Computer Architecture

Lecture 11: Control-Flow Handling

Prof. Onur Mutlu
ETH Zürich

Fall 2017

26 October 2017

Summary of Yesterday's Lecture

- Control Dependence Handling
  - Problem
  - Six solutions
- Branch Prediction

Agenda for Today

- Trace Caches
- Other Methods of Control Dependence Handling

Required Readings

- McFarling, "Combining Branch Predictors," DEC WRL Technical Report, 1993. ***Required***

- T. Yeh and Y. Patt, "Two-Level Adaptive Training Branch Prediction," Intl. Symposium on Microarchitecture, November 1991.
  - **MICRO Test of Time Award Winner (after 24 years)**
  - ***Required***

Recommended Readings

- Smith and Sohi, "The Microarchitecture of Superscalar Processors," Proceedings of the IEEE, 1995
  - More advanced pipelining
  - Interrupt and exception handling
  - Out-of-order and superscalar execution concepts
  - ***Recommended***

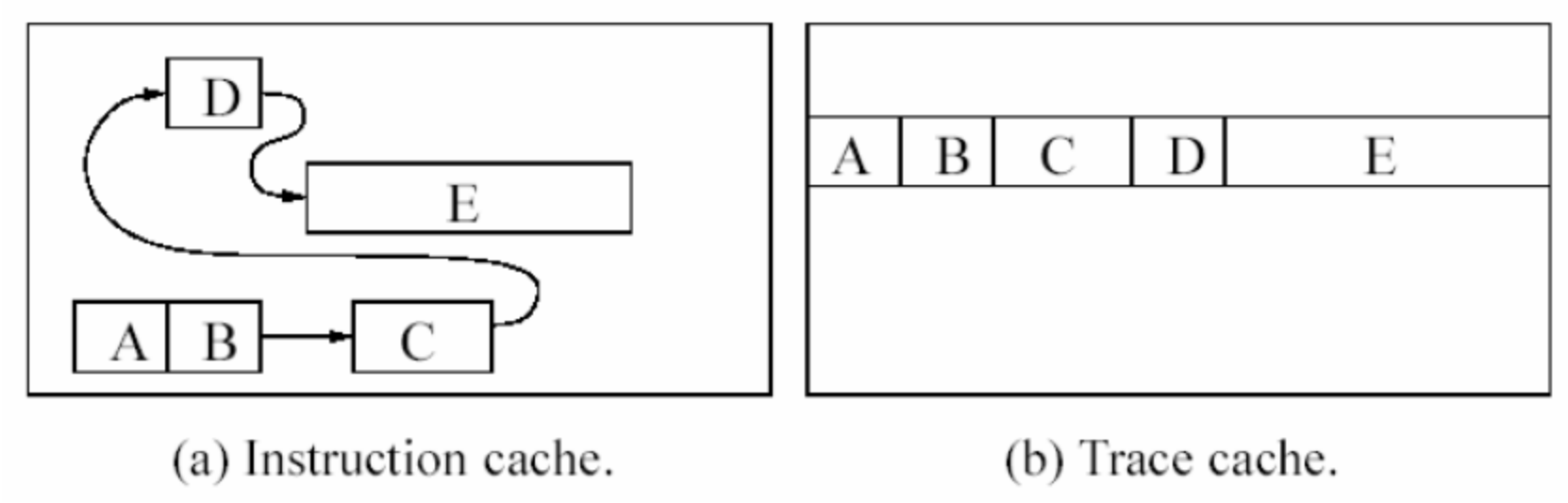
- Kessler, "The Alpha 21264 Microprocessor," IEEE Micro 1999.
  - ***Recommended***

Techniques to Reduce Fetch Breaks

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

Trace Cache: Basic Idea

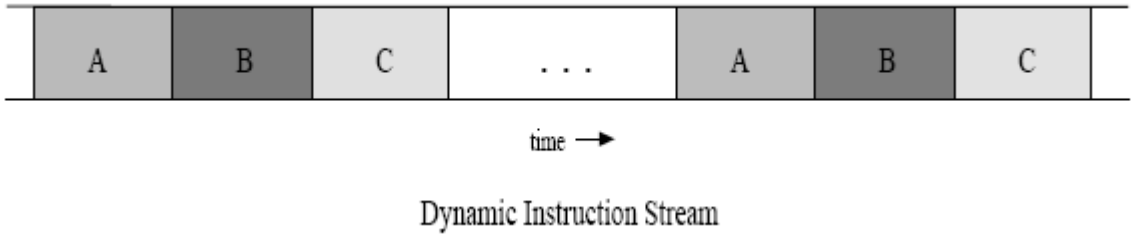
- A trace is a sequence of executed instructions.
- It is specified by a start address and the outcomes of control transfer instructions within the trace.
- Traces repeat: programs have frequently executed paths
- Trace cache idea: Store a dynamic instruction sequence in the same physical location so that it can be fetched in unison.



(a) Instruction cache. (b) Trace cache.

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)

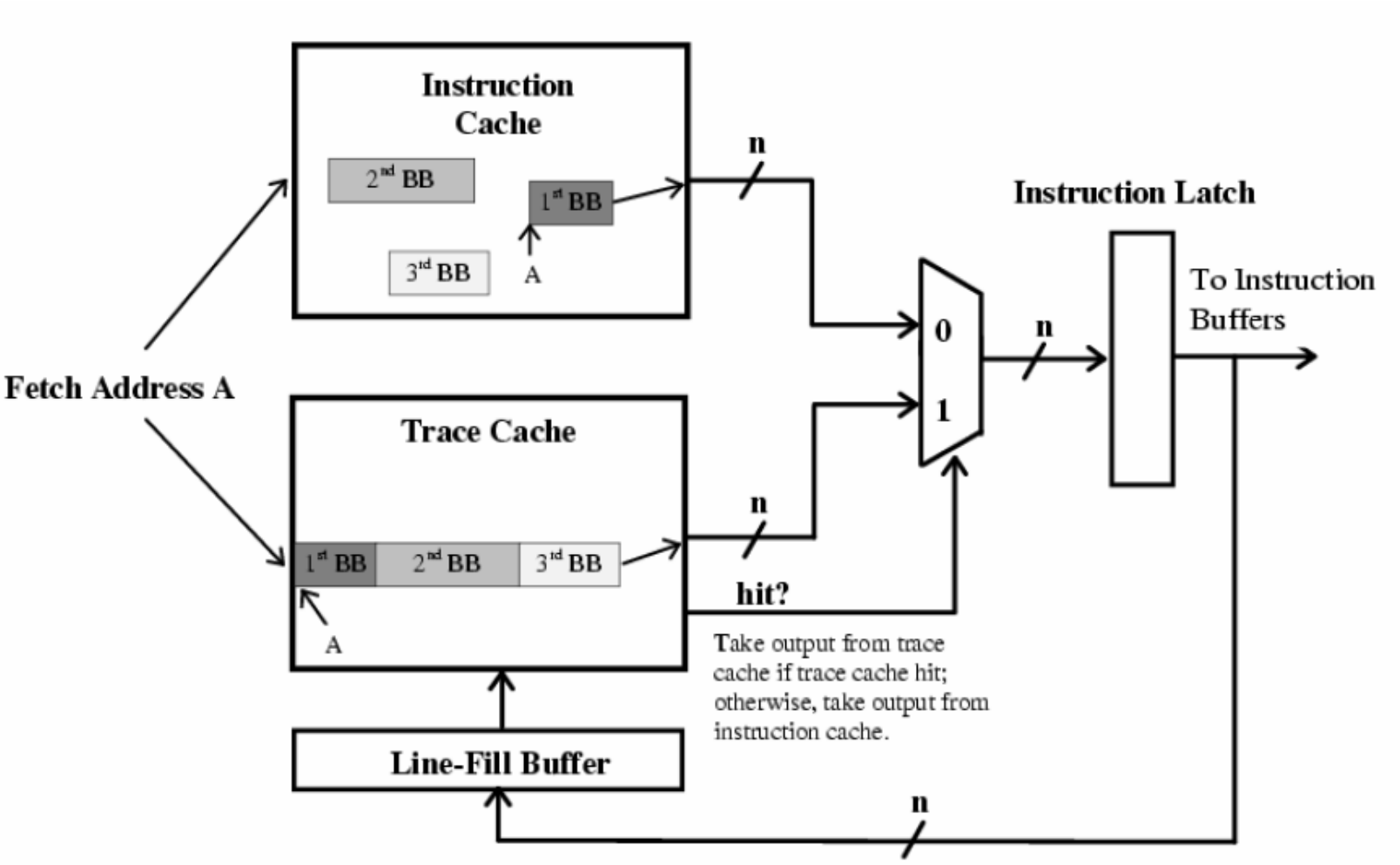


Dynamic Instruction Stream

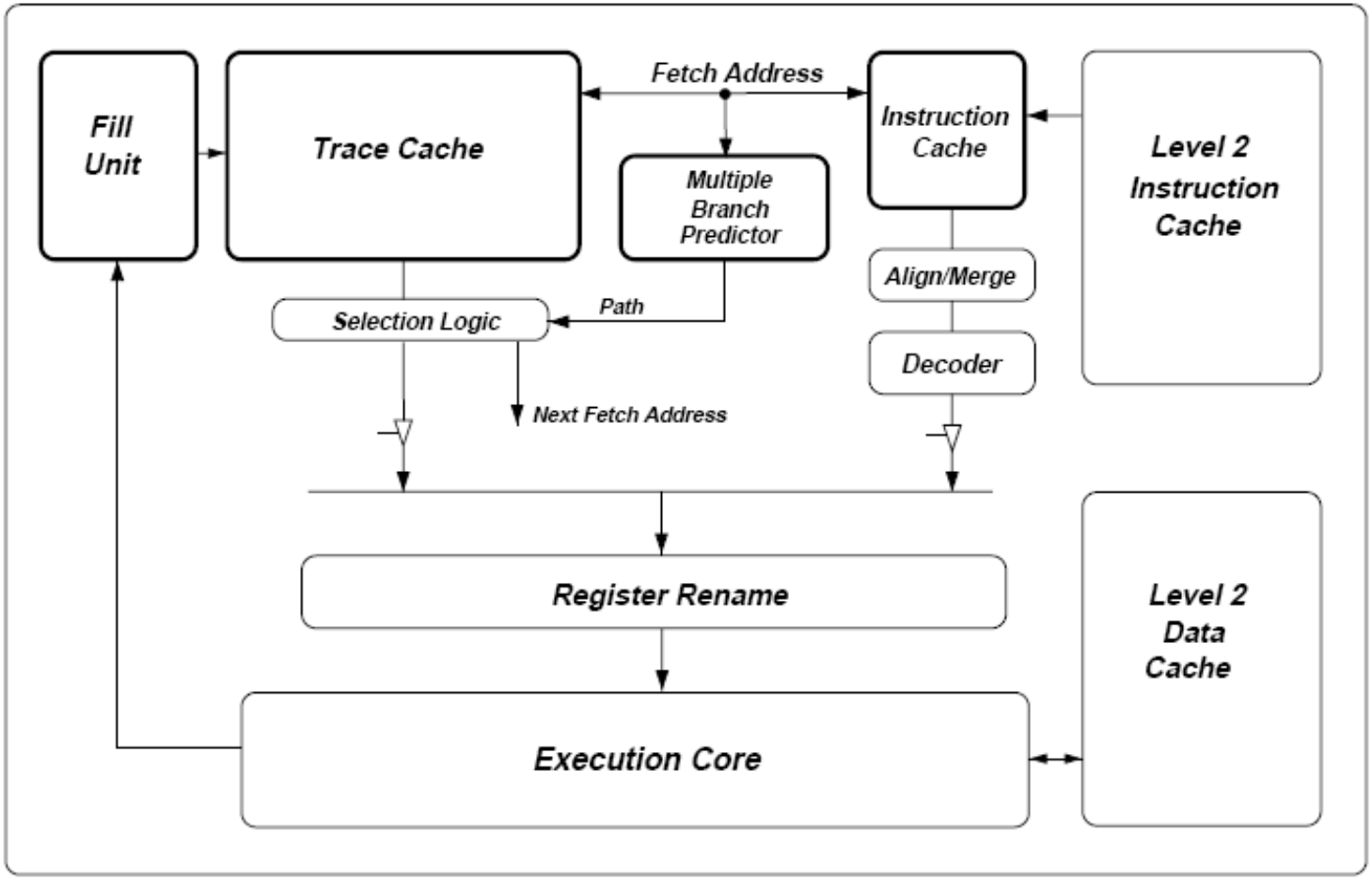
- Basic trace cache operation:
  - Fetch from consecutively-stored basic blocks (predict next trace or branches)
  - Verify the executed branch directions with the stored ones
  - If mismatch, flush the remaining portion of the trace

- Rotenberg et al., "Trace Cache: a Low Latency Approach to High Bandwidth Instruction Fetching," MICRO 1996. **Received the MICRO Test of Time Award 20 years later**
- Patel et al., "Critical Issues Regarding the Trace Cache Fetch Mechanism," Umich TR, 1997.

Trace Cache: Example



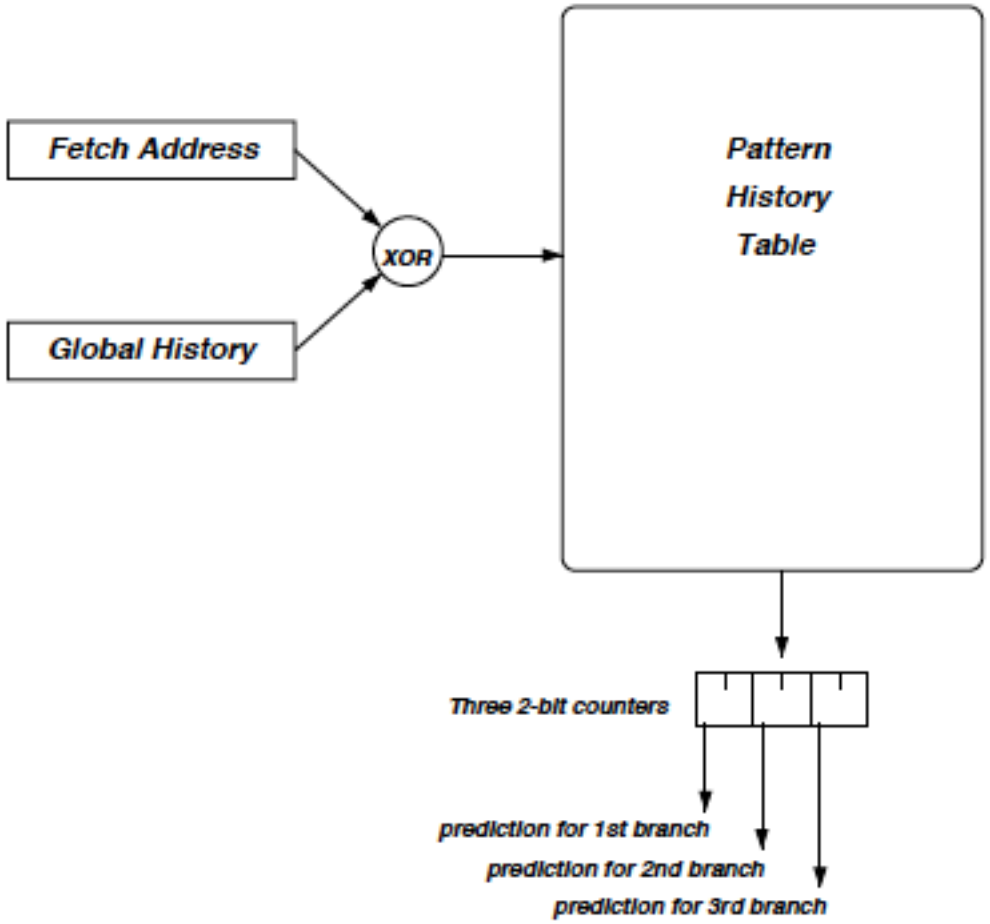
An Example Trace Cache Based Processor



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

Multiple Branch Predictor

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

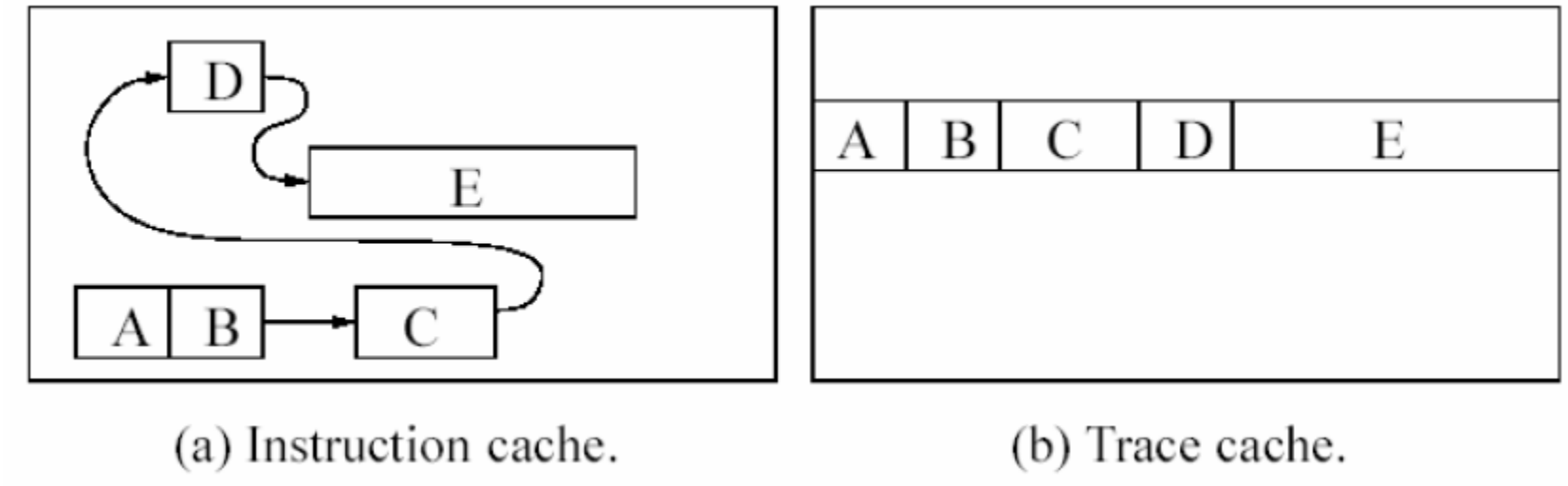


What Does A Trace Cache Line Store?

- 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

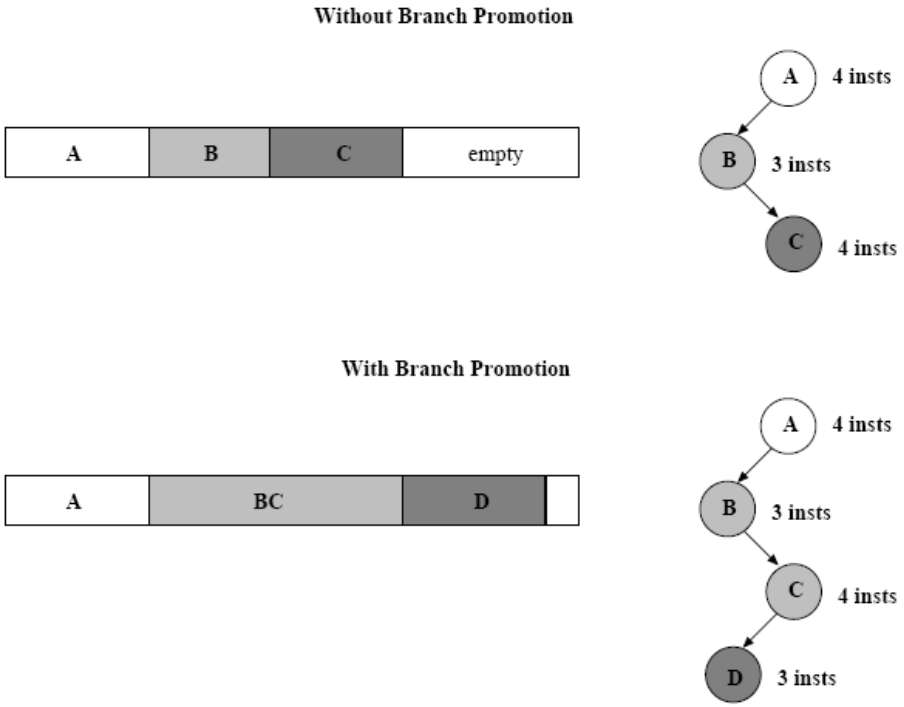


(a) Instruction cache. (b) Trace cache.

+ 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: Example

- 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

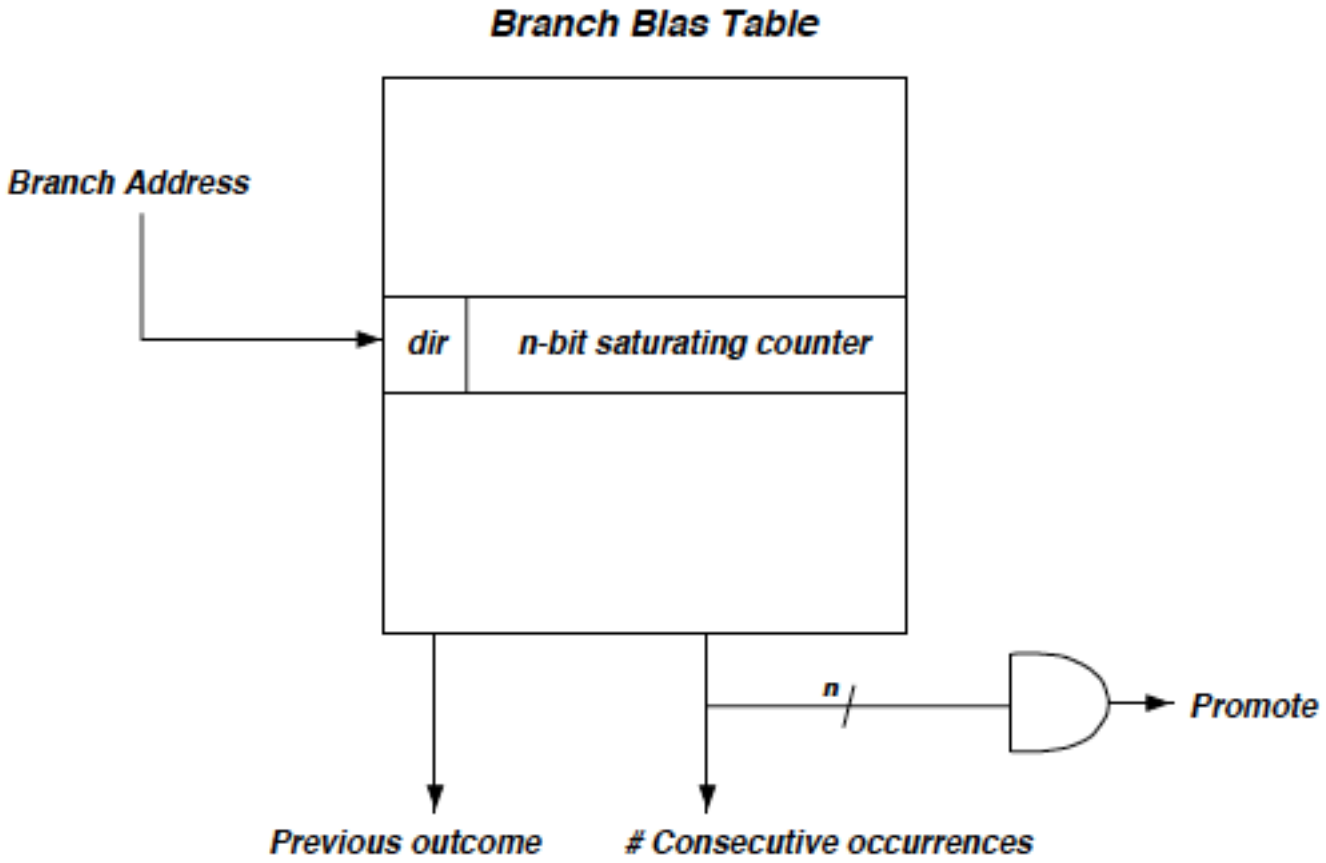


Figure 6.19: Diagram of the branch bias table.

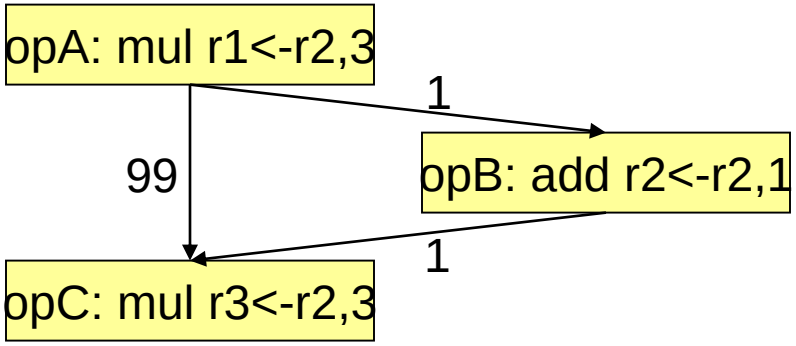
Fill Unit Optimizations

- 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

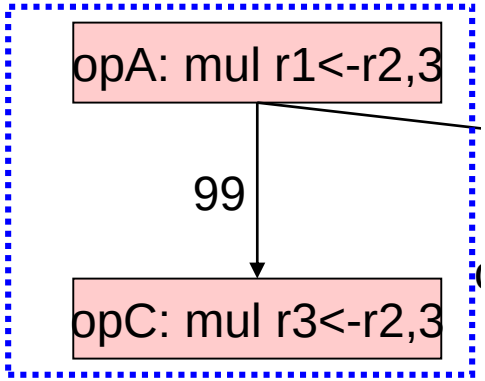
$$\begin{array}{l} ADDI\ Rx \leftarrow Ry + 4 \\ ADDI\ Rz \leftarrow Rx + 4 \end{array} \longrightarrow \begin{array}{l} ADDI\ Rx \leftarrow Ry + 4 \\ ADDI\ Rz \leftarrow Ry + 8 \end{array}$$

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

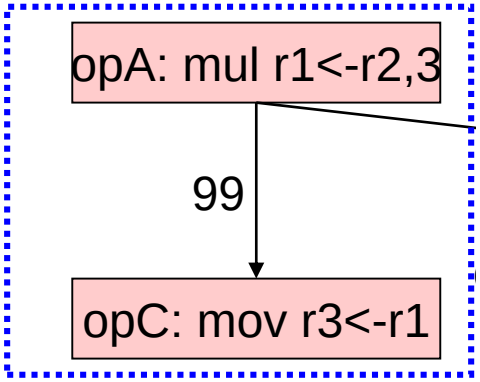
Remember This Optimization?



Original Code



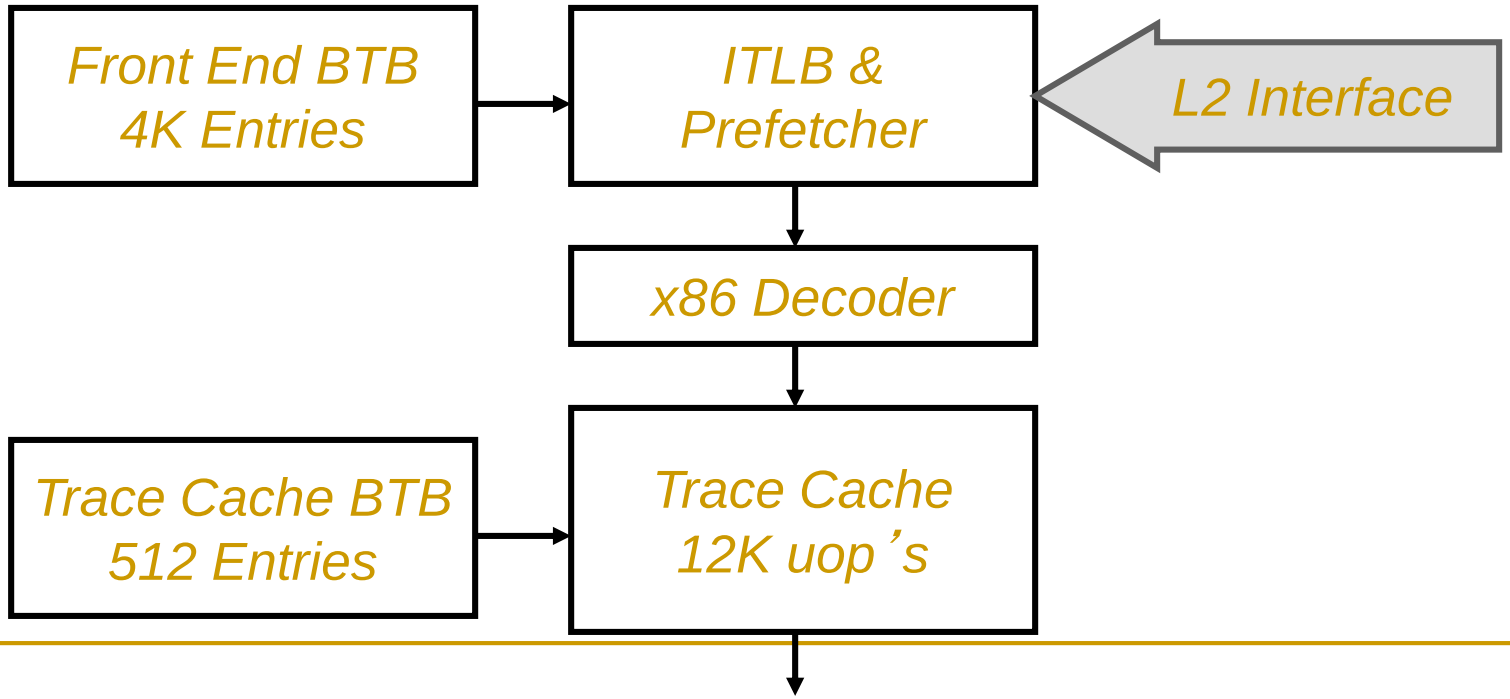
Part of Trace in Fill Unit



Optimized Trace

Intel 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



Other Ways of Handling Branches

How to Handle Control Dependences

- Critical to keep the pipeline full with correct sequence of dynamic instructions.

- Potential solutions if the instruction is a control-flow instruction:

- Stall the pipeline until we know the next fetch address
- Guess the next fetch address (branch prediction)
- Employ delayed branching (branch delay slot)
- Do something else (fine-grained multithreading)
- Eliminate control-flow instructions (predicated execution)
- Fetch from both possible paths (if you know the addresses of both possible paths) (multipath execution)

Delayed Branching (I)

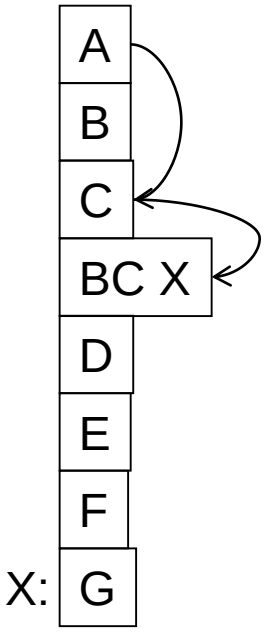
- Change the semantics of a branch instruction
  - Branch after N instructions
  - Branch after N cycles
- Idea: Delay the execution of a branch. N instructions (delay slots) that come after the branch are **always** executed regardless of branch direction.

- Problem: How do you find instructions to fill the delay slots?
  - Branch must be independent of delay slot instructions

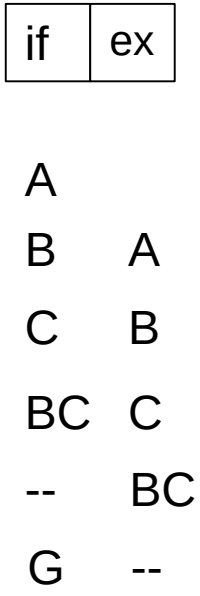
- Unconditional branch: Easier to find instructions to fill the delay slot
- Conditional branch: Condition computation should not depend on instructions in delay slots → difficult to fill the delay slot

Delayed Branching (II)

Normal code:

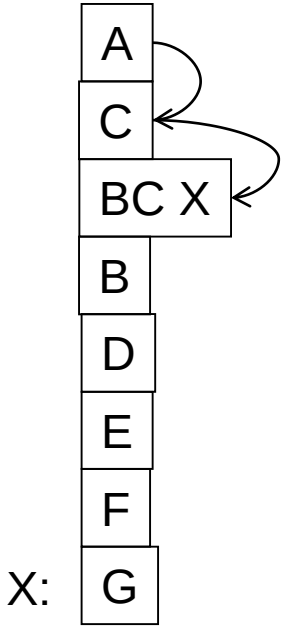


Timeline:

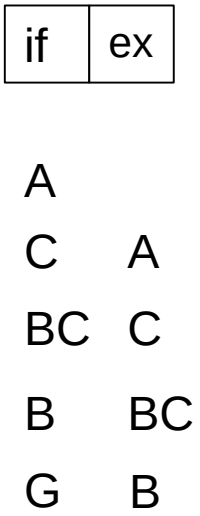


6 cycles

Delayed branch code:



Timeline:

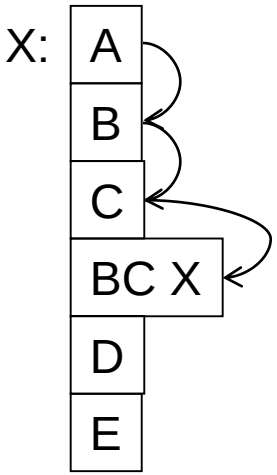


5 cycles

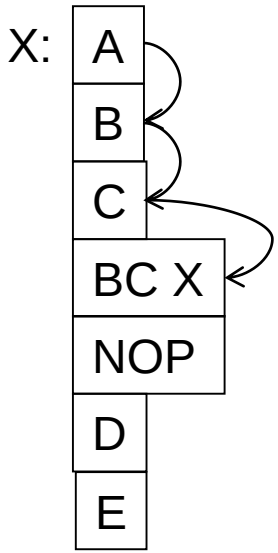
Fancy Delayed Branching (III)

- Delayed branch with squashing
  - In SPARC
  - Semantics: If the branch falls through (i.e., it is not taken), the delay slot instruction is not executed
  - Why could this help?

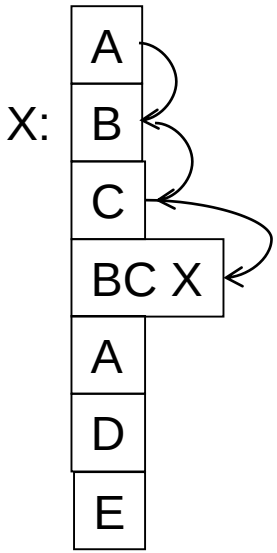
Normal code:



Delayed branch code:



Delayed branch w/ squashing:



Delayed Branching (IV)

- Advantages:

  + Keeps the pipeline full with useful instructions in a simple way assuming

    1. Number of delay slots == number of instructions to keep the pipeline full before the branch resolves

    2. All delay slots can be filled with useful instructions

- Disadvantages:

  -- Not easy to fill the delay slots (even with a 2-stage pipeline)

    1. Number of delay slots increases with pipeline depth, superscalar execution width

    2. Number of delay slots should be variable with variable latency operations. Why?

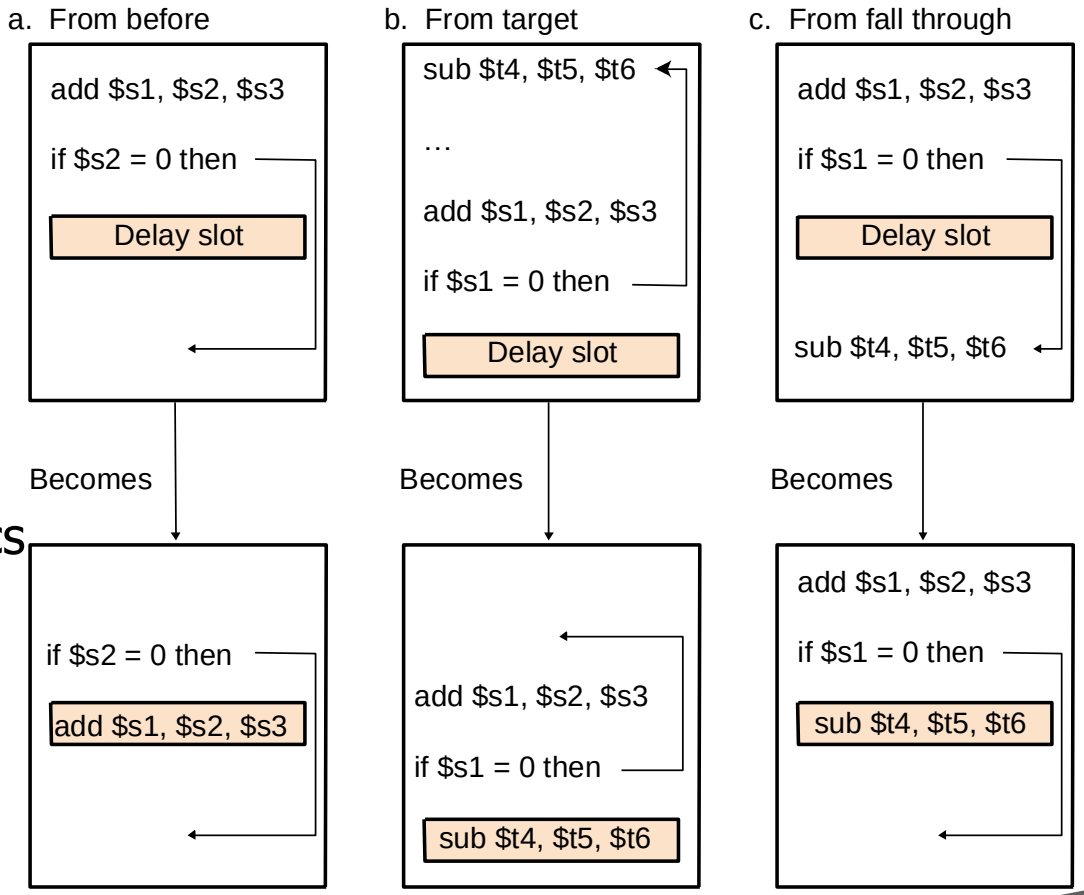
  -- Ties ISA semantics to hardware implementation

    -- SPARC, MIPS, HP-PA: 1 delay slot

    -- What if pipeline implementation changes with the next design?

An Aside: Filling the Delay Slot

reordering data independent (RAW, WAW, WAR) instructions does not change program semantics



within same basic block

For correctness: add a new instruction to the not-taken path?

For correctness: add a new instruction to the taken path?

Safe?

How to Handle Control Dependences

- Critical to keep the pipeline full with correct sequence of dynamic instructions.

- Potential solutions if the instruction is a control-flow instruction:

- Stall the pipeline until we know the next fetch address
- Guess the next fetch address (branch prediction)
- Employ delayed branching (branch delay slot)
- Do something else (fine-grained multithreading)
- Eliminate control-flow instructions (predicated execution)
- Fetch from both possible paths (if you know the addresses of both possible paths) (multipath execution)

Fine-Grained Multithreading

Fine-Grained Multithreading

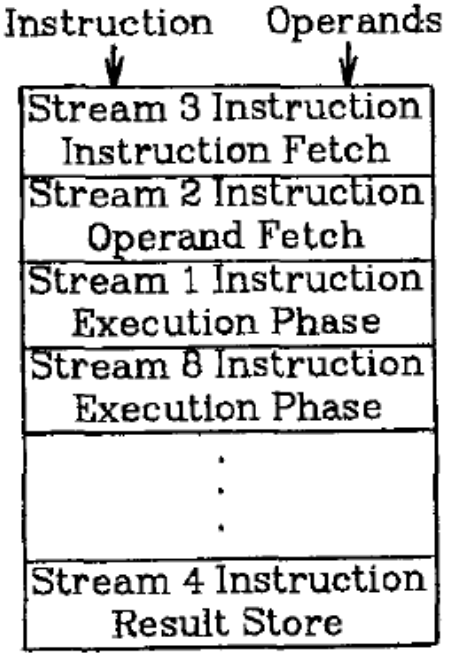
- Idea: Hardware has multiple thread contexts (PC+registers). Each cycle, fetch engine fetches from a different thread.
  - By the time the fetched branch/instruction resolves, no instruction is fetched from the same thread
  - Branch/instruction resolution latency overlapped with execution of other threads' instructions

+ No logic needed for handling control and data dependences within a thread

-- Single thread performance suffers

-- Extra logic for keeping thread contexts

-- Does not overlap latency if not enough threads to cover the whole pipeline



Fine-Grained Multithreading (II)

- Idea: Switch to another thread every cycle such that no two instructions from a thread are in the pipeline concurrently

- Tolerates the control and data dependency latencies by overlapping the latency with useful work from other threads
- Improves pipeline utilization by taking advantage of multiple threads

- Thornton, "Parallel Operation in the Control Data 6600," AFIPS 1964.
- Smith, "A pipelined, shared resource MIMD computer," ICPP 1978.

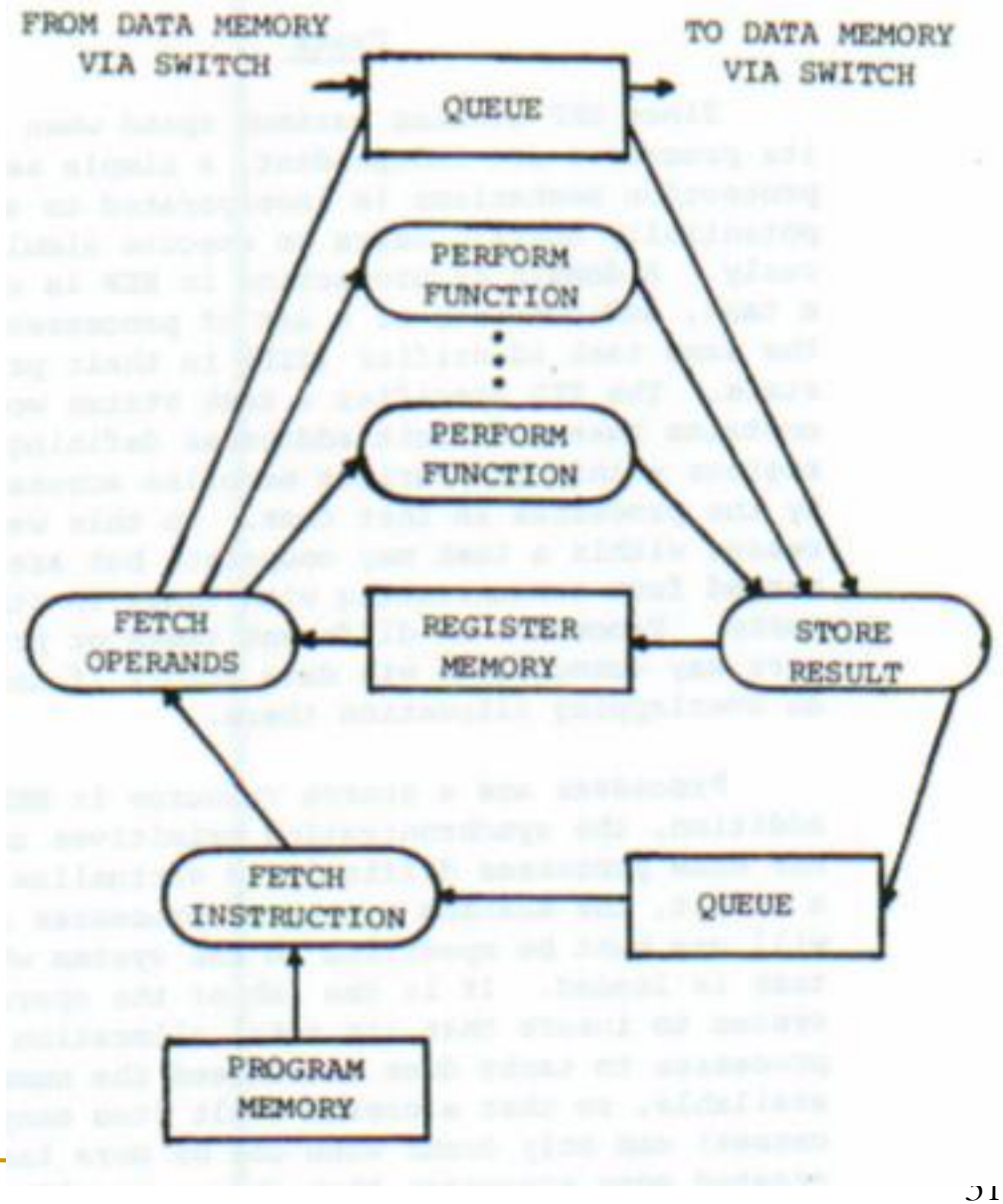
Fine-Grained Multithreading: History

- CDC 6600's peripheral processing unit is fine-grained multithreaded
  - Thornton, "Parallel Operation in the Control Data 6600," AFIPS 1964.
  - Processor executes a different I/O thread every cycle
  - An operation from the same thread is executed every 10 cycles

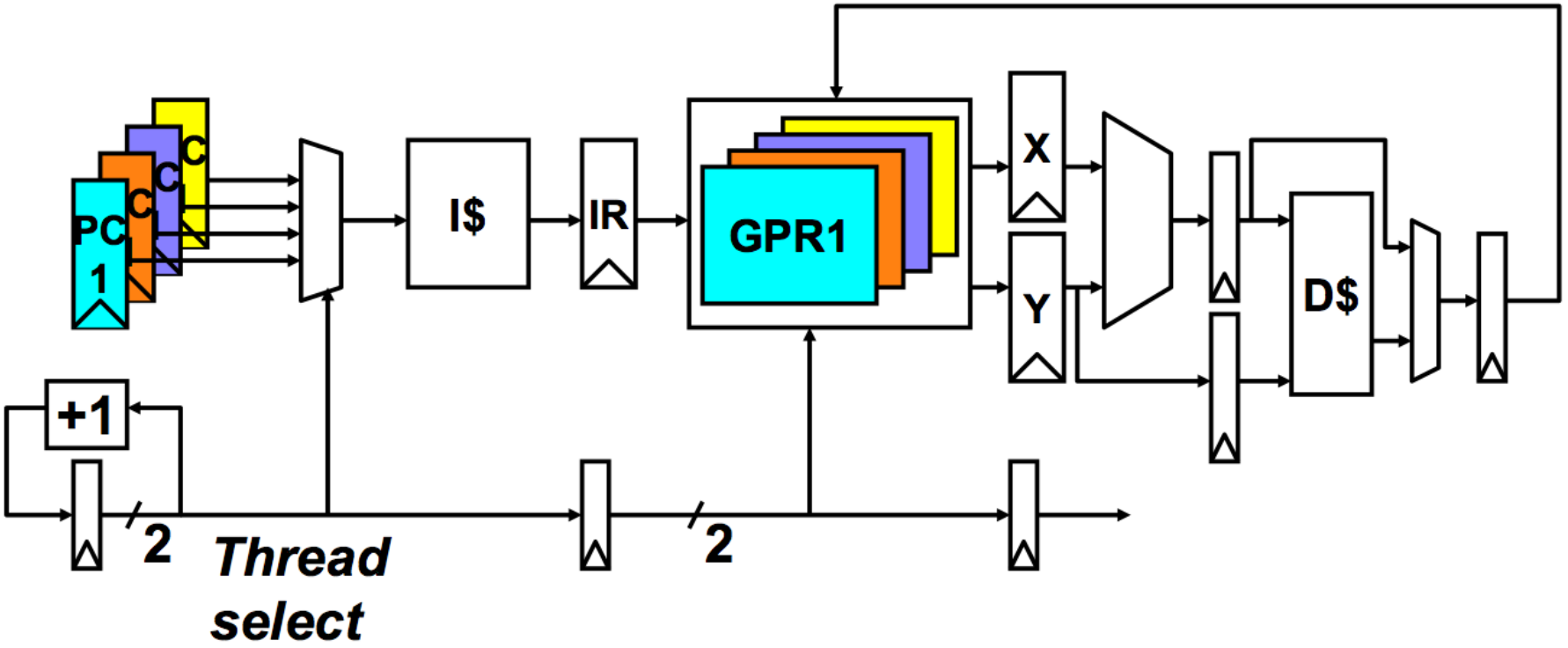
- Denelcor HEP (Heterogeneous Element Processor)
  - Smith, "A pipelined, shared resource MIMD computer," ICPP 1978.
  - 120 threads/processor
  - available queue vs. unavailable (waiting) queue for threads
  - each thread can have only 1 instruction in the processor pipeline; each thread independent
  - to each thread, processor looks like a non-pipelined machine
  - system throughput vs. single thread performance tradeoff

Fine-Grained Multithreading in HEP

- Cycle time: 100ns
- 8 stages → 800 ns to complete an instruction
  - assuming no memory access
- No control and data dependency checking

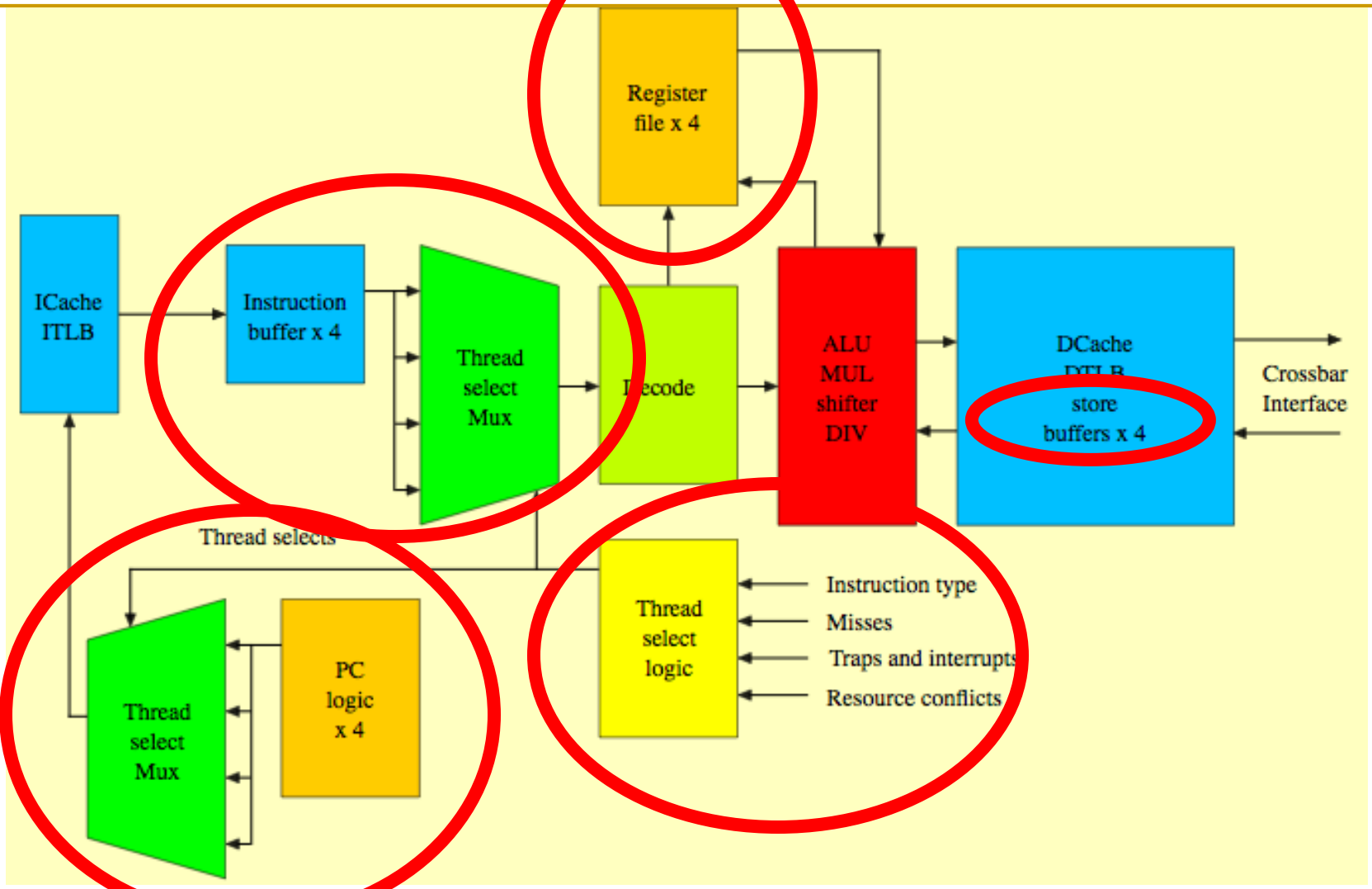


Multithreaded Pipeline Example



Slide credit: Joel Emer

Sun Niagara Multithreaded Pipeline



Kongetira et al., "Niagara: A 32-Way Multithreaded Sparc Processor," IEEE Micro 2005.

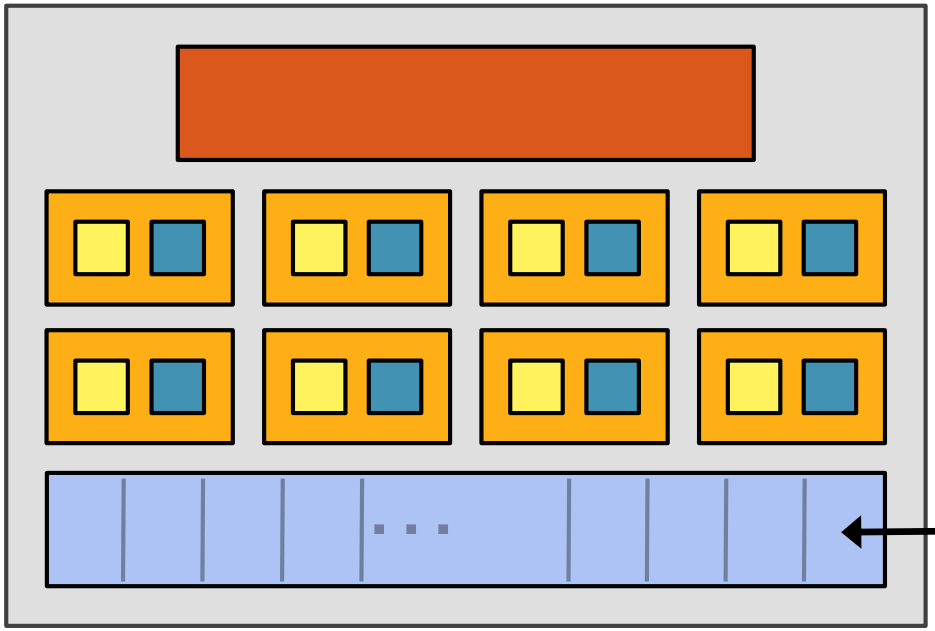
Fine-grained Multithreading

- Advantages
  - + No need for dependency checking between instructions (only one instruction in pipeline from a single thread)
  - + No need for branch prediction logic
  - + Otherwise-bubble cycles used for executing useful instructions from different threads
  - + Improved system throughput, latency tolerance, utilization

- Disadvantages
  - - Extra hardware complexity: multiple hardware contexts (PCs, register files, …), thread selection logic
  - - Reduced single thread performance (one instruction fetched every N cycles from the same thread)
  - - Resource contention between threads in caches and memory
  - - Some dependency checking logic *between* threads remains (load/store)

Modern GPUs Are FGMT Machines

NVIDIA GeForce GTX 285 "core"



64 KB of storage for thread contexts (registers)

= data-parallel (SIMD) func. unit, control shared across 8 units

= multiply-add

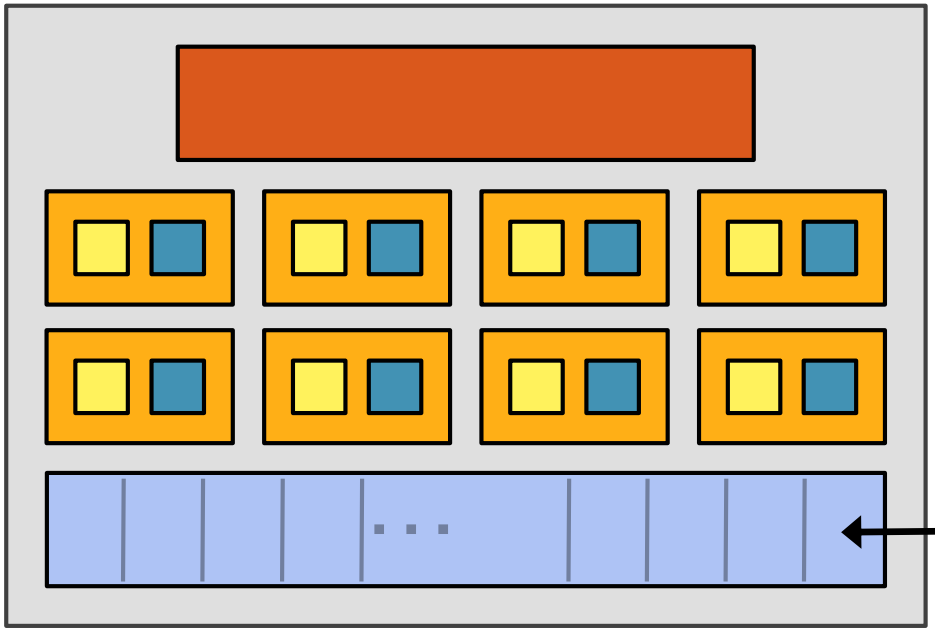
= multiply

= instruction stream decode

= execution context storage

Slide credit: Kayvon Fatahalian

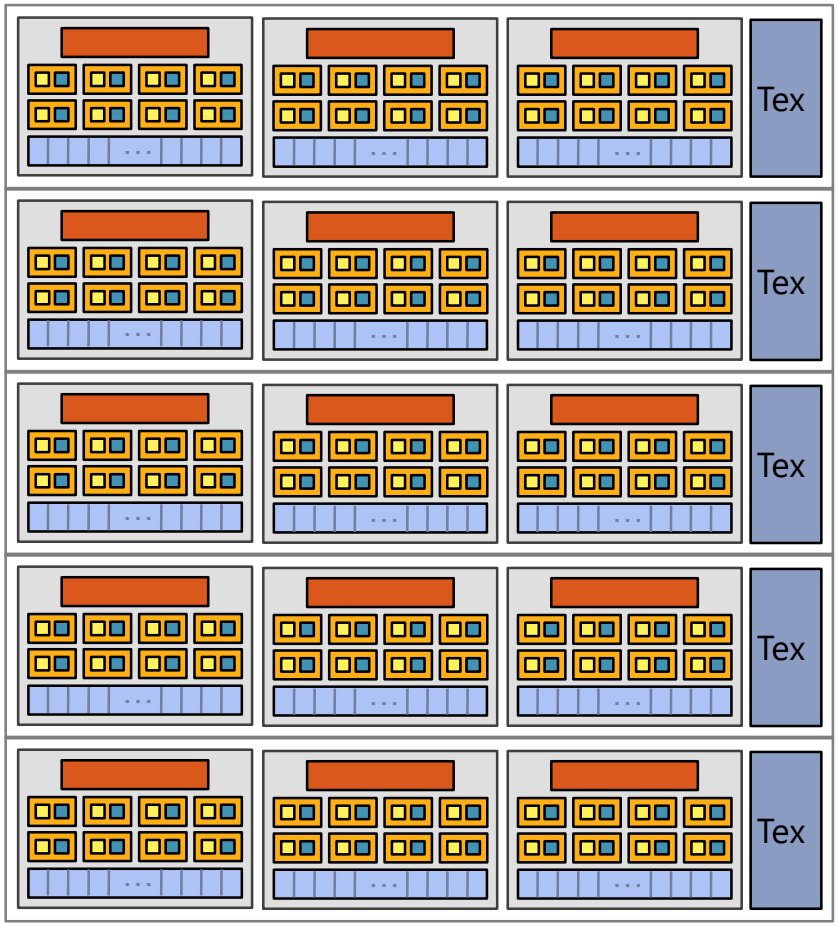
NVIDIA GeForce GTX 285 "core"

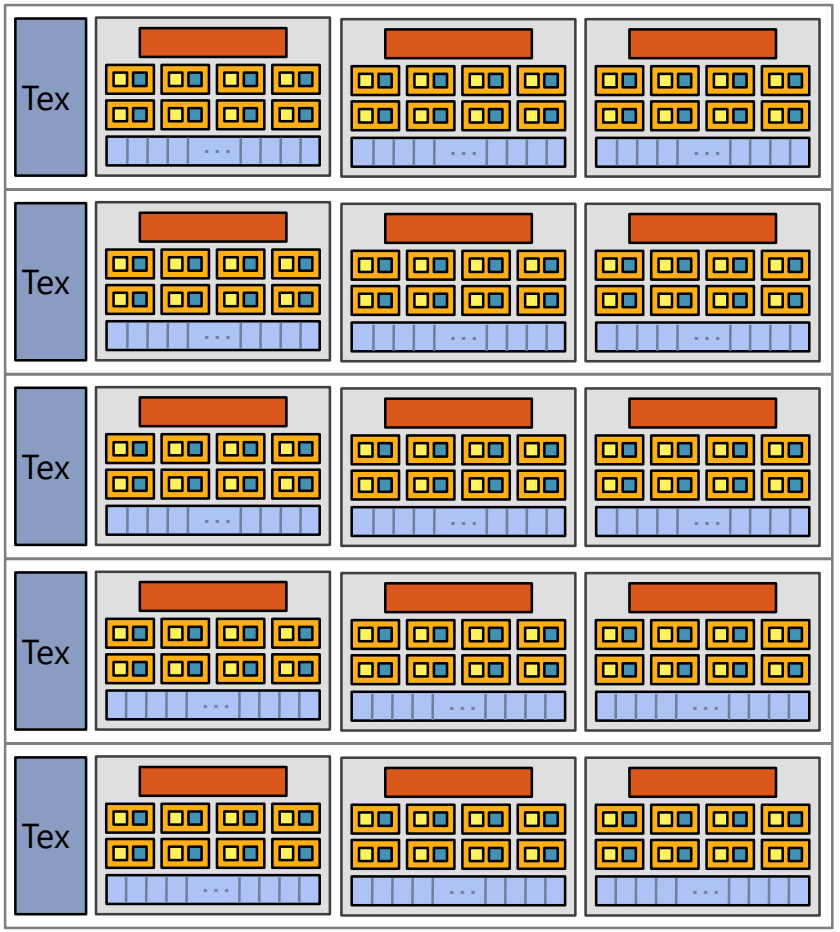


- Groups of 32 threads share instruction stream (each group is a Warp): they execute the same instruction on different data
- **Up to 32 warps are interleaved in an FGMT manner**
- Up to 1024 thread contexts can be stored

Slide credit: Kayvon Fatahalian

NVIDIA GeForce GTX 285





30 cores on the GTX 285: 30,720 threads

Slide credit: Kayvon Fatahalian

End of Fine-Grained Multithreading

How to Handle Control Dependences

- Critical to keep the pipeline full with correct sequence of dynamic instructions.

- Potential solutions if the instruction is a control-flow instruction:

- Stall the pipeline until we know the next fetch address
- Guess the next fetch address (branch prediction)
- Employ delayed branching (branch delay slot)
- Do something else (fine-grained multithreading)
- Eliminate control-flow instructions (predicated execution)
- Fetch from both possible paths (if you know the addresses of both possible paths) (multipath execution)

Predicate Combining (*not* Predicated Execution)

- Complex predicates are converted into multiple branches
  - if ((a == b) && (c < d) && (a > 5000)) { … }
    - 3 conditional branches
- Problem: This increases the number of control dependencies
- Idea: Combine predicate operations to feed a single branch instruction instead of having one branch for each
  - Predicates stored and operated on using condition registers
  - A single branch checks the value of the combined predicate

+ Fewer branches in code → fewer mipredictions/stalls

-- Possibly unnecessary work

-- If the first predicate is false, no need to compute other predicates

- Condition registers exist in IBM RS6000 and the POWER architecture

Predication (Predicated Execution)

- Idea: Convert control dependence to data dependence
- Simple example: Suppose we had a Conditional Move instruction…
  - CMOV condition, R1 ← R2
  - R1 = (condition == true) ? R2 : R1
  - Employed in most modern ISAs (x86, Alpha)
- Code example with branches vs. CMOVs

  if (a == 5) {b = 4;} else {b = 3;}

  CMPEQ condition, a, 5;

  CMOV condition, b ← 4;

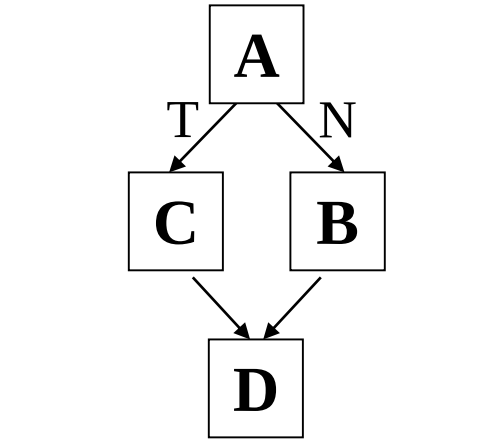
  CMOV !condition, b ← 3;

Predication (Predicated Execution)

- Idea: Compiler converts control dependence into data dependence → branch is eliminated
  - Each instruction has a predicate bit set based on the predicate computation
  - Only instructions with TRUE predicates are committed (others turned into NOPs)

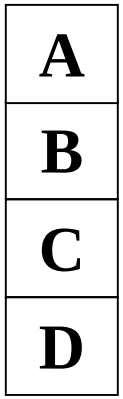
```
if (cond) {
    b = 0;
}
else {
    b = 1;
}
```

(normal branch code)




```
A   p1 = (cond)
    branch p1, TARGET
B   mov b, 1
    jmp JOIN
C   TARGET:
    mov b, 0
D   add  x, b, 1
```

(predicated code)




```
A   p1 = (cond)
B   (!p1) mov  b, 1
C   (p1) mov  b, 0
D   add  x, b, 1
```

Predicated Execution References

- Allen et al., "Conversion of control dependence to data dependence," POPL 1983.

- Kim et al., "Wish Branches: Combining Conditional Branching and Predication for Adaptive Predicated Execution," MICRO 2005.

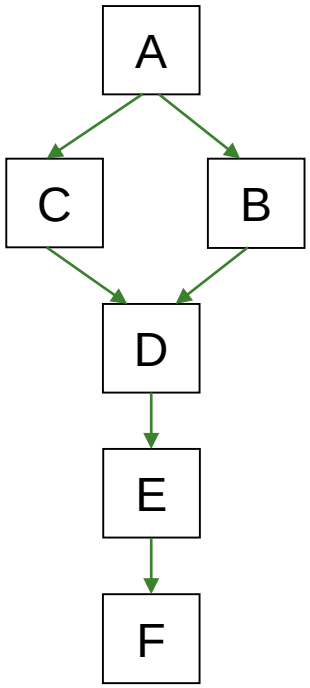
Conditional Move Operations

- Very limited form of predicated execution

- CMOV R1 ← R2
  - R1 = (ConditionCode == true) ? R2 : R1
  - Employed in most modern ISAs (x86, Alpha)

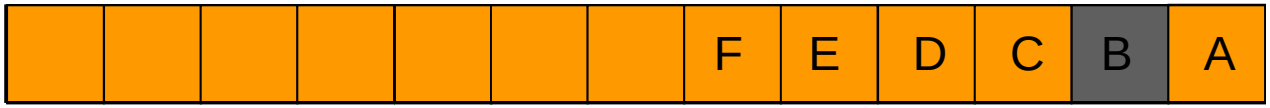
Predicated Execution (II)

- Predicated execution can be high performance and energy-efficient



Predicated Execution

Fetch Decode Rename Schedule RegisterRead Execute



nop

Branch Prediction

Fetch Decode Rename Schedule RegisterRead Execute

Pipeline flush!!

Predicated Execution

- Eliminates branches → enables straight line code (i.e., larger basic blocks in code)
- Advantages
  - Eliminates hard-to-predict branches
  - Always-not-taken prediction works better (no branches)
  - Compiler has more freedom to optimize code (no branches)
    - control flow does not hinder inst. reordering optimizations
    - code optimizations hindered only by data dependencies
- Disadvantages
  - Useless work: some instructions fetched/executed but discarded (especially bad for easy-to-predict branches)
  - Requires additional ISA (and hardware) support
  - Can we eliminate all branches this way?

Predicated Execution vs. Branch Prediction

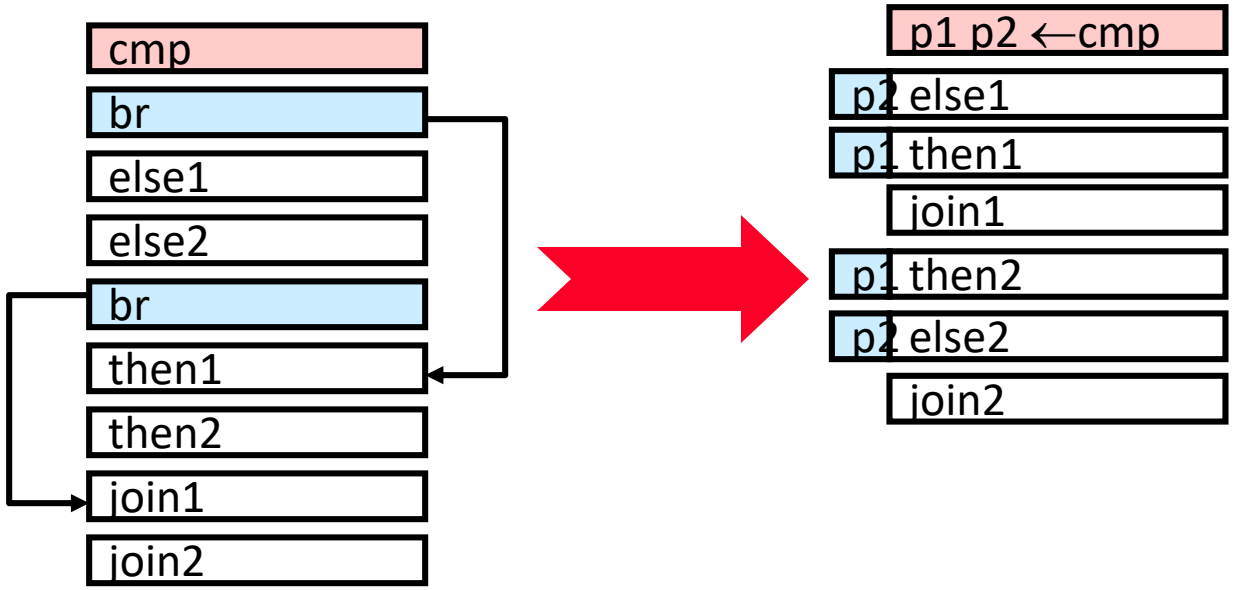
+ Eliminates mispredictions for hard-to-predict branches
  + No need for branch prediction for some branches
  + Good if misprediction cost > useless work due to predication

-- Causes useless work for branches that are easy to predict
  -- Reduces performance if misprediction cost < useless work
  -- Adaptivity: Static predication is not adaptive to run-time branch behavior. Branch behavior changes based on input set, program phase, control-flow path.

Predicated Execution in Intel Itanium

- Each instruction can be separately predicated
- 64 one-bit predicate registers

  each instruction carries a 6-bit predicate field
- An instruction is effectively a NOP if its predicate is false

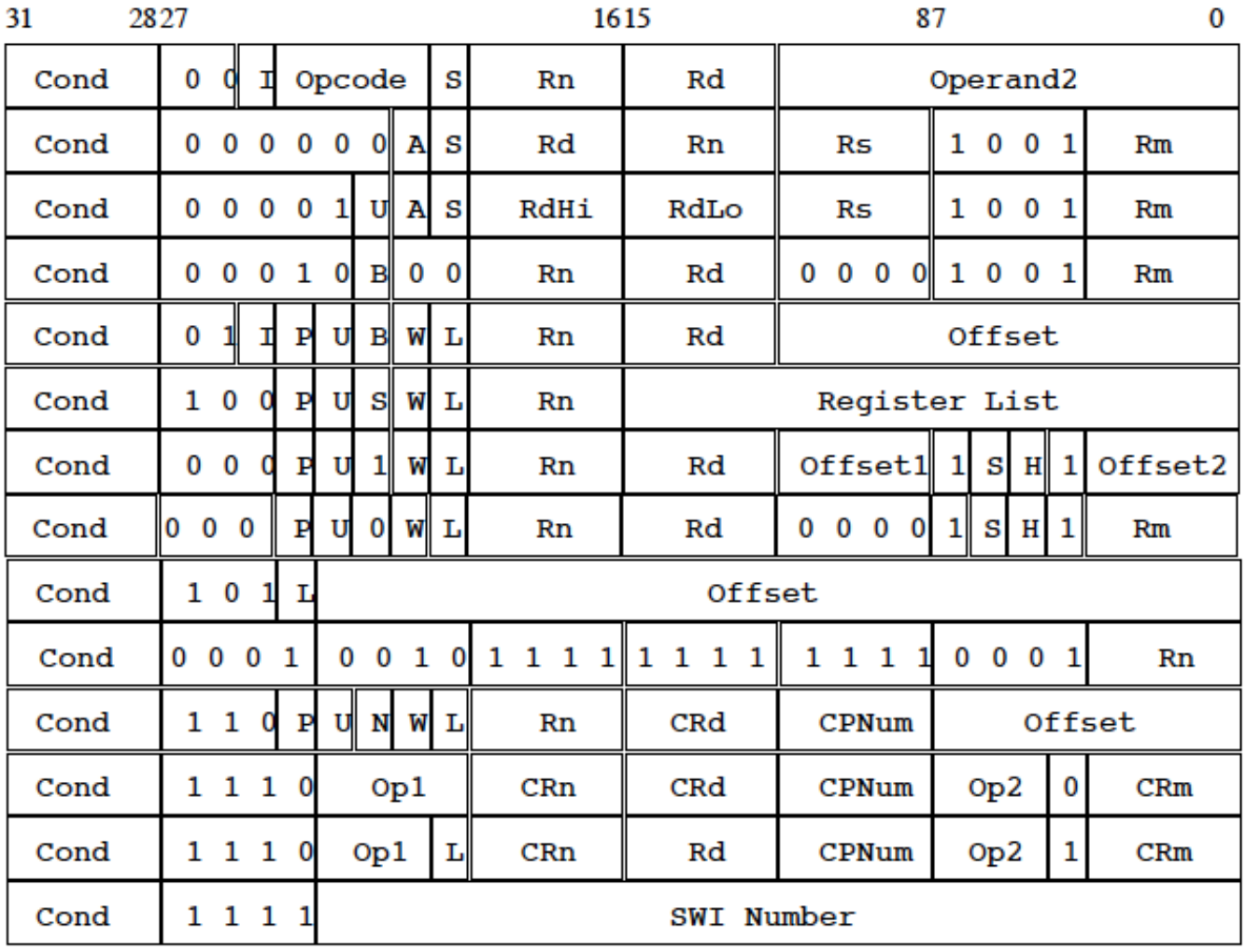


Conditional Execution in the ARM ISA

- Almost all ARM instructions can include an optional condition code.
  - Prior to ARM v8

- An instruction with a condition code is executed only if the condition code flags in the CPSR meet the specified condition.

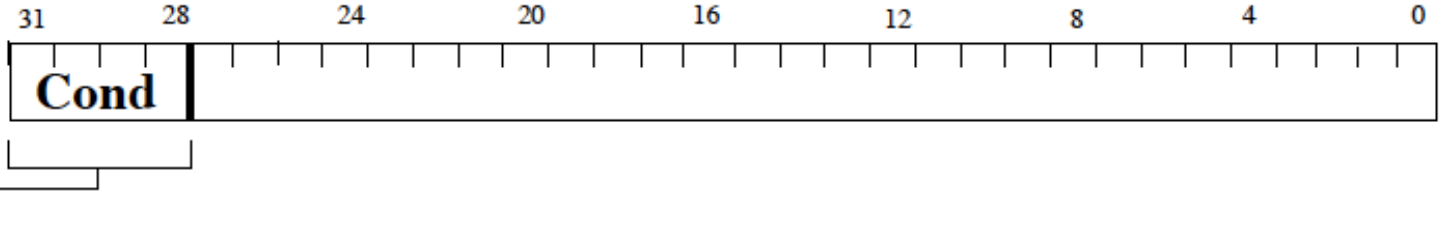
Conditional Execution in ARM ISA



Instruction type

- Data processing / PSR Transfer
- Multiply
- Long Multiply **(v3M / v4 only)**
- Swap
- Load/Store Byte/Word
- Load/Store Multiple
- Halfword transfer : Immediate offset **(v4 only)**
- Halfword transfer: Register offset **(v4 only)**
- Branch
- Branch Exchange **(v4T only)**
- Coprocessor data transfer
- Coprocessor data operation
- Coprocessor register transfer
- Software interrupt

Conditional Execution in ARM ISA



0000 = EQ - Z set (equal)

0001 = NE - Z clear (not equal)

0010 = HS / CS - C set (unsigned higher or same)

0011 = LO / CC - C clear (unsigned lower)

0100 = MI -N set (negative)

0101 = PL - N clear (positive or zero)

0110 = VS - V set (overflow)

0111 = VC - V clear (no overflow)

1000 = HI - C set and Z clear (unsigned higher)

1001 = LS - C clear or Z (set unsigned lower or same)

1010 = GE - N set and V set, or N clear and V clear (>or =)

1011 = LT - N set and V clear, or N clear and V set (>)

1100 = GT - Z clear, and either N set and V set, or N clear and V set (>)

1101 = LE - Z set, or N set and V clear,or N clear and V set (<, or =)

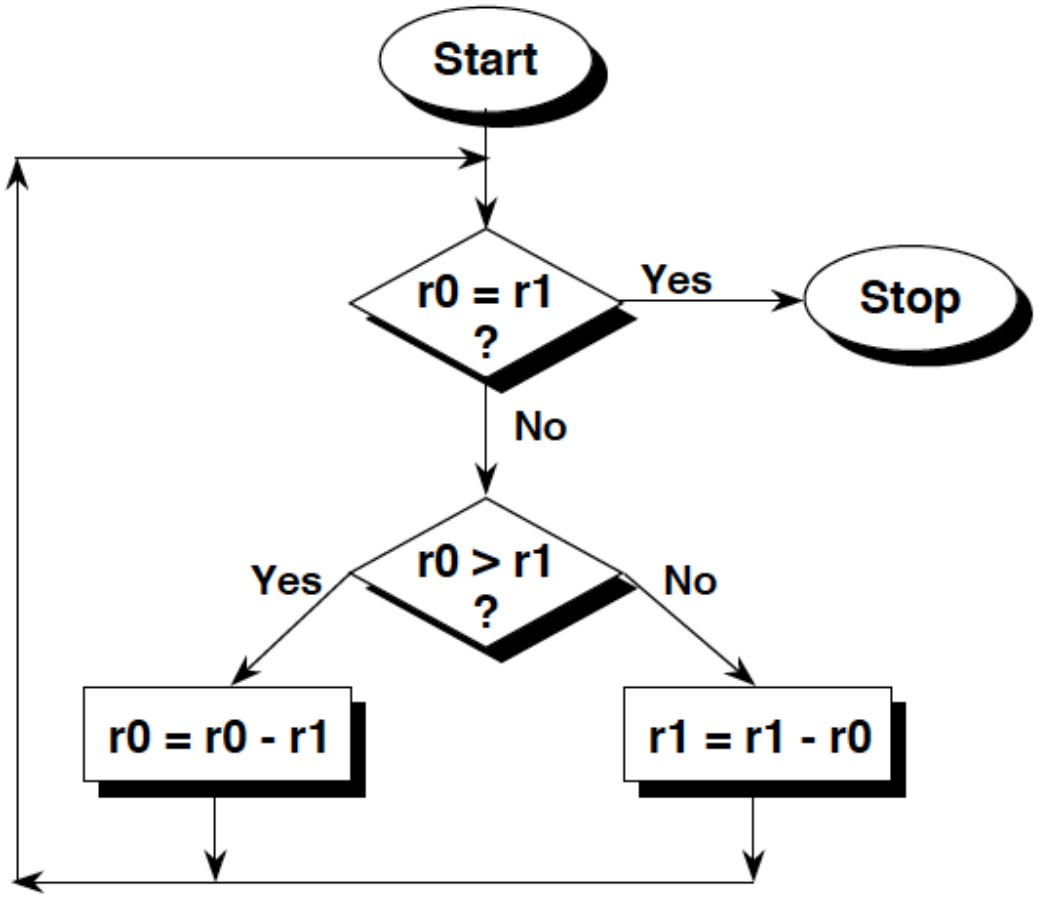
1110 = AL - always

1111 = NV - reserved.

Conditional Execution in ARM ISA

* **To execute an instruction conditionally, simply postfix it with the appropriate condition:**
  - For example an add instruction takes the form:
    - `ADD r0,r1,r2 ; r0 = r1 + r2 (ADDAL)`
  - To execute this only if the zero flag is set:
    - `ADDEQ r0,r1,r2 ; If zero flag set then...`
      `; ... r0 = r1 + r2`
* **By default, data processing operations do not affect the condition flags (apart from the comparisons where this is the only effect). To cause the condition flags to be updated, the S bit of the instruction needs to be set by postfixing the instruction (and any condition code) with an "S".**
  - For example to add two numbers and set the condition flags:
    - `ADDS r0,r1,r2 ; r0 = r1 + r2`
      `; ... and set flags`

Conditional Execution in ARM ISA



* **Convert the GCD algorithm given in this flowchart into**
  1) "Normal" assembler, where only branches can be conditional.
  2) ARM assembler, where all instructions are conditional, thus improving code density.

* **The only instructions you need are CMP, B and SUB.**

Conditional Execution in ARM ISA

"Normal" Assembler

```
gcd     cmp r0, r1        ;reached the end?
        beq stop
        blt less          ;if r0 > r1
        sub r0, r0, r1    ;subtract r1 from r0
        bal gcd
less    sub r1, r1, r0    ;subtract r0 from r1
        bal gcd
stop
```

ARM Conditional Assembler

```
gcd     cmp   r0, r1      ;if r0 > r1
        subgt r0, r0, r1  ;subtract r1 from r0
        sublt r1, r1, r0  ;else subtract r0 from r1
        bne   gcd         ;reached the end?
```

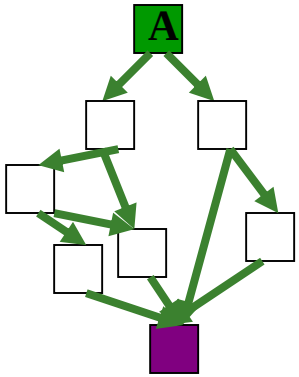
Idealism

- Wouldn't it be nice
  - If the branch is eliminated (predicated) only when it would actually be mispredicted
  - If the branch were predicted when it would actually be correctly predicted

- Wouldn't it be nice
  - If predication did not require ISA support

Improving Predicated Execution

- Three major limitations of predication
  1. Adaptivity: non-adaptive to branch behavior
  2. Complex CFG: inapplicable to loops/complex control flow graphs
  3. ISA: Requires large ISA changes



- Wish Branches [Kim+, MICRO 2005]
  - Solve 1 and partially 2 (for loops)

- Dynamic Predicated Execution
  - Diverge-Merge Processor [Kim+, MICRO 2006]
    - Solves 1, 2 (partially), 3

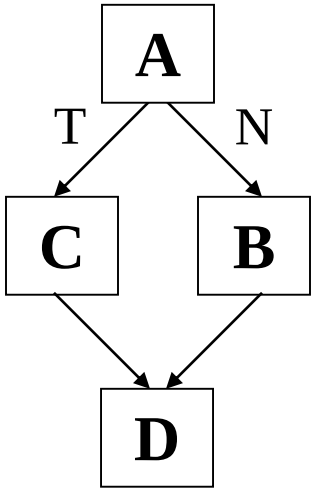
Wish Branches

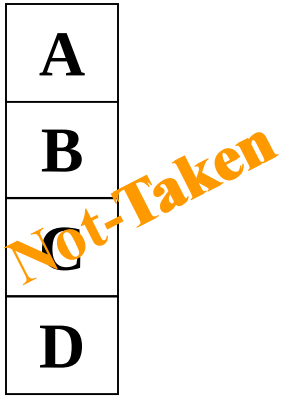
- The **compiler** generates code (with wish branches) that can be executed **either** as predicated code **or** non-predicated code (normal branch code)
- The **hardware decides** to execute predicated code or normal branch code at run-time based on the confidence of branch prediction
- **Easy to predict: normal branch code**
- **Hard to predict: predicated code**

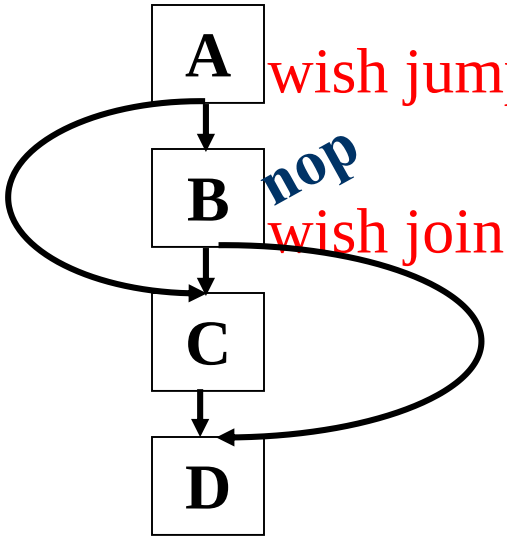
- Kim et al., "Wish Branches: Enabling Adaptive and Aggressive Predicated Execution," MICRO 2006, IEEE Micro Top Picks, Jan/Feb 2006.

Wish Jump/Join

High Confidence








```
A   p1 = (cond)
    branch p1, TARGET
B   mov b, 1
    jmp JOIN
C   TARGET:
    mov b,0
```

normal branch code

```
A   p1 = (cond)
B   (!p1) mov b,1
C   (p1) mov b,0
```

predicated code

```
A   p1=(cond)
    wish.jump p1 TARGET
B   (!p1) mov b,1
    wish.join !p1 JOIN
C   TARGET:
    (p1) mov b,0
D   JOIN:
```

nop

wish jump/join code

Wish Branches vs. Predicated Execution

- Advantages compared to predicated execution
  - **Reduces the overhead** of predication
  - Increases the benefits of predicated code by allowing the compiler to generate more **aggressively-predicated code**
  - Makes predicated code less dependent on machine configuration (e.g. branch predictor)

- Disadvantages compared to predicated execution
  - Extra branch instructions use machine resources
  - Extra branch instructions increase the contention for branch predictor table entries
  - Constrains the compiler's scope for **code optimizations**

How to Handle Control Dependences

- Critical to keep the pipeline full with correct sequence of dynamic instructions.

- Potential solutions if the instruction is a control-flow instruction:

- Stall the pipeline until we know the next fetch address
- Guess the next fetch address (branch prediction)
- Employ delayed branching (branch delay slot)
- Do something else (fine-grained multithreading)
- Eliminate control-flow instructions (predicated execution)
- Fetch from both possible paths (if you know the addresses of both possible paths) (multipath execution)

Multi-Path Execution

- Idea: Execute both paths after a conditional branch
  - For all branches: Riseman and Foster, "The inhibition of potential parallelism by conditional jumps," IEEE Transactions on Computers, 1972.
  - For a hard-to-predict branch: Use dynamic confidence estimation

- Advantages:

  + Improves performance if misprediction cost > useless work

  + No ISA change needed

- Disadvantages:

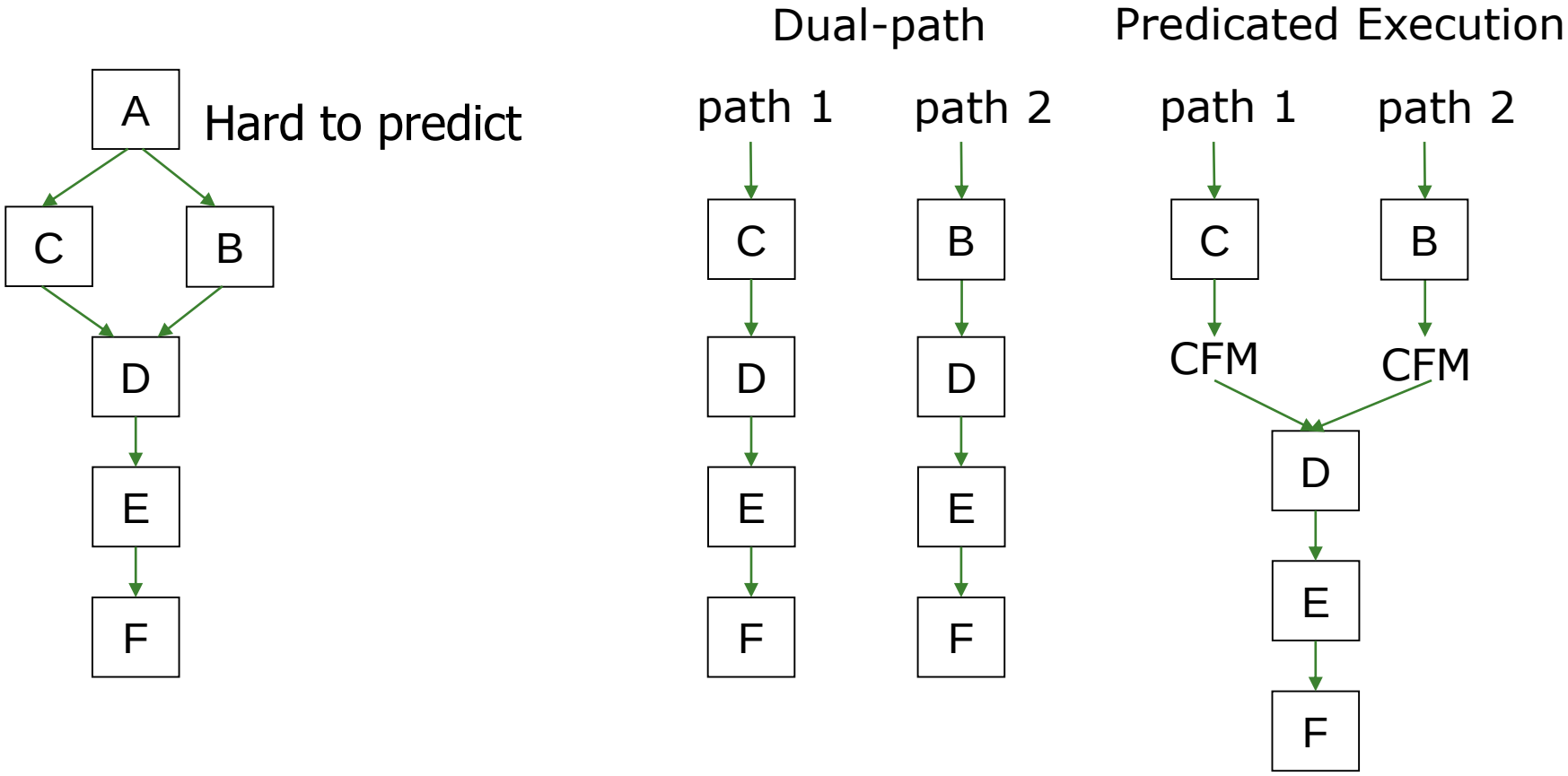
  -- What happens when the machine encounters another hard-to-predict branch? Execute both paths again?

  -- Paths followed quickly become exponential

  -- Each followed path requires its own context (registers, PC, GHR)

  -- Wasted work (and reduced performance) if paths merge

Dual-Path Execution versus Predication



Handling Other Types of Branches

Remember: Branch Types

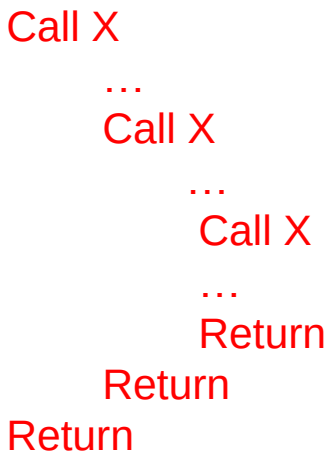
Type	Direction at fetch time	Number of possible next fetch addresses?	When is next fetch address resolved?
Conditional	Unknown	2	Execution (register dependent)
Unconditional	Always taken	1	Decode (PC + offset)
Call	Always taken	1	Decode (PC + offset)
Return	Always taken	Many	Execution (register dependent)
Indirect	Always taken	Many	Execution (register dependent)

How can we predict an indirect branch with many target addresses?

Call and Return Prediction

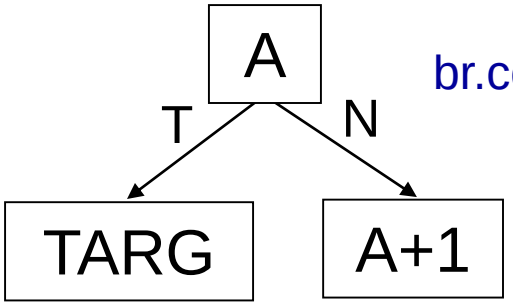
- Direct calls are easy to predict
  - Always taken, single target
  - Call marked in BTB, target predicted by BTB

- Returns are indirect branches
  - A function can be called from many points in code
  - A return instruction can have many target addresses
    - Next instruction after each call point for the same function
  - Observation: Usually a return matches a call
  - Idea: Use a stack to predict return addresses (Return Address Stack)
    - A fetched call: pushes the return (next instruction) address on the stack
    - A fetched return: pops the stack and uses the address as its predicted target
    - Accurate most of the time: 8-entry stack → > 95% accuracy



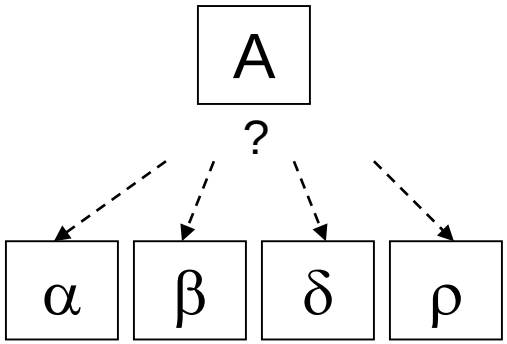
Indirect Branch Prediction (I)

- Register-indirect branches have multiple targets



br.cond TARGET

Conditional (Direct) Branch



R1 = MEM[R2]
branch R1

Indirect Jump

- Used to implement
  - Switch-case statements
  - Virtual function calls
  - Jump tables (of function pointers)
  - Interface calls

Indirect Branch Prediction (II)

- No direction prediction needed
- Idea 1: Predict the last resolved target as the next fetch address

  + Simple: Use the BTB to store the target address

  -- Inaccurate: 50% accuracy (empirical). Many indirect branches switch between different targets

- Idea 2: Use history based target prediction
  - E.g., Index the BTB with GHR XORed with Indirect Branch PC
  - Chang et al., "Target Prediction for Indirect Jumps," ISCA 1997.

  + More accurate

  -- An indirect branch maps to (too) many entries in BTB

  -- Conflict misses with other branches (direct or indirect)

  -- Inefficient use of space if branch has few target addresses

Intel Pentium M Indirect Branch Predictor

The advanced branch prediction in the Pentium M processor is based on the Intel Pentium® 4 processor's [6] branch predictor. On top of that, two additional predictors to capture special program flows, were added: a Loop Detector and an Indirect Branch Predictor.

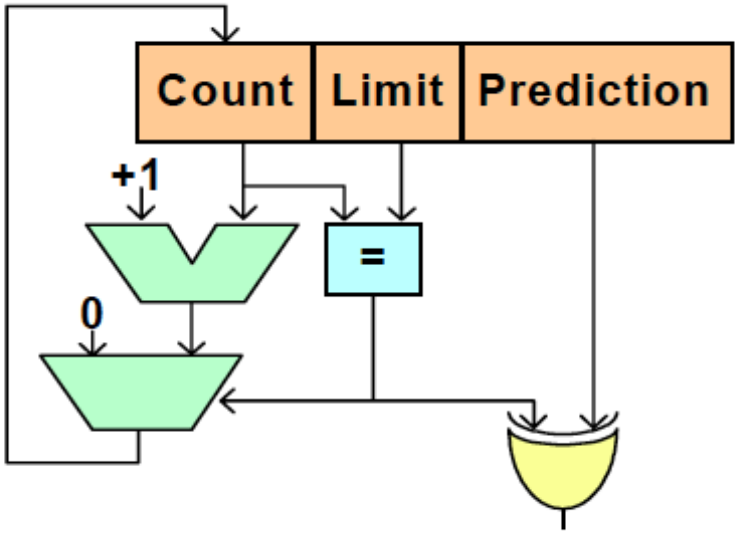


Figure 2: The Loop Detector logic

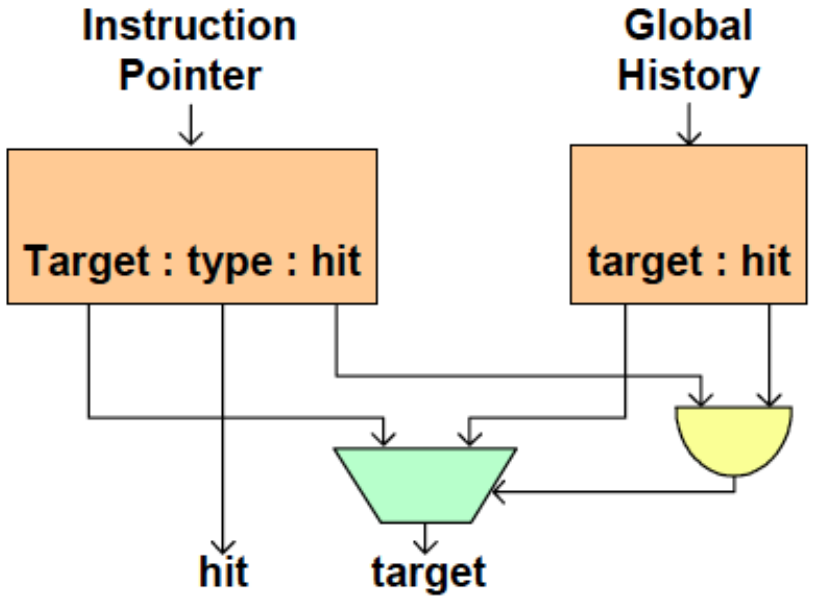


Figure 3: The Indirect Branch Predictor logic

Gochman et al.,
"The Intel Pentium M Processor: Microarchitecture and Performance,"
Intel Technology Journal, May 2003.

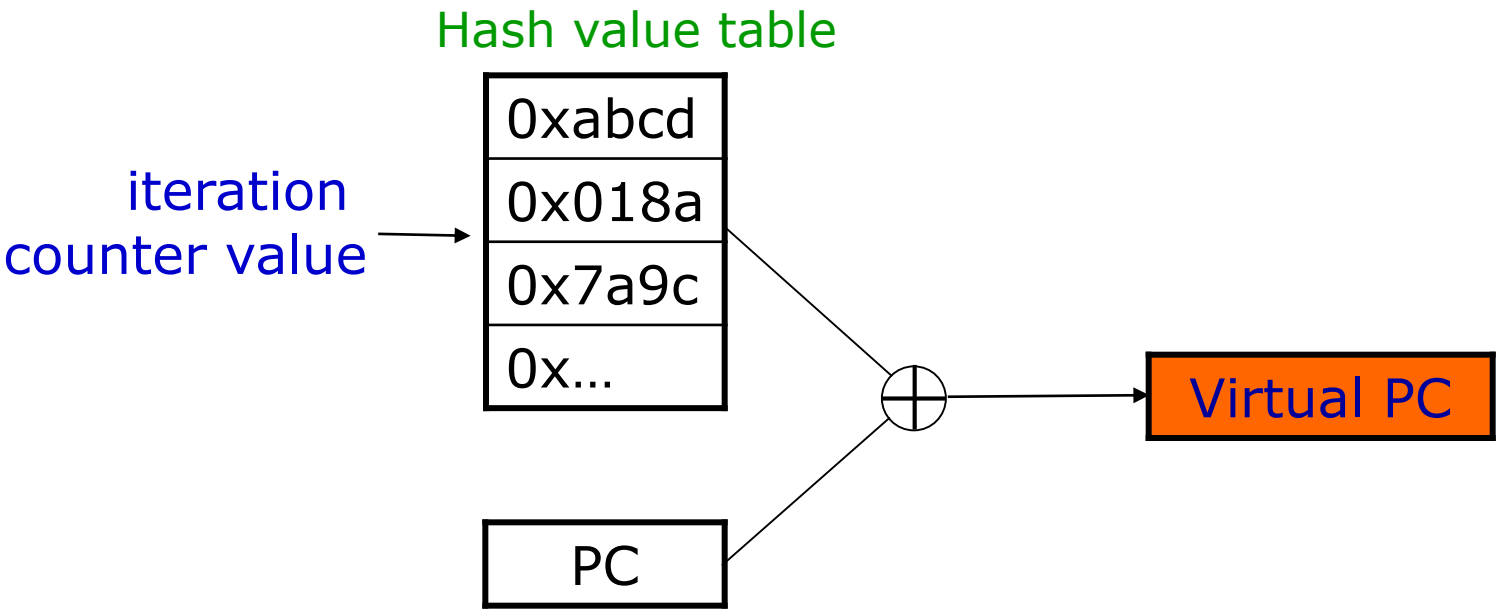
More Ideas on Indirect Branches?

- Virtual Program Counter prediction
  - Idea: Use conditional branch prediction structures ***iteratively*** to make an indirect branch prediction
  - i.e., *devirtualize* the indirect branch in hardware

- Curious?
  - Kim et al., "VPC Prediction: Reducing the Cost of Indirect Branches via Hardware-Based Dynamic Devirtualization," ISCA 2007.

Indirect Branch Prediction (III)

- Idea 3: Treat an indirect branch as "multiple virtual conditional branches" in hardware
  - Only for prediction purposes
  - Predict each "virtual conditional branch" iteratively
  - Kim et al., "VPC prediction," ISCA 2007.



VPC Prediction (I)

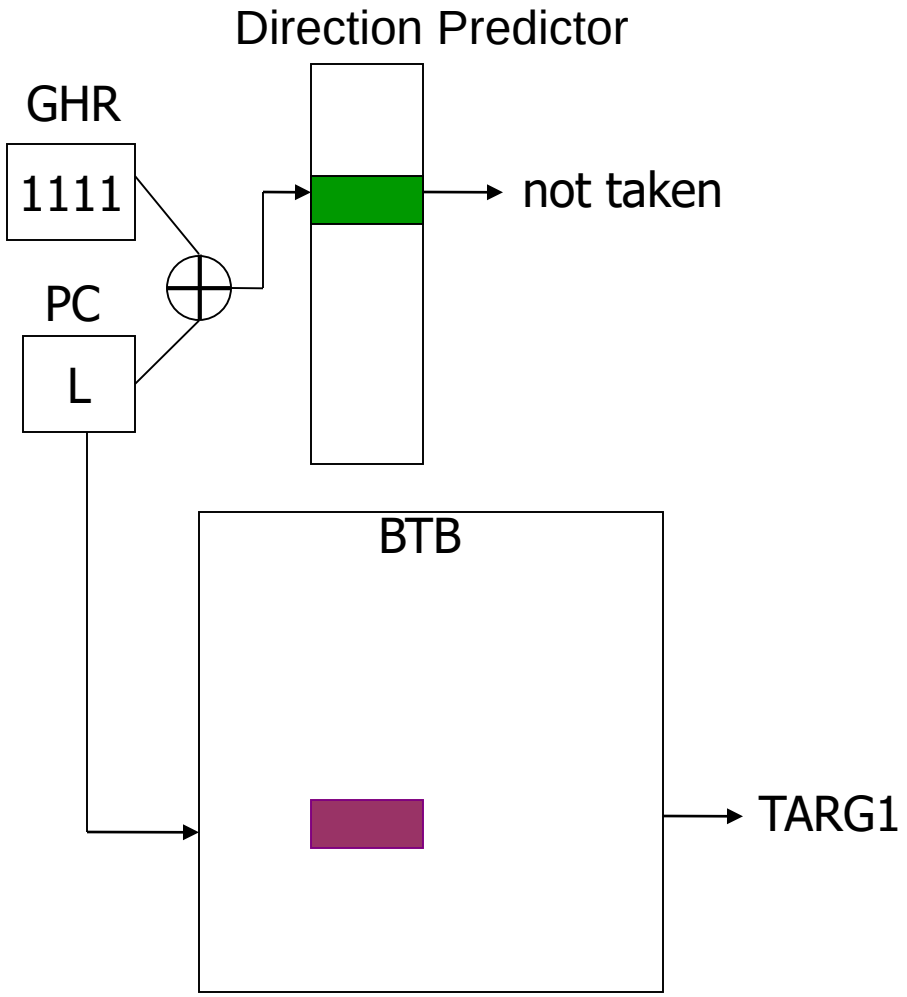
Real Instruction

call R1 // PC: L

Virtual Instructions

cond. jump TARG1 // VPC: L
cond. jump TARG2 // VPC: VL2
cond. jump TARG3 // VPC: VL3
cond. jump TARG4 // VPC: VL4

Next iteration



VPC Prediction (II)

Real Instruction

call R1 // PC: L

Virtual Instructions

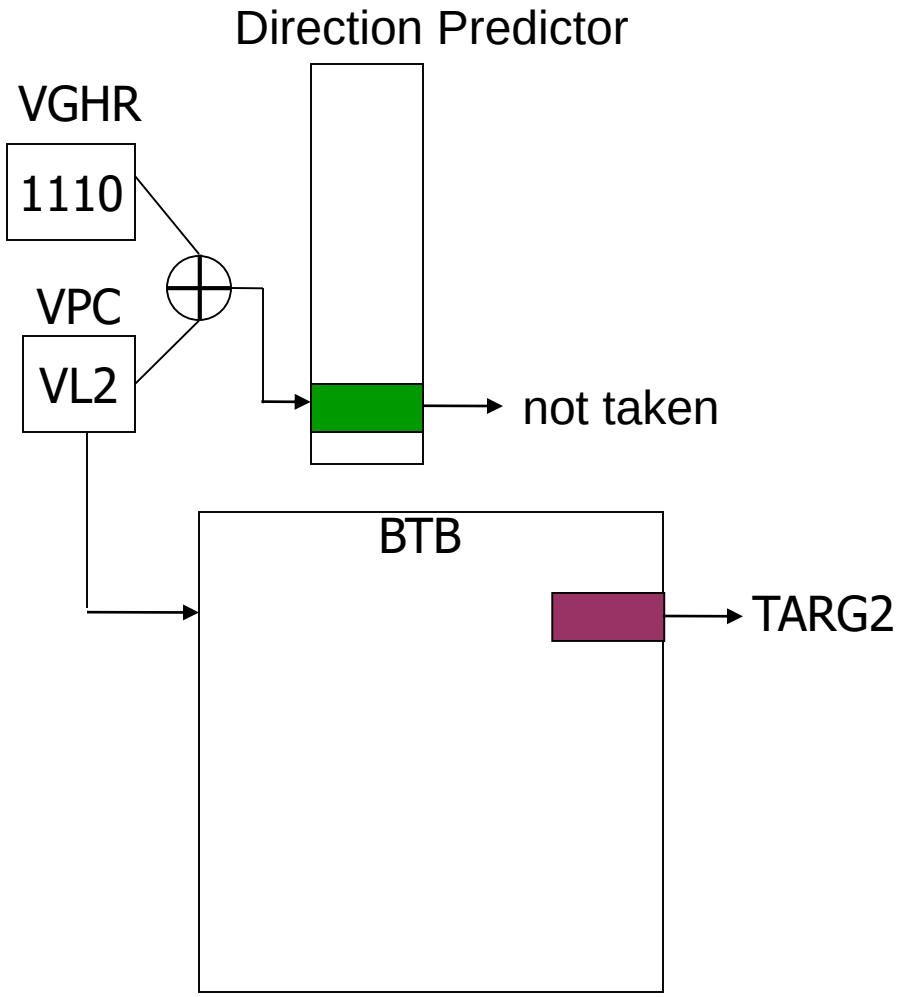
cond. jump TARG1 // VPC: L

cond. jump TARG2 // VPC: VL2

cond. jump TARG3 // VPC: VL3

cond. jump TARG4 // VPC: VL4

Next iteration



VPC Prediction (III)

Real Instruction

call R1 // PC: L

Virtual Instructions

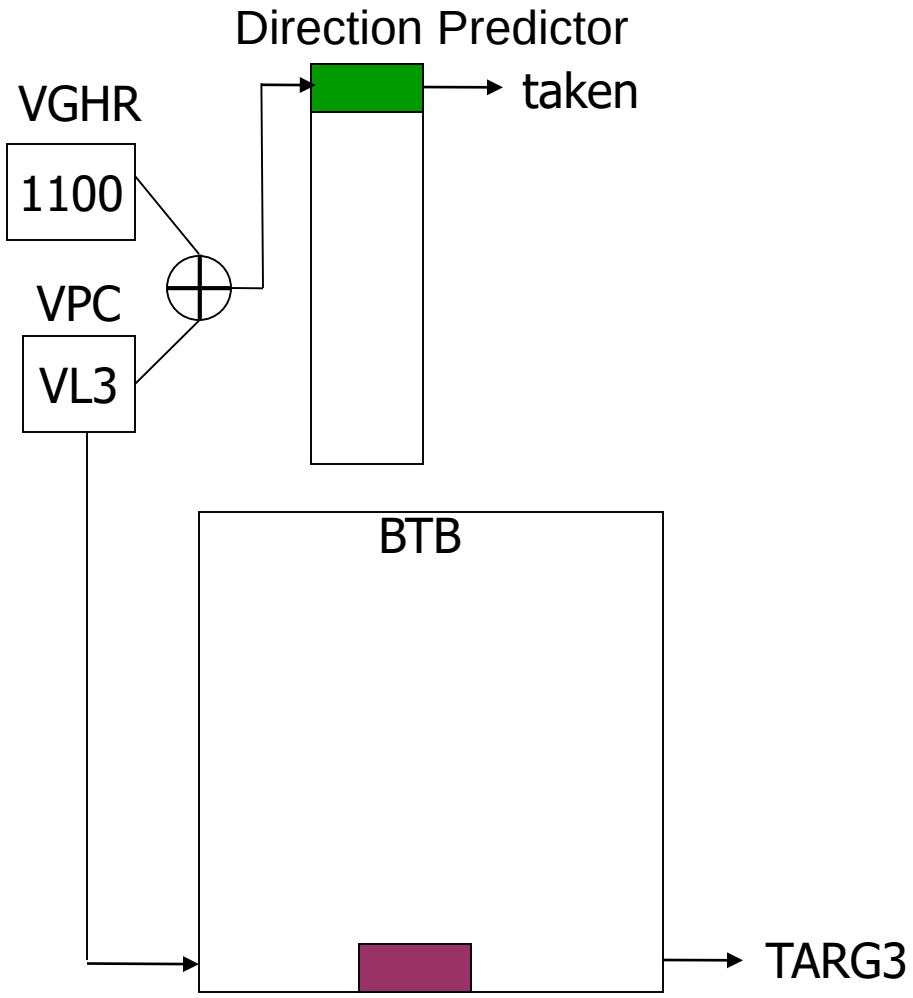
cond. jump TARG1 // VPC: L

cond. jump TARG2 // VPC: VL2

cond. jump TARG3 // VPC: VL3

cond. jump TARG4 // VPC: VL4

Predicted Target = TARG3



VPC Prediction (IV)

- Advantages:
  - + High prediction accuracy (>90%)
  - + No separate indirect branch predictor
  - + Resource efficient (reuses existing components)
  - + Improvement in conditional branch prediction algorithms also improves indirect branch prediction
  - + Number of locations in BTB consumed for a branch = number of target addresses seen

- Disadvantages:
  - -- Takes multiple cycles (sometimes) to predict the target address
  - -- More interference in direction predictor and BTB

Issues in Branch Prediction (I)

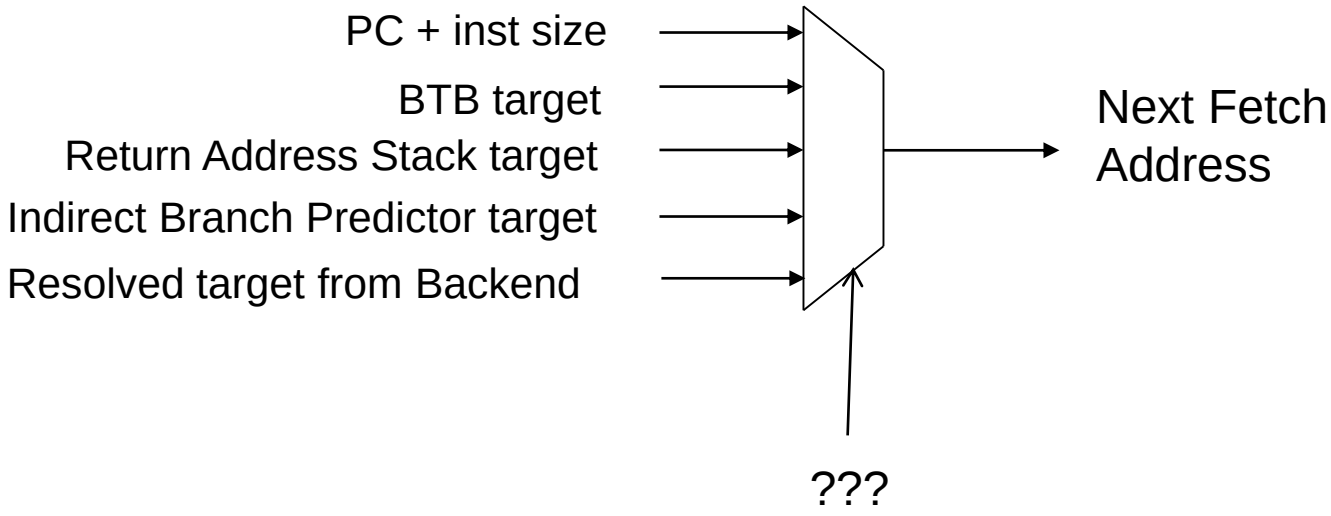
- Need to identify a branch before it is fetched

- How do we do this?
  - BTB hit → indicates that the fetched instruction is a branch
  - BTB entry contains the "type" of the branch
  - Pre-decoded "branch type" information stored in the instruction cache identifies type of branch

- What if no BTB?
  - Bubble in the pipeline until target address is computed
  - E.g., IBM POWER4

Issues in Branch Prediction (II)

- Latency: Prediction is latency critical
  - Need to generate next fetch address for the next cycle
  - Bigger, more complex predictors are more accurate but slower



Computer Architecture

Lecture 11: Control-Flow Handling

Prof. Onur Mutlu
ETH Zürich

Fall 2017

26 October 2017

We did not cover the following slides in lecture. These are for your preparation for the next lecture.

More on Wide Fetch Engines and Block-Based Execution

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)

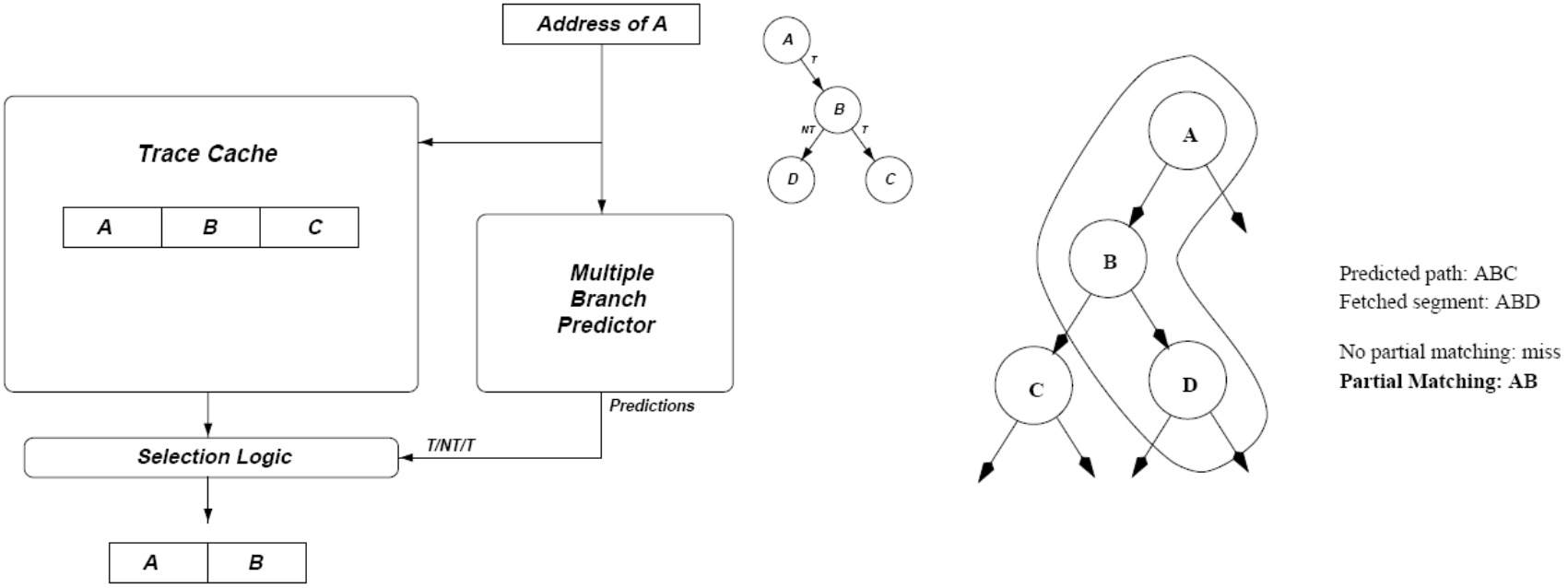


Figure 6.1: The trace cache and branch predictor are indexed with the address of block A. The inset figure shows the control flow from block A. The predictor selects the sequence ABD. The trace cache only contains ABC. AB is supplied.

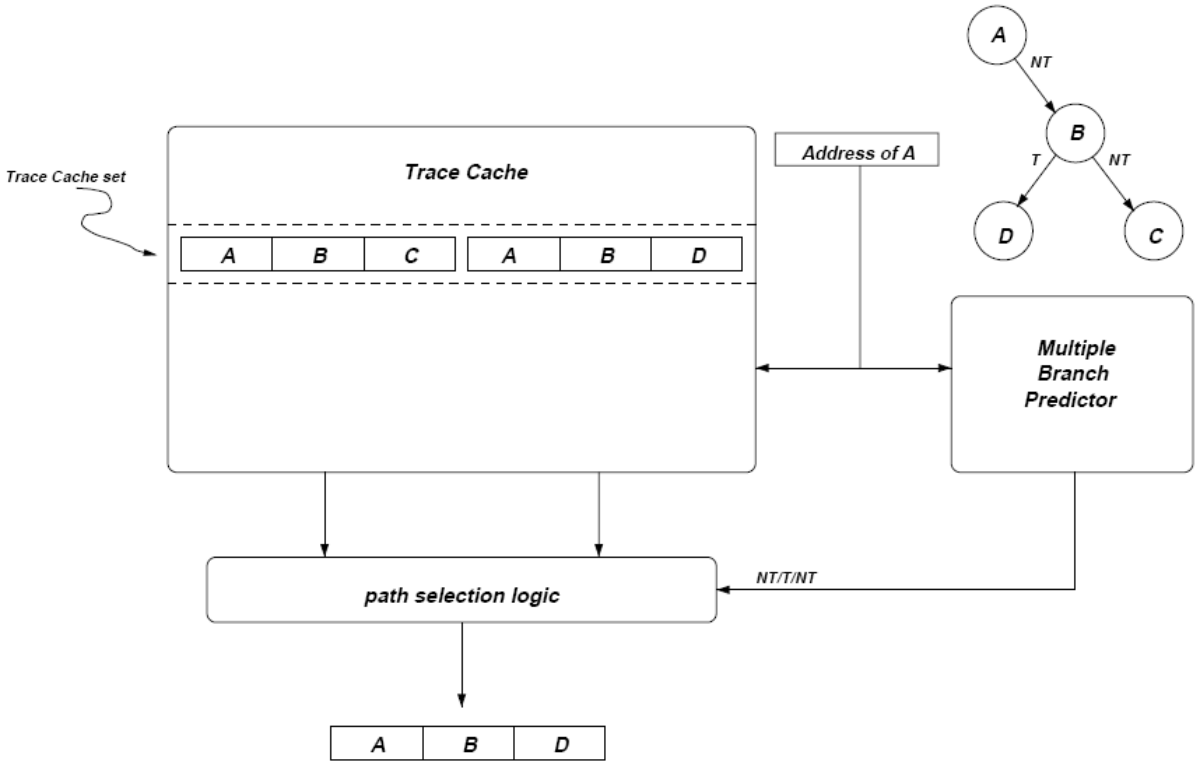
Trace Cache Design Issues (III)

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

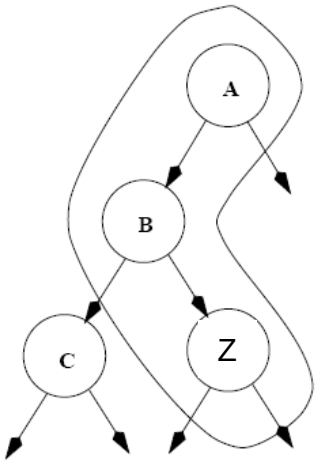
+ Good for traces with unbiased branches (e.g., ping pong between C and D)

-- Need to determine longest matching path

-- Increased cache pressure



Trace Cache Design Issues (IV)



Predicted path: ABC
Fetched segment: AB 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

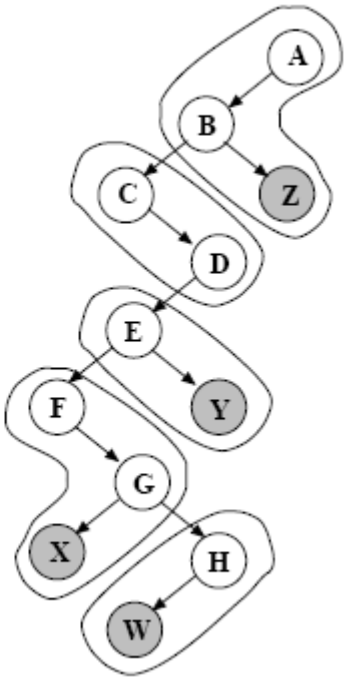
+ Reduces basic block duplication in trace cache

-- Slightly more complex scheduling/branch resolution

-- Some instructions not dispatched/flushed

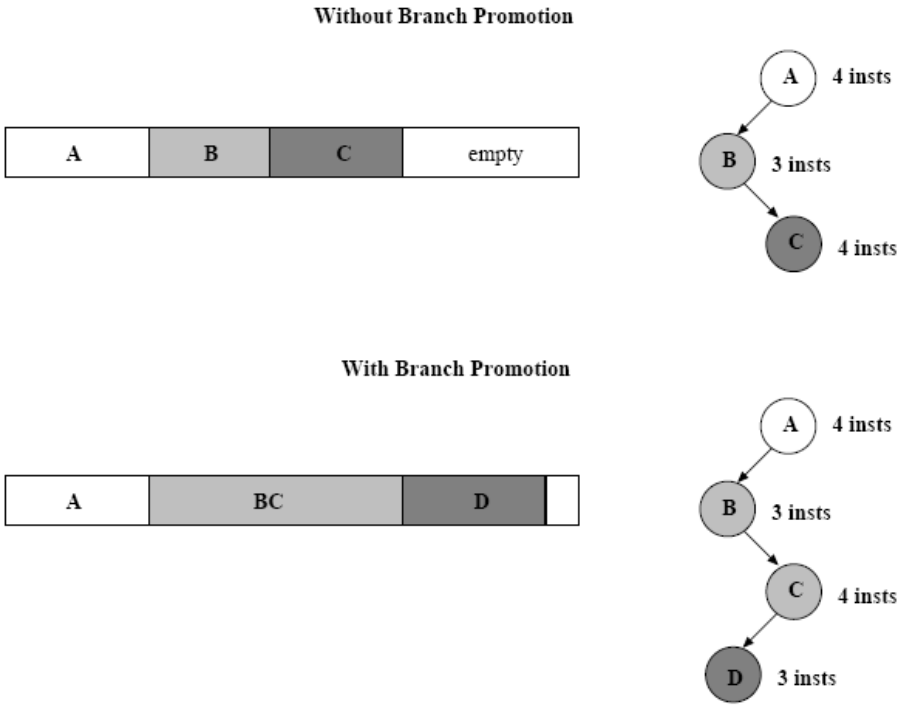
Instruction Window

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

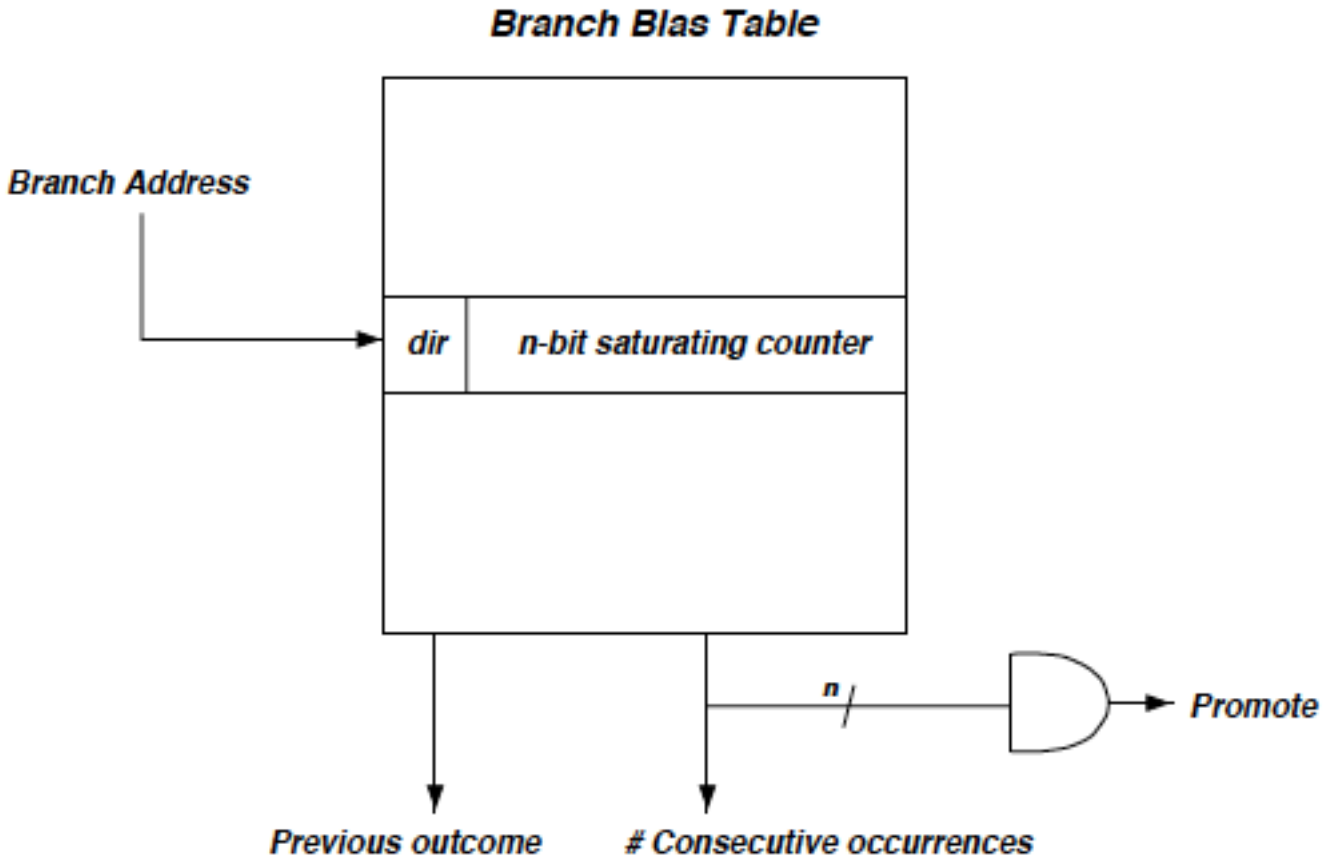
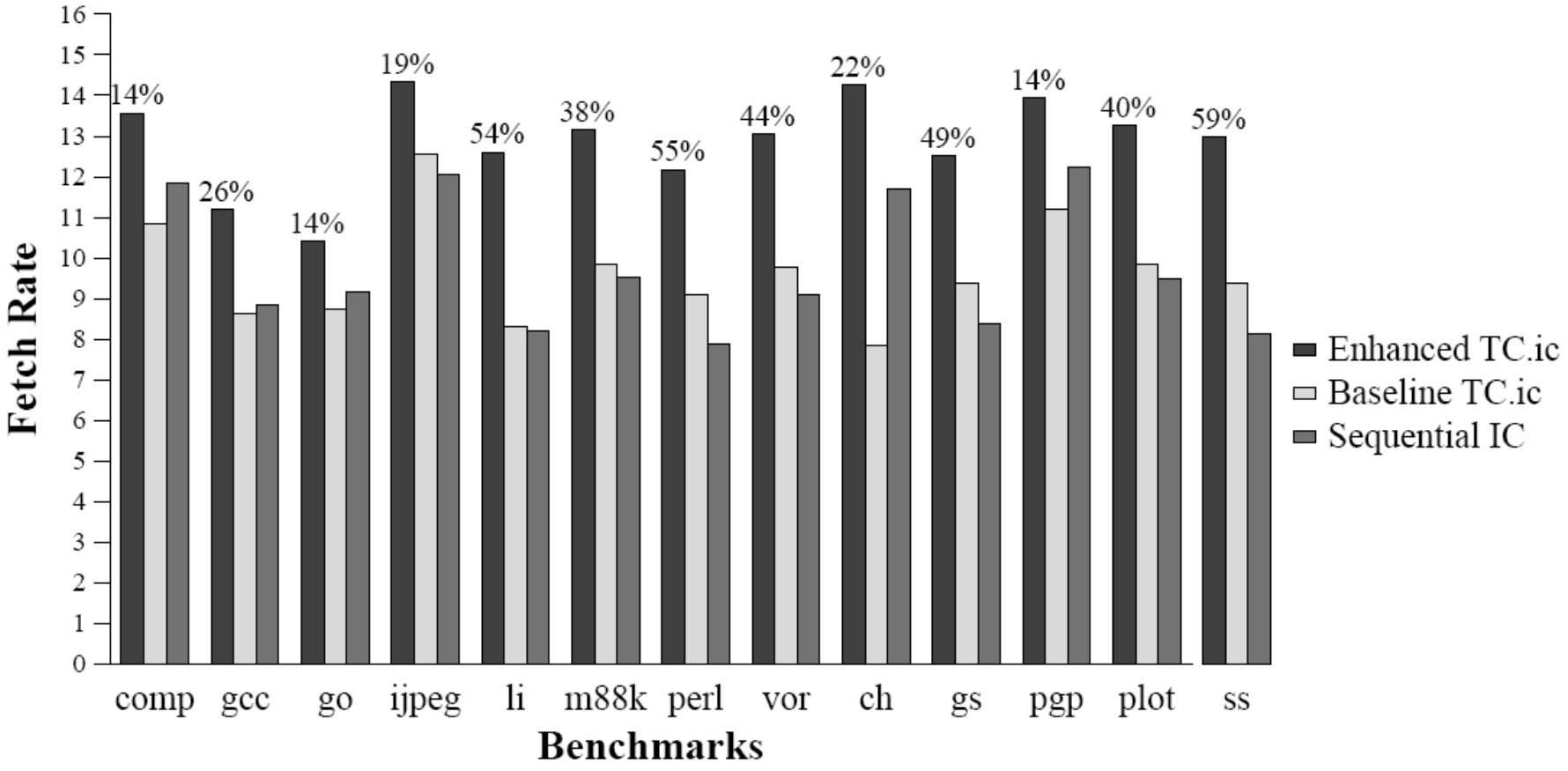
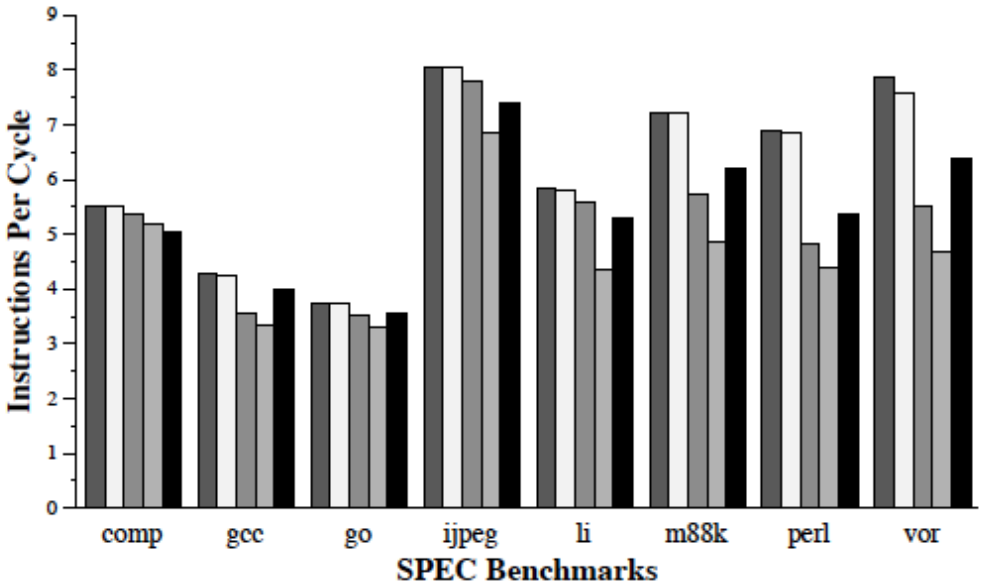


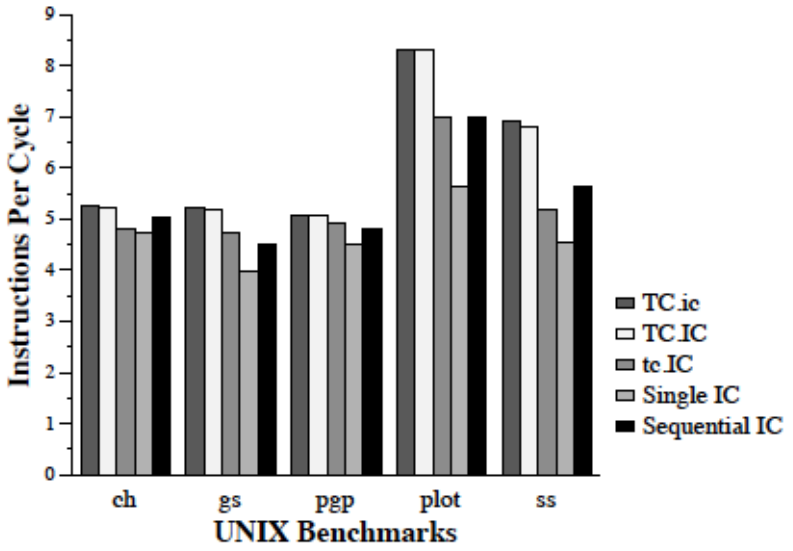
Figure 6.19: Diagram of the branch bias table.

Effect on Fetch Rate



Effect on IPC (16-wide superscalar)



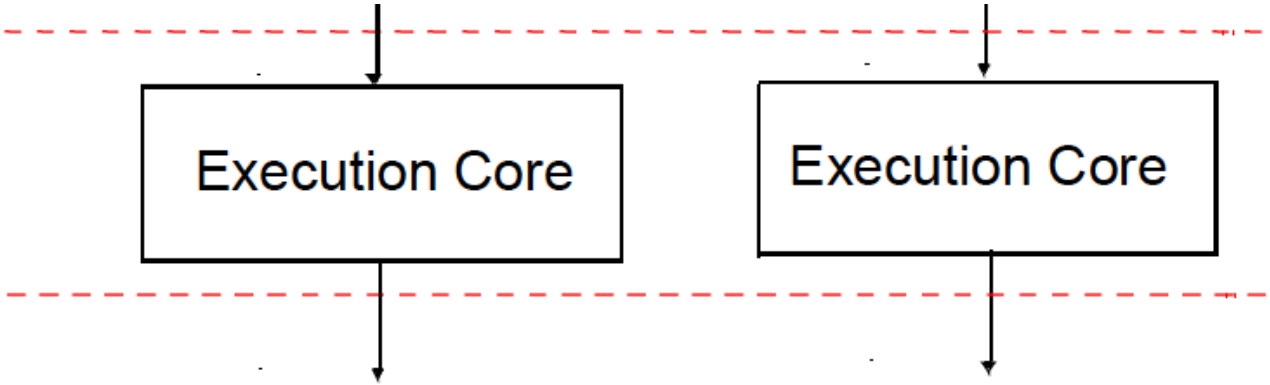


Configuration Name	TCache Size	ICache Size	Blocks per Fetch	Br Pred Type	BTB Size
TC.ic	128KB	4KB	3	Multiple	1KB
TC.IC	64KB	64KB	3	Multiple	8KB
tc.IC	4KB	128KB	3	Multiple	16KB
Single	–	128KB	1	Hybrid	20KB
Sequential	–	128KB	3	Multiple	16KB

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

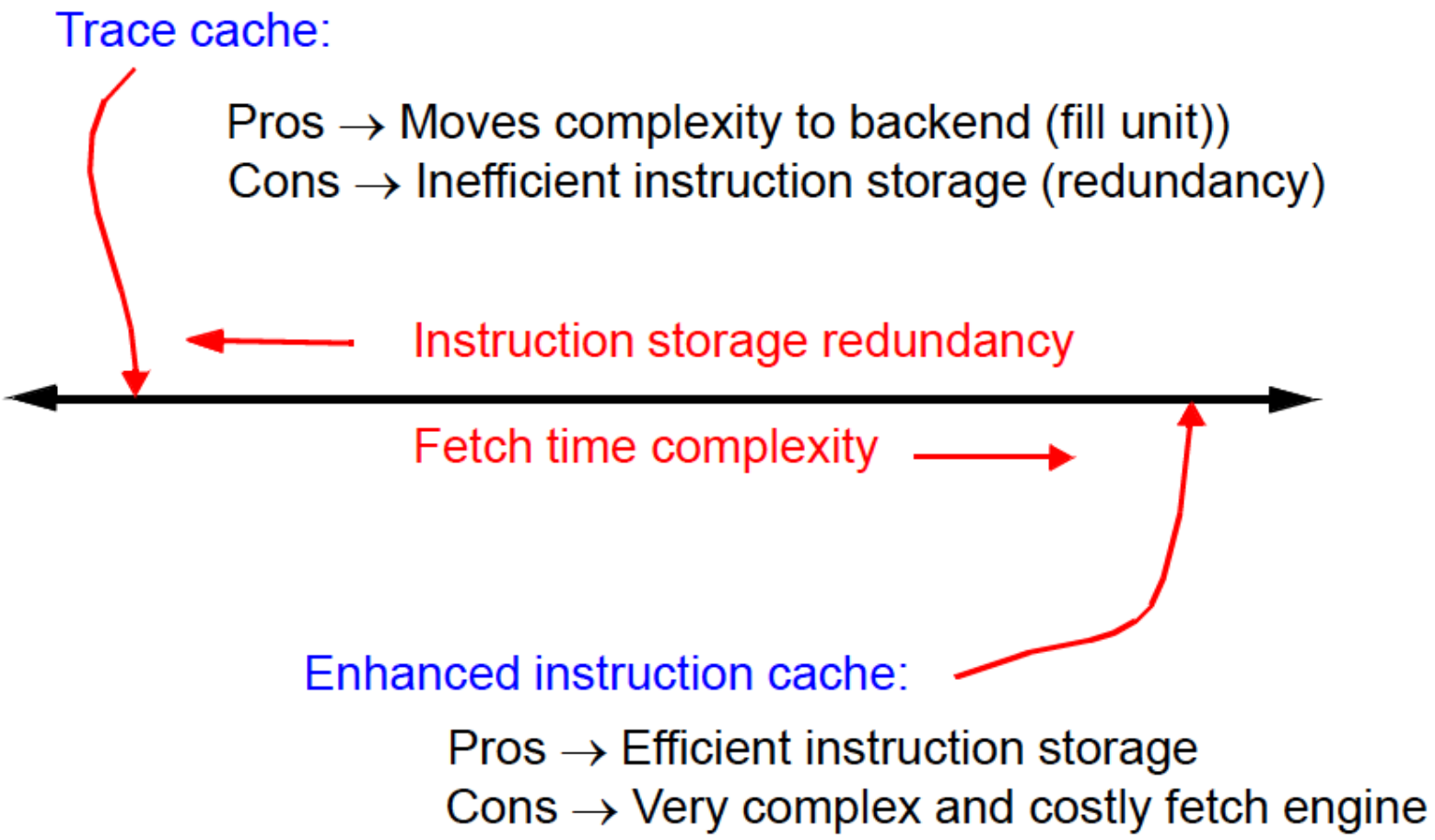
Enhanced I-Cache vs. Trace Cache (I)

	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



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

Enhanced I-Cache vs. Trace Cache (II)



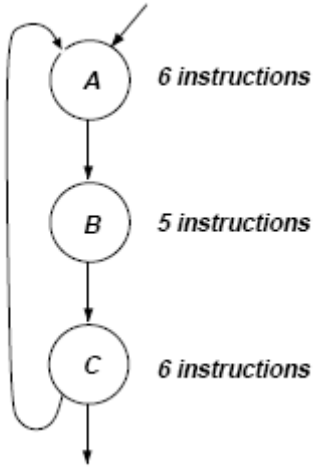
Frontend vs. Backend Complexity

- 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

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 (Black et al., ISCA 1999)
- 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)

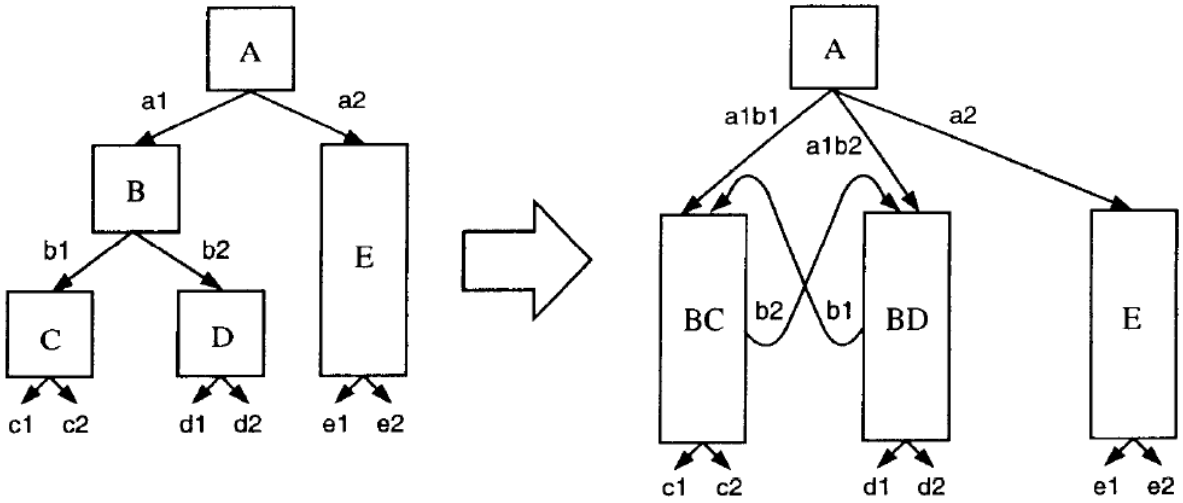
-- Block table needs to be multiported to obtain multiple blocks per cycle

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.

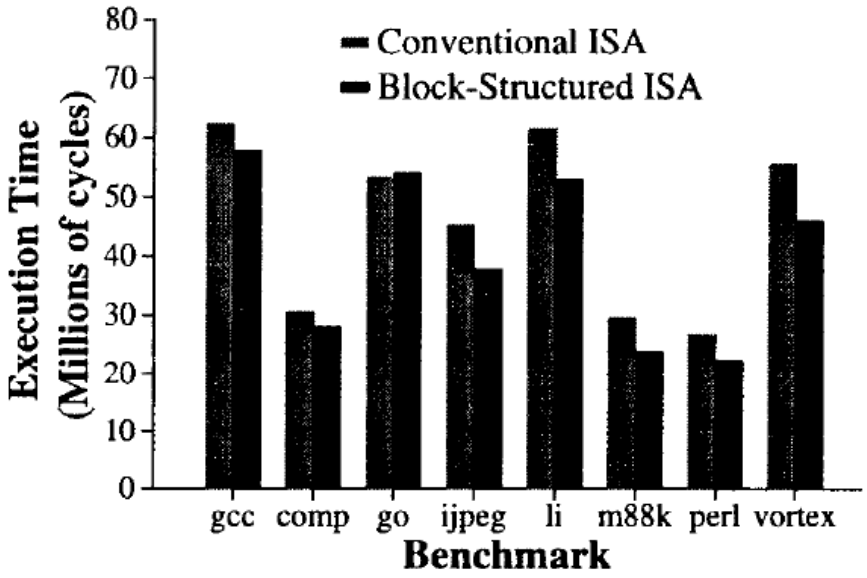


Figure 3. Performance comparison of block-structured ISA executables and conventional ISA executables.

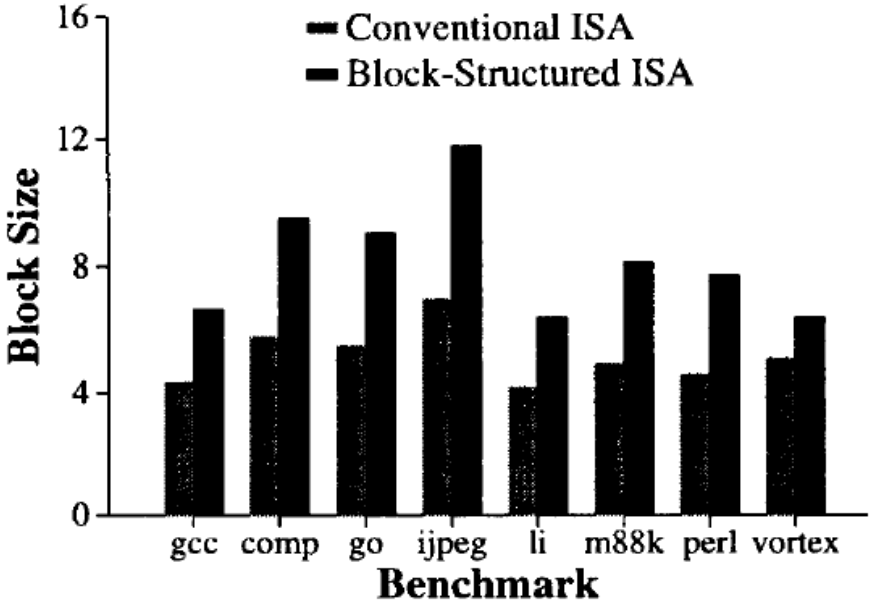


Figure 5. Average block sizes for block-structured and conventional ISA executables.

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