Speculative Execution

Instruction stream speculation
- Predict program behavior before execution flow determined
- Execute PREDICTED program flow
- Verify program results before commitment to state
- Recover from MISPREDACTION

Branch prediction
- Predict target address and action (taken/not-taken)

Instruction / data cache prefetch
- Load instructions and data to L1 cache before cache miss

Trace cache
- Cache frequent instruction flow sequences
- Span multiple control blocks

Branch predication
- Execute multiple branch flow alternatives
- Commit only verified instruction flow

Branch Behavior
if-then-else

```
<table>
<thead>
<tr>
<th>PC</th>
<th>Instruction</th>
</tr>
</thead>
<tbody>
<tr>
<td>PC+4</td>
<td>fall-through</td>
</tr>
<tr>
<td>PC+4+offset</td>
<td>target</td>
</tr>
<tr>
<td></td>
<td>continue</td>
</tr>
</tbody>
</table>
```

Speculation in dynamic scheduling
At PC < PC_{BEQZ}
- Prediction → branch taken
- Issue instructions target, continue, ...
- Cannot retire until PC > PC_{BEQZ}

At PC = PC_{BEQZ}
- Verify BEQZ
- Continue or correct misprediction

Basic Branch Prediction

Branch predictor in fetch + decode stage

<table>
<thead>
<tr>
<th>Index</th>
<th>Source identifier (hash of instruction PC)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Entry</td>
<td>Branch outcome history</td>
</tr>
<tr>
<td></td>
<td>Prediction (target address, taken/not-taken) by some algorithm</td>
</tr>
</tbody>
</table>

Branch Predictor
- PC → BHR → prediction + target address
- Predict taken ⇒ PC ← target address
- Predict not taken or miss in BHR ⇒ PC ← fall-through address

Branch validation (verification)
- Complete detailed branch execution
- Compare with prediction
- Correct prediction ⇒ continue program flow
- Misprediction ⇒ flush pipeline ⇒ restart with correct target address
- Update history table
Branch Prediction for DLX Pipeline

1. Branch predictor in IF stage
   Identifies branch instruction
   According to source address
   Predicts branch from branch history
   
   - Taken
     Predicts branch target address
     Uses fall-through address
   
   - Not-taken
     Predicts branch target address
     Uses fall-through address

2. Validate branch instruction in ID stage
   Usual Calculation:
   Target address
   Condition flag — taken or not-taken

3. After validation
   Update branch predictor
   Target address
   Branch history
   Taken/not-taken

Simple Branch Prediction Performance for Loop

Simple static loop

```plaintext
ADDI R1, R0, #N ; N iterations
L1: SUBI R1, R1, #1
    ALU Block
    BNEZ R1, L1

< B-2 lines of ALU code >
BNEZ R1, L1

R1 = N-1

< B-2 lines of ALU code >
BNEZ R1, L1

R1 = N-2

< B-2 lines of ALU code >
BNEZ R1, L1

R1 = 0
```

Branch Statistics

- Prediction Logic
- Branch Statistics
  - Branch: taken / not-taken
  - Target PC

Dynamic Branch Prediction

Past behavior predicts future branch behavior

- Assumes repetitive/cyclical behavior in branches

Possible approaches

- Global prediction
  - Predict from recent execution pattern of all recent branches
- Local prediction
  - Predict from recent execution pattern of individual branch

Combination of global and local prediction

Limitations

- Complex hardware
- Large branch history register storage
- Long prediction time

Importance of Branch Hazards

Branch instructions ⇒ program control flow

Loop instructions account for most run time

Branch statistics

- Branch ≈ 20% — 25% of instructions
- ≈ 2/3 of conditional branches are taken

Branch verification (correct execution) = N clock cycles

- N wrong instructions ⇒ pipeline before branch verification
- N wrong instructions must be flushed
- N wasted clock cycles

\[
CPI_{\text{stall}} = \frac{N \text{ wasted clock cycles}}{\text{taken branches}} \times \frac{1 \text{ taken branch}}{6 \text{ instructions}} = \frac{N \text{ wasted clock cycles}}{6 \text{ instructions}}
\]

N ≥ 1 for DLX ⇒ CPI_{\text{stall}} ≥ 1/6
One-Level Prediction

Branch History Register
- Index — hash function of branch PC Entry
- n-bit saturating up/down counter
- Target — PC of last taken branch

Prediction Logic
- n-bit counter ⇒ 2^n levels
  000...0 = Strongly Not Taken
  111...1 = Strongly Taken
- Prediction
  0xxxx...x = not taken
  1xxxx...x = taken

1-Bit Saturating Up/Down Counter

1-bit confidence level
- Remembers last action only
- Initialize counter ← 0
- Taken counter++
- Not Taken counter--

Prediction based on last action
- Counter = 0 ⇒ not taken
- Counter = 1 ⇒ taken
- Predicts single static loops with accuracy of 1 / (loop iterations)

2-Bit Saturating Up/Down Counter

2-bit confidence level
- Remembers up to 3 actions

After branch validation
- Taken ⇒ counter++
- Not Taken ⇒ counter--

Prediction
- Counter = 0x ⇒ not taken
- Counter = 1x ⇒ taken
- Predicts 2 nested loops

Initialize to 00

Alternative 2-Bit Counters

2 consecutive taken branches
- Predict strongly taken

2 consecutive not taken branches
- Predict strongly not taken
Short Nested Loop

**Branch profile — global branch actions**

1 = taken
0 = not taken

11...10111...10111...101...11...10111...100

**Example**

N = 3 → global profile

123456789ABC

11011101100

**Counter outputs**

- Branch 1-bit 2-bit 3-bit 4-bit
  - init 0 0 0 0 0
  - 1 1 01 001 0001
  - 2 1 10 010 0100
  - 3 0 01 001 0001
  - 4 1 10 010 0010
  - 5 1 11 011 0011
  - 6 1 11 100 0100

**Branch 1-bit 2-bit 3-bit 4-bit**

- Branch 7 0 10 011 0011
- Branch 8 1 11 100 0100
- Branch 9 1 11 101 0101
- Branch A 1 11 110 0110
- Branch B 0 10 101 0101
- Branch C 0 01 100 0100

**1-bit mispredictions**

1 + (N - 2) × 2 + 1

**2-bit mispredictions**

2 + (N - 2) × 1 + 2

**Large N**

- N for 2-bit
- ~ 2N for 1-bit

---

**Yeh-Patt Two-Level Adaptive Prediction**

- **Branch History Register (BHR)**
  - Profile — sequence of observed branch actions
  - Target — PC of last taken branch
  - Characterizes history of branch behavior

- **Pattern History Table (PHT)**
  - Pattern — observed next branch action for specific profile
  - Characterizes pattern of branch behavior given particular history

- **Prediction**
  - Confidence level = n-bit saturating up/down counter

**Yeh and Patt, Alternative Implementations of Two-Level Adaptive Branch Prediction, 1992**

---

**Short Nested Loop**

- ADDI R1, R0, #N
- L1: ADDI R2, R0, #N
- L2: SUBI R2, R2, #1
- BNEZ R2, L2
- SUBI R1, R1, #1
- BNEZ R1, L1

**Counters at predict time**

Based on previous Action

<table>
<thead>
<tr>
<th>Branch PC</th>
<th>Profile</th>
<th>Target PC</th>
</tr>
</thead>
<tbody>
<tr>
<td>R1 (at L2)</td>
<td>R2 (at L2)</td>
<td>Action</td>
</tr>
<tr>
<td>N</td>
<td>N</td>
<td>T</td>
</tr>
<tr>
<td>N</td>
<td>N - 1</td>
<td>T</td>
</tr>
<tr>
<td>N</td>
<td>N - 2</td>
<td>T</td>
</tr>
<tr>
<td>N</td>
<td>1</td>
<td>NT - T</td>
</tr>
<tr>
<td>N - 1</td>
<td>N</td>
<td>T</td>
</tr>
<tr>
<td>N - 1</td>
<td>N - 1</td>
<td>T</td>
</tr>
<tr>
<td>N - 1</td>
<td>N - 2</td>
<td>T</td>
</tr>
<tr>
<td>N - 2</td>
<td>N</td>
<td>T</td>
</tr>
<tr>
<td>N - 2</td>
<td>N - 1</td>
<td>T</td>
</tr>
<tr>
<td>N - 2</td>
<td>N - 2</td>
<td>T</td>
</tr>
<tr>
<td>N - 2</td>
<td>1</td>
<td>NT - T</td>
</tr>
<tr>
<td>1</td>
<td>N</td>
<td>T</td>
</tr>
<tr>
<td>1</td>
<td>N - 1</td>
<td>T</td>
</tr>
<tr>
<td>1</td>
<td>N - 2</td>
<td>T</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>NT - NT</td>
</tr>
</tbody>
</table>

**Yeh-Patt — BHR**

- **Index**
  - Global table
    - 1 entry
    - No index
  - Local table
    - 1 entry per branch
      - Index = hash function of branch source PC
      - Source address aliasing ⇒ instruction misidentified ⇒ misprediction

- **Entry**
  - Profile
    - k-bit bitmap = last k actions of b last branches
    - Taken = 1 / Not Taken = 0
    - Shift in newest action from right
  - Target — PC of last taken branch

**profile**

01110...10011

← Not Taken

↑ Target
Index
k-bit profile
Branch action sequence

Entry
s-bit bitmap = last s (out of b) next actions for profile
Pattern

Prediction Logic
n-bit saturating up/down counter — counts s-bit pattern from PHT

Profile Pattern

Pattern History Table
pattern
s bits

s-bit bitmap = last s (out of b) next actions for profile

Shift in newest action from right

Prediction Logic
n-bit saturating up/down counter — counts s-bit pattern from PHT

000...0 = Strongly Not Taken — 111...1 = Strongly Taken
Prediction

0xxxx...x = not taken
1xxxx...x = taken

Yeh-Patt — Simple Example
Profile for short nested loop with N = 3
k = 2 branch actions (PHR)
s = 1 next action (BHT)
1-bit counter (prediction = last action for pattern)

Actual branch sequence — 110111011100

Yeh-Patt — Building BHR and PHT
Branch actions for some branch instruction

ADDI R1, R0, #N
L1: ADDI R1, R0, #N
BNEZ R2, L2
L2: SUBI R2, R1, #1
BNEZ R1, R1, #1

Prediction
1x = taken
0x = not taken

Yeh-Patt — Alternating Branches 1-bit and 2-bit counter
ADD R1, R0, R0
L1: ADDI R1, R1, #1
ANDI R2, R1, #1
BNEZ R2, L1
BEQZ R2, L1

Separate Counters for BNEZ and BEQZ

Misprediction rate for BNEZ
1-bit counter — 100%
2-bit counter — 50%
Example — Alternating Branches with Y-P

ADD R1, R0, R0
L1: ADDI R1, R1, #1
ANDI R2, R1, #1
BNEZ R2, L1
BEQZ R2, L1

4-bit BHR
2-bit PHT for each BHR pattern

<table>
<thead>
<tr>
<th>BHR</th>
<th>PHT</th>
</tr>
</thead>
<tbody>
<tr>
<td>0000</td>
<td>00</td>
</tr>
<tr>
<td>0001</td>
<td>00</td>
</tr>
<tr>
<td>0010</td>
<td>00</td>
</tr>
<tr>
<td>0011</td>
<td>00</td>
</tr>
<tr>
<td>0100</td>
<td>01</td>
</tr>
<tr>
<td>0101</td>
<td>01</td>
</tr>
<tr>
<td>0110</td>
<td>01</td>
</tr>
<tr>
<td>0111</td>
<td>01</td>
</tr>
<tr>
<td>1000</td>
<td>10</td>
</tr>
<tr>
<td>1001</td>
<td>10</td>
</tr>
<tr>
<td>1010</td>
<td>10</td>
</tr>
<tr>
<td>1011</td>
<td>10</td>
</tr>
<tr>
<td>1100</td>
<td>11</td>
</tr>
<tr>
<td>1101</td>
<td>11</td>
</tr>
<tr>
<td>1110</td>
<td>11</td>
</tr>
<tr>
<td>1111</td>
<td>11</td>
</tr>
</tbody>
</table>

Steady state action for BNEZ
Correctly predicts every branch from 8th

Steady state sequence for BNEZ
Alternate between 0101 00 and 1010 11

Steady state YP misprediction rate for BNEZ — 0%

Another Example

Branch actions for some branch instruction

k = 5-bit BHR
s = 3-bit PHT

Misprediction rate
10101 50%
other 0%

DEC Alpha Branch Predictor — 1

Two-Level Local Predictor — 2

Index: Branch source address bits 2 to 11 (10 bits ⇒ 2^10 = 1024 entries)
Instruction addresses aligned on 4-byte (2-bit) boundaries

VPC[11:2]
Local History Table
1K x 10

Entry = last 10 actions of branch

Index: Last 10 actions of branch
Alias if two branches have same
10-bit action history
Entry: 3-bit state (111) to (000)

Local Branch Prediction
Predictor = (0xx) ⇒ not-taken
Predictor = (1xx) ⇒ taken

Choice Predictor — 4

Last 12 actions of all branches
2^{12} = 4096 possible action histories

Global Path History

Choice Predictor
4K x 2

Prediction Logic

Global Predictor — 3

Last 12 actions of all branches
2^{12} = 4096 possible action histories

Index: Last 12 actions of all branches
Entry: 2-bit state (11) to (00)

Global Branch Prediction

Prediction: Predictor = (0x) ⇒ not-taken
Predictor = (1x) ⇒ taken

Yeh-Patt Versus Alpha

Yeh-Patt model
Stage 1 — store k-bit profile (taken / not taken history) in BHR
Stage 2 — store s-bit pattern (next action history) in PHT
Stage 3 — count over stored pattern

Alpha model

Local
Stage 1 — store k = 10 bit profile
Stage 2 — count over all next actions for profile

Global
Stage 1 — store k = 12 bit profile
Stage 2 — count over all next actions for profile
Choice predictor — local / global
**Event Horizon in Counter**

**Event horizon**
Prediction on s-bit pattern from PHT ≠ counting all next actions

**Example**
Branch history = 11 11110 11110 11110 11110...

Next actions for profile 11

|   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 1 | 1 | 1 | 1 | 1 | 0 | 1 | 1 | 1 | 0 | 1 | 1 | 1 | 0 | 1 | 1 | 1 |
| 1 | 1 |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |
| 1 | 1 |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |
| 1 | 1 |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |
| 1 | 1 |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |

**Compare**
2-bit counter on all next actions
2-bit counter on 2-bit pattern

|   |   |   |   |   |   |   |   |
|---|---|---|---|---|---|---|
| 1 | 1 | 1 | 0 | 1 | 1 | 0 |

<p>| | | | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
</tbody>
</table>

<p>| | | | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>

<p>| | | | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>

**Address Prediction**

**Branch Target Buffer (BTB)**
Organized as cache with LRU eviction
Integrated with action predictor or separate table

**Index**
Branch PC

**Entry**
Last verified target address
Valid
Initialized to 0
Changed when entry is written
Special
Indicates low confidence in predicted address

**DLX Pipeline with Y-P Branch Prediction**

**Low Confidence Address Predictions**

**Indirect branches**

<table>
<thead>
<tr>
<th>x86</th>
<th>JMP [ESI]</th>
<th>EIP ← Mem[Regs[ESI]]</th>
</tr>
</thead>
<tbody>
<tr>
<td>DLX</td>
<td>JR R1</td>
<td>PC ← Regs[R1]</td>
</tr>
</tbody>
</table>

Pentium M caches target for indirect branches
Target prediction made from cache of register instead of last jump

**Return from subroutine**
Function call, trap, interrupt
Lowest confidence branch addresses
Enter subroutine from multiple departure points
Each subroutine call has private return point

<table>
<thead>
<tr>
<th>x86</th>
<th>CALL [ESI]</th>
<th>PUSH EIP EIP ← Mem[Regs[ESI]]</th>
</tr>
</thead>
<tbody>
<tr>
<td>RET</td>
<td>POP EIP</td>
<td></td>
</tr>
<tr>
<td>DLX</td>
<td>JALR R2, offset</td>
<td>Regs[R2] ← PC</td>
</tr>
<tr>
<td>JR R2</td>
<td>PC ← Regs[R2]</td>
<td></td>
</tr>
</tbody>
</table>


Predicting Subroutine Return Addresses

**x86 Call/Return instructions**

**CALL** pushes return address onto stack
First run of **RETURN** misses in branch history cache
Store entry after verification with Special Field
Subsequent runs of **RETURN** pre-load stack entry for target address

**RISC JALR/JR instructions**

Recognize subroutine call
Encounter instruction **JR R2**
**R2** points to instruction immediately below **JALR R2, offset**
Mark **JALR** instructions as Probable Call in BTB entry
Create BTB entry for **JALR** target (from subroutine)
Mark entry as Subroutine Entry Point (SEP)
Target address = pointer to **JR** return instruction
On **JALR** hit at PC
Target field of **JR** pointed to by SEP ← PC + 4
Returning **JR** has correct target address

Exotic Branch Mispredictions

**Exotic branch conditions**

Simple branch prediction has accuracy ≈ 90%
Typically
80% of branches ~ 100% accuracy
20% of branches ~ 50% accuracy

**Least predictable branches — on-time test of single operand**

**x86** — **CMP EAX, immediate** followed by **JZ offset**
**DLX** — **BEQZ R1, offset**

**Special treatment for tested operands (proposed)**

Maintain cache of tested operands and test
Entry for **BEQZ R1** caches value of **R1** and output of (**R1 == 0**)
On update to operand that hits in table
Update table entry
Update action prediction for associated branch instruction

CPU and Memory Hierarchy

**CPU controller accesses L1 cache**

if (address in L1 cache) {access performed in 1 clock cycle}
else {

L1 cache accesses cache controller
cache controller initiates access to L2 and main memory
if (address in L2 cache) {controller copies contents to L1 from L2}
else {controller copies location to L1 from main memory}
}

**Cache miss penalty**

Address not in L1 ⇒ delay in memory access
Cache Organization — 1

Copy memory data to cache
- How much data to copy?
- Which data is in cache?
- How to find data in cache?

Copy DATA LINE (block) to cache
- Line = B bytes
- Page size (virtual memory) = integer × block size

Sets and Slots
- Cache = S sets
- Set = W slots
- Slot = 1 data line

Copy memory line to:
- Specific Set
- Any Slot in Set

W-Slots in Set

Cache Organization — 2

Physical address space
- \( N = 2^n \) bytes

Lines
- \( B = 2^b \) bytes / line
- Line number = \( \text{int} \left( \frac{\text{Address}}{B} \right) \)

Sets
- \( S = 2^s \) sets
- Set index = s bits
- Line → line number % S

W-way associativity
- Line → any slot in set
- Identify line by Tag field in cache directory

Total Cache Size
- total cache size = \( S \times W \times B \)

Cache Definitions and Policies

Cache hit
- CPU accesses line in cache

Cache miss
- CPU accesses line not in cache

Read miss
- Load line to cache

Write miss
- Write allocate — load line to cache
- No write allocate — write to main memory (no load to cache)

Swapping out cache line
- Need new line in full set
- Remove least recently used (LRU)
- WRITE BACK — update RAM on line swap-out
- WRITE THROUGH — update RAM on every write to cache

Cache Performance Issues

L1 Cache hit
- CPU memory access = 1 clock cycle

L1 Cache Miss
- CPU stalls — L1 loads missing line

Miss Rate
- Cache misses per cache accesses
- Instruction fetch, load, store
- Depends on cache size and organization

Miss Penalty
- Cache update latency
- Stall cycles while cache ← missing line
**Performance of Two-Level Unified Cache**

**Parameters**
- \( M^I_{L1} \): Instruction Miss Rate at L1
- \( 1 - M^I_{L1} \): Instruction Hit Rate at L1
- \( M^C_{L1} \): Data Miss Rate at L1
- \( 1 - M^C_{L1} \): Data Hit Rate at L1
- \( M_{L2} \): Miss Rate at L2
- \( 1 - M_{L2} \): Hit Rate at L2

- \( P_{L1} \): Miss Penalty at L1
- \( P_{L2} \): Miss Penalty at L2

- \( L1 \) Instruction Access per Instruction = 1
- \( L1 \) Data Access per Instruction = \( \frac{IC \cdot data \ access}{IC} \)

\[ CPi^{miss}_{cache} = \text{cache miss stall cycles per instruction} = \left[ M^I_{L1} + M^C_{L1} \times \frac{IC \cdot data \ access}{IC} \right] \times \left( P_{L1} + P_{L2} \times M_{L2} \right) \]

**Issues Affecting Miss Rate**

**Compulsory miss**
- Demand access update
- Cache ← line on first access
- First access to line ⇒ cache miss
- Not affected by cache size or associativity

**Capacity miss**
- Swap out lines for required load ⇒ miss
- Cache < main memory
- Larger cache ⇒ lower capacity miss rate

**Conflict miss**
- Swap out lines for required load ⇒ miss
- Line ← specific set
  - Spare capacity in other sets
  - Example — address aliasing
- Higher associativity ⇒ lower conflict miss rate

**1-Level Cache versus 2-Level Cache**

**1 level cache**

\[ CPI_{1 \text{ level}}^{all} = \left[ M^I_{L1} + M^C_{L1} \times \frac{IC \cdot data \ access}{IC} \right] \times \text{Penalty}_{L1-to-RAM} \]

50 clock cycles

**2 level cache**

\[ CPI_{2 \text{ level}}^{all} = \left[ M^I_{L1} + M^C_{L1} \times \frac{IC \cdot data \ access}{IC} \right] \times \left( \text{Penalty}_{L1-to-L2} + M_{L2} \times \text{Penalty}_{L2-to-RAM} \right) \]

- 5 clock cycles
- \(< 0.01\)
- 50 clock cycles
- \(< 5.5\) clock cycles

**Miss Rate**

![Miss Rate Chart]

- 1-way
- 2-way
- 4-way
- 8-way
- Capacity Misses

**Total miss rate drops as capacity or associativity increases**
**Associativity Trade-Off**

Any slot in set ← line
- Find line ⇒ search all tags
- Larger associativity ⇒ more blocks per set ⇒ longer search time

For fixed cache capacity = S × W × B
- Larger associativity W
  ⇒ fewer sets S
  ⇒ smaller set index s
  ⇒ larger tag size n – (s + b)
- Longer tag search

Small advantage beyond 4-way associativity

<table>
<thead>
<tr>
<th></th>
<th>s</th>
<th>n – (s + b)</th>
<th>Tag Search Length</th>
</tr>
</thead>
<tbody>
<tr>
<td>Direct mapped</td>
<td>n – b</td>
<td>0</td>
<td>2^0 = 1</td>
</tr>
<tr>
<td>Fully associative</td>
<td>0</td>
<td>n – b</td>
<td>2^n – b</td>
</tr>
</tbody>
</table>

**Tag Search Alternatives**

**Serial access**
- Address → tag controller
- Hit ⇒ memory ← set number / way number
- Prepare tagged line for access
- Longer latency at lower power

**Parallel access**
- Address → tag controller + set controller
- Prepare set for access
- Hit ⇒ memory ← way number
- Shorter latency at higher power

**Way prediction**
- Address fed to tag controller and way predictor
- Hit ⇒ way predictor feeds set number and way number to memory
- Misprediction ⇒ tag controller feeds correct way number on next CC

**Cache Prefetching**

Prefetching "hides memory latency"
- Predict cache update before demand miss
- Initiate cache update request
- Bring memory block to cache before needed
- Memory latency does not cause delay

**Data cache prefetching**
- Useful for sequential accesses to large data structures
  - Software prefetch
    - Assembly language prefetch instructions for cache update
    - Inserted by programmer or compiler
  - Hardware — stride buffer predicts sequential accesses

**Instruction cache prefetching**
- Hardware-based prefetch mechanisms
- Predict instruction cache updates based on program structure

**Data Cache Latency Example**

Loop over large data structure
- Accesses 1024 x 1024 SSE (16-byte vector) operands = 16 MB
- Sequential vector reads — no repeated access to same data

```
for (i = 0; i < 1024; i++) {
    for (j = 0; j < 1024; j++) {
        SSE_vector_operation a[i][j];
    }
}
```

Pentium 4 with 16 KB L1 cache and 64-byte cache line
- Data miss every 64 bytes = 4 SSE operands
- Data miss in L1 1024 x 1024 / 4 = 256K compulsory misses
Software prefetch loads 128 bytes (2 cache lines)
128 bytes = (128/16) = 8 SSE operands
Prefetch 8 operands forward

```
for (i = 0; i < 1024; i++) {
    for (j = 0; j < 1024; j++) {
        prefetch a[i][j+8];
        SSE_vector_operation a[i][j];
    }
}
```

Prefetch filtering
NOP on prefetch of cache hit

```
pipeline  4 operations
       |         4 operations
       |          
I/O bus  | cache update | cache update
```

Data Cache Prefetch Instruction

General Stream Buffer

Stream
Successive accesses to sequential cache lines

Stream buffer
FIFO buffer
Generates + buffers update requests for successive cache lines
Request buffer implementation
Requested line → cache from higher memory level
Line buffer implementation
Requested line → stream buffer until demand miss

Request filtering
Stream buffer skips lines already in cache to minimize prefetches

Stream Buffer Operation

Cache miss to data line L on CC k

<table>
<thead>
<tr>
<th>CC</th>
<th>Request Buffer</th>
<th>Line Buffer</th>
</tr>
</thead>
<tbody>
<tr>
<td>k</td>
<td>Stream buffer allocated to missed line L</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Buffer generates, buffers, and issues cache request for line L</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Pipeline suffers miss penalty for line L</td>
<td></td>
</tr>
<tr>
<td>k + 1, 2, 3, ...</td>
<td>Stream buffer generates, FIFO-buffers, and issues cache request for lines L + 1, 2, 3, ...</td>
<td></td>
</tr>
<tr>
<td>k + miss penalty</td>
<td>Line L brought to cache</td>
<td></td>
</tr>
<tr>
<td>k + miss penalty + 1</td>
<td>Line L + 1 brought to cache</td>
<td></td>
</tr>
<tr>
<td>k + miss penalty + 2</td>
<td>Line L + 2 brought to cache</td>
<td></td>
</tr>
</tbody>
</table>

No further miss penalties
Stream buffer issues sequential requests until full or re-allocated to new miss

Stream Buffer Example

Lines in cache

<table>
<thead>
<tr>
<th>CC</th>
<th>Line Accessed</th>
<th>Cache</th>
<th>Stream Buffer Slots</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>2</td>
<td>Hit</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>3</td>
<td>Hit</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>4</td>
<td>Miss</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>4</td>
<td>Miss</td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>4</td>
<td>Miss</td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>4</td>
<td>Miss</td>
<td></td>
</tr>
<tr>
<td>6</td>
<td>4</td>
<td>Miss</td>
<td></td>
</tr>
<tr>
<td>10</td>
<td>4</td>
<td>Hit</td>
<td></td>
</tr>
<tr>
<td>11</td>
<td>5</td>
<td>Hit</td>
<td></td>
</tr>
</tbody>
</table>

Stream Buffer Slots

<table>
<thead>
<tr>
<th>Line</th>
<th>Cache</th>
<th>Stream Buffer Slots</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
<td>2  3  4  5</td>
</tr>
<tr>
<td>6</td>
<td>4</td>
<td>5  7  9  10</td>
</tr>
<tr>
<td>10</td>
<td>4</td>
<td>5  7  9  10</td>
</tr>
<tr>
<td>11</td>
<td>5</td>
<td>4  5  7  9  10</td>
</tr>
</tbody>
</table>
**Stride Buffer**

Sequential accesses to lines $L$, $L+s$, $L+2s$, $L+3s$, …

**Stride**

Access stride = $s$

**Stride buffer**

Variation of stream buffer

Prefetches lines according to stride $s$

Stream buffer allocated following 2 sequential cache misses

---

**Out-of-Order Fetch**

**Miss Information Status History Register (MSHR)**

Variation of stream buffer for instruction cache

Additional random access buffer for instruction blocks

**On cache miss**

Allocate MSHR slot to missed address

Requests cache update for missed address

Writes pointer for missed addresses in Instruction Queue

PC advances

$L1$ hits $\rightarrow$ Instruction Queue

$L1$ misses $\rightarrow$ Instruction Queue on update

---

**Stream Buffer for DLX Data Cache**

---

**MSHR Example**

**Blocks in cache**

<table>
<thead>
<tr>
<th>CC</th>
<th>Block in PC</th>
<th>Cache</th>
<th>MSHR Slots</th>
<th>Instruction Queue</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>2</td>
<td>Hlit</td>
<td>0 1 2</td>
<td>2</td>
</tr>
<tr>
<td>1</td>
<td>3</td>
<td>Hlit</td>
<td>0 1 2 3</td>
<td>3</td>
</tr>
<tr>
<td>2</td>
<td>4</td>
<td>Miss</td>
<td>4</td>
<td>P4</td>
</tr>
<tr>
<td>3</td>
<td>5</td>
<td>Miss</td>
<td>4 5</td>
<td>P4 P5</td>
</tr>
<tr>
<td>4</td>
<td>6</td>
<td>Hlit</td>
<td>4 5</td>
<td>P4 P5 6</td>
</tr>
<tr>
<td>5</td>
<td>7</td>
<td>Miss</td>
<td>4 5 7</td>
<td>P4 P5 6 P7</td>
</tr>
<tr>
<td>6</td>
<td>8</td>
<td>Hlit</td>
<td>4 5 7</td>
<td>P4 P5 6 P7 8</td>
</tr>
<tr>
<td>50</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>51</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Fetch Directed Instruction Prefetching

**Branch prediction → cache prefetch**
FIFO between branch predictor and L1 instruction cache

**Branch detected at address PC**
Branch prediction
Predicted next PC → address queue
Address queue initiates tag check
Cache update on miss
L1 prefetches missed line before access to fetch address

![Diagram](attachment:image.png)

**Fetch Directed Prefetching for DLX**

![Diagram](attachment:image.png)