On the complexity of Branch Prediction

Saeed Beiki
Secumania Security Group (SSG)
secumania.net

Abstract

More computing requirements have evolved more powerful processors. At the first era of industry for the aim of producing powerful processors, one of the most valuable pillars was Clock Speed. Later on when system designers came about to figure out that clock speed is not useful on its own, they planed to hire some architectural changes on the processor designing. And there have been innovations in this way, pipelines, stages and superscalar processing are well-known ones. Another one is Branch Prediction. Branch Prediction (BP) or Branch Prediction Unit (BPU) is responsible of acting when facing branch instructions. BPU itself consists of many phases and of course special calculations in order to take the appropriate branch and fulfill the present trace caches or some such for the core to execute from.

In this paper we will verify some branch prediction methodologies that can be used in real world, and add our own strategy named Return Buffer (SRB).

Background

The concept of branching within the microcode level of microinstructions in a processor is one of most important measurement elements on a specific processor. Each production company seems to have its own set of arrangements, but the core ideology behind all of them is the same. In this paper we will discuss some branch prediction methodologies and eventually come up with some tricky ways.

Why Branch Prediction is so important?

The general lookout of branch prediction unit among the other units can be seen in the picture below. As you can see the most labor of BPU is when it's being used on Out of Order (OOO) execution. Out of Order Unit in general terms executes the instruction slices out of order and commits the results back in the original order. For sufficiently executing the microinstructions the pipelines should be kept fulfilled all the time, or else bubble hazards intercept the flow of continues execution. So branch prediction is the most important factor in such a way of execution, because as a branch is taken wrong, the whole fetched pipeline should be flushed and new instructions to be fetched again. It wastes a lot of clock cycles. So at first glance BPU itself seems to be sufficient when it's considered correctly and fully working, but with consideration of pipelines and drive stages (in superscalar processing) it doubles the trouble as a wrong branch prediction.

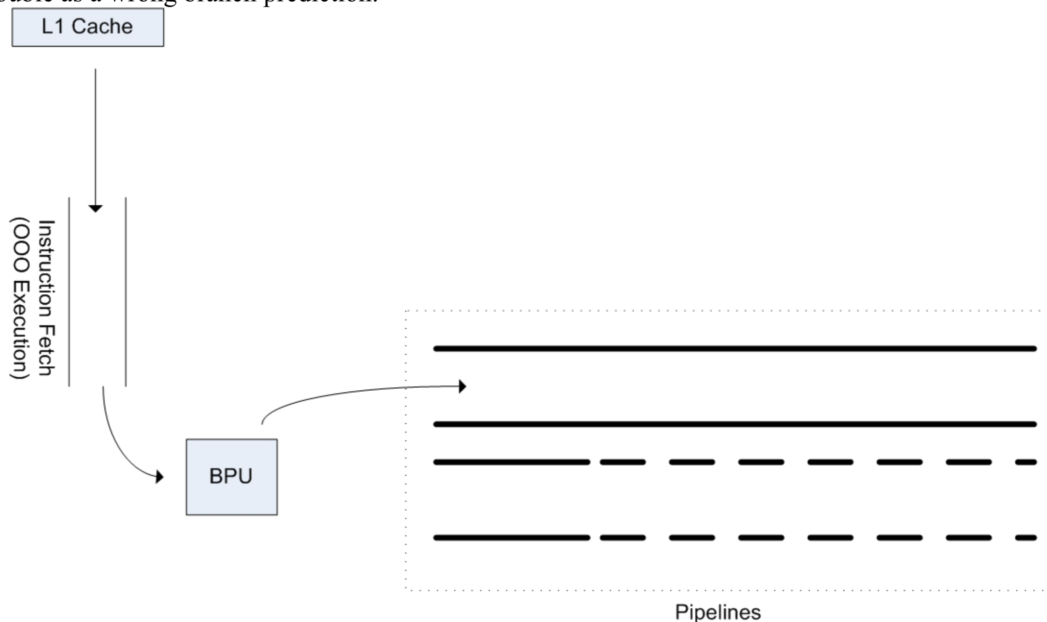

Figure 1. Possible Bubble Hazards driven by BPU in modern processor architectures

In order to avoid hazards triggered because of branch prediction unit, each vendor usually keeps its BPU details in dark. Each company's BPU has a regular accuracy that is posed by simply officials or measured by statistical analysis, but they all have a kind of inaccuracy, this is why it just remains "prediction"! For example Intel Netburst's BPU accuracy was estimated around %98. Thus we are going to know that the solution to improve the superscalar processing schemes is to speculate the most likely execution path by BPU. The success of Superscalar Processing approach is laid on accuracy of BPU.

A *branch instruction* is simply an instruction in the program flow where the next instruction is not going to be the next sequential one. There are two kinds of branch instructions: Unconditional

Branches (like jump instructions, goto commands, etc) and Conditional Branches (for instance if-else structures, for loops, while loops, etc). For the case of conditional branches the action for taking a branch depends on some condition evaluation to take the correct decision. Meanwhile to keep processor working all the time without stall times, it will take one of the possible execution paths, and fetches the instructions. So it's necessary that branch prediction algorithms try to predict the most likely execution path in a branch. If the prediction is true, then processor executes the fetched instructions; otherwise a dilemma named *misprediction* is occurred, and the instructions that speculatively were fetched have to be flushed, and the execution starts from mispredicted path again. Depending on the length of a pipeline, flushing and re-fetching instructions takes much times. On modern processors there are usually lengthy pipelines evolved (like 20-stage pipeline in NetBurst) and it keeps growing, so the wasted time because of flushing and re-fetching instructions takes long and much more clock cycles are going to be wasted. To speculatively executing instructions found right after a branch instruction, the processor needs some branching information:

- *The Branch Outcome-Result (BOS)*. In order for the processor to fetch and execute the correct instruction sequence, it should know the final result of a branch. There are two results: **Taken**, or **Not-Taken**. This information is not available before executing the instruction. Instead of waiting for the actual result of the branch, processor can do it speculatively by taking a look at history of prior branches that are already present through the Branch History Update (BHU) commands. The processor will be able to predict the instruction sequence by history of the same branch and history of other branches that have already executed before the current branch.
- *The Branch Target Address (BTA)*. When a branch is calculated to be taken, the next required information is the address of the target branch. Similar to branch result, this address may not be available as the execution. Again the processor will keep track of history of target addresses of recent branches by storing them into *Branch Target Buffer (BTB)*.

The relation of correct fetched instructions by execution advancement can be seen in the below picture:

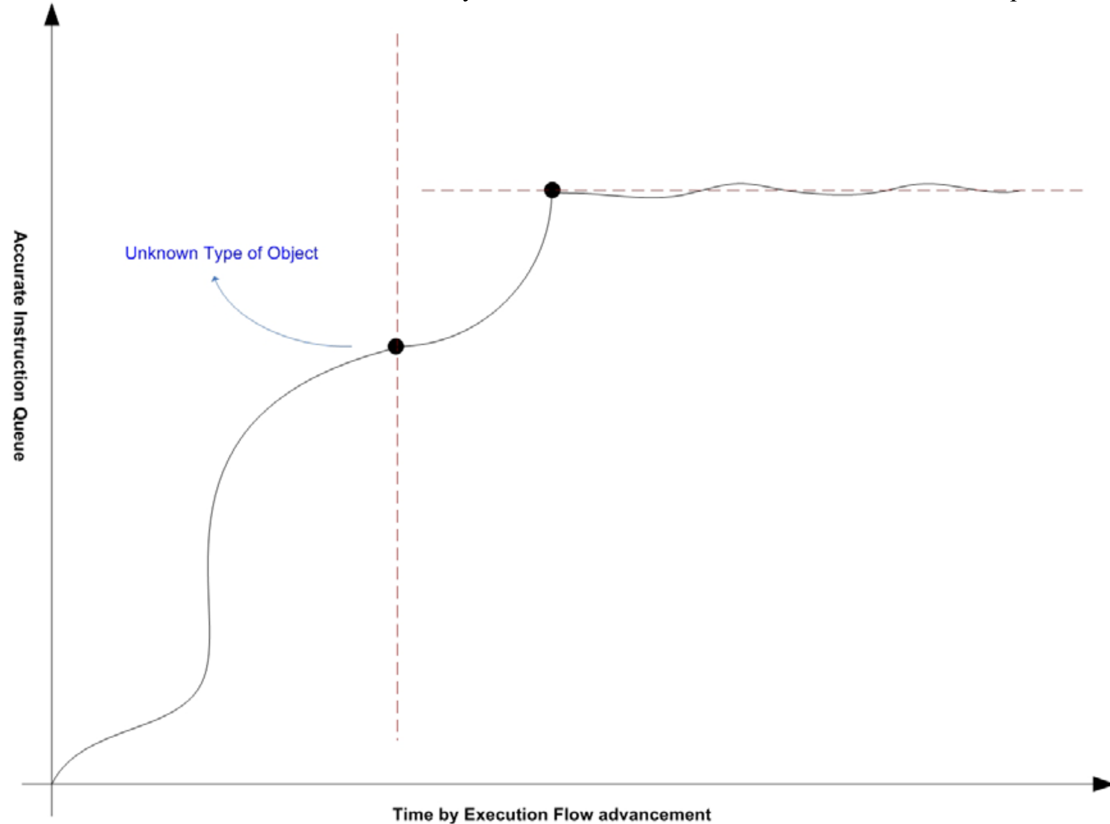

Figure 2. Relation between Accuracy and Time influenced by unknown type of object (unknown byte lengths)

By the way, the general lookout of a regular BPU is as below:

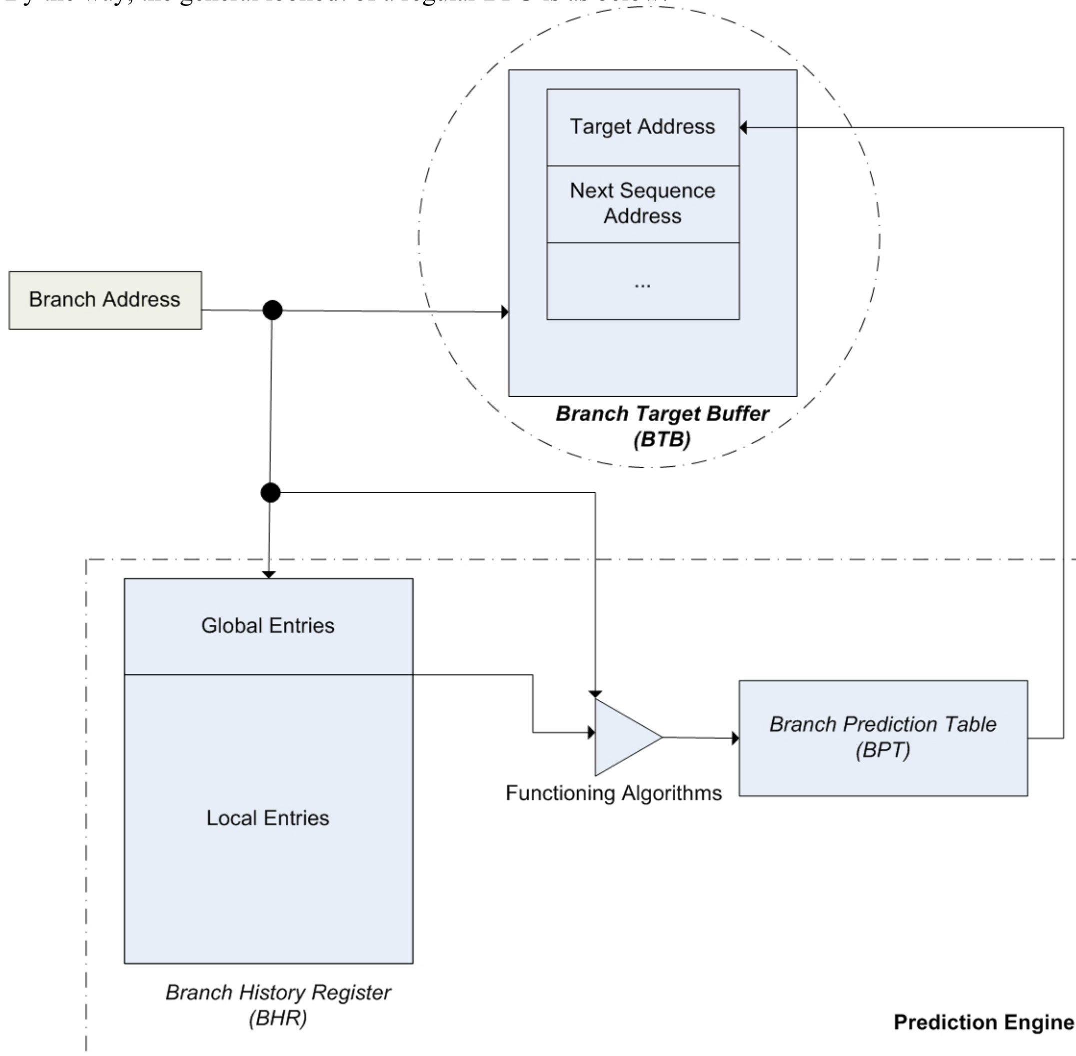

Figure 3. Branch Prediction Unit (BPU) in Basic shape

As you can see, the BPU consists of two logical portions, the BTB and the Prediction Engine. As we said already in above lines, the BTB is a buffer that target addresses of recent former branches is stored in. Obviously the size of BTB is limited. As the new addresses get added in, the older ones will be eliminated. In modern processors the size of BTB is waving around 256 to 70656 Entries.

The prediction engine is the actual identity that performs prediction based on the result of a branch (taken or not). In much more complex branch prediction other units are added in the BPU, too.

Prediction Methods

Branch Prediction can be done in either static or dynamic ways. The static way is almost discussed here; the prediction is done by copping with the results of recent branches and their addresses in BTB. So the BTB that is sometimes referred to as Branch History Table (BHT) is the key factor in Static Prediction, because all of decisions are based on it.

Dynamic Prediction is doing in another way. In this method the target address is reached by more complex mathematic operations in different methodologies, including Branch Target Buffer (BTB), Two-Level Prediction, Indexed Prediction, G-Share Prediction, Hybrid Prediction, etc.

Static Prediction using Branch Target Buffer (BTB)

Branch Target Buffer is mainly used in indirect branching. By growing the Object-oriented programming models, indirect branches are getting more and more important on branch prediction. An *indirect branch* is in fact a branch taking control to an address stored in a register (mode of indirect addressing). Their prediction accuracy is usually lower than normal conditional branches. This is because unlike conditional branches that only had "taken" or "not-taken" (can be specified by a single bit) as the results of a branch, a full 32-bit or 64-bit address is required. Although indirect branching is less frequent in applications, but they occur much in OOP model with the accuracy of %64.

There are usually three limits on indirect prediction:

1. *When a program behaves in a very unpredictable way*. In such states the only adequate prediction scheme is the *simulation* of program itself.
2. *When the predictor is not aware of some portions of program's behavior.* Consider a situation in which an OO program is going to call a routine (method) on every element of a polymorphic

collection of objects; as the types the objects in aforementioned collection is invisible to a control branch predictor, this will result in an unpredictable target sequence of instructions.

3. *When the predictor can access the whole elements changing the program's behavior but still is not able to collect the regularity correctly.* An example is when a branch predictor has small tables to fit more information into.

With all the limitations around, the BTB mechanism is still useful. It can be implemented in two ways:

- *Standard BTB (**BTB**).* Updating of target address is done after each branch execution.
- *Two-Bit Counter BTB (**BTB2bc**).* Updating of target address is done after two consequent mispredictions. Many of variations of BTB implementation use BTB2bc variation in different ways. Two points usually specify the variation:

1. What identity does comprise the history pattern?
2. How is the history pattern mapped to a target address?

Enhanced Branch Target Buffer (EBTB)

At Intel Pentium 4+ architectures a new technology was introduced as *Advanced Dynamic Execution Engine (ADE)*. The Pentium 4's branch prediction improves the prediction efficiency by 33% in comparison to Pentium 3.

First improvement was BTB size. The Pentium 4 architecture dedicates a 4KB sized BTB. However when an incorrect prediction is made, the whole pipeline should be flushed and new instructions replaced with. As we said earlier as much as pipeline is deep, the execution should be get back the same in path to find the correct branch path. To reduce the time required for the processor to fetch the instructions from the correct path, a new element is introduced by the name of *Trace Cache*.

Trace Cache is simply a cache of pre-fetched instructions located between decoders and execution units. The size of trace cache ranges from 12 KB to 40 KB. It stores the micro-operations of *already* decoded instructions. Mathematical calculations make trace cache to store special instructions, like the ones in a loop. This will remove the decoder from the main execution loop which helps reduce the stall time of the execution engine.

Dynamic Prediction using Branch History Buffer (BHB)

Using static prediction has two downsides: first, it doesn't do well in sophisticated applications. Second, it is all based on history entries. There is not any effort to calculate the correct target address.

Dynamic prediction is supposed to help speed up former structure of static prediction. To gain this goal, we create a buffer named Branch History Buffer (BHB) (a.k.a. BHT- Branch History Table) that indexes the low-order address bits of the recent branch instructions. In this structure a bit is stored for each instruction, indicating whether the branch was recently taken. This is shown in below picture:

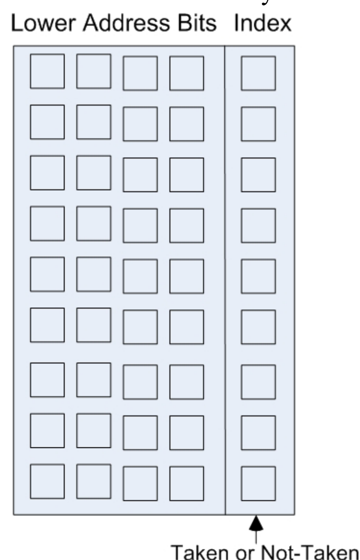

Figure 4. Indexing Taken, Not-Taken branches

The predictor will check the BHB for every branch instruction for the indexed values. If the stored prediction bit indicates the branch should be taken (i.e. it is indexed as 1 for True or Taken in BHB), then as the computation of target address the pipeline can start off fetching instruction from that new address. If the BHB was wrong and a misprediction occurred, then the whole instructions should be flushed out of the pipeline and the prediction bit in BHB should be *inverted*.

In some cases another more bit is added for the sake of reliability, it is shown in below picture too:

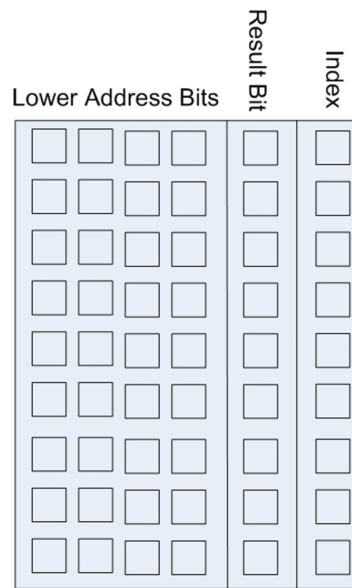

Figure 5. Indexing Taken and Not-Taken branches (plus the last comparison result)

Another dilemma is again to be faced at loops. Consider a loop being traversed in BHB; a single bit in BHB will be incorrect twice, once on the first pass of the loop (entering loop) and the other one at the end of the loop.

The BHB can be extended further to gain more accuracy and supporting more possibility space, and eventually overcome this situation. This is done by using *more bits* a indexes. It will make a counter that increments on a taken branch and decrements on an untaken branch, thus avoids aforementioned loop controversy. A simple math shows that for a table with 4 KB (4096 number) entries, a *2-bit* indicator performs about as well as even more bits, leading us to the accuracy of 80% to 99%. The size of this table is about to decreasing return for 2 bit entries, thus there is no point and benefit to extend table size because it still leads us to the same result.

Another dilemma here! The problem is on 2-bit prediction and the indexed values. Since we are indexing lower address bits, then two different branch addresses may have the same low order bits, pointing to the same record in the table.

In this way we can even extend our BHB further by correlating the behavior of branches. In the extension we allow other branches to update prediction bits of a particular branch entry in BHB. In fact *Global History Counter (GHC)* is one the schemes doing that, and one of the first approaches was *GShare* Algorithm.

G-Share uses a register to store generic result of recent branches that is named *Global Branch History Register (GBHR)*. To reach this goal, the GBHR will be hashed with bits of being predicted branch's address, and the result will act as an index in BHB, in which the entry in that location is used to predict the branch direction dynamically, as shown in below picture:

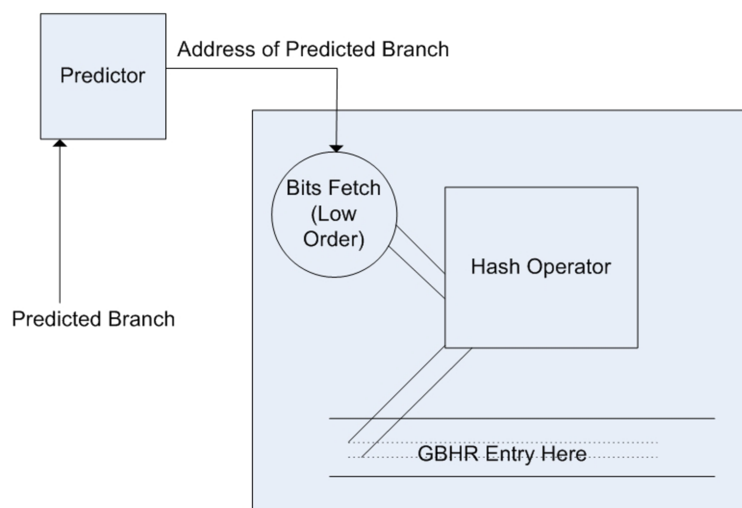

Figure 6. Global Branch History Register in G-Share algorithm

Methodologies

Different methodologies (or a combination of) can be used in BPU in different modes (static or dynamic) to improve performance affected by branch instructions.

Method #1. Single Way Prediction

The first and the most straightforward method is the *single-way prediction* in which all of branches will be predicted either as always taken or always not-taken. Because unconditional instructions are usually used in loops, it can be concluded that most unconditional branches should be always taken and the branch instruction found at the end of the loops point to the top of the loop, meaning it should be taken too. It brings the hit rate of branches to 50%. So the fist method is to predict all branches as 'taken'.

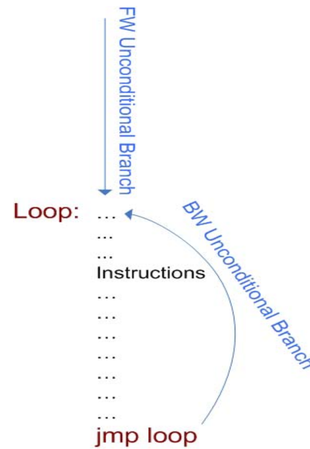

Figure 7. A sample of Single-Way Prediction

Method #2. OPCode-based Prediction (static)

The operation code matters in this method. If a specific operation code(s) does exist in a branch, it will be taken, and the other ones will be known as not-taken.

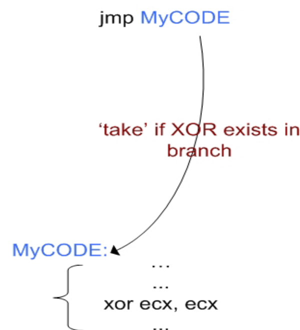

Figure 8. A sample of Operation code sensitivity in prediction direction

The *Frequency* of OPCodes in a branch block can also be a determining factor in this method.

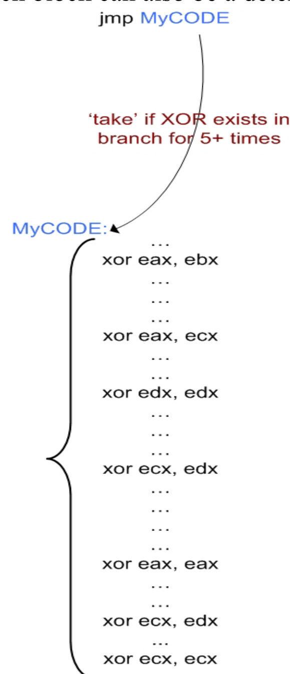

Figure 9. A sample of Frequency of Operation code sensitivity in prediction direction

Method #3. Old Line Prediction

In this method the branch will be treated the way it has been treated in previous trigger. If it has not been executed in last one, then it will be predicted as 'taken'.

This method statistically provides more prediction accuracy than the single way method. Implementing this method is physically difficult because there is not limit on the number of branch predictions (identically separated) that a program may contain. In fact Global History Counter and the subsequent outcomes are destined to do something in this way; although the whole of branch instructions can't be counted, a small history of recent branches can be recorded.

Method #4. Backward Sensitive Prediction

All backward branches will be taken, and the whole forward branches will be known as not-taken.

The story to develop this prediction method is again laid on loops. Loops are usually terminated with backward branch instructions; if they all get predicted correctly, then the accuracy will be much more improved. Generally it works better than the former approaches, however the main concern in this method is backward branches, so the programmers should write their programs compatible with this method to get the most out by. This significant attach to program literally decreases the performance on some programs. An addition performance downside is that in some cases the target address should be first calculated or compared with the program-counter before going forth for prediction.

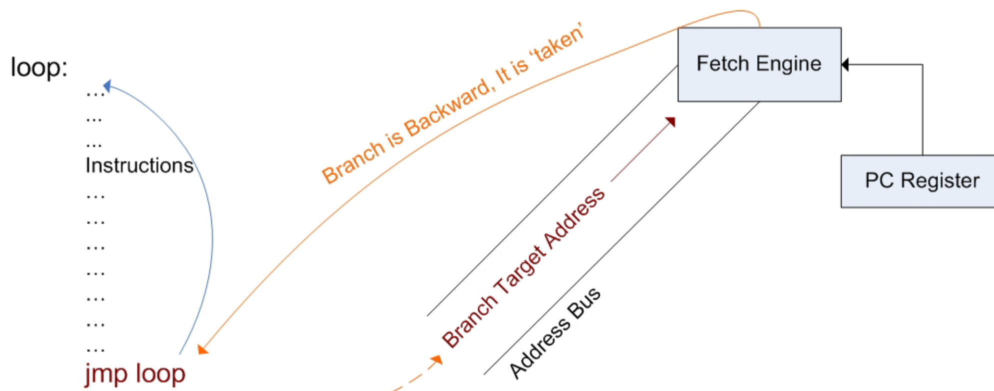

Figure 10. A sample of Backward 'taken' prediction

Method #5. Not-Taken Buffer Prediction (dynamic)

In this approach a buffer of recent *not-taken* branch instructions is maintained. If a branch instruction exists in the buffer, then it will be predicted as 'not-taken' (as is); otherwise it'll be predicted as 'taken', aftermath the taken entries are purged from the buffer and new entries are added.

This method calls another variation too, in which the state of word's sequence is determined. Before being fetch, any instruction word is compared with a memory slice (keeping track of sequencing) to see whether the word has been in-sequence or out-of-sequence in the last access. If the word was out-of-order (sequence) then that memory slice provides the address of this out-of-order word, and the fetching begins from that location. This is not an effective way because it is done before instruction decoding; but the Out of Order (OOO) engine adapts a similar method to execute instructions out of order (of course in OOO-enabled processors, there are Trace Caches and it eliminates the aforementioned inefficiency problem).

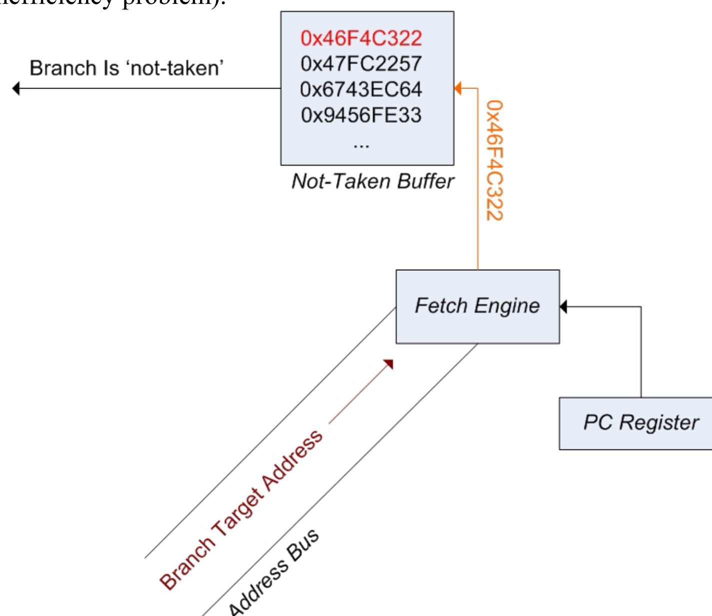

Figure 11. A demonstration of Not-Taken Buffer usage in branch instructions

Method #6. Index bit-enabled Prediction

In such methods a bit is dedicated to each instruction in the cache, determining whether it is taken or not. If the branch is selected to be taken, then the result will be the same as the last trigger (old line prediction method). And if the branch has not been executed, it will be selected as 'taken'.

Schemes like BTB and BHB use a similar prediction mechanism.

Note that in the methods above, the memory that comparison is done with, is an associative memory slice. In some advanced revisions the comparison is done with RAM.

Method #7. BTA (Branch Target Address) Hashing

In this method the branch target address is hashed, then it is used as an index to address a RAM portion that contains the result (outcome) of the most recent branch (that indexes into the same location). If the result indexes into the same location, then the branch outcome is predicted as the same.

This approach significantly depends on results. But the default prediction in which it firstly gets accessed can be controlled by filling memory with 0 or 1. For the sake of speed, the hashing pattern is usually XOR (it operates better than any other hash pattern, both in speed and byte length).

Other variations of this method use two's complement words instead of a single bit (if the sign bit is 0, the branch is taken). When two locations return the same hash value, this approaches helps out.

Stack Return Buffer (SRB)

This proposal is in fact a major change in the plan discussed by D.E. Waldecker in late 80's. In waldecker's plan a push-down memory is designed so that it the information held previously in the stack will be pushed down the stack automatically.

When facing to a branch instruction, the addresses found in branch target address plus one will be stored in the first cell of the push-down memory. In this situation the current value of program-counter plus one will be pushed-down. Eventually a return instruction will cause the return address to be pushed up to the first cell of memory.

As it is clearly shown in waldecker's plan here, each access to the memory will require a bunch of time which leads us to access slowing.

Our plan here is another architectural change in data processors. It consists of several control registers, like fetch and execution registers. The architecture also has an internal memory stack, an address register and an enumeration register placed between the actual memory stack and control registers of processor.

The enumeration register is supposed to store the latest added entry into the memory stack (the top of the stack). This way the latest entry of memory stack is quickly available to control registers, leading to reducing memory access time. The general lookout of this plan is shown below:

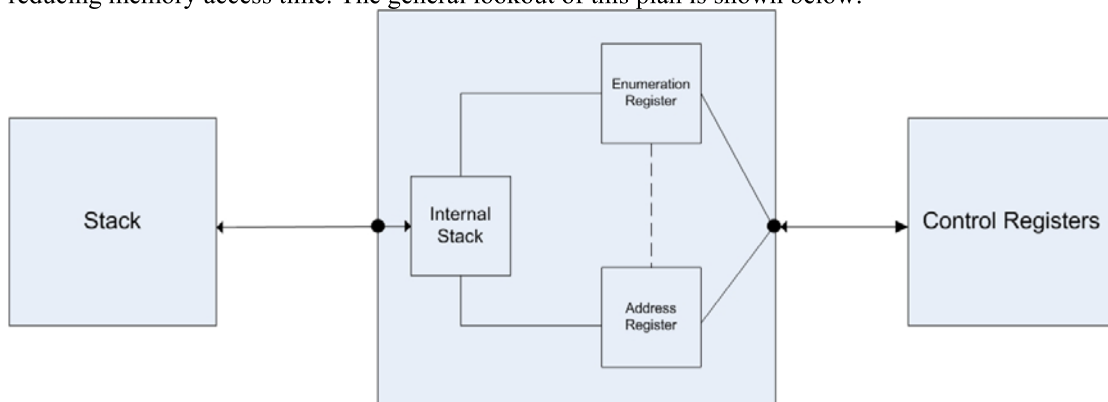

Figure 12. Stack Return Buffer (SRB) places between Memory Stack an Processor Control Registers

The plan is shown in great details below:

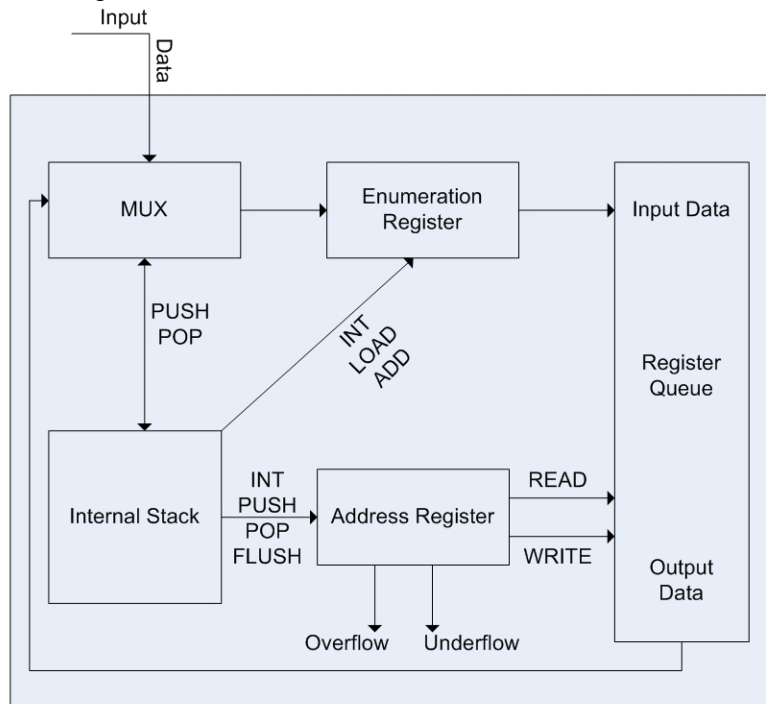

Figure 13. Stack Return Buffer (SRB) in details

The MUX unit or multiplexer is the unit that makes the direction of signals toward the board. The enumeration register is associated with some units and control registers. It acts as storage to point to the latest entry added into the stack.

The internal stack separates different data required for instructions, then dispatches them to corresponding units. They go for ALU for arithmetic operations or Memory units for Read/Write operations.

The address register is correlated with an internal execution register that executes each microinstruction passed decode and multiplexing steps.

The control register actually stores an address. In order for executing microinstructions, the address hold by control register is provided to access the recent microinstruction to be executed, or to access data referred by Register Queue.

References

- D. Grawrock. *The Intel Safer Computing Initiative: Building Blocks for Trusted Computing*, Intel Press, 2006.
- M. Joye and S.-M. Yen. The Montgomery powering ladder. *Cryptographic Hardware and Embedded Systems - CHES 2002*, Springer-Verlag, 2003.
- D.E. Waldecker, *Controller for Data Processing System*, 1990
- *Intel Software Developer's Guide, Vol1*, Intel Press, 1990
- *Intel Software Developer's Guide*, Vol3, Intel Press, 1990