Study of Speculative Execution and Branch Prediction

Credit Seminar Report

Submitted in Partial fulfilment of the requirements

for the degree of

Master of Technology

by

Maj Chetan Dewan

(06305403)

Under the guidance of

Dr MR Bhujade

Department of Computer Science and Engineering

INDIAN INSTITUTE OF TECHNOLOGY BOMBAY

November 2006

Acknowledgement

I would like to thank Dr. MR Bhujade for his invaluable support, encouragement and guidance. He is a constant source of inspiration and his guidance has made my endeavour an exciting experience.

Maj Chetan Dewan

Abstract

Current trends in processor design are pointing to deeper and wider pipelines and superscalar architectures. Modern superscalar processors rely heavily on speculative execution for performance. Despite continuous improvement in branch prediction algorithms, branch misprediction remains a major limitation on microprocessor performance. As pipelines are widened or stretched deeper, branch prediction assumes an even more crucial role in processor performance. This paper proposes to look at branch prediction techniques in aid of speculative execution, whereby the processor continues executing the predicted path of a branch before the branch condition is resolved. In this paper we look at some popular static and dynamic prediction schemes their advantages disadvantages. We further take a closer look at some of the popular microprocessors and various modifications to these techniques that are in use to improve performance.

CONTENTS

Abstract
Table of Contents

1. INTRODUCTION **1**
- 1.1. Motivation 1
- 1.2. Scope of the Seminar 2

2. SUPERSCALARS PROCESSORS : AN OVERVIEW **3**
- 2.1. Pipelining 3
- 2.2. Superscalar processors 4
- 2.3. Constraints/Hazards in pipeling 4
- 2.4 Speculative execution 5
- 2.5 Speculation and prediction 5
- 2.6 Reorder Buffer 6

3. SPECULATION **8**
- 3.1. Speculation by example 8
- 3.2 Wide issue speculation 10
- 3.3 Static Superscalar 11
- 3.4 Dynamic Superscalar 11
- 3.5. Speculative Superscalar 11
- 3.6 Speculative SMT 11

4. BRANCH PREDICTION **13**
- 4.1. Why predict 13
  - 4.1.1 Speculative execution to enhance instruction-level parallelism 13
  - 4.1.2 Compiler Optimisations 13
  - 4.1.3. Hardware reasons 13
- 4.2. Speculative Performance 14
- 4.3 Types of Predictors 15
  - 4.3.1 Trivial prediction 15
  - 4.3.2 Static prediction 15
  - 4.3.3 Dynamic prediction 15
  - 4.3.4 PHT 16
  - 4.3.5 NLP 16
  - 4.3.6 Bimodal Predictor 17
  - 4.3.7 Local BP 18
  - 4.3.8 Global BP 19
  - 4.3.9 Gslect 20

4.3.10 Gshare 20
4.3.11 Combined BP 21
4.3.12 Two levell adaptive BP 22
4.3.13 Aliasing 23
4.3.14 Multiple BP 23
4.3.15 Multiple 2 level adaptive BP 23
4.3.16 Agree 24
4.3.17 Overriding BP 24
4.3.18 Neural BP 25
4.4 Value prediction 27

5. CURRENT METHODS USED IN PROCESSORS **29**
5.1. Branching improvements in modern processors 29
5.1.1 Predication 29
5.1.2 Early Branch condition testing 29
5.1.3 branch hinting 29
5.2 Control Speculation 30
5.3 Data Speculation 30
5.4 Explicit cache line prefetching 30
5.5 Cache hints 31
5.6 Intel Wide Dynamic Execution 31
5.7. Intel Itanium (IA-64) 32
5.8 AMD Opetron 32
5.9 IBM Cell Processor 33

6. CONLUSION **35**
REFERENCES **36**

Chapter-1
INTRODUCTION

1.1 Motivation

As most computer programs respond to user inputs, there is no way around the fact that portions of a program need to be executed conditionally. Micro programmed processors of the 80’s took multiple cycles per instruction, and generally did not require branch prediction. The VAX 9000 was both micro programmed and pipelined, and probably did some branch prediction. The first commercial RISC processors, MIPS and SPARC, did only trivial "not-taken" branch prediction. Because they used branch delay slots, fetched just one instruction per cycle and executed in-order, there was no performance loss. These architectures advocate simplicity and had instructions that operated only on registers. Only load/store instructions involve memory. The RISC architectures did not have any hazard detection/resolution hardware as compilers were given the responsibility to generate hazard free programs. Compilers for these machines generated code such that instructions active inside the pipeline did not face data dependencies despite being simultaneously active at different stages in the pipeline by keeping such instructions away from each other. They employed in order issue, in order execution and in order completion of instructions. This limited the IPC to 1 but seldom reached close to 1.

Branches are very common in x86 code and represent a real problem for pipelining, because they mean that you can't always rely on the fact that instructions will follow a linear sequence. Pipelining means starting the next instruction before the first one has completed. When we come to a conditional test instruction (an "if.. then" instruction), we don't know until the condition test has been executed, which instruction is next, so what are we supposed to put into the pipeline next? A less sophisticated processor will stall the pipeline until the result is known, hurting performance. More advanced processors will speculatively execute the next instruction anyway, with the hope that it will be able to use the results if the branch goes the way it thinks it will. Still more advanced processors combine this with branch prediction, where the processor can actually predict with fairly good accuracy which way the branch will go based on past history.

Current trends in processor design are pointing to deeper and wider pipelines. Despite continuous improvement in branch prediction algorithms, branch misprediction remains a major limitation on microprocessor performance. To reduce the instances of branch misprediction a technique called speculative execution is used to execute instructions that exist beyond a conditional branch that have not yet been resolved, and ultimately to commit the results in the order of the original instruction stream.

1.2 Scope of Report

This report is meant to be a survey of speculative execution technique which aims at execution of instructions before it is known, if it is safe to do so ie it is free of hazards. The scope is restricted to the understanding of a few techniques only. The focus is mainly on the study of some basic branch prediction techniques and how they have been modified in some current architectures to aid speculative execution.

Chapter-2
SUPERSCALAR PROCESSORS: AN OVERVIEW

2.1 Pipelining

When programmers (or compiler) write assembly code, they make the assumption that each instruction is executed before execution of the subsequent instruction is begun. This assumption is invalidated by pipelining. Unfortunately, not all instructions are independent. In a simple pipeline, completing an instruction may require five stages. To operate at full performance, this pipeline will need to run four subsequent independent instructions while the first is completing. If fourth instruction depends on the output of the first instruction which is not available, the pipeline control logic must insert a stall or wasted clock cycle into the pipeline until the dependency is resolved. While pipelining can in theory increase performance over an unpipelined core by a factor of the number of stages (assuming the clock frequency also scales with the number of stages), in reality, most code does not allow for ideal execution.

A processor employing a pipeline in its implementation architecture (micro architecture) is said to implement Instruction Level Parallelism (ILP), another type implemented in today's processors is multithreading. A thread is a light weight process having independent sequence of instructions. ILP as well as Multithreading are commonly implemented in today's high performance microprocessors to enhance performance. ILP performance is measured by number of Clocks per instruction (CPI) Conversely the term instructions executed per clock (IPC) is also used. Architects want to achieve 1 CPI or > 1 IPC. Today, the term ILP is almost synonymous with pipeline execution of the instructions along with possible multithreading. A single threaded Pipeline processor ideally executes one instruction per clock, however achieving this goal is not easy. Primary trouble areas are conditional branch instructions, latencies in memory and floating point operations. These often stall the pipeline resulting in loss of performance due to lost clock cycles.

Many designs have pipelines as long as 7, 10 and even 31 stages (like in the Intel Pentium 4)[Pentium 4]. The Xelerator X10q has a pipeline more than a thousand stages long [Xelerator]. The downside of a long pipeline is when a program branches and wrong direction is taken, the entire pipeline must be flushed, a problem that branch prediction helps to alleviate. Branch prediction itself can end up exacerbating the problem if branches are predicted poorly. In certain applications, such as supercomputing, programs are specially written to rarely branch and so very long pipelines are ideal to speed up the computations, as long pipelines are designed to reduce clocks per instruction (CPI). Branching happens constantly in many common applications such as office software etc, significantly reducing the speed gain of pipelining.

2.2 Superscalar processors [Superscalar]

A processor which can issue and or execute more than one instruction per clock is called a superscalar processor. These processors employ out of order execution governed by data dependencies. Usually these processors have many functional units operating in parallel. To keep these units from idling a steady supply of instructions is required every clock. These processors do not depend on the compiler to reorder the code but has built in scheduling hardware that dynamically executes the instructions. Although super scalar machines exhibit a good instruction per cycle (IPC) rate, complicated hardware is required since they rely on dynamic instruction scheduling through hardware to exploit ILP. This complication imposes penalties on both hardware cost and cycle time. Furthermore, a run-time scheduler is unable to achieve sophisticated instruction scheduling due to complexity limits. The schedulers for these and a overall mechanism was first implemented in CDC 6600 called 'Scoreboard' a method of implementing dynamic scheduling [James E. Thornton-64]. Like any good dynamically scheduled machine, the scoreboard monitors each instruction waiting to be dispatched. Once it determines that all the source operands and the required functional units are available, it dispatches the instruction so that it can be executed. However, the scoreboard is limited in that it does not handle WAR and WAW hazards very well. Most processors today use ideas and principles of CDC Scoreboard with changes and ides borrowed from Tomasulo's algorithm. Fig 2.1 shows a block schematic of a superscalar processor.

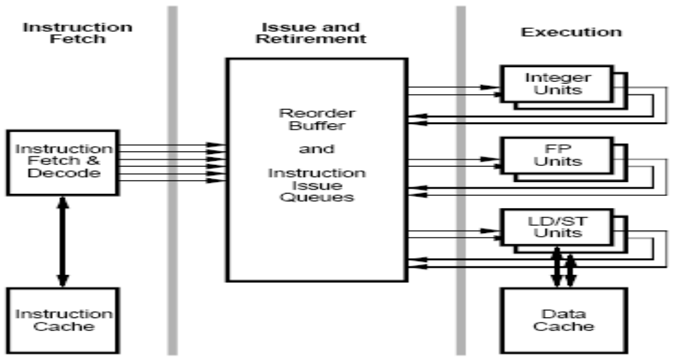


Fig 2.1: Representation of Superscalar Processor

[Courtesy Kunle , Basem , Lance , Ken , Kunyung]

2.3 *Constraints/Hazards in Pipelining*

Three types of constraints exist in exploiting ILP:-

(a) resource conflicts
(b) data dependence
(c) control dependence

Reducing constraints is necessary to increase exploitable ILP. Speculative execution is a technique to remove control dependence among these constraints.

2.4 *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 Speculative execution aims at execution of instructions before it is known if it is safe to do so ie it is free of hazards. It relies on branch prediction to get the branch direction right most of the times.

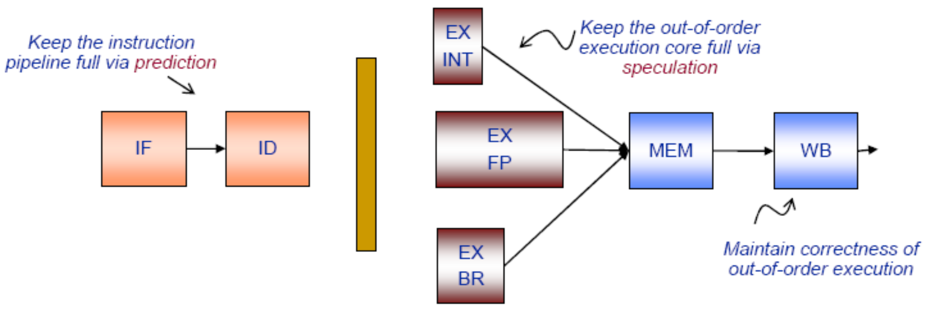

Keep the instruction pipeline full via prediction
IF
ID
EX INT
EX FP
EX BR
Keep the out-of-order execution core full via speculation
MEM
WB
Maintain correctness of out-of-order execution

Fig 2.2: Representation of Prediction and Speculation
[Courtesy presentation by Krishna V. Palem, Weng Fai Wong, and Sudhakar Yalamanchili]

2.5 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 as an extension of dynamic scheduling is composed of :-

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

The Challenges to this are :-

- Handling multiple executions completions/cycle
- Enforcing dependencies to ensure correctness
- Handling exceptions

2.6 The Reorder Buffer

Principle

Fig 2.3 shows a basic processor data path. Instructions are fetched , decoded , executed and finally retired. A basic block is defined to be instructions between Branches, whether they are taken or not taken in any program. Therefore a block simply is a group of sequential

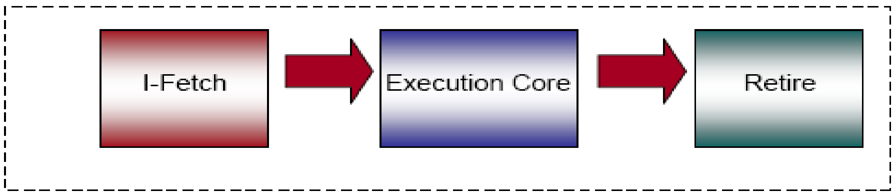


Fig 2.3: Processor Datapath
[Courtesy presentation by Krishna V. Palem, Weng Fai Wong, and Sudhakar Yalamanchili]

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. While the basic block sizes are not very large, Prediction can increase the issue rate but not the 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. The basic idea is 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. The criteria for this is that commitment should be performed in program order. To enforce this criteria we have to reorder the instructions that complete out-of-order. This is where the Reorder Buffer comes into play. The Reorder Buffer is a FIFO circular queue which does not take care of flow dependences fig 2.4 .

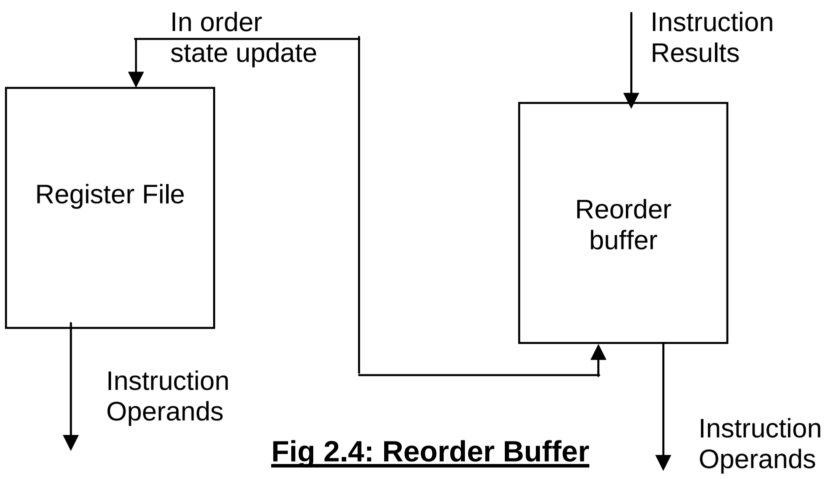


Fig 2.4: Reorder Buffer

There are basically three steps to reordering as shown in fig 2.5 below:-

- Every instruction gets a reorder entry allocated in the order it is issued – the entry is marked “invalid”
- When an instruction completes, it writes its result to the corresponding entry in the reorder buffer – the entry is now valid
- When the entry at the head of the reorder buffer is “valid” it is committed to the register file.

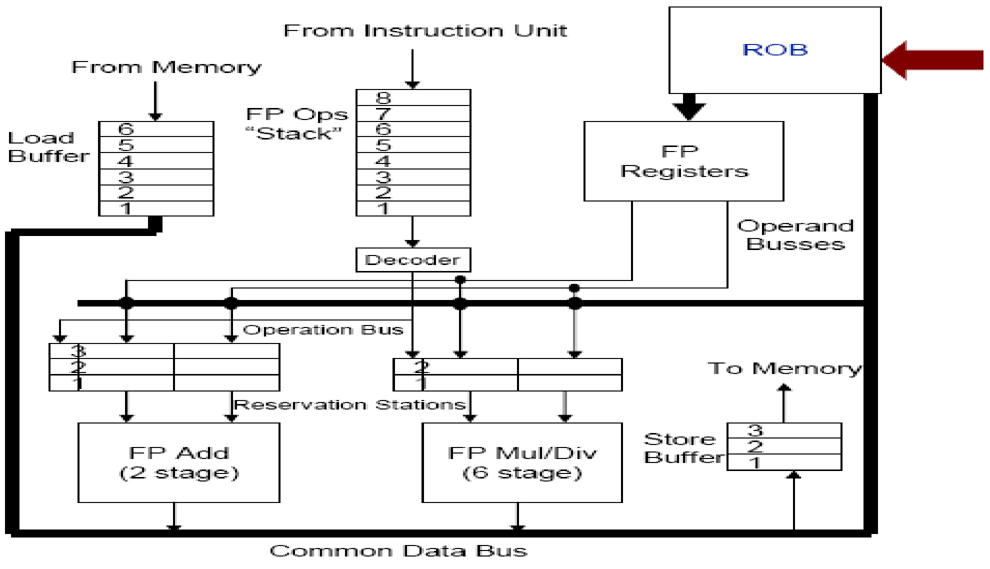


Fig 2.5: Reordering Steps
[Courtesy presentation by Krishna V. Palem, Weng Fai Wong, and Sudhakar Yalamanchili]

Chapter-3
SPECULATION

3.1 Speculation by example

An instruction is said to be control-dependent upon a conditional Branch, it is unknown that instruction should be executed or not unless that branch is determined. [Hideki, Chikako, Tetsuya, Masao 05] Consider the following code sample:-

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

We do not know whether instruction 20 or 30 should be executed or not until branch instruction 10 is executed. Speculative execution allows instruction 20 and/or 30 to be executed before the execution of branch instruction 10 is completed. In static instruction scheduling, the compiler moves instruction 20 and/or 30 above branch instruction 10 for speculative execution.

There are two problems in the speculative code motions. The first problem is to preserve program semantics. A code motion is said to be illegal if the moved operation changes the original program semantics. In the example above, the code motion of instruction 20 above instruction 10 is illegal because instruction 20 overwrites register r 2 whose previous value is necessary for instruction 30, similarly, an illegal code motion exists on memory locations. The compiler can transform the illegal code motion on a register to legal through register renaming. In register renaming, the compiler assigns a register which is not live on the side-effects causing path as the destination register. The compiler then inserts an instruction which copies the value from the newly assigned register to the original destination register. In the example above, the compiler can speculatively move branch instruction 20 above instruction 10 through register renaming as the following:-

```
20:     r5 = load r3;
10:     if (rl)
40:     r2 = r5;
```

```
        else
30:     r4 = r2 + r3;
```

Here register r5 is not live on the path to the ***ELSE*** part. The destination register of instruction 20 is renamed into a dead register (r5) in instruction 20 , and copy instruction 40 which copies the result into the original register is inserted. This renaming technique, however, cannot be applied to illegal code motion on a memory location.

The second problem is to handle an exception caused by a speculative operation. A code motion is said to be unsafe If the moved operation may cause an exception. In the example above, the code motion of instruction 20 above branch instruction 10 is unsafe because the load instruction may cause an exception. An exception caused by a speculative instruction is termed a speculative exception. If a speculative exception occurs, immediate handling like non-speculative exception handling incorrectly terminates the program or decreases performance because it is unknown whether that handling is necessary or not. Thus, handling of speculative exceptions should be postponed until the result of the excepting instruction is found to be really necessary. Although a compiler can transform an illegal code motion to legal, a compiler cannot transform an unsafe code motion to safe. Therefore, a compiler must conservatively schedule instructions which may cause an exception.

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.

Besides postponing handling, there is one more requirement for the handling of speculative exceptions: restarting the process. That is, if a caused speculative exception is non-fatal, the process should be restarted after the handling. This restart problem includes two difficult problems. The first problem is to select instructions which must be re-executed. 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. The second problem exists in the recovery process.

To preserve the program semantics, all of the operands of the re-execution instructions in the recovery process must be available. 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. For more understanding of these problems, consider the following program segment:-

```
10:     rl = load r2 ;

20:    r3=r3+l;

30:    r4 = rl + r5;

40:     r6=r4 &l;

50:     branch LAB if (r3 )
```

Here instructions 10, 30, and 40 are speculative instructions upon branch instruction 50, while instruction 20 is a non-speculative instruction If speculative instruction 10 causes an exception, exception handling must be postponed until branch instruction 50 is executed. In other words, the handling of the exception must be postponed until the exception is committed. If the exception is committed, the exception is handled; otherwise the exception is squashed. Since 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.

3.2 Wide issue and speculation

A wide issue processor has an instruction fetch unit that fetches and supplies multiple instructions per clock(fetch bandwidth) to the decoder stage which in turn should issue multiple instruction per clock (issue bandwidth) to functional units. However, issuing multiple instructions per cycle adds more complexity to the micro architecture including the instruction issue logic which has to check for the hazards among these **N** instructions (issue packet) that are to be issued in parallel, There are many variations in wide issue processor micro architectures which are discussed briefly here.

3.3 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.

3.4 Dynamic superscalar (In 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.

3.5 Speculative Superscalar

In superscalar processors speculation hides branch latencies and thereby boosts performance 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.

3.6 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.***

In a VLIW processor or in any other machine in which compilers must **arrange** instructions into data-independent groups, the compiler must look 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. Since the compiler is bundling several instructions into one long instruction at compile time, practitioners often avoid confusion by using the term operation for single RISC-level instructions, and instruction for the long instruction produced from several operations. In this paper, however, we refer to the smaller, individual operations as instructions.) The candidate set usually contains many potentially speculative instructions, so the compiler must be able to use branch prediction to schedule high probability and reject low probability speculative instructions. Techniques like software pipelining concentrate on generating code for tight inner loops. In those cases, the only required branch prediction is implicit in the assumption that loops are repeated. For code other than tight loops, code generation techniques like trace scheduling **,** or others that deal with flow of control more general than tight loops, must rely on branch predictions to select candidate instructions. Obviously, since this work is being done by the compiler, static branch prediction must be used.

To avoid complexity and problems of parallel instruction issue in main scheduler in decode stage, a very large instruction word(VLIW) processor was suggested. 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. VLIW instruction forms an issue packet that is issued all at once in a clock. The compiler is completely responsible for making sure that instructions within long instructions are independent, and that instruction dependencies will all have been satisfied.

Chapter-4
BRANCH PREDICTION

4.1 Why Predict Conditional Branch Directions-----?

There are several important classes of reasons for attempting to predict which direction a conditional branch will go in before it is executed:

4.1.1 **Speculative execution to enhance instruction-level parallelism*.*** Machines that do speculative execution start the execute phase of instructions before it is certain that they should be executed, possibly allowing their execution to use otherwise idle machine resources. Predictions can allow us to execute speculatively the instructions that are most likely to be profitable.

4.1.2 **Compiler Optimizations.** Important classes of compiler optimizations rely on dynamic information to decide among alternative code motions, transformations, etc.

4.1.3 **Hardware reasons*.*** CPU fetch execute pipelines can fetch and start to decode an instruction before it is certain that a conditional branch it follows will go in that direction. If a branch goes in the expected direction, a pipeline bubble can be avoided. In addition, elements in the memory hierarchy, such as instruction caches, can be instructed to prepare for the coming instruction stream, lowering the latency of memory instructions.

Almost all pipelined processors do branch prediction of some form, because they must guess the address of the next instruction to fetch before the current instruction has been executed. Many earlier micro programmed CPUs did not do branch prediction because there was little or no performance penalty for altering the flow of the instruction stream. Branch predictors are crucial in today's modern, superscalar processors for achieving high performance. They allow processors to fetch and execute instructions without waiting for a branch to be resolved. To get a lot of instruction-level parallelism, many instructions must be moved up past conditional branches that they followed in the source. If one blindly executes all instructions that are data-ready at all times, an enormous amount of hardware will be required, and most of it will be wasted doing instructions that were not in the path of the flow of control that eventually occurred. Thus we must have some way of picking only the highest probability instructions to execute in a given cycle, and thus we must predict which way the branches in the program are likely to go.

4.2 **Speculative Performance**

There is a cost associated with tracking speculative instructions that caused faults, since one does not know whether or not these faults should be serviced until the flow of control is resolved. One must also deal with the possibility of degraded performance due to page and cache misses caused by unnecessary speculative instructions. This is an active research area .We must therefore have some method of gauging which speculative instructions are worth the trouble, and which aren't.

Branches are usually handled by dynamic branch prediction and speculative execution of the branch path with the highest likelihood. 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. However, some branches are unpredictable, resulting in high misprediction rates. Nevertheless such branches are speculatively executed in contemporary microprocessors.

One way out is predication that loads and decodes instructions of both branch paths, even though only one branch path is executed. Only instructions with the predicate 'true' as additional operand input are executed, but all instructions of both paths are loaded, decoded, and dispatched to the respective instruction window. Compiler scheduling is improved by larger basic blocks, but the instructions that are on the wrong path consume fetch and decode bandwidth and may clog the instruction window.

Another way to continue executing instructions in the pipeline for good Instruction-level parallelism (ILP) decode and instruction fetch from branch target address may be done in advance. Advance fetch is possible only if we know the branch target address which may involve a register that is not yet updated by previous instruction, and hence computing branch target address is also a source of data hazard. Architects use speculation on the branch target address value. This is done by using Branch Target Buffers (BTB) and a separate cache for this purpose. When the branch occurs first time, its target address is not in the BTB cache and miss occurs and processor stalls till the address is computed and is put in BTB cache. When it encounters the same branch again, its target address exists in the BTB which is read and passed on as target address. On the other hand if actual target address is different (which happens quite rarely), speculative execution is undone and execution begins from this address. Branch target prediction

(BTP) success rate is very high in most cases. Let us look at some common Branch prediction Schemes.

4.3 Types of Predictors

4.3.1 **Trivial Prediction** The early implementations of SPARC and MIPS (two of the first commercial RISC architectures) did trivial branch prediction: they 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. Both CPUs evaluated branches in the decode stage and had a single cycle instruction fetch. As a result, the branch target recurrence was two cycles long, and the machine would always fetch the instruction immediately after any taken branch. Both architectures defined branch delay slots in order to utilize these fetched instructions.

4.3.2 **Static Branch 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 mis-predicts the very last branch of the loop . Static prediction is used as a fall-back technique in most processors with dynamic branch prediction when there isn't any information for dynamic predictors to use. Both the Motorola MPC7450 (G4e) and the Intel Pentium 4 use this technique.

Static methods, attach one direction to each conditional branch at compile time. The branch is then always predicted to go in that direction. Static methods usually require little or no hardware and allow time to compute whatever quantities are required.
Some methods of static branch prediction are:-

1. The programmer inserts directives.
2. The compiler examines the source and uses heuristics (which might have any degree of sophistication).
3. The program is run, statistics are gathered and fed back into the source code, and the program is recompiled using those statistics.

4.3.3 **Dynamic Branch Prediction** 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. There are several predictors in use or being researched nowadays. Most predictors use two types of history:-

1. Branch direction pattern history(taken or not taken sequence)
2. Branch history (taken or not taken).

4.3.4 **Branch direction pattern History** A branch direction history per branch is maintained by a branch Pattern history Table (PHT). This is based on info whether a branch was recently taken or not taken and this info is maintained by a 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. A saturating counter is one which when incremented, increments by 1 till the count reaches the maximum and thereafter subsequent increments keeps the maximum count (instead of going to zero and overflowing). Similarly on each decrement, counter gets decremented by 1 but retains value 0 for subsequent decrements after reaching zero. It can give four states viz Strongly not taken, Weakly not taken, Weakly taken and Strongly taken.

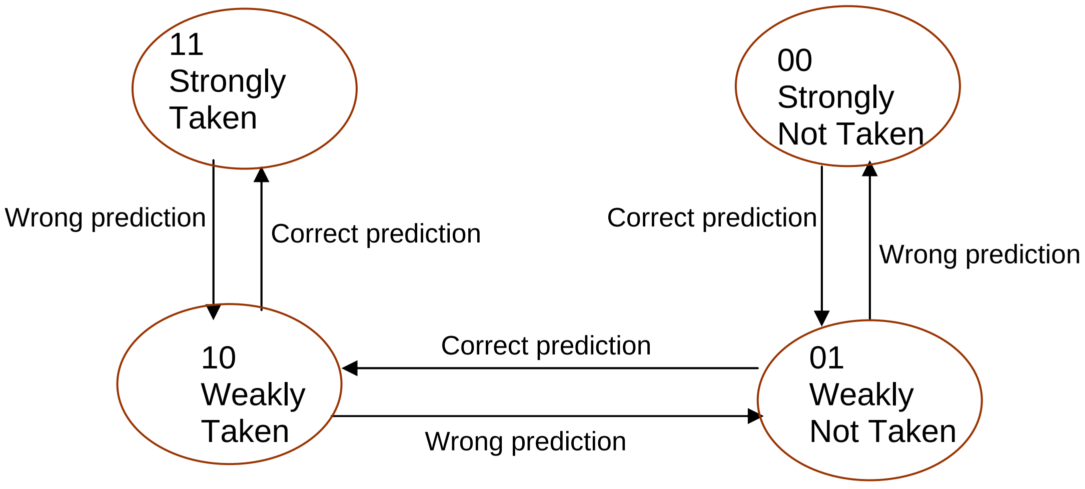


Fig 4.1:State Diagram of Saturating Counter

4.3.5 **Next Line Prediction.** Some superscalar processors (MIPS R8000, DEC Alpha EV6 and EV8) fetched with each line of instructions a pointer to the next line. This next line predictor is not directly comparable to the other predictors listed here because it handles 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.

4.3.6 **<u>Bimodal Branch Predictio*n*</u>** A bimodal branch predictor has a table of two-bit entries, indexed with the least significant bits of the instruction addresses. simplest one level predictor is based on a branch pattern History table(PHT), which is array of 2-bit saturated counters. Unlike the instruction cache, bimodal predictor entries typically do not have tags, and so a particular counter may be mapped to different branch instructions (this is called branch interference or branch aliasing), in which case it is likely to be less accurate. Each counter has one of four states:-

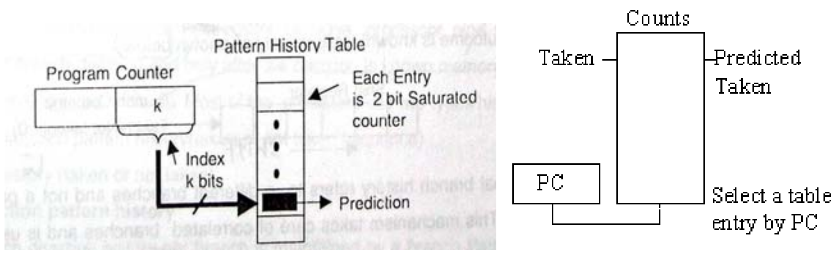
(a) Strongly not taken

(b) Weakly not taken

(c) Weakly taken

(d) Strongly taken

It is addressed by the lower order k bits from the branch address.



<u>Fig 4.2 One Levell Bimodal predictor.</u>

The k bit {index bits} from program counter addresses the table and gets the value of the counter. If count is more than half the max value(most significant bit=1), the branch is predicted as taken, otherwise it is predicted not to be taken. Subsequently when the branch is resolved the code is updated

When a branch is evaluated, the corresponding entry is updated according to the underlying Moore machine. In the case of two-bit saturating counters ("2-bit Smith Counters with Saturation"), branches evaluated as not taken decrement the state towards strongly not taken, and branches evaluated as taken increment the state towards strongly taken. The primary benefit of this two-bit saturating counter scheme is that loop closing branches are always predicted taken. A one-bit scheme (like the R8000), mispredicts both the first and last branch of a loop. A two-bit scheme mispredicts just the last branch. Similarly, on heavily biased branches which almost always go one way, a one-bit scheme mispredicts twice for each odd branch, and a two-bit scheme mispredicts once.

On the SPEC'89 benchmarks, very large bimodal predictors saturate at **93.5% correct**, once every branch maps to a unique counter. Because the bimodal counter table is indexed with the instruction address bits, a superscalar processor can split the table into separate SRAM for each instruction fetched, and fetch a prediction for every instruction in parallel with fetching the instruction, so that the branch prediction is available as soon as the branch is decoded.

4.3.7 **Local Branch Prediction** Bimodal branch prediction mispredicts the exit of every loop. For loops which tend to have the same loop count every time (and for many other branches with repetitive behaviour), we can do much better.

Local branch predictors keep two tables. The first table is the local branch history table. It is indexed by the low-order bits of the branch instruction's address, and it records the taken/not-taken history of the n most recent executions of the branch.

The other table is the pattern history table. Like the bimodal predictor, this table contains bimodal counters; however, 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.

On the SPEC'89 benchmarks, very large local predictors saturate at **97.1% correct**. A branch prediction method close to one developed by Yeh and Patt [YP92] that can take advantage of is shown in below figure.

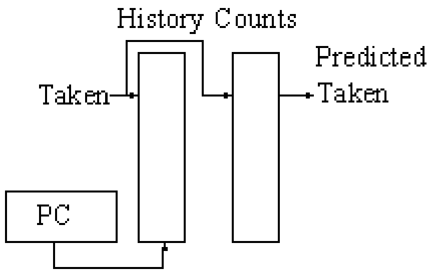


Fig 4.2 Local branch predictor.

The first table records the history of recent branches. The second table is an array of 2-bit counters identical to those used for bimodal branch prediction. But, if there are more branches in the program, a local prediction can suffer from two kinds of contention. First, the branch history may reflect a mix of histories of all the branches that map to each history entry. Second, since there is only one counter array for all branches, there may be conflict between patterns.

However local prediction is slower than bimodal prediction because it requires two sequential table lookups for each prediction. A fast implementation would use a separate bimodal counter array for each instruction fetched, so that the second array access can proceed in parallel with instruction fetch. These arrays are not redundant, as each counter is intended to store the behaviour of a single branch

4.3.8 **Global Branch Prediction** Global branch predictors make use of the fact that the behaviour of many branches is strongly correlated with the history of other recently taken branches. We can keep a single shift register updated with the recent history of every branch executed, and use this value to index into a table of bimodal counters. This scheme, by itself, is only better than the bimodal scheme for large table sizes, and is never as good as local prediction.

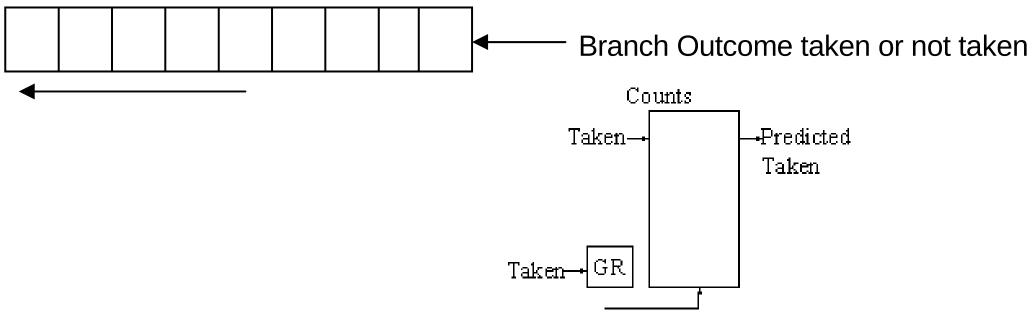


Fig 4.3: Global branch predictor.

A single shift global register GR records the direction taken by the most recent n conditional branches. This global branch prediction is able to take advantage of below patter.

If (x < 1) ….

If (x > 1)…

This method works particularly well when the direction taken by sequentially executed branches is highly correlated. But there are some other schemes like this using other information.

4.3.9 **Global predictor with Index Selection (gselect)** In fact, global history information is less efficient at identifying the current branch than simply using the branch address. [Chang 02]. The main point of this is that 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.

4.3.10 **Global History with Index sharing (gshare)** Another variation in global prediction was proposed by S. McFarling [Scott93] is using the exclusive OR of the branch address with the global history to have more information than either component alone. The reason that this is better than Global predictor with Index Selection is the exclusive OR of the branch address with the global history is more space efficient that just concatenation of them.

If, instead, we index the table of bimodal counters with the recent history concatenated with a few bits of the branch instruction's address, we get the **gselect** predictor. Gselect does better than local prediction for small table sizes, and local prediction is only slightly better for table storage larger than 1KB.

We can do slightly better than gselect by XORing the branch instruction address with the global history, rather than concatenating. The result is gshare, which is a little better than gselect for tables larger than 256 bytes.

On the SPEC'89 benchmarks, very large gshare predictors saturate at **96.6% correct**, which is just a little worse than large local predictors. **Gselect** and **gshare** are easier to make fast than local prediction, because they require a single table lookup per branch. As with bimodal prediction, the table can be split so that parallel lookups can be made for each instruction fetched, so that the table lookup can proceed in parallel with instruction load

4.3.11 **Combined branch predictio*n*** Scott McFarling [Scott93] proposed combined branch prediction in his 1993 paper. Combined branch prediction is about as accurate as local prediction, and almost as fast as global prediction.

As we have seen, the 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. These predictors could be one of the predictors discussed before. But we must maintain an additional counter array which serves to select the best predictor to use. This is shown in below figure.

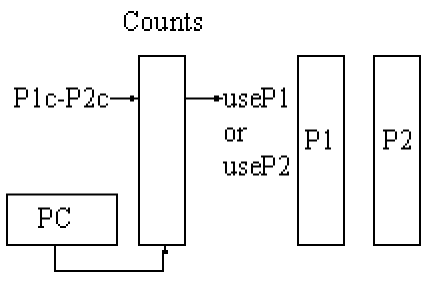


Fig 4.4: Combined branch predictor.

In here, P1c and P2c denote whether predictors P1 and P2 are correct respectively. So P1c-P2c is greater than 0, we choose P1 predictor to improve branch prediction accuracy.

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.

4.3.12 **Two-Level Adaptive Branch Predictors** [YP92] Two-Level Adaptive Branch prediction uses two levels of branch history information to make prediction. The first level is the history of the last k branches encountered. This information is saved in Branch history register (BHR). And we call the collection of BHRs the branch history table(BHT). The second level of the predictor records the branch behavior for the last j occurrences of the specific pattern of the k branches. We call the structure saving this information the pattern history table (PHT). In other words, this scheme is a kind of combining of local and global branch prediction.

In the first level, depending on which variation of the model we implement, the last branches can mean the actual last k branches encountered (G), the last k occurrences of the same branch instruction (P), or the last k occurrences of the branch instruction from the same set (S). Also, there is 3 choices in second level. So, we have 9 variations on this scheme, which is summarized in below table[YP93].

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.

Global history schemes make effective prediction due to their correlation with previous branches. On the other hand, when the global history is used, the pattern history of different branches interfere with each other if they map to the same pattern history table. Therefore, global history schemes require long branch history table.

When the per-address branch history is used, the pattern history of different branches tend to interfere less with each other, therefore, fewer pattern history tables are needed. But per-set history schemes require even higher implementation costs than global history schemes due to the separate pattern history tables of each set.

4.3.13 **Aliasing Problem** The one of the biggest problems in global schemes like two-level adaptive branch prediction is aliasing between two indices (an index is typically formed from history and address bits) that map to the same entry in Pattern History Table (PHT). This results in increasing a misprediction. So, many papers are published to address the aliasing problems. But I'm not sure that these algorithms are used in current processors like Pentium family.

4.3.14 **Multiple Branch Prediction** Superscalar machines, where the hardware can issue more than one instruction each cycle, are now common nowadays. So, in superscalar machine, we have to exploit predicate execution to schedule instruction execution along multiple execution paths in one cycle.

This means superscalar machines require a multiple branch prediction algorithm. At the same time that multiple branch paths are being predicted, the addresses of the basic blocks following those branches must be determined.

4.3.15 **Multiple Branch Two-Level Adaptive Branch Predictor** A method was proposed in [YP93] achieve more high prediction accuracy using Two-Level Adaptive Branch Predictor and multiple branch prediction. In here, there are the Branch Address Cache (BAC) which is a hardware structure to provide multiple fetch addresses of the basic blocks each branch and an instruction cache with enough bandwidth to supply a large number of instruction from non-consecutive basic blocks.

The primary basic block is the basic block dynamically following the primary branch. There are two possibilities for the primary basic block : the target and the fall-through basic blocks of the primary branch. These will be denoted as T or N. the secondary basic block is the basic block following the secondary branch. The secondary basic block can be one of up to 4 different blocks depending on the direction of the primary and the secondary branches. Below figure is a detail algorithm to make 2 branch predictions from a single branch history register.

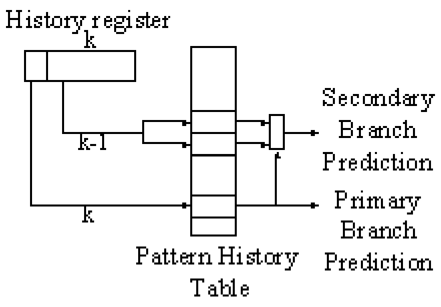


Fig 4.5: Two levell predictor.

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. This method improves performance in superscalar machine due to the multiple branch prediction, but as stated before, to fetch more basic block, BAC and an instruction cache with enough bandwidth are necessary.

4.3.16 **Agree prediction** Another technique to reduce destructive aliasing within the pattern history tables is an agree predictor. Some method is used to establish a relatively static prediction for the branch, perhaps a bimodal predictor or hint bits within the branch instruction. Another predictor (e.g. a gskew predictor) makes predictions, but rather than predicting taken/not-taken, the predictor predicts agree/disagree with the base prediction.

The intention is that if branches covered by the gskew predictor tend to be a bit biased in one direction, perhaps 70%/30%, then all those biases can be aligned so that the gskew pattern history table will tend to have more agree entries than disagree entries. This reduces the likelihood that two aliasing branches would best have opposite values in the PHT.

Agree predictors work well with combined predictors, because the combined predictor usually has a bimodal predictor which can be used as the base for the agree predictor. Agree predictors do less well with branches that are not biased in one direction, if that causes the base predictor to give changing predictions. So an agree predictor may work best as part of a three-predictor scheme, with one agree predictor and another non-agree type predictor

4.3.17 **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

4.3.18 **Neural Branch Predictors** The first dynamic neural branch predictors (LVQ and MLP) were proposed by Lucian Vintan ("L. Blaga" University of Sibiu, Romania) in his paper entitled "Towards a High Performance Neural Branch Predictor", Proceedings of The International Joint Conference on Neural Networks - IJCNN '99, Washington DC, USA, 1999. The neural branch predictor research was consistently developed further by Daniel Jimenez (Rutgers University, USA). He proposed in 2001 (HPCA

Conference) the first perceptron predictor, feasible to be implemented in hardware.

Branch predictors based on neural methods have been recently studied [Matt-04] showing that they are the most accurate predictors in the literature. In fact, neural networks can exploit much longer histories than conventional branch predictors. The simplest neural network is the perceptron, whose diagram is shown in Figure below :-

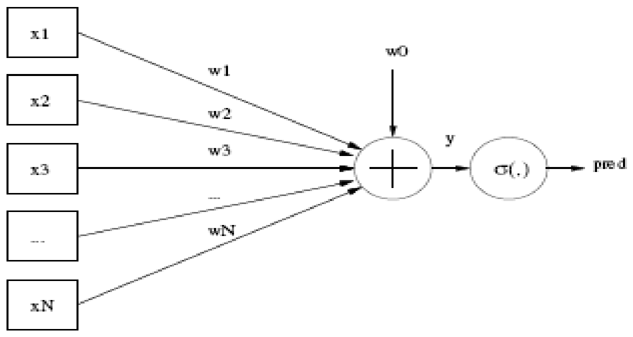


Fig 4.6: Perceptron neural network

The main disadvantage of the perceptron predictor consists in its high latency. Even if we are using some high-speed arithmetic tricks, the computation latency is relatively high comparing with the clock period of a deeply pipeline microarchitecture (Intel Pentium IV has 20 stages in its integer pipeline and researchers conclude that performance of aggressively clocked microarchitectures continues to improve until 52 stages).

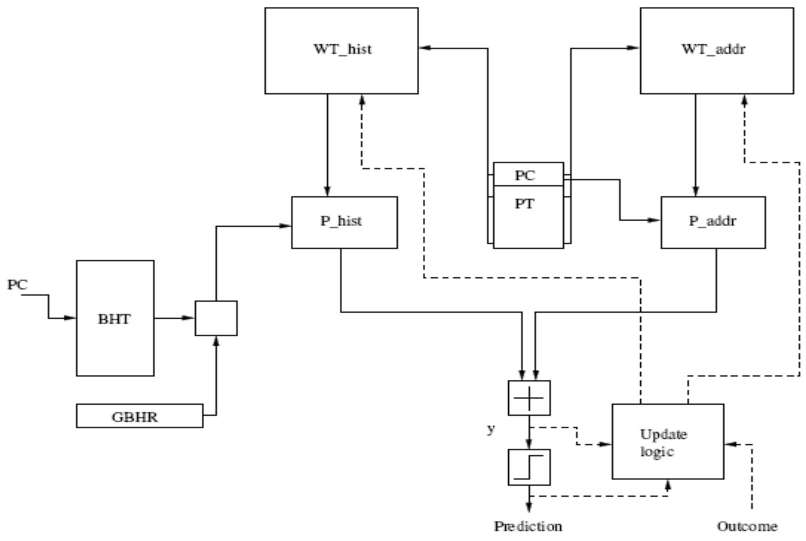


Fig 4.7: Combined Branch predictor

The *Combined Perceptron Branch Predictor*, proposed [Matt-04] is based on the idea to combine two different kinds of *Perceptron*: a *history-based* one and a *address-based* one. 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. The neural branch predictor concept is very promising. Most of the state of the art branch predictors are using a perceptron predictor (Courtesy Intel's Championship Branch Prediction Competition -). Intel already implements this idea in one of the Itanium's simulators.

4.4 **Value Prediction in Microprocessors**

One prominent trend in micro-architectural research is improving system performance by adding prediction and speculation to a processor’s core. Value prediction [Mikko John-05] is a type of prediction that has quite recently emerged from the research community, and numerous recent papers have demonstrated its performance potential. For decades, the 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, which describes the likelihood of the recurrence of a previously-seen value within a storage location inside a computer system. Value prediction consists of predicting entire 32- and 64-bit register values based on previously-seen values. This mechanism 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; namely, that real-world programs, run-time environments, and operating systems incur severe performance penalties because they are general by design. That is, they are implemented to handle not only contingencies, exceptional conditions, and erroneous inputs, all of which occur relatively rarely in real life, but they are also often designed with future expansion and code reuse in mind. Code that is aggressively optimized by modern, state-of-the-art compilers exhibits these tendencies.

Furthermore, it is possible that with micro architectural enhancements to a microprocessor implementation such as that in PowerPC 620 that enable value prediction can effectively exploit value locality to collapse true dependences, reduce average result latency, and provide performance gains.

Chapter-5
CURRENT METHODS : POPULAR PROCESSORS

In this section, we will see currently used speculative methods in popular processors from Intel AMD, IBM etc , but, It is extremely hard to find correct reference about performance improvement techniques used in these new processors. The reason for this is secrecy and competition in the processor market and large economic value involved so companies seem not to want to disclose how things work. Some improvements of modern processors are listed below.

5.1 Branching Improvements in modern processors

5.1.1 ***Predication.*** Predication allows a compiler to 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. [Itanium2-02]This is a compiler technique called if-conversion. The EPIC architecture provides for 63 addressable predicate bits, and these predicate bits control almost all EPIC instructions. In the illustration below, the normal coding involves a potentially mispredicted branch and a second branch to the continue block, but the predicated code removes both branches.

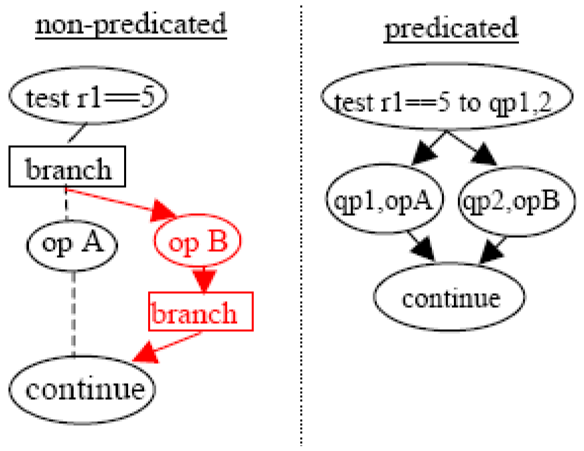


Fig 5.1.1 Predication Example

5.1.2 **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.

5.1.3 **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).

5.2 <u>Control Speculation</u>

This allows loads to be executed early by moving them ahead of branches without the penalty of spurious faults. [Itanium2-02]. If a speculative access is made to a page of memory that is absent, invalid, or one the user is not permitted to access, a conventional architecture would require a costly fault to be taken. 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 for integers, or a NaTVal is encoded in a floating-point operand. Most Itanium instructions will propagate an input NaT to their results, thus allowing a speculative code sequence to continue through multiple instructions correctly and efficiently.

5.3 <u>Data Speculation</u>

This allows loads to be advanced safely before stores that may change the value of the load. In modern programming languages it is common to allow programs to use pointers to data types. [Itanium2-02]. Sometimes when using pointers it is 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. Any store to that address between the advanced load and a later check will invalidate the entry in the ALAT. When the results of the advanced load are to be used in the code, a check instruction (ld.c or chk.a) is issued along with the use of the result of the advanced load. The check instruction does not have any execution latency. The check instruction queries the ALAT structure to make sure the advanced load's entry is still valid (i.e. a store to the same address hasn't been encountered between the advanced load and its use). If the ALAT indicates that the load data has been modified, then the load (and sometimes other code) is executed again, and the updated data value is loaded into the register file.

5.4 Explicit cache line prefetching

This is another 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. [Itanium2-02]. 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.

5.5 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. [Itanium2-02]. 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.

5.6 Intel Processors (Intel Wide Dynamic Execution)

True performance is a combination of both clock frequency (GHz) and IPC. As such performance can be computed as a product of frequency and instructions per clock cycle:

Performance = Frequency x Instructions per Clock Cycle

Dynamic execution is a combination of techniques (data flow analysis, speculative execution, out of order execution, and super scalar) that Intel first implemented in the P6 microarchitecture [Ofri -06] used in the Pentium Pro processor, Pentium II processor, and Pentium III processors. 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.

Now with the Intel Core microarchitecture, Intel significantly enhances this capability with Intel Wide Dynamic Execution; It enables delivery of more instructions per clock cycle to improve execution time and energy efficiency. Every execution core is wider, allowing each core to fetch, dispatch, execute, and return up to four full instructions simultaneously, (Intel's Mobile and Intel Net Burst microarchitectures could handle three instructions at a time,) Further efficiencies include more accurate branch prediction, deeper instruction buffers for greater execution flexibility, and additional features to reduce execution time.

One such feature for reducing execution time is macrofusion. 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. This increases the overall number of instructions that can be run within any given period of time or reduces the amount of time to run a set number of instructions. By doing more in less time, macrofusion improves overall performance and energy efficiency. The Intel Core microarchitecture also includes an enhanced Arithmetic Logic Unit (ALU) to further facilitate macrofusion. Its single cycle execution of combined instruction pairs results in increased performance for less power.

5.7 <u>Intel Itanium (IA-64)</u>

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. The EPIC architecture is in part an answer to the increasing complexity of OOO RISC designs, and an attempt to reset the hardware to a less complex design point [Itanium2-02]. OOO RISC designs are very complex and require extensive time-consuming processor validation efforts. It 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 as well as the micro architecture of clocks due to stalls on hazards. 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. However a program could be optimized for its execution time if the compilers take efforts to minimize the hazards by carrying out code movement.

5.8 AMD Opetron

Unlike the Xeon family processors, the Opteron processor has been optimized for a highly efficient pipeline at the expense of high clock speeds [Hewlet Packard -2005] . 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. As of this writing, the Opteron processor operates at up to 2.8 GHz for single-core versions. However, the 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—three integer operations per cycle. 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.

5.9 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 based on the POWER Architecture (a two-way SMT multithreaded Power 970 architecture compliant core), which is the basis of IBM's line of POWER and PowerPC offerings. 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

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. Cell .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.

The schematic below shows performance improvements from some of the new architectures.

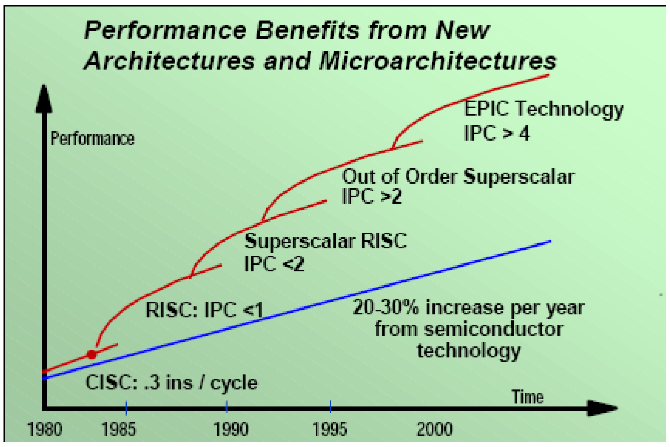


Fig 5.1: Performance chart various architectures
{Courtesy Hewlet Packard Technical White Paper}

Chapter-6
CONCLUSION

We have seen that compiler-based speculative execution has the potential to achieve both a high instruction per cycle rate and high clock rate. Pure compiler-based approaches, however have greatly limited instruction scheduling due to a limited ability to handle side effects of speculative execution. Significant performance improvement is, thus, difficult in non-numerical applications. We have also seen that most state-of-the-art microprocessors exploit ILP through superscalar techniques. Although super scalar machines exhibit a good instruction per cycle (IPC) rate, complicated hardware is required since they rely on dynamic instruction scheduling through hardware to exploit ILP. This complication imposes penalties on both hardware amount and cycle time. Furthermore, a run-time scheduler is unable to achieve sophisticated instruction scheduling due to complexity limits. On the other hand Very long instruction word (VLIW) machines, can potentially overcome these problems. In VLIW machines, instruction scheduling is optimized by the compiler, and consequently they need only simple hardware. The compiler fundamentally has the ability to optimize schedule code through analyzing critical paths from a large window of instructions and using sophisticated instruction heuristics for scheduling. Therefore most modern processors have started using a mix of these techniques such as those used by Intel Wide dynamic Execution as mentioned in chapter 5.

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

1. [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. [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

10. [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

11. [**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

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

13. [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

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

15 [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

1 http://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