Experiment Flows and Microbenchmarks for Reverse Engineering of Branch Predictor Structures

Vladimir Uzelac, Aleksandar Milenković
Electrical and Computer Engineering Department, The University of Alabama in Huntsville
Email: {uzelacv, milenka}@ece.uah.edu

Abstract

Insights into branch predictor organization and operation can be used in architecture-aware compiler optimizations to improve program performance. Unfortunately, such details are rarely publicly disclosed. In this paper we introduce a set of experiment flows and corresponding microbenchmarks for reverse engineering cache-like branch target and outcome predictor structures, indexed by branch address or program path information. The experiment flows are demonstrated on the Intel Pentium M branch predictor. We have been able to determine the size, organization, internal operation, and interactions between various hardware structures used in the Pentium M branch predictor, namely the branch target buffer, indirect branch target buffer, loop branch predictor buffer, global predictor, and bimodal predictor. These findings have been validated using a functional PIN model.

1. INTRODUCTION

Branch predictors are one of the key units in the front-end of modern high-performance microprocessors. They detect branches and predict the branch target address and the branch outcome in the early pipeline stages, thus reducing the number of wasted clock cycles due to control hazards. The target of a direct branch is predicted using a branch target buffer (BTB) [1] – a cache structure indexed by a portion of the branch address. Each BTB entry typically includes the tag field, the offset field, the branch type field (e.g., direct/indirect, unconditional/conditional), the valid bit, the replacement bits for multi-way BTBs, and the target address. A separate hardware structure named an indirect branch target buffer (iBTB) can be employed to handle indirect branches with multiple target addresses [2-4]. The branch outcome predictors have evolved from a simple linear branch history table (BHT) with 2-bit saturating counters (2bc) [5] to very sophisticated branch predictor structures found in recent commercial microprocessors [6-9]. A number of advanced predictor structures have been proposed, including (i) two-level adaptive predictors that exploit global or local branch histories of branch outcomes to achieve a better mapping into the BHT [10, 11] (ii) de-interference predictors which reduce negative effects of branch interference [12-15], (iii) hybrid predictors that include multiple specialized structures [16-18], and (iv) perceptron predictors [19, 20].

Code optimizations based on the information about branch predictor structures can greatly increase overall program performance [21, 22]. For example, if the compiler is aware of the BTB size and organization, it can prevent branch interference in critical portions of the code by re-aligning the branch instructions. Next, if the compiler is aware of local and global branch history lengths, it can employ code duplication or loop unrolling transformations to alleviate mispredictions [21]. Jimenez introduced the Camino C compiler [22] that exploits knowledge about branch predictor internal structures. It performs feedback-directed code placement to reduce the number of branch mispredictions in the NetBurst architecture. This optimization reduces the number of branch mispredictions in the SPEC CPU2K benchmarks in the range of 22% to 3.5%.

Unfortunately, microprocessor manufacturers rarely fully disclose information about the branch predictor organization thus preventing efforts aimed at better code optimization. This problem can be addressed by employing reverse engineering techniques aimed at branch predictor units. A prior reverse engineering flow focusing on P6 and NetBurst architectures [21] has been successful in determining the size and organization of the BTB and the presence and lengths of global and local histories. However, this flow does not include any experiments for determining the organization of predictor structures indexed by program path information nor their internal operation. In addition, it does not include any experiments to uncover update and allocation policies and the exact type, size, and organization of outcome predictors.

In this paper we introduce a set of new experiment flows and corresponding microbenchmarks that can be used in determining the organization and operation of modern branch predictor units. Specifically, we introduce new tests for (a) determining the size, organization, and operation of target predictors (BTB and iBTB), indexed by the branch address (IP-based) and the program path (path-based) information, (b) determining the size, organization, and operation of complex hybrid outcome predictors, and (c) uncovering interdependencies between predictor structures. The experiment flows are demonstrated through a reverse engineering of the Pentium M (Dothan core) branch predictor unit, a sophisticated predictor that significantly outperforms previous Intel branch predictors [6].

The experiment flows indicate the following. The Pentium M target predictor includes a 4-way set-associative BTB with 2048 entries and a direct-mapped iBTB with 256 entries for

indirect branches. The BTB is indexed by the instruction pointer bits IP[12:4] and the tag field is IP[21:13]. The iBTB is indexed using a hash access function of a path information register (PIR) and the branch address bits. We have uncovered the size and the update policy for the PIR, the hash access function, the iBTB tag bits, and the iBTB index. The outcome predictor is similar to McFarling's serial BLG predictor [16] with the following structures: (a) a 4096-entry bimodal predictor indexed by IP[11:0]; (b) a tagged 2-way loop predictor with 128 entries, indexed by IP[9:4]; and (c) a tagged 4-way global predictor with 2048 entries, accessed in a similar fashion as the iBTB. In addition to these structural parameters, we have been able to answer a number of questions regarding interactions between predictor structures. To validate our approach, we have developed a functional model of the Pentium M branch predictor using PIN [23]. We collect statistics about relevant events while running several SPEC benchmarks on the PIN model and compare them with the statistics collected using Intel VTune [24]: the results show that the PIN model is accurate, confirming our findings.

The contributions of this paper are as follows: (i) we introduce new experiment flows and microbenchmarks for determining the organization and operation of target predictors, specifically the BTB (Section 3) and the iBTB (Section 4); (ii) we introduce new experiment flows and microbenchmarks for determining organization and operation of outcome predictors, specifically the loop predictor (Section 5), and the global and bimodal predictors (Section 6), and (iii) we develop a functional PIN model of the entire Pentium M branch predictor and validate it by running SPEC benchmark programs (Section 7). In Section 8 we describe applicability of the experiment flows to other branch predictors and discuss their limitations.

In addition to assisting code optimizations, the proposed flows and microbenchmarks can also expedite verification efforts and rapid design-space exploration in the design phase of branch predictors, allowing for shorter time-to-market and more reliable designs. Our findings can help researchers in academia and industry who develop new branch predictors by offering a good starting point for comparison. The following Section gives preliminaries, sets goals, and describes the experimental methodology used in this effort.

2. PRELIMINARIES

Branch Prediction in the Pentium M. The Pentium M is Intel's first microprocessor specifically designed for mobility, with a goal to achieve the best performance at given power and thermal constraints [6]. Improved branch prediction is desirable not only for increased performance, but also for reduced power consumption. The architects disclose that the advanced Pentium M branch predictor is based on the Pentium 4 branch predictor, with two new structures: the loop predictor and the indirect branch target predictors.

The loop predictor captures the behavior of loop-like branches that cannot be accurately predicted by global or local history predictors. A loop branch moves in one direction many times with a single instance in the opposite direction, e.g., taken k times and not taken once – $\{\{T\}^k.NT\}$. When a loop branch is detected, an entry in the loop predictor is allocated. Each entry contains fields to record the current iteration counter (*Count*), the iteration limit (*Limit*), and the direction of the branch. The iBTB predicts targets of indirect branches using global branch history. When an indirect branch is mispredicted in the BTB due to an incorrect target, the iBTB allocates a new entry using global history leading to this instance. The entries in the iBTB are tagged.

Assumptions. Based on the available information we assume that the Pentium M branch predictor unit consists of a BTB, an iBTB, and a hybrid outcome predictor. Intel software optimization manuals [25] state that the Pentium M predicts all branches dynamically, indicating a possible multi-layer outcome predictor with a bimodal predictor in the first level. Hence, we assume that the outcome predictor features the bimodal predictor (a simple table of 2-bit counters), the loop predictor, and a global predictor of unknown structure. It should be noted that these assumptions are not necessary for the proposed flows, because presence or absence of these structures can be established by additional tests.

What Would We Like to Know? For all predictor structures we seek to determine structural parameters such as their size, organization, and relevant fields and their lengths. Another group of parameters is related to allocation, update, and replacement policies employed for each of the predictor structures. Next, we need to determine hash functions used for accessing these structures. For example, the BTB is usually indexed by a portion of the current branch address. However, the Pentium M architects state that the iBTB is accessed using global path information. Hence, the global program path information has to be dynamically maintained in a dedicated register called a path information register (PIR). A number of questions arise concerning the PIR, including its size, update policy (type of branches affecting the PIR, type of information used, e.g., address bits or outcomes or both), and its use in the iBTB hash access function.

We also need to determine the predictor-specific parameters. For example, for the loop predictor we would like to know the size of counters as well as the training policy. For the BTB we would like to know how it handles bogus branches. Finally, for a complete reverse engineering we need to determine the relationship between different predictor structures, for example, BTB/iBTB relationship, or how the final outcome prediction is created from individual predictors.

Experimental Methodology. We employ the following experimental methodology. First, we make a hypothesis on a particular predictor structure or mechanism and assess its testing space. Next, a microbenchmark (or a series of microbenchmarks) is developed in C and/or assembly language. A typical microbenchmark has to do the following: (a) identify and isolate parameters or mechanisms for hypothesis testing; (b) amplify the parameters or mechanisms of interest, so they can be easily measured or observed; and (c) mask out the

effects of all other microarchitectural parameters or mechanisms that can obstruct or prevent analysis. We also need to select a list of observable events for the parameters or mechanisms tested in the hypothesis. This step ends with an analysis to establish expectations for the given hypothesis.
The microbenchmarks are executed and event statistics are collected using Intel's VTune [24]. The results are analyzed and compared with the expectations. If the results confirm the hypothesis, the hypothesis becomes a new finding, and the process continues with other hypotheses. If possible, we develop several alternative tests targeting the same hypothesis, thus increasing the confidence in the findings.

The design of experiments and microbenchmarks is strongly influenced by the observable microachitectural events. VTune offers several events that are related to branch instructions in Pentium M. The most important events of interest for our flows are: mispredicted branches at decoding (MBIDEC), mispredicted branches at execution (MBIEXEC), mispredicted indirect branches (MIBIE), indirect branches executed (IBIE), conditional branches executed (CBIE), and mispredicted conditional branches (MCBIE).

3. BTB TESTS

3.1 BTB Capacity Test

In determining BTB organization, we start with a so-called *BTB Capacity* test [21]. This test stresses the BTB structure trying to find the maximum number of branches *B* that can fit in the BTB. The branches are placed in a loop at equidistant memory locations with distance *D* (Figure 1*A*). The branches at labels *l0*, *l1*, ..., *lm1* are called *spy* branches. By varying the parameters *B* and *D*, we can, under certain conditions, determine the BTB size (N_{BTB}) and organization.

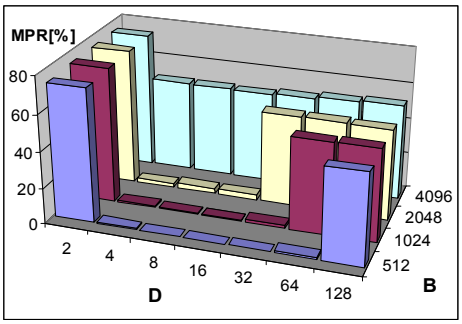
Generally, when $B=N_{BTB}$ and the test gives *m* "fitting" distances *D* (that produce no mispredictions), the number of ways in the BTB is $N_{WAYS}=2^{m-1}$ and the BTB index is IP[i+j-m:i], where the maximum fitting distance is $D_{MAX}=2^i$ and $j=log_2N_{BTB}$ (the least and the most significant bits of the index field are *Index.LSB=i*, *Index.MSB=i+j-m*, respectively). The *BTB Capacity* test analysis is based on Eq. 1, where MPR is the misprediction rate defined as the number of mispredicted branches at decoding divided by the total number of spy branches executed.

The *BTB Capacity* test will suffice in determining the BTB size, associativity, and index bits, if the following conditions are met: (a) a portion of the branch address is used as the BTB index; (b) a portion of the branch address adjacent to the index field is used as the BTB tag; and (c) the replacement policy is round-robin, or the least recently used (LRU), or one of the LRU derivatives.

However, the *BTB Capacity* test results on Pentium M are inconclusive. A portion of the results indicates a 4-way BTB with 1K entries, and the other portion a 4-way BTB with 2K entries. We prove that this is due to the BTB allocation and replacement policies. Therefore, we devise a test to stress allocation or replacement policies by controlling the BTB

```
Offset    Code
          // rpt. n times
    0     l0: jmp l1
          //non-branch ins.
    D     l1: jmp l2
          //non-branch ins.
   2D     l2: jmp l3
          ...
(B-2)D    lm2:jmp lm1
          //non-branch ins.
(B-1)D    lm1: jmp l0
```

A



B

Figure 1. BTB Capacity test (*A*); BTB Capacity Hit test results (*B*).

$$MPR=\begin{Bmatrix}0,\ B\le N_{BTB}\ and\ i-m\le k\le i+j-l\\ 100\%,\ B>N_{BTB}\end{Bmatrix},\ B=2^l,\ D=2^k \quad \text{Eq. 1}$$

access pattern. The execution pattern of the spy branches from Figure 1*A* $\{l0, l1, \ldots, lm1\}^n$ is replaced by an alternative execution pattern $\{\{l0\}^2, \{l1\}^2, \ldots, \{lm1\}^2\}^n$, where each spy branch is executed twice consecutively. This pattern ensures that each spy branch finds itself in the BTB. This variant of the test is called the *BTB Capacity Hit* test. We may also need to control the rate by which the branches are presented to the BTB, to give enough time for preceding branches to retire and update the BTB.

The *BTB Capacity Hit* test results are shown in Figure 1*B*. According to Eq. 1, the results indicate a 4-way BTB that can hold up to 2048 branches, with IP[12:4] used as the BTB index. Note: high MPR when $B>N_{BTB}$ is 50% because each branch is touched twice consecutively – one miss and one hit.

3.2 BTB Set Tests

BTB Set tests are similar to the *BTB Capacity* test, but instead of stressing the whole BTB, we stress a single BTB set. We vary the distance in memory between spy branches *D*, trying to find the maximum number of branches *B* that map into a single set. Again, by observing the MPR as a function of *B* and *D*, we can determine the BTB parameters. We introduce three new tests as shown in Figure 2. These tests can be used not only to confirm the findings from the *BTB Capacity Hit* test, but also to determine the BTB tag field.

The test searching for the tag MSB, (*Tag.MSB*) (Figure 2*A*) includes only two spy branches placed at arbitrary distance *D* that result in no mispredictions. The distance *D* is increased until the BTB cannot distinguish between the spy branches resulting in mispredictions; that distance is $D=2^{Tag.MSB+1}$.

The test searching for the *Index.MSB* and N_{WAYS} (Figure 2*B*) starts from $B=2$ and an arbitrary D, $D<2^{Index.MSB}$, and B and D are varied as described in Figure 2*B*. When the spy branches map into a single set ($D>2^{Index.MSB}$), the MPR depends only on B and will exist when $B \geq 1+N_{WAYS}$. When the spy branches map into multiple BTB sets ($D \leq 2^{Index.MSB}$), the MPR depends on both B and D.

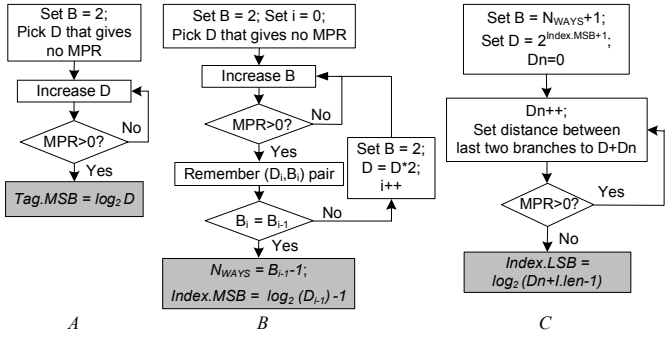


Figure 2. BTB Set tests.

Finally, to determine the *Index.LSB* (Figure 2*C*), we set $B=N_{WAYS}+1$, and all spy branches are mapped into a single set (e.g., $D=2^{Index.MSB+1}$), causing all spy branches to be mispredicted by the BTB. The distance between the last two spy branches is then increased for Dn, $1 \leq Dn \leq 2^{Index.LSB}-1$, until the mispredictions disappear because the last spy branch maps into the next set. It should be noted that the Pentium M architecture takes the address of the last byte as the branch instruction address [26]. For example, the branch address of a 5-byte instruction starting at the address 400Ch is 4010h (IP=FirstByteAddress + I.len – 1, where I.len is instruction length). Note: the existence of this mechanism can be verified by a microbenchmark.

Figure 3*A* shows the MPR when distance D is varied for $B=2$ (tests from Figure 2*A*). The MPR=100% when $D=2^{22}$, indicating that the *Tag.MSB* is IP[21] and the tag field is IP[21:13]. The results from Figure 3*C* confirm previous findings about the BTB index bits and associativity (tests from Figure 2*B*). Figure 3*B* shows that the MPR becomes low when $Dn=12$. The spy branches are 5 bytes long, thus we conclude that the *Index.LSB*=4.

3.3 Miscellaneous BTB Details

We can identify several other BTB design issues that can be easily explored using variants of the tests described above. We find that '*always not taken*' branches are not allocated in the BTB. Next, a bogus branch, a non-branch instruction that hits in the BTB (has the same index and tag fields as an allocated branch), discards the whole BTB set.

Finally, we verify the existence of the so-called *offset algorithm*. Each BTB entry includes the offset field that is IP[3:0] [26]. A single fetch line (16 bytes) may have several branch instructions, resulting in multiple BTB hits. The offset algorithm chooses the final prediction in the presence of multiple BTB hits – it is the BTB entry with the smallest offset, yet not smaller than the current IP offset.

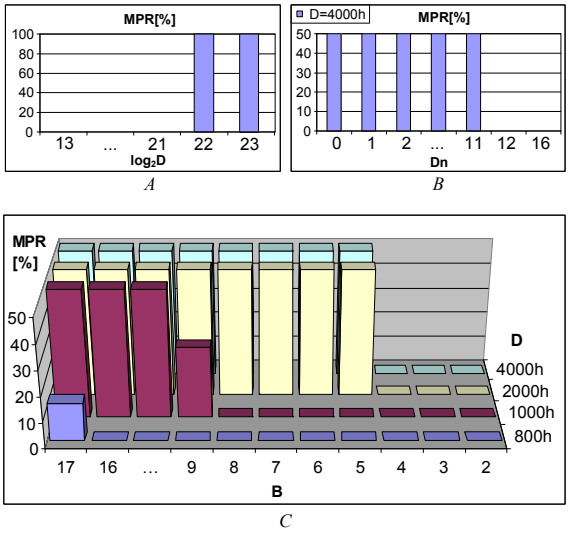


Figure 3. BTB Set test results.

The *BTB Capacity Hit* and the *BTB Set Hit* tests indicate a non-random BTB replacement policy. For example, the *BTB Set Hit* test with $B=5$ and the execution pattern $\{\{l1\}^2, \{l2\}^2, \ldots \{l5\}^2\}^n$ produces MPR=50%. Either the LRU, round-robin, or a pseudo-LRU replacement policy would give the same results. To determine which policy is used we vary the execution pattern in the execution sequence. For example, the pattern $\{\{l1\}^2, \{l2\}^2, \{l3\}^2, \{l1\}^2, \{l4\}^2, \{l5\}^2\}^n$ makes the spy branches *l3* and *l5* compete for a single BTB entry, while other branches keep their entries. This indicates the tree-based pseudo-LRU replacement policy.

4. INDIRECT BTB TESTS

We assume that the iBTB is a tagged, cache-like structure accessed using a hash function, likely an XOR, of a portion of the indirect branch address, IP[n:m], and the path information register (PIR) (Figure 4*A*). The PIR is typically updated using various program path information, such as: branch address bits IP[q:p], branch target address bits TA[s:r], branch type, and branch outcome [2-4] (Figure 4*B*). In addition, different types of branch instructions may affect the PIR differently. To reverse engineer the iBTB, we need to determine the size and update policy of the PIR, the iBTB hash access function, and the iBTB size and organization.

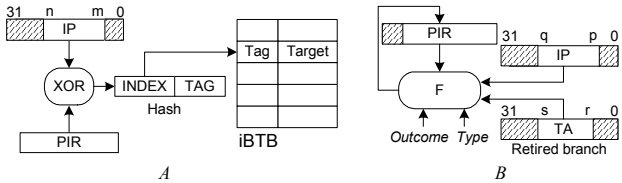


Figure 4. iBTB access hash (*A*); PIR update (*B*).

4.1 PIR Tests

The PIR tests determine the PIR history length, the PIR length, the type of branches affecting the PIR, and the PIR update policy. In the PIR tests we try to control the number of indirect branch collisions by changing the PIR value only.

A key test is shown in Figure 5*A*. The test has an indirect branch called *iSpy* with two targets *Target1* and *Target2*. The test alternates between two program paths, *P1* and *P2*, as follows: *P1→iSpy→Target1* and *P2→iSpy→Target2*. The paths *P1* and *P2* have *N* setup branches (*Pi.SB1, Pi.SB2, ... Pi.SBN, i=1, 2*). *N* should be larger than the PIR history depth, i.e., only the setup branches in *P1* or *P2* are affecting the PIR. By controlling the placement of the setup branches, their outcomes and targets, we control the PIR content observed by the *iSpy* branch. For example, we can ensure that both paths produce the same PIR by employing an equal number of conditional branches with the same outcomes, placed in memory in such a way that $IP(P1.SBi)=IP(P2.SBi)+2^{q+1}$, where IP[q:p] are branch address bits that affect the PIR (see Figure 4*B*), and *i=1...N*. If both paths yield the same PIR, the iBTB will not distinguish between two program paths, which will result in iBTB mispredictions due to incorrect target addresses for the *iSpy* branch. The BTB will also mispredict the target because it has only one target field and the program alternates between two paths. Similarly, distinctive PIR contents for program paths *P1* and *P2* may be achieved by setting two setup branches to have different address fields that affect the PIR. For example, the setup branches *P1.SB1* and *P2.SB1* are laid out in memory in such a way to satisfy the following equation: $IP(P1.SB1)=IP(P2.SB1)+2^{q+1}+D$, where $D=2^k$. If bit *k* is affecting the PIR, i.e., $p \le k \le q$, the PIR values are distinctive, causing no mispredictions in the iBTB. Otherwise, both program paths will have the same PIR value, causing both targets to be mispredicted.

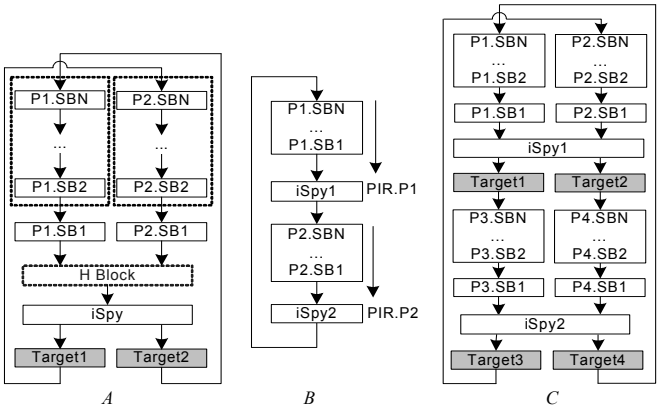


Figure 5. PIR test (*A*). iBTB access hash tests (*B*, *C*).

We expect three possible situations to occur with the *iSpy* target prediction in the experiment: (a) an iBTB hit with incorrect target prediction (misprediction), (b) an iBTB hit with correct target prediction, and (c) an iBTB miss with the BTB giving prediction. An iBTB miss occurs only when the iBTB is a direct-mapped cache and the targeted bits belong to the iBTB tag field. Depending on the BTB and iBTB update policies, the BTB can provide a correct or an incorrect target.

The next step is to determine which address and target bits are affecting the PIR and how the PIR is updated with new branch information. First, the setup branches are laid out in memory to produce distinctive PIR contents. Next, the test is modified to include a code block H (Figure 5*A*) in front of the *iSpy* branch. The block H consists of *H* branches affecting the PIR. If *H* becomes large enough, the setup branches *P1.SB1* and *P2.SB1* no longer affect the PIR, consequently causing mispredictions in the iBTB. The minimum *H* causing mispredictions regardless of the distance *D* determines the PIR history depth.

Figure 6 shows the results of the PIR test. We plot the MPR, calculated as the number of mispredicted indirect branches at execution (MIBIE) divided by the number of indirect branches (IBIE), as a function of *D* and *H*. When *H=0*, we can observe 3 distinct MPR values: 0%, ~40% and 100%. The MPR=100% occurs for $D \notin (2^4\text{-}2^{18})$ due to mispredictions with iBTB hit. MPR=0% occurs for $D=2^{10}\text{-}2^{18}$ due to iBTB hits with correct predictions. Finally, MPR of ~40% for $D=2^4\text{-}2^9$ and $D=2^{18}$ can only be due to iBTB miss events (this case is further discussed at the end of this section). Thus, we conclude that IP[18:4] affects the PIR. When *H=1*, the address bits affecting the PIR are IP[16:4], indicating that a 2-bit shift is used in the PIR update. When *H=8*, the MPR=100% regardless of the distance *D*, indicating that the PIR history depth is 8.

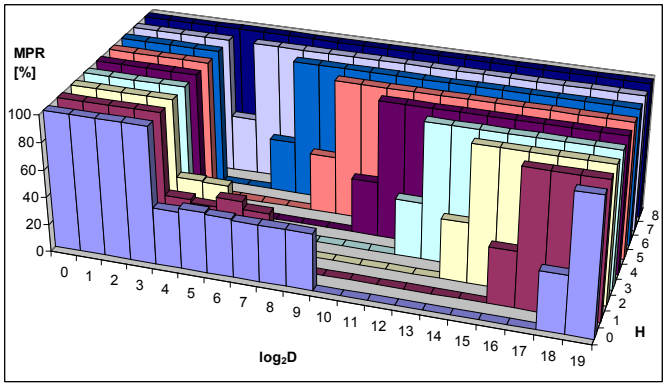


Figure 6. PIR test results.

The same test is repeated while varying (a) the type of branches in the program paths, (b) target addresses, and (c) branch addresses. The results of these experiments indicate the following findings. The PIR is affected by branch address bits IP[18:4] of *taken conditional branches*; and IP[18:10] and TA[5:0] of *indirect branches*. *Direct unconditional branches*, *not taken conditional branches*, *targets of taken conditional branches*, and *branch outcomes* do not affect the PIR.

The next step is to determine the size of the PIR and its updating function *F* (Figure 4*B*). We consider two basic approaches to PIR update: shift-and-xor, and shift-and-add. With shift-and-xor, *S* bits of branch information are XOR-ed with *S* bits of the PIR, where length(PIR)≥*S*. With shift-and-add, *S* bits of branch information are shifted in, where length(PIR)=*S**(PIR history depth). To find which policy is used, we devise a test that stresses the PIR update policy (xor or add). With the shift-and-xor policy asserted address bits in

one path may interact and cancel out, thus making the path indistinguishable from another path. This cannot happen with shift-and-add policy.

The test from Figure 5*A* is employed to prove that the shift-and-xor policy is used. The setup branches are laid out as follows (Eq.2):

$$\begin{aligned} &IP(P1.SBi) = IP(P2.SBi) + 2^{q+1},\ i = 3...N \\ &IP(P2.SB2) = IP(P1.SB2) + 2^{q+1} + 2^{k_1} \\ &IP(P2.SB1) = IP(P1.SB1) + 2^{q+1} + 2^{k_2},\quad p \le k_1, k_2 \le q \end{aligned} \qquad \text{Eq. 2}$$

The bits k_1 and k_2 are synchronously set in the path *P2*. If the PIR update uses an XOR, the effect of these two offsets will be annulled, resulting in the same PIR value for both paths, and consequently in *iSpy* mispredictions. We conclude that the PIR length is equal to 15, i.e., the number of IP bits used for the PIR when H=0, $l_{H=0}$. If the PIR length were larger than 15, $l_{H=1}$ would also be 15 since no IP bits would be shifted out of the PIR. In our case $l_{H=1}$=13, so the PIR length has to be 15. The PIR update policy is specified in Eq. 3 (*cbt=1* for conditional taken branches, and *ibt=1* for indirect branches).

$$\begin{aligned} &PIR[14:0] = (PIR[12:0] << 2)\ xor \\ &\quad (cbt \cdot IP[18:4]\ or\ ibt \cdot (IP[18:10]\ concat\ TA[5:0])) \end{aligned} \qquad \text{Eq. 3}$$

4.2 iBTB Hash Access Function Test

The first step is to determine IP address bits of the indirect branch used in computation of the iBTB hash access function (Figure 4*A*). Similarly to the PIR test, our approach is to observe the number of collisions in the iBTB as a function of the indirect spy branch address only. This test is illustrated in Figure 5*B*. It includes two indirect spy branches *iSpy1* and *iSpy2*, each preceded by a setup block, P1 and P2, respectively. The setup blocks consist of *N* conditional always taken branches, laid out in memory to yield the same PIR for two spy branches, i.e., PIR.P1=PIR.P2. We change the distance between the spy branches, thus affecting the hash access function of the iBTB. The spy branches are placed in memory so the following equation holds: $IP(iSpy2) = IP(iSpy1)+2^{n+1}+D$, where $D=2^l$. If the bit *l* affects the hash access function, i.e., $m \le l \le n$ (see Figure 4*A*), the spy branches will not produce mispredictions. This test is a subset of the following test and its results will be discussed later.

The next step is to determine how the iBTB hash access function is computed from the PIR and the indirect branch address bits. We hypothesize that an XOR function is used. To prove this assumption we stress the hash function by synchronously controlling the PIR bits and the spy branch address bit in order to cancel out the asserted bits. This is achieved by combining approaches from two previous tests, as illustrated in Figure 5*C*. The test includes two indirect spy branches *iSpy1* and *iSpy2*, each spy branch preceded by setup branches laid out in memory to produce the same PIR for different paths. The setup branches P2.SB1 (P1.SB1) and P4.SB1 (P3.SB1) differ only at a PIR-affecting address bit *k*: that is, $IP(Pi.SB1)=IP(Pi-1.SB1)+2^{q+1}+2^k$, where $p \le k \le q$, $i=2,4$. The spy branches *iSpy1* and *iSpy2* differ only at a bit *l*, such that the following equation holds: $IP(iSpy2) = IP(iSpy1)+2^{n+1}+2^l$, where $m \le l \le n$. If bits *k* and *l* are combined in the hash access function, *Target2* and *Target4* collide (as well as *Target1* and *Target3*) causing mispredictions in the iBTB, in spite of unique PIR contents observed by *iSpy1* and *iSpy2*, as illustrated below:

$$P1.SB1.IP[k] = '0';\ iSpy1.IP[l] = '0';\ \rightarrow \text{XOR}=0$$
$$P3.SB1.IP[k] = '1';\ iSpy2.IP[l] = '1';\ \rightarrow \text{XOR}=0$$

The test advances through all (*k, l*) pairs, where $k, l \in [4\text{-}18]$. The results indicate that the following hash access function is used (Eq. 4).

$$\begin{aligned} HASH[14:0] = &(IP[18:13]\ xor\ PIR[5:0])\ concat \\ &(IP[12:4]\ xor\ PIR[14:6]) \end{aligned} \qquad \text{Eq. 4}$$

4.3 iBTB Organization Test

In determining the iBTB size and organization, we try to fill the whole iBTB with a number of indirect branch targets. We devise a test where the MPR is a function of the number of program paths and their respective PIR values. The iBTB size test (Figure 7) has a single indirect spy branch *iSpy* with multiple target addresses; each target is preceded by a unique program path *Pi*, *i=1..N*. To fill the iBTB with a maximum number of indirect targets, we generate all indexes of the iBTB access hash function.

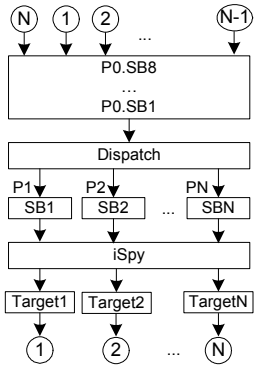


Figure 7. iBTB size and organization test.

The iBTB access hash function is controlled indirectly as follows. The *iSpy* access hash function depends on its branch address and the observed PIR. A path *Pi* includes a setup branch *SBi*, a dispatch branch, and 8 (PIR depth) branches preceding the dispatch branch (*P0.SB1-P0.SB8*). Hence, by controlling the placement of the setup branches *SB1-SBN* and the targets of the dispatch branches, we control directly the *iSpy*'s PIR, and indirectly the iBTB access hash. More specifically, the setup branches satisfy the following equation: $IP[SBi]=2^{q+1}+(i-1)\cdot D$, where $D=2^k$, $i=1..N$, $p \le k \le q$.

The test traverses program paths in sequence: $\{P1 \rightarrow P2 ... \rightarrow PN\}$. We observe the MPR as a function of the distance *D* and the number of unique paths *N* (or spy targets). Based on these findings we can determine the size and organization of the iBTB. Note: The dispatch branch is also an indirect branch preceded by 8 conditional taken branches, so it has a single hash access value regardless of its target addresses. This way, it will take only one iBTB entry.

To illustrate this test, let us consider a direct-mapped iBTB with 256 entries, with the upper 8 bits of the iBTB access hash used as the iBTB index, and the remaining 7 lower bits used as the iBTB tag. When $D=2^{11}$ ($D=800h$), we can fit

N=256 of *iSpy*'s targets in the iBTB. When $D=2^{12}$, the number of targets fitting in the iBTB is 128. Hence, the LSB bit of the index field can be determined by finding the maximum *N* that causes no mispredictions. For all distances *D*, $2^4 \le D \le 2^{11}$, only one *iSpy* target will fit in the iBTB.

The test results for *N=2* and $2^4 \le D \le 2^{18}$ are equivalent to those shown in Figure 6 when *H=0*. Figure 8*A* shows the results for *N=3* and $2^4 \le D \le 2^{19}$. Small distances (*D<400h*) produce an MPR of 100%, indicating a change in iBTB tag bits, as already observed for *N=2*. When *D=400h*, we observe no mispredictions for the *iSpy* branch for both *N=3* and *N=2*, indicating that the *Index.LSB* bit corresponds to the PIR[6].

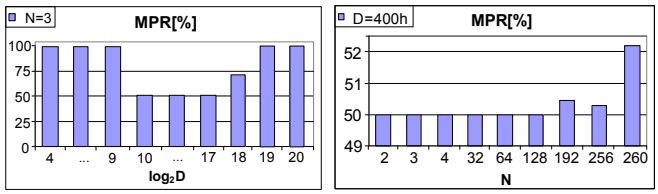


Figure 8. iBTB size test results.

Further testing can be done by setting a fixed distance *D=400h* and varying the number of *iSpy* targets (Figure 8*B*). The results indicate that the iBTB size is 256 entries. Consequently, iBTB index bits are PIR bits PIR[13:6]. Other PIR bits, PIR[5:0] and PIR[14] are iBTB tag bits.

The results in Figure 6 show two distinct MPR values for two events, iBTB miss, and iBTB hit with misprediction. We hypothesize that this difference comes from two different allocation policies employed for these two events (in both BTB and iBTB) in presence of the test's execution pattern. The original execution pattern with alternating paths $\{P1, P2\}^n$ is a corner case where observed results are due to the iBTB and BTB interdependencies in the presence of constant misses or hits with mispredictions. Hence, we replace the original execution pattern by a new pattern $\{\{P1\}^2,\{P2\}^2\}^n$ that eliminates the corner case. This test shows that the MPR=50% for both cases, when PIR.P1=PIR.P2 and PIR.P1 differs from PIR.P2 only in iBTB tag bits. Consequently, we verify our observation that the iBTB is direct-mapped.

5. LOOP PREDICTOR TESTS

We assume that the loop predictor is a cache-like structure indexed by a portion of the branch address only, because the loop predictor exploits local correlations. The reverse engineering of the loop predictor involves determining the counters' length; the size and organization of the loop predictor buffer (LPB); access function; allocation, replacement, and training policies; and its relationship to other predictor structures.

5.1 Loop Counters Length Test

To determine the maximum counter length (*CL*) we use a test as shown in Figure 9*A*. A spy conditional branch, *lSpy*, exhibits loop-like behavior, e.g., $\{\{T\}^L.NT\}$. The *lSpy* is placed inside a loop with a large number of iterations. We observe the MPR defined as the number of mispredicted branches at execution divided by the total number of executed spy loops. As long as $L \le 2^{CL}$, we expect the MPR=0%; if $L > 2^{CL}$, the expected MPR=*1/(L+1)*.

The test results from Figure 9*B* show that the maximum predictable pattern length is *L=64* and confirm expectations for the MPR when *L>64*. Consequently, counters *Count* and *Limit* are 6-bit long.

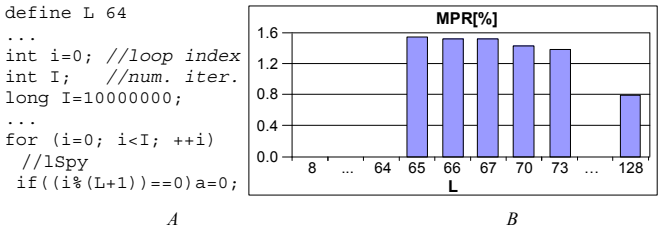


Figure 9. Counter length test (*A*) and test results (*B*).

5.2 LPB Size and Organization Tests

In determining the LPB size and organization, we use the same approach as in the *BTB Capacity* and *BTB Set* tests discussed in Section 3.1 and Section 3.2. The only modification is that spy branches used in the BTB tests are replaced by spy loops in the LPB tests. The loop is formed by replacing an always Taken branch with a conditional branch. We observe the MPR while varying the number of loops *B* and the distance *D* as shown in Figure 2.

Figure 10 shows the results of the *LPB Set* tests for *B=2-4* and $D=2^8$-2^{16} and for $Dn=0$-2^4. Together with the capacity test, these results indicate that the LP is a 128-entry, 2-way cache, indexed by address bits IP[9:4], and tagged by address bits IP[15:10].

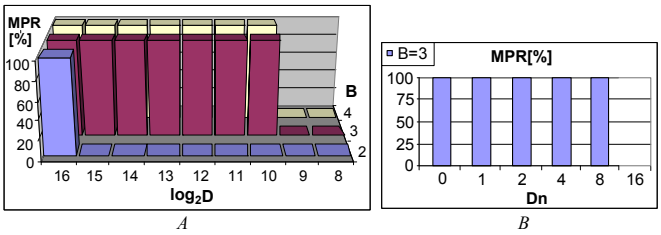


Figure 10. LP-set test results; Searching for the *Index.MSB* and the number of ways (*A*); Searching for the *Index.LSB* (*B*).

5.3 Miscellaneous Loop Predictor Details

This section addresses the issues of loop training, allocation, and replacement policies, as well as the relationship between the LPB and the BTB.

A branch can be allocated in the LPB on the first opposite outcome, or, if separate training logic exists, once a real loop-like behavior is detected. A test to determine the LPB allocation policy includes 3 spy branches that map into a single set. Two spy branches *lSpy1* and *lSpy2* exhibit real loop-like behavior, e.g., $\{\{T\}^{L1}.NT\}$, $\{\{T\}^{L2}.NT\}$, while the third one *lSpy3* has the pattern $\{\{T\}^3.\{NT\}^2\}$. If an LPB entry is allo-

cated on the first opposite outcome, before the real loop-like behavior is detected, all three branches will compete for two LPB entries, causing the MPR to vary with the parameters L1 and L2. Otherwise, the two loop branches will always be predicted correctly, and the MPR will be independent from L1 and L2 (it will come only from *lSpy3*). The results (not shown here) show that the MPR depends on L1 and L2, indicating that the allocation is done on the first opposite outcome and that the training takes place in the LPB.

To determine the LPB replacement policy, we employ 3 spy loop branches that map into a single LPB set. We control loop occurrence pattern, e.g. {*lSpy1, lSpy2, lSpy1, lSpy3*}. If the LRU policy is used, the *lSpy1* will never be mispredicted, while *lSpy2* and *lSpy3* will compete for the same entry. If the round-robin policy is used, all three loops are mispredicted. The results (not shown) indicate that the LRU policy is employed.

The next step is to verify whether an LPB hit is conditional upon a BTB hit. The BTB can better distinguish between branches because it has a longer tag field. A *BTB Set* test is expanded to include 4 always taken spy branches and one *lSpy* branch, all mapping into the same BTB set. As the always taken branches do not consume LPB resources, we expect no mispredictions, if filtering is not employed. With filtering, we will see a certain number of *lSpy* mispredictions. Our measurements show that the number of mispredictions is reciprocal to (L+1), indicating that LPB hits are conditioned by the corresponding BTB hits.

6. GLOBAL OUTCOME PREDICTOR TESTS

We assume a general structure of the global predictor that includes a branch history table (BHT) with 2-bit saturating counters (2bC). The BHT is accessed by a hash of the branch history register (BHR) and a portion of the branch address bits (IP). The reverse engineering of the global predictor involves determining: (a) predictor element, (b) size and update policy of the BHR, (c) access function, (d) BHT size and organization, and (e) relationship with the loop and bimodal predictors.

First we verify that the BHT consists of the 2bC elements. The test includes a conditional branch instruction *cSpy* with the following outcome pattern: $\{\{T\}^3,\{NT\}^2\}$. The *cSpy* is preceded by a number of conditional branches ensuring that all instances of the *cSpy* see the same BHR, so it targets a single cell in the BHT. The MPR of 60% confirms that the 2bC is used.

6.1 BHR Tests

In determining the BHR's size, organization, and update policy we use the same methodology as presented in the PIR tests (Figure 5*A*). Instead of an indirect branch, we use a conditional spy branch *cSpy* with a controllable outcome pattern. The *cSpy* is always taken when reached through the path P1, and always not taken when reached through the path P2. Thus, if BHR.P1=BHR.P2, the *cSpy* will be mispredicted in the global predictor. To prevent other predictors delivering a correct prediction, the *cSpy* employs the outcome pattern $\{\{T\}^v, \{NT\}^v\}$, where v is large enough to eliminate any feasible local prediction.

By varying the distance D between the setup branches *P1.SB1* and *P2.SB1*, we find that branch address bits IP[18:4] are affecting the BHR – exactly the same bits found to affect the PIR (Figure 6). Similarly to the PIR tests, we expand this experimental flow to find the BHR length, type of branches affecting the BHR, and the update policy (see Section 4.2). We find that the BHR is identical to the PIR, meaning that only one path register PIR is maintained and it is used both to access the iBTB and the global outcome predictor.

6.2 BHT Hash Access Function Test

We hypothesize that the same hash access function is used for the outcome predictor and the iBTB. To verify this hypothesis, we need to adapt the experimental flows described in Section 4.1 (Figure 5*C*) for outcome prediction. This adaptation is rather straightforward: the indirect spy branches are replaced by conditional branches. However, to be able to observe events when both outcomes of a spy branch are predicted by a single 2bC element in the global predictor, we need to prevent possible correct predictions coming from the loop and bimodal predictors. Hence, the conditional spy branches employ the outcome pattern $\{\{T\}^{v-1},NT\}^a$, where v=65. If both *cSpy1* and *cSpy2* map into a single 2bC cell, the pattern observed by the cell is $\{\{T\}^{2(v-1)},\{NT\}^2\}^a$ and both NTs are mispredicted together with first T, thus the MPR is $3/(2v)$; otherwise only NTs are mispredicted and the MPR is $2/(2v)$. By varying the placement of the setup branches and spy branches, we find that the hash access function is identical to the one shown in Eq. 4.

6.3 BHT Size and Organization Tests

We hypothesize that the global predictor is also a tagged cache-like structure. Consequently, our approach in determining BHT size and organization closely follows the methodology used in determining the iBTB size and organization; we observe the number of misses in the BHT as a function of the number of program paths and their respective PIR values.

A BHT size and organization test is shown in Figure 11*A*. We consider two conditional spy branches: an always not taken *cSpyN*, and an always taken *cSpyT*. The *cSpyN* can be reached through any of N possible program paths (*PN1-PNN*). By controlling the placement of the setup branches *SB1-SBN*, we can control the PIR observed by the *cSpyN*. These setup branches are placed at regular distance D_G, such that $IP(SBi)=IP(SBi-1)+D_G$, where $D_G=2^k$, $p\le k\le q$, $i=2...N$. Consequently, the *cSpyN* may target up to N entries in the BHT. In this way, the number of misses in the global predictor becomes a function of D_G and N, similarly to the iBTB size test in Section 4.3.

To be able to observe a BHT miss as a misprediction event, we need to prevent the bimodal predictor from providing a correct prediction. This is achieved by using the *cSpyT*

branch and a specific program execution pattern. The spy branches are placed at a large distance D_S, so that both branches map into a single entry of the bimodal predictor, i.e., $IP(cSpyT)=IP(cSpyN)+D_S$. The test repeatedly traverses the paths as follows: $\{\{PT\}^v, PN1, \{PT\}^v, PN2, \ldots\{PT\}^v, PNN\}$. This execution pattern ensures that each instance of the *cSpyN* branch gets an incorrect prediction from the bimodal predictor, because the *cSpyT* branch moves the corresponding 2bC cell into a strongly taken state. Another issue is that due to the large distance D_S, the spy branches appear as a single branch to the loop predictor. To eliminate possible correct outcome prediction coming from the loop predictor, we add an always not taken branch *cSpyNH* that cannot be distinguished from *cSpyN* and *cSpyT* by the LPB. Finally, we vary the distance D_G and the number of paths N preceding the *cSpyN* branch, and observe the MPR measured as the number of mispredicted conditional branches at execution divided by the total number of possible mispredictions ($1/v$).

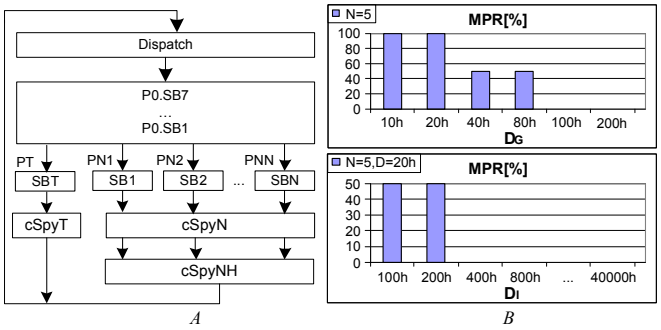


Figure 11. BHT Size and Organization Test (*A*) and test results (*B*).

First, we perform tests with N=3, 4 and $D=2^4$-2^{18} and observe the zero MPR. This implies that the global predictor is at least a 4-way structure. For $N=5$ and $D_G=2^8$ the zero MPR indicates that 5 instances of *cSpyN* map into two different sets (Figure 11*B* upper). When $D_G=2^8$ (100h), the setup branches *SB1-SB5* have the following address offsets: 0h, 100h, 200h, 300h, 400h; these correspond to the PIR values h0, 10h, 20h, 30h, and 40h. Thus, the IP bit 10 that sets the PIR bit 6 (PIR=40h) is the global predictor LSB index bit.

The original test from Figure 11*A* is not feasible for a large number of paths N. Instead we fix the distance D_G to map 4 spy outcomes into a single BHT set (e.g., $D_G=2^5$). Then we control the displacement of the fifth spy outcome, $D_I=2^i$, $8 \le i \le 18$. Figure 11*B* (lower) shows the MPR as a function of D_I. The results indicate that IP[18:10] is affecting the BHT index. This means that the global predictor is a 4-way structure with 512 sets, where PIR[5:0] is used as the tag and PIR[14:6] as the index field.

Lastly, we address the issue of which predictor hit has priority, the global or the loop. We use a test which includes a spy conditional branch, and we ensure that both the loop and the global predictor give hits. However, the loop predictor is controlled in such a way to give an incorrect prediction for the spy branch, while the global predictor delivers a correct prediction. The test results in no mispredictions, indicating that the global predictor hit overrides the loop predictor hit.

6.4 Bimodal Predictor Details

The size of the bimodal predictor is determined using the BHT test shown in Figure 11*A*. The number of paths leading to *cSpyN* is set to $N=5$. All paths map into a single BHT entry, causing collisions in the BHT. A miss in the BHT structure enables the prediction from the bimodal predictor. As *cSpyT* and *cSpyN* map into a single bimodal entry, the execution pattern will guarantee an outcome misprediction for all *PNi* paths. By changing the *cSpyT* address bit-by-bit, $IP(cSpyT)=IP(cSpyN)+2^{n+1}+2^b$, the *cSpyT* and *cSpyN* will map into two bimodal entries causing no outcome mispredictions. The results show that IP[11:0] are used as an index for the bimodal predictor. This indicates a 4096-entry bimodal predictor.

A similar test, where unconditional taken branch *uSpyT* replaces *cSpyT*, proves that unconditional branches are not allocated in the bimodal predictor (the same approach and results apply to the Global predictor). This confirms that the BTB has a type field that indicates the type of branch.

7. VALIDATION

To validate our findings we employ several tests targeting a single hypothesis (e.g., BTB Set and BTB Capacity tests). Due to space limitations, the paper describes only the tests that led us directly to the conclusions. The actual number of tests performed is much larger and is documented in a thesis [27] (e.g., Capacity-like and Set-like tests for all structures, tests targeting folding access functions, YAGS-like outcome predictors, etc.).

In order to further validate our effort, we have developed a PIN functional model of the Pentium M branch predictor. The model is based on our findings about the structure and operation of the branch predictor. We run several SPEC 2000/2006 benchmarks and compare the number of relevant events collected on the PIN model versus those collected using VTune on a real machine. We choose *gcc*, *vpr*, *mcf*, and *astar* benchmarks because they stress the branch predictor, having relatively large numbers of branches and mispredictions.

Program *Input* num. br.		MBIDEC [%]	MCBIE [%]	MIBIE [%]
gcc	Pent.M	1.05	4.03	23.78
scilab.i	PIN	0.95	3.81	23.52
8.6 bil.	Rel. Diff.	**9.14**	**5.29**	**1.12**
gcc	Pent.M	2.09	6.11	28.08
cccp.i	PIN	2.08	5.87	26.74
360 mil.	Rel. Diff.	**0.48**	**3.98**	**4.77**
gcc	Pent.M	0.94	3.16	27.21
expr.i	PIN	0.95	2.95	26.84
1.6 bil.	Rel. Diff.	**1.76**	**6.58**	**1.36**
gcc	Pent.M	1.45	4.74	27.19
cp-decl.i	PIN	1.46	4.51	26.31
790 mil.	Rel. Diff.	**0.90**	**4.98**	**3.24**
gcc	Pent.M	0.64	2.17	27.36
integrate.i	PIN	0.61	1.99	26.99
1.5 bil.	Rel. Diff.	**4.05**	**8.35**	**1.39**
vpr	Pent.M	~0	10.88	~0
test	PIN	~0	10.25	~0
290 mil.	Rel. Diff.	~0	**5.82**	~0
mcf	Pent.M	~0	10.40	~0
test	PIN	~0	10.41	~0
30 mil.	Rel. Diff.	~0	**0.12**	~0
astar	Pent.M	~0	18.12	~0
test	PIN	~0	18.22	~0
4.3 bil.	Rel. Diff.	~0	**0.55**	~0

Figure 12. Misprediction rates for *gcc* program with different input files.

Figure 12 shows three misprediction rates collected on the PIN model and a Pentium M machine. The results show a

rather small relative discrepancy: 1.5% on average for indirect branches (MIBIE), 4.5% on average for outcome prediction (MCBIE), and 6.9% on average for mispredictions at decoding (MBIDEC). We consider these results quite satisfactory, having in mind the imprecise nature of hardware counters, effect of other processes on the branch predictor on the real machine, as well as the timing issues not captured by our functional PIN model.

8. APPLICABILITY AND LIMITATIONS

The presented experiment flow is tailored for cache-based, PC-indexed and path-indexed target and outcome predictor structures. As long as a branch predictor consists of these building blocks and a processor supports measuring branch-related events, the flow could be used to reverse engineer the individual predictor blocks. If the predictor consists of completely different building blocks, the flow may not be sufficient or applicable at all. However, to the best of our knowledge, the majority of commercial branch predictors use the common predictor structures addressed in the paper.

A generalized experiment flow that can be applied to branch predictors beyond Pentium M includes the following three steps. Step #1 encompasses tests that establish presence or absence of known predictor structures, such as BTB, iBTB, and local and global outcome predictor and their histories. Step #2 encompasses tests for uncovering whether each predictor structure is tagged or non-tagged. Finally, Step #3 includes a number of tests to uncover interdependencies between different predictor structures. The exact number and type of these tests depends on findings in the previous steps and is difficult to formalize.

The Step #3 is an iterative process, where each test builds on the knowledge acquired in the previous tests in this step. The success depends on our ability to formulate and test (prove or disprove) a number of hypotheses spanning the predictor design space. If none of the hypotheses yields a conclusive finding, and we are unable to formulate any new hypotheses, we cannot complete the reverse engineering effort. However, the failed hypotheses at least may indicate what mechanisms or structures are likely not used.

In spite of somewhat ad-hoc nature of the design space exploration in Step 3, we were able to formulate and perform tests that yielded conclusive findings related to complex interdependencies and intricate mechanisms found in Pentium M, one of the most sophisticated commercial branch predictors. To demonstrate applicability of the presented experiment flow on other branch predictors, we applied it to the Intel's Core 2 Duo (Conroe core) and Pentium 4 predictor (Northwood core). The main findings from these experiments are as follows. The branch predictor in the Core 2 Duo is very similar to the one described in this paper. However, we have been able to identify that the iBTB and the global predictor use a slightly different hash access function. The Pentium 4's outcome predictor has the following characteristics: (a) the bimodal predictor is tagged and attached to the BTB, (b) the global predictor is not tagged, and (c) the global outcome predictor appears to be an Agree-like predictor. The Pentium 4 BTB characteristics are described in [21].

9. CONCLUSIONS

This paper introduces reverse engineering flows for modern branch predictor structures. The proposed flows are successfully demonstrated on a branch predictor unit found in Intel Pentium M processor. The flows encompass a number of experiments targeting various predictor structures used for branch target address prediction (BTB, iBTB) and branch outcome prediction (bimodal, loop, and global predictors). Using these experiments, we have been able to determine not only structural parameters of predictor units, but their relationship, update mechanisms, and access functions. Figure 13 illustrates the main hardware structures of the Pentium M branch predictor uncovered by the proposed reverse engineering flows.

We believe that the presented experimental flows for determining program paths, hash access functions, size and organization of predictor structures can be easily adapted and extended for reverse engineering of any cache-based branch predictor structures. In addition, this research can serve as a starting point in developing a software tool for architectural exploration that will automatically generate tests and analyze the results. Finally, the PIN model of the Pentium M branch predictor can help future branch predictor research efforts.

10. REFERENCES

[1] J. K. E. Lee and A. J. Smith, "Branch prediction strategies and branch target buffer design," *IEEE Computer,* vol. 21, pp. 6-22, 1984.

[2] P. Chang, et al., "Target Prediction for Indirect Jumps," in *24th ISCA*, 1997, pp. 274-283.

[3] K. Driesen and U. Hölze, "Accurate Indirect Branch Prediction " in *25th ISCA*, 1998, pp. 167-168.

[4] L. Rappoport, et al., "Method and system for branch target prediction using path information," U.S. Patent 6601161, 2003.

[5] J. E. Smith, "A Study of Branch Prediction Strategies," in *8th ISCA*, 1981, pp. 135–148.

[6] S. Gochman, et al., "The Intel Pentium M Processor: Microarhitecture and Performance," *Intel Technology Journal,* vol. 07, pp. 21-36, 2003.

[7] S. Gochman, et al., "Introduction to Intel Core Duo Processor Architecture," *Intel Technology Journal,* vol. 10, pp. 89-98, 2006.

[8] R. E. Kessler, et al., "The Alpha 21264 Microprocessor Architecture," *IEEE Micro,* vol. 19 1999

[9] A. Seznec, et al., "Design Tradeoffs for the Alpha EV8 Conditional Branch Predictor," in *29th ISCA*, 2002, pp. 295-306.

[10] T.-Y. Yeh and Y. N. Patt, "Alternative Implementations of Two-level Adaptive Branch Predictions," in *19th ISCA*, 1992, pp. 124-134.

[11] S.-T. Pan, et al., "lmproving the accuracy of dynamic branch prediction using branch correlation," in *5th ASPLOS*, 1992, pp. 76-84.

[12] E. Sprangle, et al., "The Agree Predictor: A Mechanism for Reducing Negative Branch History Interference," in *24th ISCA*, 1997, pp. 284-291.

[13] C.-C. Lee, et al., "The Bi-Mode Branch Predictor," in *30th MICRO*, 1997, pp. 4-13.

[14] P. Michaud, et al., "Trading Conflict and Capacity Aliasing in Conditional Branch Predictors," in *24th ISCA*, 1997, pp. 292-303.

[15] A. Eden and T. Mudge, "The Yags Branch Prediction Scheme," in *31th MICRO*, 1998, pp. 69-77.

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

[17] A. Seznec, "The O-GEHL branch predictor," *The 1st JILP Championship Branch Prediction Competition (CBP--1), in conjunction with MICRO-37,* 2004.

[18] A. Seznec, "The L-TAGE Branch Predictor," *Journal of Instruction Level Parallelism,* vol. 9, pp. 1-13, 2007.

[19] D. Jimenez and C. Lin, "Dynamic Branch Prediction with Perceptrons," in *7th HPCA*, 2001, pp. 197-206.

[20] D. Jimenez, "Fast Path-Based Neural Branch Prediction," in *36th MICRO*, 2003.

[21] M. Milenkovic, et al., "Microbenchmarks for determining branch predictor organization," in *Software Practice and Experience*. vol. 34, 2004, pp. 465-487.

[22] D. Jimenez, "Code Placement for Improving Dynamic Branch Prediction Accuracy," in *Proceedings of the 2005 PLDI*, 2005, pp. 107-116.

[23] C.-K. Luk, et al., "Pin: Building Customized Program Analysis Tools with Dynamic Instrumentation," in *PLDI*, 2005, pp. 190 - 200.

[24] Intel VTune™ Performance Analyzer, http://www.intel.com

[25] Intel® Architecture Software Optimization Reference Manual, http://www.intel.com/products/processor/manuals/

[26] B. D. Hoyt, et al., "Method and apparatus for implementing a set-associative branch target buffer," U.S. Patent 5574871, 1996.

[27] V. Uzelac, "Microbenchmarks and Mechanisms for Reverse Engineering of Modern Branch Predictor Units," Masters Thesis, University of Alabama in Huntsville, 2007.

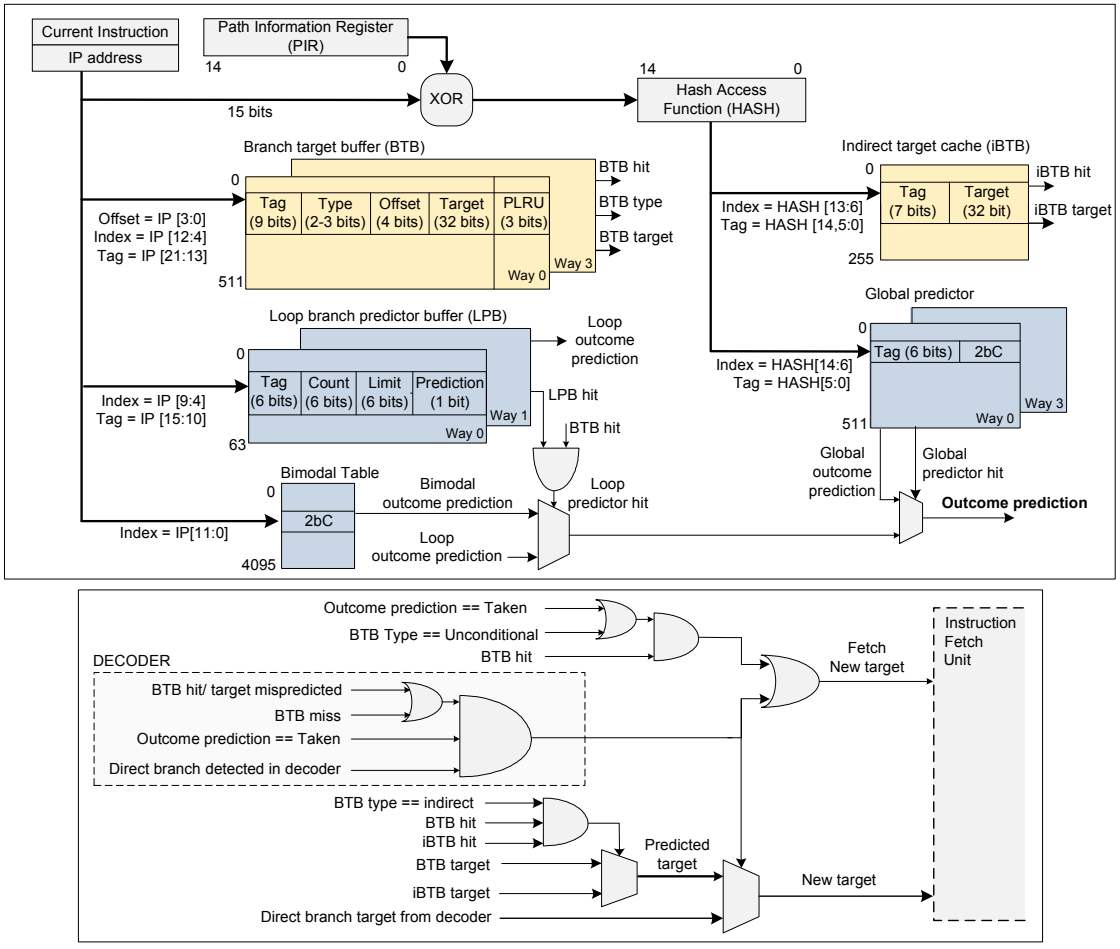


Figure 13. Putting it all together: Pentium M branch predictor unit.