Branch Prediction

•Branch prediction is one of the ancient performance improving techniques which still finds relevance into modern architectures. While the simple prediction techniques provide fast lookup and power efficiency they suffer from high misprediction rate.

•On the other hand, complex branch predictions – either neural based or variants of two-level branch prediction – provide better prediction accuracy but consume more power and complexity increases exponentially.

• In addition to this, in complex prediction techniques the time taken to predict the branches is itself very high – ranging from 2 to 5 cycles – which is comparable to the execution time of actual branches.

•Branch prediction is essentially an optimization (minimization) problem where the emphasis is on to achieve lowest possible miss rate, low power consumption and low complexity with minimum resources.

A Closer Look At Branch Prediction

There really are three different kinds of branches:

- **Forward conditional branches** - based on a run-time condition, the PC (Program Counter) is changed to point to an address forward in the instruction stream.

- **Backward conditional branches** - the PC is changed to point backward in the instruction stream. The branch is based on some condition, such as branching backwards to the beginning of a program loop when a test at the end of the loop states the loop should be executed again.

- **Unconditional branches** - this includes jumps, procedure calls and returns that have no specific condition. For example, an unconditional jump instruction might be coded in assembly language as simply "jmp", and the instruction stream must immediately be directed to the target location pointed to by the jump instruction, whereas a conditional jump that might be coded as "jmpne" would redirect the instruction stream only if the result of a comparison of two values in a previous "compare" instructions shows the values to not be equal. (The segmented addressing scheme used by the x86 architecture adds extra complexity, since jumps can be either "near" (within a segment) or "far" (outside the segment). Each type has different effects on branch prediction algorithms.)

Static Branch Prediction

Static Branch Prediction predicts always the same direction for the same branch during the whole program execution.

It comprises hardware-fixed prediction and compiler-directed prediction.

Simple hardware-fixed direction mechanisms can be:

- Predict always not taken
- Predict always taken
- Backward branch predict taken, forward branch predict not taken

Sometimes a bit in the branch opcode allows the compiler to decide the prediction direction.

Dynamic Branch Prediction

Dynamic Branch Prediction: the hardware influences the prediction while execution proceeds.

Prediction is decided on the computation history of the program.

During the start-up phase of the program execution, where a static branch prediction might be effective, the history information is gathered and dynamic branch prediction gets effective.

In general, dynamic branch prediction gives better results than static branch prediction, but at the cost of increased hardware complexity.

Using Branch Statistics for Static Prediction

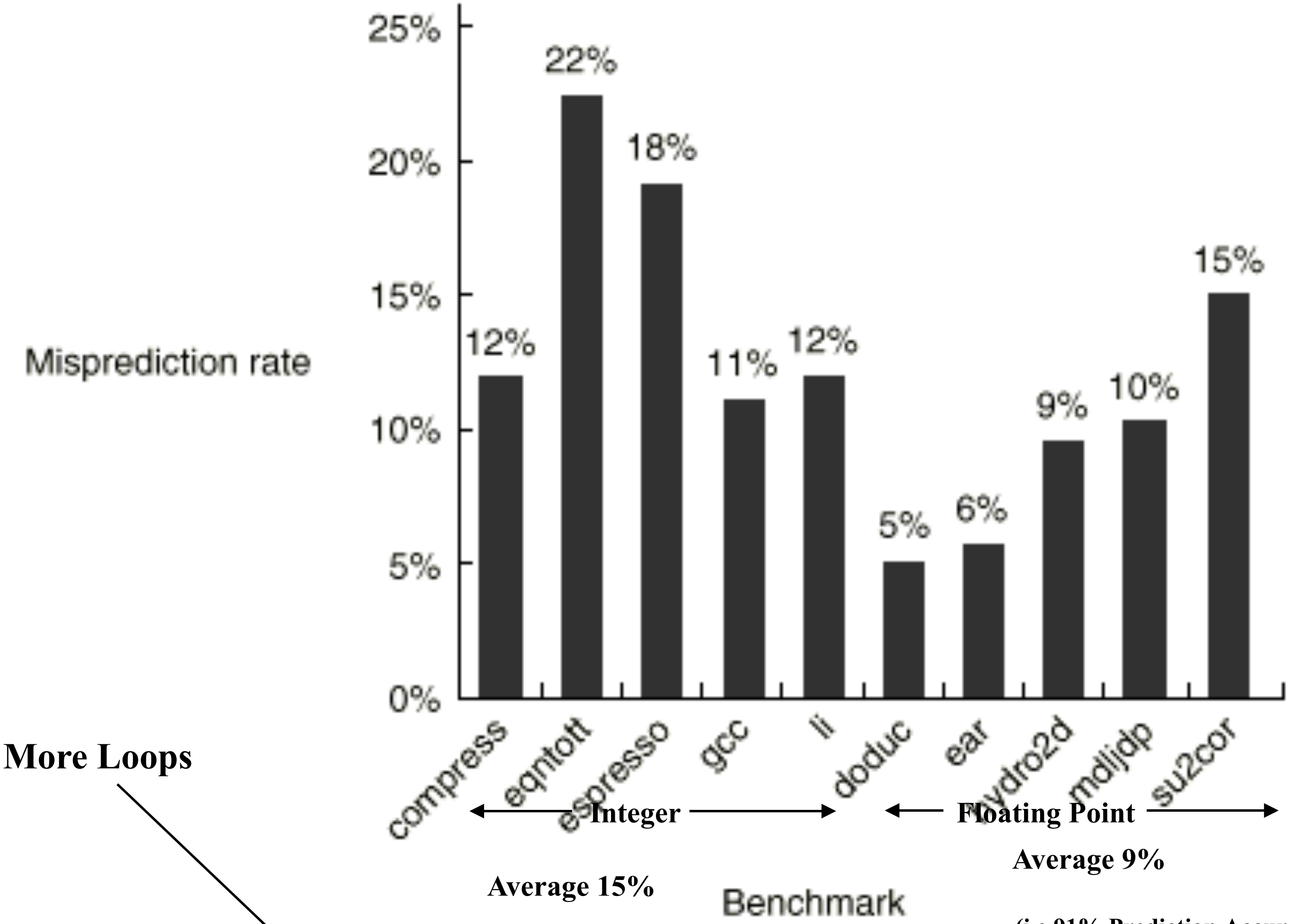
Forward branches dominate backward branches by about 4 to 1 (whether conditional or not). About 60% of the forward conditional branches are taken, while approximately 85% of the backward conditional branches are taken (because of the prevalence of program loops).

Just knowing this data about average code behavior, we could optimize our architecture for the common cases. A "Static Predictor" can just look at the offset (distance forward or backward from current PC) for conditional branches as soon as the instruction is decoded.

Backward branches will be predicted to be taken, since that is the most common case. The accuracy of the static predictor will depend on the type of code being executed, as well as the coding style used by the programmer.

These statistics were derived from the SPEC suite of benchmarks, and many PC software workloads will favor slightly different static behavior.

Static Profile-Based Compiler Branch Misprediction Rates for SPEC92



More Loops

Integer

Average 15%

(i.e 85% Prediction Accuracy)

Floating Point

Average 9%

(i.e 91% Prediction Accuracy)

FIGURE 3.36 Misprediction rate for a profile-based predictor varies widely but is generally better for the FP programs, which have an average misprediction rate of 9% with a standard deviation of 4%, than for the integer programs, which have an average misprediction rate of 15% with a standard deviation of 5%.

Dynamic Conditional Branch Prediction

- Dynamic branch prediction schemes are different from static mechanisms because they utilize hardware-based mechanisms that use the run-time behavior of branches to make more accurate predictions than possible using static prediction.
- Usually information about outcomes of previous occurrences of branches (branching history) is used to dynamically predict the outcome of the current branch. Some of the proposed dynamic branch prediction mechanisms include:
  - **One-level or Bimodal: Uses a Branch History Table (BHT), a table of usually two-bit saturating counters which is indexed by a portion of the branch address (low bits of address). (First proposed mid 1980s)**
  - **Two-Level Adaptive Branch Prediction. (First proposed early 1990s),**
  - **MCFarling's Two-Level Prediction with index sharing (gshare, 1993).**
  - **Hybrid or Tournament Predictors: Uses a combinations of two or more (usually two) branch prediction mechanisms (1993).**
- To reduce the stall cycles resulting from correctly predicted taken branches to zero cycles, a Branch Target Buffer (BTB) that includes the addresses of conditional branches that were taken along with their targets is added to the fetch stage.

How to further reduce the impact of branches on pipeline processor performance

Dynamic Branch Prediction:

Hardware-based schemes that utilize run-time behavior of branches to make dynamic predictions:

Information about outcomes of previous occurrences of branches are used to dynamically predict the outcome of the current branch.

Why? Better branch prediction accuracy and thus fewer branch stalls

Branch Target Buffer (BTB):

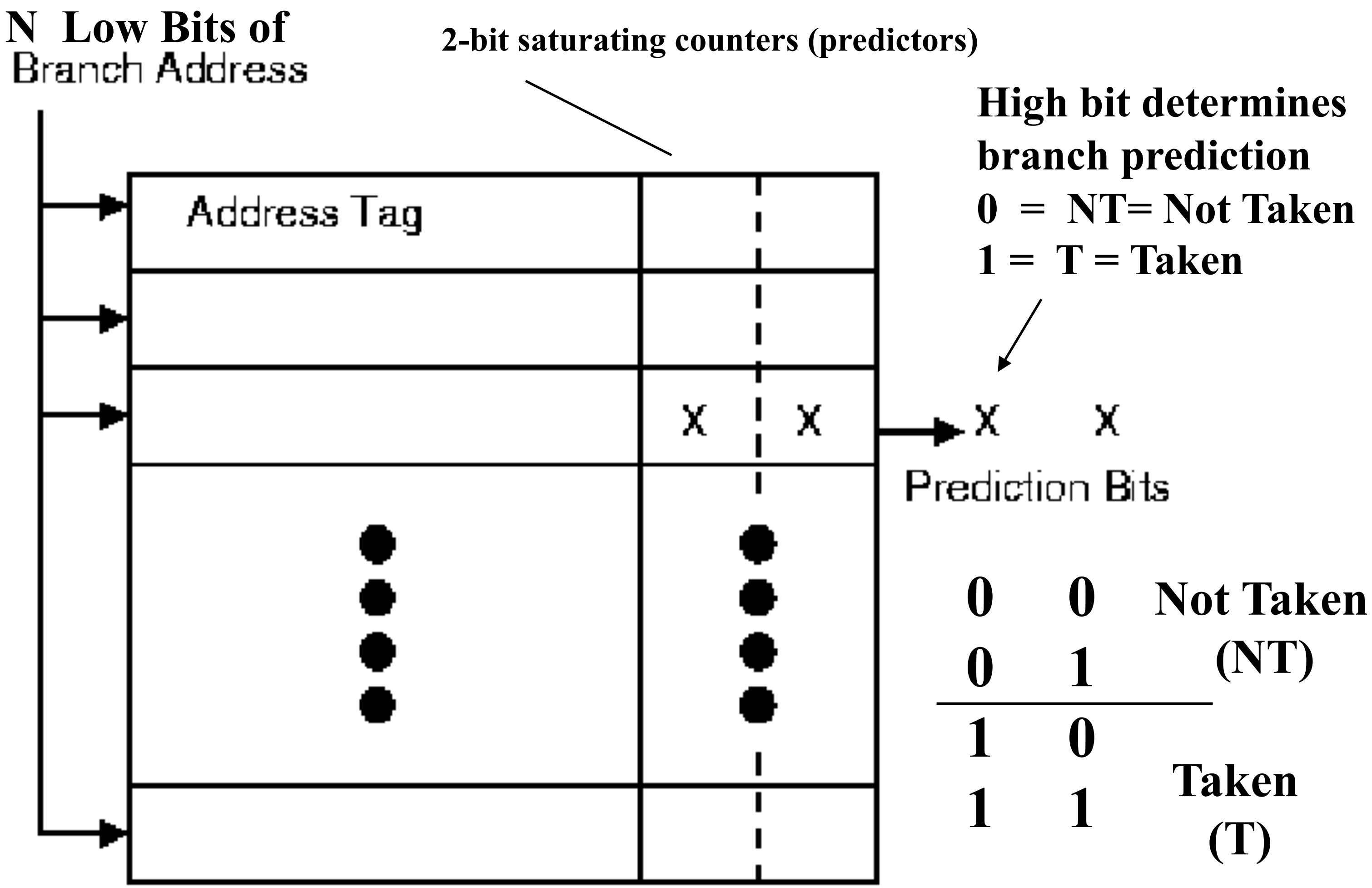
A hardware mechanism that aims at reducing the stall cycles resulting from correctly predicted taken branches to zero cycles.

Dynamic Branch Prediction with a Branch History Buffer (BHB)

To refine our branch prediction, we could create a buffer that is indexed by the low-order address bits of recent branch instructions. In this BHB (sometimes called a "Branch History Table (BHT)"), for each branch instruction, we'd store a bit that indicates whether the branch was recently taken. A simple way to implement a dynamic branch predictor would be to check the BHB for every branch instruction. If the BHB's prediction bit indicates the branch should be taken, then the pipeline can go ahead and start fetching instructions from the new address (once it computes the target address).

By the time the branch instruction works its way down the pipeline and actually causes a branch, then the correct instructions are already in the pipeline. If the BHB was wrong, a "misprediction" occurred, and we'll have to flush out the incorrectly fetched instructions and invert the BHB prediction bit.

Branch History Table (BHT)



N Low Bits of Branch Address

2-bit saturating counters (predictors)

High bit determines branch prediction
0 = NT= Not Taken
1 = T = Taken

Address Tag

X X

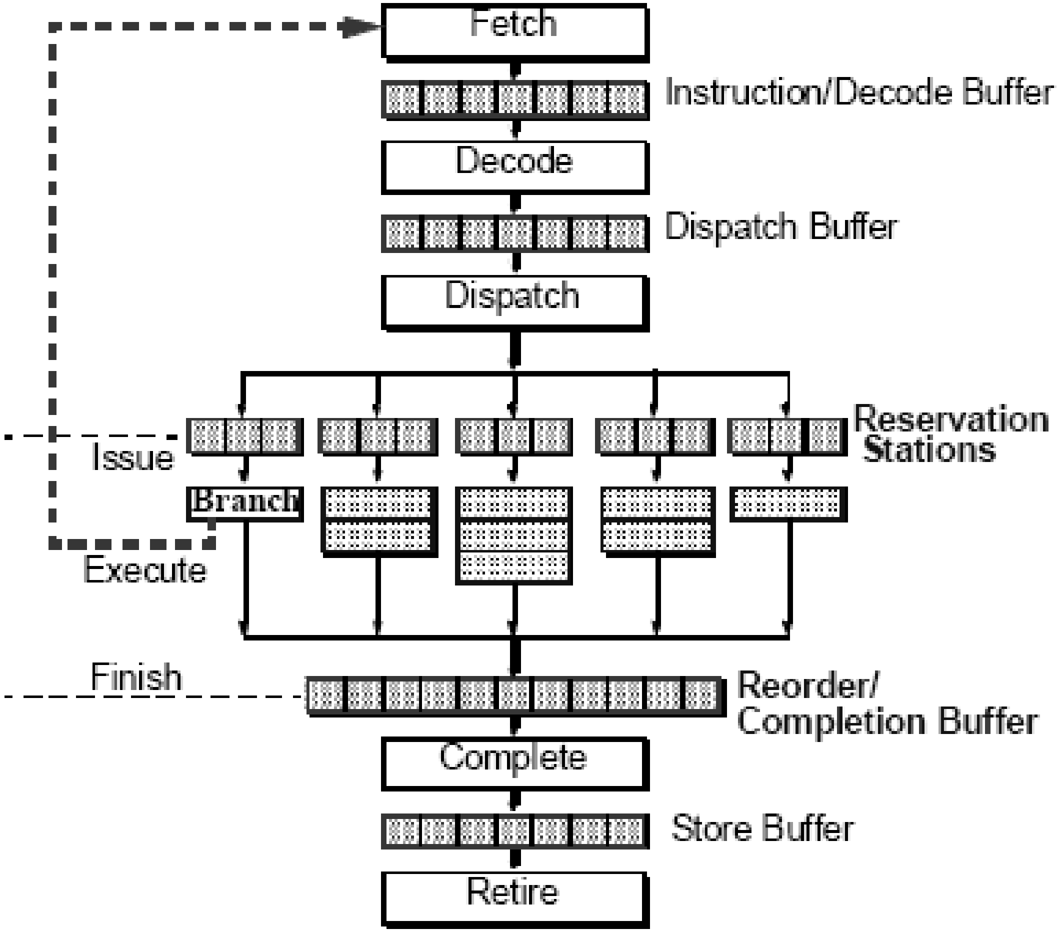
X X

Prediction Bits

0	0	Not Taken (NT)
0	1	
1	0	Taken (T)
1	1	

Dynamic Branch Prediction with a Branch History Buffer (BHB)

Disruption of Sequential Control Flow



Refining Our BHB by Storing More Bits

It turns out that **a single bit in the BHB will be wrong twice for a loop--once on the first pass of the loop and once at the end of the loop**. We can get better prediction accuracy by using more bits to create a "saturating counter" that is incremented on a taken branch and decremented on an untaken branch**. It turns out that a 2-bit predictor does about as well as you could get with more bits, achieving anywhere from 82% to 99% prediction accuracy with a table of 4096 entries.**
This size of table is at the point of diminishing returns for 2 bit entries, so there isn't much point in storing more. Since we're only indexing by the lower address bits, notice that 2 different branch addresses might have the same low-order bits and could point to the same place in our table--one reason not to let the table get too small.

Two-Level Predictors and the GShare Algorithm

There is a further refinement we can make to our BHB by correlating the behavior of other branches. Often called a "**Global History Counter**", this "two-level predictor" allows the behavior of other branches to also update the predictor bits for a particular branch instruction and achieve slightly better overall prediction accuracy. One implementation is called the "**GShare algorithm**".

This approach uses a "**Global Branch History Register**" (a register that stores the global result of recent branches) that gets "hashed" with bits from the address of the branch being predicted. The resulting value is used as an index into the BHB where the prediction entry at that location is used to **dynamically predict the branch direction**. Yes, this is complicated stuff, but it's being used in several modern processors.

Two-Level Predictors and the GShare Algorithm

Combined branch prediction*

Scott McFarling proposed **combined branch prediction** in his 1993 paper 2. *Combined branch prediction* is about as accurate as local prediction, and almost as fast as global prediction.

Combined branch prediction uses three predictors in parallel: bimodal, gshare, and a bimodal-like predictor to pick which of bimodal or gshare to use on a branch-by-branch basis. The choice predictor is yet another 2-bit up/down saturating counter, in this case the MSB choosing the prediction to use. In this case the counter is updated whenever the bimodal and gshare predictions disagree, to favor whichever predictor was actually right.

On the SPEC'89 benchmarks, such a predictor is about as good as the local predictor.

Another way of combining branch predictors is to have e.g. 3 different branch predictors, and merge their results by a majority vote.

Predictors like gshare use multiple table entries to track the behavior of any particular branch. This multiplication of entries makes it much more likely that two branches will map to the same table entry (a situation called aliasing), which in turn makes it much more likely that prediction accuracy will suffer for those branches. Once you have multiple predictors, it is beneficial to arrange that each predictor will have different aliasing patterns, so that it is more likely that at least one predictor will have no aliasing**. Combined predictors with different indexing functions for the different predictors are called *gskew* predictors**, and are analogous to skewed associative caches used for data and instruction caching.

* From : http://en.wikipedia.org/wiki/Branch_prediction

Using a Branch Target Buffer (BTB) to Further Reduce the Branch Penalty

In addition to a large BHB, **most predictors also include a buffer that stores the actual target address** of taken branches (along with optional prediction bits). This table allows the CPU to look to see if an instruction is a branch and start fetching at the target address early on in the pipeline processing. By storing the instruction address and the target address, even before the processor decodes the instruction, it can know that it is a branch.

A large BTB can completely remove most branch penalties (for correctly-predicted branches) if the CPU looks far enough ahead to make sure the target instructions are pre-fetched. Using a Return Address Buffer to predict the return from a subroutine **One technique for dealing with the unconditional branch at the end of a subroutine is to create a buffer of the most recent return addresses.**

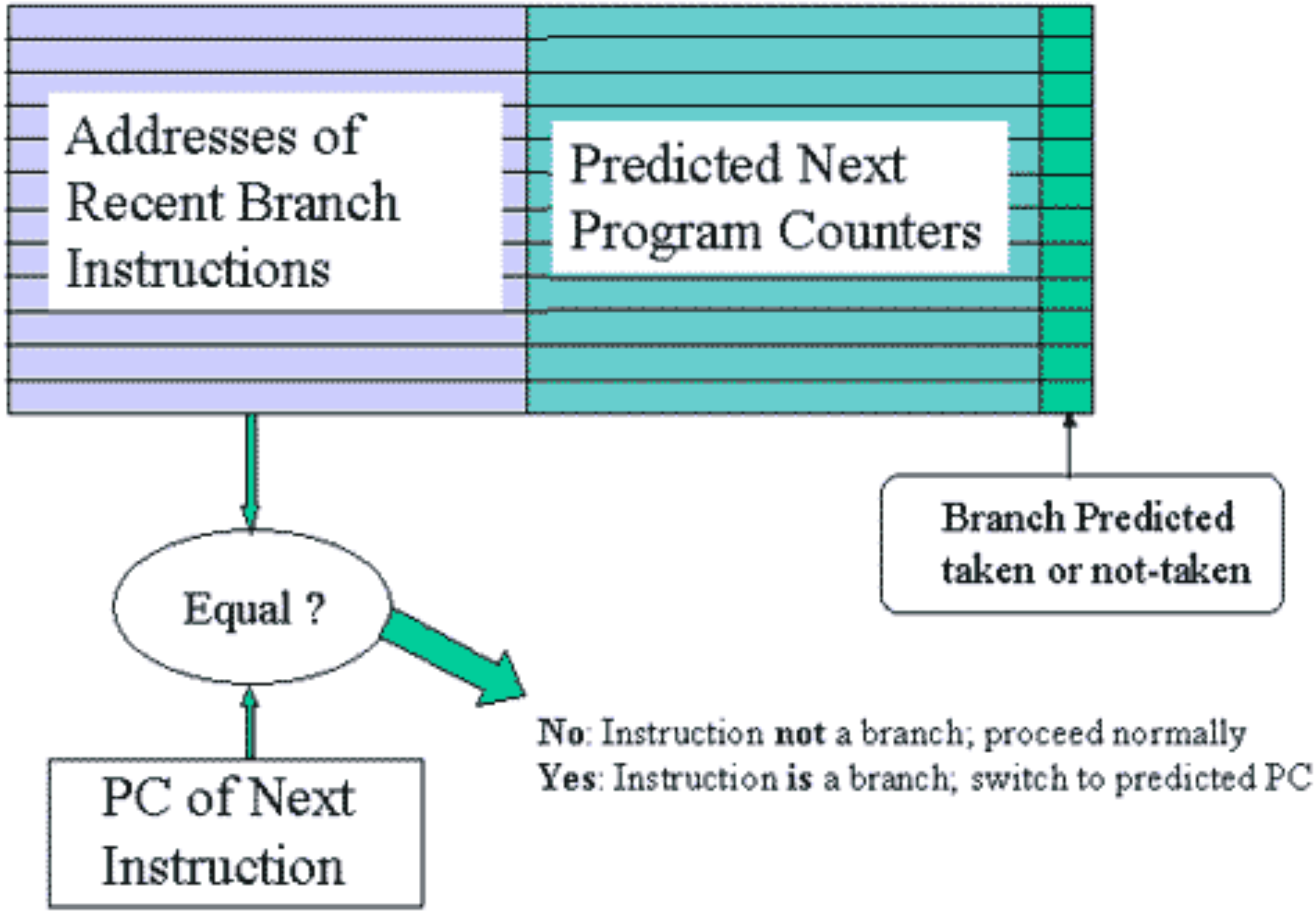
There are usually some subroutines that get called quite often in a program, and a return address buffer can make sure that the correct instructions are in the pipeline after the return instruction.

Branch Target Buffer (BTB)

- **Effective branch prediction requires the target of the branch at an early pipeline stage.** (resolve the branch early in the pipeline)
- **One can use additional adders to calculate the target, as soon as the branch instruction is decoded.** This would mean that one has to wait until the ID stage before the target of the branch can be fetched, taken branches would be fetched with a one-cycle penalty (this was done in the enhanced MIPS pipeline).
- **To avoid this problem one can use a Branch Target Buffer (BTB)**. A typical BTB is an associative memory where the addresses of taken branch instructions are stored together with their target addresses.
- **Some designs store n prediction bits as well, implementing a combined BTB and Branch history Table (BHT).**
- **Instructions are fetched from the target stored in the BTB in case the branch is predicted-taken and found in BTB**. After the branch has been resolved the BTB is updated. If a branch is encountered for the first time a new entry is created once it is resolved as taken.
- **Branch Target Instruction Cache (BTIC): A variation of BTB which caches also the code of the branch target instruction in addition to its address**. This eliminates the need to fetch the target instruction from the instruction cache or from memory.

BTB

Branch Target Buffer



Addresses of Recent Branch Instructions
Predicted Next Program Counters
Branch Predicted taken or not-taken
Equal ?
PC of Next Instruction

No: Instruction **not** a branch; proceed normally

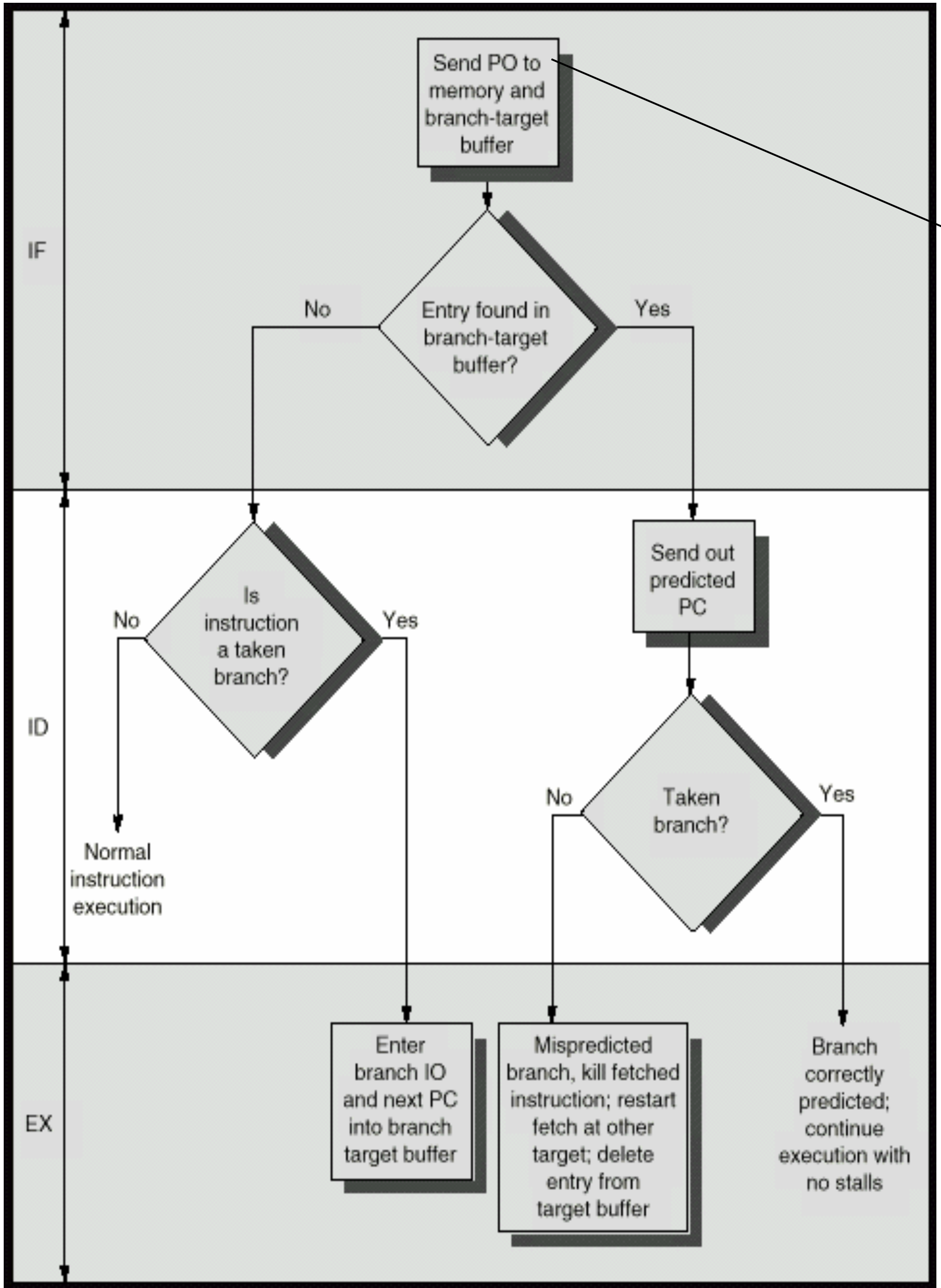
Yes: Instruction **is** a branch; switch to predicted PC

BTB Flow

Fetch

Decode

Execute

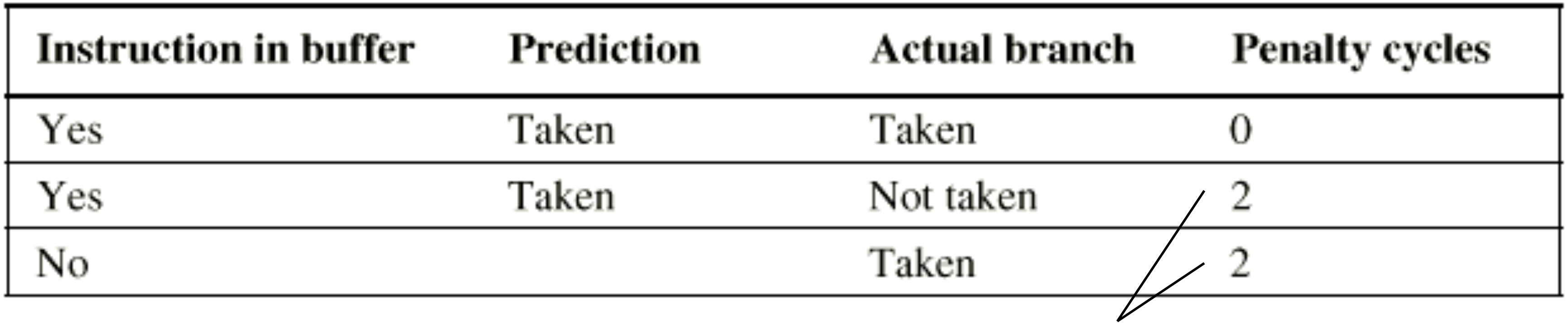

IF
Send PO to memory and branch-target buffer
Entry found in branch-target buffer?
No
Yes
ID
Is instruction a taken branch?
No
Yes
Normal instruction execution
Send out predicted PC
Taken branch?
No
Yes
EX
Enter branch IO and next PC into branch target buffer
Mispredicted branch, kill fetched instruction; restart fetch at other target; delete entry from target buffer
Branch correctly predicted; continue execution with no stalls

Prediction Output

BTB Penalties

Branch Penalty Cycles
Using A Branch-Target Buffer (BTB)

Base Pipeline Taken Branch Penalty = 1 cycle (i.e. branches resolved in ID)



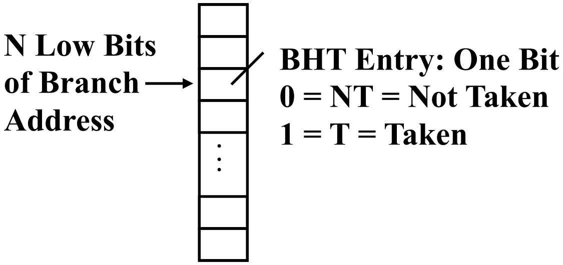
No	**Not Taken**	**Not Taken**	**0**
Instruction in buffer	**Prediction**	**Actual branch**	**Penalty cycles**
Yes	Taken	Taken	0
Yes	Taken	Not taken	2
No		Taken	2

Assuming one more stall cycle to update BTB
Penalty = 1 + 1 = 2 cycles

Penalties for all possible combinations of whether the branch is in the buffer and what it actually does, assuming we store only taken branches in the buffer.

Dynamic Branch Prediction

- Simplest method: (One-Level)
  - **A branch prediction buffer or Branch History Table (BHT) indexed by low address bits of the branch instruction.**
  - **Each buffer location (or BHT entry) contains one bit indicating whether the branch was recently taken or not**
    - **e.g 0 = not taken , 1 =taken**
  - **Always mispredicts in first and last loop iterations.**



- To improve prediction accuracy, two-bit prediction is used:
  - **A prediction must miss twice before it is changed.**
    - **Thus, a branch involved in a loop will be mispredicted only once when encountered the next time as opposed to twice when one bit is used.**
  - **Two-bit prediction is a specific case of n-bit saturating counter incremented when the branch is taken and decremented when the branch is not taken.**
  - **Two-bit prediction counters are usually always used based on observations that the performance of two-bit BHT prediction is comparable to that of n-bit predictors.**

Why 2-bit Prediction?

The counter (predictor) used is updated after the branch is resolved

Smith Algorithm

One-Level (Bimodal) Branch Predictors

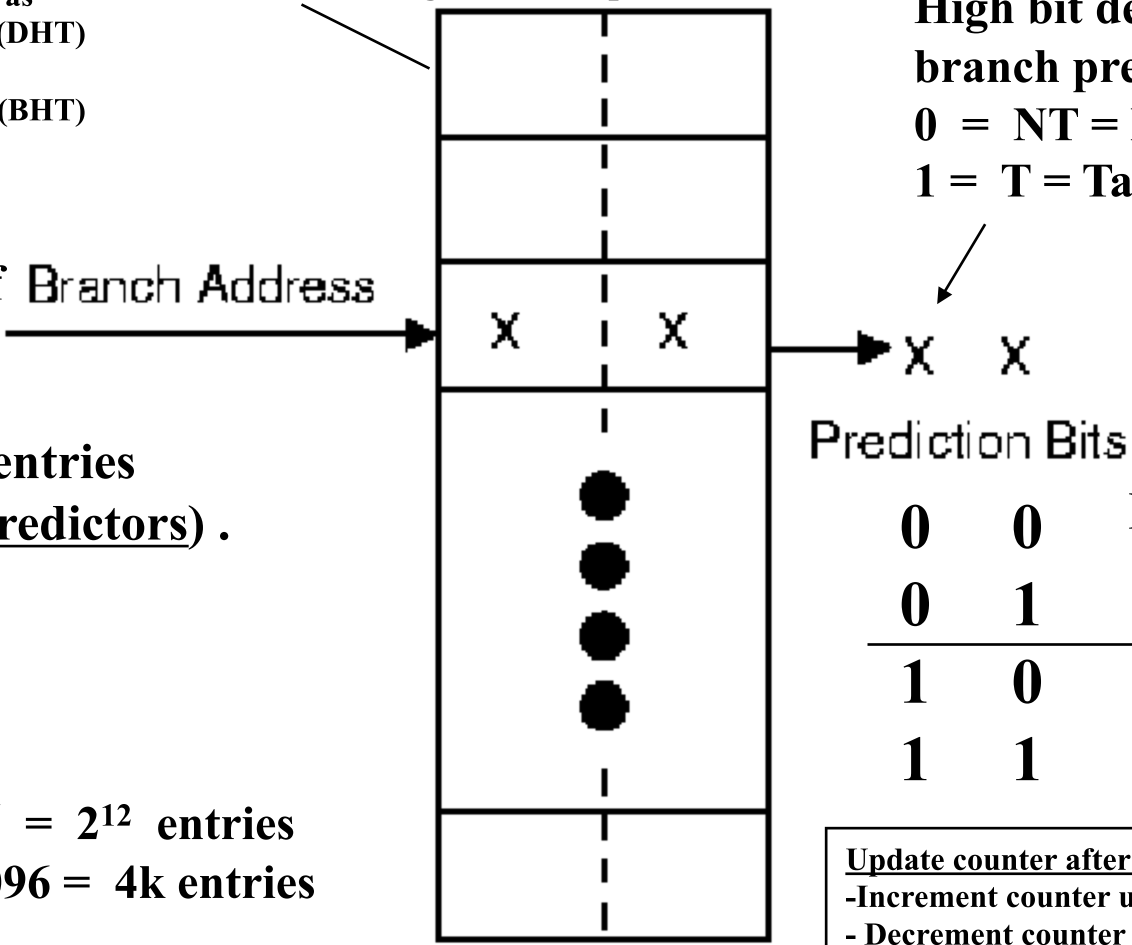
- One-level or bimodal branch prediction uses only one level of branch history.
- These mechanisms usually employ a table which is indexed by lower N bits of the branch address.
- Each table entry (or predictor) consists of n history bits, which form an n-bit automaton or saturating counters.
- Smith proposed such a scheme, known as the Smith Algorithm, that uses a table of two-bit saturating counters. (1985)
- One rarely finds the use of more than 3 history bits in the literature.
- Two variations of this mechanism:
  - **Pattern History Table: Consists of directly mapped entries.**
  - **Branch History Table (BHT): Stores the branch address as a tag. It is associative and enables one to identify the branch instruction during IF by comparing the address of an instruction with the stored branch addresses in the table (similar to BTB).**

One-Level (Bimodal) Branch Predictors

Sometimes referred to as
Decode History Table (DHT)
or
Branch History Table (BHT)

2-bit saturating counters (predictors)

N Low Bits of Branch Address



High bit determines branch prediction
0 = NT = Not Taken
1 = T = Taken

0	0	Not Taken (NT)
0	1	
1	0	Taken (T)
1	1	

Table has 2^N entries (also called predictors) .

Example:

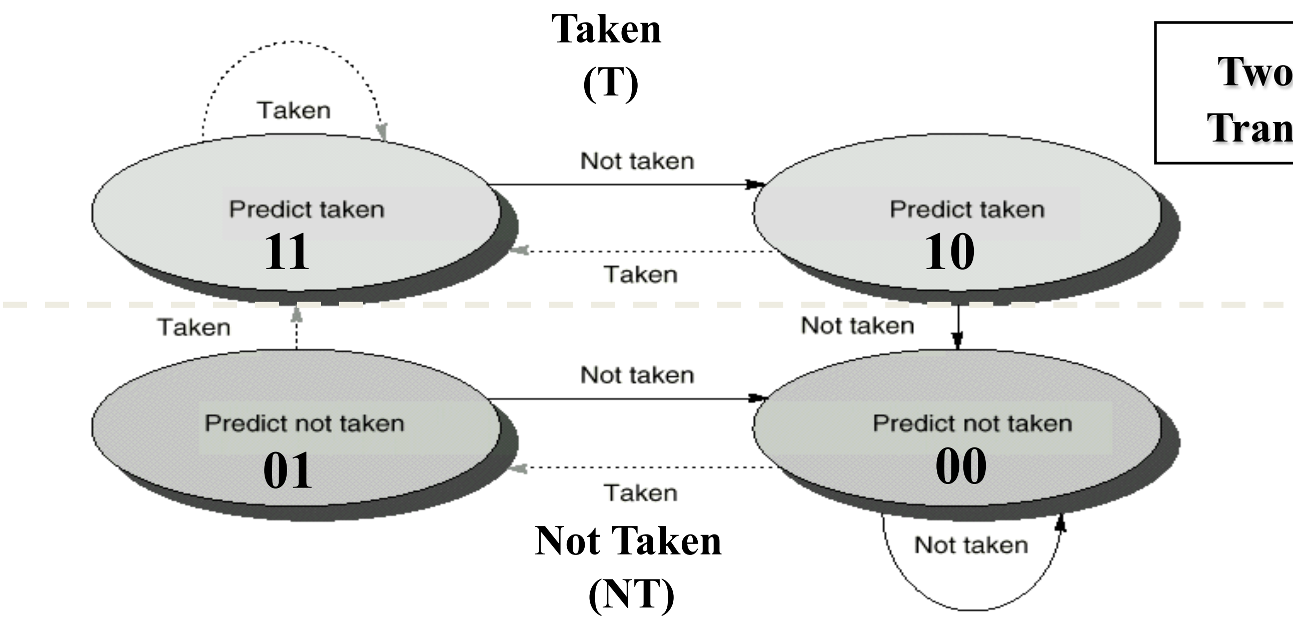
For N =12
Table has $2^N = 2^{12}$ entries
= 4096 = 4k entries

Number of bits needed = 2 x 4k = 8k bits

Update counter after branch is resolved:
-Increment counter used if branch is taken
- Decrement counter used if branch is not taken

What if different branches map to the same predictor (counter)?
This is called branch address aliasing and leads to interference with current branch prediction by other branches and may lower branch prediction accuracy for programs with aliasing.

Basic Dynamic Two-Bit Branch Prediction:



Taken (T)

Not Taken (NT)

Two-bit Predictor State Transition Diagram

0	0	Not Taken (NT)
0	1	
1	0	Taken (T)
1	1	

Or Two-bit saturating counter predictor state transition diagram (Smith Algorithm):

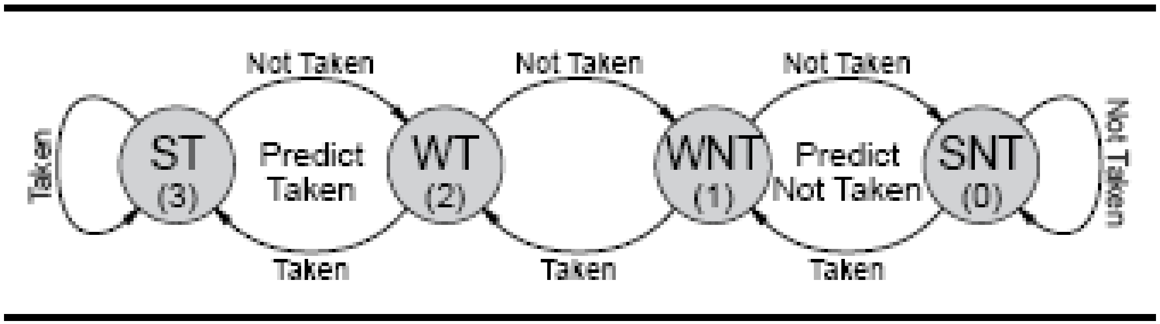


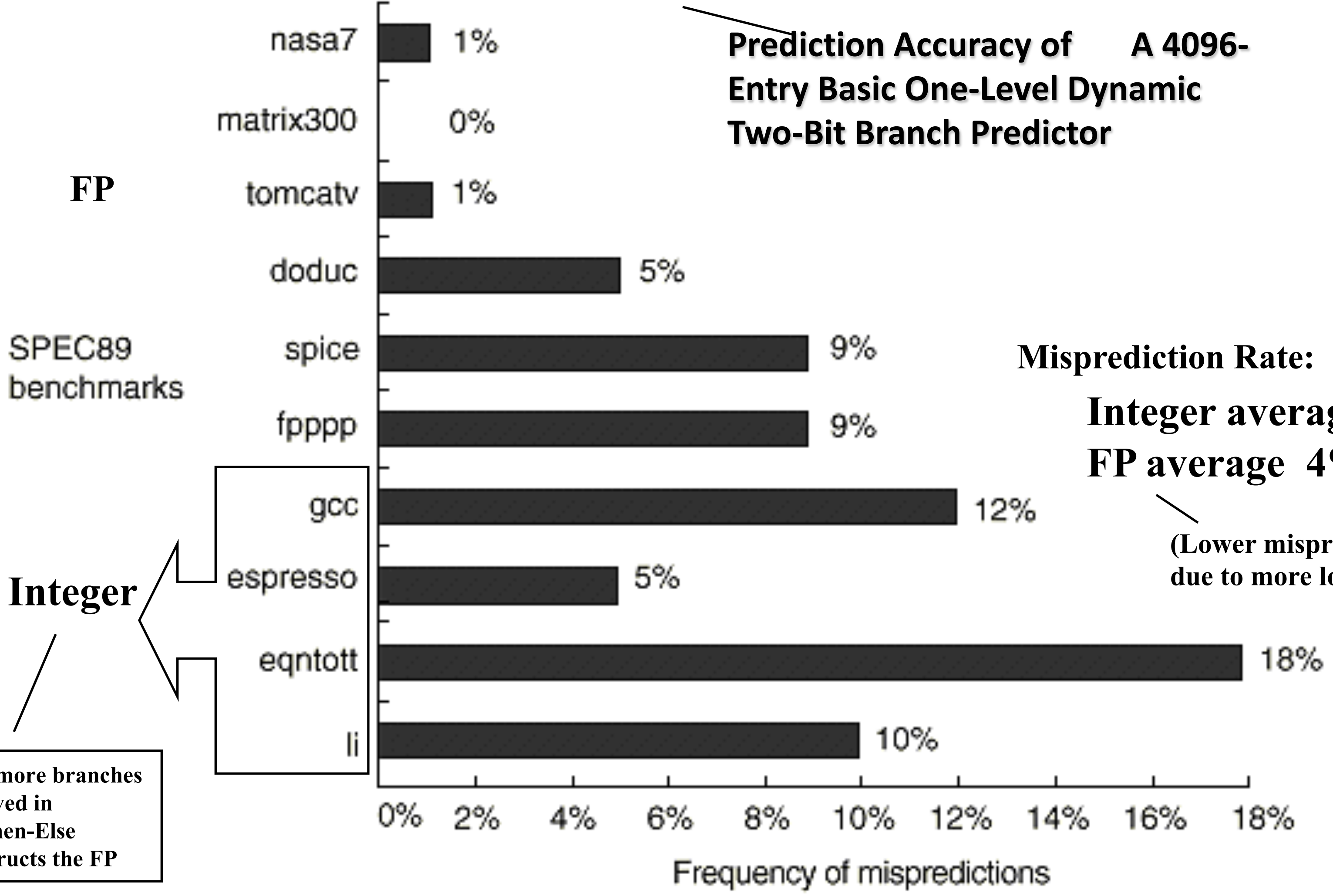
Figure 1. In the two-bit Smith algorithm, the two history bits implement a state machine with four possible states: strongly taken (ST), weakly taken (WT), weakly not taken (WNT), and strongly not taken (SNT). In ST and WT, future branches are predicted taken; in WNT and SNT, branches are predicted not taken.

* From: New Algorithm Improves Branch Prediction Vol. 9, No. 4, March 27, 1995 © 1995 MicroDesign Resources

N=12

$2^N = 4096$

Prediction Accuracy of A 4096-Entry Basic One-Level Dynamic Two-Bit Branch Predictor



FP

Integer

Has, more branches involved in IF-Then-Else constructs the FP

Misprediction Rate:

Integer average 11%

FP average 4%

(Lower misprediction rate due to more loops)

Prediction accuracy of a 4096-entry two-bit prediction buffer for the SPEC89 benchmarks.

MCFarling's gshare Predictor

gshare = global history with index sharing

- McFarling noted (1993) that using global history information might be less efficient than simply using the address of the branch instruction, especially for small predictors.
- He suggests using both global history (BHR) and branch address by hashing them together. He proposed using the XOR of global branch history register (BHR) and branch address since he expects that this value has more information than either one of its components. The result is that this mechanism outperforms GAp scheme by a small margin.

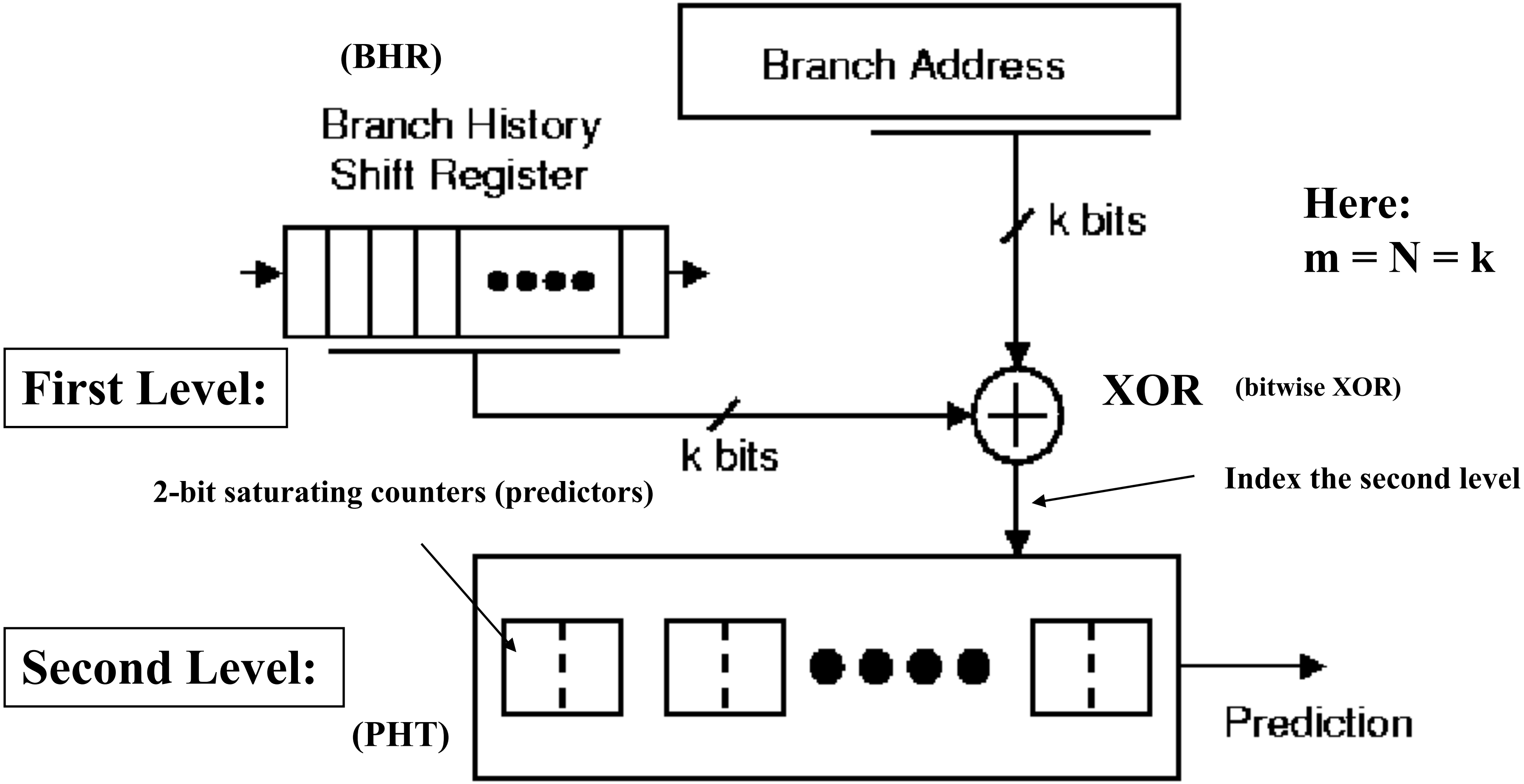
operation

- The hardware cost for k history bits is $k + 2 \times 2^k$ bits, neglecting costs for logic.

gshare is one one the most widely implemented two level dynamic branch prediction schemes

gshare Predictor

Branch and pattern history are kept globally. History and branch address are XORed and the result is used to index the pattern history table.



(BHR)

Branch History Shift Register

Branch Address

k bits

Here:
m = N = k

First Level:

XOR (bitwise XOR)

k bits

2-bit saturating counters (predictors)

Index the second level

Second Level:

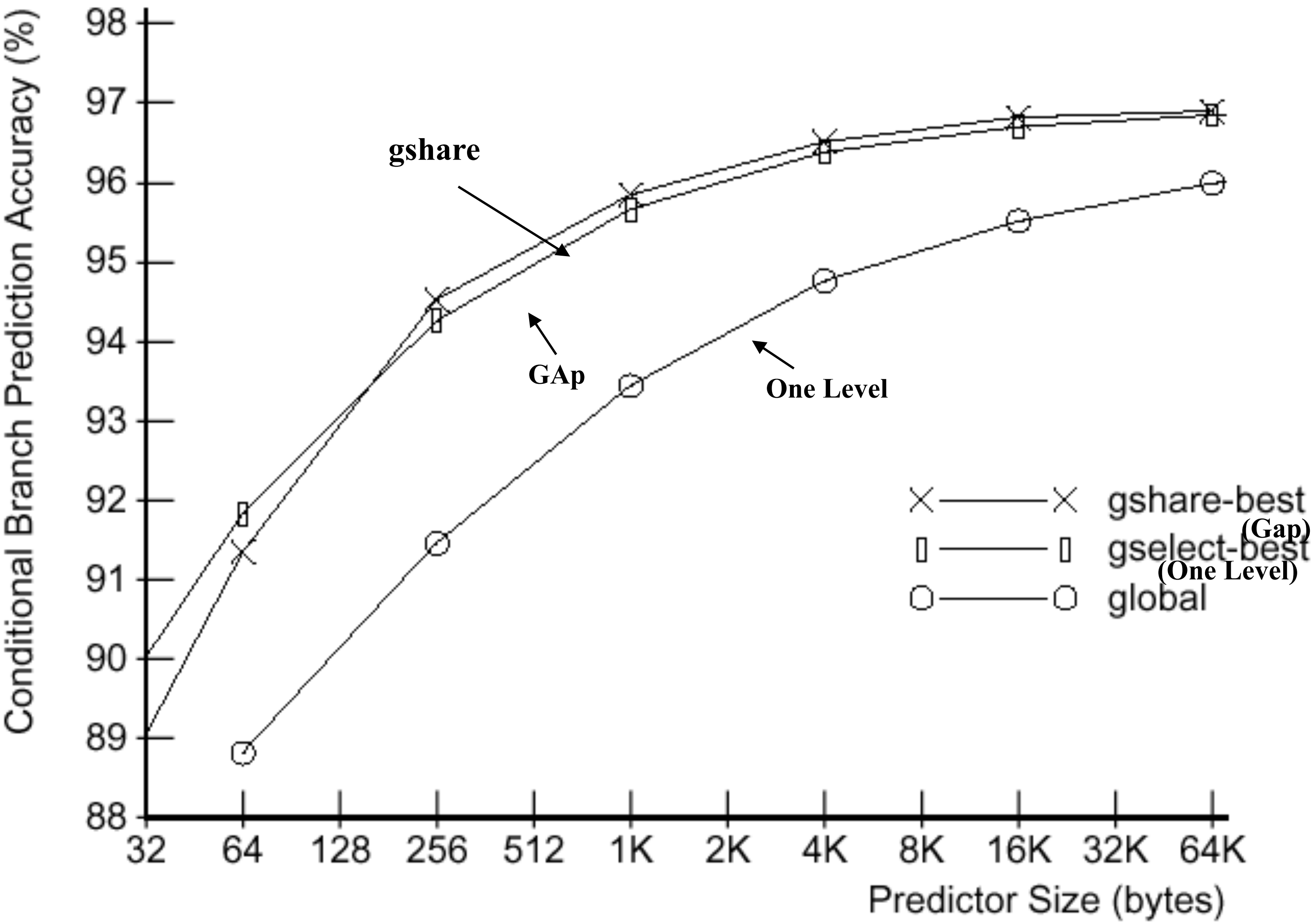
(PHT)

Prediction

One Pattern History Table (PHT) with 2^k entries (predictors)

gshare = global history with index sharing

gshare Performance



GAp = Global, Adaptive, per address branch predictor

Hybrid Predictors

(Also known as tournament or combined predictors)

- Hybrid predictors are simply combinations of two or more branch prediction mechanisms.
- This approach takes into account that different mechanisms may perform best for different branch scenarios.
- McFarling presented (1993) a number of different combinations of two branch prediction mechanisms.
- He proposed to use an additional 2-bit counter selector array which serves to select the appropriate predictor for each branch.
- One predictor is chosen for the higher two counts, the second one for the lower two counts.
- If the first predictor is wrong and the second one is right the counter is decremented, if the first one is right and the second one is wrong, the counter is incremented. No changes are carried out if both predictors are correct or wrong.

Intel Pentium 1

- It uses a single-level 2-bit Smith algorithm BHT associated with a four way associative BTB which contains the branch history information.
- The Pentium does not fetch non-predicted targets and does not employ a return address stack (RAS) for subroutine return addresses.
- It does not allow multiple branches to be in flight at the same time.
- Due to the short Pentium pipeline the misprediction penalty is only three or four cycles, depending on what pipeline the branch takes.

Intel P6,II,III

- Like Pentium, the P6 uses a BTB that retains both branch history information and the predicted target of the branch. However the BTB of P6 has 512 entries reducing BTB misses. Since the
- The average misprediction penalty is 15 cycles. Misses in the BTB cause a significant 7 cycle penalty if the branch is backward.
- To improve prediction accuracy a two-level branch history algorithm is used.
- Although the P6 has a fairly satisfactory accuracy of about 90%, the enormous misprediction penalty should lead to reduced performance. Assuming a branch every 5 instructions and 10% mispredicted branches with 15 cycles per misprediction the overall penalty resulting from mispredicted branches is 0.3 cycles per instruction. This number may be slightly lower since BTB misses take only seven cycles.

AMD K6

- Uses a two-level adaptive branch history algorithm implemented in a BHT (gshare) with 8192 entries (16 times the size of the P6).
- However, the size of the BHT prevents AMD from using a BTB or even storing branch target address information in the instruction cache. Instead, the branch target addresses are calculated on-the-fly using ALUs during the decode stage. The adders calculate all possible target addresses before the instruction are fully decoded and the processor chooses which addresses are valid.
- A small branch target cache (BTC) is implemented to avoid a one cycle fetch penalty when a branch is predicted taken.
- The BTC supplies the first 16 bytes of instructions directly to the instruction buffer.
- Like the Cyrix 6x86 the K6 employs a return address stack (RAS) for subroutines.
- The K6 is able to support up to 7 outstanding branches.
- With a prediction accuracy of more than 95% the K6 outperformed all other microprocessors when introduced in 1997 (except the Alpha).

Motorola PowerPC 750

- A dynamic branch prediction algorithm is combined with static branch prediction which enables or disables the dynamic prediction mode and predicts the outcome of branches when the dynamic mode is disabled.
- Uses a single-level Smith algorithm 512-entry BHT and a 64-entry Branch Target Instruction Cache (BTIC), which contains the most recently used branch target instructions, typically in pairs. When an instruction fetch does not hit in the BTIC the branch target address is calculated by adders.
- The return address for subroutine calls is also calculated and stored in user-controlled special purpose registers.
- The PowerPC 750 supports up to two branches, although instructions from the second predicted instruction stream can only be fetched but not dispatched.

The SUN UltraSparc

- Uses a dynamic single-level BHT Smith algorithm.
- It employs a static prediction which is used to initialize the state machine (saturated up and down counters).
- However, the UltraSparc maintains a large number of branch history entries (up to 2048 or every other line of the I-cache).
- To predict branch target addresses a branch following mechanism is implemented in the instruction cache. The branch following mechanism also allows several levels of speculative execution.
- The overall claimed performance of UltraSparc is 94% for FP applications and 88% for integer applications.

Branch Prediction comparisons					
	Pentium1	**Pentium Pro, II, III**	**AMD K6**	**Motorola PowerPC 750**	**Sun UltraSparc**
Branch type	2-level Smith	2-level Smith	2-level adaptive branch	1-level Smith	1-level Smith
BHT	2048 entries	4096 entries	8192 entries	512 entries	2048 entries
BTB	no	512 entries	512 entries	no	no
Static	no	no	no	Forward branches are not-taken and backward branches are taken	yes, until the state machine is initialized
Latency	3 - 4 cycles	7 - 15 cycles	1 - 4 cycles	3 - 4 cycles	9 to 14 cycles
Performance	80%	90%	95%	96%	94%
Additional features			branch target cache(BTC), Return address stack (RAS), up to 7 outstanding branches	branch target instruction cache (BTIC), Return address stack (RAS), up to 2 outstanding branches	

Speculative Execution

Speculative execution is a technique CPU designers use to improve CPU performance.

- It’s one of three components of out-of-order execution, also known as dynamic execution.
- Along with multiple branch prediction (used to predict the instructions most likely to be needed in the near future) and dataflow analysis (used to align instructions for optimal execution, as opposed to executing them in the order they came in), speculative execution delivered a dramatic performance improvement over previous Intel processors.

Because these techniques worked so well, they were quickly adopted by AMD, which used out-of-order processing beginning with the K5.

ARM’s focus on low-power mobile processors initially kept it out of the OOoE playing field, but the company adopted out-of-order execution when it built the Cortex A9 and has continued to expand its use of the technique with later, more powerful Cortex-branded CPUs.

Speculative Execution - Operation

Modern CPUs are all pipelined, which means they're capable of executing multiple instructions in parallel:

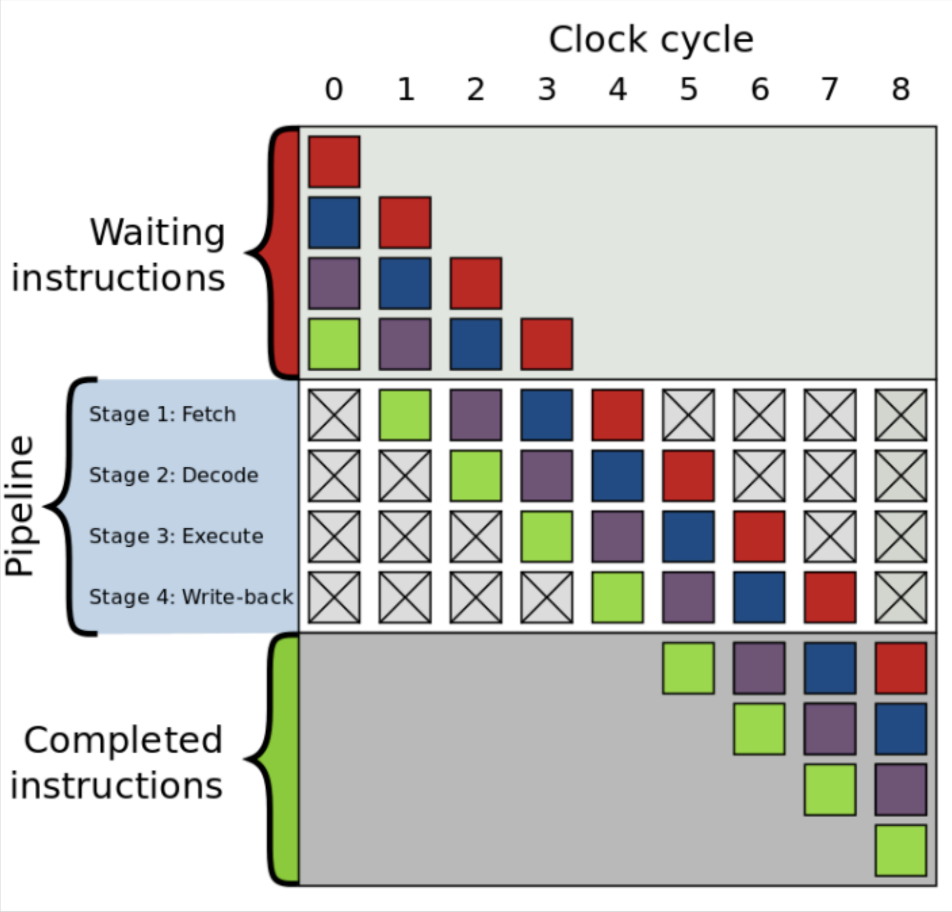


Figure 1: Example of 4-stage pipeline. The colored boxes represent instructions independent of each other

Imagine that the green block represents an if-then-else branch. The branch predictor calculates which branch is more likely to be taken, fetches the next set of instructions associated with that branch, and begins speculatively executing them before it knows which of the two code branches it'll be using.

These speculative instructions are represented as the purple box. If the branch predictor guessed correctly, then the next set of instructions the CPU needed are lined up and ready to go, with no pipeline stall or execution delay.

Speculative Execution - Operation

Without branch prediction and speculative execution, the CPU doesn’t know which branch it will take until the first instruction in the pipeline (the green box) finishes executing and moves to Stage 4. Instead of having moving straight from one set of instructions to the next, the CPU has to wait for the appropriate instructions to arrive. This hurts system performance since it’s time the CPU could be performing useful work.

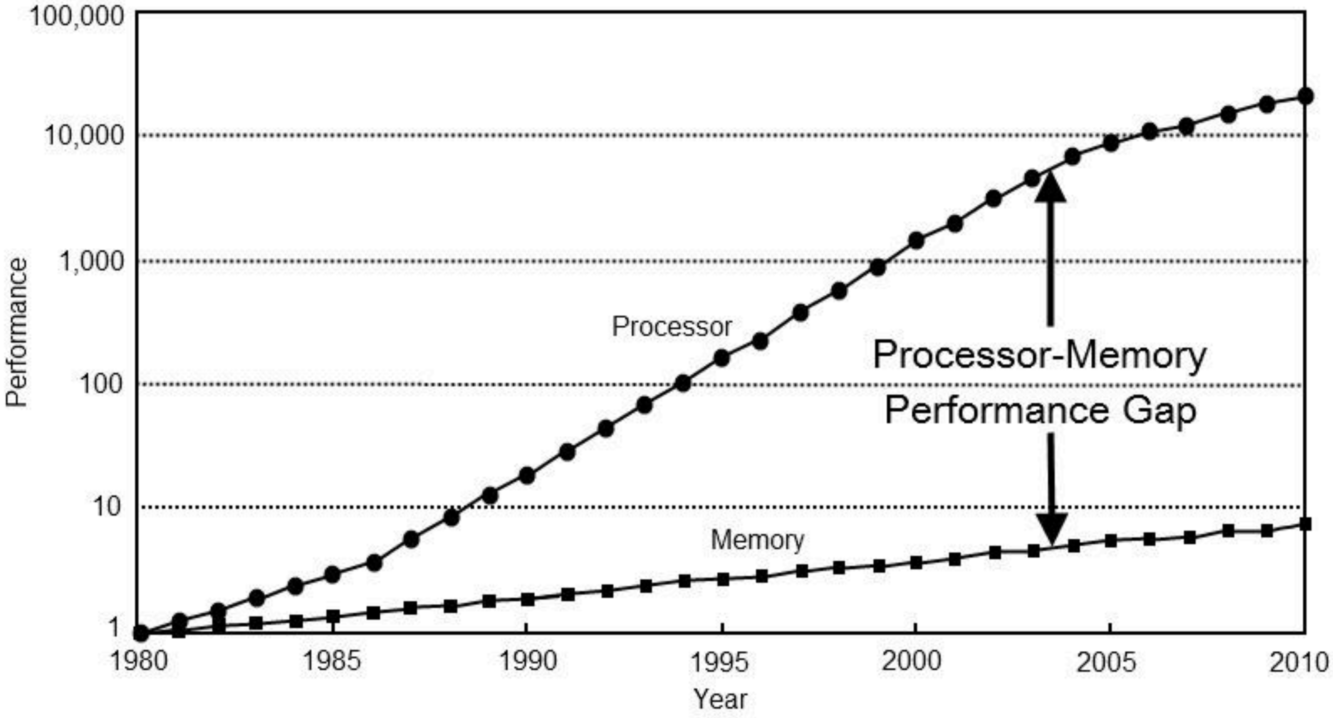
The reason its “speculative” execution, of course, is because the CPU might be wrong. If it is, the system loads the appropriate data and executes those instructions instead. But branch predictors aren’t wrong very often; accuracy rates are typically above 95 percent.

Speculative Execution - Advancement

Decades ago, before out-of-order execution was invented, CPUs were what we today call “in order” designs. Instructions executed in the order they were received, with no attempt to reorder them or execute them more efficiently. One of the major problems with in-order execution is that a pipeline stall stops the entire CPU until the issue is resolved.

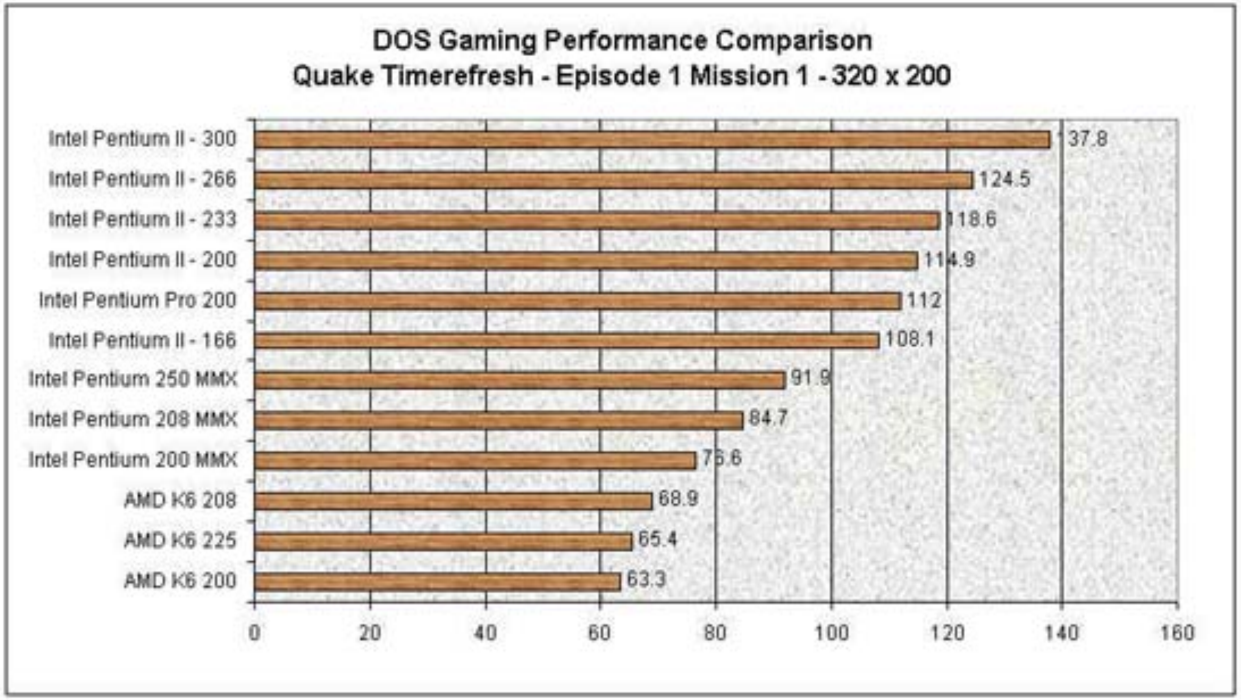
The other problem that drove the development of speculative execution was the gap between CPU and main memory speeds. The graph below shows the gap between CPU and memory clocks. As the gap grew, the amount of time the CPU spent waiting on main memory to deliver information grew as well. Features like L1, L2, and L3 caches and speculative execution were designed to keep the CPU busy and minimize the time it spent idling.

If memory could match the performance of the CPU there would be no need for caches.



Speculative Execution - Advancement

The combination of large off-die caches and out-of-order execution gave Intel’s Pentium Pro and Pentium II new performance opportunities. This graph from a 1997 Anandtech article shows the advantage clearly.



Thanks to the combination of speculative execution and large caches, the Pentium II 166 decisively outperforms a Pentium 250 MMX, despite the fact that the latter has a 1.51x clock speed advantage over the former.

Ultimately, it was the Pentium II that delivered the benefits of out-of-order execution to most consumers. The Pentium II was a fast microprocessor relative to the Pentium systems that had been top-end just a short while before. AMD was an absolutely capable second-tier option, but until the original Athlon launched, Intel had a lock on the absolute performance crown.

Speculative Execution - Advancement

The Pentium Pro and the later Pentium II were far faster than the earlier architectures Intel used. This wasn’t guaranteed. When Intel designed the Pentium Pro it spent a significant amount of its die and power budget enabling out of order execution. But the bet paid off, big time.

There are differences between how Intel, AMD, and ARM implement speculative execution, and those differences are part of why Intel is exposed to some security attacks in ways that the other vendors aren’t. But speculative execution, as a technique, is simply far too valuable to stop using. Every single high-end CPU architecture today — AMD, ARM, IBM, Intel, SPARC — uses out-of-order execution. And speculative execution, while implemented differently from company to company, is used by each of them. Without speculative execution, out-of-order execution as we know it wouldn’t function.

Speculative Execution – Meltdown, Spectre

In 2018, two new security attacks were identified by research teams for x86 and some ARM processors. The amazing thing is that these vulnerabilities have been present in microprocessors since 1995.

Meltdown and Spectre exploit critical vulnerabilities in these modern processors. These hardware vulnerabilities allow programs to steal data which is currently processed on the computer. While programs are typically not permitted to read data from other programs, a malicious program can exploit Meltdown and Spectre to get hold of secrets stored in the memory of other running programs

The reason **Meltdown** causes such unique headaches for Intel is because Intel allows *speculative* execution to access privileged memory a user-space application would never be allowed to touch.

Code that's running under speculative execution *doesn't* do the check whether or not memory accesses from cache are accessing privileged memory. It starts running the instructions without the privilege check, and when it's time to commit to whether or not the speculative execution should be continued, the check will occur. But during that window, you've got the opportunity to run a batch of instructions against the cache without privilege checks. So you can write code with the right sequence of branch instructions to get branch prediction to work the way you want it to; and then you can use that to read memory that you shouldn't be able to read.

Speculative Execution – Meltdown, Spectre

The speculative prediction implementations of other CPU vendors don’t allow user-space applications to probe the contents of kernel space memory at any point. The only way to mitigate Meltdown in software is to force the system to perform a full context switch every time it switches between kernel and user memory space. The reason the performance impact from Meltdown is so varied is that how much this patch hurts is a function of how often an application has to context switch. The performance issues, however, appear to be limited to servers and have not generally been seen on the consumer side — at least, not very much.

Spectre is a vulnerability that affects modern microprocessors that perform branch prediction. It tricks a program into accessing arbitrary locations (shadow cache) in the program's memory space. An attacker may read the content of accessed memory, and thus potentially obtain sensitive data. On most processors, the speculative execution resulting from a branch misprediction may leave observable side effects that may reveal private data to attackers. For example, if the pattern of memory accesses performed by such speculative execution depends on private data, the resulting state of the data cache constitutes a side channel through which an attacker may be able to extract information about the private data using a timing attack.

Speculative Execution – Meltdown, Spectre - mitigation

One of the mitigation strategies we’ve seen proposed, particularly more recently, is disabling Hyper-Threading. Apple has issued an update, notifying its users that they can disable hyperthreading if they want to limit the ability of data to leak between multiple threads within the same CPU core.
They’ve also stated that this can hit performance by up to 40 percent. That’s an extreme case because HT isn’t generally “worth” that much performance to an Intel CPU — we’d expect the typical impact to be in the 20-30 percent range

Some of the patches associated with fixing Spectre and Meltdown have also had performance impacts, though some of the impacts were then reduced by further patches, and the degree of slowdown is workload and, to some extent, CPU architecture dependent in the first place.

In the long run, we expect AMD, Intel, and other vendors to continue patching these issues as they arise, with a combination of hardware, software, and firmware updates. Conceptually, side channel attacks like these are extremely difficult, if not impossible, to prevent. Specific issues can be mitigated or worked around, but the nature of speculative execution means that a certain amount of data is going to leak under specific circumstances. It may not be possible to prevent it without giving up far more performance than most users would ever want to accept.

Speculative Execution – Meltdown, Spectre - mitigation

Companies began to release patches for CPU microcodes, operating systems, and individual programs back in January 2018, looking to put a stop to these nuisances. Unfortunately, Spectre and Meltdown are hardware vulnerabilities; they exist at the hardware level, so they can’t be cured completely with software patches.

Thus, one of the patches was implemented inside the Linux OS core, but it was slowing the system down too much, so after a while it was removed from the code.
Intel has introduced 9th generation processors that include permanent fixes to the vulnerabilities that Meltdown and Spectre exploit.

At Intel's Fall Desktop Launch event, they stated "...[our] new desktop processors include protections for the security vulnerabilities commonly referred to as 'Spectre,' 'Meltdown,' and 'L1TF.' These protections include a combination of the hardware design changes we announced earlier this year as well as software and microcode updates."