SOFTWARE—PRACTICE AND EXPERIENCE
Softw. Pract. Exper. 2004; (in press) (DOI: 10.1002/spe.572)

Microbenchmarks for determining branch predictor organization

Milena Milenkovic, Aleksandar Milenkovic[*,†] and Jeffrey Kulick

*Electrical and Computer Engineering Department, The University of Alabama in Huntsville,
301 Sparkman Drive, Huntsville, AL 35899, U.S.A.*

SUMMARY

In order to achieve an optimum performance of a given application on a given computer platform, a program developer or compiler must be aware of computer architecture parameters, including those related to branch predictors. Although dynamic branch predictors are designed with the aim of automatically adapting to changes in branch behavior during program execution, code optimizations based on the information about predictor structure can greatly increase overall program performance. Yet, exact predictor implementations are seldom made public, even though processor manuals provide valuable optimization tips.

This paper presents an experimental flow with a series of microbenchmarks that determine the organization and size of a branch predictor using on-chip performance monitoring registers. Such knowledge can be used either for manual code optimization or for design of new, more architecture-aware compilers. Three examples illustrate how insight into exact branch predictor organization can be directly applied to code optimization. The proposed experimental flow is illustrated with microbenchmarks tuned for Intel Pentium III and Pentium 4 processors, although they can easily be adapted for other architectures. The described approach can also be used during processor design for performance evaluation of various branch predictor organizations and for testing and validation during implementation. Copyright © 2004 John Wiley & Sons, Ltd.

KEY WORDS: compiler optimizations; microbenchmarks; branch predictor; performance monitoring

INTRODUCTION

Improved performance of today's microprocessors is not only due to the increase in the operating frequency, but also due to the increase in processor complexity in every new generation. Compilers must keep up with new processor features, such as extended instruction sets, pipelining,

[*] Correspondence to: Aleksandar Milenkovic, Electrical and Computer Engineering Department, The University of Alabama in Huntsville, 301 Sparkman Drive, Huntsville, AL 35899, U.S.A.
[†] E-mail: milenka@ece.uah.edu

Received 4 December 2002
Revised 28 July 2003
Accepted 28 July 2003

multiple-level cache hierarchy, instruction-level parallelism, and branch prediction, exploiting new optimization possibilities. Although compilers for new processors do include some advanced optimization features, for instance the Intel C++ Compiler [1], future compilers must be even more aware of the underlying architecture. Currently, program developers must specifically set compiler switches that notify the compiler which architecture to optimize the code for. The Intel processors also include CPUID (CPU identification instruction) that provides information about some of the processor features, such as cache and TLB (Translation Look-aside Buffer) [1]. Another way of extracting the required information is to perform a series of microbenchmarks that experimentally explore the architectural properties. For instance, a program can automatically determine memory hierarchy parameters [2,3]. This kind of program can be incorporated into future compilers: the compiler would first assess any relevant architectural parameters of a processor and then optimize the code according to the obtained parameter values. The information about the underlying architecture can also be applied to manual code optimizations, such as a blocking transformation that improves code spatial locality [4].

The successful resolution of conditional branches is a crucial performance issue in modern superscalar processors. When a conditional branch enters the execution pipeline, all instructions following the branch must wait for branch resolution. A common solution to this problem is speculative execution: the branch outcome and/or its target are dynamically or statically predicted, so the execution can go on without stalling. If a branch is mispredicted, speculatively executed instructions must be flushed and their results discarded, thus wasting a significant number of processor clock cycles. For example, the Pentium 4 has a misprediction penalty of 20 clock cycles [5], and future processors may have even higher penalties, up to 50 clock cycles [6], since deep pipelines are necessary for achieving very high clock frequencies.

With static branch prediction, a branch outcome is predicted statically at compile time using the branch type, branch direction, and/or profiling information. Although static prediction may work well for some applications, dynamic prediction solves more general cases, since it is able to automatically adapt to changes in branch behavior during program execution. Predictor size and organization may limit its ability to give a correct prediction. If the compiler/developer is aware of the intricacies of the branch predictor, the code can be optimized to overcome some limitations, and consequently overall program performance increases.

Modern processors, such as the Intel Pentium III (P6 architecture) and the Pentium 4 (NetBurst architecture), include some form of dynamic branch prediction mechanisms, but information about exact predictor organization is rather scarce. On the other hand, almost all modern processors include performance-monitoring registers that can count several branch-related events, and quite powerful tools for easy access to these registers are available [7,8].

This paper presents an experiment flow that uncovers branch predictor organization using performance-monitoring registers and illustrates how such knowledge can improve code optimization. A set of 'spy' microbenchmarks tests the existence and/or value of particular branch predictor parameters: the use of global and/or local branch history, the number of history bits, and the predictor size and organization (http://www.ece.uah.edu/∼lacasa/). Another application of the proposed experimental flow is for testing and validation of the branch predictor design during processor implementation. The microbenchmarks can also be used in research looking for better branch predictors.

The proposed experimental flow is illustrated on Pentium III and Pentium 4 processors, though only minor modifications are necessary to adapt the proposed microbenchmarks to other processor

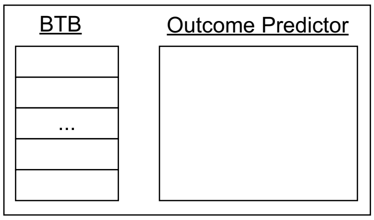


Figure 1. General branch predictor scheme.

architectures. The results indicate that Pentium III has a local branch predictor with four branch history bits, and the Pentium 4 uses a global branch predictor with 16 history bits. The experiments also determine the organization of the branch target buffer (BTB) and the address bits used to access it.

The next section provides an overview of dynamic branch prediction, followed by examples of predictor-aware code optimizations. A description of the experimental environment sets the stage for a detailed explanation of the proposed experimental flow. Finally, the results of the experiments for observed architectures are presented.

DYNAMIC BRANCH PREDICTION

No matter how complex a branch predictor is, it can be described by a variation of the general scheme (Figure 1); this consists of two major parts: a BTB for the prediction of branch targets, and an outcome predictor for the prediction of branch outcomes.

The BTB is a cache structure, where a part of the branch address is used as the cache index, and the cache data is the last target address of that branch. More complex BTBs can hold more than one possible target address and some type of mechanism to choose which target instructions should be speculatively executed. Some implementations can also store target instructions, and even whole target basic blocks [4]. The prediction of branch outcomes can be coupled or decoupled with the BTB: if the outcome predictor and the BTB are coupled, only branches that hit in the BTB are predicted, while a static prediction algorithm is used on a BTB miss. If the BTB is decoupled from the outcome predictor, all branch outcomes are predicted using the outcome predictor.

Dynamic prediction of a branch outcome is based on the state of a finite-state machine, which is usually a two-bit saturating counter [9], depicted in Figure 2. In the states *strongly taken* and *weakly taken* a branch is predicted as taken, and it is predicted as not taken in the other two states, *weakly not taken* and *strongly not taken*.

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 a part of it with a branch history register (BHR). The BHR is a shift register that keeps the history of N most recent branch outcomes, where N represents the number of bits of the shift register [10,11]. The BPT index function is usually a concatenation or exclusive *OR* of the branch

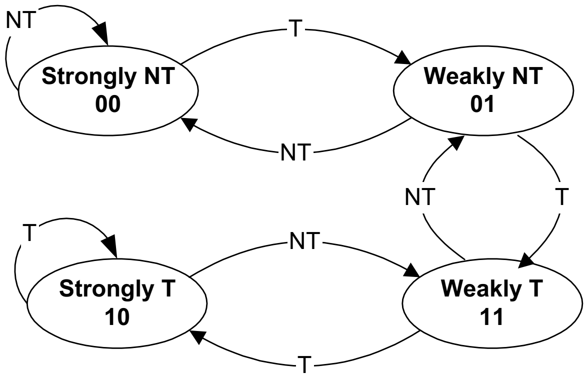


Figure 2. Two-bit saturating counter: T = taken branch; NT = not taken.

address and the corresponding BHR. Based on the type of recorded branch history, the predictors can be global and local. Global two-level predictors benefit from correlations between subsequent branches in the program execution flow (Figure 3(a)), while local predictors are based on correlation between subsequent executions of the same branch (Figure 3(b)).

In order to further reduce the number of branch mispredictions in wide-issue superscalar processors, more advanced mechanisms have been proposed, for example, hybrid branch predictors. Hybrid branch predictors can include both global and local prediction mechanisms, as well as some other prediction schemes, for example, specialized loop predictors [12]. Instead of exploiting the correlation between outcomes of the last N branches (pattern based), the dynamic branch predictor can use the information of the path to the current branch (path based) [13]. The path history register stores address bits from each of the most recently executed P branches, thus making the prediction path dependent. One predictor can combine both pattern-based and path-based approaches. Specialized predictors can handle some special branch types, such as returns and loops.

In order to reduce the number of mispredictions, branch predictors are getting larger and more complex. However, code optimizations are still vital for processor performance, since the large number of pipeline stages and superscalar fetch/decode make modern processors more sensitive to branch mispredictions.

EXAMPLES OF BRANCH OPTIMIZATION BY ARCHITECTURE-AWARE COMPILERS

The following three examples illustrate how knowledge about the underlying branch predictor structure can improve code optimization. The first example deals with processor architectures with global branch predictors, and it is inspired by the code generation guidelines explained in Sun's *UltraSparc User's Manual* [14]. The second example shows a possible optimization for local branch predictors, and it is based on tips given in one of the Intel Pentium III optimization guidelines [15]. Finally, the last

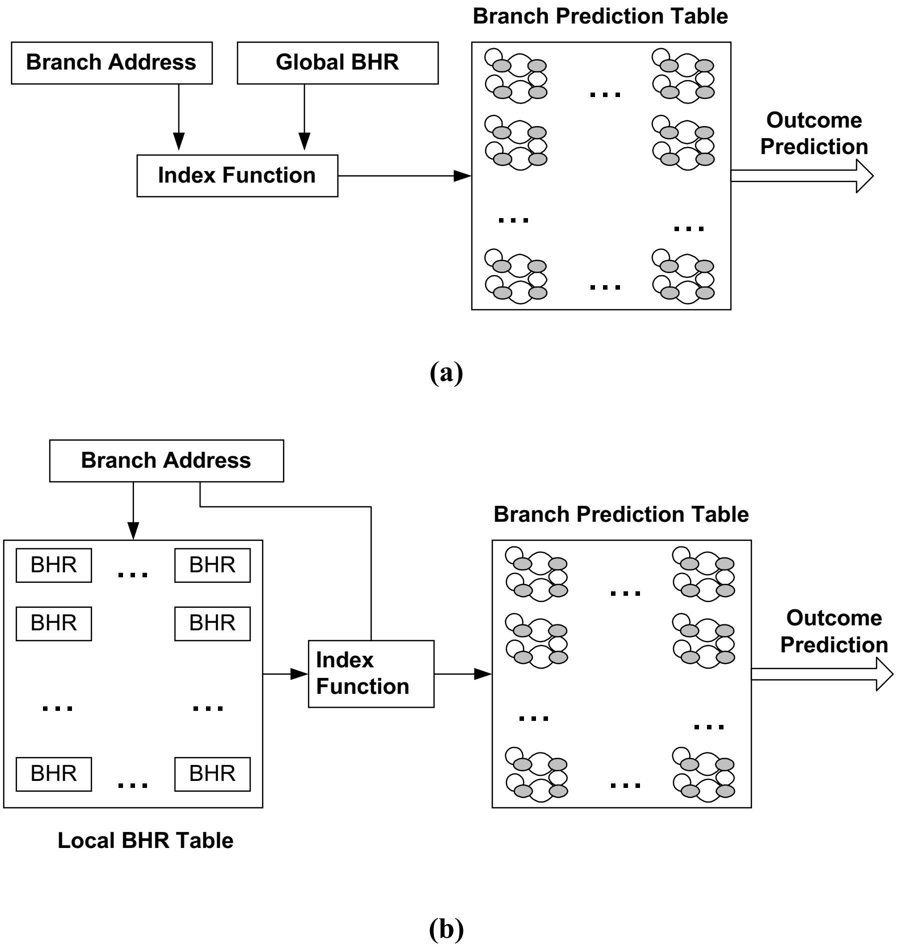


Figure 3. Global (a) and local (b) two-level branch predictor.

example shows how knowledge about the size and organization of the branch predictor structure can reduce branch interference. Actual implementation of an architecture-aware compiler, which is outside the scope of this paper, must also take into account other performance factors, such as possible cache miss increases due to changes in the executed code length.

Let us first consider a processor with a global branch predictor that uses N global history bits. This predictor is able to correctly predict the outcome of a branch correlated with up to N previously executed branches, while correlations longer than N cannot be detected. If the outcome of a particular

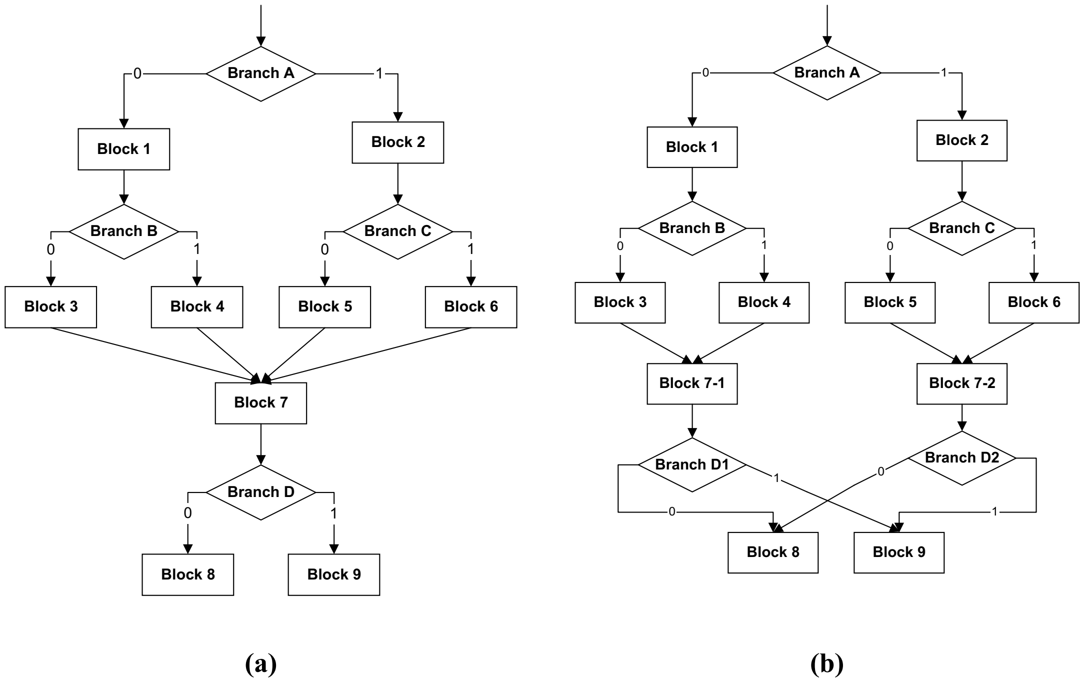


Figure 4. Original (a) and optimized (b) code structure: branch D depends on two previously executed branches, and the branch predictor is global with one history bit.

branch depends on more than N previous branches, the compiler can split the code and duplicate branches as necessary, replacing a long branch correlation with several shorter ones.

Figure 4(a) shows the control flow for one such scenario, where the branch D outcome is the *AND* function of the outcomes of two previously executed branches: A and B, or A and C (Table I). In this example the branch predictor uses only one bit of global history ($N = 1$). Since the predictor is able to 'remember' only one previous branch, it cannot distinguish between the branch histories 01, when D is not taken, and 11, when D is taken. The BPT index function takes as arguments the branch D address and one history bit, so the same BPT cell is accessed in both cases. Assuming a two-bit saturating counter, one or both corresponding branch D outcomes are mispredicted, depending on the counter start state (Table I(a)). In the other two cases—that is, branch histories 00 and 10—one history bit is enough for a correct prediction, since the outcome of branch D is equal to the outcome of the previous branch. Figure 4(b) shows the code modified by an architecture-aware compiler. Block 7 code and branch D are duplicated to blocks 7-1 and 7-2 and branches D1 and D2, respectively, where branch D1 is on the not taken path of branch A, and branch D2 is on the taken path. Now the outcome of branch D1 is always 0, and the outcome of branch D2 is equal to the branch C outcome (Table I(b)).

Table I. Branch outcome scenarios for original (a) and optimized code structure (b). In the branch predictor with one global history bit, a 2-bit saturation counter with starting state WT (weakly taken) mispredicts both bold outcomes and with other starting states, mispredicts one of them.

(a)

Branch A	Branch B/C	Branch D
0	0	0
0	1	**0**
1	0	0
1	1	**1**

(b)

Branch A	Branch B	Branch D1
0	0	0
0	1	0
Branch A	Branch C	Branch D2
1	0	0
1	1	1

Since branches D1 and D2 are located at different addresses, they have separate predictor entries for the same branch histories, so both are correctly predicted by separate two-bit counters. In all cases we assume that this control flow is a part of a loop, and the predictor can be dynamically 'trained'.

A similar optimization can be done for processors with a local branch predictor. Let us consider a processor with a local branch predictor using N bits of local branch history. The outcome of a loop condition branch can be correctly predicted if the loop does not have more than N iterations. Loops with more than N iterations can be unrolled, so the existing predictor can predict each unrolled loop. The compiler should perform loop unrolling if one such loop belongs to a critical portion of the code that executes frequently, and if it should be unrolled relatively few times. Figure 5(a) shows an example of code where the inner loop executes eight times and then exits, while the outer loop executes 1 million times. If this code executes on a processor with a local branch predictor using four bits of local history and two-bit saturation counters, the inner loop condition branch is mispredicted once every nine goes, thus having 1 million branch mispredictions. After eight loop iterations, the two-bit counter for the 1111 history will be in the *strong taken* state. At the loop exit, four bits of local history are the same as in the previous four iterations (1111); hence the exit case uses the same BPT cell as others, and cannot be predicted correctly. An architecture-aware compiler can unroll the inner loop, and in this example twice is enough (Figure 5(b)). Both new loop branches can be correctly predicted with four bits of local history: now the four-bit branch history at the exit is unique with pattern 1111, so the exit case is mapped to the separate BPT entry. The number of lost execution cycles due to mispredicted branches is significantly reduced.

```
for (i=0;i<1000000;++i){
...
  //original inner loop
  for (j=0;j<8;++j){
  ...
  }
...
}
```

(a)

```
for (i=0;i<100000;++i){
...
  //inner loop unrolled twice
  for (j=0;j<4;++j){ ...
  ...
  }
  for (j=0;j<4;++j){
  ...
  }
...
}
```

(b)

Figure 5. Original (a) and optimized (b) C code: original inner loop depends on its eight previous executions.

```
              ...
addr512:      jle l1
              ...
l1:           ...
              ...
addr1024:     jle l2
              ...
```

(a)

```
              ...
addr512:      jle l1
              ...
l1:           ...
              ...
              noop
addr1025:     jle l2
              ...
```

(b)

Figure 6. Original (a) and optimized (b) assembly code: branches mapping to the same predictor entry.

If the compiler is aware of predictor size and organization (i.e. the number of ways and sets and function used for the index), it can prevent branch interference in the critical portions of the code. For example, inserting a required number of noop instructions in the code can separate branches that map to the same predictor entry. Figure 6(a) shows an example of two branches mapping to the same BTB entry, where it is assumed that branches at a distance of 512 bytes access the same BTB cell. If the BTB is always updated, both branch targets will be mispredicted, one always replacing the other. If both branches belong to a frequently executed portion of the code, an architecture-aware compiler can insert a *noop* instruction before one of the branches, thus preventing interference in the BTB (Figure 6(b)).

An architecture-aware compiler can encompass these mechanisms and other similar techniques. If applied to critical portions of the code, these optimizations can significantly increase performance. However, to be able to do so, the compiler needs to know the details about the branch predictor organization: the use of global, local, or both types of branch history; the number of history bits; and the BTB size and organization.

EXPERIMENTAL ENVIRONMENT

This paper focuses on the widely used Intel P6 (Pentium III) and NetBurst (Pentium 4) architectures, although the proposed microbenchmarks can be applied, with some modifications, to other microprocessor architectures. For both P6 and NetBurst architectures, Intel sources [1,5,15] do not provide exact descriptions of the implemented branch predictors. Rather, they provide the exact number of BTB entries and several hints on program optimization that indicate some outcome predictor parameters. If a branch is not in the BTB, a static branch prediction is used, which means that the BTB and outcome predictor are coupled. The static prediction mechanism predicts backward conditional branches as taken, and forward branches as not taken. A return address stack of a known size predicts return addresses.

The P6 optimization reference manual states that the prediction algorithm includes pattern matching and can track up to the last four branch directions per branch address [15], most probably meaning that the P6 branch predictor has a local history component with four history bits. The P6 BTB has 512 entries.

In the NetBurst architecture implemented in the Pentium 4, Intel claims to use a new prediction algorithm, 33% better than in the P6. One of the assembly/compiler coding rules for the Pentium 4 states that frequently executed loops with a 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 $16/N$ [1]. This rule indicates that the Pentium 4 uses a 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 [5], which stores and delivers sequences of traces, built from decoded instructions according to 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 a 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 [15], but apart from trace cache size (12K micro-ops), and the trace cache line size (6 micro-ops), Intel does not disclose many details about its implementation. This work considers only the front-end BTB, and more experiments with the trace cache component can be found in [16].

Both P6 and NetBurst architectures have several performance counters, that are able to measure various branch-related events, such as the number of retired branches, including unconditional branches, and the number of mispredicted branches, using event-based sampling. Since the number of branches depends on a particular microbenchmark and the number of times it executes, throughout the paper the MPR (misprediction ratio) is often used instead of the number of mispredicted branches. The MPR is the number of mispredicted branches divided by the total number of conditional branch instructions.

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 overflow after a specific number of events have been counted. In this research the Intel VTune Performance Analyzer version 5.0 was used for the configuration and access of performance counters. Performance counters on most non-Intel architectures, as well as Intel processors, can be accessed using the freeware PAPI (performance application programming interface) tool, developed at the University of Tennessee [8].

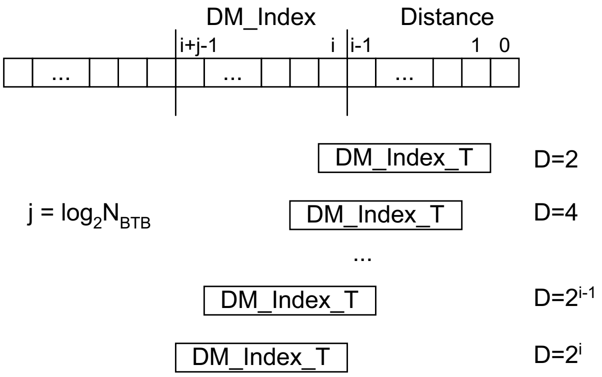


Figure 7. BTB size and organization: varying the distance.

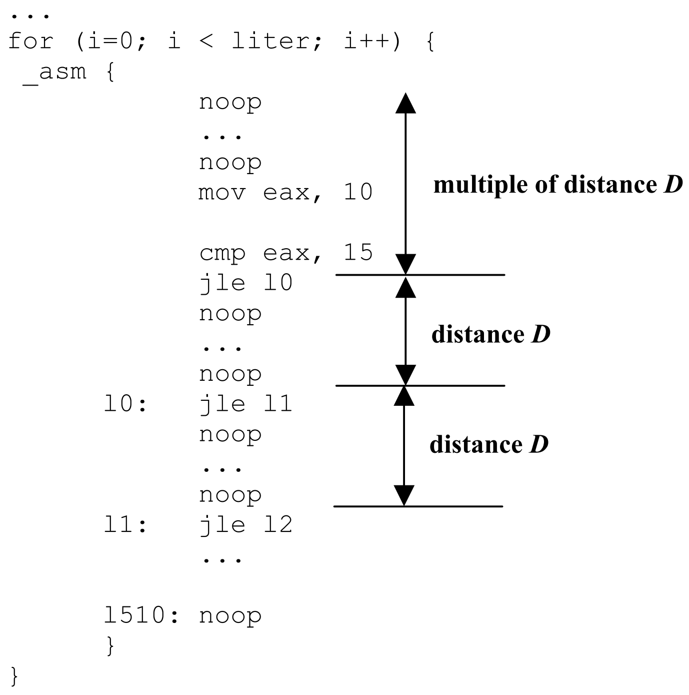
All test benchmarks are compiled using a Microsoft Visual Studio 6.0 C compiler, with disabled optimization, preventing the compiler optimizations from changing the order and number of conditional branches. For experiments with a relatively large number of branches, we have also developed programs to generate benchmarks to our assembly specifications. In order to get reliable values for the performance counters, the execution time of the monitored code must be significantly larger than that of the interrupt service routine. Therefore, the test code is placed within a loop that executes a relatively large number of times.

EXPERIMENTAL FLOW

The experimental flow consists of two groups of experiments targeting the BTB and the outcome predictor. BTB experiments uncover the BTB organization and address bits used as an index, and outcome predictor experiments determine the existence of local and global prediction components, and the length of the corresponding history registers.

BTB experiments

The Intel documentation for P6 does not describe its BTB organization (i.e. whether it is direct mapped or set associative), and the degree of associativity. One way to determine the number of ways and the address bits used as the BTB index is to run a set of microbenchmarks varying the address distance D between the branch instructions (Figure 7). Each microbenchmark has $N_{\mathrm{BTB}} - 1$ conditional branches in a loop, which makes a total of N_{BTB} conditional branches, where N_{BTB} is the number of BTB entries. These conditional branches are always taken, so they are mispredicted by the static algorithm if not present in the BTB. Figure 8 shows the fragment of the microbenchmark code.



```
...
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
       }
}
```

Figure 8. Benchmark for testing BTB organization.

For a ‘fitting’ distance D_F, when all branches under consideration can fit in the BTB, the number of mispredictions is close to zero, i.e. the performance counter counts only a negligible number of mispredictions. If there is only one distance D_F, then the BTB is direct mapped, and the address bits used as the BTB index are Addr[$i + j - 1 : i$] (Figure 7). If there are exactly two distances D_F, the BTB is two-way set associative, and bits used as the index are Addr[$i + j - 2 : i$]. Similarly, if there are exactly three distances D_F, the BTB is four-way set associative. In general, if there are m ‘fitting’ distances, the BTB is 2^{m-1}-way set associative, and the index bits are Addr[$i + j - m : i$].

There is one exception to this experiment, and that is the unlikely border case in which low-order address bits are used as the index, i.e. Addr[$j - 1 : 0$]. For any degree of associativity, this BTB will have only one ‘fitting’ distance $D_F = 1$. In this case, an additional experiment is necessary to establish the number of BTB ways. Instead of finding the number of branches that would fill the whole BTB, this additional experiment finds the number of branches that fill a BTB set, and a distance D_S such that those branches map into the same set. If there are more branches than ways mapping into the same set, the MPR will be high. The same number of branches at some other distance might also produce a high MPR, if there are sets where the number of competing branches is larger than the number of

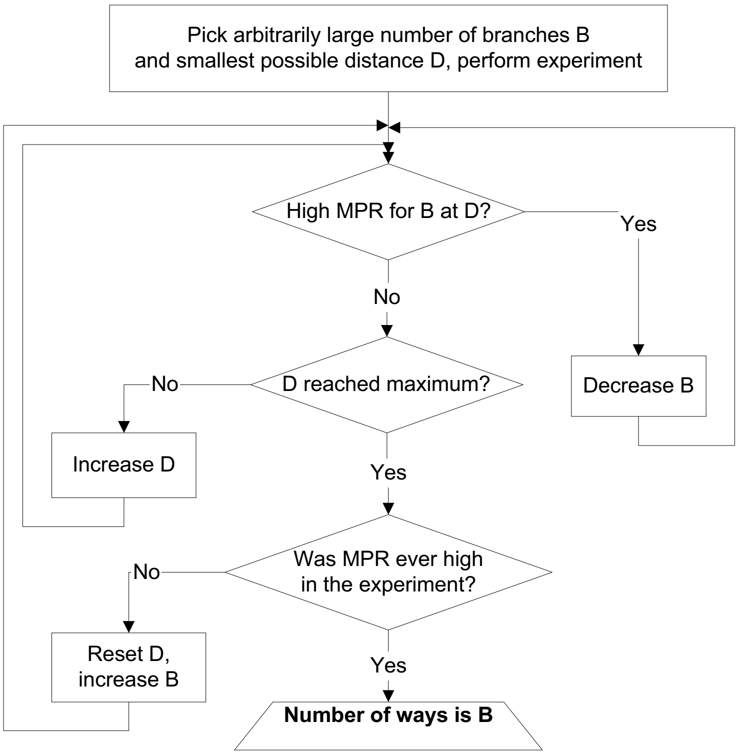

Pick arbitrarily large number of branches B and smallest possible distance D, perform experiment
High MPR for B at D?
Yes
No
Decrease B
D reached maximum?
No
Increase D
Yes
Was MPR ever high in the experiment?
No
Reset D, increase B
Yes
Number of ways is B

Figure 9. Searching the number of branches that fill a cache set.

the BTB ways. For example, 16 branches mapping into a four-way set will have a high MPR, as well as 16 branches mapping into two four-way sets. If the number of branches is equal or less than the number of ways, they do not collide at any distance. The corresponding microbenchmark is similar to the one described for the previous experiment (Figure 8), but in general, it requires a larger number of runs to establish correct BTB organization since both the number of branches fitting in the set and the branch distance must be varied. Figure 9 shows the search process for the correct number of BTB ways. The algorithm first picks an arbitrarily large number of branches and sets them at the smallest possible distance D. If the MPR is low, the distance is increased and the experiment is repeated. When a high MPR is reached, it means that B branches collide in the same set, and the number of branches is decreased. The process stops when the maximum distance is reached, unless the number of branches picked at the beginning is smaller than the number of ways. In this case, the MPR is low throughout the series of experiments, and the number of branches B should be increased.

A variation in the microbenchmark shown in Figure 8 can be used to verify the assumption about the number of BTB entries, by increasing the number of branches for the ‘fitting’ distances. For example, if the actual number of BTB entries is twice as large as the assumed value, and the previous experiments have found m distances D_{F}, the set of experiments with the actual number of entries should find $m - 1$ such distances; i.e. the BTB would be 2^{m-2}-way set associative. In general, if the actual number of BTB entries is 2^n times greater than the assumed value, the experiments should find $m - n$ ‘fitting’ distances. If the experiments with a larger number of conditional branches do not find any such distances, the assumption about the size is correct.

Outcome predictor experiments

The set of experiments for uncovering the characteristics of outcome predictor component (Figure 10) is devised in such a way that most of the branches are easily predictable; i.e. a few ‘spy’ branches generate the misprediction rate for the whole microbenchmark. The microbenchmarks should be carefully tuned to avoid any interference between different branches in the branch predictor. Since the BTB organization is known from the previous set of experiments, it is possible to check the assembly code for branch interference and insert dummy instructions if necessary.

Step 1

This step determines the maximum length of a local history pattern that the predictor can correctly predict, for just one branch in the loop, i.e. the ‘spy’ branch. The loop condition branch has just one outcome not taken, when it exits; otherwise it is taken. After enough iterations, misprediction due to this branch is negligible. For the ‘spy’ branch, different repeating local history patterns of length *LSpy* can be used; however, the simplest pattern has all its outcomes the same except for the last one. If ‘1’ means that the branch is taken, and ‘0’ not taken, the local history patterns are 1111...110 and 0000...001.

Figure 11(a) shows the code for the Step 1 experiment, and Figure 11(b) shows a fragment of the corresponding assembly code for Intel ×86 architecture, when the pattern length $LSpy = 4$. Note that the ‘spy’ branch *if* $((i\%4) == 0)$ is compiled as *jne* (*jump short if not equal*), so the local history pattern for this branch is 1110. The fragment does not show the loop, which is compiled as the combination of instructions *jae* (*jump short if above or equal*) at the beginning of the loop and unconditional *jmp* at the end, so the *jae* outcome is 0 until the loop exit.

The MPR is low for all *LSpy* pattern lengths up to a certain number L, and then the outcome predictor is not able to predict the last outcome of the ‘spy’ branch. That is, for each pattern of length $LSpy > L$, the ‘spy’ branch is mispredicted once in *LSpy* times. However, this experiment does not tell whether the predictor has a local prediction component with history registers of length $L - 1$, or a global predictor component with a history register of length $2^*(L - 1)$. Two cases must be considered, as depicted in Figure 12.

(i) The outcome predictor has a local history component, so any local pattern of the length L can be correctly predicted, including the ‘spy’ pattern.

(ii) The outcome predictor has a global history component, so the local history pattern 11...10 of the ‘spy’ branch with $L - 1$ 1s is correctly predicted, but by using the global history of

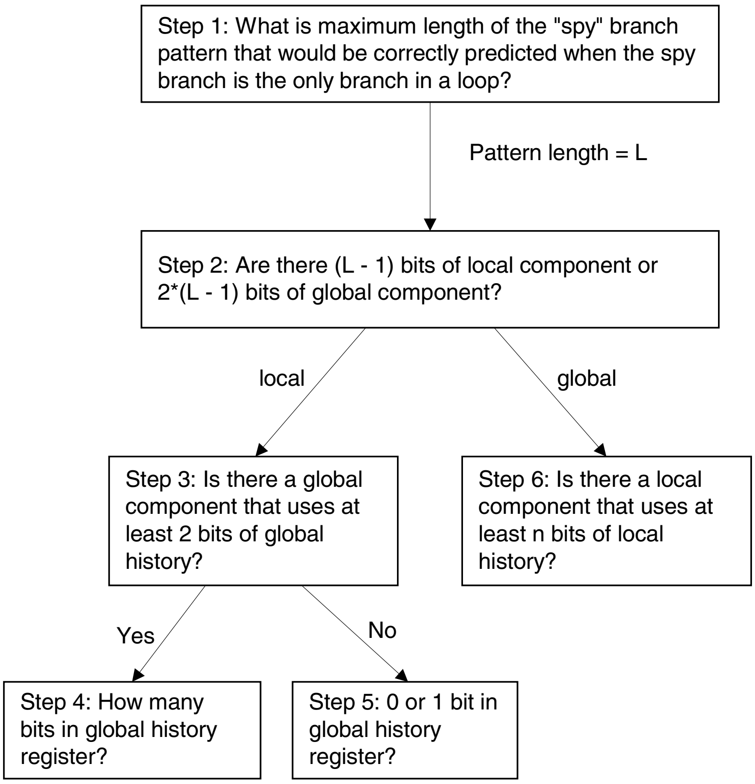

Step 1: What is maximum length of the "spy" branch pattern that would be correctly predicted when the spy branch is the only branch in a loop?
Pattern length = L
Step 2: Are there (L - 1) bits of local component or 2*(L - 1) bits of global component?
local
global
Step 3: Is there a global component that uses at least 2 bits of global history?
Step 6: Is there a local component that uses at least n bits of local history?
Yes
No
Step 4: How many bits in global history register?
Step 5: 0 or 1 bit in global history register?

Figure 10. Experimental flow for the 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%LSpy) ==0) a=0; //spy branch
 }
}
```

(a)

```
; 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:
```

(b)

Figure 11. Step 1 microbenchmark and the assembly fragment, when *LSpy* = 4.

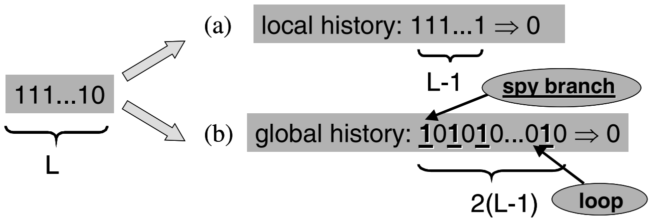


Figure 12. Two possible cases for the maximum predictable pattern length L in Step 1.

```
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 13. Step 2 microbenchmark.

previous $2^*(L-1)$ branches. Since the microbenchmark has just the loop condition and the ‘spy’ branch, all predictions are correct if all relevant local history fits into the global history register. For example, just before execution of the ‘spy’ branch with 0 outcome, the content of the global history register is **1**0 **1**0**1**0. . . **1**0, where the underlined and bold 1s are outcomes of the ‘spy’ branch, and the 0s are the outcomes of the loop condition branch.

Step 2

Step 2 verifies which one of these two hypotheses matches the predictor under test. If the conditional branch in the loop is preceded by $2^*(L-1)$ ‘dummy’ conditional branches, having always the same outcome, then no local ‘spy’ history is present in the global history register when the ‘spy’ branch prediction is generated. One example for the ‘dummy’ branch is *if* $(i < 0)\ a = 1$ (Figure 13). If the MPR still stays low, the correct hypothesis is (i); i.e. the predictor has a local history component. The experiment flow proceeds to Step 3, which determines whether the outcome predictor also has a global history component. If the MPR increases, the correct hypothesis is (ii); i.e. the predictor has a global history component. In this case, the experiment flow proceeds to 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%L1) == 0) a=1;
      else a=0;
      if ((i%L2) == 0) b=1;
      else b=0;
      if ((a*b) == 1) c=1; // spy branch
 }
}
```

Figure 14. Step 3 microbenchmark.

Step 3

The Step 3 microbenchmark has three conditional branches in a loop, where the first two have predictable patterns 11. . . 10 of different pattern lengths $L1$ and $L2$, such that $L1, L2 \leq L$, and the smallest common denominator for $(L1, L2)$ is greater than L. For example, if $L = 4$, the values for $L1, L2$ may be $L1 = 3$ and $L2 = 2$. The third branch, the 'spy', is correlated with the first two, and is not taken when both previous branches are not taken (Figure 14). The pattern of the third branch is 11. . . 10, and its length is greater than L, so it cannot be predicted by the local component, while both the first and second branch will be correctly predicted. That is, the local predictor can correctly predict all 1 outcomes of the 'spy' branch, but a global predictor with at least two history bits is needed for a correct prediction of the 'spy' 0 outcome. Hence, if the MPR is low, the number of global history bits is equal to or greater than 2, and the next step is Step 4. Otherwise, there is no global component or there is just one bit of global history, and the next step is Step 5.

Step 4

This step determines the length of the global history register. The simplest way is to insert 'dummy' conditional branches (e.g. pattern 111. . . 11) before the 'spy' conditional branch. The 'spy' branch is not predicted correctly if the number of 'dummy' branches is greater than the number of *global history bits*−2, so the number of global history bits is determined by varying the number of 'dummy' branches (Figure 15).

Step 5

The Step 5 microbenchmark has just two conditional branches in the loop, where the first one has the local history pattern 111. . . 110 of length $L3 > L$, and the second one has the same outcome as the first, as shown in Figure 16. Since it is known from Step 3 that the predictor does not use more than

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

 for (i=1;i<=10000000;++i){
      if ((i%L1) == 0) a=1;
      else a=0;
      if ((i%L2) == 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 15. Step 4 microbenchmark.

```
void main(void){
 int a;
 int long unsigned i;
 int long unsigned liter = 10000000;
 for (i=1;i<=liter;++i){
      if ((i%L3) == 0) a=1; //L3 > L
      if ((i%L3) == 0) a=1; //spy branch
 }
}
```

Figure 16. Step 5 microbenchmark.

one global history bit, the first conditional branch is mispredicted once every $L3$ times. If there is no global component at all, the second branch is also mispredicted once every $L3$ times, while it is always predicted correctly if there is a one-bit global history component. The number of mispredictions in this experiment determines the existence of a one-bit global history predictor.

Step 6

The presence of a global component with $2^*(L-1)$ history bits is proved in the previous steps, and this step probes for the presence of a local component. The Step 6 microbenchmark has $2^*(L-1)$ 'dummy' branches (Figure 17) and varies the pattern length *LSpy* of the 'spy' branch. If the MPR is low for some *LSpy*, there is an equivalent local component with at least *LSpy* − 1 history bits. Depending on the decision mechanism involved, there could be more local history bits, so further experiments might be needed. This, however, is outside 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%LSpy) == 0) a=0; //spy branch
 }
}
```

Figure 17. Step 6 microbenchmark.

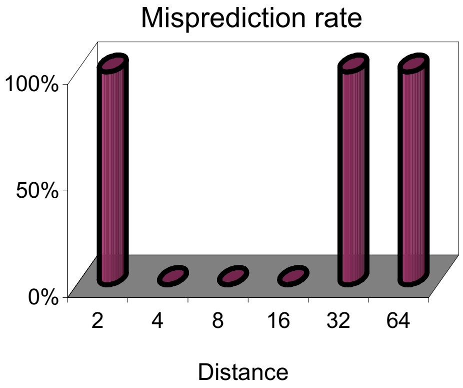


Figure 18. Misprediction rate for N_{BTB} conditional branches with varying distances.

RESULTS

BTB results

For the P6 architecture ($N_{\mathrm{BTB}} = 512$) the MPR is close to 0% when the distance between addresses of subsequent branches is 4, 8, or 16; and it is close to 100% for other distances (Figure 18). Since three different distances produce the low MPR, the P6 architecture has the BTB organized in four ways, 128 sets. Address bits 4–10 are used as the set index.

This result can also be obtained by trying to map B branches in the same set, varying the distance between them, and the number of branches (Table II). It can be seen that 16 branches collide in the same set when at a distance of 16, and eight branches collide at a distance of 2048, while four branches do not collide at any distance. Hence, the conclusion is the same: the P6 architecture has four cache ways (Figure 19).

Table II. P6 branch mispredictions when trying to map B branches in the same set.

Iterations:	Distance	Mispredicted branches
1M, B = 16	512	1953
	1024	14 938 664
1M, B = 8	1024	2520
	2048	6 927 480
1M, B = 4	2048	2400
	4096	4097

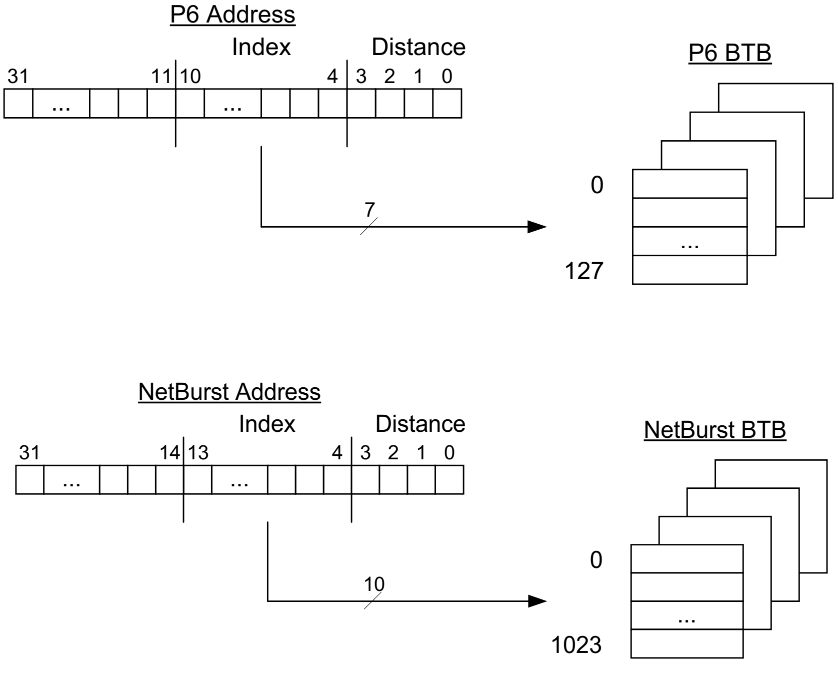


Figure 19. P6 and NetBurst BTB size and organization.

Table III. P6 branch mispredictions when the total number of branches is $2 * N_{\mathrm{BTB}}$. Iterations: 1M, $B = 1024$.

Distance	Mispredicted branches
4	1 017 750 000
8	1 016 900 000
16	1 020 700 000

Table IV. Results of the Step 1 experiment. Iterations: 10M.

Architecture	Pattern length	Mispredicted branches
P6		
	4	420
	5	432
	6	1 545 480
Netburst		
	5	987
	6	973
	7	957
	8	1256
	9	918
	10	964 830

Finally, to verify the correctness of the assumption about BTB size, the different distance experiment is performed with twice as many branches. Table III shows the results for the P6 architecture for 1024 branches. The distances that produced the low MPR when the number of branches was 512 now produce an MPR close to 100%. Hence, the actual number of BTB entries is 512.

The results are similar for the NetBurst architecture ($N_{\mathrm{BTB-FE}} = 4096$); i.e. the MPR is close to 0% when the distance between addresses of subsequent branches is 4, 8, or 16; and it is close to 100% for other distances. Therefore, the front-end BTB has four ways and 1024 sets, with bits 4–13 being used as the set index (Figure 19).

Outcome predictor results—P6 architecture

Step 1

Table IV shows the results of the Step 1 experiment (Figure 11). The maximum length of a correctly predicted pattern is 5, since the spy branch with a pattern of length 6 is mispredicted once every six times ($10\,000\,000/6 = 1\,666\,666$), which is close to the number of mispredicted branches shown in Table IV. This result can be caused by a local predictor component that uses four bits of local history, or a global component that uses eight global history bits.

Step 2

The microbenchmark has eight ‘dummy’ conditional branches before the ‘spy’ branch. Since the MPR is still close to 0 for longer global history pattern, the P6 architecture uses a local branch history of length 4.

Step 3

The microbenchmark has three conditional branches in a loop, where the first two have patterns 11. . . 10 of length 5 and 2, and hence are predictable by the local predictor component. The outcome of the third branch is correlated with the previous two. Since it has a pattern 11. . . 10 of length 10, it is not predictable by the local component with four history bits. The MPR is about 10%, which means that the third branch is mispredicted once every 10 times, when its outcome is 0. Hence, the P6 architecture does not use a global history pattern of length greater than or equal to 2.

Step 4

The Step 4 experiment is a 10 million iteration loop, with two conditional branches. The first branch has a pattern 111110 of length 6, so it is not predictable by the local component, and 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 every six times. Therefore, the P6 architecture does not include global prediction component.

Outcome predictor results—NetBurst architecture

Step 1

Table IV shows the results of the Step 1 experiment: the maximum length of a correctly predicted pattern is 9, since the ‘spy’ branch with a pattern of length 10 is mispredicted once every 10 times—about 1 million mispredictions. These results can be explained by either an eight-bit local history register or a 16-bit global history register.

Step 2

The microbenchmark has 16 ‘dummy’ branches before the ‘spy’ branch with a local pattern of length 9. The measured MPR is about 10%; i.e. the ‘spy’ branch is mispredicted once every nine times. Therefore, the Step 1 result is caused by a global component that uses 16 global history bits.

Step 6

After several runs of different Step 6 experiments, the first conclusion might be that the NetBurst architecture uses one local history bit for predictions, since a pattern length 2 is predicted correctly (Table V). As this architecture includes the trace cache, an additional experiment is needed, with the structure from the Step 6 experiment repeated 10 times in sequence: 16 ‘dummy’ branches, and one

Table V. Results of the Step 6 experiment.

Iteration	Pattern length	Mispredicted spy branches (%)
10M	2	0
	3	33
	4	25
	5	20

'spy' branch with a local history pattern of length 2. The 'spy' branches have an MPR of about 50%, which is expected for the outcome predictor without any local component. Hence, the low MPR in Step 6 with pattern length 2 is due to the trace cache, since it is able to store the sequence 'loop, 16 dummy branches, spy taken, loop, 16 dummy branches, spy not taken' as one continuous trace.

CONCLUSION

The continual growth in complexity of processor features, such as wide issue, deep pipelining, branch predictor, multiple levels of cache hierarchy, etc., puts more demand on code optimizations to achieve optimal performance. While current compilers depend on a programmer to specify which architecture to optimize the code for, and to manually adjust the code to a specific architecture, future compilers should be more architecture aware and be able to discover the relevant characteristics of the underlying architecture without a programmer's input. Consequently, the burden of optimization for different architectures will shift from a program developer to the compiler, and optimization will become more automated. Unfortunately, not all architecture details are publicly available, so the optimization process cannot rely solely on the information given in manufacturers' manuals. To determine architecture intricacies, an architecture-aware compiler should run a set of carefully tuned microbenchmarks.

This paper presents a systematic approach to uncovering the basic characteristics of branch predictors. The proposed experiment flow encompasses microbenchmarks aimed at determining relevant branch predictor parameters—namely, BTB associativity and address bits used as the index, the existence of local and global branch history components, and the number of corresponding history bits. These parameters can be used for automatic or manual code optimization. The proposed experiments can also be applied during the verification phase of processor design, and used as a starting point for comparison in future predictor research. Lastly, the experiments are of educational value, providing better understanding of branch predictor mechanisms. Although the proposed approach is demonstrated for Intel P6 and NetBurst architectures, with minor modifications it can also be used for other architectures.

ACKNOWLEDGEMENTS

The authors are grateful to the anonymous referees for their insights and suggestions for strengthening this paper. This work has been partially supported by the SED of the AMCOM.

REFERENCES

1. Intel® architecture optimization—reference manual, *IA-32*. Intel.
   http://www.intel.com/design/pentium4/manuals/248966.htm [July 2003].
2. Coleman CL, Davidson JW. Automatic memory hierarchy characterization. *Proceedings 2001 IEEE International Symposium on Performance Analysis of Systems and Software*, Tucson, AZ, 4–6 November 2001. IEEE, 2001; 103–110.
3. Saavedra-Barrera R. CPU performance evaluation and execution time prediction using narrow spectrum benchmarking. *PhD Thesis*, Computer Science Division, U.C. Berkeley, 1992.
4. Hennessy J, Patterson D. *Computer Architecture: A Quantitative Approach*. Morgan Kaufmann: San Mateo, CA, 2003.
5. Hinton G, Sager D, Upton M, Boggs D, Carmean D, Kyker A, Roussel P. The microarchitecture of the Pentium® 4 processor. *Intel Technology Journal*. http://www.intel.com/technology/itj/q12001.htm [July 2003].
6. Sprangle E, Carmean D. Increasing processor performance by implementing deeper pipelines. *Proceedings of the 29th Annual International Symposium on Computer Architecture*, Anchorage, AK, 25–29 May 2002. ACM Press, 2002; 25–34.
7. IntelVTune™ performance analyzer. Intel. www.intel.com/software/products/vtune/ [August 2002].
8. London K, Dongarra J, Moore S, Mucci P, Seymour K, Spencer T. End-user tools for application performance analysis using hardware counters. *Proceedings of the ISCA 4th International Conference on Parallel and Distributed Computing Systems*, Richardson, TX, 8–10 August 2001.
9. Smith JE. A study of branch prediction strategies. *Proceedings of the 8th Annual International Symposium on Computer Architecture*, Minneapolis, MN, 12–14 May 1981. ACM Press, 1981; 135–148.
10. Yeh TY, Patt YN. Two level adaptive training branch prediction. *Proceedings of the 24th Annual International Symposium on Microarchitecture*, Albuquerque, NM, 1991. ACM Press, 1991; 51–61.
11. Pan ST, So K, Rahmeh JT. Improving the accuracy of dynamic branch prediction using branch correlation. *Proceedings of the 5th International Conference on Architectural Support for Programming Languages and Operating Systems*, Boston, MA, 12–15 October 1992. ACM Press, 1992; 76–84.
12. Evers M, Chang PY, Patt YN. Using hybrid branch prediction to improve branch prediction accuracy in the presence of context switches. *Proceedings of the 23rd Annual International Symposium on Computer Architecture*, Philadelphia, PA, 22–24 May 1996. ACM Press, 1996; 3–10.
13. Nair R. Dynamic path-based branch correlation. *Proceedings of the 28th Annual International Symposium on Microarchitecture*, Ann Arbor, MI, 29 November–1 December 1995. ACM Press, 1995; 15–23.
14. Sun Microelectronics. UltraSPARC user's manual. http://www.sun.com/processors/manuals/802-7220-02.pdf [July 2003].
15. Intel. Intel® architecture software optimization reference manual.
    http://www.intel.com/design/PentiumIII/manuals/ [December 2001].
16. Milenkovic M, Milenkovic A, Kulick J. Demystifying Intel branch predictors. *Proceedings of the Workshop on Duplicating, Deconstructing, and Debunking*, Anchorage, AK, 26 May 2002; 52–61.