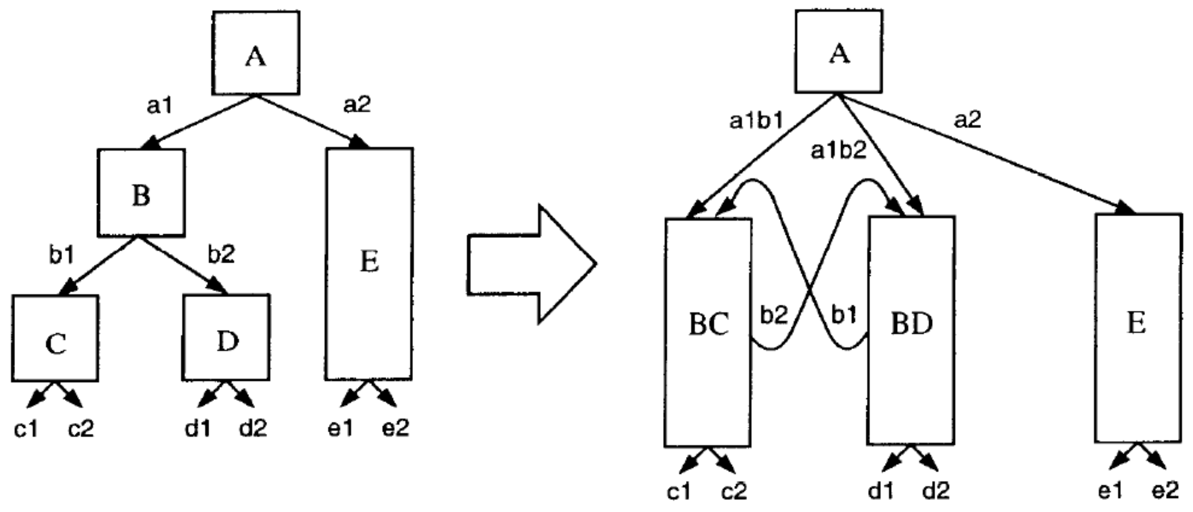


Techniques to Reduce Fetch Breaks

- Compiler
  - Code reordering (basic block reordering)
  - Superblock
- Hardware
  - Trace cache
- Hardware/software cooperative
  - Block structured ISA

Block Structured ISA

- Blocks (> instructions) are atomic (all-or-none) operations
  - Either all of the block is committed or none of it
- Compiler enlarges blocks by combining basic blocks with their control flow successors
  - Branches within the enlarged block converted to "fault" operations → if the fault operation evaluates to true, the block is discarded and the target of fault is fetched



Block Structured ISA (II)

- Advantages:

  + Larger blocks → larger units can be fetched from I-cache

  + Aggressive compiler optimizations (e.g. reordering) can be enabled within atomic blocks

  + Can explicitly represent dependencies among operations within an enlarged block

- Disadvantages:

  -- "Fault operations" can lead to work to be wasted (atomicity)

  -- Code bloat (multiple copies of the same basic block exists in the binary and possibly in I-cache)

  -- Need to predict which enlarged block comes next

- Optimizations
  - Within an enlarged block, the compiler can perform optimizations that cannot normally be performed across basic blocks

Block Structured ISA (III)

- Hao et al., "Increasing the instruction fetch rate via block-structured instruction set architectures," MICRO 1996.

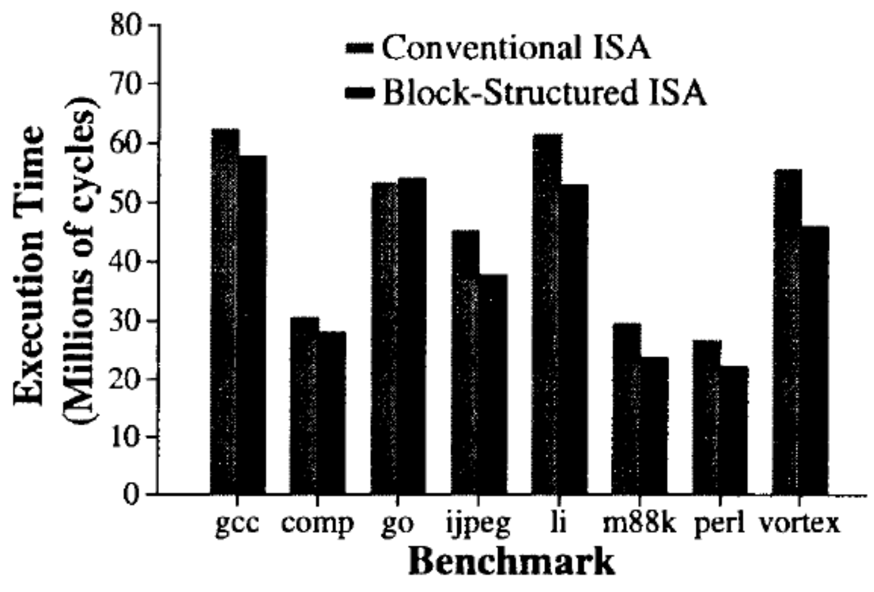


Figure 3. Performance comparison of block-structured ISA executables and conventional ISA executables.

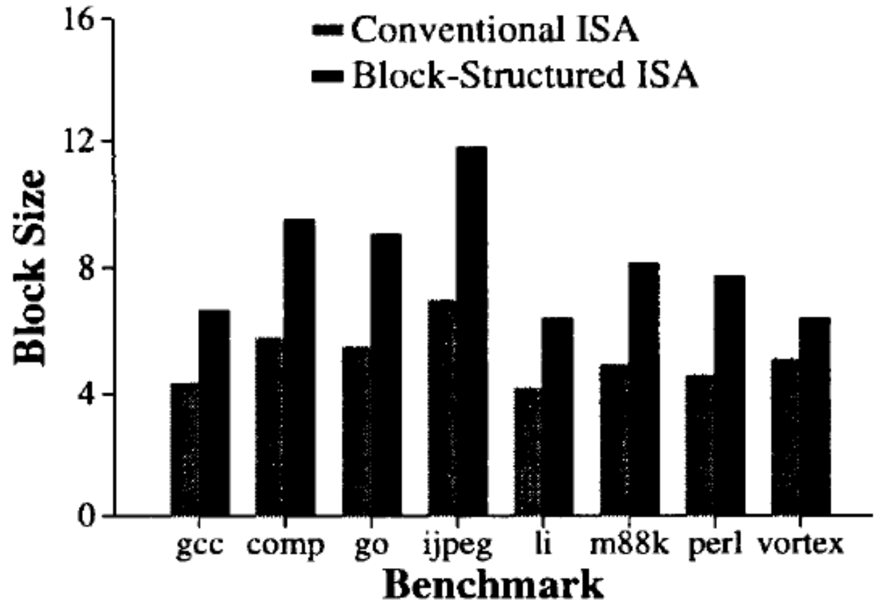


Figure 5. Average block sizes for block-structured and conventional ISA executables.

Superblock vs. BS-ISA

- Superblock
  - Single-entry, multiple exit code block
  - Not atomic
  - Compiler inserts fix-up code on superblock side exit
- BS-ISA blocks
  - Single-entry, single exit
  - Atomic

Superblock vs. BS-ISA

- Superblock

  + No ISA support needed

  -- Optimizes for only 1 frequently executed path

    -- Not good if dynamic path deviates from profiled path → missed opportunity to optimize another path

- Block Structured ISA

  + Enables optimization of multiple paths and their dynamic selection.

  + Dynamic prediction to choose the next enlarged block. Can dynamically adapt to changes in frequently executed paths at run-time

  + Atomicity can enable more aggressive code optimization

  -- Code bloat becomes severe as more blocks are combined

  -- Requires "next enlarged block" prediction, ISA+HW support

  -- More wasted work on "fault" due to atomicity requirement

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)

Decoding Multiple Instructions Per Cycle

- Fixed length instructions
  - Relatively easy: each instruction can be decoded independently

- Variable length instructions
  - Instruction boundaries not known before decode
    - Later instructions' decode dependent on earlier ones in the same cycle

  -- Increases decoder latency

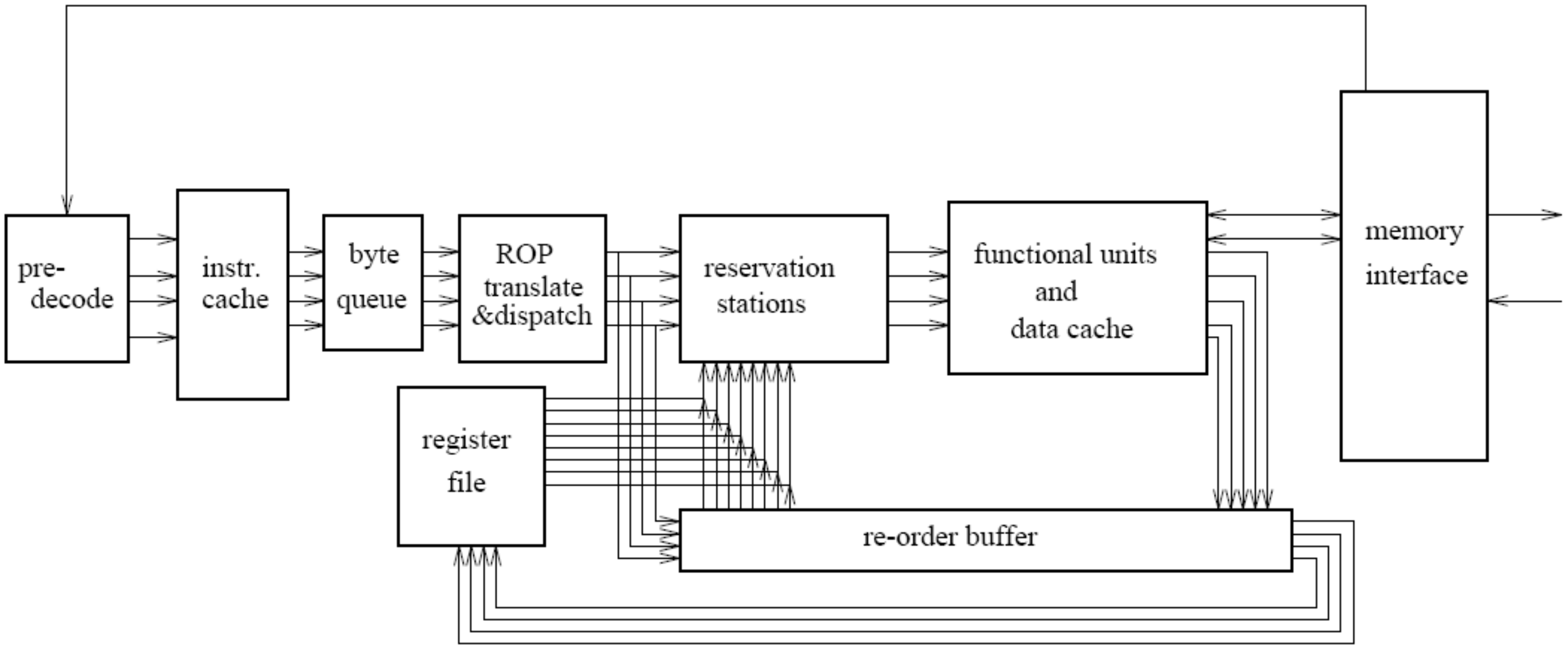
- Two techniques ease decoding (especially variable-length instructions)
  - Pre-decoding
  - Decoded I-cache (or trace cache)

Pre-decoding

- Instruction pre-decoding:
  - Store information on instruction boundaries in the I-cache
    - Before inserting instruction into I-cache, pre-decode
  - Mark start/end bytes of instructions
  - This information used to convey the correct instruction bytes to the parallel decoders
  - Implemented in AMD K5

- What other pre-decode information can be useful?
  - Branch or not
  - Type of branch
  - Usually anything that 1) can ease decode 2) can reduce the latency of predicting the next fetch address (for the next cycle)

AMD K5 Pre-decode



Pre-decoded I-Cache

- Advantages:
  - + Simplifies variable length decode
  - + Could reduce decode pipeline depth
    - + Partial decoding done on cache fill
  - + Could reduce next fetch address calculation latency (can be the critical path in many designs)

- Disadvantages:
  - -- Increases I-cache fill latency
  - -- Reduces I-cache storage efficiency

Decode Cache

- Decode cache
  - Idea: Store decoded instructions in a separate cache
  - Access decode cache and I-cache in parallel or series
  - If decode cache miss, decode instructions (perhaps serially) and insert into decode cache
  - Next time, decode cache hit: no need to decode multiple instructions
    - Pentium 4 works similarly with its trace cache
      - Trace cache miss: Decode only 1 x86 instruction per cycle
      - Trace cache hit: Supplies 3 micro-instructions per cycle

+ Eases parallel decode of variable length instructions

+ Eliminates decoding from critical path (decode cache hit)

+ Can reduce energy consumption (less decoding)

-- Increases front-end complexity

CISC to RISC Translation (I)

- Complex instructions harder to implement in hardware
  - More costly to implement, require multiple resources (e.g., memory and functional units), dependency checks in OoO execution become more complex
- Simple instructions easier to pipeline, implement, and optimize for
- Can we get the "simplicity" benefits of a simple ISA while executing a complex one?

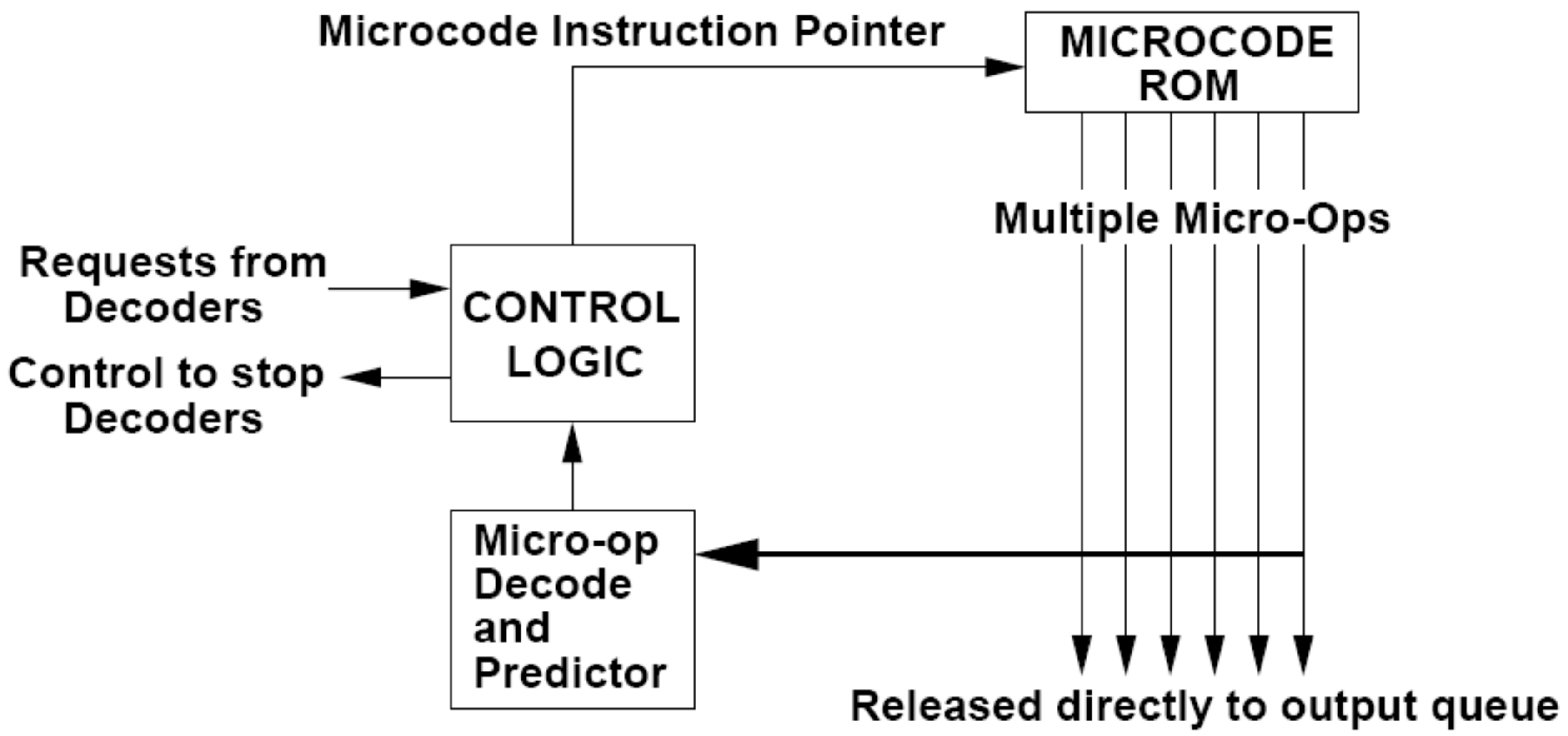
- Idea: Decoder dynamically translates complex instructions into simple ones
  - Called instruction cracking into micro-operations (uops)
    - Uops not visible to software

Complex to Simple Translation (II)

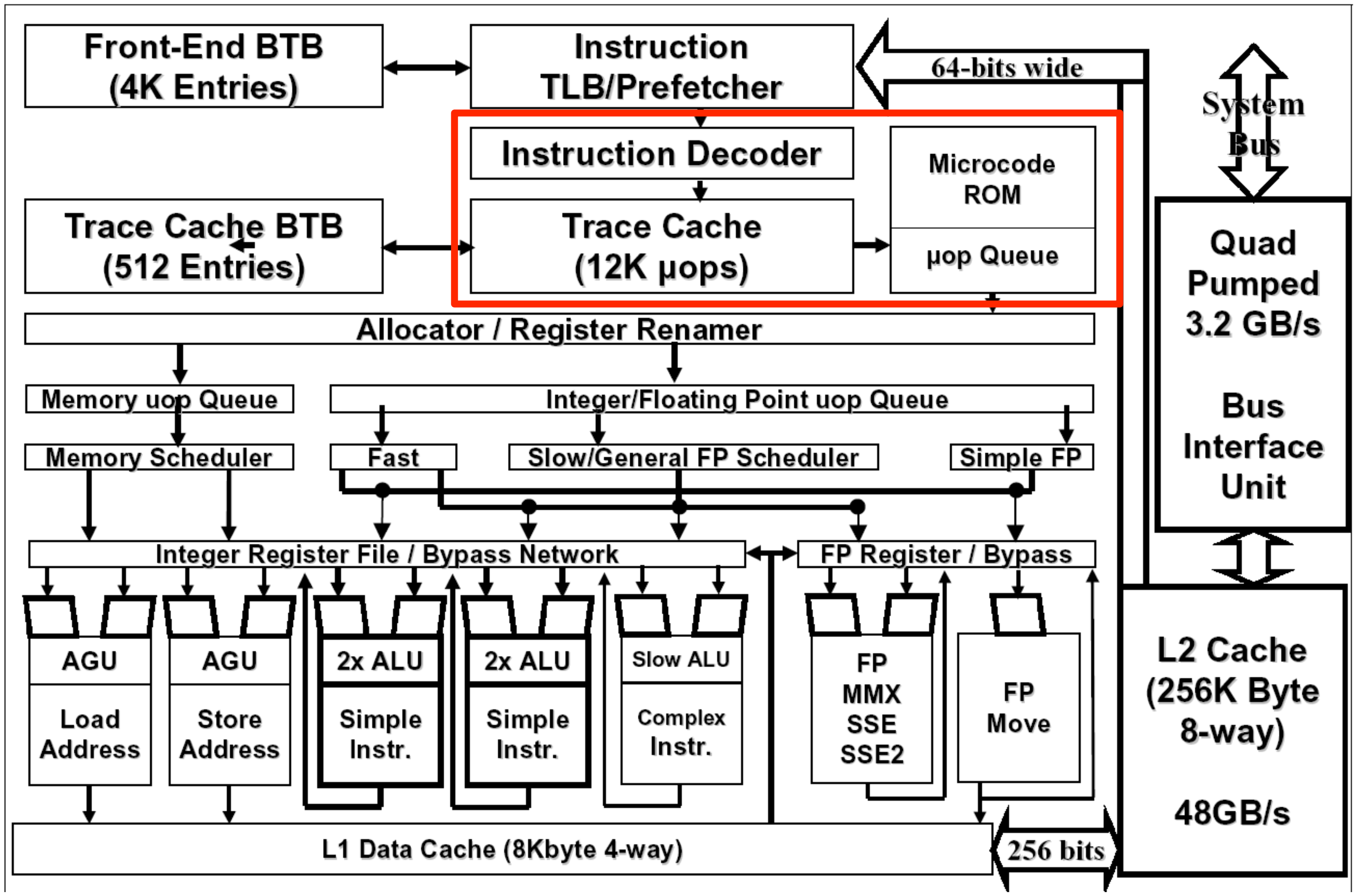
- Two methods for cracking
  - Hardware combinational logic
    - Decoder translates a complex instruction into multiple simple ones
  - Microsequencer and microcode ROM
    - Microsequencer sequences through simple instructions stored in Microcode ROM

- Pentium 4 employs both
  - A complex instruction that requires >4 uops is translated by the Microcode ROM (e.g., REP MOVS)
  - A complex instruction <= 4 uops ins inserted into the trace cache (cracked and decoded) after the x86 decoder handles it

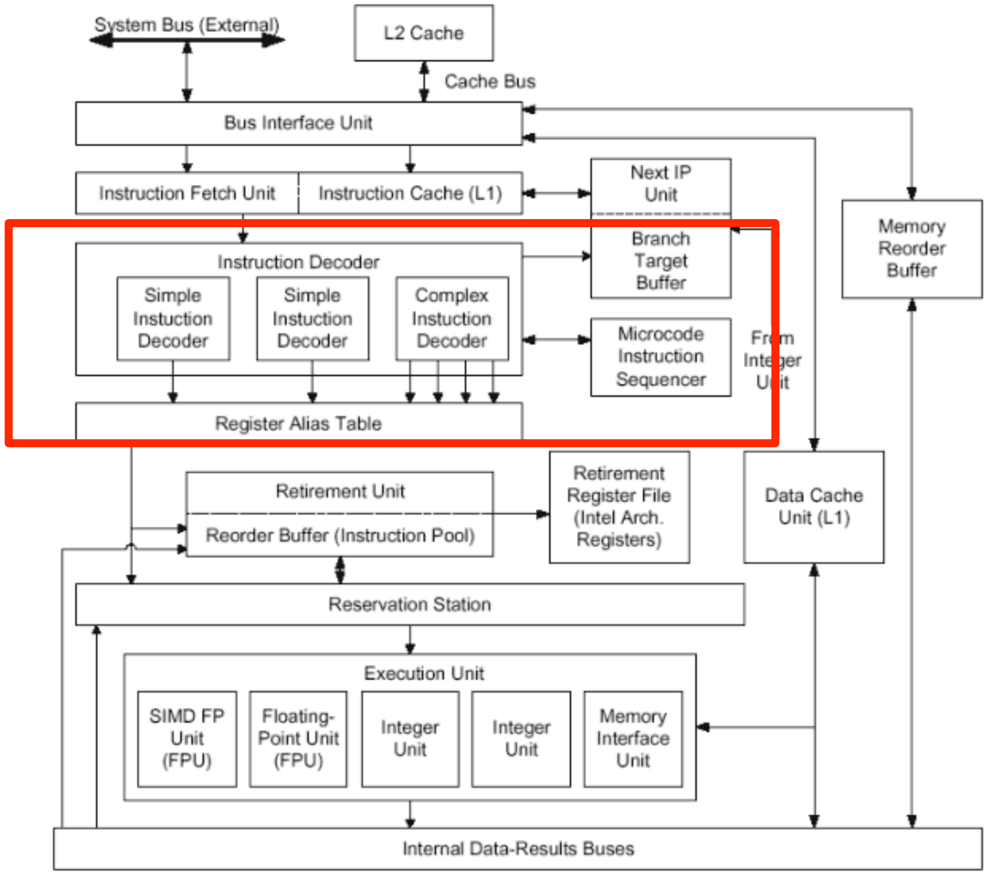
Microsequencing


Microcode Instruction Pointer
MICROCODE ROM
Multiple Micro-Ops
Requests from Decoders
CONTROL LOGIC
Control to stop Decoders
Micro-op Decode and Predictor
Released directly to output queue

Pentium 4 Decoders



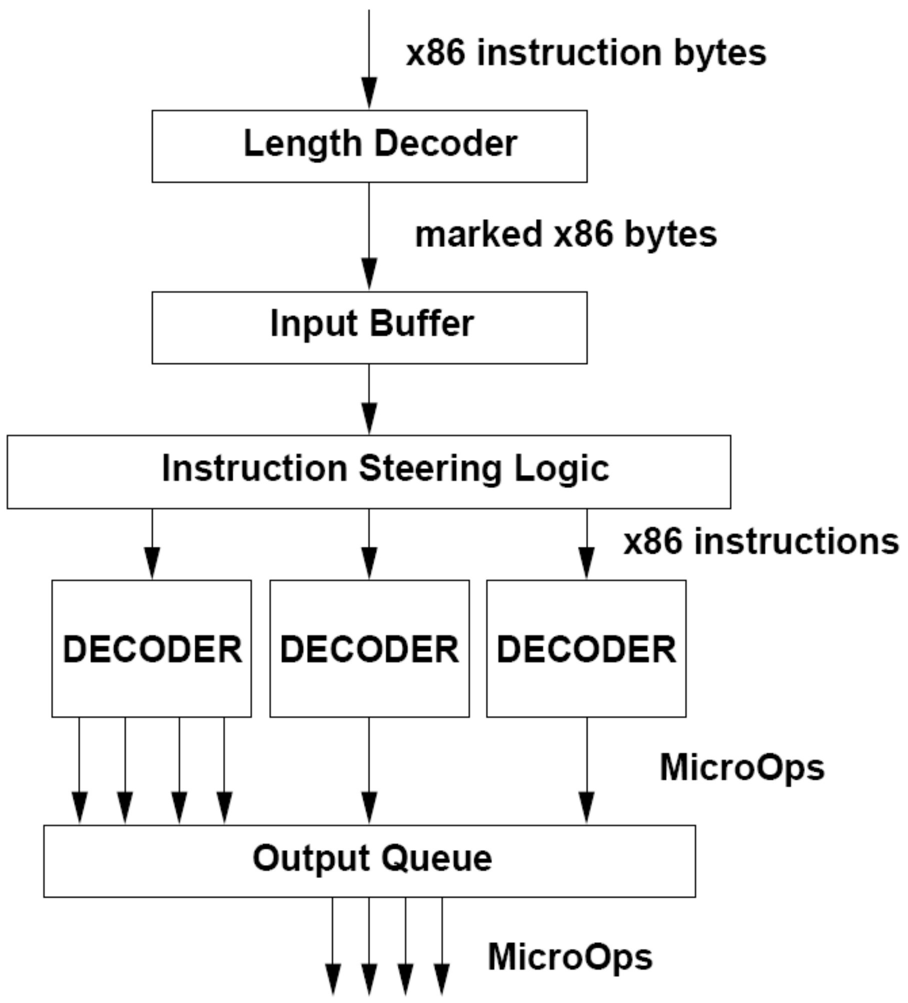
Pentium Pro Decoders (I)



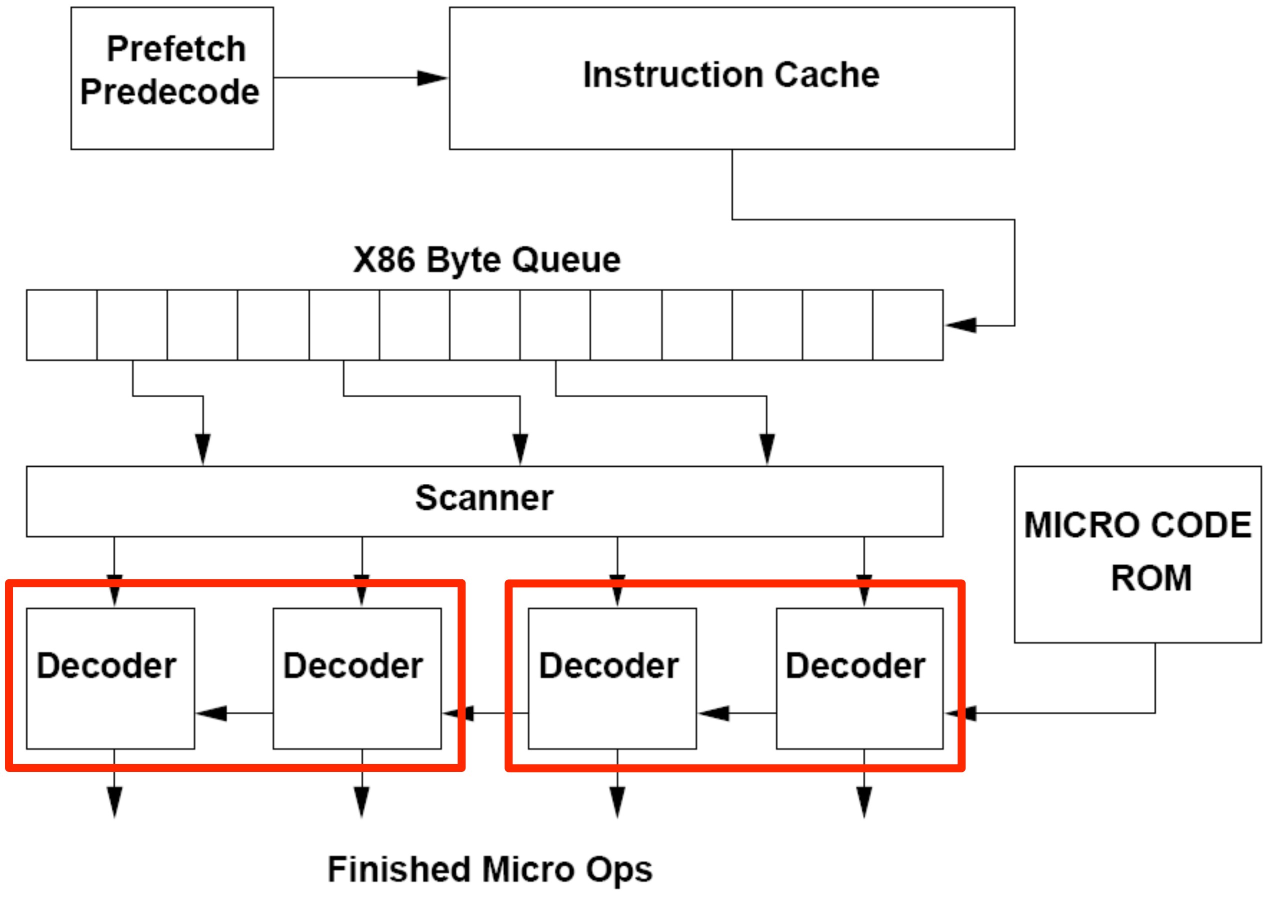
1	2	3	4	5	6	7	8	9	10
Fetch	Fetch	Decode	Decode	Decode	Rename	ROB Rd	Rdy/Sch	Dispatch	Exec

- No trace or decode cache
- 3 parallel decoders
  - 1 complex (max 4 uops)
  - 2 simple (max 1 uop)
  - Up to 6 uops
- Microsequencer
  - > 4 uop instructions
  - 4 uops/cycle
- Decoding consumes 3 cycles

Pentium Pro Decoders (II)



AMD K6 Decoders



- 2 full x86 decoders
- Up to 4 uops

Instruction Buffering

- Decouples one pipeline stage from another

- E.g., buffering between fetch and decode
  - Sometimes decode can take too long or stalls
    - Microsequenced instructions
    - Insufficient decoder strength (simple decoder and complex instruction)
    - Backend stalls (e.g. full window stall)

+ Fetch can continue filling the buffer when decode stalls

+ When fetch stalls, the decoder will be supplied instructions from the buffer

-- Extra complexity and buffer