Presentation 4

Speculative Execution

ביצוע ספקולטיבי מתייחס לשיטות לקביעת תחזית (ניחוש מושכל) על ההתנהגות העתידית של תהליך עיבוד כדי למנוע או להסתיר latency. השימוש בשיטות לזיהוי תבניות מאפשר למעבד עכשווי לדייק בחיזוי התנהגות פקודות סיעוף (branch) ועדכון מטמון (cache) — שני מקורות עיקריים לשיהוי בריצת תוכניות. בנוסף, היכולת לחזות התנהגות מבני בקרה מורכבים מאפשרת ביצוע פקודות באופן ספקולטיבי גם להשיג סיום מוקדם של משימות וגם להעלות את ניצולת המשאבים במעבד.

Slide 2 **Speculative Execution**

Instruction stream speculation refers to methods of program execution that:

1. Predict program behavior before the actual execution flow is precisely determined,
2. Execute the predicted program flow,
3. Verify the execution flow and program results before committing results to state,
4. Recover from misprediction.

The most common method of speculation is **branch prediction**, in which the results of a branch instruction, the branch **action** (taken/not-taken) and the **target** address, are predicted by hardware. The prediction is based on pattern recognition techniques applied to branch statistics dynamically stored in the CPU during program execution. Modern processors can achieve a 90% correct prediction rate or better, depending on the type of code.

A related method is instruction and data **cache prefetch**. Normally, cache updates are performed on a cache miss — this involves a cache penalty (delay) while the update is processed. Prefetch permits identification of instruction and data flows and enables speculative cache updates and prevent the cache penalty.

A more advanced method is **trace cache**, which caches frequent instruction flow sequences that can span multiple control blocks. When running complex loop oriented program sequences, trace cache can improve performance of repeated branch sequences.

Branch predication is a replacement for branch instructions in many situations, especially if-then-else structures. Instead using conditional branch instructions to jump over instructions, instructions can be predicated, that is, assigned a predicate flag set by various program conditions. The predicate flag enables conditional execution — the instruction is either executed or cancelled. It is possible to execute multiple branch flow alternatives and commit to architectural state only the correctly verified instruction flow.

Slide 3 **Branch Behavior**

The typical **if-then-else** structure is illustrated in the following fragment:

PC (address)	Instruction
PC_{BEQZ}	`BEQZ REG, offset`
PC_{BEQZ} + 4	`fall-through`
	…
PC_{BEQZ} + 4 + offset	`target`
	`continue`

The conditional branch BEQZ implements:
if (REG == 0)
then PC + 4 ← PC + 4 + offset ⇒ program control continues at the target instruction
else the program continues from the fall-through instruction at PC + 4

In dynamic scheduling, instructions are executed out-of-order, but results are committed to state in program order as indexed by the PC of the original program listing (slide 46 in presentation 3). In principle, out-of-order execution could create a **barrier at branch instructions**, because it is not known whether to execute the target or fall-through until the branch is correctly evaluated. A barrier would mean that instructions following a branch could never be executed out-of-order (before the branch itself), even when they are marked ready.

Speculation removes the barrier in the following way:

When PC < PC_{branch} (meaning that the branch has not yet been evaluated and committed to state), the CPU predicts the branch action (taken or not-taken) and the target address (used if taken). Instructions are decoded to the ROB, executed in the EUs and results are returned to the ROB according to the prediction. These speculative instructions are kept in the ROB and marked as **speculative** — they cannot be retired (committed to state) until the branch is **verified**.

The verification of the branch occurs when PC = PC_{branch} and the **branch result is committed**. After verification, the speculative instructions in the ROB are marked as **valid** (for a correct prediction) or cancelled. Valid instructions are then retired in program order.

Slide 4 Basic Branch Prediction

Branch prediction requires two new hardware elements in the fetch+decode stage of the pipeline:

The first element is a **Branch History Register** (BHR). The index is the source identifier (PC) of a particular branch instruction in the program. The entry consists of the branch action history and a branch prediction made by some algorithm. The branch prediction includes the target address and the branch action (taken or not-taken).

The second element is **Branch Predictor** logic that enters the PC of each branch instruction into the BHR and updates the instruction fetch stage with the PC for fetches past the PC of the branch. Generally, the PC of the branch is ahead of the PC used by the retirement unit, and behind the PC used by the instruction fetch unit. For example, at any moment it may be that

1. The retirement unit has committed instructions up to PC = 100,
2. There are finished instruction in the ROB instructions up to PC = 150 and unfinished instructions in the ROB up to PC = 250,
3. A branch instruction was found in the fetch stream at PC = 200,
4. The fetch unit is currently fetching instructions from PC = target = 750. These instructions are marked speculative.

The branch is ready for execution when all of its sources (the condition to be checked) are ready. When the branch is evaluated, the result is compared with the prediction and the branch history is updated in the BHR. If the prediction is correct, program flow continues. Otherwise, there was a **misprediction** — the speculative instructions are now marked invalid. The invalid locations in the ROB are now available for filling by new instructions fetched from the address found in evaluating the branch instruction.

Slide 5 – 7 Branch Prediction for DLX Pipeline

Slide 5 shows a simple branch prediction system that could be added to the 5 stage DLX pipeline. The branch prediction logic and BHR are located in the IF stage. A branch instruction is fetched and identified on CC1, and a prediction is made for the next instruction — target or fall-through. On CC2, IF fetches from the predicted address and ID evaluates the branch. The BHR is updated in the IF stage. If the prediction is correct, the IF stage continues fetching instructions from the predicted location. On a misprediction, the predicted instruction is flushed (marked a NOP, introducing a pipeline bubble) and the correct instruction is fetched on CC3.

Slide 6 shows a simple example for the DLX with simple branch prediction. The main loop consists of a block of ALU instructions followed by a branch. On the first loop iteration, the branch is loaded for the first time and has no history in the BHR. In this case, IF must predict the fall-through instruction (it has no other options, since the target address is still unknown). When the branch evaluates in stage ID, the BHR is updated, the fall-through is flushed and the target is loaded. The pipeline flush causes a stall of 1 CC. At the end of the second iteration, the branch is predicted as taken and the target address in the BHR is used. This prediction behavior continues until the last iteration. On the last iteration, the branch is predicted as taken, but that is now a misprediction. The BHR is updated, the target instruction is flushed and the fall-through instruction is loaded after a 1 CC stall. After N iterations the number of instructions is $N \times B + 2$ (the 2 are the initial ADDI and the fall-through that runs once) so $CPI^{stall} = 2 / (N \times B + 2) \rightarrow 0$ for large N. On programs of this simple type, branch prediction is very effective, removing the 1 CC branch stall on almost every loop iteration.

In the typical case, branch stalls cause significant performance degradation. Most program run time is devoted to control loops, with branch instructions about 20% to 25% of all instructions run. Since about 2/3 of all branches are taken, in the worst case taken branch instructions make up $1/4 \times 2/3 = 1/6$ of all instructions. If the branch penalty (the number of instructions to be flushed after a misprediction), then CPI^{stall} must be at least $N / 6$. Even with $N = 1$ on the DLX, this large value of CPI^{stall} limits performance.

Slide 8 Dynamic Branch Prediction

General **Dynamic Branch Prediction** uses pattern recognition techniques to predict future branch behavior based on past behavior. The simplest techniques involve one-level prediction, in which the recent branch history leads directly to a prediction based on the relative number of taken and not-taken results. Two-level prediction examines recent history to find patterns in the branch behavior that followed prior events matching some particular branch history. The patterns can be global (predicting from the recent execution pattern of all recent branches), local (predicting from the recent execution pattern of each individual branch), or some combination.

Branch prediction is limited by the complex hardware required to store and analyze the long-term behavior of many branch instructions. The BHR can become very large making both the time to learn the behavior and the time to make a prediction long.

Slide 9 One-Level Prediction

In **one-level prediction**, the Branch History Register (BHR) is a table whose index is a hash function of the branch source PC (the address of the branch instruction in memory). The entry consists of an n-bit saturating up/down counter and the branch target (the target PC of last taken branch).

A saturating up/down counter is an n-bit register initiated to 0. On each taken branch the counter is incremented by 1, but saturates at 11...111 (the value $2^n - 1$). On each not-taken branch the counter is decremented by 1, but saturates at 00...000.

The branch prediction is taken from the value of the high order bit. The branch will only be predicted as taken when the counter is at least 10...000 (the value 2^{n-1}), following about 2^{n-1} taken branches.

Slide 10 1-Bit Saturating Up/Down Counter

A 1-bit counter is either 0 or 1. Its prediction is always equal to the previous branch action. It can predict a single static loop with an accuracy of about 1 / (loop iterations). On an alternating taken / not-taken branch it will mispredict the branch action 100% of the time.

Slide 11 – 12 2-Bit Saturating Up/Down Counter

The 2-bit counter counts from 0 to 3. It can predict a pair of nested loops with some accuracy. The prediction states are called

00 — strongly not-taken (a strong prediction of not-taken)

01 — weakly not-taken (a weak prediction of not-taken)

10 — weakly taken (a weak prediction of taken)

11 — strongly taken (a strong predication of taken)

Alternative 2-bit counters are shown on slide 12.

The first counter predicts strongly taken after 2 consecutive taken branches and predicts strongly not-taken after 2 consecutive not-taken branches.

The second counter also predicts strongly taken after 2 consecutive taken branches, but counts down more gradually. This counter favors a prediction of taken, consistent with realistic branch statistics.

Most research suggests that neither alternative counter significantly improves prediction performance.

Slide 13 – 14 Short Nested Loop

Slide 13 shows the outputs of 1, 2, 3, and 4 bit counters for 12 iterations of a short nested loop, based on global branch behavior.

The inner loop (in blue) runs N times in index R2 per outer loop. The outer loop (in green) runs N times in index R1. The general pattern of taken and not-taken branches is

$$\underbrace{11\ldots101}_{1}\underbrace{11\ldots101}_{2}\underbrace{11\ldots101}_{3}\ldots\underbrace{11\ldots101}_{N-1}\underbrace{11\ldots100}_{N}$$

where 1 represents a taken branch and 0 represents a not-taken branch.

For N = 3, the counter outputs are shown in the table. The upper bit of the counter value is the prediction for the next branch run (the row just below the value).

The 4-bit counter never makes a prediction of taken.

The 3-bit counter is initialized to 000, which mispredicts not-taken for the first iteration of the inner loop. The taken branch on the first iteration raises the counter to 001, which mispredicts not-taken for the second iteration. Similarly the taken branch on the second iteration raises the counter to 010, which correctly predicts not-taken for the third iteration. The not-taken branch on the third iteration lower the counter to 001, which mispredicts not-taken for the outer loop. Continuing this analysis, the 3-bit counter mispredicts most branch iterations.

Slide 14 compares the branch predictions for the 1 and 2 bit counters with the actual branch actions. The two left columns show the value of registers R1 and R2 when the program executes the instruction at label L2.

The 1-bit counter initializes to 0 and mispredicts the first branch (mispredictions are shown in a red square). The first branch raises the counter to 1, which correctly predicts the next N – 1 taken branches. The last inner loop branch is mispredicted and the not-taken lowers the counter to 0 which mispredicts the taken branch in the outer loop. The total number of mispredictions for the 1-bit counter is $1 + (N - 1) \times 2 + 1 = 2N$.

The 2-bit counter initializes to 0 and mispredicts the first two branches (this represents the longer learning curve for the wider counter). It correctly predicts the next N – 1 taken branches, but mispredicts the last iteration of the inner loop. The two bit counter weakly (but correctly) predicts taken for the branch in the outer loop. The total number of mispredictions for the 2-bit counter is $2 + (N - 1) \times 1 + 2 = N + 3$, which is half the value of the 2-bit counter.

Most 1-step branch predictors use a 2-bit counter.

Slide 15 – 17 Yeh-Patt Two-Level Adaptive Prediction

The **Yeh-Patt** method for two-level branch prediction is implemented in Intel x86 CPUs. As in one-level prediction, the index to the Branch History Register (BHR) is a hash of the source PC of a branch instruction. But in this method, the BHR does not count recent branch actions — it stores the **profile** (the actual sequence of recent branch actions), along with the target address of the last taken branch. The stored profile is used as the index to a Pattern History Table (PHT), whose entry is called a **pattern** — the history of next branch actions that followed a specific profile of branch actions. The PHT entry characterizes the pattern of branch behavior when it has repeated a given branch history. The prediction logic feeds the pattern sequence into an n-bit saturating counter — the prediction is the high order bit.

The profile stored in the BHR is a k-bit bitmap shifted in from the right. If a certain branch has been observed (executed and verified) b times, then the bitmap consists of k bits (1 for taken and 0 for not-taken) starting on the right with result b, then (moving left) result b – 1, b – 2, and so on down to bit b – k – 1 on the left. The profile records the history of k branch results — the 6-bit bitmap 011101 records a branch that was not-taken, then taken 3 times, then not-taken, and then taken.

For each profile, the PHT records the next observed action. For example, if the profile 011101 points to the 5-bit PHT entry 00101, this says that on the last 5 times that the branch had behaved as **011101**, its subsequent actions were **0**, and then **0**, and then **1**, and then **0**, and then **1**. This records 5 branch actions from the history (distributed among many other results):

0111010, **0111010**, **0111011**, **0111010**, **0111011**

The prediction for the next result is made by entering the pattern into an n-bit counter. The output of a 2-bit counter (initialized to 00) from the pattern **00101** is

00	→	00	→	00	→	01	→	00	→	01
init	**0**		**0**		**1**		**0**		**1**	

The prediction based on this pattern is not-taken.

Slide 18 Yeh-Patt — Simple Example

The short nested loop with N = 3 produces the global sequence of branches **110111011100**. A 2-bit global BHR is initialized to zero and new branch actions are shifted in from the right, as shown in the table columns labeled **Global Profile m**. The columns labeled **Observed Profiles** show the 2-bit combinations that have appeared in the BHR up to each branch event. The columns labeled **Actual Next Action** lists the observed result that follows each of the observed profiles.

For example, the BHR is initialized to 00. One the first execution of a branch, the action is taken and so a 1 is shifted into the BHR from the right — the new BHR entry is 01. At this point, 2 separate profiles have been observed — 00 and 01. The branch action following the appearance of profile 00 is 1, and the branch action following the appearance of profile 01 is also 1. This 1 is shifted into the BHR from the right, producing entry 11 (also a new profile appearing for the first time). The action of the next branch after the appearance of 11 in BHR is 0.

The prediction is a 1-bit counter (last action). On initialization, no next action for the profile 00 has been recorded in the PHT and so there is no prediction (predict not-taken). The next action following 00 is 1, and on the next branch following the pattern 00 (if there is one), the prediction will be 1. The next action after the appearance of the profile 01 is 1 — on the next branch following the pattern 01 the prediction will be 1.

No actions have been observed for the profiles 00, 01, 11, 10 in the first 3 branches. No predictions are possible for these branches — they are predicted not-taken. Only one prediction is correct (at random), because the third branch result is actually not-taken.

For the next branches in the program, observed next actions are available for all 4 possible profiles. On the 4th branch, the profile in the BHR is 01. The entry in the BHT for index 01 is 1, and so this branch is correctly predicted as taken. The next 2 branches cannot be correctly predicted with this configuration because the profile 11 is followed by 0 half the time and followed by 1 half the time. The last branch is mispredicted because the program ends.

Two basic observations are (1) short loops are difficult to predict, and (2) 1-bit counters do not predict accurately.

Slide 19 shows an example of how the BHR and PHT are constructed dynamically.

Some branch instruction executes an alternating sequence of 12 taken / not-taken actions, 101010101010. The 5-bit BHR is initialized to 00000. It is convenient to present the sequence of branches in a table, in order to identify the profile in the BHR Entry and the actual branch (next) that will be executed following the observed profile.

						Branch Number													
						1	2	3	4	5	6	7	8	9	10	11	12		
	Init					1	0	1	0	1	0	1	0	1	0	1	0	BHR Entry	Next
0	0	0	0	0	0													00000	1
1		0	0	0	0	1												00001	0
2			0	0	0	1	0											00010	1
3				0	0	1	0	1										00101	0
4					0	1	0	1	0									01010	1
5						1	0	1	0	1								10101	0
6							0	1	0	1	0							01010	1
7								1	0	1	0	1						10101	0
8									0	1	0	1	0					01010	1
9										1	0	1	0	1				10101	0
10											0	1	0	1	0			01010	1
11												1	0	1	0	1		10101	0
12													0	1	0	1	0	01010	—

Each row shows the 5-bit entry in the BHR found from shifting in from the fright the most recent branch action. After branch 1, the action 1 is shifted in from the right, producing the profile 00001. The actual action of the next branch (branch 2) is 0. There are 12 branches in the profile, so there is no action after the 12th branch.

The PHT is initialized to 000 for all profiles. The PHT entry for profile 01010 evolves as:

After Observed Branch	Pattern			
1	0	0	0	01010 not yet observed
2	0	0	0	
3	0	0	0	
4	0	0	0	
5	0	0	1	01010 followed by 1
6	0	0	1	
7	0	1	1	01010 followed by 1
8	0	1	1	
9	1	1	1	01010 followed by 1
10	1	1	1	
11	1	1	1	01010 followed by 1
12	1	1	1	

The observed profiles and observed next actions after all 12 branches are:

Profile				
00000	0	0	1	These profile are observed only 1 time
00001	0	0	0	
00010	0	0	1	
00101	0	0	0	
01010	1	1	1	This profile is observed 5 times after branches 4, 6, 8, 10, 12
10101	0	0	0	This profile is observed 4 times after branches 5, 7, 9, 11

Since the pattern for 10101 remains unchanged (000), the entries in the PHT are stable after branch 8.

Using a 2-bit counter to predict the next branch, the prediction for the profile 10101 will always be correct — the PHT entry is 000 and the next branch not-taken is correctly predicted. For the profile 01010, the PHT entry used to predict branch 9 is 011, so the branch is correctly predicted as taken. Similarly, the PHT entry used to predict branch 11 is 111, and the branch is correctly predicted as taken.

Slide 20 – 22 Example — Alternating Branches

Slide 20 shows the performance of one step prediction with 1-bit and 2-bit counters (per branch) on a sequence of alternating branches. The BNEZ in the inner loop is taken on alternate iterations, generating a sequence of alternating taken / not-taken branches. The BEQZ in the outer loop is only reached when BNEZ falls through and so it is always taken.

The 1-bit counter always predicts the previous result, so that the 1-bit counter for the BNEZ inner loop is always mispredicted. The 1-bit counter for the BEQZ outer loop is initialized to 0 and updates to 1 after the first branch, providing a correct prediction on branch 4 and onward.

The 2-bit counter value for the BNEZ inner loop alternated between 00 and 01. It always predicts not-taken and is correct half the time (equivalent to the outcome of no branch prediction at all). The 2-bit counter value for the BEQZ outer loop is initialized to 00 and updates to 01, 10, 11. It provides a correct prediction on branch 6.

Slide 21 shows the evolution of a 4-bit BHR and 2-bit PHT in two-step prediction for the same sequence of BNEZ results. The grayed area shows the learning stage for the branch predictor. The required branch profiles 0101 and 1010 are observed on the 3^{rd} and 4^{th} branches — the 2-bit PHT registers hold valid observations after 7 branches. Branch 8 and onward will be correctly predicted for the BNEZ.

Slide 22 shows the program and BHR / PHT evolution. Initialized zeros are shown in gray. Observed next branches (the BNEZ result in that row) are shown in blue — these values are used to predict the next occurrence of the profile. The right column (Y-P) shows the prediction for the branch in that row.

Slide 23 Example — Another Example

Slide 23 shows a more irregular example. The BHR shows 17 profiles (after the learning phase) and the PHT shows patterns for 7 observed profiles (after the learning phase). Except for the profile 10101, all profiles will produce correct predictions.

Slide 24 – 29 DEC Alpha Branch Predictor

Slides 24 to 29 show the branch predictor in the Alpha RISC CPU designed by the Digital Equipment Corporation (DEC) as a RISC evolution from the VAX. It was manufactured until 2007. In its time, the Alpha was the fastest CPU available. In the Alpha, a local predictor records the last 10 actions of each branch instruction (indexed by its PC). This branch profile is the index of a local predictor (a 3-bit saturating counter). The highest order bit of the counter forms the prediction. The global path history records the last 12 actions of all branches. This global profile is used as the index for two separate units: a global predictor (a 2-bit saturating counter), and a choice predictor whose entry is a 2-bit counter of correct predictions made by the global predictor. When the counter in the choice predictor is 00 or 01, the result of the local branch predictor is used for the prediction. When the counter in the choice predictor is 10 or 11, the result of the global branch predictor is used for the prediction.

Slide 28 compares the Yeh-Patt two-step prediction with the Alpha prediction. Aside from the Alpha's use of local and global prediction (Y-P is always local or global), the main difference is the way branches are counted. In the Yeh-Patt method, the most recent s-bit pattern is sent to the counter — any information from before s instances of a profile is lost (this is called an **event horizon**). In the Alpha, the counter is continuously updated with the most recent result, so that the counter output is based on the results of all previous instances of a given profile.

The first table on slide 29 shows (in red) the next actions following the profile 11 in the branch sequence 11 11110 11110 11110 11110... The initial 1 bits sets a bias in the data.

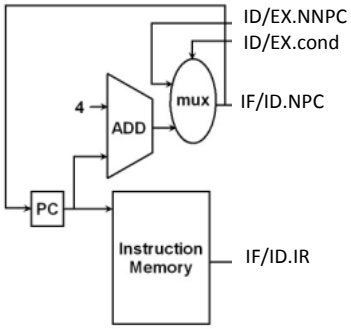
The second table shows this sequence of next results (first row) and the two methods of prediction. The second row shows the output of a 2-bit counter operating on this sequence of next results. Because of the bias in the data, the direct counting (Alpha method) predicts taken for all branches after the 2^{nd}. Since most branches are actually taken, only the not-taken branches are mispredicted. The third row shows the patterns that will be saved in the PHT (Y-P method) as new results are shifted into the 2-bit PHT entry from the right. The 4^{th} row shows the output of a 2-bit counter fed the 2-bit PHT entry. The predictions from these outputs are incorrect most of the time. This example only shows that the results of the Alpha and Yeh-Patt methods differ — no further conclusions can be drawn from this unrealistic example.

Slide 30 Address Prediction

The predicted target address if found from the **Branch Target Buffer** (BTB). The BTB is organized as cache with LRU eviction (the oldest, least recently used data is deleted to make room for new branch targets to be stored). The BTB index is the PC of the branch instruction. The entry is the last verified target address, a status (valid) field, and a special field used to indicate low confidence in the predicted address.

Slide 31 DLX Pipeline with Y-P Branch Prediction

Slide 31 shows the addition of Yeh-Patt two-step branch prediction to the IF hardware in the DLX. The standard DLX IF stage appears below.



With the addition of the branch prediction, the **cond** flag is used to control storage of branch history and target address (from NNPC — the calculated target address). The current PC is fed to the prediction logic. When the address is recognized as a previously observed branch instruction, the predicted target is used as NPC, which provides the next instruction fetch address to the PC register. If the current PC is not recognized, the default PC + 4 is used as the next PC.

Slide 32 – 33 Low Confidence Address Predictions

Certain branch instructions are unusually **difficult to predict**, especially indirect branches and returns from function calls, traps, and interrupts.

The Intel x86 ISA permits branches of the type **`JMP [ESI]`** which loads reads a data memory address from a register, and loads the program counter with an instruction address stored at in data memory. Because the register and / or data memory can change at run time, the target address for the branch can be highly dynamic, with no observable pattern. RISC ISAs provide a similar function in the jump register instruction. The Pentium M introduced a cache mechanism to follow indirect branches of this type.

A return from a subroutine can be difficult to predict, because a given function may be entered from various points in a program. The return point depends on the point in the program that called the subroutine.

Slide 33 shows a proposed (but not yet implemented) method for handling return branches.

In the x86 architecture, a subroutine call stores the return address in the hardware stack. A call instruction can trigger copying of the return address to the BTB for use on the return branch.

In a RISC ISA, function calls are implemented with the JALR / JR instruction pair. This permits the system to identify a function call and copy the return address to the BTB.

Slide 34 Exotic Branch Mispredictions

Because of the loop structure of most programs, simple branch prediction is accurate in about 90% of all branches. Of all branches, about 80% of branches (especially loop iterations) have an

accuracy of nearly 100%, while 20% of branches have a 50% accuracy (as good as a random guess).

The least accuracy is found on the one-time test of a single operand, typically found in branches for if-then-else structures. In a loop of 1000 iterations that performs `if (x == 0) {...}` in each iteration, the loop branch can be predicted with high accuracy, but value of variable x is unknown until tested.

A proposal (not yet implemented) method for handling this situation is to maintain a separate cache of operand values that are tested. Any update to a tested operand will update the prediction table for the associated branch instruction.

Slide 35 – 36 DLX Memory Hierarchy

Slides 35 and 36 show the relationships among memory units in the DLX. The Instruction Memory in stage IF and the Data Memory in stage MEM of the pipeline are level 1 (L1) cache. The L1 cache memories are connected to the cache controller. When a memory address (PC or data address formed in the ALU) is not located in L1 (L1 cache miss), the controller simultaneously accesses level 2 (L2) cache in the DLX package and Main Memory (through the external I/O controller). If the memory location is found in L2 cache (L2 cache hit), the Main Memory access is cancelled. If the location is not found in L2 cache (L2 cache miss), then the location is copied to L2 and to L1.

Slide 37 – 38 Cache Organization

Cache memory holds a copy of a section of main memory. If main memory contains 4 GB of memory locations, with a 32-bit address space of 00000000 to FFFFFFFF (hexadecimal), then L1 cache typical contains a copy of 64 KB of these locations while L2 cache contains a copy of 5 MB of these locations. In order to find the required locations in the cache, they must be copied in an orderly fashion according to a well-defined **cache organization**.

To support cache, the memory space is divided (logically, not physically) into **blocks** (also called **lines**). Each block contains B bytes, where B is a power of 2. In most cases, the page size in virtual memory (under OS control) is an integer multiple of B.

Similarly, the cache memory is divided (logically, not physically) into S **sets** and each set is divided (logically, not physically) into W **slots**. Each memory block copied to cache must be placed into a specific set, but it can be placed into any unused slot in the set. There are thus W different ways to place a block in a set, and so W is called **associativity**.

A slot holds 1 data line, so that

> **Size of the cache** = B bytes per line × W slots per set × S sets per cache.

In detail:

> The memory space is given as $N = 2^n$ bytes, where the address is an n-bit number.
>
> Each line has $B = 2^b$ bytes. The line number is given by the integer part of N / B, the upper n – b bits of the address. The lower b bits of an address is called the **byte offset** (the count from 0 to B – 1 of the byte location in the line).

The cache is logically divided into $S = 2^s$ sets. The **set index** is the set number to which the line must be copied in cache. The set index = (line number) mod S = lower s bits of the line number.

The line can be copied to any of W slots in its appropriate set.

The **upper n – (b + s) bits of the address** is called the **tag**. When a line is stored in cache, the tag is written to a table for the set. The CPU looking for a memory address can find the line number and set number by the known method. To search for the line in cache, the CPU must search all the tag numbers listed in the table of the appropriate set.

Slide 39 Cache Definitions and Policies

The CPU accesses cache, either by sending the PC to instruction cache or by sending an address calculated in the ALU to data cache. When the cache line containing the address is found in cache, this is called a **cache hit**. If the cache line is not found in cache, it is a **cache miss**.

Every **read miss** (a cache miss on a read access) causes the line to be copied to cache. A **write miss** can be handled in one of two ways (policies):

Write allocate policy means that the line is loaded to cache to be written. Most write misses are handled this way.

No write allocate policy means that the address is written directly in main memory, without copying the line to cache.

If a cache set is full (all the slots hold valid cache lines copied from main memory), but a new line must be added to the set, one of the lines in the set must be **swapped out**. This means that a line must be copied back to main memory and overwritten with a new cache line. The line chosen for swapping out is the **least recently used** (LRU) — the line whose last access was earliest (oldest). Two policies are relevant to swapping out:

Write back policy allows the CPU to write to cache without updating main memory. Main memory is only updates when the line is swapped out. This policy simplifies memory organization and is used on most cache writes.

Write through policy requires the CPU to update main memory on every write to cache.

Slide 40 – 42 Cache Performance Issues

The performance of cache memory depends on the number of clock cycles required for the CPU to access a memory location (data or instruction). On an L1 cache hit, the CPU memory access takes place in just 1 clock cycle, as described in RISC implementation. But on an L1 cache miss, the CPU stalls while L1 cache copies the missing line from L2 cache or main memory.

The **miss rate** is the relative number of cache misses per cache accesses. The miss rate depends on the cache size and organization, the size of the program and data structures. The **miss penalty** is the cache update latency, the number of stall cycles while the cache copies the missing line from L2 or main memory.

A CPU with a two-level cache is shown on slide 41. The miss rate and miss penalty can be different at L1 and L2. Since L1 is split into instruction (I) and data (D), the respective miss rates can be different. The parameters for the hit and miss rates are written in the following notation:

M_{L1}^{I} = Instruction Miss Rate at L1

$1 - M_{L1}^{I}$ = Instruction Hit Rate at L1

M_{L1}^{D} = Data Miss Rate at L1

$1 - M_{L1}^{D}$ = Data Hit Rate at L1

M_{L2} = Miss Rate at L2

1-M_{L2} = Hit Rate at L2

The miss penalties at L1 and L2 are written:

P_{L1} = Miss Penalty at L1 $\qquad$ P_{L2} = Miss Penalty at L2

Analysis of the two-level cache can be simplified by the following assumptions (as shown in the probability tree diagram):

1. L1 instruction cache is accessed for every instruction fetch
2. L1 data cache is accessed for every data load or data store
3. On an L1 hit there is no penalty
4. On an L1 miss, the CPU accesses L2 cache.
    a. On an L2 hit, the total cache penalty is P_{L1}.
    b. On an L2 miss, the L2 must access main memory with an additional cache penalty given by P_{L2}.

The CPI^{stall} is stall cycles per cache miss (miss penalty) × cache miss per access (miss rate). There is one instruction fetch access at L1 (I) for every fetched instruction, which misses with the probability $M_{L1}{}^{I}$. There are ($IC^{data\ access}$ / IC) data accesses per instruction, which miss with probability $M_{L1}{}^{D}$. The weighted average miss rate is

$$\text{Cache miss per access} = \left[M_{L1}^{I} + M_{L1}^{D} \times \frac{IC^{data\ access}}{IC} \right]$$

The miss penalty is P_{L1} on a simple L1 miss, with the addition of P_{L2} when L2 also misses (with probability M_{L2}. Together the average miss penalty is

$$\text{stall cycles per cache miss} = \left(P_{L1} + P_{L2} \times M_{L2} \right)$$

Together,

$$CPI_{cache}^{stall} = \frac{\text{cache miss stall cycles}}{\text{instruction}} = \left[M_{L1}^{I} + M_{L1}^{D} \times \frac{IC^{data\ access}}{IC} \right] \times \left(P_{L1} + P_{L2} \times M_{L2} \right)$$

This expression demonstrates the advantage of the two-level cache system. If there is only one level of cache, the cache penalty is L1 to main memory, about 50 clock cycles. In two level cache, an L1 miss causes a short 5 CC miss penalty to L2 and an additional 50 CC penalty to main memory only on the small number of L2 misses. If the miss rate at L2 is 1%, then the miss penalty in two-level cache is 5.5 CC compared to 50 CC for the one-level system.

Slide 43 – 46 Issues Affecting Miss Rate

The causes of cache miss can be divided into three groups.

A **compulsory miss** occurs because updates to cache only occur on the first access to a cache line (known as a **demand access** update). This means that the first access to any cache line must be a miss. Cache size and associativity have no effect on compulsory misses.

A **capacity miss** occurs because a line that was previously in cache was swapped out to make room for another cache line. The larger the cache, the more room for many lines, which lowers the capacity miss rate.

A **conflict miss** is similar to a capacity miss — a line that was previously in cache was swapped out to make room for another cache line in a specific set, even when there is spare capacity in some other set. The conflict miss rate is lower when the cache has greater associativity, because there is more room for lines in each set. If the total cache size is fixed, then a cache with fewer sets must have more slots per set. But this makes locating a cache line for complex, because more cache tags must be searched.

The graph on slide 44 shows measured miss rate as a function of cache size and associativity. For all associativity levels, cache miss rate falls as cache size increases, as expected. At any given size, the cache miss rate drops with increased associativity (this is the effect of conflict miss). But for common cache sizes (16 KB and above), there is little advantage to associativity above 4. As seen on slide 45, larger associativity means fewer sets, a smaller set index, and so a longer tag number. So, more associativity means search through more slots per set to find a longer tag, making associativity above 4 inefficient.

The tags can be searched in three ways:

> In **serial access** the tag controller searches the tag for each way, one by one. This trades longer latency for lower power.
>
> In **parallel access**, the tags controller searches all tags at the same time. This trades shorter latency for higher power.
>
> In **way prediction**, the way is predicted in the manner of branch prediction, based on patterns of previous accesses.

Slide 47 Cache Prefetching

The **demand access** method in which cache is updated in response to a cache miss involves latency associated with the cache penalty. The method of **prefetching** "hides cache latency" by updating cache before a miss, and bringing the memory block to cache before it is needed.

Data cache prefetching is useful for sequential accesses to large data structures. It can be implemented by a software prefetch mechanism, in which assembly language **prefetch instructions** perform the cache update. These instructions can be inserted by the programmer or automatically by the compiler. The MIPS architecture was seen to provide a prefetch instruction. Prefetch can also be implemented in hardware with a **stride buffer** that automatically predicts sequential accesses.

Instruction cache prefetching must use a hardware-based prefetch mechanism, which predicts instruction cache updates based on program structure.

Slide 48 – 49 Data Cache Latency Example

Consider a loop over some operation involving a large data structure, such as a 1024 by 1024 element array of 16-byte SSE operands. The data structure is 16 MB and each element is accessed one time. A Pentium 4 processor with a 64-byte cache line can hold 4 operands, so that every 4 operands there is an L1 data cache miss — a total of 256 Kmisses.

The software prefetch on Intel CPUs fetches 2 cache lines (128 bytes), which provide 8 SSE operands. If the line is already in cache, the prefetch instruction is cancelled to a NOP. The code fragment on slide 49 identifies and loads the next 2 data lines for each data line in the current loop iteration. The prefetch hides the latency by initiating the cache update while performing useful work on the current data.

Slide 50 – 54 General Stream Buffer

In the context of cache, a **stream** is a set of successive accesses to sequential cache lines. A stream buffer is a FIFO buffer that generates and buffers update requests for successive cache lines.

The stream buffer can be implemented in two ways:

> A request buffer receives the requested line from the higher memory level and copies it directly to cache.
>
> A line buffer keeps the requested line in the buffer until the demand miss.

In both cases, the stream buffer performs request filtering — it skips lines that are already in cache, in order to minimize prefetches.

Slide 51 provides more details. On clock cycle k, the stream buffer is allocated to missed line L. The miss penalty causes latency while line L is updated. On subsequent clock cycles, k + 1, k + 2, and so on, the stream buffer requests line L + 1, L + 2, and so on. There are no additional miss penalties following the miss of line L.

Slide 52 shows a numerical example. The content of cache is shown in the first line. Access to lines 2 and 3 are hits, but the access to 4 is a miss. This generates update requests for lines 4, 5, and so on. No update requests are generated for line 6 and 8 because these lines are already present in cache.

A **stride buffer** operates in the same manner as a stream buffer, except that subsequent cache updates are separated by the stride s.

Slide 54 shows a stream buffer added to the DLX MEM stage. A data cache miss in L1 triggers the stream buffer to request sequential cache updates through the cache controller to L2 and main memory.

Slide 55 – 56 Out-of-Order Fetch

The **Miss Information Status History Register (MSHR)** is a variation of the stream buffer for instruction cache. An instruction queue is inserted between the IF stage and the ID stage. On a cache hit, the fetched instruction is placed in the queue. On a cache miss, a pointer is to the MSHR is placed into the queue instead of the instruction. The MSHR acts as a stream buffer initiating cache updates. When the cache line becomes available, the instruction queue copies the instruction instead of the pointer. An example is shown on slide 56.

Slide 56 – 57 Fetch Directed Instruction Prefetching

When branch prediction is used, it can also be used to predict instruction cache prefetches. In this method, instead of inserting an instruction buffer between the IF and ID stages, an address buffer is inserted between PC and the IF logic. The instruction address sent by the branch predictor to the PC is placed into the buffer. Search logic immediately checks the tags in the L1 cache to see if the required line is in cache. If the line is missing, then L1 cache can initiate the cache update immediately, while earlier instructions are being fetched according to addresses in the queue. Slide 58 shows the DLX with the prefetch included in the IF stage.