Demystifying Intel Branch Predictors

Milena Milenkovic, Aleksandar Milenkovic, Jeffrey Kulick
Electrical and Computer Engineering Department, University of Alabama in Huntsville
Email: {milenkm, milenka, kulick}@ece.uah.edu

Abstract

Improvement of branch predictors has been one of the focal points of computer architecture research during the last decade, ranging from two-level predictors to complex hybrid mechanisms. Most research efforts try to use real, already implemented, branch predictor sizes and organizations for comparison and evaluation.

Yet, little is known about exact predictor implementation in Intel processors, apart from a few hints in the Intel manuals and valuable but unverified hacker efforts. Intel processors include performance monitoring counters that can count the events related to branches, and Intel provides a powerful VTune Performance Analyzer tool enabling easy access to performance counters. In this paper, we propose a series of experiments that explore the organization and size of a branch predictor, and use it to investigate Pentium III and Pentium 4 predictor implementations. Such knowledge could be used in further predictor research, as well as in the design of new, architecture-aware compilers.

1. Introduction

Conditional branches are one of the major barriers to successful program parallelization: when a conditional branch enters the execution pipeline, all instructions following the branch must wait for the branch resolution. A common solution to this problem is the speculative execution: branch outcome and/or its target are dynamically or statically predicted, so the execution can go on without stalls. If the prediction was incorrect, speculatively executed instructions must be flushed and their results discarded, which could produce a significant number of lost execution cycles. While static prediction can work well for some benchmarks, dynamic prediction solves the more general case. Hence, methods for better prediction are continuously investigated and improved, and in the last decade dynamic branch prediction schemes have been one of the focal points of computer architecture research, from two-level predictors to complex hybrid schemes.

We know that modern commercial processors, such as Intel Pentium III (P6 architecture) and Pentium 4 (NetBurst architecture) include some form of dynamic branch prediction mechanisms, but detailed information is rather scarce. On the other hand, these architectures include performance monitoring registers that can count several branch-related events, and Intel provides a quite powerful tool for easy access to these registers, the VTune Performance Analyzer [1].

The purpose of this research is to devise a set of experiments that will explore some relevant parameters of the branch predictor structure, and to test it on P6 and NetBurst architectures. These parameters can be used for code optimization and as a starting point for comparison in future predictor research. The experiments have also educational value, providing better understanding of branch predictor mechanisms.

The experiments are based on the values of performance counters during the execution of "spy" microbenchmarks, designed to test the existence and/or value of a particular branch predictor parameter. While microbenchmarks have been previously used to determine the characteristics of memory hierarchy [2] [3], such as cache line size, cache size, associativity, etc., we are not aware of similar efforts aimed to explore branch predictor structures.

This paper is organized into seven sections. The second section gives a short overview of dynamic branch prediction schemes, and the third section defines the problem statement. The fourth section describes the experimental environment. The fifth section explains the proposed experiments, and the sixth section presents and discusses the results. The last section gives concluding remarks and indicates possible directions for further research.

2. Dynamic Branch Prediction

No matter how complex a branch predictor is, it could be described by some variation of the general scheme (Figure 1), consisting of two major parts: branch target buffer (BTB), and outcome predictor, for prediction of branch targets and branch outcomes, respectively.

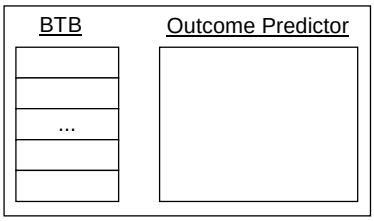

Figure 1 General Branch Predictor Scheme.

Dynamic prediction of a branch outcome is based on the state of a finite-state machine, which is usually a two-bit saturating counter [4]. This counter is a cell of a branch prediction table (BPT), which could be accessed in different ways. The simplest BPT index is a portion of the branch address. More complex two-level predictors combine the branch address or its part with shift register representing the history of branch outcomes [5][6][7]. Global two-level predictors benefit from correlation between subsequent branches during program execution (global branch history), while local predictors are based on correlation between subsequent executions of the same branch (local branch history). Hybrid branch predictors can include both global and local prediction mechanisms, as well as some other prediction schemes, such as specialized loop predictors, or simple BPT [8]. Some hybrid predictors have parallel organization, where different predictor components predict branch outcome in parallel, and a selection mechanism, a predictor itself, decides which outcome to choose. Other hybrid predictors have serial (cascaded) organization, where the output of one predictor stage (predicted outcome) is part of the input to the other predictor stage [9][10]. Higher predictor stages are used only when lower stages are not able to predict the branch outcome correctly. Serial and parallel predictor organization can also be combined [11]. A very interesting solution accesses branch predictor table using "alloyed" global and local branch history as part of the BPT index [12], thus predicting correctly branches that depend both on global and local history. It is also possible to add a third adaptivity level to a predictor [13], dynamically determining the optimum history length.

Instead of exploiting correlation between outcomes of last *h* branches (pattern-based), dynamic branch predictor can use the information of the path to the current branch (path-based) [14]. Path history register stores address bits from each of last executed *b* branches, thus making the prediction path-dependant. One predictor can combine both pattern-based and path-based approaches.

Prediction of branch outcome could be coupled or decoupled with a BTB. The BTB can hold one or more possible target addresses, even target instructions.

Since every branch prediction table is of a finite size, different branches will use the same cell. This effect is called interference or aliasing [15], and lot of research has been dedicated to the interference problem [16][17][18].

Some special branch types, such as returns and loops, could be handled by specialized predictors.

3. Problem Statement

For both P6 and NetBurst architectures, Intel sources [19], [20], [21] do not provide the exact description of the implemented branch predictors. Rather, they provide the exact number of BTB entries and several hints about program optimization that indicate some outcome predictor parameters. They state that the static prediction mechanism predicts backward conditional branches as taken, and forward branches as not taken. In the P6 architecture, "prediction algorithm includes pattern matching and can track up to the last four branch directions per branch address," [20], which most probably means that the P6 branch predictor has a local history component with 4 history bits. The P6 BTB has 512 entries.

In the NetBurst architecture implemented in Pentium 4, Intel claims to use some new prediction algorithm, 33% better than in P6. One of the assembly/compiler coding rules for Pentium 4 states that frequently executed loops with predictable number of iterations should be unrolled to reduce the number of iterations to 16 or fewer, and if the loop has *n* conditional branches, it should be unrolled so that the number of iterations is *16/n* [19]. This rule indicates that Pentium 4 uses global outcome history, with probably 16 history bits, but the Intel sources never specifically say so.

Another interesting characteristic of the NetBurst architecture, tightly coupled with the branch prediction mechanism, is an execution trace cache [21], which stores and delivers sequences of traces, built from decoded instructions following the execution flow. Intel sources explain that the trace cache and front-end translation engine have cooperating branch prediction hardware, so branch targets can be fetched from the trace cache, or in the case of trace cache miss, from the second level cache or memory. The trace cache BTB is smaller (512 entries) compared to the front-end BTB (4K entries). It seems that both the trace cache and front-end share the same outcome predictor mechanism [20], but apart from trace cache size (12K micro-ops), and cache line size (6 micro-ops), Intel does not disclose too many details about its implementation. For example, one interesting question is whether just the most likely branch path is stored in the trace cache, or can it store more paths. More stored branch paths would reduce the number of lost cycles in the case of misprediction, since the correct instructions could be fetched from the trace cache instead from higher levels of memory hierarchy.

Since the exact predictor parameters are important to software developers, some hacker efforts have been dedicated to this problem. In his Pentium optimization manual [22] A. Fog gives a short description of prediction mechanisms in Pentiums, up to the Pentium III. His findings include 4 local history bits for P6 architecture, and a 512-entry BTB organized as 16 ways * 32 sets, where bits 4-8 define the set. Unfortunately, he did not present any details about the nature of his experiments, so one of our goals is to verify his results.

The goal of this research is to determine the branch predictor parameters most important for code optimization, by treating branch predictor structure as a black box and using a set of carefully designed

microbenchmarks. In this paper, we restricted our efforts to the following parameters:

1. Branch Target Buffer component

- Size and organization.

2. Outcome predictor component

- The existence of a local prediction component, and the number of local history bits in history register;
- The existence of a global prediction component, and the number of global history bits in history register.

Considering the specifics of the NetBurst architecture and importance of the trace cache in the branch prediction mechanism, we also want to verify whether the trace cache is able to store both taken and not taken branch path.

Once determined, these characteristics could help the code optimization. For example, the size and organization of BTB indicate how many branches can fit into it in the critical portion of the code, and the number of local/global history bits indicates the maximum branch correlation that a given predictor can recognize.

We are aware that more predictor parameters are needed for better code optimization. Other relevant BTB parameters are the number of branch targets that could be stored per branch, BTB replacement policy, and address bits used for BTB tag. More complex hybrid predictor organizations, with several components, require further experiments to determine the exact component layout and a way to decide between predictions of different components. Due to out-of-order instruction execution, it is not easy to establish whether predictor mechanism is speculatively updated, or only after branch resolution. The replacement policy, trace length, and the possibility of speculative trace constructing are some of the trace cache parameters that are out of the scope of this paper. Design of microbenchmarks that would determine some of these parameters is part of the ongoing research.

Both P6 and NetBurst architectures use return address stack to predict return addresses, and the size of this stack is known, so we do not consider any experiments related to function returns.

Finally, in the case of Intel predictors we were able to assume some of the predictor characteristics, making the black box testing more transparent. In the general case, given a completely unknown predictor mechanism, more microbenchmarks must be used to determine its nature and parameters. For example, the outcome predictor does not have to be coupled with BTB, so it could predict all branches, and not just those stored in BTB. Loops could be predicted by a dedicated predictor component, and some predictor components can be path-based instead of pattern-based. All predictor components could use some of the mechanisms aimed to reduce branch interference. More general framework that would describe experiments for testing a completely black box predictor will be considered in future research.

4. Experimental Environment

Both P6 and NetBurst architectures have several performance counters, and several branch-related events can be measured. In this research, we consider the number of retired branches, including unconditional branches, and the number of mispredicted branches, using event-based sampling. In some NetBurst experiments, we also consider the number of execution cycles versus the number of cycles processor spent in trace cache delivery mode.

Although event-based sampling is not precise, it gives a good estimation of the number of events. A performance counter is configured to count one or more types of events and to generate an interrupt when it overflows. The counter is preset to a modulus value that will cause the counter to overflow after a specific number of events have been counted. When the counter overflows, the processor generates a performance monitoring interrupt, and the corresponding interrupt service routine then records the return instruction pointer (RIP), and restarts the counter. Code performance can be analyzed by examining the distribution of RIPs with a tool like the Intel VTune Performance Analyzer. In this research we used VTune version 5.0.

All test benchmarks are compiled using Microsoft Visual Studio 6.0 C compiler, with disabled optimization, so we are certain that the compiler optimizations do not change the order and number of conditional branches. For experiments with relatively large number of branches, we have also developed programs to generate benchmarks to our specification.

In order to get reliable values of performance counters, the execution time of the monitored code must be significantly larger than the execution of interrupt service routine. Therefore, all microbenchmark code is placed within a loop executing a relatively large number of times.

5. Experiments and Spy Microbenchmarks

We perform two sets of experiments, one for the P6 and another for the NetBurst architecture.

5.1. Experiments for P6 architecture

Experiments for the P6 architecture consist of two sets, one aimed at exploring size and organization of the BTB component, and another exploring the parameters of the outcome predictor component.

5.1.1. BTB component

The Intel documentation provides the size of BTB, i.e. N_{BTB} entries [19], [20], but does not describe its organization - whether it is direct-mapped, 2-way, 4-way, etc. We perform the experiment with $N_{BTB} - 1$ conditional

branches in a loop, which makes a total of N_{BTB} conditional branches in the code. The conditional branches in the loop are always taken, so they will be mispredicted by static algorithm if they are not present in the BTB. We vary the distance between the branches (fixed for one experiment), so the *DM_Index_T* bits that differ one branch address from another are in different position for different distances (Figure 2). Figure 3 shows the fragment of the code used for testing BTB organization.

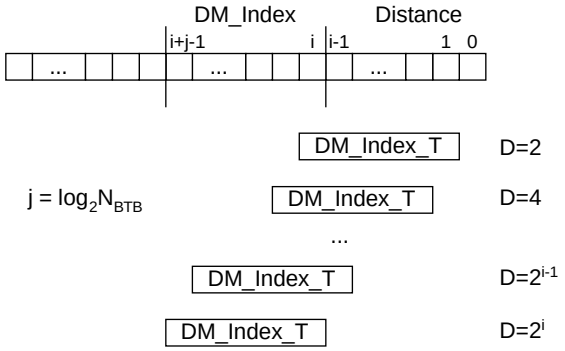

Figure 2 BTB size and organization: varying the distance.

```
...
for (i=0; i < liter; i++) {
 _asm {
              noop
              ...
              noop
              mov eax, 10
              cmp eax, 15
              jle l0
              noop
              ...
              noop
       l0:    jle l1
              noop
              ...
              noop
       l1:    jle l2
              ...

       l510:  noop
       }
}
```

multiple of distance *D*

distance *D*

distance *D*

Figure 3 Benchmark for testing BTB organization.

This experiment discovers the values of "fitting" distances D_F, when all considered branches at distance D_F can fit in the BTB. Hence, for a distance D_F the number of mispredictions (MPR) will be close to zero, i.e. the performance counter should count only the negligible number of mispredictions.

If there is only distance D_F, then we can conclude that the BTB is direct-mapped. Bits used to address the BTB are Addr[i+j-1 : i] (Figure 2, *DM_Index* bits). From the distance D_F and the number of BTB entries we can determine exactly which address bits are used to address the BTB.

If there are exactly two distances D_F, we conclude that we have the 2-way set-associative organization of the BTB. Bits used to address the BTB are Addr[i+j-2 : i]. Similarly, if there are exactly three distances D_F, the BTB is 4-way set-associative. In general, if there are *m* "fitting" distances, the BTB is 2^{m-1}-way set associative. Bits used to address the BTB are Addr[i+j-m : i].

Now we can verify the assumption about the number of BTB entries by repeating the experiments for the "fitting" distances and larger number of branches. For example, if the real number of BTB entries is twice as large as the assumed one, and our experiments have found *m* distances D_F, the set of experiments with the real number of entries should find *m-1* such distances, i.e., the BTB would be 2^{m-2}-way set associative. In the general case, if the real number of BTB entries is 2^n times greater than the assumed one, the experiments should find *m-n* "fitting" distances. If the experiments with larger number of conditional branches do not find any such distance, our assumption about the size is correct.

The number of ways can be also verified by trying to find a number of branches to fill a set, and to find a distance such that those branches will map into the same set, conditions necessary to increase the number of mispredictions.

5.1.2. Outcome predictor component

The set of experiments for exploring the characteristics of outcome predictor component is devised in such a way that most of the branches in the code are easily predictable, so we can concentrate on one conditional branch and its MPR, i.e., the MPR of whole program is generated by that branch. We call this branch a "spy" branch. Figure 4 explains the required experiment flow, step by step.

In the Step 1, we try to determine the maximum length of a local history pattern that our predictor can correctly predict, for just one branch in the loop, i.e., the "spy" branch. The loop condition will have just one different outcome, on the exit, which is negligible compared to the number of iterations. Different repeating local history patterns can be used for this experiment; however, the simplest pattern has all outcomes the same but the last one. If "1" means that the branch is taken, and "0" not taken, such local history patterns are 1111...110 and 0000...001.

Figure 5 shows the code for one such pattern of length 4, while Figure 6 shows the fragment of the corresponding assembly code. Note that the "spy" branch *if ((i%4)==0)* is compiled as *jne*, so the local history pattern for this branch is 1110 (pattern length is four). The fragment does not show the loop, which is compiled as the combination of instructions *jae* at the beginning of the

loop and unconditional *jmp* at the end, so the *jae* outcome is 0 until the loop exit.

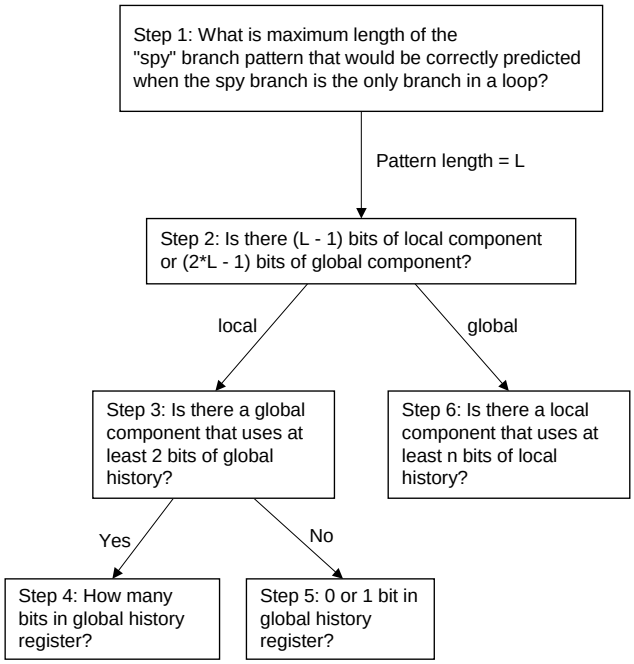

Figure 4 Experiment flow for outcome predictor.

```
void main(void) {
   int long unsigned i;
   int a=1;
   int long unsigned liter = 10000000;
   for (i=0; i<liter; ++i){
     if ((i%4) ==0) a=0; //spy branch
   }
}
```

Figure 5 Microbenchmark for Step 1 experiment.

```
; Line 6
  0002e      mov     eax,DWORD PTR _i$[ebp]
  00031      xor     edx, edx
  00033      mov     ecx, 4
  00038      div     ecx
  0003a      test    edx, edx
  0003c      jne     SHORT $L38
  0003e      mov     DWORD PTR _a$[ebp], 0
$L38:
```

Figure 6 Fragment of the assembly code for Figure 5 code.

We should get low MPR for all pattern lengths up to a certain number *L*, and then the outcome predictor will not be able to predict the last outcome of "spy" branch. That is, for each pattern of length *l*, *l>L*, the "spy" branch will be mispredicted once in *l* times.

However, from this experiment we still can not conclude whether predictor has a local prediction component with *lh=L-1* history register, or a global predictor component with *gh=2*(L-1)* history register. We must consider two cases:

(a) The outcome predictor has local history component, so the pattern 11...10 with *L-1* 1's and one 0 is correctly predicted; when the pattern is 111...1 (*L-1* 1's), the next outcome is predicted to be 0, for 11...10 (*L-2* 1's), the next outcome is predicted to be 1, etc.

(b) The outcome predictor has global history component, so the local history pattern 11...10 of the "spy" branch with *L-1* 1's was correctly predicted, but by using global history of previous *2*(L-1)* branches. Since we have just the loop and "spy" branch, there will be no mispredictions if all relevant local history fits into global history register. For example, just before executing the "spy" branch with 0 outcome, the content of the global history register is **1**0**1**0**1**0...**1**0, where underlined and bolded 1's are outcomes of the "spy" branch, and 0's are the outcomes of the loop condition branch.

In Step 2, we verify which hypothesis matches our predictor. If the conditional branch in the loop is preceded by *2*(L-1)* "dummy" conditional branches, having always the same outcome, we can be certain that no local "spy" history enters global history register. One example for the "dummy" branch is *if (i<0) a=1*. If in this experiment the MPR still stays low, the correct hypothesis is (a), a local history component, since the "spy" outcomes are still correctly predicted. We should proceed to the Step 3, to determine whether the outcome predictor also has a global history component.

If the MPR increases (last conditional branch is mispredicted once in *L* times), we conclude that the correct hypothesis is (b), a global history component. We could get the same result with insertion of just one "dummy" branch, but we wanted to be sure that there is no local history in the global history register. In this case, we should proceed to the Step 6, to determine whether the outcome predictor also has a local history component.

```
void main(void)
{ int a,b,c;
  int long unsigned i;

  for (i=1;i<=10000000;++i)
  { if ((i%2) == 0) a=1;
    else a=0;
    if ((i%5) == 0) b=1;
    else b=0;
    if ( (a*b) == 1) c=1;
  }
}
```

Figure 7 Microbenchmark for Step 3 experiment.

The Step 3 microbenchmark has three conditional branches in a loop, where first two have predictable patterns 11...10 of different length *l1* and *l2*, such that *l1*, *l2* < *L*, and the smallest common denominator for (*l1*, *l2*) is greater then *L*. The third branch will be correlated with the first two, by having a pattern 11...10 of length greater

than *L*, so it cannot be predicted by local component. It will be not taken when both previous branches are not taken (Figure 7).

If this experiment still has a low MPR, the predictor also has a global component with at least two global history bits. The next step, Step 4, verifies the length of the global history register. The simplest way is to insert "dummy" conditional branches (pattern 111...11) before the "spy" conditional branch. By varying the number of "dummy" branches, we will get the number of global history bits, since the "spy" branch will not be predicted correctly if the number of "dummy" branches is greater than the number of *global history bits – 2*, and will always be predicted correctly for the smaller number of "dummy" branches.

```
void main(void)
{ int a,b,c;
  int long unsigned i;

  for (i=1;i<=10000000;++i)
  { if ((i%2) == 0) a=1;
    else a=0;
    if ((i%5) == 0) b=1;
    else b=0;
    if (i<0) a=1; //dummy branch
    ...
    if (i<0) a=1; //dummy branch
    if ( (a*b) == 1) c=1;
  }
}
```

Figure 8 Microbenchmark for Step 4 experiment.

If the 0 "spy" outcome is mispredicted in the Step 3 experiment, it means that there is no global component or there is just one bit of global history. The Step 5 microbenchmark has just two conditional branches in the loop, where the first one has local history pattern 111...110 of a length *l>L*, and the second one has the same outcome as the first. Since from the Step 3 we know there is no more than one global history bit, the first conditional branch will always be mispredicted once in *l* times. If there is no global component at all, the second branch will be mispredicted the same number of times, while it should always be predicted correctly if there is one bit global history component. We could determine the existence of one bit global history predictor by examining the number of mispredictions in this experiment.

If we have a global component with *2*(L-1)* history bits, do we also have a local component? The Step 6 microbenchmark has *2*(L-1)* "dummy" branches (Figure 9), and varies the pattern length *l* of the "spy" branch. If the MPR is low for some *l*, there is an equivalent of local component with at least *l-1* history bits. Depending on the decision mechanism, there could be more local history bits, so further experiments might be needed. This is out of the scope of this paper.

```
void main(void) {
  int long unsigned i;
  int a=1;
  int long unsigned liter = 10000000;
  for (i=0; i<liter; ++i){
   if (i<0) a=1;//dummy branch 1
   ...
   if (i<0) a=1;//dummy branch 2*(L-1)
   if ((i%l) == 0) a=0; //spy branch
  }
}
```

Figure 9 Microbenchmark for Step 6 experiment.

5.2. Experiments for NetBurst architecture

Since the NetBurst architecture includes dual BTB structures, one for front-end processor part, BTB_{FE}, and another for the trace cache, BTB_{TC}, the experiments consist of three sets: one set aimed at exploring the size and organization of front-end BTB component, another to explore the parameters of the outcome predictor component, and the third one targeted at the trace cache BTB.

5.2.1. Front-end BTB component

We will first verify the size and organization of BTB_{FE}, using the same set of experiments as defined for the P6 BTB in section 5.2.1.

5.2.2. Outcome predictor

To verify the existence and length of global and local history registers, we will use the same set of experiments as defined for the P6 outcome predictor in section 5.2.2.

5.2.3. Trace cache BTB component

We first verify whether both taken and not taken paths are stored in the trace cache, if executed code includes both paths. This experiment uses similar microbenchmark to the one from Step 1 described in section 5.2.2, where a "spy" branch has both *if* and *else* paths. We compare the number of cycles spent in the trace cache delivery mode with the total number of cycles, when the number of mispredictions is low. If the trace cache delivers traces for most of time, it means that both taken and not taken paths are stored in the trace cache, and that the trace cache prediction mechanism points to the correct target.

Microbenchmark for the next set of experiments includes N "spy" branches in a loop, with the same behavior and both taken and not taken paths visited during execution. The "spy" branch behavior is set according to the results of section 5.2.2 experiments, so that the number of mispredictions should be relatively low with the given outcome predictor. For example, if the outcome predictor has a global component and no local component, the "spy" branch condition may be *if ((i%l) == 0)*, so the first "spy" branch will be mispredicted once in *l* times, and all other branches should be predicted correctly, since their outcomes are the same as the

outcome of the first branch and hence predicted by global component.

We will consider three different cases, where N conditional branches in each case fit into BTB_{FE}:

(a) Executed code does not fit into trace cache,

(b) Executed code fits into trace cache, but branches do not fit into BTB_{TC},

(c) Executed code fits into trace cache, and branches fit into BTB_{TC}.

In the case (a) the MPR is low, and trace cache spends relatively few cycles in the delivery mode.

Although in the case (b) all executed code can fit into the trace cache, some branches will be mispredicted due to the size of BTB_{TC}. Trace cache spends relatively few cycles in the delivery mode.

When all branches in the loop fit in the BTB_{TC}, MPR will be low, and the number of cycles spent in delivery mode will be close to the total number of cycles.

This set of experiments tries to establish the boundary values of N for all three cases.

Similarly to the BTB_{FE} experiment, we may try to establish the number of micro-operations between two conditional branches in the execution flow, thus determining the organization of BTB_{TC}.

6. Results

6.1. Results for P6 architecture

6.1.1. Branch Target Buffer component

We assume N_{BTB}=512. The MPR is close to 0%, when the distance between addresses of subsequent branches is 4, 8, or 16, and close to 100% for other distances (Figure 10). Since we have three different distances producing low MPR, this result means that P6 architecture has BTB organized in 4 ways, 128 sets.

From the result of this experiment, we are also able to determine that address bits 4-10 are used as the set index. Note that our experiments prove the Pentium III to have different organization than stated in A. Fog's optimization manual [22].

This result can be also obtained by trying to map N_{WAY} + 1 branches in the same set, varying the distance between them and the number of branches (Table 1).

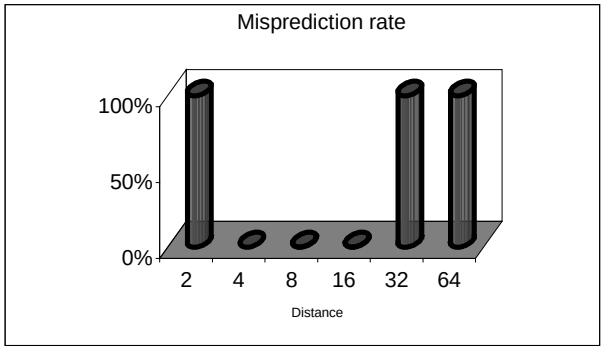

Figure 10 Misprediction rate for N_{BTB} conditional branches, varying the distance.

Table 1 P6 branches retired and mispredicted when N_{WAY} + 1 branches map in the same set.

Iterations: 10M, NoBranches: 17		
Distance	**Branches retired**	**Mispredicted**
256	168,676,020	2,860
512	175,629,710	39,113,116
1024	179,564,544	159,796,902
2048	178,080,370	157,893,705
Iterations: 10M, NoBranches: 9		
Distance	**Branches retired**	**Mispredicted**
512	97,956,788	2,436
1024	98,966,308	39,204,144
2048	99,457,820	79,125,818
Iterations: 10M, NoBranches: 5		
Distance	**Branches retired**	**Mispredicted**
512	58,681,763	1,253
1024	58,219,608	29,484
2048	59,219,391	9,792,944
4096	59,807,200	19,687,990
8192	59,736,047	19,548,096
Iterations: 1M, NoBranches: 16		
Distance	**Branches retired**	**Mispredicted**
512	17,018,520	1,953
1024	17,050,360	14,938,664
Iterations: 1M, NoBranches: 8		
Distance	**Branches retired**	**Mispredicted**
1024	9,028,895	2,520
2048	9,034,300	6,927,480
Iterations: 1M, NoBranches: 4		
Distance	**Branches retired**	**Mispredicted**
2048	5,057,136	2,400
4096	5,018,825	4,097

Finally, to verify whether the size assumption was correct, we run the different distance experiment with twice as many branches. Table 2 shows results for the P6 architecture for 1024 branches. The distances that produced low MPR when the number of branches was 512 now produce the MPR close to 100%. From this we conclude that the number of entries is really 512. Figure 11 shows the BTB size and organization as established by our experiments.

Table 2 P6 number of branches retired and mispredicted when the total number of branches is twice as much as N_{BTB}.

Iter. 1M, NoBranches 1024		
Distance	**Branches retired**	**Mispredicted**
4	1,023,000,000	1,017,750,000
8	1,017,400,000	1,016,900,000
16	1,018,900,000	1,020,700,000

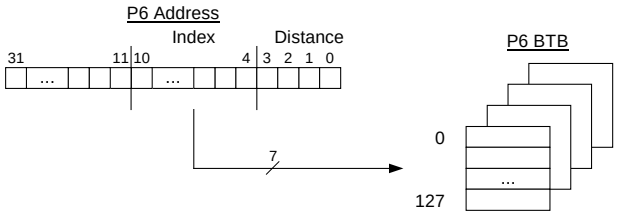

Figure 11 BTB size and organization.

6.1.2. Prediction component

Table 3 shows the results of Step 1 experiment (Figure 5). The maximum length of a correctly predicted pattern is 5. This result can be caused by a local component that uses 4 bits of local history, or a global component that uses 8 global history bits.

Table 3 Results of Step 1 experiment.

Iter.	Pattern length	Branches retired	Mispredicted
10 M	4	27,035,204	420
	5	28,468,884	432
	6	28,205,352	1,545,480

We proceed with the Step 2 experiment that inserts 8 "dummy" conditional branches before the "spy" branch. Since the MPR is still close to 0 when we have longer global history pattern, the P6 architecture really uses a local branch history of length 4.

In the Step 3 experiment, the microbenchmark has 3 conditional branches in a loop, where first two have patterns 11...10 of length 5 and 2, respectively, predictable by the local component. The outcome of the third branch is correlated with the previous two, having a pattern 11...10 of length ten, not predictable by local component. The MPR is about 10%, so the P6 architecture does not use a global history pattern of length greater or equal to two.

The Step 4 experiment is a 10 million iteration loop, with two conditional branches. First branch has a pattern 111110 of length 6, hence not predictable by the local component, while the second branch is correlated with it by having the same outcome. The result is about 3 million mispredicted branches, so both conditional branches are mispredicted once in six times. Therefore, the P6 architecture does not include global prediction component.

6.2. Results for NetBurst architecture

6.2.1. Front-end BTB component

We assume $N_{BTB\text{-}FE}$=4096. Results are similar to the results of the same experiment for the P6 architecture, i.e., the MPR is close to 0%, when the distance between addresses of subsequent branches was 4, 8, or 16, and close to 100% for other distances. Therefore, the front-end BTB has 4 ways and 1024, while bits 4-13 are used as the set index (Figure 12).

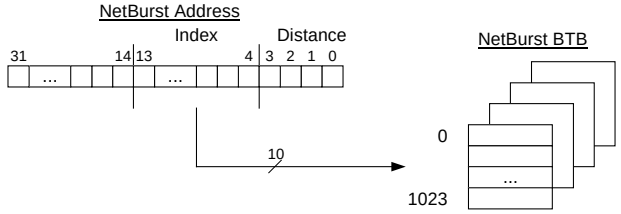

Figure 12 Front-end BTB size and organization.

6.2.2. Outcome predictor

Table 4 shows the results of the Step 1 experiment: the maximum length of a correctly predicted pattern is 9, which could be explained either by 8 bit local history register, or 16 bit global history register.

Table 4 Results of Step 1 experiment.

Iter.	Pattern length	Branches retired	Mispredicted
10 M	5	30,357,420	987
	6	30,362,568	973
	7	30,401,322	957
	8	30,318,387	1,256
	9	30,326,432	918
	10	30,352,542	964,830

In the Step 2 experiment, we insert 16 "dummy" branches before the "spy" branch with a local pattern of length 9, and the measured MPR is about 10%. Therefore, the Step 1 result is caused by a global component that

uses 16 global history bits (the "spy" branch in the Step 2 is mispredicted once in 9 times).

After several runs of different Step 6 experiments, first conclusion might be that the NetBurst architecture uses one local history bit for prediction (Table 5). Because this architecture includes the trace cache, we run an additional experiment, with structure from the Step 6 repeated 10 times (16 "dummy" branches, one "spy" with local history pattern of length 2). Ten "spy" branches have the MPR of about 50%, which is expected for the outcome predictor without any local component. Hence, low MPR in the Step 6 with pattern length 2 is due to the trace cache, most probably since it was able to store the sequence "loop, 16 dummy branches, spy taken, loop, 16 dummy branches, spy not taken" as one continuous trace.

Table 5 Results of Step 6 experiment.

Iter.	Pattern length	Mispredicted branches
10 M	2	0%
	3	33%
	4	25%
	5	20%

6.2.3. Trace cache BTB component

In the experiment with one "spy" branch in a loop, having both *if* and *else* paths and pattern of length 9, the number of cycles the trace cache spent in delivery mode is close to the total number of execution cycles. This result indicates that trace cache is able to store both branch paths, if both are executed.

Since in the previous section we ascertain that the NetBurst architecture does not have a local component, we might use N "spy" branches *if (i%3==0)*. In this paper, we discuss one example for each of (a), (b), (c) cases, while the determining of exact border values of N is not considered here. The distance between spy branch addresses is 32.

For N=2048, the MPR is relatively low (case (a)), since for this value of N microbenchmark code does not fit into the trace cache, but its branches fit into BTB_{FE}. For N=512, the MPR is close to 100%, since executed code can fit into the trace cache, but executed branches do not fit into BTB_{TC} (case (b)). The example for case (c) is when N=127 – MPR is close to 0%, since branches fit into BTB_{TC}.

7. Conclusion

Although a lot of research effort has been dedicated to the branch predictors in the last decades, modern processors still hide the exact predictor implementation details. In this paper we propose a set of experiments aimed at systematically determining the organization of the BTB components, i.e., number of BTB ways and bits used for the set index, the existence of local and global history component, and the corresponding number of history bits. We use Microsoft C compiler and Intel VTune Performance Analyzer tool.

Our experiments show that the BTB in the P6 architecture is 4 ways, same as the NetBurst's front-end BTB. The P6's 128 sets are accessed by address bits 4-10, while the NetBurst's 1024 sets are accessed using bits 4-13. The P6 predictor has a local history component with 4 local history bits, while the NetBurst architecture has a global history component with 16 global history bits. The trace cache in the NetBurst architecture is able to store both taken and not taken branch paths, if both paths are visited during program execution. The determining of exact organization of the trace cache BTB is part of the ongoing research.

We also plan to design experiments aimed to discover other predictor parameters, such as the choice of bits for BTB tag, number of different branch targets stored in BTB, number of bits and starting state of prediction state machines, organization of the outcome predictor component in more complex predictors, etc. We hope that further research in this direction will improve the code optimization and understanding of the predictor mechanisms.

References

[1] Intel VTune™ Performance Analyzer, www.intel.com/software/products/vtune/

[2] C.L. Coleman, J.W. Davidson, "Automatic memory hierarchy characterization," ISPASS 2001, pp. 103 – 110.

[3] R. Saavedra-Barrera, "CPU Performance Evaluation and Execution Time Prediction Using Narrow Spectrum Benchmarking," PhD Thesis, Berkeley, 1992

[4] J.E. Smith, "A study of Branch Prediction Strategies," 8th ISCA, 1981, pp. 135–148.

[5] T-Y. Yeh, Y.N. Patt, "Two Level Adaptive Training Branch Prediction," Micro-24, 1991, pp. 51-61.

[6] S-T. Pan, K. So, J.T. Rahmeh, "Improving the Accuracy of Dynamic Branch Prediction Using Branch Correlation," ASPLOS V, 1992, pp. 76-84.

[7] S. McFarling, "Combining Branch Predictors," WRL Technical Note TN-36, Digital Equipment Corporation, June 1993.

[8] M. Evers, P.-Y. Chang, Y.N. Patt, "Using Hybrid Branch Prediction to Improve Branch Prediction Accuracy in the Presence of Context Switches," 23rd ISCA, 1996, pp. 3–10.

[9] K. Driesen and U. Hölzle, "The Cascaded Predictor: Economical and Adaptive Branch Target Prediction," MICRO-31, pp. 249 -258.

[10] S. McFarling, "Branch predictor with serially connected predictor stages for improving branch prediction accuracy," US Patent 6374349, 2002.

[11] Baweja, et al., “Branch prediction architecture,” US Patent 6332189, 2001.
[12] K. Skadron et al., “A taxonomy of branch mispredictions, and alloyed prediction as a robust solution to wrong-history mispredictions,” PACT 2000, pp. 199-206.
[13] T. Juan, S. Sanjeevan, J. Navarro, “Dynamic History Length Fitting: A Third Level of Adaptivity for Branch Prediction,” 25th ISCA, 1998, pp. 155–166.
[14] R. Nair, “Dynamic Path-Based Branch Corelation,” Micro-28, 1995, pp. 15-23.
[15] S. Sechrest, C.-C. Lee, T. Mudge, “Correlation and Aliasing in Dynamic Branch Predictors,” 23rd ISCA, 1996, pp. 22–32.
[16] P. Chang, M. Evers, Y. Patt, “Improving Branch Prediction Accuracy by Reducing Pattern History Table Interference,” PACT 1996.
[17] E. Sprangle, R.S. Chappell, M. Alsup, Y.N. Patt, “The Agree Predictor: A Mechanism for Reducing Negative Branch History Interference,” 24th ISCA, 1997, pp. 284-291.
[18] A.N. Eden, T. Mudge, “The YAGS Branch Prediction Scheme,” MICRO-31, USA, 1998.
[19] Intel ® Pentium ® 4 and Intel ® Xeon™ Processor Optimization –Reference Manual, www.intel.com
[20] Intel® Architecture Software Optimization Reference Manual, www.intel.com
[21] G. Hinton et al, “The Microarchitecture of the Pentium® 4 Processor,” Intel Technology Journal, 1st quarter 2001.
[22] A. Fog, “How to optimize for the Pentium® microprocessors,” www.agner.org/assem/