Characterizing the Impact of Predicated Execution on Branch Prediction

Scott A. Mahlke Richard E. Hank Roger A. Bringmann John C. Gyllenhaal
David M. Gallagher Wen-mei W. Hwu

Center for Reliable and High-Performance Computing
University of Illinois
Urbana-Champaign, IL 61801

Abstract

Branch instructions are recognized as a major impediment to exploiting instruction level parallelism. Even with sophisticated branch prediction techniques, many frequently executed branches remain difficult to predict. An architecture supporting predicated execution may allow the compiler to remove many of these hard-to-predict branches, reducing the number of branch mispredictions and thereby improving performance. We present an in-depth analysis of the characteristics of those branches which are frequently mispredicted and examine the effectiveness of an advanced compiler to eliminate these branches. Over the benchmarks studied, an average of 27% of the dynamic branches and 56% of the dynamic branch mispredictions are eliminated with predicated execution support.

1 Introduction

Branch instructions are recognized as a major impediment to exploiting instruction level parallelism (ILP). They force the compiler and hardware to make frequent predictions of branch directions in an attempt to find sufficient parallelism. Misprediction of these branches can result in severe performance degradation through the introduction of wasted cycles into the instruction stream. This problem is especially serious for superscalar and VLIW processors, where each wasted cycle potentially costs multiple instructions. Branch prediction strategies reduce this problem by allowing the compiler and hardware to continue processing instructions along the predicted control path, thus eliminating these wasted cycles.

There are two basic classes of branch prediction strategies: *static branch prediction* and *dynamic branch prediction.* Static branch prediction utilizes information available at compile-time to make predictions. Example static prediction schemes are prediction using branch direction (backward taken, forward not taken) [1], heuristics based on the program structure [2], and profile information [3] [4]. For compilers employing scheduling techniques such as trace scheduling [5] or superblock scheduling [6], static branch prediction is used to identify likely sequences of basic blocks which can be scheduled as single units. Dynamic branch prediction utilizes runtime behavior to make predictions. Example dynamic branch prediction schemes are the branch target buffer (BTB) with a 2-bit saturating counter [7] and two-level adaptive training [8]. For processors employing hardware scheduling, dynamic branch prediction is used to identify a continuous window of instructions. Regardless of the prediction strategy employed, the software or hardware scheduler is presented with a larger block of instructions, enabling it to expose greater amount of ILP.

While correct branch prediction can increase ILP, incorrect prediction often results in large performance penalties. Recent studies have shown that imperfect branch prediction can reduce performance by a factor of two to more than ten [9] [10] [11]. These performance penalties are attributed to several conditions. First, a large number of instructions, termed *speculative instructions*, are often executed from the predicted direction of each branch. When the branch is mispredicted, all speculative instructions must be discarded since they were improperly executed. Thus, the processor wastes a large number of instruction slots when a branch is mispredicted. Note that the amount of speculation grows as the issue width of the processor increases; therefore the negative impact of branch prediction misses also increases as the issue width of the processor increases.

The second source of performance loss due to mispredicted branches is the time necessary to undo the effects of the improperly initiated speculative instructions. This involves allowing pipelines to drain, and invalidating the appropriate instructions from processor buffers so they do not update the processor state. Third, after a mispredicted branch is discovered, execution must resume on the correct path. This involves computing the proper target address and initiating instruction fetch along this path. At a minimum, several empty pipeline cycles are required for this procedure.

Finally, the presence of a large number of branches in the instruction stream places a limit on the potential ILP. A superscalar processor may have to execute multiple branches per cycle to sustain execution of multiple instructions per cycle. Under the assumption that an instruction stream contains 25% branches, an 8-issue superscalar processor must have the capability to execute at least 2 branches per cycle. Handling multiple branches per cycle requires additional pipeline complexity, as well as designing multi-ported structures such as the BTB. In high issue rate processors, it is much easier to duplicate arithmetic function units than to predict and execute multiple branches per cycle. Therefore, a technique that eliminates branches from the instruction stream can significantly reduce the cost for achieving high issue rates for branch intensive programs.

Predicated execution support provides an effective means to eliminate branches from an instruction stream. Predicated execution refers to the conditional execution of an instruction based on the value of a boolean source operand, referred to as the predicate of the instruction [12] [13]. This architectural support allows the compiler to use an *if-conversion* algorithm to convert conditional branches into predicate defining instructions, and instructions along alternative paths of each branch into predicated instructions [14] [15] [16]. Predicated instructions are fetched regardless of their predicate value. Instructions whose predicate value is true are executed normally. Conversely, instructions whose predicate is false are nullified, and thus are prevented from modifying the processor state. Predicated execution allows the compiler to trade instruction fetch efficiency for the capability to expose ILP to the hardware along multiple execution paths.

Predicated execution offers the opportunity to improve branch handling in superscalar processors. Eliminating frequently mispredicted branches may lead to a substantial reduction in branch prediction misses. As a result, the performance penalties associated with the eliminated branches are removed. Eliminating branches also reduces the need to handle multiple branches per cycle for wide issue processors. As a side effect of reducing the number of branches in the instruction stream, the amount of speculation required to sustain full processor utilization is reduced. Therefore, in the case of a mispredicted branch, fewer speculative instructions must be discarded. Finally, predicated execution provides an efficient interface for the compiler to expose multiple execution paths to the hardware. Without compiler support, the cost of maintaining multiple execution paths in hardware grows rapidly.

In this paper, we investigate the impact of predicated execution on branch behavior. The objectives of the paper are two-fold. First, the characteristics of branches for a set of benchmarks are analyzed. Branches are characterized based on several features, including type, location, frequency, and bias. Branches which contribute large numbers of mispredictions are isolated and targeted for elimination with predicated execution support. The second objective is to analyze the effects that predicated execution has on branches and branch prediction characteristics. The ability of the compiler to eliminate the problematic branches with predicated execution is assessed. The analysis presented in this paper is based on a superscalar microarchitectural model that efficiently supports predicated execution. This model is an extension of the HP PA-RISC architecture. Hyperblock optimization and scheduling techniques are utilized by the compiler to exploit the predicated execution support [16].

```
for ( i = 0; i < 100; i++ )
    if (A[i] <= 50 )
        j = j + 1;
    else
        k = k + 1;
```
(a)

```
      mov  r1,0
      mov  r2,0
      ld   r3,addr(A)
L1:   ld   r4,mem(r3+r2)
      bgt  r4,50,L2
      add  r5,r5,1
      jump L3
L2:   add  r6,r6,1
L3:   add  r1,r1,1
      add  r2,r2,4
      blt  r1,100,L1
```
(b)

```
      mov      r1,0
      mov      r2,0
      ld       r3,addr(A)
L1:   ld       r4,mem(r3+r2)
      pred_gt  p1U,p2Ū,r4,50
      add      r5,r5,1 (p2)
      add      r6,r6,1 (p1)
      add      r1,r1,1
      add      r2,r2,4
      blt      r1,100,L1
```
(c)

Figure 1: Example of if-conversion, (a) source code segment, (b) assembly code segment, (c) assembly code segment after if-conversion.

2 Predicated Execution

In this section, the underlying predicated execution model as well as the architecture and compiler support for predicated execution are summarized. This is necessary to provide a basic understanding of the underlying framework for predicated execution used in this paper.

2.1 Overview of Predicated Execution

Predicated execution refers to the conditional execution of instructions based on the boolean value of a source operand, referred to as the *predicate*. If the value of the predicate is true (a logic 1), the instruction is allowed to execute normally, otherwise the instruction is nullified, preventing it from modifying the processor state. Figure 1 contains a simple example to illustrate the concept of predicated execution. For each iteration of the loop in Figure 1(a), either the value of **j** or **k** is conditionally incremented. The basic compiler transformation to exploit predicated execution is known as *if-conversion* [15]. If-conversion replaces conditional branches in the code with comparison instructions which define one or more predicates. Instructions control dependent on the branch are then converted to predicated instructions, utilizing the appropriate predicate value. In this manner, control dependences are converted to data dependences.

Figures 1(b) and 1(c) show the assembly code for the loop example before and after if-conversion. Note that the variables **j** and **k** have been placed in registers **r5** and **r6**. The conditional branch, **bgt**, in Figure 1(b) is replaced by a predicate define instruction, **pred_gt**, in Figure 1(c). The actual semantics of the **pred_gt** instruction will be discussed in the next subsection. It is sufficient for this example to say that the predicate **p1** is assigned the value 1 if **r4** > **50** and 0 otherwise, and the predicate **p2** is assigned the complement of **p1**. The instructions incrementing the values of **r5** and **r6** are converted to predicated instructions, associated with predicates **p1** and **p2**, respectively. For each loop iteration, either **r5** and **r6** will be incremented by the predicated add instructions, contingent on the results of the predicate define instruction. Note also that the jump instruction becomes unnecessary after if-conversion.

2.2 Architectural Support

The architectural extensions assumed to support predicated execution are based on the Cydra 5 architecture [13] and the HPL Playdoh Architecture [17]. They consist of four major components: an Nx1-bit predicate register file to store the predicate values, an additional source operand for each instruction to specify a predicate for instruction execution, a modified decode/issue stage to nullify instructions whose predicate is false, and a set of predicate defining instructions.

The set of predicate defining instructions consist of a complete set of integer, unsigned, float, and double comparison opcodes of the form shown below.

$$\text{pred\_}{<}cmp{>}\ \text{Pout1}_{<type>},\ \text{Pout2}_{<type>},\ \text{src1},\ \text{src2}\ (\text{P}_{in})$$

This instruction assigns values to **Pout1** and **Pout2** according to a comparison of **src1** and **src2** specified by <*cmp*> and the predicate <*type*> specified for each destination predicate. The comparison <*cmp*> is: equal (eq), not equal (ne), greater than (gt), etc. The boolean value written to a predicate register is a function of the result of the comparison, the input predicate of the definition instruction ($\mathbf{P}_{in}$), and the <*type*> field. The predicate <*type*> specifies one of eight possible functions: unconditional, conditional, OR, AND, or each of their complements. For a typical predicate definition instruction, the two destination predicates are a given predicate type and its complement to reflect the "then" and "else" paths of an if statement. The reader is referred to [17] for more details regarding the predicate definition semantics.

2.3 Compiler Support

The compilation techniques utilized in this paper to exploit predicated execution are based on a abstract structure called a *hyperblock* [16]. A hyperblock is a collection of basic blocks in which control may only enter at the first basic block, designated as the entry block. Control flow may leave from one or more of the basic blocks in the hyperblock. All control flow between basic blocks in a hyperblock is eliminated via if-conversion. The goal of hyperblocks is to intelligently group basic blocks from many different control flow paths into a single manageable block for compiler optimization and scheduling. Basic blocks are systematically included based on two high level goals. First, performance is maximized when the hyperblock captures a large fraction of the predicted control flow paths.[1] Thus, any likely blocks to which control may flow should be added to the hyperblock. Second, resources (fetch bandwidth and function units) are limited; therefore including too many blocks will likely result in an overall performance loss. It may be better to leave an infrequently executed block out of the hyperblock rather than insert more predicated instructions which may saturate the processors resources. Exclusion of a block from the hyperblock will require a branch instruction to be left in the hyperblock; however, if the branch is infrequently taken, it should be a highly-predictable branch. Hyperblock formation focuses on eliminating unbiased branches, while leaving highly biased branches alone since little performance gain is achieved by eliminating them. Though, careful attention is paid to the size and dependence height of each block selected for inclusion in the hyperblock. Therefore, a 50-50 branch may be left in the program if one of the target blocks requires significantly more resources than the other. An example using hyperblock compilation techniques is presented in the next section.

[1] Predicted control paths are identified using static branch prediction (profile information in this implementation).

```
linect = wordct = charct = token = 0;
for (;;) {
    if (--(fp)->cnt < 0)   c = filbuf(fp);
    else   c = *(fp)->ptr++;
    if (c == EOF)   break;
    charct++;
    if ((' ' < c) && (c < 0177))
        if (! token) {
            wordct++;
            token++;
        }
        continue;
    if (c == '\n')   linect++;
    else if ((c != ' ') && (c != '\t'))   continue;
    token = 0;
}
```

Figure 2: Source code for the inner loop of *wc*.

3 A Case Study

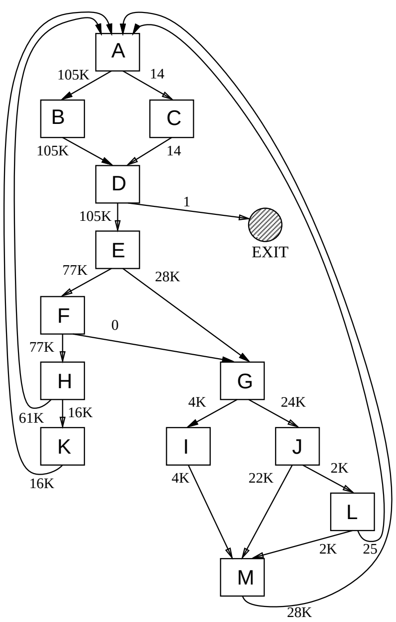
In order to provide a deeper understanding of the branch characteristics of non-numeric applications and how these branches are effected by predicated execution, a detailed analysis of one of the benchmark programs is presented in this section. The benchmark chosen is *word count* (wc). This benchmark was chosen for two reasons. First, it contains a loop which accounts for a large fraction of its execution time, yet is small enough to be presented in the context of a paper. Second, the loop has a non-trivial control structure which presents a challenge to branch handling strategies.

The preprocessed C source code for the loop segment is presented in Figure 2. The purpose of this program is to count the number of characters, words, and lines in an input file. A character buffer is processed in the loop and re-filled as necessary until the end-of-file marker is encountered. The corresponding assembly code and control flow graph for the loop segment are presented in Figure 3. The control flow graph is augmented with the execution frequencies of each control transfer for the measured run of the program. This loop is characterized by small basic blocks and a large percentage of branches. Overall, the loop segment contains 13 basic blocks with a total of 34 instructions. Of the 34 instructions, 14 are branches, 8 conditional, 5 unconditional, and 1 subroutine call.

Elimination of Branches with Hyperblock Formation. Branches are eliminated and replaced with predicated instructions using hyperblock formation. Hyperblock formation consists of several steps. First, all loop-back branches of innermost loops are coalesced into a single backedge. This procedure is illustrated for the example loop in Figure 4a. A new block, N, is created with a single jump instruction to the loop header, A. All loop-back branches are then adjusted to target the new block. Therefore, in Figure 4a, the branches in blocks H, K, L, and M are retargeted from A to N. The purpose of this step is to convert as many loop-back branches to non-loop (i.e. intra-loop) branches as possible. Because

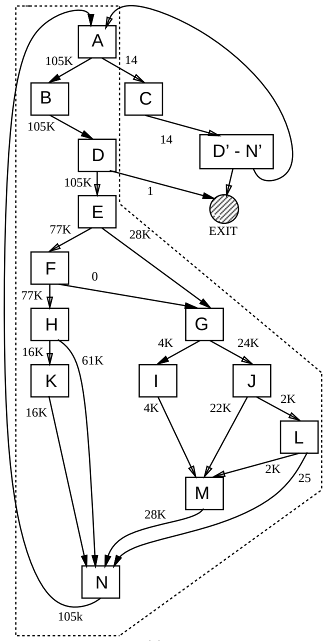
```
LA: ld  r98, mem(r3 + 0)
    add  r27, r98, -1
    st  mem(r3 + 0), r27
    blt  r98, 1, LC
LB: ld  r30, mem(r3 + 4)
    add  r29, r30, 1
    st  mem(r3 + 4), r29
    ld  r4, mem(r30+0)
LD: beq  r4, -1, EXIT
LE: ld  r33, mem(r73 + 0)
    add  r32, r33, 1
    st  mem(r73 + 0), r32
    bge  32, r4, LG
LF: bge  r4, 127, LG
LH: bne  0, r2, LA
LK: ld  r36, mem(r72 + 0)
    add  r35, r36, 1
    st  mem(r72 + 0), r35
    add  r2, r2, 1
    jump  LA
LG: beq  r4, 10, LI
LJ: bne  r4, 32, LL
LM: mov  r2, 0
    jump  LA
LI: ld  r39, mem(r71 + 0)
    add  r38, r39, 1
    st  mem(r71+0), r38
    jump  LM
LL: bne  r4, 9, LA
    jump  LM
LC: mov  Parm0, r3
    jsr  filbuf
    mov  r4, Ret0
    jump  LD
```

(a)



(b)

Figure 3: Inner loop segment of *wc*, (a) assembly code, (b) control flow graph.



(a)

```
LA: pred_clr  p4, p6
    ld  r98, mem(r3 + 0)
    add  r27, r98, -1
    st  mem(r3 + 0), r27
    blt  r98, 1, LC
    ld  r30, mem(r3 + 4)
    add  r29, r30, 1
    st  mem(r3 + 4), r29
    ld  r4, mem(r30 + 0)
    beq  r4, -1, EXIT
    ld  r33, mem(r73 + 0)
    add  r32, r33, 1
    st  mem(r73+0), r32
    pred_ge  p4(OR), p1(Ū), 32, r4
    pred_ge  p4(OR), p2(Ū), r4, 127  (p1)
    pred_eq  p3(U), -, 0, r2 (p2)
    pred_eq  p6(OR), p5(Ū), r4, 10  (p4)
    pred_eq  p7(U), -, r4, 10 (p4)
    pred_eq  p6(OR), p8(Ū), r4, 32  (p5)
    ld  r36, mem(r72+0)  (p3)
    add  r35, r36, 1  (p3)
    st  mem(r72+0), r35  (p3)
    add  r2, r2, 1  (p3)
    ld  r39, mem(r71 + 0)  (p7)
    add  r38, r39, 1  (p7)
    st  mem(r71 + 0), r38  (p7)
    pred_eq  p6(OR), -, r4, 9  (p8)
    mov  r2, 0  (p6)
    jump  LA
```

(b)

Figure 4: Inner loop segment for *wc*, (a) control flow graph after loop-back branch coalescing, block selection, and tail duplication (b) assembly code after if-conversion.

if-conversion can only remove non-loop branches, this procedure provides more opportunity for eliminating branches with multiple back branches. A similar procedure can be applied for each set of loop-exit branches to the same target. In this loop example, there is only one loop exit, so there is no opportunity for coalescing exit blocks.

The second step of hyperblock formation is choosing the set of blocks to be included in the hyperblock. In this example, all blocks are selected for inclusion with the exception of block C. The priority for C is very low due to its low execution frequency and the hazardous instruction (subroutine call to *filbuf*) which it contains. The blocks selected for inclusion for the example loop are outlined by the dashed line in Figure 4a. In order to perform if-conversion on the selected blocks, control flow from non-selected blocks to selected blocks must be eliminated. Such paths of control are referred to as *side-entry points* into the hyperblock. The third step, tail duplication, eliminates side-entry points by duplicating a portion of the selected blocks and re-adjusting the appropriate control flow arcs. In the example loop, a side entry point exists from C to D. This is eliminated by duplicating blocks D through N (pictured as block D′-N′) and re-adjusting the C-D control flow arc to C-D′. The control flow graph for the loop after tail duplication is shown in Figure 4a.

The final step of hyperblock formation is to perform if-conversion on the blocks selected for the hyperblock. In our current implementation, a variant of the RK if-conversion algorithm is utilized [15]. The if-conversion algorithm first calculates the localized control dependence information among the selected basic blocks. One predicate register is then assigned to all basic blocks with the same set of control dependences. Predicate register defining instructions are inserted into all basic blocks which are the source of the control dependences associated with a particular predicate. Next, all instructions in each selected block are predicated based on the predicate assigned to their block. Finally, all conditional and unconditional branches from selected blocks to other selected blocks are removed.

The resultant assembly code for the loop body of the example is presented in Figure 4b. The loop contains eight unique control dependences; thus eight predicate registers are required. Of the 12 original branches in the blocks selected for inclusion, all but three are removed. The remaining branches in the hyperblock are two infrequent exit branches and the unconditional loop-back branch at the bottom of the hyperblock.

Comparison of Branch Characteristics. The branch characteristics for *wc* before and after hyperblock formation are analyzed in the remainder of this section. Tables 1 – 5 present a detailed breakdown of the branch behavior for *wc*. Each table consists of two parts: the behavior for the base architecture (Base), and the base architecture with predicated execution support (Pred). The base architecture is an 8-issue superscalar processor with uniform function units. The instruction latencies assumed are those of the HP PA-RISC 7100. More details regarding the architecture model and the simulation model are presented in Section 4.1.

An overall breakdown of the dynamic branches is presented in Table 1. Branches are broken down into three categories: type (conditional, unconditional, subroutine call/return, or indirect), class (loop or non-loop), and location (inner-loop, outer-loop, or straight-line). For the class

Type	Conditional					Unconditional					Jsr/Rts	Ind	Total
Class	Loop		Non-loop			Loop		Non-loop					
Location	Inner	Outer	Inner	Outer	StrLn	Inner	Outer	Inner	Outer	StrLn			
Base	184K	1	339K	3	4	43726	1	5355	0	0	29	0	572K
Pred	105K	1	105K	3	4	105K	0	23	0	0	29	0	315K

Table 1: Dynamic branch breakdown for *wc*.

	0	1-10	11-20	21-30	31-40	41-50	51-60	61-70	71-80	81-90	91-100
Base	287K	24898	27606	105K	3	2	0	3	77590	0	3
Pred	210K	14	5	0	16	8	0	3	8	0	3

Table 2: Taken frequency distribution for *wc*.

category, loop branches refer to loop-back branches as well as loop-exit branches. All other branches are grouped in non-loop class. For the location category, inner-loop specifies branches contained within innermost loops. Outer-loop specific branches contained within a loop which is not an innermost loop. All other branches do not reside explicitly within any loop body within their respective function and are thus placed in the straight-line category. From Table 1, with predicated execution support, the total number of dynamic branches was reduced from 572K to 315K, approximately a 45% reduction. This is mainly attributable to the reduction in branches shown in column [conditional, non-loop, inner]. The branches in this category removed are from blocks E, F, G, and J (Figure 3). The dynamic number of branches is also reduced in column [conditional, loop, inner]. Although if-conversion does not directly remove loop branches, the loop-back and exit coalescing performed during hyperblock formation allows this number to be reduced with predicated execution support.

One interesting note from Table 1, column [unconditional, loop, inner], is that the number of branches is increased with predicated execution support. This is a result of loop-back branch coalescing. In the transformed code, there is a single unconditional back branch which is executed on every iteration of the loop. Since unconditional loop-back branches were only executed on a subset of iterations in the original code, an increase in the number is observed as shown in column [unconditional, loop, inner].

To further break down the conditional branches, the taken frequency distribution is presented in Table 2. The data shown are the dynamic counts of all conditional branches whose taken percentages lie within the range at the top of each column. For example, for the base architecture, 8% (27,606) of the branches are taken 11-20% of the time. From Table 2, a drastic change is observed from the base to the predicate architecture. With predicated execution support, the taken frequency of nearly all conditional branches is reduced to 0%. This behavior is a direct result of the hyperblock formation procedure. All conditional branches in the loop are eliminated with the exception of the two branches to blocks C and EXIT (Figure 3). These remaining branches are heavily biased to the fall-through path, taken only 1 and 14 times, respectively.

The taken frequency distribution chart is a direct measure of the effectiveness of the compiler at eliminating unbiased branches with predicated execution support. The desired trend is for the compiler to eliminate all unbiased branches, while leaving highly biased branches in the code. If only highly biased conditional branches remain with predicated execution support, very few mispredictions for conditional branches will occur.

Table 3 presents the branch misprediction breakdown for *wc*. Data for three dynamic prediction models is reported: BTB with a 2-bit saturating counter (Ctr), BTB with profile-based direction prediction (Pro), and a branch target cache (Btc). For each scheme, a 1024-entry buffer is utilized. In our model, the branch target cache is similar to a BTB, except that the buffer is addressed by the cache block number instead of the instruction address, and simply stores the address to which control flowed the last time this cache block was executed. Thus, there is only one branch prediction for all branches within a single cache block. The cache block size simulated is 64 bytes (16 instructions).

From Table 3, the most important data is the huge drop in branch prediction misses from the base to the predicate architecture. The number of misses drops by factor of almost 1000 for all models. The reason for this huge drop the compiler successfully removes the branches causing the majority of the misses. Considering the Ctr and Pro models, the two problematic categories of branches are shown in columns [conditional, loop, inner] and [conditional, non-loop, inner]. As shown in column [conditional, non-loop, inner] the hyperblock formation procedure removes all branches of this category, with the exception of 2. The remaining 2 are highly biased as fall-through, and are only mispredicted a total of 33 times. The branches in column [conditional, loop, inner] have been coalesced into a single unconditional branch which is mispredicted only when it is not in the BTB.

An interesting pattern is observed in the Btc model. In the base architecture, it predicts [conditional, loop, inner] branches more effectively than either Ctr or Pro. All models have approximately the same performance for [conditional, non-loop, inner] branches. The poorer performance of the Btc model is the result of the large number of mispredictions for [unconditional, loop, inner] branches. This behavior occurs because there are two unconditional branches which share the same cache block. Therefore, they are constantly changing the predicted target block and causing the other to miss. However, the unconditional branches causing this problem are eliminated by the compiler with predicated execution support. The performance level of all three branch prediction models is approximately equal with predicated execution support for *wc*.

Table 4 presents the static misprediction coverage of all branches. The branch prediction scheme is a BTB with 2-bit saturating counter and the percentages shown are the sum of the conditional and unconditional branch values. Misprediction coverage is defined as the percentage of static branches which account for a given percentage of dynamic mispredictions. For example, in the base architecture, 8% of the

Type	Conditional					Unconditional					Jsr/Rts	Ind	Total	MPR
Class	Loop		Non-loop			Loop		Non-loop						
Location	Inner	Outer	Inner	Outer	StrLn	Inner	Outer	Inner	Outer	StrLn				
Base Ctr	19521	0	32604	0	2	2	1	5	0	0	9	0	52144	0.09
Pro	16173	0	32987	0	2	2	1	5	0	0	9	0	49179	0.09
Btc	14047	0	33000	0	2	43726	1	4082	0	0	9	0	94867	0.17
Pred Ctr	3	1	33	0	2	2	0	6	0	0	9	0	56	0.00
Pro	3	1	31	0	2	2	0	6	0	0	9	0	54	0.00
Btc	3	1	47	0	2	26	0	23	0	0	9	0	111	0.00

Table 3: Branch misprediction breakdown for *wc*.

	10	20	30	40	50	60	70	80	90	100
Base	0.06	0.06	0.06	0.06	0.06	0.06	0.06	0.08	0.08	0.56
Pred	0.07	0.07	0.07	0.14	0.14	0.20	0.29	0.34	0.45	0.52

Table 4: Branch misprediction coverage for *wc* using a BTB with 2-bit counter.

static branches account for 90% of the dynamic mispredictions. Note that 100% of all mispredictions occur in 56% of the static branches; the remaining 44% of branches are either never executed or never mispredicted. The desired distribution is that a small number of static branches should account for a large portion of the misses, as seen in this table. In this situation, the compiler can focus its efforts on eliminating this small number of branches, reducing mispredictions and improving performance.

The distribution of the number of instructions between branches and the number of instructions between mispredicted branches using the BTB with 2-bit counter scheme is presented in Table 5. All branch categories are considered in this data. The data shown are the dynamic counts of the number of branches which are separated by the specified number of instructions. For example, in the base architecture, 59% (335K) of branches are separated by 3-4 instructions, and 11% (5561) of mispredicted branches are separated by 1 instruction. In general, for the base architecture, there is a very small number of instructions between branches and mispredicted branches. The average values are 2.02 instructions between branches and 32.15 instructions between mispredicted branches. For a superscalar processor, this low number of instructions between mispredicted branches indicates a significant branch misprediction overhead. For the predicated architecture, the distribution is somewhat distorted since there are only 56 mispredicted branches in the entire program. The average values are 9.00 instructions between branches and 56355.55 instructions between mispredicted branches. By eliminating branches, the compiler has successfully increased the distance between branches and mispredicted branches for the predicate architecture.

Summary. This section demonstrates how predicated execution can be applied to the benchmark *wc* with the use of hyperblocks. The compiler was able to reduce the number of dynamic branch instructions by 45%. More importantly, we were able to remove almost all hard to predict branches. As a result, the number of dynamic mispredictions was reduced dramatically (1000 times), regardless of the branch prediction scheme employed. Thus, the results for *wc* indicated that predicated execution may lessen the need to employ a sophisticated branch prediction scheme. Finally, we saw that hyperblocks greatly increased the number of instructions between mispredicted branches. In the next section, we examine whether the the branch characteristics exhibited by *wc* are consistent across a set of benchmarks.

4 Experimental Evaluation

4.1 Methodology

The impact of predicated execution on branch behavior is evaluated using hyperblock compilation techniques in this section. The benchmarks studied consist of *022.li*, *023.eqntott*, *026.compress*, *056.ear* from SPEC-92, and the Unix utilities *cmp*, *grep*, *lex*, *quick_sort*, *wc*. The benchmarks are compiled to produce an intermediate code for the target architecture, either base or predicate. The base architecture is an 8 issue superscalar processor, with no limitation placed on the combination of instructions which may be issued each cycle. The base architecture is further assumed to have 64 integer and 64 floating-point registers. The memory system consists of a 64K direct mapped instruction cache and a 64K direct mapped, blocking data cache; both with 64 byte block size. The data cache is write-through with no write allocate and has a miss penalty of 12 cycles. The dynamic branch prediction strategy is one of three models: a 1K entry BTB with 2 bit counter, a 1K entry BTB with profile-based direction prediction, or a 1K entry branch target cache. The instruction latencies assumed are those of the HP PA-RISC 7100. The predicate architecture is the same as the base architecture with extensions to support predicated execution as described in Section 2.2. A 64-entry predicate register file is assumed in the predicate architecture.

Following register allocation and code scheduling, the intermediate code is in a form which could be executed by the target architecture. To allow simulation of the predicated code on the host HP PA-RISC processor, the code is modified to remove all predicated instructions. Instructions to emulate the effects of predicated instructions are inserted by the compiler, using the bit manipulation and conditional nullification capabilities of the PA-RISC instruction set. This emulation code is then probed. Execution of the probed code demonstrates correctness of the target architecture code, and also generates an instruction trace containing memory address information, predicate register contents, and branch directions. Note the code utilized specifically for emulation and trace generation is not simulated; therefore it is not counted in any of the measured statistics.

The compiler support utilized in this evaluation is more sophisticated than what was presented in Section 3 for *wc*. For the base architecture, superblock formation, superblock ILP optimizations, and superblock scheduling are applied to effectively generate code for a wide issue processor without predicated execution support [6]. For the predicate architecture, the ILP optimizations and scheduling techniques are

		0	1	2	3-4	5-8	9-16	17-32	33-64	65-128	129-256	257+
Base	Br	209K	27586	17	335K	4	3	2	0	0	0	0
	MP br	4060	5561	4	8	9	5487	23980	5444	5689	1742	160
Pred	Br	58	12	105K	24	1	210K	2	0	0	0	0
	MP br	5	7	4	8	9	6	3	1	0	0	13

Table 5: Distance between branches and mispredicted branches for *wc* using a BTB with 2-bit counter.

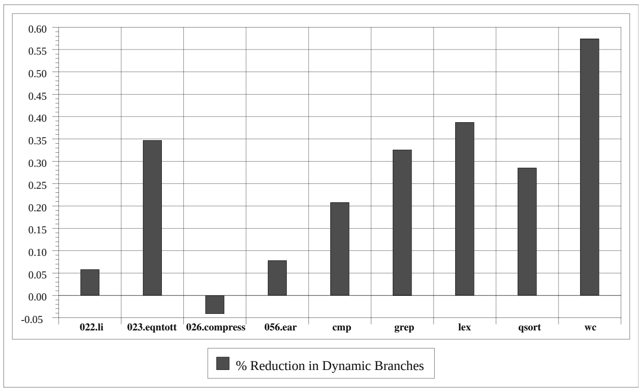


Figure 5: Reduction in total dynamic branches with predicated execution support.

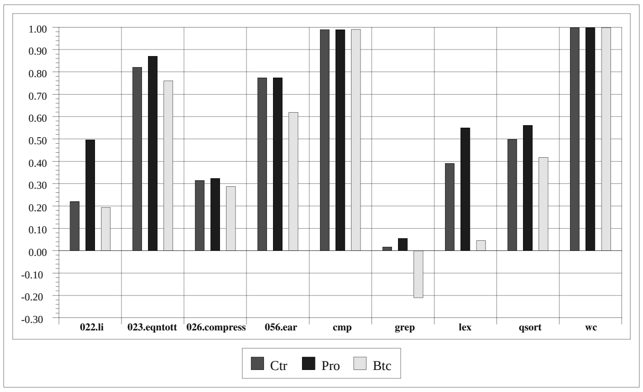


Figure 6: Reduction in the number branch mispredictions with predicated execution support.

applied to hyperblocks to expose additional ILP for a wide issue processor with predicated execution support. An important item to note is the branch behavior data for *wc* presented in this section is similar to that presented in Section 3. However, there are some distinct differences as to the categorization of branches. These differences are caused by the changes to the control flow graph and loop structure introduced by superblock and ILP transformations.

4.2 Results and Analysis

The branch analysis introduced in Section 3 was performed for each of the benchmarks. Here, much of the data is presented in graphical form rather than tables. Appendix A contains a complete tabular listing of the data generated for all benchmarks. The table format and contents is the same as that used in Section 3.

Total Dynamic Branches. Predicated execution enables the compiler to remove hard to predict branches from the instruction stream. Thus, the total number of dynamic branches in the program would be expected to decrease as the result of hyperblock techniques. Figure 5 shows the reduction in dynamic branches for each benchmark for the predicate architecture. As expected, predication significantly reduces the number of dynamic branches for most benchmarks, averaging a 27% reduction across the benchmarks. The one exception was *026.compress*, which experiences a small increase in the number of branches for the predicated execution case (12343K to 12852K total dynamic branches, Table 7). The increase in the number of dynamic branches is a side-effect of the hyperblock formation procedure. As a hyperblock is formed, branches that cannot be removed from the merged blocks are predicated. These branches are predicted regardless of the value of their predicate, increasing the number of times each branch is fetched, and hence increasing the dynamic count for that branch. The reader is referred to Table 7 in Appendix A for a complete breakdown of the dynamic branch counts for each benchmark.

Total Mispredicted Branches. By allowing hard to predict branches to be removed from the instruction stream, the total number of mispredicted branches should be decreased with predicated execution support. Figure 6 shows this reduction for each of the benchmarks, using the three branch prediction schemes. A complete breakdown of the branch misprediction counts is provided in Appendix A, Table 8. For the majority of benchmarks, the number of mispredictions decreases significantly regardless of the prediction scheme employed. For *wc* and *cmp*, virtually all mispredictions are removed, regardless of the branch prediction scheme. Although the numbers are not as significant as the *wc* example presented in Section 3, the number of mispredictions is reduced by a factor of 6 for *023.eqntott* and a factor of 4 for *056.ear* (Table 8). Even though there is a 4% (Figure 5) increase in the number of dynamic branches for *026.compress*, all branch prediction schemes achieve a 30% decrease in the number of branch mispredictions. The opposite trend appears for *grep*: the number of dynamic branches is reduced by 30% (Figure 5) with predication, but there is little decrease in the number of branch mispredictions and even an increase for the Btc model. This behavior occurs because *grep* is dominated by heavily biased branches. As a result, branches which are eliminated do not significantly reduce the number of mispredictions. The observed increase in mispredictions for the Btc model occurs because the branches in the hyperblock happen to be scheduled into the same cache block. Since the Btc model only allows a single prediction per cache block, prediction accuracy is lost.

Dynamic Distance Between Branches. Figure 7 presents the distribution, averaged across all benchmarks, of the number of instructions between branches and mispredicted branches. The dynamic branch prediction scheme utilized to derive this data is the BTB with 2-bit counter. The desired trend is for the distributions to shift to the right with predicated execution support. This indicates the compiler is

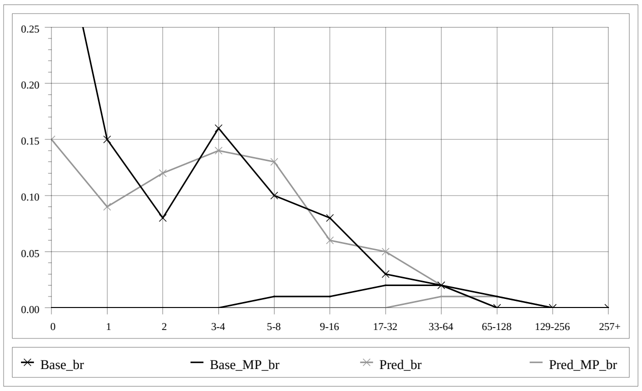


Figure 7: Distribution of the distance between branches and mispredicted branches using a BTB with 2-bit counter.

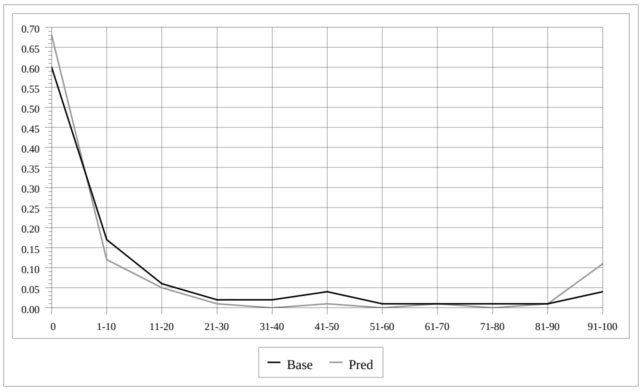


Figure 8: Average taken frequency distribution of conditional branches across all benchmarks.

	Base		Pred	
	Branches	Mispred Branches	Branches	Mispred Branches
022.li	3.5	43	3.7	62
023.eqntott	2.1	28	4.8	195
026.compress	7.3	78	8.7	137
056.ear	4.7	129	5.3	492
cmp	2.6	433	3.6	43940
grep	0.9	126	1.2	100
lex	1.6	160	6.4	290
qsort	8.5	54	14.7	128
wc	2.0	46	6.8	28464
Average	3.7	122	6.1	8201

Table 6: Average distance between branches and mispredicted branches using a BTB with 2-bit counter.

successfully removing branches from the instruction stream, thereby increasing the distances between branches and mispredicted branches. The distributions indicate the desired behavior is obtained for the predicate architecture. The most notable change is the distance between mispredicted branches distribution which starts in the 3-4 instruction category for the base architecture. However, for the predicate architecture the distribution has been shifted to start in the 17-32 instruction range. This indicates the predicate architecture can consistently process a moderate number of instructions (17-32) before encountering a misprediction. This is especially important for wider issue processors, which cannot be fully utilized unless sufficient distance can be established between mispredicted branches. The individual dynamic distributions for each benchmark are given in Table 11 of the appendix.

On an individual benchmark basis, the average distance between branches and mispredicted branches is presented in Table 6. With predicated execution support, the compiler has effectively eliminated branches to increase distance between the remaining branches. The benchmark improved the largest amount is *lex*, 1.6 to 6.4 instructions. A more magnified effect is observed for the increased distance between mispredicted branches. For *cmp* and *wc*, almost all mispredictions are eliminated, thus the average distance between mispredictions is extremely high. The anomaly is *grep*, in which the average distance between mispredicted branches is reduced with predicated execution support. This behavior occurs because the number of mispredictions is relatively unchanged in the predicate architecture (Table 8). However, the total number of instructions executed in *grep* is reduced in the predicate architecture due to additional compiler optimization opportunities exposed by hyperblock formation. As a result, a net reduction in instructions per misprediction is observed.

Taken Branch Frequency. Figure 8 presents the average taken frequency distribution for conditional branches across all benchmarks. As discussed in Section 3, the effectiveness of the compiler support for predicated execution can be shown by eliminating or reducing the less biased branches and leaving the branches that are more heavily biased towards taken or not taken. As the figure shows, this was indeed the case. In particular, the number of branches that fall into the range of taken from 1% of the time to 90% of the time were reduced. In addition, branches that were never taken or taken 91-100% of the time have been increased.

5 Related Work

Predicated or guarded execution has been examined by several researchers. Decision tree scheduling utilizes guarded instructions to achieve large performance improvements on deeply pipelined processors [12]. Guarded instructions allow instructions from multiple execution paths to be placed in load/branch delay slots to effectively hide long execution latencies. Predicated execution support was used extensively in the Cydra 5 system [13] [18]. Predicated execution is integrated into the optimized execution of modulo scheduled inner loops to control the prologue, epilogue, and iteration initiation. Predicated execution also allows loops with conditional branches to be efficiently modulo scheduled.

Pnevmatikatos and Sohi examine the effectiveness of guarded execution on dynamically scheduled superscalar processors [19]. They show that full guarding can significantly increase the average basic block size and the average dynamic window size. They also show moderate increases in both may be obtained with restricted guarding. A major difference with this work is that we focus on characterizing the behavior of the branches responsible for mispredictions and the effectiveness of predicated execution to deal with these branches using hyperblock compilation techniques [16].

6 Conclusions

Branch instructions pose serious difficulties to exploiting instruction-level parallelism. Even with sophisticated branch prediction techniques, a large percentage of frequently executed branches remain difficult to predict. In this paper, an in-depth analysis of the characteristics of branches is presented. Branches which contribute large numbers of mispredictions are isolated and targeted for elimination with predicated execution support. Compiler support for predicated execution is based on a structure called a hyperblock. The goal of hyperblock formation is to intelligently group basic blocks from many different control flow paths into a single manageable block for compiler optimization and scheduling. Hyperblock formation focuses on eliminating unbiased branches, while leaving highly biased branches alone, since little performance gain is achieved by eliminating them.

Results show the compiler can substantially reduce the number of dynamic branches with predicated execution support. Across all benchmarks, an average of 27% reduction in the dynamic branches was observed. More importantly, though, the compiler is able to remove a large fraction of the difficult to predict branches. Therefore, the number of mispredictions is also considerably reduced. The addition of predicate support to an architecture utilizing a BTB with a 2-bit counter reduced the branch prediction misses by 75% in 4 of the 9 benchmarks. Over 20% of the branch prediction misses are eliminated in all but one of the benchmarks. The distance between branches and mispredicted branches is also an important measure. With predicated execution support the average distance between branches is increased from 3.7 to 6.1 instructions, and the average distance between mispredicted branches is increased from 122 to 8201, for the benchmarks studied.

While predicated execution support was able to reduce the number of mispredicted branches, there are still a large number of problematic branches remaining. These problematic branches motivate future architecture and compiler studies on predicated execution.

Acknowledgements

The authors would like to thank all members of the IMPACT research group and the anonymous referees whose comments and suggestions helped to improve the quality of this paper significantly. This research has been supported by the National Science Foundation (NSF) under grant MIP-9308013, Joint Services Engineering Programs (JSEP) under Contract N00014-90-J-1270, Intel Corporation, the AMD 29K Advanced Processor Development Division, Hewlett-Packard, SUN Microsystems, NCR, and NASA under Contract NASA NAG 1-613 in cooperation with ICLASS. John Gyllenhaal was also supported by an NSF Graduate Fellowship.

References

[1] J. E. Smith, "A study of branch prediction strategies," in *Proceedings of the 8th International Symposium on Computer Architecture*, pp. 135–148, May 1981.

[2] T. Ball and J. R. Larus, "Branch prediction for free," in *Proceedings of the ACM SIGPLAN 1993 Conference on Programming Language Design and Implementation*, pp. 300–313, June 1993.

[3] W. W. Hwu, T. M. Conte, and P. P. Chang, "Comparing software and hardware schemes for reducing the cost of branches," in *Proceedings of the 16th International Symposium on Computer Architecture*, pp. 224–233, May 1989.

[4] J. A. Fisher and S. M. Freudenberger, "Predicting conditional branch directions from previous runs of a program," in *Proceedings of 5th International Conference on Architectual Support for Programming Languages and Operating Systems*, pp. 85–95, October 1992.

[5] J. A. Fisher, "Trace scheduling: A technique for global microcode compaction," *IEEE Transactions on Computers*, vol. c-30, pp. 478–490, July 1981.

[6] W. W. Hwu, S. A. Mahlke, W. Y. Chen, P. P. Chang, N. J. Warter, R. A. Bringmann, R. G. Ouellette, R. E. Hank, T. Kiyohara, G. E. Haab, J. G. Holm, and D. M. Lavery, "The Superblock: An effective technique for VLIW and superscalar compilation," *Journal of Supercomputing*, vol. 7, pp. 229–248, January 1993.

[7] J. Lee and A. J. Smith, "Branch prediction strategies and branch target buffer design," *IEEE Computer*, pp. 6–22, January 1984.

[8] T. Y. Yeh and Y. N. Patt, "Two-level adaptive training branch prediction," in *Proceedings of the 24th Annual International Symposium on Microarchitecture*, pp. 51–61, November 1991.

[9] M. D. Smith, M. Johnson, and M. A. Horowitz, "Limits on multiple instruction issue," in *Proceedings of the 3rd International Conference on Architectural Support for Programming Languages and Operating Systems*, pp. 290–302, April 1989.

[10] D. W. Wall, "Limits of instruction-level parallelism," in *Proceedings of the 4th International Conference on Architectural Support for Programming Languages and Operating Systems*, pp. 176–188, April 1991.

[11] M. Butler, T. Yeh, Y. Patt, M. Alsup, H. Scales, and M. Shebanow, "Single instruction stream parallelism is greater than two," in *Proceedings of the 18th International Symposium on Computer Architecture*, pp. 276–286, May 1991.

[12] P. Y. Hsu and E. S. Davidson, "Highly concurrent scalar processing," in *Proceedings of the 13th International Symposium on Computer Architecture*, pp. 386–395, June 1986.

[13] B. R. Rau, D. W. L. Yen, W. Yen, and R. A. Towle, "The Cydra 5 departmental supercomputer," *IEEE Computer*, vol. 22, pp. 12–35, January 1989.

[14] J. R. Allen, K. Kennedy, C. Porterfield, and J. Warren, "Conversion of control dependence to data dependence," in *Proceedings of the 10th ACM Symposium on Principles of Programming Languages*, pp. 177–189, January 1983.

[15] J. C. Park and M. S. Schlansker, "On predicated execution," Tech. Rep. HPL-91-58, Hewlett Packard Laboratories, Palo Alto, CA, May 1991.

[16] S. A. Mahlke, D. C. Lin, W. Y. Chen, R. E. Hank, and R. A. Bringmann, "Effective compiler support for predicated execution using the hyperblock," in *Proceedings of the 25th International Symposium on Microarchitecture*, pp. 45–54, December 1992.

[17] V. Kathail, M. S. Schlansker, and B. R. Rau, "HPL playdoh architecture specification: Version 1.0," Tech. Rep. HPL-93-80, Hewlett-Packard Laboratories, Palo Alto, CA 94303, February 1994.

[18] J. C. Dehnert, P. Y. Hsu, and J. P. Bratt, "Overlapped loop support in the Cydra 5," in *Proceedings of the Third International Conference on Architectural Support for Programming Languages and Operating Systems*, pp. 26–38, April 1989.

[19] D. N. Pnevmatikatos and G. S. Sohi, "Guarded execution and branch prediction in dynamic ILP processors," in *Proceedings of the 21st International Symposium on Computer Architecture*, pp. 120–129, April 1994.

A Appendix

Type		Conditional					Unconditional					Jsr/Rts	Ind	Total
Class		Loop		Non-loop			Loop		Non-loop					
Location		Inner	Outer	Inner	Outer	StrLn	Inner	Outer	Inner	Outer	StrLn			
022.li														
	Base	2419K	57667	147K	428K	3405K	138K	4822	2687	263	121K	549K	0	7275K
	Pred	2149K	191K	1141K	341K	1959K	398K	10376	26629	2725	81179	550K	0	6853K
023.eqntott														
	Base	28708K	101M	2285K	149M	5017K	172K	19578	4465	129K	1614K	6873K	66	296M
	Pred	176M	263K	956K	1903K	4707K	301K	56317	21	176K	1625K	6886K	66	193M
026.compress														
	Base	5217K	1758K	39	4760K	88	49590	93008	1	463K	22	250	0	12343K
	Pred	4383K	4264K	39	2664K	88	306K	871K	1	361K	22	250	0	12852K
056.ear														
	Base	1065M	117M	234M	43240K	13047K	54997	41728	0	111	2480K	15005K	0	1491M
	Pred	878M	6320	21	18675	12420K	233M	1	233M	6311	2480K	15433K	0	1375M
cmp														
	Base	210K	13	315K	1	11	4037	1	42	0	0	31	0	530K
	Pred	420K	13	0	44	11	0	2	0	43	0	31	0	420K
grep														
	Base	4300	312K	0	340K	465	0	8831	0	5539	0	308	1	672K
	Pred	210K	108K	0	118K	465	4792	856	0	9483	0	308	1	453K
lex														
	Base	5866K	778K	242K	6958K	31689	51243	105K	10190	42230	4814	15600	678	14107K
	Pred	6195K	902K	69121	1294K	24227	60470	54295	4392	18143	5122	16708	3881	8648K
qsort														
	Base	5512K	0	242K	0	1062K	236K	0	0	0	108K	614K	0	7777K
	Pred	2075K	0	135K	0	455K	1879K	0	0	0	399K	614K	0	5559K
wc														
	Base	222K	58237	6	199K	4	11245	26489	2	4927	0	29	0	523K
	Pred	210K	23	6	35	4	13137	7	2	29	0	29	0	223K

Table 7: Dynamic branch breakdown for all benchmarks.

Type		Conditional					Unconditional					Jsr/Rts	Ind	Total	MPR
Class		Loop		Non-loop			Loop		Non-loop						
Location		Inner	Outer	Inner	Outer	StrLn	Inner	Outer	Inner	Outer	StrLn				
022.li															
Base	Ctr	197K	852	29688	11775	218K	118	153	12	36	4021	261K	0	724K	0.10
	Pro	409K	1568	27645	15357	368K	76	80	12	33	3474	256K	0	1082K	0.15
	Btc	326K	965	45494	16607	402K	47614	1103	802	149	83303	445K	0	1370K	0.19
Pred	Ctr	216K	37652	32243	8955	50515	55	12	25	128	1500	218K	0	565K	0.08
	Pro	187K	33123	26367	15556	68877	55	12	25	125	772	213K	0	545K	0.08
	Btc	349K	65170	34196	13297	77203	88911	6132	13878	891	48920	405K	0	1104K	0.16
023.eqntott															
Base	Ctr	1119K	2133K	182K	24080K	120K	8571	1078	490	2450	4118	3356K	98	31010K	0.10
	Pro	3708K	2819K	277K	37272K	227K	7979	1078	490	2450	3724	3355K	98	47676K	0.16
	Btc	2944K	6263K	361K	30110K	197K	36177	48995	885	54506	735K	6614K	3154	47370K	0.16
Pred	Ctr	2290K	106K	4334	215K	24064	3151	1960	294	3245	9459	2884K	0	5543K	0.03
	Pro	2860K	124K	6406	242K	24459	882	1960	294	3443	9459	2872K	0	6147K	0.03
	Btc	4712K	116K	6112	400K	26728	113K	60461	294	16269	1115K	4765K	0	11334K	0.06
026.compress															
Base	Ctr	274K	49181	41	953K	104	104	1974	13	8541	26	78	0	1287K	0.10
	Pro	863K	164K	13	1512K	104	104	1974	13	416	26	78	0	2543K	0.21
	Btc	411K	116K	13	1167K	104	7761	63251	13	215K	26	78	0	1981K	0.16
Pred	Ctr	404K	82058	41	385K	143	26	10835	13	286	52	104	0	883K	0.07
	Pro	624K	341K	13	740K	143	26	11783	13	286	52	104	0	1719K	0.13
	Btc	544K	142K	13	463K	143	77686	78255	13	104K	52	104	0	1411K	0.11
056.ear															
Base	Ctr	23091K	15885K	6592	19769K	3077	396	693	0	198	693	6805K	0	65564K	0.04
	Pro	21644K	14567K	6392	23206K	2277	396	693	0	198	693	6175K	0	65606K	0.04
	Btc	32762K	19815K	6392	19900K	2777	58031	39550	0	198	7086	9959K	0	82552K	0.06
Pred	Ctr	9334K	6994	0	998	3077	99	0	0	297	693	5534K	0	14880K	0.01
	Pro	9311K	7894	0	798	3377	99	0	0	297	693	5534K	0	14859K	0.01
	Btc	22953K	7094	0	2398	2377	99	0	0	6894	74452	8369K	0	31416K	0.02
cmp															
Base	Ctr	1	2	4374	1	10	10	0	8	0	0	1	0	4407	0.01
	Pro	1	2	4064	1	10	10	0	8	0	0	1	0	4097	0.01
	Btc	397	3	7797	1	10	19	0	42	0	0	1	0	8270	0.02
Pred	Ctr	19	2	0	5	10	0	1	0	6	0	1	0	44	0.00
	Pro	15	12	0	5	10	0	1	0	6	0	1	0	50	0.00
	Btc	28	3	0	4	10	0	1	0	32	0	1	0	79	0.00
grep															
Base	Ctr	452	4329	0	5121	6	0	5	0	52	0	96	1	10062	0.01
	Pro	559	4053	0	5322	6	0	5	0	38	0	7	1	9991	0.01
	Btc	606	8463	0	9154	6	0	425	0	973	0	185	1	19813	0.03
Pred	Ctr	4366	4666	0	772	6	1	5	0	65	0	7	1	9889	0.02
	Pro	4049	4582	0	745	6	1	5	0	39	0	7	1	9435	0.02
	Btc	7590	12720	0	1349	6	235	420	0	1583	0	96	1	24000	0.05
lex															
Base	Ctr	39381	6429	7019	160K	4169	590	2735	958	724	856	5181	771	228K	0.02
	Pro	92560	25859	9857	269K	4510	469	2403	710	608	733	5119	771	412K	0.03
	Btc	71315	30734	7057	231K	4827	5306	12798	3070	12641	4618	11773	874	396K	0.03
Pred	Ctr	78864	13942	600	35689	2649	190	245	55	100	874	5962	819	139K	0.02
	Pro	82835	16872	23082	51618	3290	185	245	55	100	1569	5932	819	186K	0.02
	Btc	142K	55869	1737	84510	4454	32441	37525	274	1020	3865	13479	830	378K	0.04
qsort															
Base	Ctr	740K	0	106K	0	274K	42	0	0	0	42	129K	0	1250K	0.16
	Pro	697K	0	101K	0	272K	42	0	0	0	42	129K	0	1200K	0.15
	Btc	775K	0	106K	0	265K	106K	0	0	0	29399	129K	0	1413K	0.18
Pred	Ctr	133K	0	41875	0	137K	131K	0	0	0	51325	133K	0	627K	0.11
	Pro	104K	0	34671	0	103K	104K	0	0	0	46621	133K	0	526K	0.09
	Btc	171K	0	52996	0	74833	190K	0	0	0	124K	208K	0	822K	0.15
wc															
Base	Ctr	13704	35	6	19492	2	2	5	2	7	0	9	0	33264	0.06
	Pro	13313	28	3	19674	2	2	5	2	7	0	9	0	33045	0.06
	Btc	16019	29	4	24993	2	3734	10887	2	678	0	9	0	56357	0.11
Pred	Ctr	20	4	5	10	2	1	3	2	5	0	9	0	61	0.00
	Pro	16	8	3	10	2	1	3	2	5	0	9	0	59	0.00
	Btc	29	5	3	9	2	1	7	2	5	0	9	0	72	0.00

Table 8: Branch misprediction breakdown for all benchmarks.

		0	1-10	11-20	21-30	31-40	41-50	51-60	61-70	71-80	81-90	91-100
022.li												
	Base	4202K	382K	452K	182K	219K	158K	93281	166K	130K	84425	370K
	Pred	4336K	114K	126K	116K	146K	50258	104K	622K	19165	28929	122K
023.eqntott												
	Base	125M	40752K	34506K	4328K	18539K	8855K	9926K	8043K	419K	6897K	29826K
	Pred	158M	2124K	13088K	293K	163K	102K	14406	56995	130K	9391K	934K
026.compress												
	Base	3980K	2651K	661K	1011K	803K	720K	207K	196K	383K	311K	804K
	Pred	5973K	2638K	2262K	4	0	261K	225K	228K	145K	36	273K
056.ear												
	Base	1175M	86393K	38305K	14876K	19817K	43835K	18465K	7005K	4011K	14	65451K
	Pred	1216M	37355K	37200K	70	382	19	57	0	0	14	65258K
cmp												
	Base	409K	105K	0	0	0	0	0	0	0	0	11168
	Pred	407K	14	0	0	0	0	0	0	0	0	13175
grep												
	Base	445K	195K	0	0	0	1011	0	0	0	6	14128
	Pred	223K	197K	160	0	250	0	0	723	0	0	14952
lex												
	Base	10848K	1850K	234K	125K	88545	44539	38135	91013	33594	30043	490K
	Pred	823K	583K	12479	240K	6015	71519	54663	35157	47455	116K	6496K
qsort												
	Base	1483K	2539K	565K	12718	114K	1503K	295K	0	0	0	300K
	Pred	2313K	1083K	558K	78642	0	204K	0	0	0	0	198K
wc												
	Base	331K	22270	70264	34951	16	15584	0	1306	8	4033	3
	Pred	210K	14	5	0	16	8	0	3	8	0	3

Table 9: Taken frequency distribution for all benchmarks.

		10	20	30	40	50	60	70	80	90	100
022.li											
	Base	0.00	0.00	0.01	0.01	0.01	0.01	0.02	0.02	0.02	0.15
	Pred	0.00	0.00	0.00	0.01	0.01	0.01	0.01	0.02	0.03	0.14
023.eqntott											
	Base	0.00	0.00	0.00	0.00	0.00	0.00	0.01	0.01	0.01	0.15
	Pred	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.01	0.01	0.12
026.compress											
	Base	0.00	0.01	0.01	0.01	0.02	0.03	0.03	0.04	0.07	0.31
	Pred	0.01	0.01	0.02	0.02	0.02	0.03	0.03	0.04	0.06	0.27
056.ear											
	Base	0.00	0.00	0.01	0.01	0.02	0.02	0.02	0.03	0.04	0.18
	Pred	0.00	0.00	0.01	0.01	0.02	0.02	0.03	0.04	0.04	0.09
cmp											
	Base	0.02	0.03	0.05	0.07	0.09	0.11	0.14	0.16	0.17	0.33
	Pred	0.02	0.03	0.05	0.06	0.08	0.09	0.12	0.14	0.17	0.21
grep											
	Base	0.00	0.00	0.01	0.01	0.01	0.02	0.02	0.03	0.03	0.10
	Pred	0.00	0.01	0.02	0.02	0.03	0.04	0.04	0.05	0.07	0.18
lex											
	Base	0.00	0.00	0.00	0.00	0.00	0.01	0.01	0.03	0.06	0.29
	Pred	0.00	0.00	0.00	0.01	0.01	0.01	0.01	0.01	0.02	0.33
qsort											
	Base	0.02	0.02	0.02	0.03	0.04	0.06	0.07	0.09	0.13	0.45
	Pred	0.03	0.03	0.04	0.04	0.05	0.08	0.09	0.10	0.15	0.50
wc											
	Base	0.02	0.02	0.02	0.03	0.05	0.05	0.06	0.06	0.09	0.50
	Pred	0.03	0.03	0.04	0.04	0.05	0.11	0.17	0.24	0.32	0.38

Table 10: Misprediction coverage for all benchmarks using a BTB with 2-bit counter.

			0	1	2	3-4	5-8	9-16	17-32	33-64	65-128	129-256	257+
022.li													
	Base	Br	2675K	942K	834K	792K	1187K	369K	414K	0	0	0	0
		MP br	46872	6843	9795	37061	42965	105K	182K	168K	83099	38330	2303
	Pred	Br	827K	2307K	910K	1354K	539K	1723K	338K	0	0	0	0
		MP br	8699	77905	17157	35384	48790	94316	170K	396K	136K	49752	4915
023.eqntott													
	Base	Br	119M	42025K	10517K	74928K	42437K	4393K	564K	5323	0	0	0
		MP br	1110K	2745K	923K	2510K	1528K	6969K	6538K	5229K	2411K	1034K	7587
	Pred	Br	12308K	5236K	110M	29587K	6171K	10489K	13706K	44191	60473	0	0
		MP br	28313	5820	9076	23577	2101K	37682	183K	194K	1133K	83633	1741K
026.compress													
	Base	Br	2671K	2073K	1346K	1587K	1384K	1163K	1108K	867K	0	6899	0
		MP br	16343	11972	11683	22967	70153	118K	231K	228K	321K	216K	37067
	Pred	Br	3094K	2151K	508K	1583K	2307K	952K	1108K	945K	39	9289	0
		MP br	14549	125	153	1617	70444	32889	30654	158K	165K	319K	88904
056.ear													
	Base	Br	66504K	110M	500M	440M	63641K	180M	54812K	36680K	29738K	7429K	99
		MP br	1106K	715K	209K	7329K	3980K	4435K	21368K	12436K	4204K	1706K	8069K
	Pred	Br	394M	27198K	307M	336M	119M	60810K	59685K	30493K	27265K	9909K	0
		MP br	1699	1205K	617K	1898	1235K	4792	2481K	7687	6384	19961	9297K
cmp													
	Base	Br	199K	115K	11539	117K	27589	59111	0	0	0	0	0
		MP br	5	7	3	2	8	14	23	766	375	587	2617
	Pred	Br	46	52614	131K	144K	78920	13165	0	0	0	0	0
		MP br	5	4	3	1	9	2	4	0	0	3	13
grep													
	Base	Br	471K	131K	320	33591	22061	8615	3948	118	0	0	0
		MP br	5	34	8	32	45	330	1581	1879	2503	2453	1192
	Pred	Br	270K	98732	17668	31131	21933	8820	4154	0	0	0	0
		MP br	5	78	24	40	67	102	1807	2134	2844	2194	594
lex													
	Base	Br	6802K	3911K	1165K	600K	970K	320K	180K	7265	1350	1144	0
		MP br	3980	9389	3382	5372	10799	18677	38512	60385	39225	16704	22785
	Pred	Br	559K	446K	589K	849K	5739K	362K	59628	118	231	0	0
		MP br	3185	14677	3421	2054	39177	27609	35996	39095	23024	11130	17856
qsort													
	Base	Br	1212K	456K	2393	1840K	1557K	1373K	562K	421K	6	0	0
		MP br	13	4170	13	15653	225K	15801	319K	442K	191K	31413	4478
	Pred	Br	359K	216K	14801	1623K	459K	538K	1519K	692K	14	0	0
		MP br	7	0	14	4809	5174	24833	69725	253K	211K	42847	15459
wc													
	Base	Br	348K	48759	1341	26531	50051	31671	16145	0	0	0	0
		MP br	66	11	39	592	207	6120	8089	10476	6565	1064	35
	Pred	Br	92054	23	4	34	78902	26316	13151	13151	0	0	0
		MP br	6	5	3	10	3	3	14	1	1	2	13

Table 11: Distance between branches and mispredicted branches for all benchmarks using a BTB with 2-bit counter.