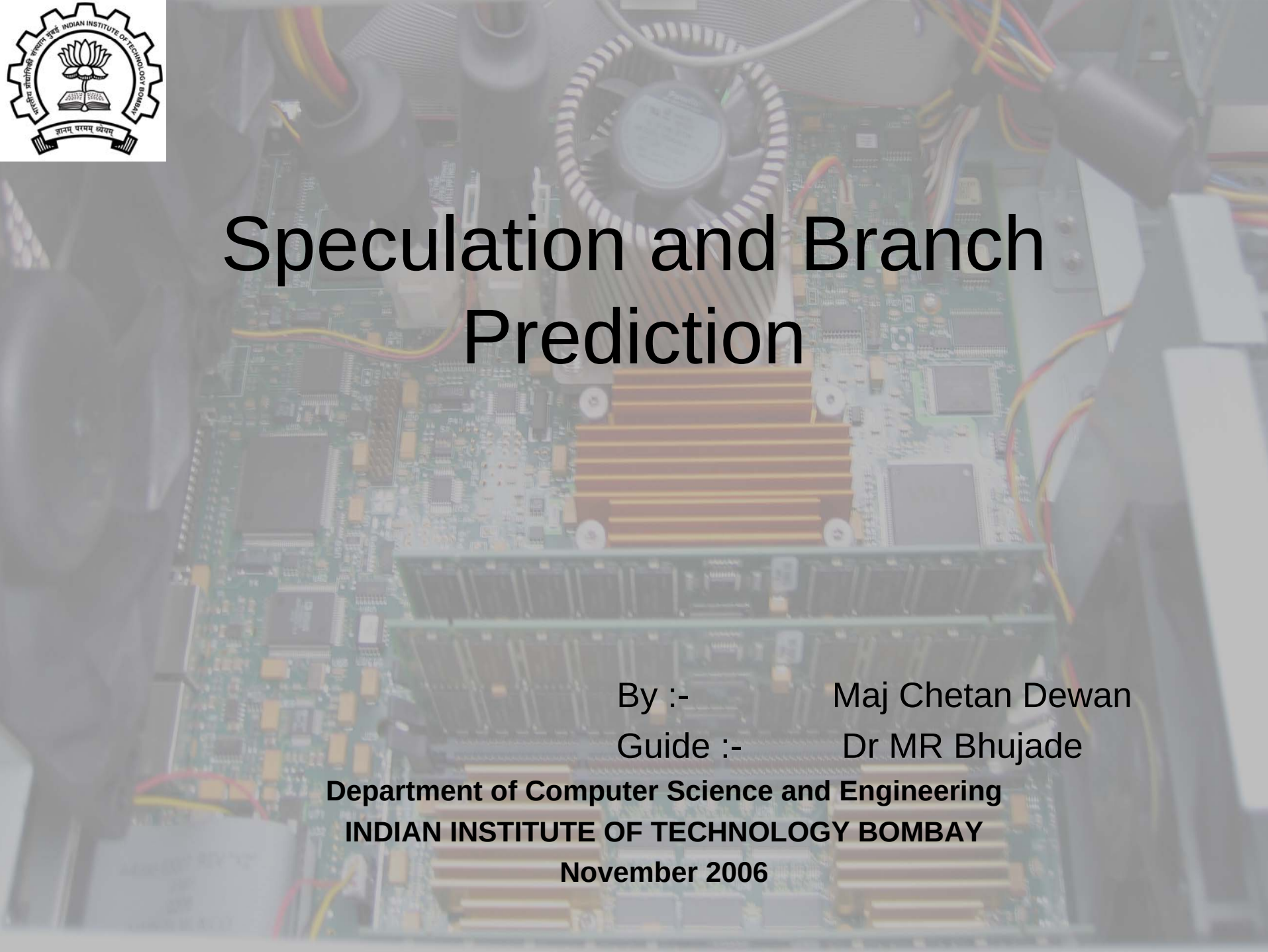


Speculation and Branch Prediction

By :- Maj Chetan Dewan

Guide :- Dr MR Bhujade

Department of Computer Science and Engineering

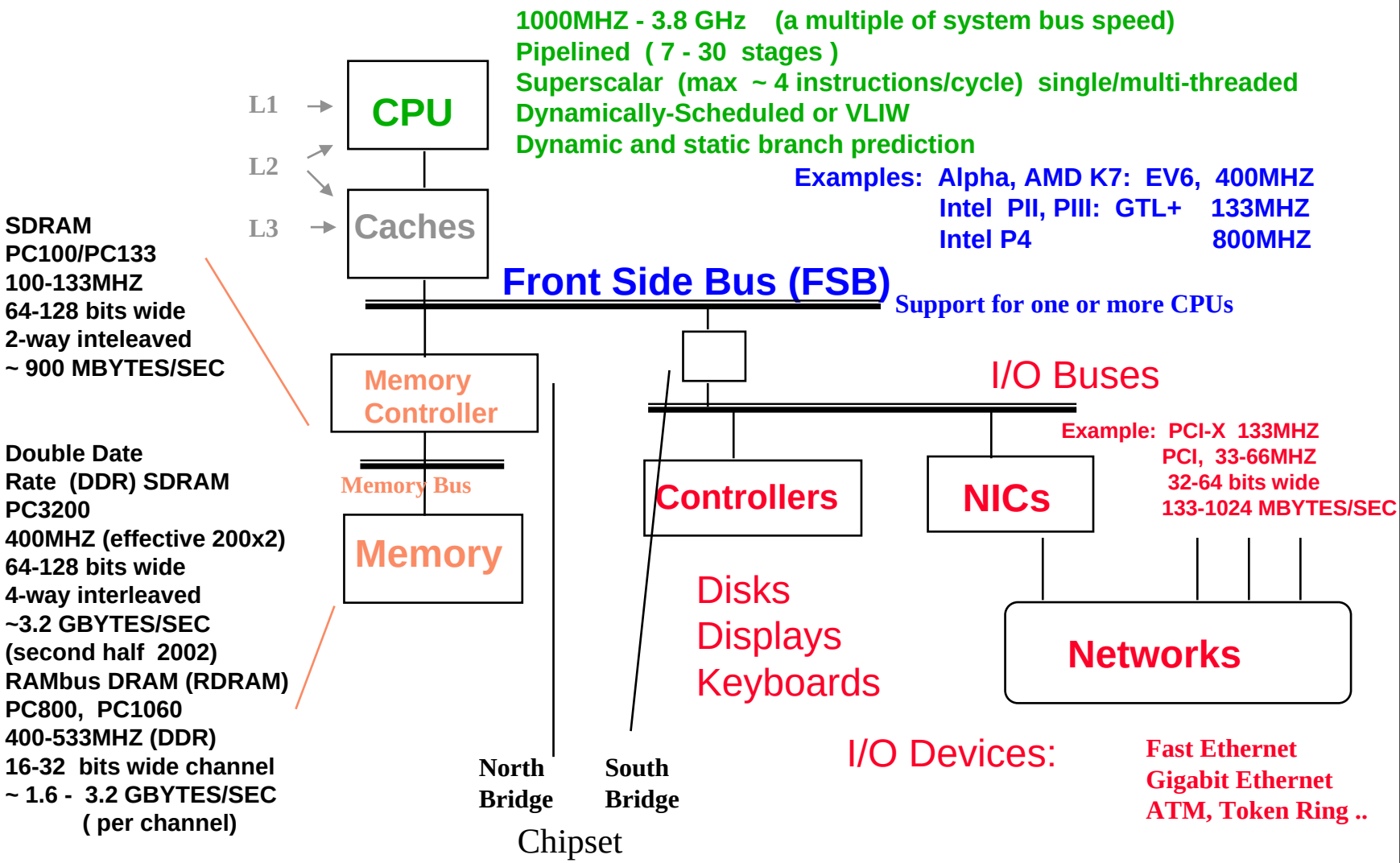
INDIAN INSTITUTE OF TECHNOLOGY BOMBAY

November 2006

Conduct

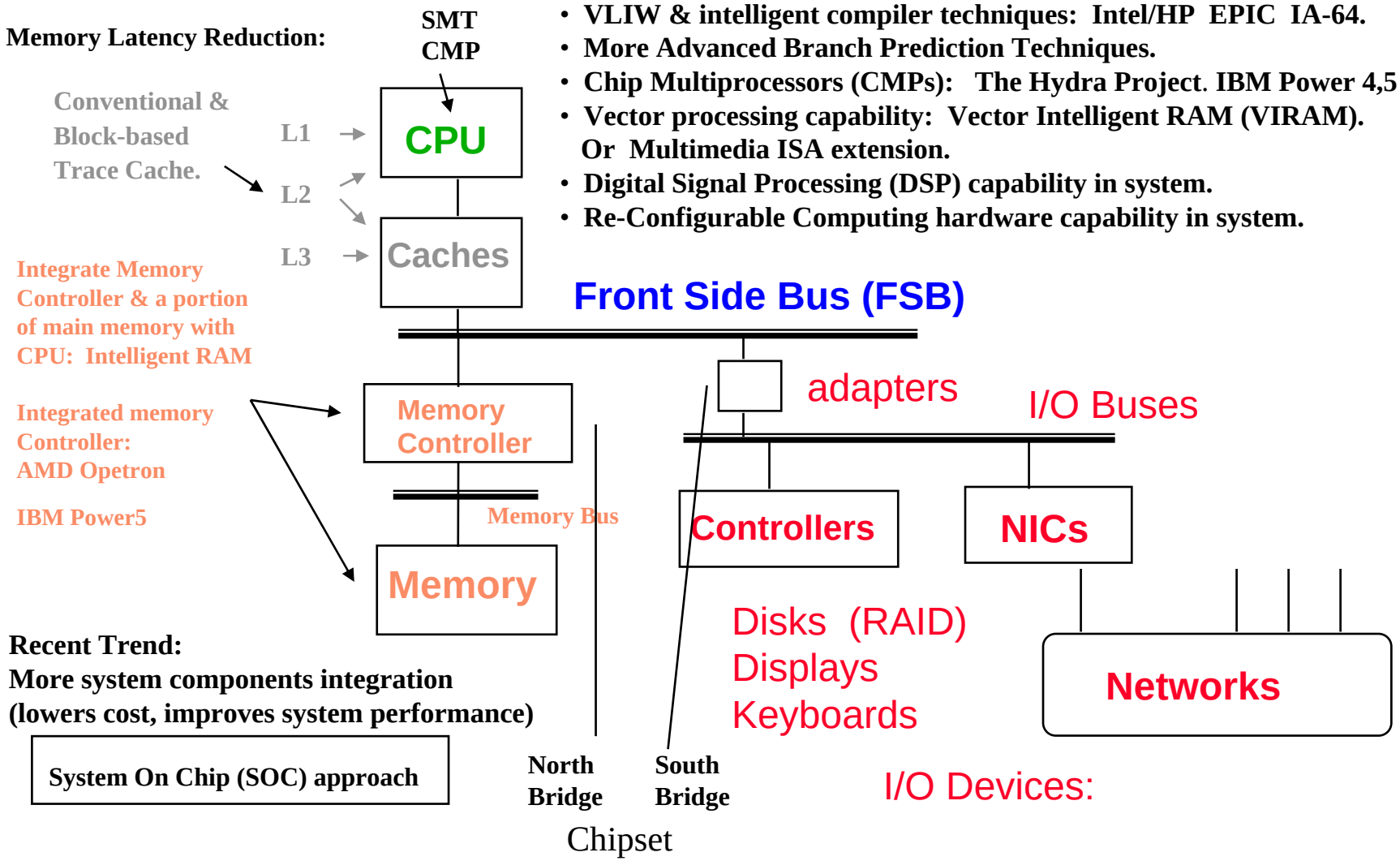
- Backdrop
- Introduction
- Speculation
- Prediction
- Modern trends
- Conclusion

Mainstream Computer System Components

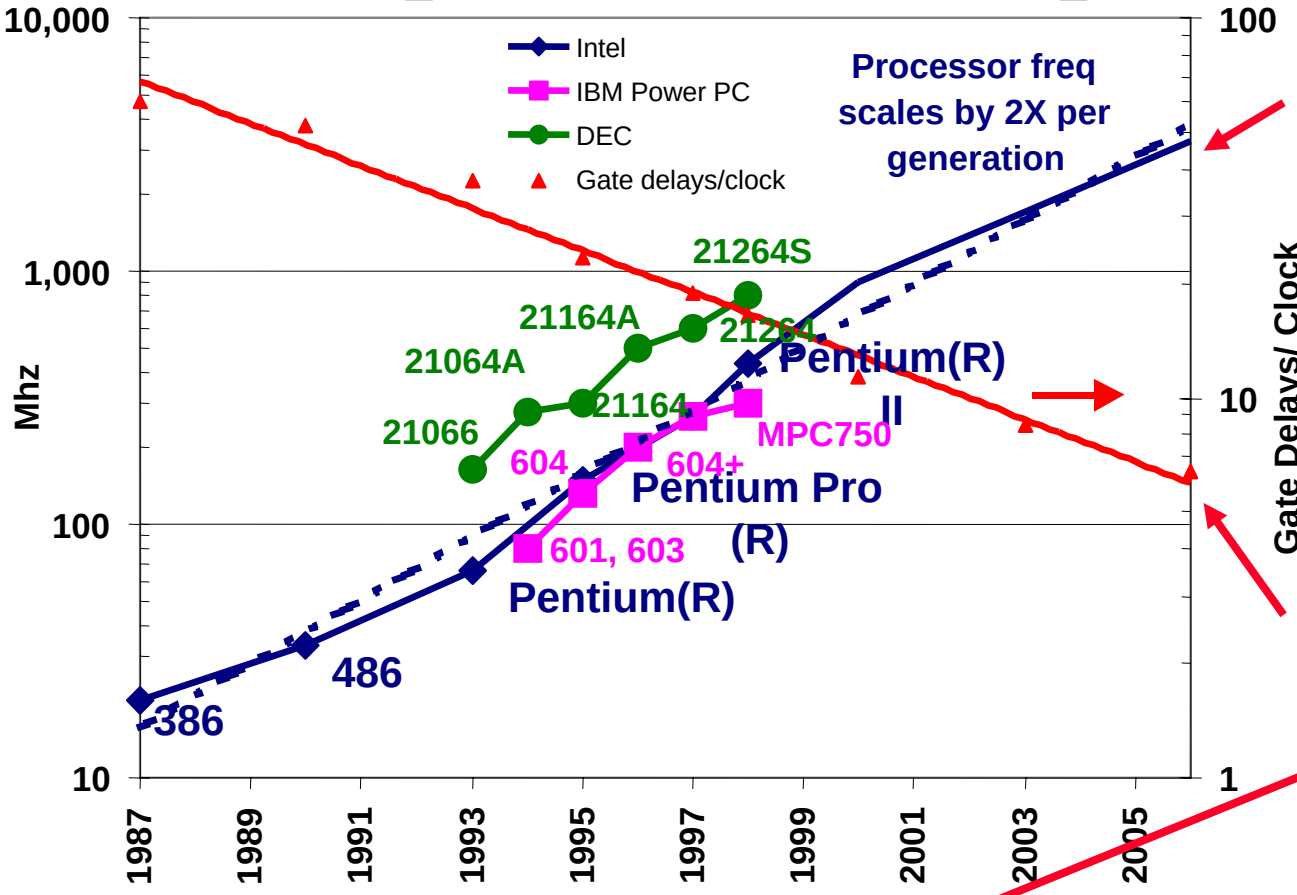


Enhancing Computing Performance & Capabilities: How To ?

- Support for Simultaneous Multithreading (SMT): Intel HT.
- VLIW & intelligent compiler techniques: Intel/HP EPIC IA-64.
- More Advanced Branch Prediction Techniques.
- Chip Multiprocessors (CMPs): The Hydra Project. IBM Power 4,5
- Vector processing capability: Vector Intelligent RAM (VIRAM). Or Multimedia ISA extension.
- Digital Signal Processing (DSP) capability in system.
- Re-Configurable Computing hardware capability in system.



Microprocessor Frequency Trend



Realty Check:
Clock frequency scaling is slowing down!

Why?
1- Power leakage
2- Clock distribution delays

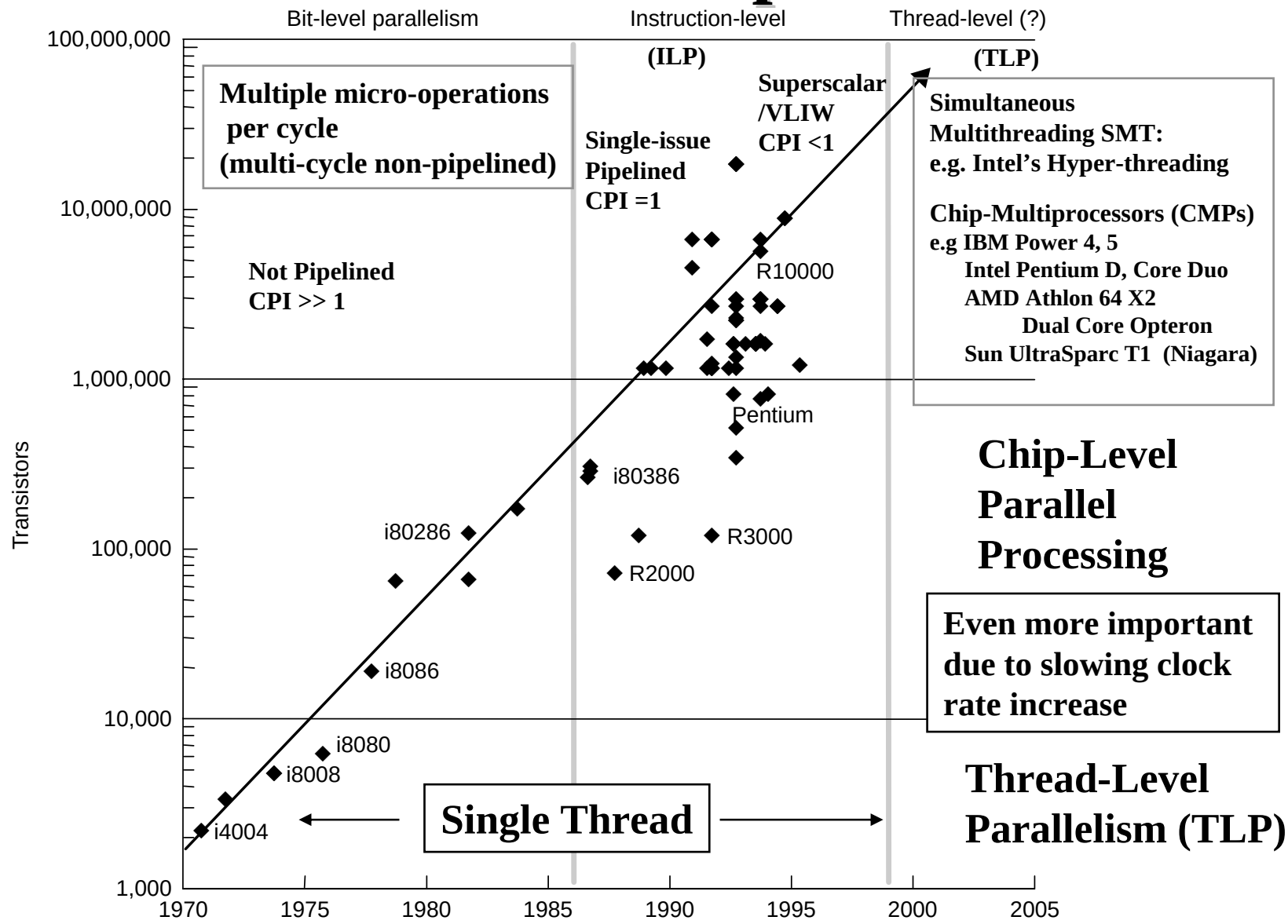
Result:
Deeper Pipelines
Longer stalls
Higher CPI (lowers effective performance per cycle)

1. Frequency used to double each generation
2. Number of gates/clock reduce by 25%
3. Leads to deeper pipelines with more stages

(e.g Intel Pentium 4E has 30+ pipeline stages)

Possible Solutions?
- Branch Prediction, Speculative execution
- Exploit Thread-Level Parallelism (TLB) at the chip level (SMT/CMP)
- Utilize/integrate more-specialized computing elements other than GPPs

Parallelism: Microprocessor



Improving microprocessor generation performance by exploiting more levels of parallelism

Architectures Over the Years

General Purpose Processor (GPP)

Single Threaded

CISC Machines
instructions take variable times to complete

RISC Machines (microcode)
simple instructions, optimized for speed

RISC Machines (pipelined)
same individual instruction latency
greater throughput through instruction "overlap"

Superscalar Processors
multiple instructions executing simultaneously

Multithreaded Processors
additional HW resources (regs, PC, SP)
each context gets processor for x cycles

VLIW
"Superinstructions" grouped together
decreased HW control complexity
(Single or Multi-Threaded)

CMPs
Single Chip Multiprocessors
duplicate entire processors
(e.g IBM Power 4/5, AMD X2, Intel Core 2)

SIMULTANEOUS MULTITHREADING (SMT)
multiple HW contexts (regs, PC, SP)
each cycle, any context may execute

e.g. Intel's HyperThreading (P4)

SMT/CMPs
(e.g. IBM Power5 , Intel Pentium D, Sun Niagara - (UltraSparc T1))

CPU Execution Time: The CPU Equation

- A program is comprised of a number of instructions executed , I
  - Measured in: instructions/program

- The average instruction executed takes a number of *cycles per instruction (CPI)* to be completed.
  - Measured in: cycles/instruction, CPI

Or Instructions Per Cycle (IPC):
IPC= 1/CPI

- CPU has a fixed clock cycle time C = 1/clock rate
  - Measured in: seconds/cycle

- CPU execution time is the product of the above three parameters as follows:

Executed

$$\text{CPU time} = \frac{\text{Seconds}}{\text{Program}} = \frac{\text{Instructions}}{\text{Program}} \times \frac{\text{Cycles}}{\text{Instruction}} \times \frac{\text{Seconds}}{\text{Cycle}}$$

$$T = I \times CPI \times C$$

execution Time per program in seconds

Number of instructions executed

Average CPI for program

CPU Clock Cycle

(This equation is commonly known as the CPU performance equation)

Factors Affecting CPU Performance

	Instruction Count I	CPI IPC	Clock Cycle C
Program	X	X	
Compiler	X	X	
Instruction Set Architecture (ISA)	X	X	
Organization (Micro-Architecture)		X	X
Technology			X

$$T = I \times CPI \times C$$

In-Order Single-Issue Integer Pipeline Ideal Operation

(No stall cycles)

Fill Cycles = number of stages -1

	Clock Number → Time in clock cycles →								
Instruction Number	1	2	3	4	5	6	7	8	9
Instruction I	IF	ID	EX	MEM	WB				
Instruction I+1		IF	ID	EX	MEM	WB			
Instruction I+2			IF	ID	EX	MEM	WB		
Instruction I+3				IF	ID	EX	MEM	WB	
Instruction I +4					IF	ID	EX	MEM	WB

4 cycles = n -1

Time to fill the pipeline

First instruction, I Completed

Last instruction, I+4 completed

MIPS Pipeline Stages:

IF = Instruction Fetch

ID = Instruction Decode

EX = Execution

MEM = Memory Access

WB = Write Back

n= 5 pipeline stages

Ideal CPI =1

(or IPC =1)

In-order = instructions executed in original program order

Ideal pipeline operation without any stall cycles

Pipeline Hazards/Constraints

- **Hazards are situations in pipelining which prevent the next instruction in the instruction stream from executing during the designated clock cycle possibly resulting in one or more stall (or wait) cycles.**

  i.e A resource the instruction requires for correct execution is not available in the cycle needed

- **Hazards reduce the ideal speedup (increase CPI > 1) gained from pipelining and are classified into three classes:**

  Resource Not available: Hardware Component

  - ***Structural hazards*: Arise from hardware resource conflicts when the available hardware cannot support all possible combinations of instructions.**

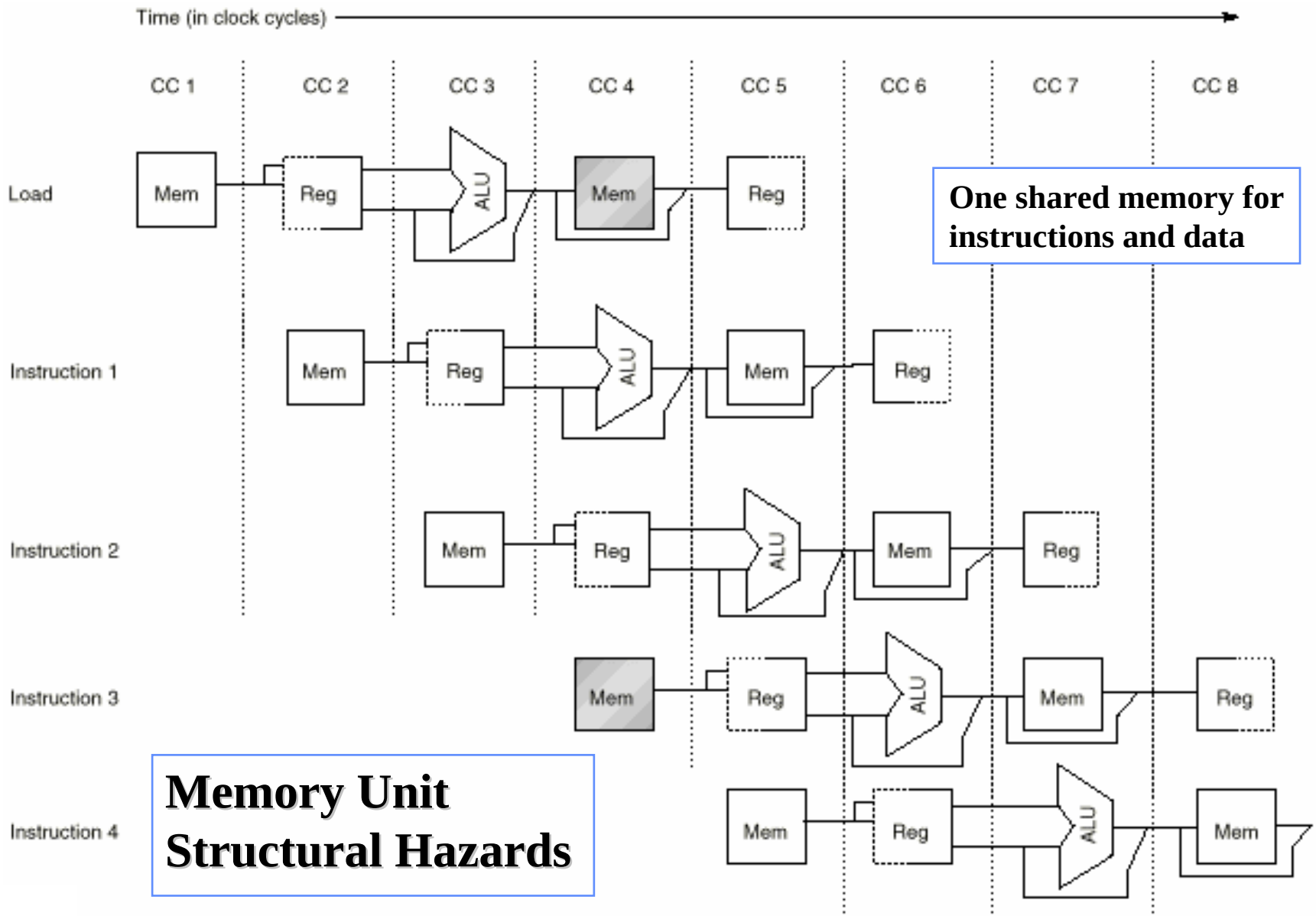
  Correct Operand (data) value

  - ***Data hazards*: Arise when an instruction depends on the result of a previous instruction in a way that is exposed by the overlapping of instructions in the pipeline**

  Correct PC

  - ***Control hazards*: Arise from the pipelining of conditional branches and other instructions that change the PC**

Reduce constraints to increase exploitable ILP


Time (in clock cycles)
CC 1
CC 2
CC 3
CC 4
CC 5
CC 6
CC 7
CC 8
Load
Instruction 1
Instruction 2
Instruction 3
Instruction 4
Mem
Reg
ALU
One shared memory for instructions and data
Memory Unit Structural Hazards
A machine with only one memory port will generate a conflict whenever a memory reference occurs.

Data and Control Hazard

1. Store R4, A — R4 → A
2. Sub R3, A — R3 ← R3 -contentsA
3. Store R3, A — R3 → A
4. Add R3, A — R3 ← R3+contents A
5. Store R3, A — R3 → A

RAW---------1&2 Inst 2 to read after 1 writes

WAR---------2 &3 Inst 3 should write after 2 reads

WAW--------3&5 Inst 5 should write after 3 writes

Pipelining and Exploiting Instruction-Level Parallelism (ILP)

- Pipelining increases performance by overlapping the execution of independent instructions.
- The CPI of a real-life pipeline is given by (assuming ideal memory):

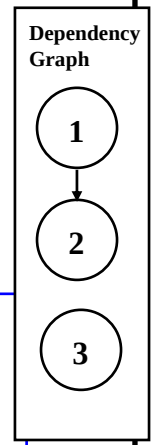
Pipeline CPI = Ideal Pipeline CPI + Structural Stalls + RAW Stalls + WAR Stalls + WAW Stalls + Control Stalls

- ***A basic instruction block*** is a straight-line code sequence with no branches in, except at the entry point, and no branches out except at the exit point of the sequence .
- The amount of parallelism in a basic block is limited by instruction dependence present and size of the basic block.
- In typical integer code, dynamic branch frequency is about 15% (average basic block size of 7 instructions).

Dynamic Pipeline Scheduling: *The Concept*

(Out-of-order execution) i.e Start of instruction execution is not in program order

- Dynamic pipeline scheduling overcomes the limitations of in-order pipelined execution by allowing out-of-order instruction execution.
- Instruction are allowed to start executing out-of-order as soon as their operands are available.
  - Better dynamic exploitation of instruction-level parallelism (ILP).



Example:

True Data Dependency

```
1  DIV.D   F0, F2, F4
2  ADD.D   F10, F0, F8
3  SUB.D   F12, F8, F14
```

Does not depend on DIV.D or ADD.D

In the case of in-order pipelined execution SUB.D must wait for DIV.D to complete which stalled ADD.D before starting execution
In out-of-order execution SUBD can start as soon as the values of its operands F8, F14 are available.

- This implies allowing out-of-order instruction commit (completion).
- May lead to imprecise exceptions if an instruction issued earlier raises an exception.

Speculative Execution

Speculative execution is the processor's ability to execute instructions that exist beyond a conditional branch that has not yet been resolved, and ultimately to commit the results in the order of the original instruction stream

Speculation and Prediction

Prediction is targeted at instruction fetch ie Prediction is de-coupled from the decision to execute fetched instructions (ref fig 2.2 above), whereas Prediction helps boost the issue rate. **Speculation refers to the execution of predicted instructions.**

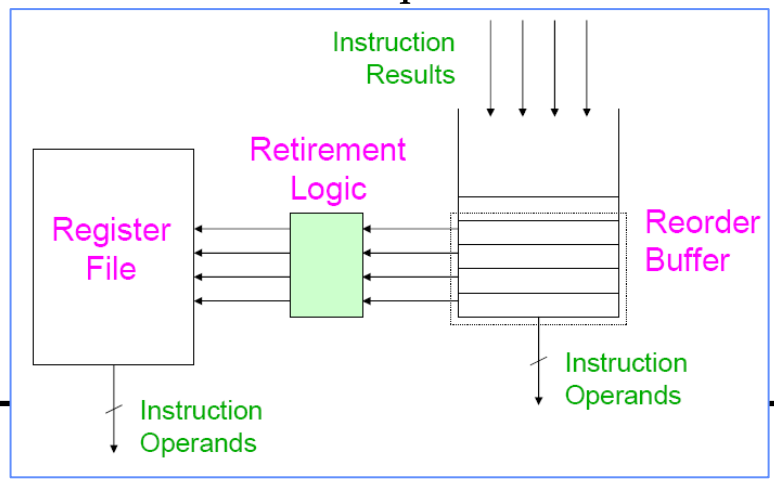
Hardware based speculation an extension of dynamic scheduling

- Branch prediction -- to select instructions to be speculatively executed
- Dynamic scheduling
- Execution
- Commitment--update machine state
- Exception handling

In basic block Prediction can increase issue rate but not completion rate. Boosting issue rate by itself is insufficient, the completion rate also has to be increased to keep up with the issue rate. For this we require speculative execution.

Reorder Buffer

- Require to separate instruction execution from instruction commitment.
- Therefore *we* compute on a need-to-know basis until speculation outcome is determined.
- At **commitment the registers are updated** and hence the **machine state also gets updated.**
- Criteria: **commitment in program order.**
- Therefore have to **reorder the instructions** that complete out-of-order. (Reorder Buffer).
- The Reorder Buffer is a FIFO circular queue which does not take care of flow dependences



Speculation

Compiler
illegal to legal-yes
Unsafe to safe-no

```
10: if (r1)
20: r2 = load r3;
else
30: r4 = r2 + r3;
```

Branch 20/30 depends on 10

Static scheduling 20&30 moved before 10

1

Preserve pgm semantics
Code motion illegal as 30:r2 initial val over written by 20

2

Exception handling

Load instruction in 20 can cause exception
Handling to be done only if result required
Else may terminate pgm

Sol
Register renaming by compiler

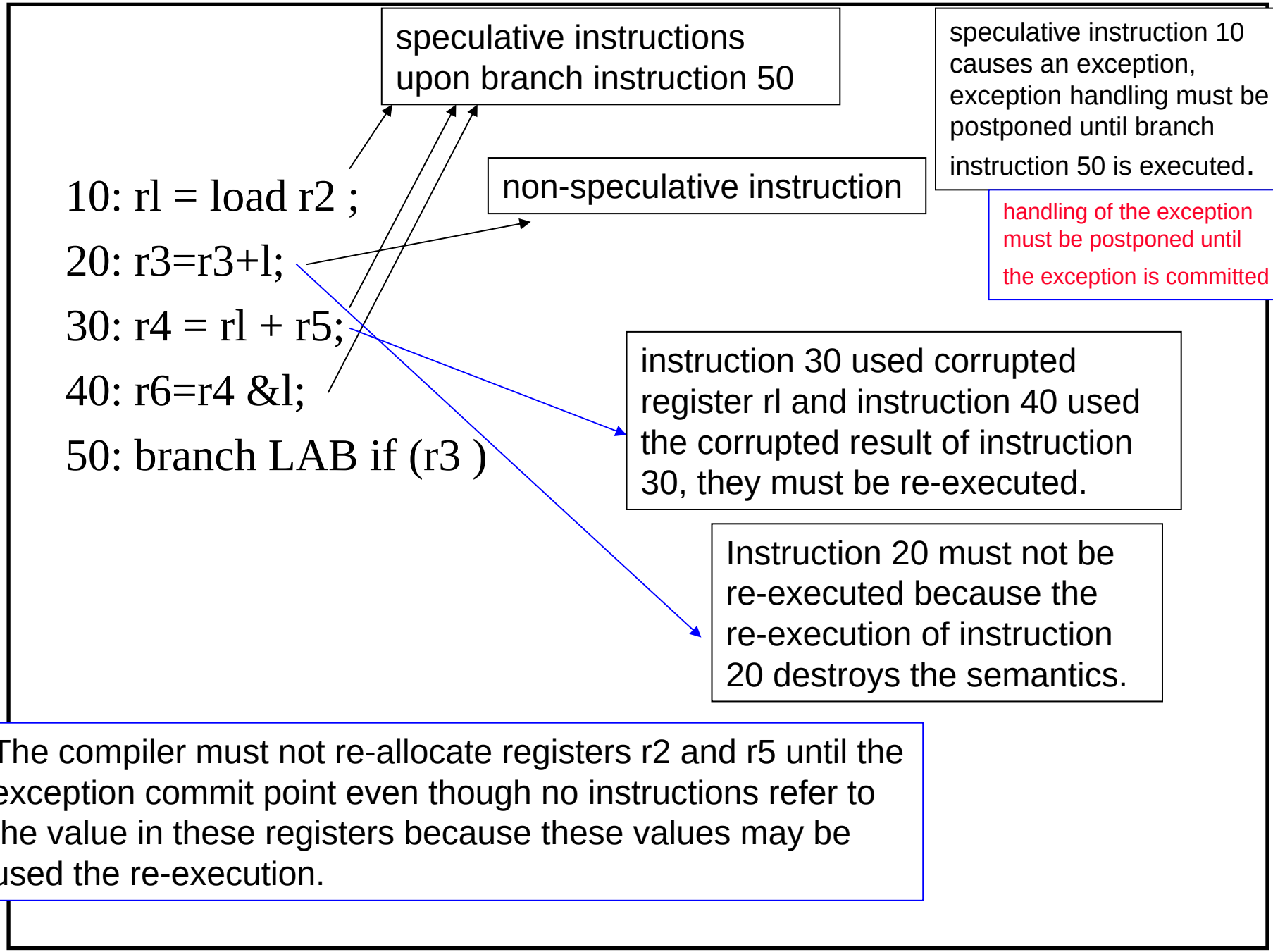
```
20: r5 = load r3;
10: if (rl)
40: r2 = r5;
else
30: r4 = r2 + r3;
```

Unsafe code motion can increase exceptions

restarting the process.

problem

- to select instructions which must be re-executed.
- operands of the re-execution instructions in the recovery process must be available


speculative instructions upon branch instruction 50
speculative instruction 10 causes an exception, exception handling must be postponed until branch instruction 50 is executed.
non-speculative instruction
handling of the exception must be postponed until the exception is committed
10: rl = load r2 ;
20: r3=r3+l;
30: r4 = rl + r5;
40: r6=r4 &l;
50: branch LAB if (r3)
instruction 30 used corrupted register rl and instruction 40 used the corrupted result of instruction 30, they must be re-executed.
Instruction 20 must not be re-executed because the re-execution of instruction 20 destroys the semantics.
The compiler must not re-allocate registers r2 and r5 until the exception commit point even though no instructions refer to the value in these registers because these values may be used the re-execution.

Branches Why ?

Most computer programs respond to a user, there is no way around the fact that portions of a program need to be executed conditionally

Why Predict Conditional Branch Directions-----?

Speculative execution to enhance instruction-level parallelism.

Compiler Optimizations

Hardware reasons.

Pipeline penalties occur because of the timing of
Branch target address generation
– PC-relative address generation "can" occur after instruction fetch
Branch condition resolution
– What cycle is the condition known?

Speculative Performance

•Branches handled by branch prediction and speculative execution

•In case of a misprediction, the speculative execution forces a complete reload of the pipeline and possibly suffers from additional penalty cycles for cancelling the wrongly issued and executed instructions.

For example mispredicted branches incur a misprediction penalty of at least 11 cycles with an average penalty of 15 cycles in the Pentium II processor.

Two types of branches: unconditional and conditional

some branches are unpredictable, resulting in high misprediction rates.
Still speculatively executed in contemporary microprocessors.

Predictor Types

- **Trivial prediction**
- **Static**
- **Semi Static**
- **Dynamic**

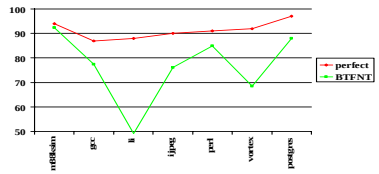
•early implementations of SPARC and MIPS
•always predicted that a branch (or unconditional jump) would not be taken, so they always fetched the next sequential instruction. Only when the branch or jump was evaluated did the instruction fetch pointer get set to a non sequential address.

Simple and complex

Simple heuristics
- Always taken
- Always not taken
- Backwards taken / Forward not taken

Programmer provided directives

Relies on the compiler

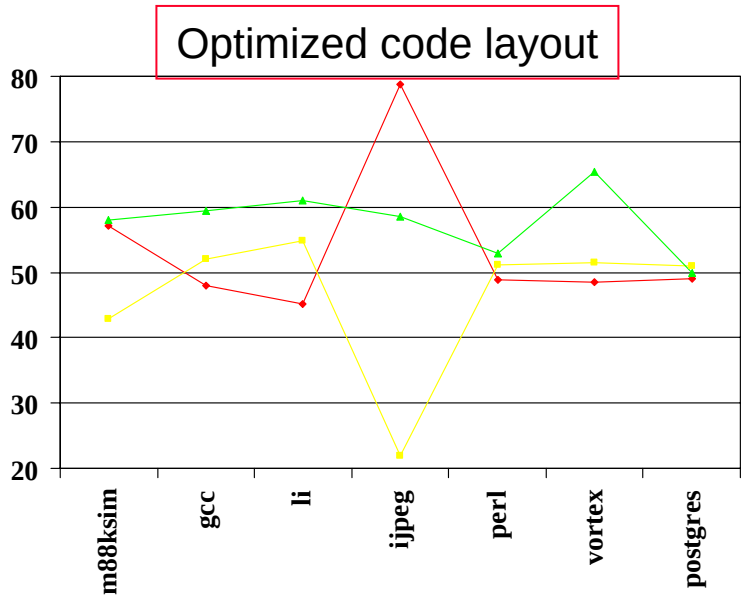


Use profile information from previous program runs
- Branches tend to behave in a fixed way
- Branches tend to behave in the same way across different program executions

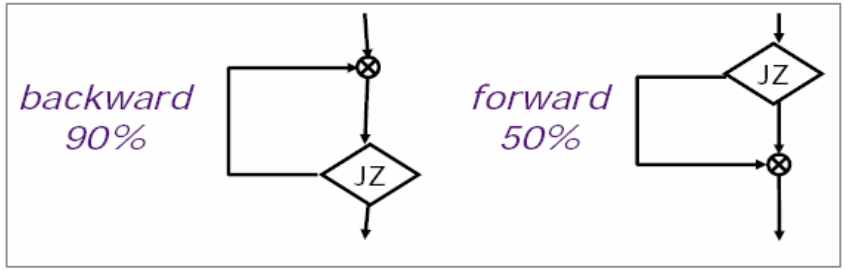
Performance metric:
- How close to a perfect static predictor
  - **Best direction to statically predict a branch**
- Real static predictors
  - **Based on a past dataset / group of datasets**

Simple Static prediction

- **Processors that implement "Static prediction" predict that backward pointing branches will be taken (assuming that the backwards branch is the bottom of a program loop), and forward-pointing branches will not be taken (assuming they are early exits from the loop or other processing code).**
- **For a loop that executes many times, this only mispredicts the very last branch of the loop.**



Both the Motorola MPC7450 (G4e) and the Intel Pentium 4 use this technique

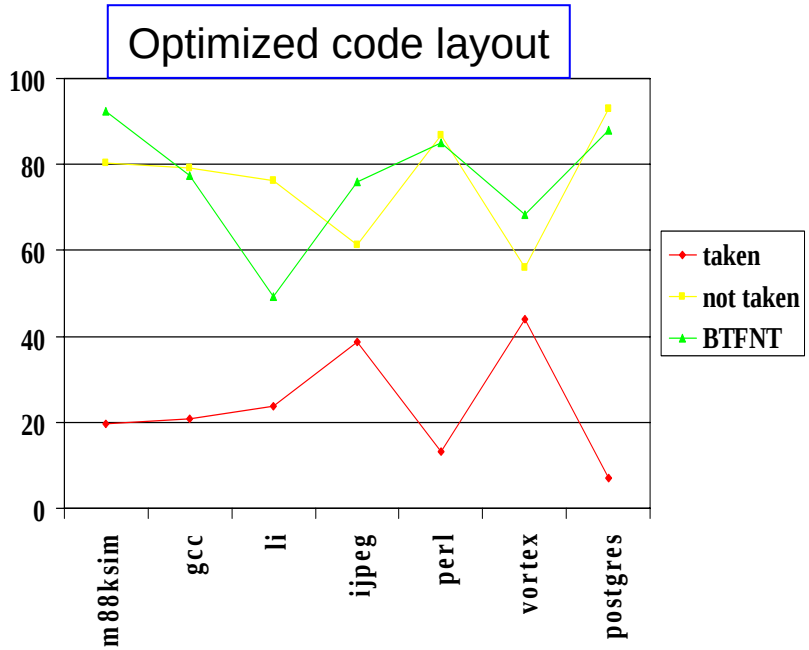


Static prediction --- fall-back technique in processors with dynamic branch prediction when there isn't any information for dynamic predictors to use.

Complex Static Predictors

- **Based on:**
  - Control flow analysis of the code to determine loops
  - Classify branches

Misprediction rate about 20%



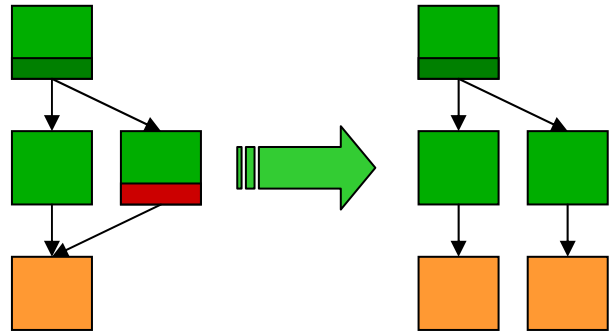
Pointer comparison: false
Avoid executing subroutine calls
exception handlers
Avoid returning from a subroutine
recursion base case
Avoid blocks containing a *store* instruction
Go towards loop headers, avoid exiting loops
Favor reuse of the branch operand

Improving Static Predictors

- The compiler can lay out the code to match the static prediction
- Code replicating techniques

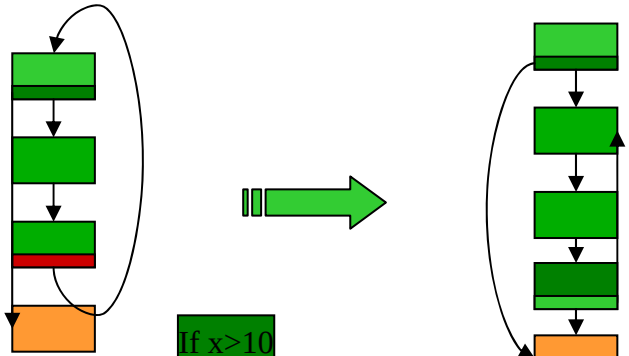
Unconditional jump elimination

Replicate code after the if-then-else convergence in both conditional paths Change unconditional loop edges to conditional ones



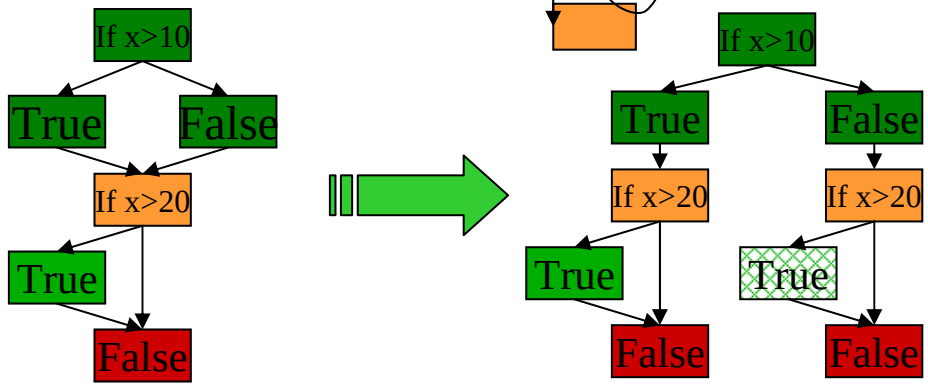
Conditional branch elimination

In some situations the branch outcome is knownTest a variable which has not been modified or has just been set



Static prediction using branch history information

Replicate if-then-else bodies to know the previous branch outcomes

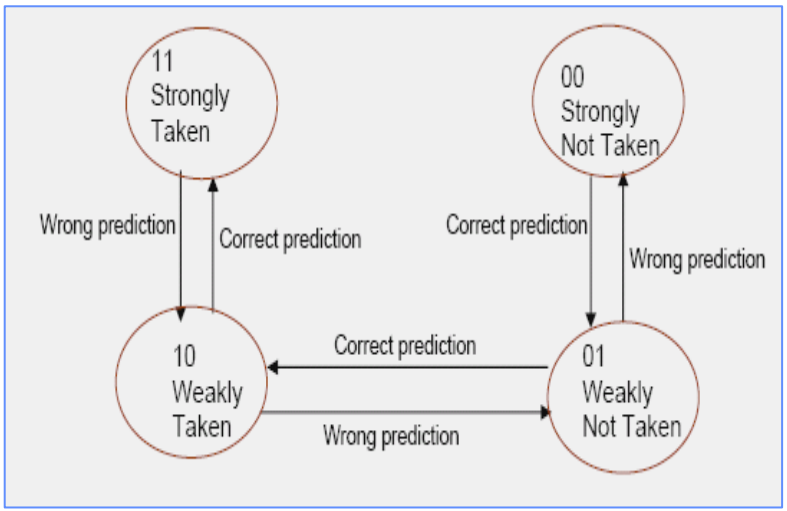


Dynamic predictors

•If one predicts conditional branch directions while a program is running, one is said to be doing dynamic branch prediction.
•Dynamic branch prediction uses history of branches stored in history register/s during the program execution, from the history, the branch outcome is predicted.

History types

Branch direction pattern history (taken or not taken sequence) PHT
Branch history (taken or not taken). BHR



State Diagram

Branch direction pattern History

Maintained by a branch Pattern history Table (PHT)
Based on whether branch recently taken or not taken.
Uses saturated counter.
Early predictors used 1 bit and if the address was 1 it was taken as branch taken otherwise as not taken. If prediction was found incorrect the bit was complemented and stored back. Later on the bits were increased to 2 bits, increasing bits beyond 2 did not give any extra advantage.

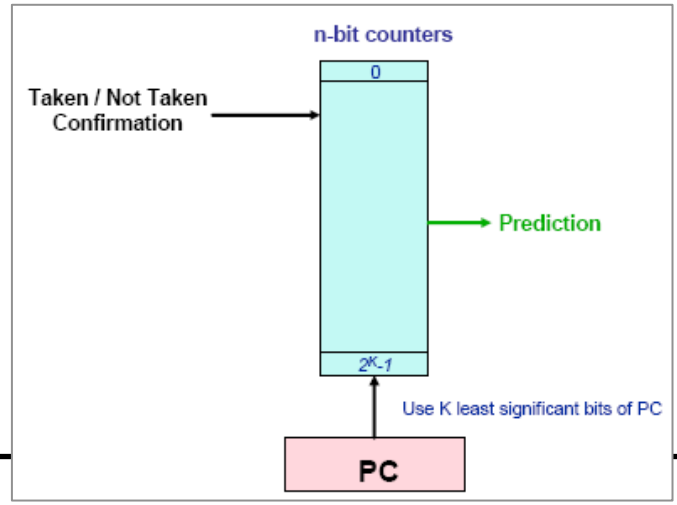
saturating counter max min no difference (instead of going to zero and overflowing).

Bimodal branch prediction

- PC is used to index a set of counters rather that branch prediction bits
- Each counter is an *n-bit* unsigned integer
- If branch is confirmed taken, counter is incremented by one. If it is confirmed not taken, counter is decremented by one
- Counters are *saturating* – if value is zero, decrement operations are ignored; if value is at the maximum, increment operations are ignored
  - Strongly not taken
  - Weakly not taken
  - Weakly taken
  - Strongly taken

A one-bit scheme (like in R8000), mispredicts both the first and last branch of a loop.

A two-bit scheme mispredicts just the last branch

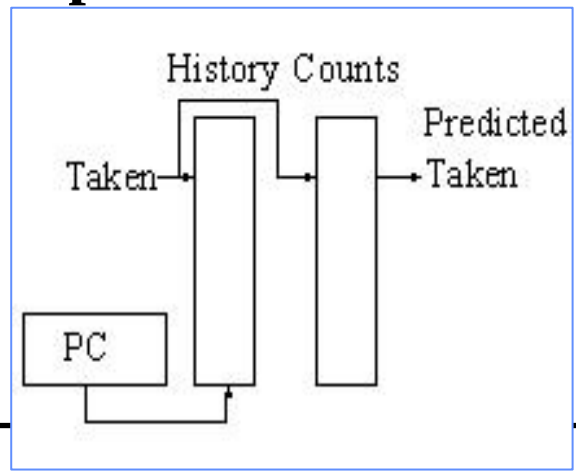


On the SPEC'89 benchmarks, very large local predictors saturate at 97.1% correct

Local branch prediction

- **Bimodal branch prediction mispredicts the exit of every loop.**
- **Local branch predictors keep two tables.**
  - The first table is the local branch history table. It records the taken/not-taken history of the n most recent executions of the branch.
  - The other table is the pattern history table(PHT). Like the bimodal predictor, this table contains bimodal counters
  - Its index is generated from the branch history in the first table. To predict a branch, the branch history is looked up, and that history is then used to look up a bimodal counter which makes a prediction.
- **Local prediction is slower than bimodal prediction**

On the SPEC'89 benchmarks, very large local predictors saturate at 97.1% correct.



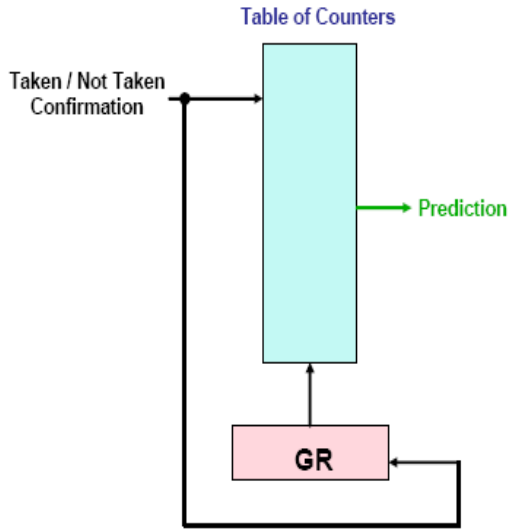
Global branch prediction

- **Global branch predictors make use of the fact that the behavior of many branches is strongly correlated with the history of other recently taken branches**
- **We can keep a single shift register (GR) updated with the recent history of every branch executed, and use this value to index into a table of bimodal counters**
- **Better than the bimodal scheme for large table sizes, and is never as good as local prediction**

Global predictor with Index Selection (gselect)

•Global history information is less efficient at identifying the current branch than simply using the branch address

• The counter table is indexed with a concatenation of global history and branch address bits. So there is a trade off between using more history bits or more address bits



On the SPEC'89 benchmarks, very large gshare predictors saturate at 96.6% correct, which is just a little worse than large local predictors

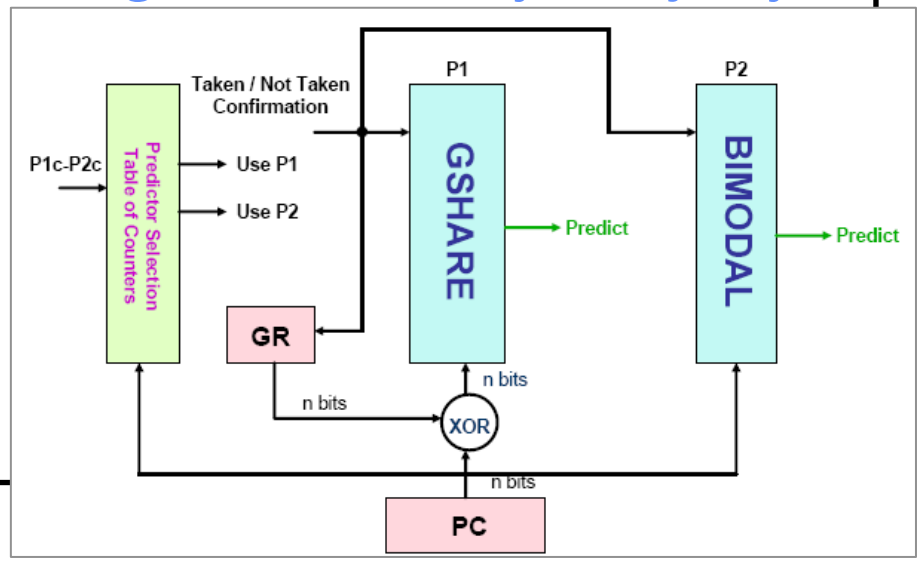
Global History with Index sharing (gshare)

Uses exclusive OR of the branch address with the global history to have more information than either component alone.

XOR of the branch address with the global history is more space efficient that just concatenation

Combined branch prediction

- **Different branch prediction schemes have different advantages. So, there is one method proposed by S. McFarling[Scott93] to combine two predictors P1, and P2 to take advantage of each.**
- **Combined branch prediction is about as accurate as local prediction, and almost as fast as global prediction.**
- **Some schemes**
  - 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.
  - Another way of combining branch predictors is to have e.g. 3 different branch predictors, and merge their results by a majority vote.



- The table of counters to determine whether P1 or P2 is to be used is updated as follows:

P1 correct?	P2 correct?	P1c-P2c
0	0	0
0	1	-1
1	0	1
1	1	0

- The P1c-P2c value is added to the counter addressed by PC

- Flexible and dynamic use of whichever is the more accurate predictor

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

Two-level adaptive Branch Predictors

- Two-Level Adaptive Branch prediction uses two levels of branch history information to make prediction.

First level: Branch History Registers (BHR)

- •Global history / Branch correlation: past executions of all branches
- •Self history / Private history: past executions of the same branch

Second level: Pattern History Table (PHT)

- •Use first level information to index a table **Possibly XOR with the branch address *[McFarling ´93]***
- •PHT: Usually saturating 2 bit counters
- •Also private, shared or global

Branch History register (BHR) and Pattern history table (PHT) can be one of 9 variations

Variation	Description
1	Global Adaptive Branch Prediction using one global pattern history table.
2	Global Adaptive Branch Prediction using per-set pattern history table.
3	Global Adaptive Branch Prediction using per-address pattern history table.
4	Per-address Adaptive Branch Prediction using one global pattern history table.
5	Per-address Adaptive Branch Prediction using per- set pattern history table
6	Per-address Adaptive Branch Prediction using per- address pattern history table
7	Per-Set Adaptive Branch Prediction using one global pattern history table.
8	Per-Set Adaptive Branch Prediction using per-set pattern history table.
9	Per-Set Adaptive Branch Prediction using per- address pattern history table.

Scheme like combining of local and global branch prediction.

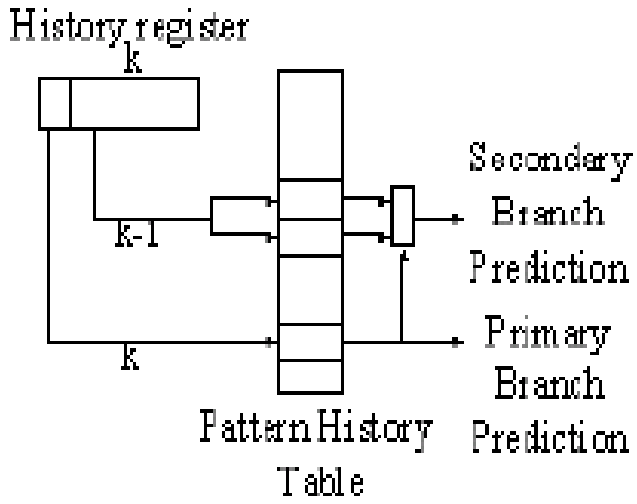
Multiple Branch Two-Level Adaptive Branch Predictor

•In Two-Level Adaptive Branch Prediction, History register and Pattern history table can be one of 9 variations.

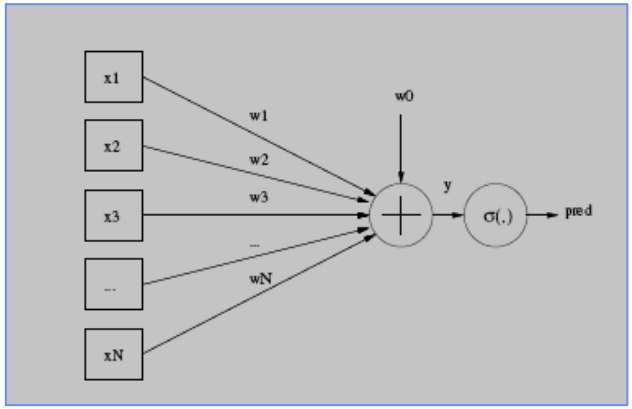
• k bit history register are used to index into the pattern history table to make a primary branch prediction.

•To predict the secondary branch prediction, the right most k-1 branch history bits are used to index into the pattern history table and the primary branch prediction is used to select one of the entries to make the secondary branch prediction.

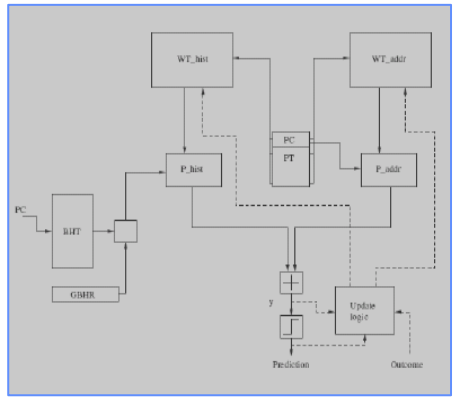
•Method improves performance in superscalar machine due to the multiple branch prediction



Neural Branch Predictors



Perceptron: simplest Neural predictor



The *Combined Perceptron Branch Predictor*

- **predictor consists of two concurrent perceptron-like neural networks**
  - *history-based*
  - *address-based*
- The *address-based Perceptron* has as inputs some bits of the PC. Its output is sensitive to the branch address and, if combined with the output of the *history-based Perceptron*, which is sensitive to branch history, it adds a contribution which significantly improves the prediction accuracy.

Approach provides lower misprediction rate than state-of the art conventional predictors.

Value Prediction

•Serialization constraints imposed by true data dependences have been regarded as an absolute dataflow limit--on the parallel execution of serial programs.
•Value prediction allows for exceeding that limit by allowing data dependent instructions to issue and execute in parallel without violating program semantics.

•This technique is built on the concept of value locality- recurrence of previously-seen value within a storage location
•Value prediction consists of predicting entire 32- and 64-bit register values based on previously-seen values.
•Predicts a complete value (e.g., a 32/64-bit integer), in contrast to a one-bit branch outcome resulting from branch prediction.
• In principle, value prediction can enable program execution in less time overcoming restrictions imposed by dataflow limit.

A 32 bit register would mean any one of 4 billion values --how could one possibly predict which of those is even somewhat likely to occur next?

•A probable method is to narrow the scope of the prediction mechanism by considering each static instruction individually, thus making the task somewhat easier thus accurately predicting a significant fraction of register values being written by machine instructions.
• These values are predictable in real-world programs because, value locality exists, primarily for the reason that partial evaluation is such an effective compile-time optimization.

Type of prediction that has quite recently emerged from the research community.

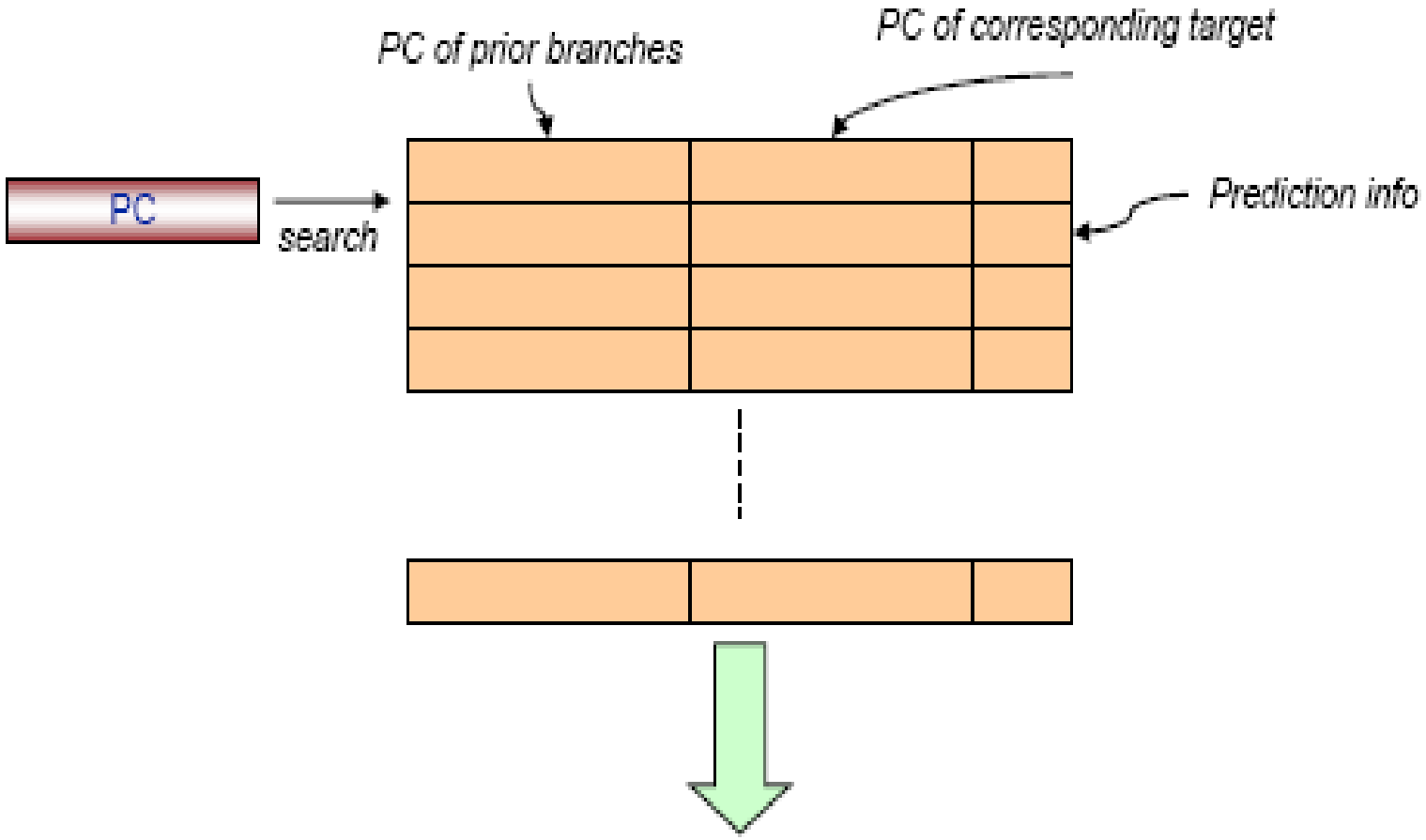
Misprediction Recovery

•What actions must be taken on a misprediction?
– Remove “predicted” instructions
– Start fetching from the correct branch target(s)
• What information is necessary to recover from misprediction?
– Address information for non-predicted branch target address
– Identification of those instructions that are “predicted”
– To be invalidated and prevented from completion
– Association between “predicted” instructions and specific branch
– When *that* branch is mispredicted then only *those* instructions
must be squashed

Branch Target Buffer

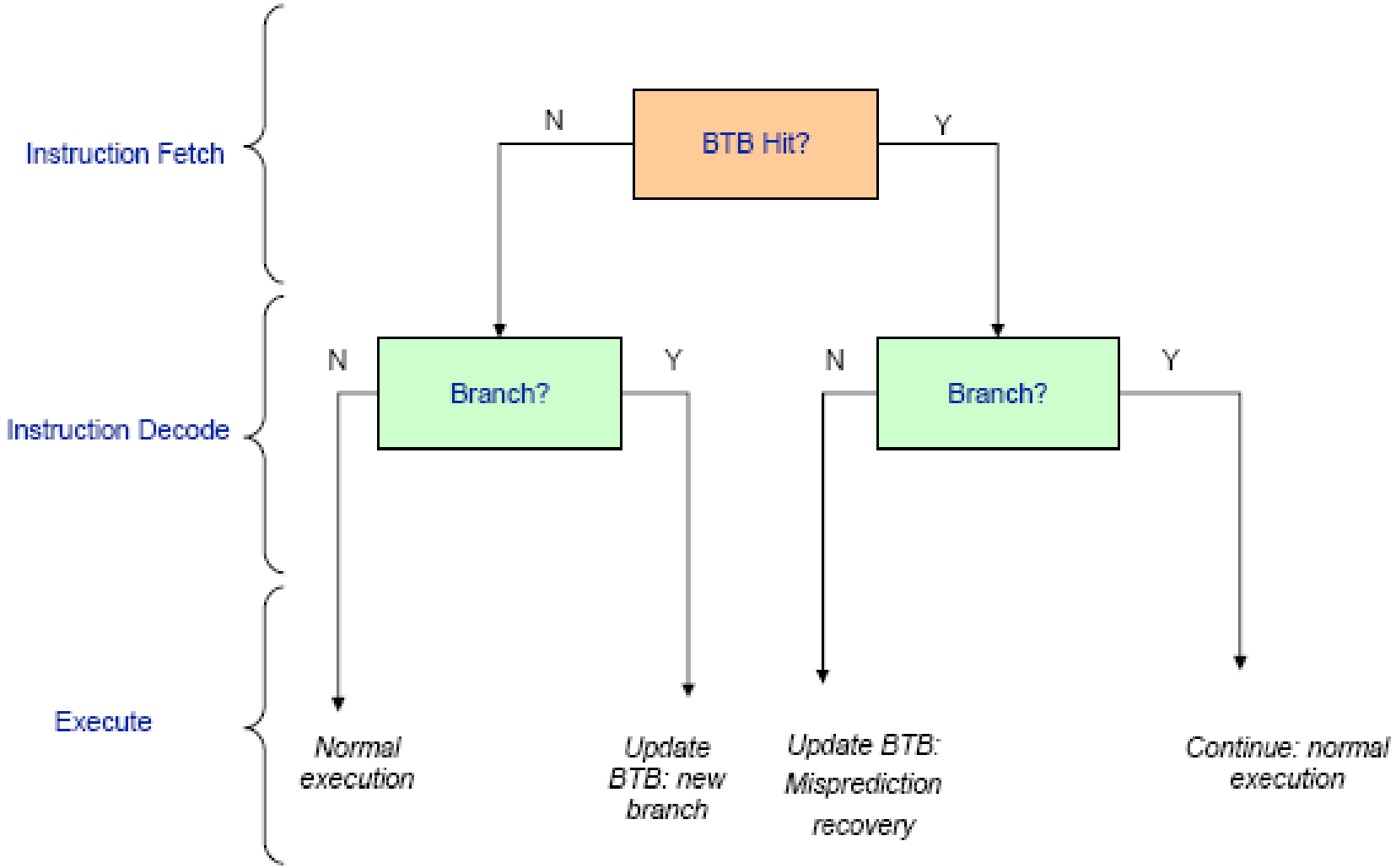
A cache that contains three pieces of information:

- The address of branch instructions
  - The BTB is managed like a cache and the addresses of branch instructions are kept for lookup purpose
- Branch target address
  - To avoid re-computation of branch target address where possible
- Prediction statistics
  - Different strategies are possible to maintain this portion of the BTB



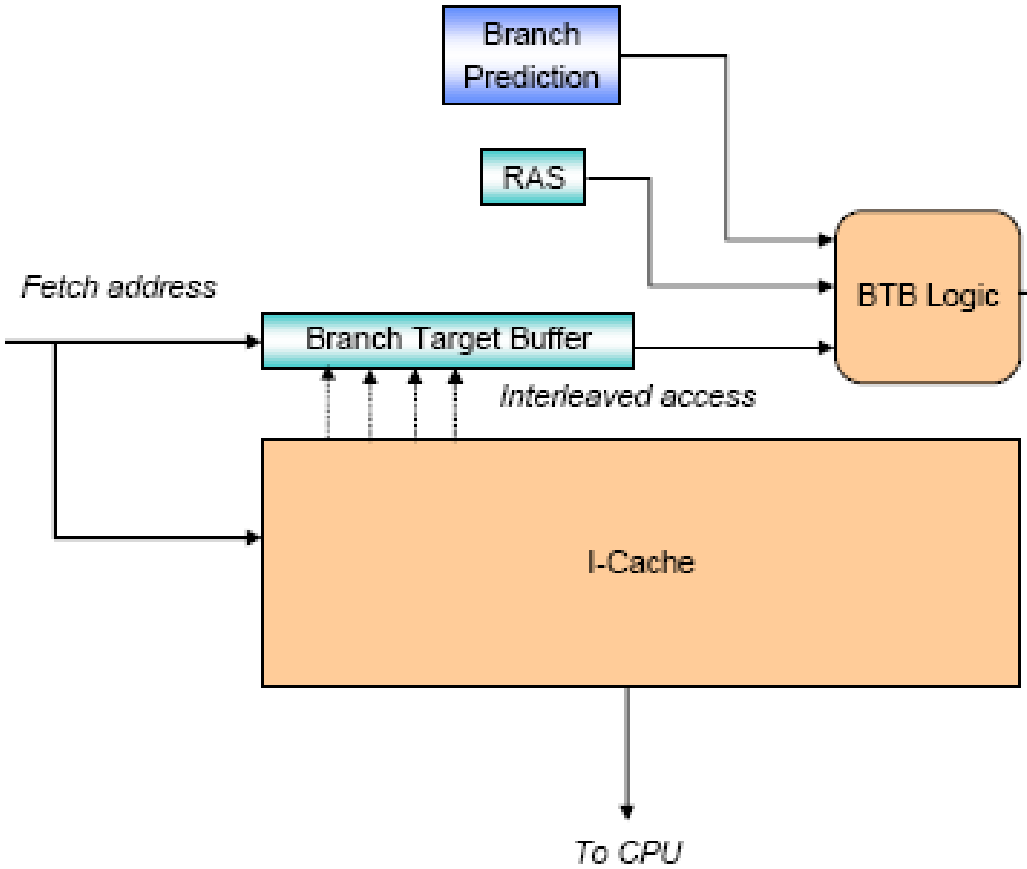
Hit produces the branch target address

Branch Target Buffer Operation



Any of the preceding history based techniques can be used for branch condition speculation

Integrated Solution



- Concurrently check whether each entry in an I-cache line is a branch
- Multi-branch prediction using a variant of global history

MODERN TRENDS

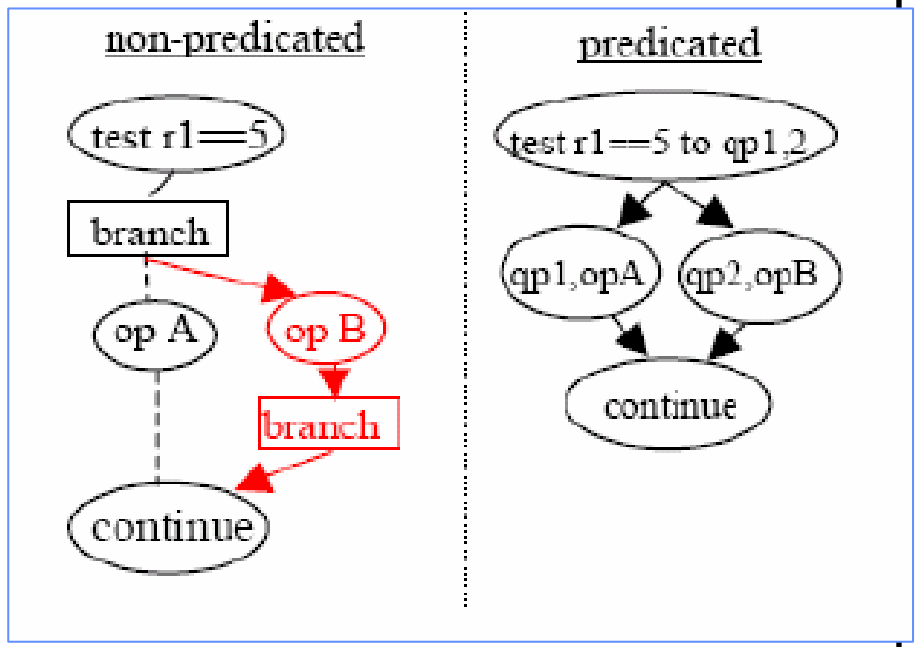
Branching Improvements in modern processors

Predication.

•**Eliminate an unpredictable branch**.

•**Both paths** of a branch can be executed in **parallel** and the results from the correct path enabled with a single predicate bit.[Itanium2

•This is a **compiler technique called if-conversion**. The EPIC architecture provides for **63 addressable predicate bits**



Early Branch Condition Testing.

•The EPIC architecture **separates** out the **test for a branch** condition from the **actual branch instruction.**

•This feature allows the branch test to be done early to **allow hardware to know the correct branch direction** before the branch instruction actually generates a misprediction resteer.

Branch hinting.

• The architecture provides a **special branch hint instruction** and allows **branch hints on all branch instructions**.

•These hints can indicate whether to **use the branch taken (or untaken) information from software inputs (statically)** or from **prediction hardware (dynamically).**

Control Speculation

•This **allows loads to be executed early by moving them ahead of branches** without the penalty of spurious faults.

•Using a **speculative load instruction**, Itanium processors **defer a fault until the results of the load are actually used** (determined to be nonspeculative).

• A faulting speculative load result is given a NaT (not-a-thing) deferral token

•**NaT add** to their **results**, thus allowing a speculative code sequence to continue through multiple instructions correctly and efficiently.

Data Speculation

•This allows **loads to be advanced safely before stores** that may change the value of the load.

•**Pointers to data types** makes it hard for the compiler to determine if load and store operations are **referencing the same memory location** or not. To be safe, a compiler for a conventional architecture must not reorder a load above the store.

• Using an **advanced load (ld.a)** instruction, modern processors can **move loads ahead of stores**, while **tracking the integrity of the advanced load data** until the load data is used.

•The **advanced load works like a normal load** except that it also maintains an **entry in an on-chip Advanced Load Address Table (ALAT)** with the load's register number, address and size.

Explicit cache line prefetching

•Way to **reduce effective memory latency**.

• In the Itanium instruction set defines a **cache line fetch hint** instruction which **anticipates the use of cache data** and brings in a cache line from memory with specific privileges (read only or write), and it can **direct that cache line to a specific cache level**.

•There are versions of this instruction that allow a **fault (for example, a page fault) to be taken immediately or delayed** until the cache line is actually used.

Cache hints

•They are provided in the EPIC architecture to **optimize the use of multilevel cache structures**.

•Hints are provided as to which **cache level** a cache line should be **promoted**, and **how long is it expected to be used**.

• If a cache line is classified as **non-temporal (*.nta*),** then the cache may choose to **replace that cache line before others**.

•These hints can help **prevent polluting smaller first or second level caches** with large data streams that are used briefly.

•In addition, they can be used to **reduce the effective latency of a memory access** by **bringing data into the caches before the program needs them.**

Intel Processors (Intel Wide Dynamic Execution)

•Dynamic execution is a **combination of techniques** (data flow analysis, speculative execution, out of order execution, and super scalar)

•For Intel **NetBurst** microarchitecture, Intel introduced its **Advanced Dynamic Execution engine**, **a very deep, out-of-order speculative execution engine** designed to keep the processor's execution units executing instructions.

• It also featured an **enhanced branch-prediction algorithm** to reduce the number of branch mispredictions.

•Every **execution core is wider**, allowing each core to fetch, dispatch, execute, and return up to **four full instructions simultaneously**,

•**Macrofusion** for **reducing execution**. In previous generation processors, each incoming instruction was individually decoded and executed, Macrofusion enables **common instruction pairs (such as a compare followed by a conditional jump**) to be combined into a single internal instruction (micro-op) during decoding.

• Two program instructions can then be executed as one micro-op, **reducing the overall amount of work** the processor has to do

Intel Itanium (IA-64)

- Intel Itanium (IA-64) architecture uses **Explicitly Parallel Instruction Computing (EPIC).**
- EPIC designs **move the complexity** of Out-of-Order RISC from **hardware to software**.
- EPIC has **VLIW Instruction** set and some dynamic checks. The **compiler is still responsible for scheduling instructions**, but there is also **speculative executions that can be controlled by the compiler.**
- Processors are **built to handle hazards by hardware or software.** (Unlike in **RISC processors**, a processor having hazard detection and resolution logic (stall logic) behaves like a conventional processor and **compiler has no role for the problems created by hazards**.)

AMD Opetron

- According to AMD, the **pipeline's front end instruction fetch and decode logic** has been optimized to pack multiple decoded, micro-op instructions together to **execute them in parallel**.
- The Opteron processor has a **12-stage integer pipeline**, much **shorter** than the pipeline for Xeon processors. The **shorter pipeline requires a slower frequency**.
- **Shorter pipeline reduces the risk of delays** due to branch mispredicts and cache misses.
- The shorter pipeline also requires **less extensive branch prediction algorithms and target buffers.**
- To make operations more parallel, the Opteron processor also has **more execution units and decode units than Xeon processors**.
- The Opteron processor includes three ALUs, three AGUs, and three floating-point execution units . Although it has more individual execution units than Xeon processors, the maximum effective **throughput of the Opteron execution units is the same as in a Xeon processor**.
- A shorter, more efficient micro-architecture such as that implemented in the AMD Opteron processor would **not benefit from Hyper-Threading** as much as the hyper-pipelined Xeon micro-architecture, nor does it require this **level of complexity** to provide high levels of multithreaded performance.

IBM Cell Processor

- The Cell chip can have a number of different configurations, the basic configuration is composed of one **"Power Processor Element" ("PPE") (sometimes called "Processing Element", or "PE"),** and **multiple "Synergistic Processing Elements" ("SPE").** The PPE and SPEs are linked together by an **internal high speed bus dubbed "Element Interconnect Bus" ("EIB").**
- The **PPE is not intended to perform all primary processing** for the system, but rather to act as a **controller for the other eight SPEs**, which handle most of the computational workload.
- The PPE will work with conventional operating systems due to its similarity to other 64-bit PowerPC processors, while the **SPEs are designed for vectorized floating point code execution**

CONCLUSION

- Most modern processors have started using a mix of techniques such as those used by Intel Wide dynamic Execution
- Therefore we conclude that to achieve high performance no single technique is good enough. Manufacturers of high performance processors such as Intel, Sun, AMD, IBM etc use a mix match of these techniques under some coded name from which it is not quite evident what all they are using and how they are using. This is done to maintain a high levell of secrecy in the highly competitive market of today.

Reference

- [Kunle, Basem, Lance, Ken, Kunyung] **The Case for a Single-Chip Multiprocessor** Kunle Olukotun, Basem Nayfeh, Lance Hammond, Ken Wilson, and Kunyung Chang Pages 1-2 Computer Systems Laboratory Stanford University

- **2.** [Mikko John] **Exceeding the Dataflow Limit via Value Prediction**
  Mikko H. Lipasti and John Paul Shen, Department of Electrical and Computer Engineering, Carnegie Mellon University, Pittsburgh PA, 152 13

- **3.** [Fisher70] **Predicting Conditional Branch Directions From Previous Runs of a Program** Joseph A. Fisher and Stefan M. Freudenberger, Pages 85-87

- **4.** [Matt-04] **The Combined Perceptron Branch Predictor** Page 1-2
  Matteo Monchiero Gianluca Palermo Politecnico di Milano . Dipartimento di Elettronica e Informazione Via Ponzio, 34/5, 20133 Milan, Italy fmonchier, Report 04 palermog@elet.polimi.it

- **5.** [Steven, Luke, Swift, Susan, Henry] **An Evaluation of Speculative Instruction Execution on Simultaneous Multithreaded Processors** Steven Swanson, Luke K. Mcdowell, Michael M. Swift, Susan J. Eggers And Henry M. Levy University of Washington Pages 1-3

- **6.** [Scott-93] **Combining Branch Predictor.** Scott McFarling. Technical Report. Technical Note TN-36, Digital Equipment Corporation, Western Research Lab. June 93

- **7.** [Yeh Patt-92] [YP-92] **Alternative implementation of two-level adaptive branch prediction.** In Proceedings of 19th International Symposium. T.Y. Yeh and Y.N. Patt. On Computer Architecture, pages 124-134, May 1992

- **8.** [Yeh Patt-93] [YP-93] **A comparison of dynamic branch predictors that use two levels of branch history.** T.Y. Yeh and Y.N. Patt. In Proceedings of 20th annual International Symposium. On Computer Architecture, pages 257-266, IEEE and ACM, May 1993

- **9.** [Hideki, Chikako, Tetsuya, Masao-05] **Unconstrained Speculative Execution with Predicated State Buffering** Pages 126-127 Hideki Ando, Chikako Nakanishi, Tetsuya Hara, Masao Nakaya , System LSI Laboratory Mitsubishi Electric Corporation 4-1 Mizuhara, Itami, Hyogo, 664 Japan

- **10.** [**Ofri** -06] **Inside Intel's Core Micro architecture.** Setting New Standards for Energy-Efficient Performance **Ofri Wechsler** Intel Fellow, Mobility Croup Director, Mobility Microprocessor Architecture Intel Corporation White Paper 06

- **11.** [Hewlet Packard -2005] **Characterizing x86 processors for industry-standard servers:** AMD Opteron and Intel Xeon Whitepaper , technology brief, 2nd Edition Hewlet Packard -2005

- **12.** [Itanium2-02] **Inside the Intel_ Itanium_ 2 Processor** Pages 1-12
  IItanium Processor for balanced performance over a wide range of applications
  Hewlet Packard Technical White Paper

- **13.** [James E. Thornton-64] **Considerations in computer design leading upto the CONTROL DATA@6600** James E. Thornton Control Data Chippewa laboratory

- **14.** Cvetanovic and Kessler] **Performance analysis of the Alpha 21264-based Compaq ES40 system**. Notes on *Proceedings of the 27th ACM International Symposium on Computer Architecture*, CVETANOVIC, Z. AND KESSLER, R. E. 2000. Vancouver, Canada.

Web Sources

1http://gamma.cs.unc.edu/QVDR/temp_homepage/project/comp206/survey.html

2 http://en.wikipedia.org/wiki/Pentium_4

3. http://en.wikipedia.org/wiki/Instruction_pipeline

4. http://en.wikipedia.org/wiki/Superscalar

5. http://developer.intel.com/design/itanium

6. http://www.hp.com/go/itaniumdeveloper

7. http://developer.intel.com/

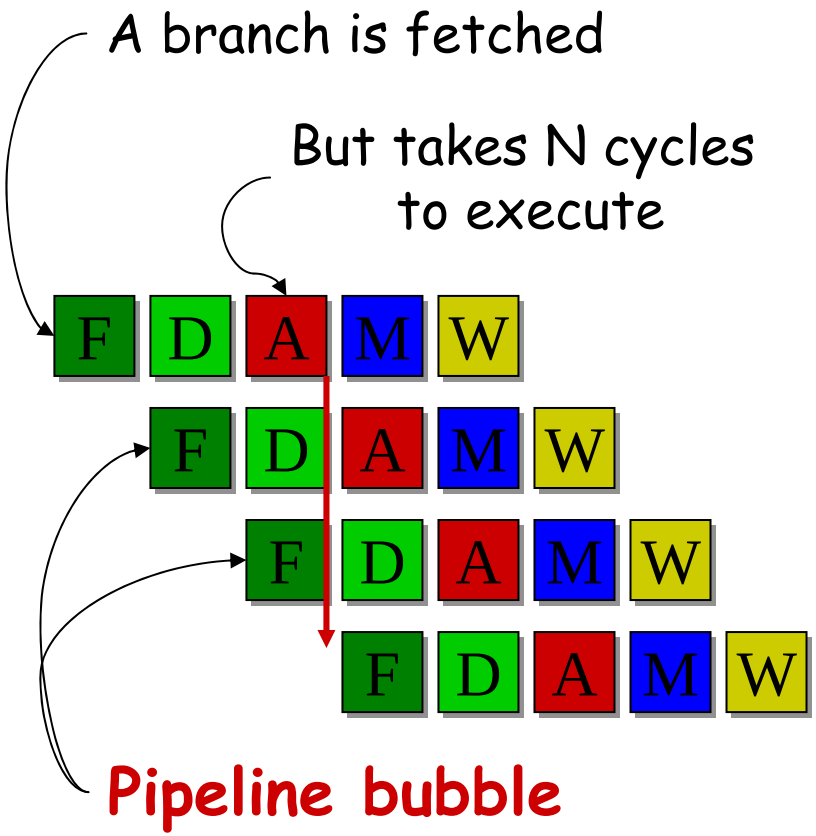
8. http://en.wikipedia.org/wiki/CDC_6600

9. http://www-hydra.stanford.edu

Thank You

???????

Pipeline bubble



- **Pipelined execution**
  - A new intruction enters the pipeline every cycle...
  - …but still takes several cycles to execute
- **Control flow changes**
  - Two possible paths after a branch is fetched
  - Introduces pipeline "bubbles"
    - **Branch delay slots**
  - Prediction offers a chance to avoid this bubbles

Problem increases with superscalar execution

Eliminating hazards

- **Bubbling the pipeline (a technique also known as a pipeline break or pipeline stall) is a method for preventing data, structural, and branch hazards from occurring. As instructions are fetched, control logic determines whether a hazard could/will occur. If this is true, then the control logic inserts NOPs into the pipeline. Thus, before the next instruction (which will cause the hazard) is executed, the previous one will have had sufficient time to complete and prevent the hazard. If the number of NOPs is equal to the number of stages in the pipeline, the processor has been cleared of all instructions and can proceed free from hazards. This is called flushing the pipeline. All forms of stalling introduce a delay before the processor can resume execution.**

- Since exceptions occur infrequently, one may think that the adverse effect of unsafe code motions is negligibly small. This is not the case in speculative execution. Suppose that a load instruction dereferences a pointer to a next element in a loop program which traverses a linked list. If the load instruction is speculatively executed, it attempts to dereference a NULL pointer in the last iteration, and thus an exception occurs. This type of speculative code motion is quite effective for performance improvement because dereferences are often in a critical path. As a result, aggressive unsafe code motions considerably increase the frequency of exceptions.

- Just re-execution of the excepting instruction is not sufficient. The speculative instructions which are directly or indirectly dependent upon the excepting instruction must be re-executed since they used polluted operands. This re-execution of the speculative instructions is termed recovery from a speculative exception.

- That is, if the compiler moves an unsafe instruction, operand registers of succeeding instructions which maybe re-executed must be live until the commit point of that unsafe instruction. This increases the number of live registers, and thus puts pressure on the compiler register allocation

- **Static superscalar (In order Issue, In order Execution)**

  Multiple instructions are fetched in a single cycle. The dependencies are checked in the second part of the issue phase, just like the single-issue case, the instructions are scheduled in order, so the compiler has to do a good job of scheduling, so that issued packets are hazard free.

- **Dynamic superscalar (ln order Issue, Out of Order execution)**

  Multiple instructions are fetched in a single cycle, and they enter dynamic scheduling algorithm with reservation stations, algorithm passes them to the FUs every clock where FUs initiates operations when operands arrive in the allocated reservation, stations.

- **Speculative Superscalar**

  In superscalar processors speculation hides branch latencies and thereby boosts Architectures Over the Yearsperformance by executing the likely branch path without stalling the pipeline. Branch predictors, which provide accuracies up to 96% (excluding OS code), are the key to effective speculation. The primary disadvantage of speculation is that some processor resources are invariably allocated to useless, wrong-path instructions that must be flushed from the pipeline. However, since resources on superscalar's are often underutilized because of low single-thread instruction level parallelism (ILP) [Cvetanovic and Kessler 2000], the benefit of speculation far outweighs this disadvantage and the decision to speculate as aggressively as possible is an easy one. Dynamic scheduling in superscalar is done by speculative execution using branch prediction hardware. They also employ speculative loads to hide the memory latencies.

- **Speculative Instruction Execution on SMT Processors**

  In contrast to superscalar's, simultaneous multithreading (SMT) processors [Steven, Luke, Swift, Susan, Henry] operate with high processor utilization, because they issue and execute instructions from multiple threads in each cycle, with all threads dynamically sharing hardware resources. If some threads have low ILP, utilization is improved by executing instructions from additional threads; if only one or a few threads are executing, then all critical hardware resources are available to them. Consequently, instruction throughput on a fully loaded SMT processor is two to four times higher than on a superscalar with comparable hardware on a variety of integer, scientific, database, and web service workloads

- **VLIW systems.**

  Compilers **arranges** instructions into data-independent groups, looks at a large group of instructions in order to use the machine's resources well. The group of instructions the compiler considers at once while scheduling is often called the candidate set. This is analogous to the instruction window of superscalar systems. The VLIW processor essentially has compiler decided multiple independent instructions packed in a single word which can be issued to the FUs in parallel. VLIW architecture require compilers that explicitly specify parallelism. The compiler is completely responsible for making sure that instructions within long instructions are independent, and that instruction dependencies will all have been satisfied

Three Stages of Tomasulo Algorithm

1 **Issue:** Get instruction from pending Instruction Queue (IQ).

Always done in program order

- Instruction issued to a free reservation station(RS) (no structural hazard).
- Selected RS is marked busy.
- Control sends available instruction operands values (from ISA registers) to assigned RS.
- Operands not available yet are renamed to RSs that will produce the operand (register renaming). (Dynamic construction of data dependency graph)

2 **Execution (EX):** Operate on operands.

- When both operands are ready then start executing on assigned FU.
- If all operands are not ready, watch Common Data Bus (CDB) for needed result (forwarding done via CDB). (i.e. wait on any remaining operands, no RAW)

Data dependencies observed

3 **Write result (WB):** Finish execution.

- Write result on Common Data Bus (CDB) to all awaiting units (RSs)
- Mark reservation station as available.

i.e broadcast result on CDB

- Normal data bus: data + destination ("go to" bus).

Common Data Bus (CDB): data + source ("come from" bus):

- 64 bits for data + 4 bits for Functional Unit source address.
- Write data to waiting RS if source matches expected RS (that produces result).
- Does the result forwarding via broadcast to waiting RSs.

Including destination register

Can be done out of program order

(In Chapter 3.2)

BP and BTP

- **Branch prediction attempts to guess whether a conditional branch will be taken or not.**
- **Branch target prediction attempts to guess the target of the branch or unconditional jump before it is computed from parsing the instruction itself.**

ILP for high-performance Microprocessors

- **Modern high-performance microprocessors rely on sophisticated and accurate branch predictors to efficiently exploit Instruction Level Parallelism (ILP).**
- **Complex front-ends, capable of filling large instruction windows, are required to provide high frequency of operations and aggressive parallelism.**
- **Branch prediction is a key element of such a system, providing correct fetch beyond branch boundary and, therefore, large throughput instruction deliver.**

basic block

- **Defined to be instructions between Branches, whether they are taken or not taken.**
  - a block simply is a group of sequential instructions up to a predefined limit, or up to the end of a line.
  - Instructions after the first control transfer in a block are not used.
  - A *line* of instructions refers to the group of instructions physically accessed in the instruction cache.

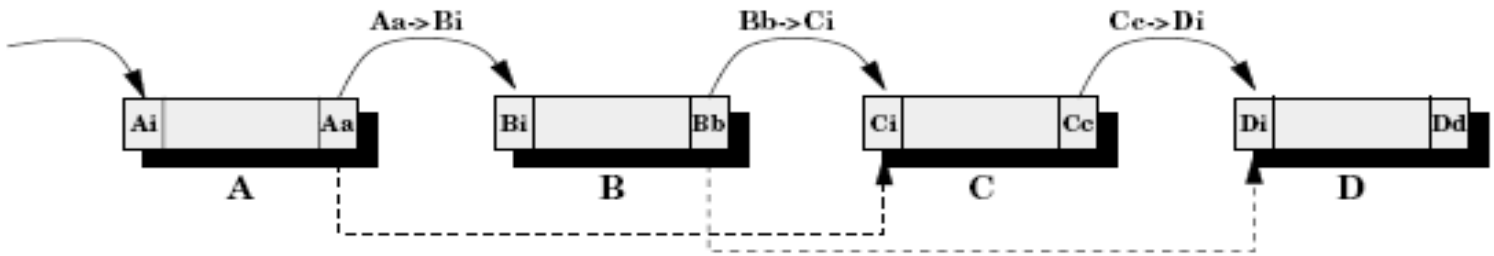
IF Mechanism

- **Fetch mechanism consists mainly of three parts :-**
  - An instruction cache from where the instructions are fetched
  - An instruction-dispatch buffer where the instructions are maintained waiting to be dispatched
  - Some branch prediction structures predicting the outcome and the target address of any fetched branch

Dispatch buffer

- **The dispatch buffer decouples the instruction fetching from the dispatch process, sustaining a better throughput in the presence of cycles in which only a small number of instructions can be fetched**

Two-Block Ahead Branch Predictor



- **Two-block ahead branch predictor uses information as-sociated with the current instruction block to predict the address of the instruction block that is two blocks ahead.**
  - Ai, Bi. Ci, and Di are the basic block starting addresses
  - Aa. Bb, Cc. Dd are the branch addresses
- **While the instruction blocks A and B are fetched, the two-block ahead branch predictor uses Aa instead of Bb to predict Ci, and Bb to predict Di**

Types of predictors

- **CPUs evaluated branches in the decode stage and had a single cycle instruction fetch.**
- **result, the branch target recurrence was two cycles long, and the machine would always fetch the instruction immediately after any taken branch.**
- **Both defined branch delay slots in order to utilize these fetched instructions.**

Next line prediction

- **superscalar processors (MIPS R8000, DEC Alpha EV6 and EV8)**
  - fetched with each line of instructions a pointer to the next line.
  - handles both branch target prediction as well as branch direction prediction.

- **When a next line predictor points to aligned groups of 2, 4 or 8 instructions, the branch target will usually not be the first instruction fetched, and so the initial instructions fetched are wasted. Assuming for simplicity a uniform distribution of branch targets, 0.5, 1.5, and 3.5 instructions fetched are discarded, respectively.**

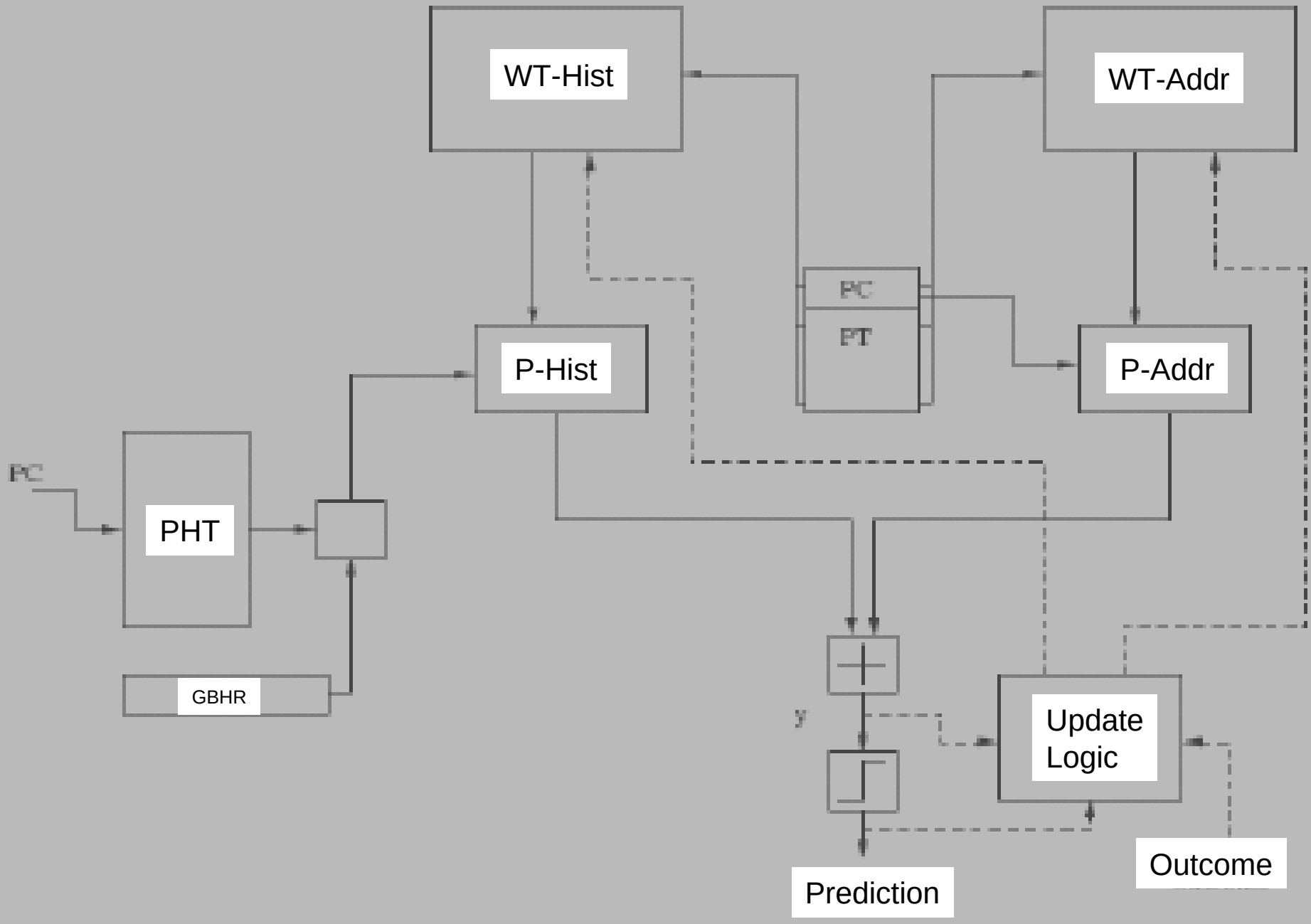
- **Since the branch itself will generally not be the last instruction in an aligned group, instructions after the taken branch (or its delay slot) will be discarded. Once again assuming a uniform distribution of branch instruction placements, 0.5, 1.5, and 3.5 instructions fetched are discarded.**

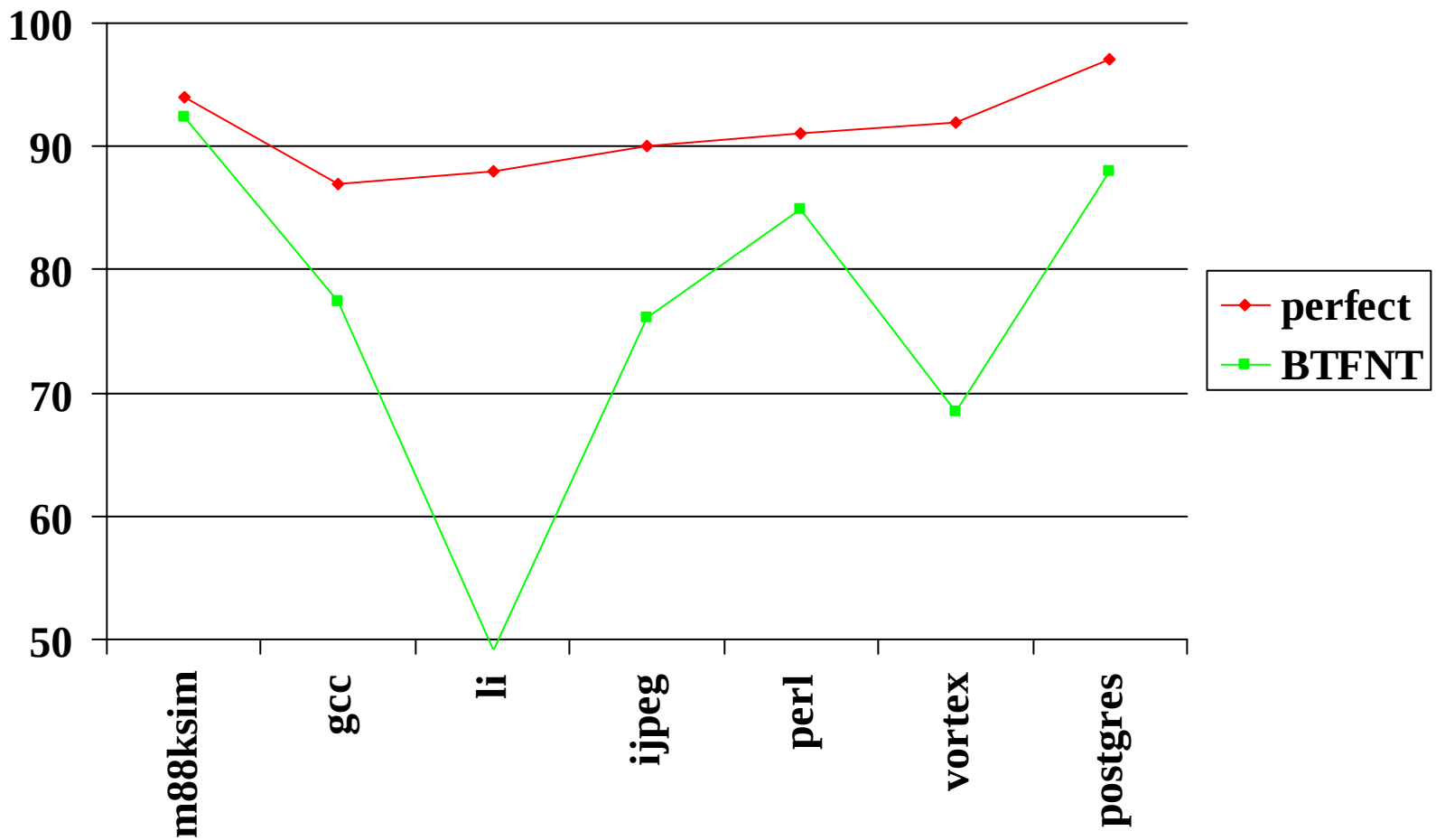
- **The discarded instructions at the branch and destination lines add up to nearly a complete fetch cycle, even for a single-cycle next-line predictor.**

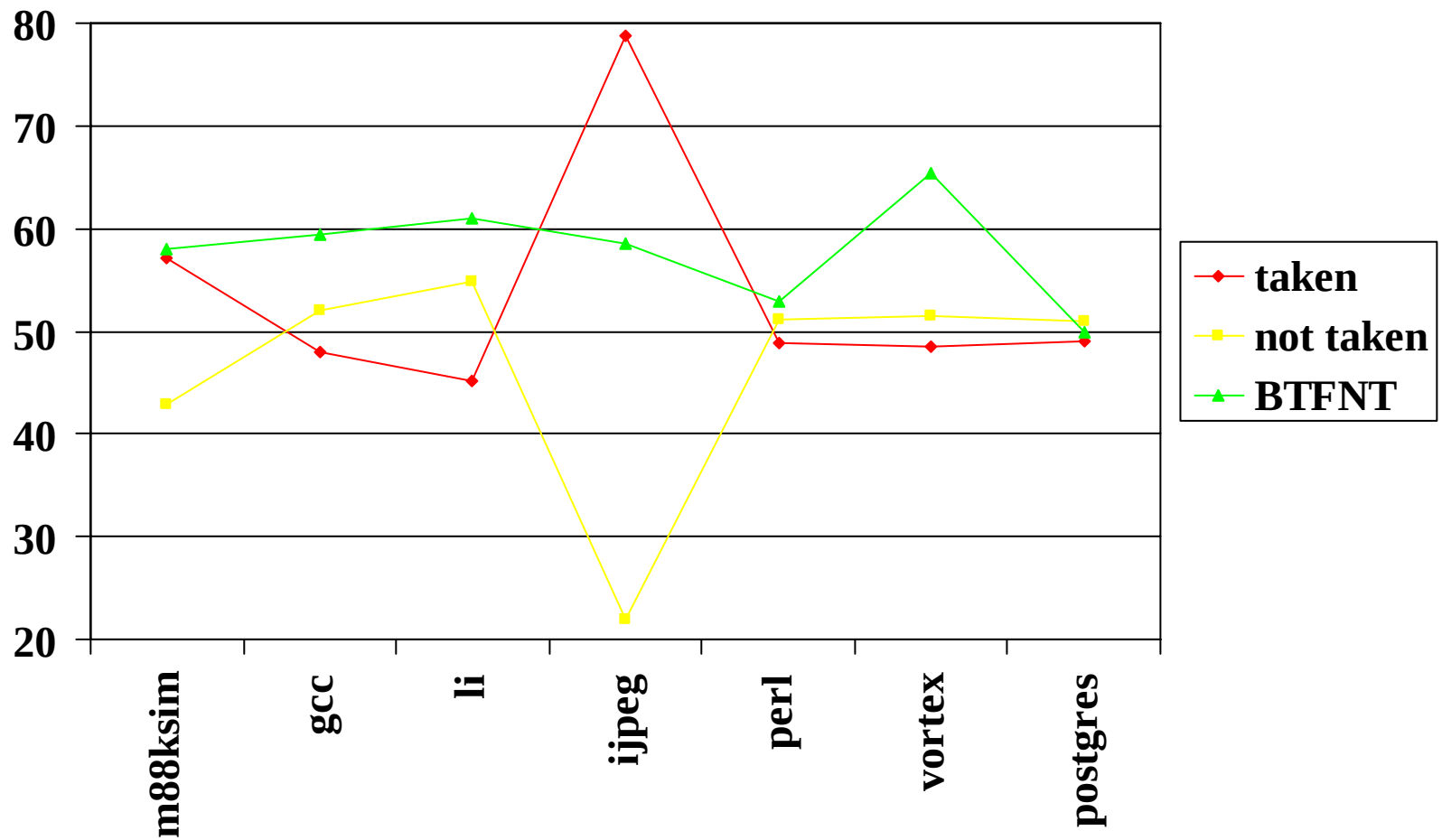
Overriding branch prediction

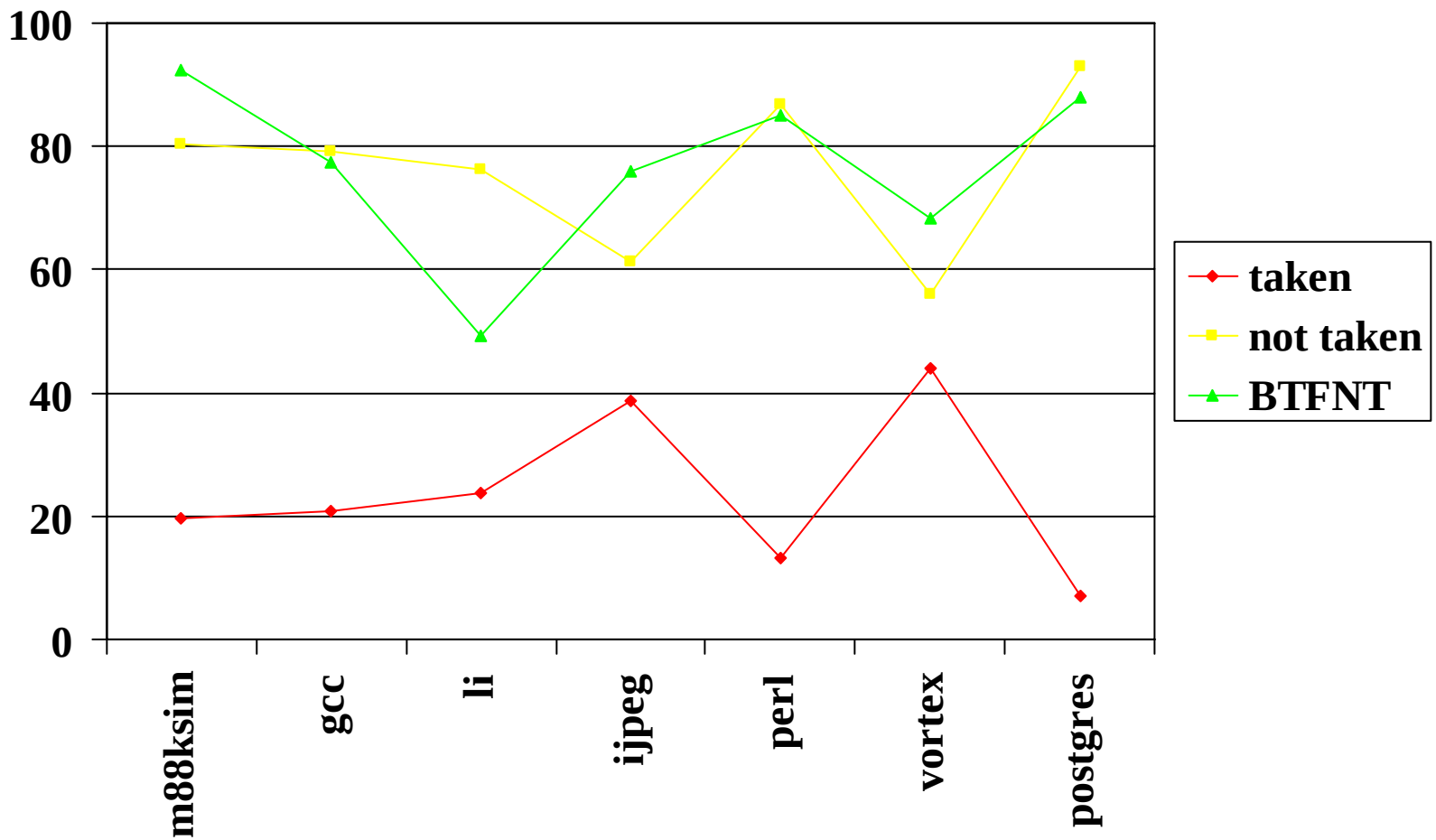
- **The EV6 and EV8 cores used a fast single-cycle next line predictor to handle the branch target recurrence and provide a simple and fast branch prediction. Because the next line predictor is so inaccurate, and the branch resolution recurrence takes so long, both cores have two-cycle secondary branch predictors which can override the prediction of the next line predictor at the cost of a single lost fetch cycle.**

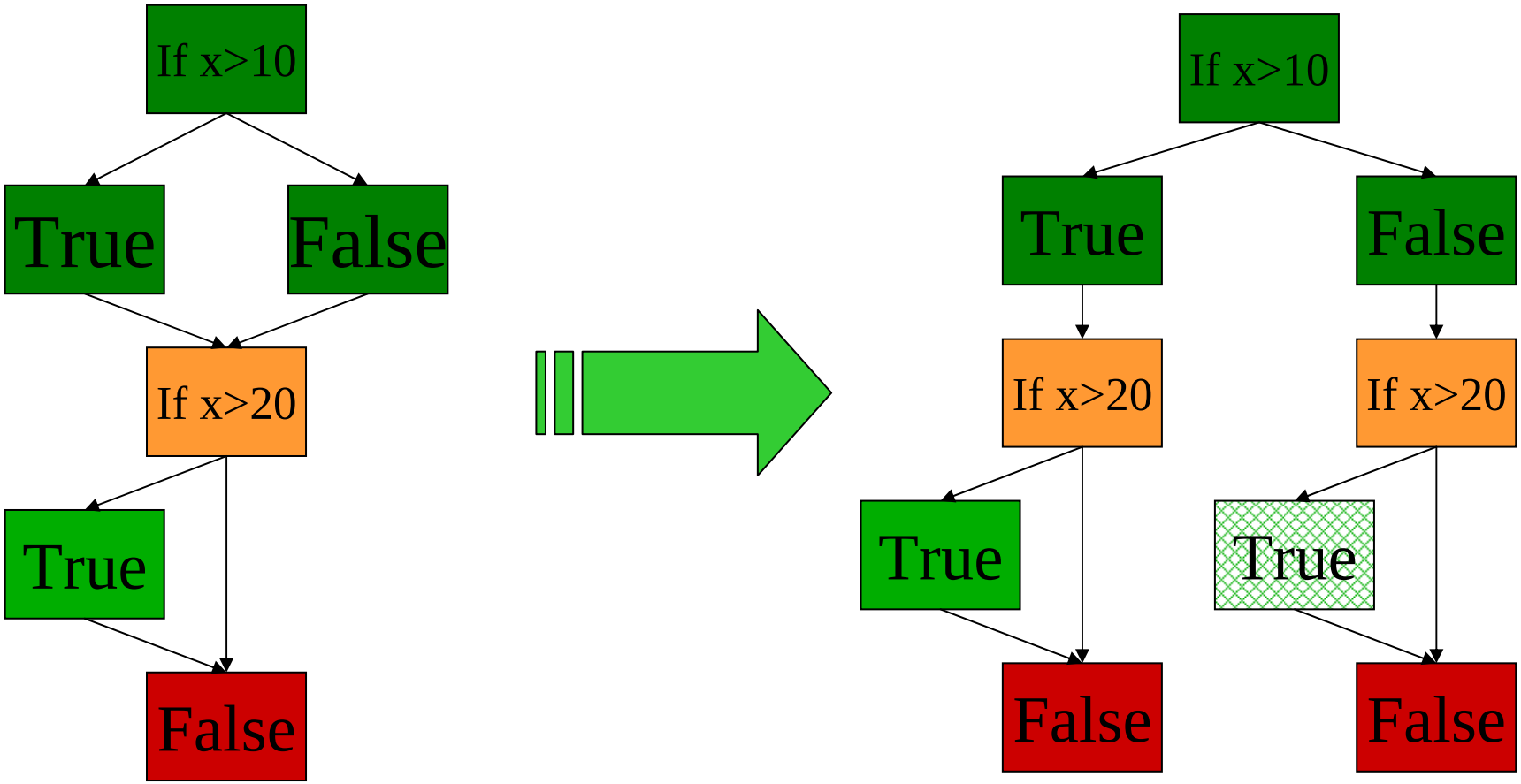
- **Since a fetch of 4 instructions or more may contain more than one branch, an overriding predictor will sometimes have to predict a next PC which is neither the fall-through nor the earlier predicted next line. The overriding predictor usually extracts the target address from the instruction bytes themselves since they are available by the time the predictor must generate a predicted next fetch address.**


WT-Hist
WT-Addr
PC
PT
P-Hist
P-Addr
PC
PHT
GBHR
Update
Logic
Prediction
Outcome

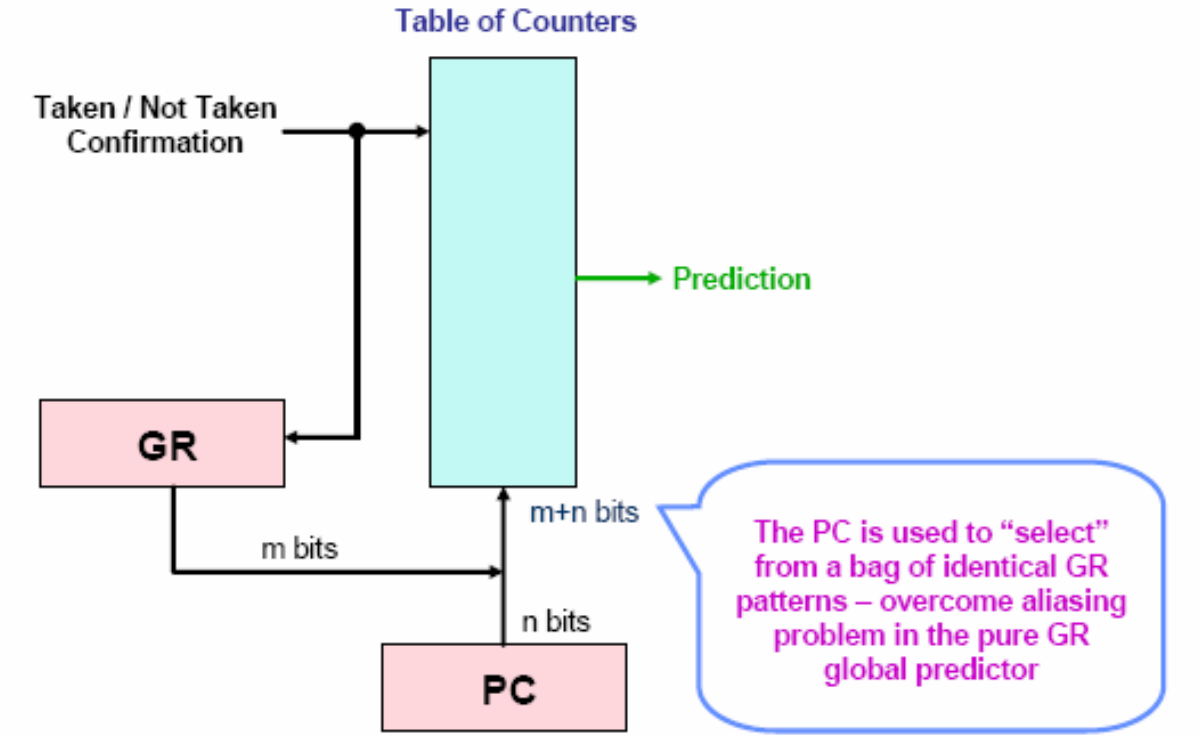

100
90
80
70
60
50
m88ksim
gcc
li
ijpeg
perl
vortex
postgres
perfect
BTFNT

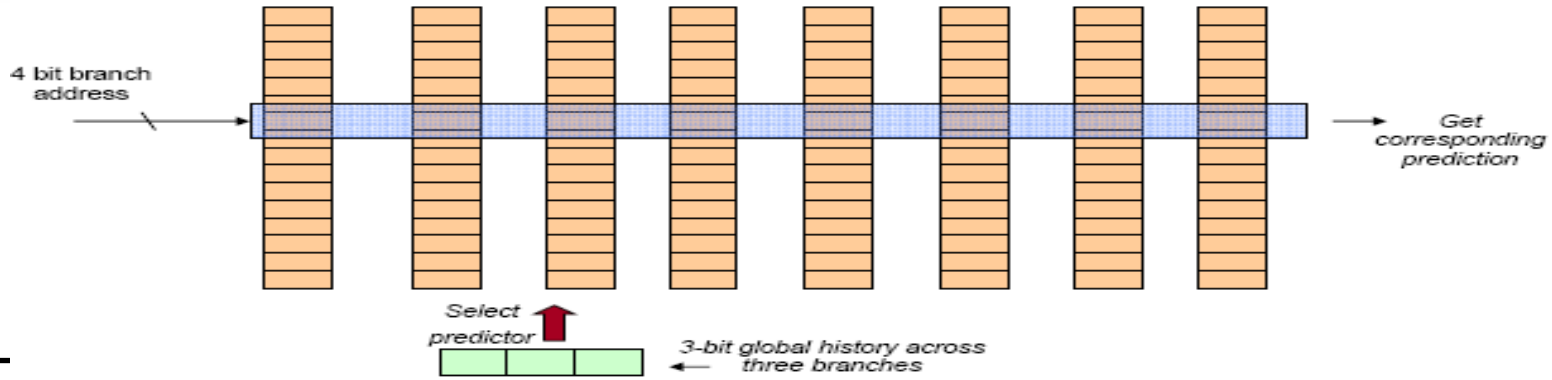

80
70
60
50
40
30
20
m88ksim
gcc
li
ijpeg
perl
vortex
postgres
taken
not taken
BTFNT


100
80
60
40
20
0
m88ksim
gcc
li
ijpeg
perl
vortex
postgres
taken
not taken
BTFNT

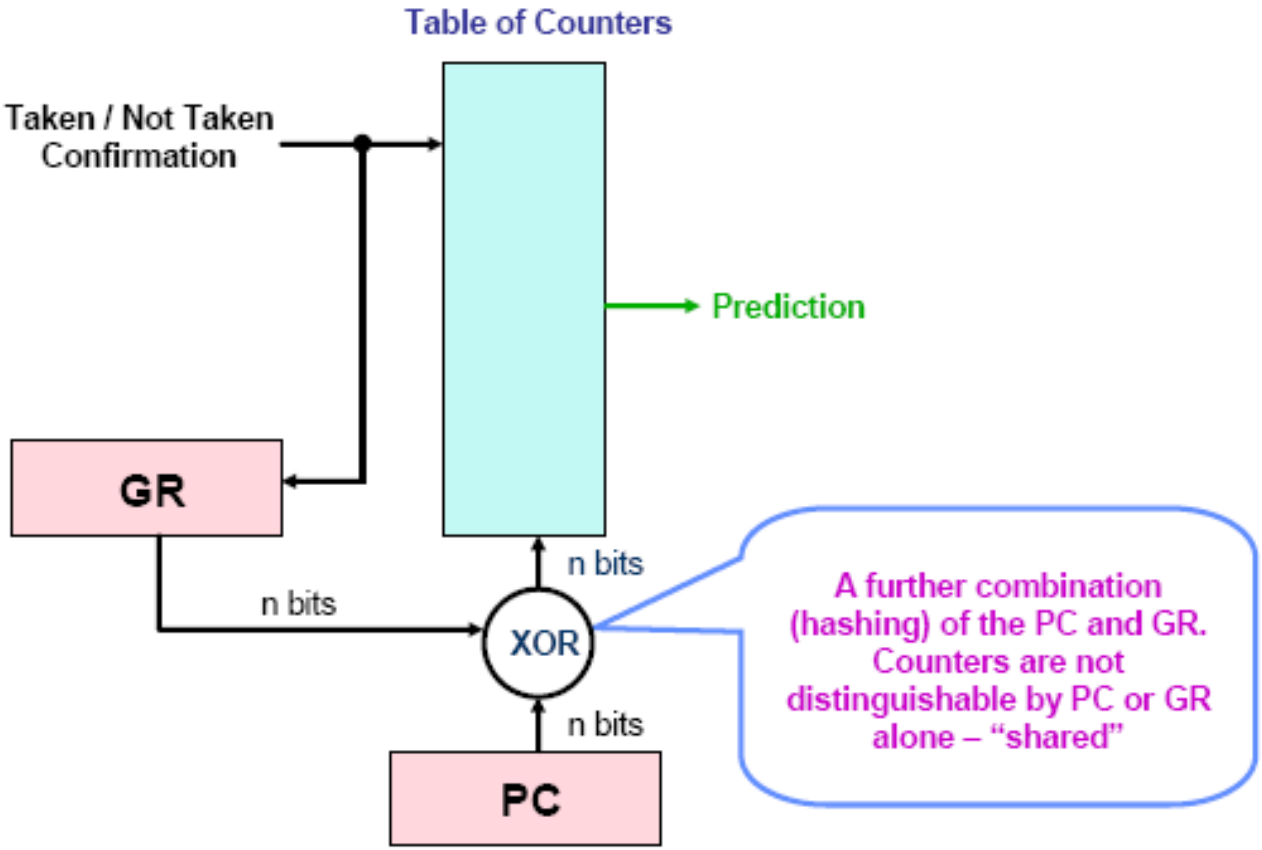

If x>10
True
False
If x>20
True
False
If x>10
True
False
If x>20
If x>20
True
True
False
False

The gselect Predictor



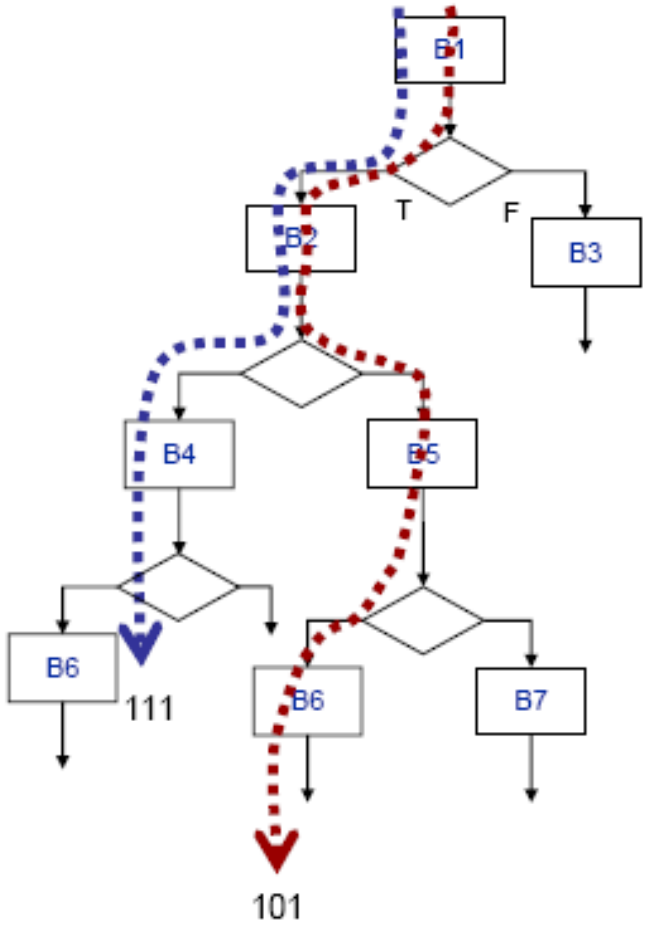


The gshare Predictor

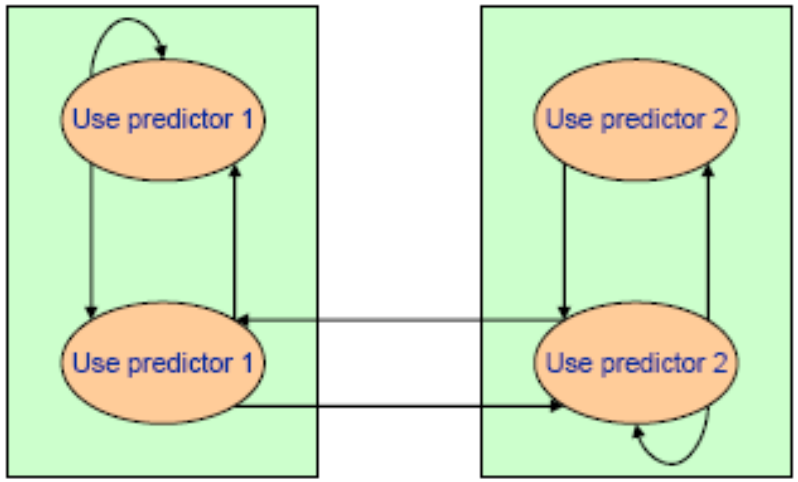


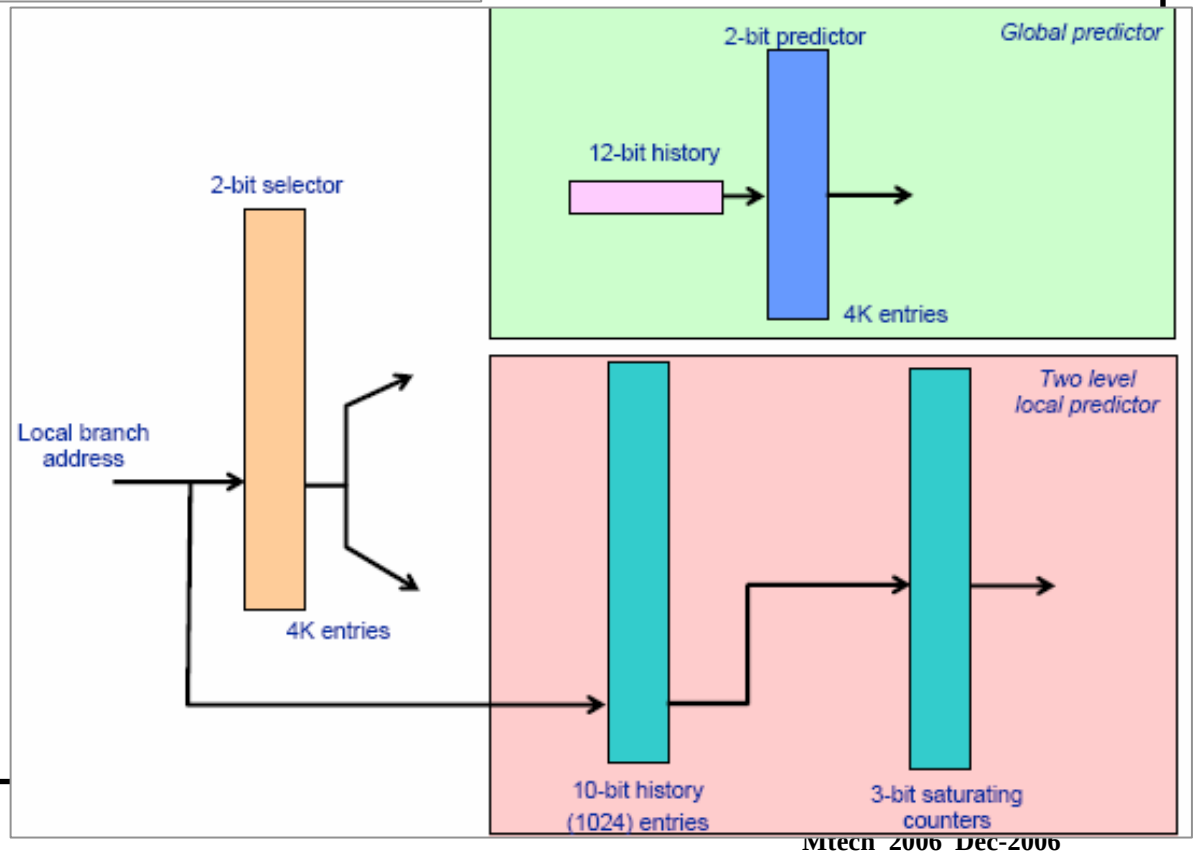
Global Predictor

- The shift register captures the path through the program
- For each unique path a counter is maintained
- Prediction is based on the behavior history of each path
- Shift register length determines program region size



Specific transitions between states determined accuracy of individual predictors





- Each SPE is composed of a "Synergistic Processing Unit" ("SPU"), and a "Memory Flow Controller" ("MFC") (DMA, MMU, and bus interface).
- An SPE is a RISC processor with 128-bit SIMD organization . With the current generation of the Cell, each SPE contains a 256 KiB instruction and data local memory area (called "local store") which is visible to the PPE and can be addressed directly by software.
- Each SPE can support up to 4 GB of local store memory.
- SPEs can be loaded with small programs (similar to threads), chaining the SPEs together to handle each step in a complex operation. (For instance, a set-top box might load programs for reading a DVD, video and audio decoding, and display, and the data would be passed off from SPE to SPE until finally ending up on the TV.)
- Another possibility is to partition the input data set and have several SPEs performing the same kind of operation in parallel.
- At 3.2 GHz, each SPU gives a theoretical 25.6 GFLOPS of single precision performance.
- Cell is overall 3 to 12 times faster on every type of high performance computation tasks.