Evaluating the Effects of Predicated Execution on Branch Prediction

Gary Tyson

Department of Computer Science
University of California, Riverside
Riverside, CA 92521
(tyson@cs.ucr.edu)

Matthew Farrens

Department of Computer Science
University of California, Davis
Davis, CA 95616
(farrens@cs.ucdavis.edu)

Abstract

As microprocessor designs move towards deeper pipelines and support for multiple instruction issue, steps must be taken to alleviate the negative impact of branch operations on processor performance. One approach is to use branch prediction hardware and perform speculative execution of the instructions following an unresolved branch. Another technique is to eliminate certain branch instructions altogether by translating the instructions following a forward branch into *predicate* form. Both these techniques are employed in many current processor designs.

This paper investigates the relationship between branch prediction techniques and branch predication. In particular, we are interested in how using predication to remove a certain class of poorly predicted branches affects the prediction accuracy of the remaining branches. A variety of existing predication models for eliminating branch operations are presented, and the effect that eliminating branches has on branch prediction schemes ranging from simple prediction mechanisms to the newer more sophisticated branch predictors is studied. We also examine the impact of predication on basic block size, and how the two techniques used together affect overall processor performance.

Keywords: Predication, Branch Prediction, PA-RISC, Alpha, ATOM, Pentium, PowerPC

This work was supported by National Science Foundation Grants CCR-87-06722, CCR-90-11535, CCR-94-03651, CCR-92-13627, MIP-92-57259, and grants from the SUN Microsystems and Tektronix corporations.

---

1. Introduction and Background

All architectures that support some degree of instruction level parallelism must deal with the performance-limiting effects of changes in control flow (branches). This is certainly not a new problem -- pipelined processors have been struggling with this problem for decades, and with the advent of multiple-issue architectures the problem has become even more important. Much of the problem can be eliminated if conditional branches can either be correctly predicted or removed entirely. This paper will focus on two approaches to achieving this goal, *Branch Prediction* and *Predication*.

Branch prediction is a technique used to forcast the direction a branch will take before the actual direction is known. This information is used by the instruction fetch logic in order to determine which instruction pathway to continue fetching from. The effectiveness of this technique is clearly dependent on the accuracy of the prognostication, and improving the accuracy has been an active area of research for many years [Smit81] [LeS84] [FiFr92] [YeP93]. There are two main approaches for predicting branches - static schemes, which predict at compile time, and dynamic schemes, which use hardware to try to capture the dynamic behavior of branches. In either case, if the branch is predicted incorrectly, there is a penalty that must be paid to undo the incorrect prediction and proceed down the proper path. This is often referred to as the *misprediction* penalty.

Predication, on the other hand, is a technique for completely removing conditional branches from the instruction stream via the conditional execution (or completion) of individual instructions based on the result of a boolean condition. This is a promising area of research because, in addition to removing the branch itself from the instruction stream, it potentially provides the additional benefit of improving scheduling capability. Although vector machines like the CRAY [Russ78] have long supported a type of predicated execution with their vector masks, most research on this subject is more recent [CMCW91, DeHB89].

Branch prediction and predication can also be used together in a complimentary fashion. For example, the number of clock cycles lost due to incorrectly predicted branches can be reduced if the number of instructions that are tagged with a false predicate is smaller than the branch misprediction penalty itself. In addition, overall dynamic branch prediction accuracy may improve if the branches removed by predication are some of the least predictable branches.

This paper presents an examination of the the effect of augmenting branch prediction schemes found in a number of existing processors with the ability to do predicated execution. Different predication models will be examined to determine their effectiveness in removing branches, with the focus on how predication affects the accuracy of the branch prediction schemes, the branch penalty, and basic block size. Emphasis will be placed on whether predication will benefit the newer architectural designs, with their more accurate predictors, as much as the simpler less accurate branch predictors.

In the following sections we present the different branch prediction techniques used in this study, a description of various predication schemes, an analysis of the effects of predication on branch

prediction, branch penalty and basic block size, and a conclusion.

2. Branch Prediction Schemes

Branch prediction schemes range in accuracy (and complexity) from simple static techniques exhibiting moderate accuracy (≈ 60%) to sophisticated dynamic prediction methods that achieve prediction accuracies of over 97%. In this study, seven approaches are modeled - five used in commercially available products, and two others included for the sake of completeness. The seven schemes are:

Branch Always,
the HP's Precision Architecture branch backward,
the Alpha's branch backward,
the Alpha's 1-bit counter,
the Pentium's 2-bit counter,
the PowerPC 604 2-bit counter, and
the two level adaptive method introduced by Yeh and Patt.

2.1. Branch Always

The simplest (and least accurate) of schemes predicts that all branches will be taken - the processor will always attempt to fetch instructions from the target of the branch.

2.2. HP Precision Architecture (Branch Backwards)

The Hewlett Packard Precision RISC Architecture (PA-RISC) [AADM93] uses a static prediction method for calculating the direction of instruction flow across branches termed *Branch Backward*. In this scheme, all backward branches are predicted to be taken, and all forward branches are predicted to be **not** taken. This scheme outperforms branch always in applications that contain forward branches that are not taken more often than they are taken.

2.3. Alpha (Branch Backwards and 1-bit)

The Alpha processor [McLe93, Site92] supports three different prediction methods: opcode specified hints, a branch backward strategy, and a one bit branch history table. While a given implementation of the Alpha architecture may use any or all of these methods, in this study only the last two were modeled. The branch backward strategy operates the same way as in the PA-RISC. The one-bit branch history table approach features a direct mapped, 2048 entry, single bit history table. In this scheme, the low order bits of the address of a branch instruction are used to select a one bit entry in the history table, which is in turn used to predict the branch direction. The entry is later updated to reflect the actual condition of the branch.

2.4. Pentium and PowerPC 604 (2-bit counter)

The prediction approach used by the Pentium processor [AlAv93] features a 256 entry Branch Target Buffer (BTB). Each BTB entry contains the target address of the branch and a two bit counter used to store previous branch activity associated with that address. The BTB is 4-way set associative and uses a random replacement strategy. Branches which are not in the BTB are assumed to be **not** taken.

The PowerPC 604 processor [94] employs a fully associative, 64 entry BTB and a separate direct mapped, 512 entry, 2 bit branch history table in the following manner: When a branch is encountered, the BTB is searched (by branch address) in an attempt to locate the target address of the branch. If an entry is found corresponding to the branch address, the branch is predicted **taken**; otherwise it is predicted **not taken** and instruction flow continues sequentially.

Once the branch outcome has been determined, the branch history table is updated. If the resulting history table value will predict **taken** on the next execution of the branch, then the branch address is added to the BTB. If the history table value predicts **not taken**, then the branch address is removed from the BTB (if it currently resides there).

2.5. Two Level Adaptive

The final branch prediction strategy modeled is the two level adaptive strategy developed by Yeh and Patt, and first presented in [YeP91]. This strategy requires considerably more hardware resources than the other methods, but also provides substantially greater branch prediction accuracy.

This scheme features a set of branch history registers in addition to a branch history pattern table. When a branch instruction is executed, the lower bits of the branch address are used to index into the set of history registers. Each history register (implemented as a shift register) contains information about the branch history of those branches that map into that register. This information is then used to index into the branch history pattern table, which contains the information necessary to determine the actual branch prediction.

In the model used in this study the branch history register file contained 512 13-bit entries (which limits the branch history pattern table to 8192 entries) and each branch history pattern table entry contains a 2-bit counter. A two-level scheme similar to this has recently been implemented in the Intel P6 (Pentium Pro) architecture.

3. Predicated Execution Models

As mentioned in the introduction, predicated execution refers to the conditional execution (or completion) of instructions based on the result of a boolean condition. Several different approaches to providing predicated execution have been proposed and implemented. These approaches fall into two broad categories, referred to as **restricted** (or *partial*) and **unrestricted** (or *complete*).

In the restricted model a limited number of new predicate instructions are introduced which are used to explicitly delay the effect of executing a statement on program variables. In the unrestricted predication model, on the other hand, *all* instructions can be predicated. This can be accomplished in a number of ways. One way is to include an additional operand field for each instruction, as was done in the Cydra 5 [RYY89]. Another way is to introduce a special instruction which controls the conditional execution of some number of instructions following the special instruction. An example of this approach is seen in the *guarded execution model* proposed by Pnevmatikatos and Sohi [PnSo94].

This section concludes by presenting an overview of 4 different existing predication models; the one used in the Alpha, the one used in the PA-RISC, the guarded execution model, and the one used by the Cydra 5. A simple example will be used to demonstrate the use of each method of predication. The source statement and an intermediate form without predication is shown in figure 1. Each example code assumes that the variables **a**, **b** and **c** are live-in values located in registers **R4**, **R5** and **R6** respectively; **R3** contains the value **ZERO** and is used by the compare operation.

```
                                        ; HLL:  if (b != 0)   a = a * b + c;
        compare     R5, R3              ; test condition (b != 0)
        beq         L1                  ; branch over expr if b equals 0
        mult        R4 <- R4, R5        ; execute expr (temp = a * b)
        add         R4 <- R4, R6        ; execute expr (a = temp + c)
L1:
```

Figure 1. Example Intermediate code for HLL statement with no Predication

3.1. Alpha Conditional Move

The Alpha processor supports the restricted model of predication via the use of a *conditional move* instruction - if the condition is satisfied then the register movement is allowed, otherwise a state change is prevented. This approach requires a minimal modification to an instruction set, since a standard move instruction only requires 2 operands (source and destination), leaving one field free to specify the conditional value in the conditional move. This approach toward partial predication has increased in popularity among new implementations; the Intel P6 (Pentium Pro) architecture has introduced conditional move instructions for both floating point and integer operations into the *x86* architecture.

The Alpha compiler uses the conditional move instruction in the following way: An expression calculation is rescheduled to preceed a conditional branch, and is modified to write its result to a free register instead of a live program variable. A conditional move of the temporary value into the live register is then used in place of the original branch instruction. After this code executes, the original

destination of the expression will contain the result if the condition was satisfied; if the condition is not satisfied, then the conditional move operation is not performed, and the live variable remains unchanged.

There are several restrictions on this form of predication. For example, the compiler must ensure that no exceptions (e.g. division by zero) will be generated by rescheduling the expression calculation code. There are also instructions that cannot be predicated, such as some memory access instructions and flow control operations. Finally, the compiler must allocate free registers to store any results prior to the conditional move instruction(s). This may not be feasible if it results in register spilling.

Figure 2 shows how a single free register (R4) must be allocated to calculate the expression before the condition is evaluated. This free register is used to temporarily store the result of the calculation, the condition is then evaluated, and finally the contents of R4 are conditionally moved into the original register (R1) if and only if the condition is satisfied. Note that the net result is the same as the original code, but the branch delay (and potentially the branch misprediction penalty) is removed in favor of always executing the expression and conditionally reclaiming the result.

```
                                          ; HLL:  if (b != 0)   a = a * b + c;
          mult          R7 <- R4, R5      ; use free register R7 to store
          add           R7 <- R7, R6      ; the temporary result of the expr
          compare_ne    R8 <- R5, R3      ; test condition, result in R8
if R8     move          R4 <- R7          ; conditionally move R7 to R4
```

Figure 2. Example Intermediate code for HLL statement using Conditional Move

3.2. HP-PA Conditional Skip

The HP Precision Architecture uses *instruction nullification* to provide a less restricted form of predication. In this approach, control flow and arithmetic instructions can specify whether the instruction *following* them should execute. The ability of arithmetic instructions to nullify the following instruction allows the compiler to remove branch instructions. A transformation similar to that of the Alpha's can be performed with the added capacity to include additional *skip* instructions over exception producing instructions.

In the example of this approach shown in figure 3, register R4 is again used to store the temporary value of the expression. The condition is then evaluated, which also determines whether or not the following instruction should be executed. The following instruction that will get conditionally executed in this case is a move operation from R4 to R1. As was the case in the Alpha, whether the condition is true or false, the state of the machine remains consistent with the original code.

```
                                                ; HLL:  if (b != 0)   a = a * b + c;
          mult          R7 <- R4, R5            ; use free register R4 to store
          add           R7 <- R7, R6            ; temporary result of the expr
skip on   compare_ne    R0 <- R5, R3            ; test condition, skip next if false
          move          R4 <- R7                ; move R7 to R4 (if not skipped)
```

Figure 3. Example Intermediate code for HLL statement using Arithmetic Skip

3.3. Guarded Execution Model

Pnevmatikatos and Sohi [PnSo94] propose the use of a **guard** instruction to control the execution of a sequence of instructions. A guard instruction specifies two things - a condition register and a mask value to indicate which of the following instructions are dependent on the contents of that condition register. The processor hardware then uses this information to create a dynamic *scalar mask* which is used to determine whether a given instruction should be allowed to modify the state of the processor. Support for multiple guards can be provided by allowing additional guard instructions to modify those entries in the *scalar mask* that have not been previously marked for elimination. This approach is reminiscent of the vector mask register approach used on earlier vector processors [Russ78], with the bit mask controlling the issue of a sequence of instructions in the instruction stream instead of the ALU operations in vector instruction. It is unclear how the nullification capability or the guarded branch instruction will migrate to multi-issue implementations. These instructions are similar in concept to the load delay slots found in early RISC designs and may have similar deficiencies.

The example code in figure 4 shows the result of the compare instruction being saved in R8. This result is then used by the GUARD instruction to set the appropriate bits in the *scalar mask*. The mask value used in the GUARD instruction (11) indicates that the next two instructions are to be conditionally executed based on the contents of register R8. The two guarded instructions then perform the calculation if and only if the guard (R8) is false. The use of a bit pattern field in the GUARD instruction allows the mixing of conditionally executed instructions and instructions independent of this guard, providing greater flexibility for scheduling (this feature is not shown in this example).

```
                                        ; HLL:  if (b != 0)   a = a * b + c;
        compare_ne      R8 <- R5, R3    ; test condition, result in R8
        GUARD           R8,  0b11       ; establish guard on R8, next 2
                                        ; instruction only (0b11)
        mult            R4 <- R4, R5    ; multiply if guard false
        add             R4 <- R4, R6    ; add if guard false
```

Figure 4. Example Intermediate code for HLL statement using Arithmetic Skip

3.4. Cydra 5

The Cydra 5 system [RYY89] supports the most general form of predicated execution. Each Cydra 5 operation can be predicated by specifying which of the 128 boolean predicate registers contains the desired execution condition. An operation is allowed to complete (write-back) and modify the state of the machine if the selected predicate register evaluates to non-zero. Since all instructions can (and must) reference a predicate register, all instruction sequences can be predicated.

In the example code in figure 5, the comparison is performed as it was in the original code (figure 1), with the result being *stuffed* into predicate register P0. The multiply and add instructions then specify conditional execution based on the contents of predicate register P0. This implementation results in the removal of the branch operation altogether, as opposed to its replacement with a specialized instruction.

```
                                                ; HLL:  if (b != 0)   a = a * b + c;
                compare_ne      R8 <- R5, R3    ; test condition, result in R8
                stuff           P0 <- R8        ; move condition to predicate P0
    if P0       mult            R4 <- R4, R5    ; multiply if P0 != 0
    if P0       add             R4 <- R4, R6    ; add if P0 != 0
```

Figure 5. Example Intermediate code for HLL statement using Predication Registers

4. The Experiments

Studies have shown that a large percentage of branches are to a destination less than 16 instructions away ([HePa90] pg 106). However, these studies look only at total branch distance, not directed distance. In order to examine the relationship between branch prediction accuracy and predication, the accuracy of the prediction hardware on the class of branches that can be predicated (short *forward* branches) had to be measured.

4.1. The Simulation Environment

The benchmarks selected for this experiment were the fourteen floating point and five of the integer programs from the SPEC92 suite of programs. Each program was compiled on an Alpha-based DEC 3000/400 workstation, using the native compiler and **-O2 -non_shared** compiler flags. The ATOM [SrEu94] toolkit was used to generate and help analyze the data gathered for this study. In order to perform the study, branch instructions were instrumented and the branch prediction schemes outlined in section 2 were simulated; predicate transformation was then implemented on the instrumented code to account for the removal of some branches.

As mentioned earlier, the Alpha compiler itself is able to eliminate some branches using the conditional-move instruction. In order to make fair comparisons between the different predication schemes, these operations were transformed back into branch operations by trapping those instructions and treating them as conditional branches of unit distance. (Almost all of the conditional-move instructions were found in system libraries.) Table 1 presents benchmark program details (the inputs used, the total number of instructions executed, the number of branches executed and the total number of jumps executed).

Table 1: SPEC Benchmark Information

Benchmark	Input	Instructions	Branches	Jumps
alvinn	*	3,554,909,199	167,524,380	30,274,280
doduc	doducin.in	1,149,864,381	84,622,229	13,463,476
ear	args.ref	17,005,800,990	896,667,195	481,185,420
fpppp	natoms	4,333,190,502	115,692,876	6,508,987
hydro2d	hydro2d.in	5,682,547,494	347,131,563	9,876,849
mdljdp2	input.file	6,856,424,748	320,407,455	71,051
mdljsp2	input.file	2,898,940,578	352,481,636	1,024,674
nasa7	*	6128388226	165,573,273	23,201,959
ora	params	6,036,097,727	365,260,354	87,096,763
spice2g6	greycode.in	16,148,172,367	1,933,779,718	93,549,064
su2cor	su2cor.in	4,776,761,988	178,137,872	30,247,847
swm256	swm256.in	11,037,397,686	182,031,528	407,016
tomcatv	N/A	899,655,110	30,183,243	27,571
wave5	N/A	3,554,909,199	167,524,380	30,274,280
compress	in	92,628,682	12,379,188	502,775
espresso	bca.in	424,397,814	74,371,674	3,603,641
espresso	cps.in	513,006,475	83,519,772	4,051,876
espresso	ti.in	568,262,371	89,791,260	5,083,215
espresso	tial.in	983,531,458	167,375,187	12,731,193
gcc	cexp.i	23,535,507	3,098,517	506,777
gcc	jump.i	143,737,833	6,901,206	1,063,385
gcc	stmt.i	51,359,229	19,426,084	2,947,663
xlisp	li-input.lsp	6,856,424,748	867,585,743	316,747,153
eqntott	int_pri_3.eqn	1,810,542,679	199,198,053	9,669,793
sc	loada1	1,450,169,424	277,556,127	29,920,923
sc	loada2	1,634,276,007	328,405,132	46,912,615
sc	loada3	412,097,081	91,802,679	6,375,876

4.2. Branch Characteristics of SPEC Benchmarks

Examining the branch characteristics of these programs in more detail reveals why predication may help performance. Figure 6 shows the total number of branches taken as a function of the distance (in instructions) between the branch instruction and the branch target, for both the integer and floating point benchmarks. These figures include both conditional and unconditional branches, but exclude subroutine calls.

Notice that in both integer and floating point applications there is a high percentage of short forward branches, which are prime candidates for predication. Also of note is the percentage of branches taken; almost all backward branches are taken, while many forward branches are not taken. This is as expected [HePa96]. Of the forward branches, those between distance 1 and 12 show the greatest frequency of execution.

Figure 7 shows the branch prediction accuracy of the 7 branch prediction schemes analyzed, separated into three components: the prediction accuracy of all backward branches, the accuracy of branches in the range 0-12, and the accuracy of forward branches of distance 13 or more. This figure shows that the prediction schemes are all able to predict backward branches quite accurately, because backward branches are almost always taken (figure 6), which is what the static schemes assume and

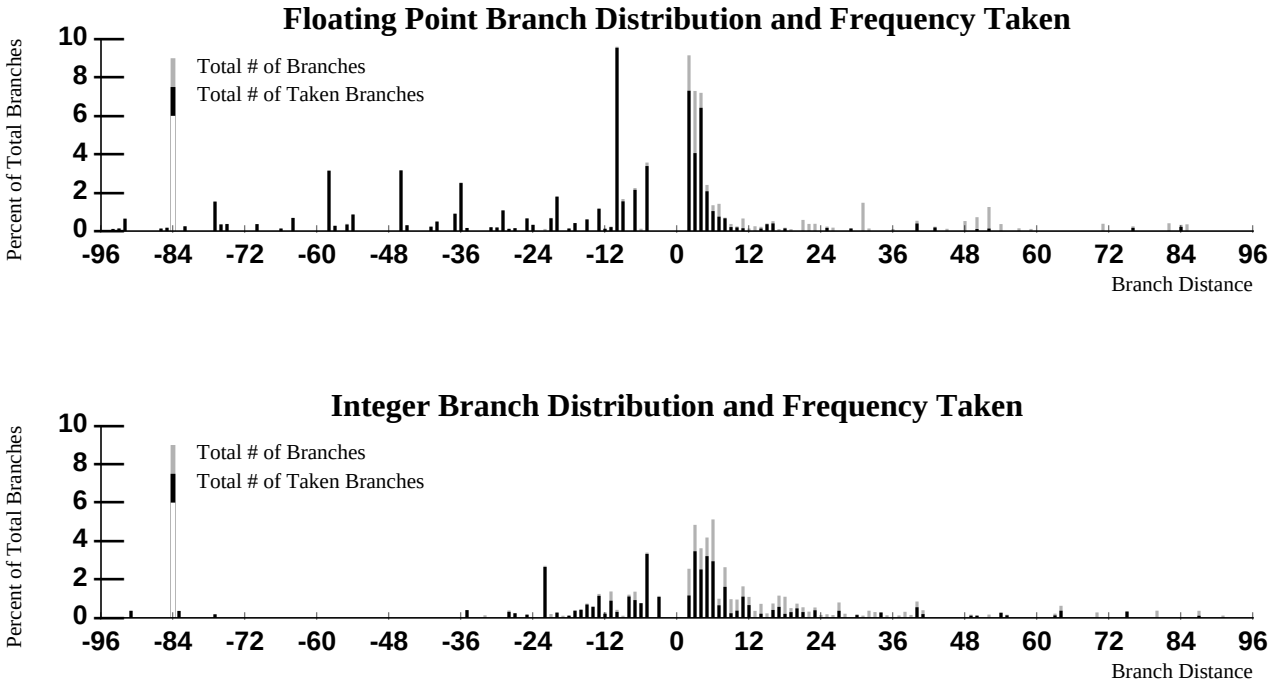

Figure 6: Branch Distribution and Frequence Taken

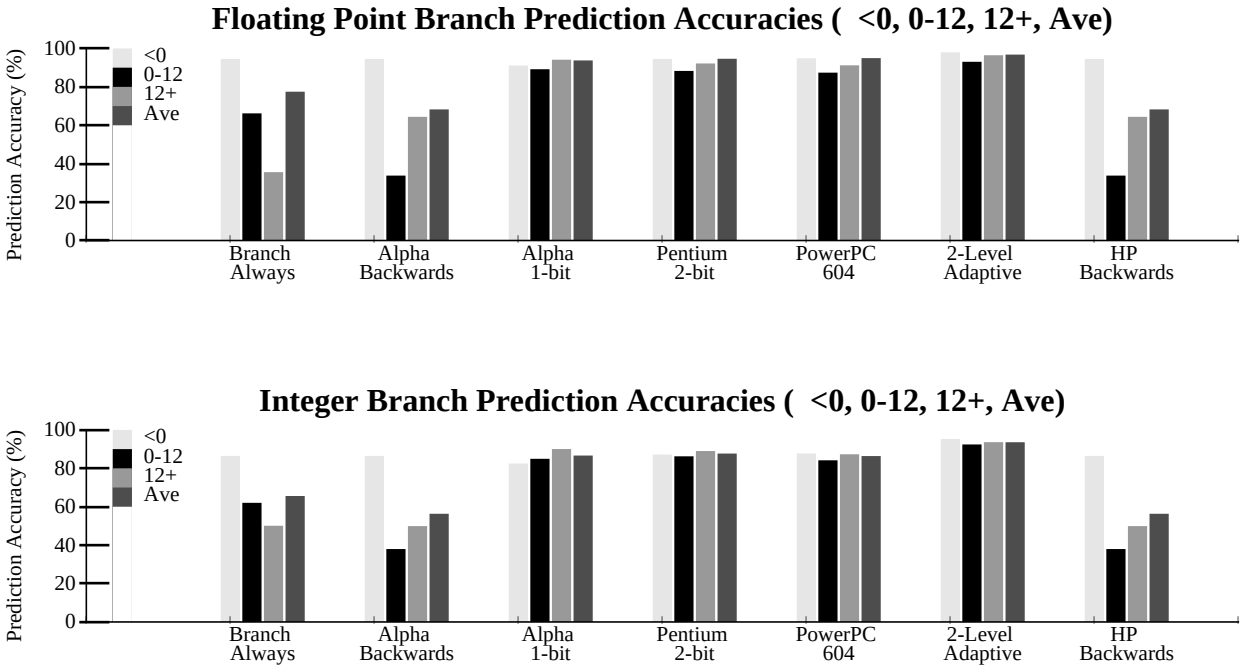

Figure 7: Branch Accuracies Before Predication

the table based schemes quickly determine. In addition, for most prediction schemes the accuracy can be seen to be the lowest for branch distances in the range 0-12. This is because these branches have the highest rate of changing their branch condition between branch executions (this was measured by counting the number of times the branch chose the opposite path from its previous execution and averaging this with total executions of that branch). There were several benchmarks that showed greater predictability in the 0-12 range than outside this range, which was generally due to branches around function exit conditions that were almost always taken. Since the 0-12 range branches are those most eligible for predication, and approximately 1/3 of all branches fall in this range, it appears that removing these branches may lead to an overall improvement in branch prediction accuracy.

However, branch accuracy alone is not a sufficient metric. Some schemes incur a greater penalty for predicting incorrectly than others, and this information must be considered. Table 2 presents the different branch prediction schemes and the number of clock cycles and instruction slots lost each time a branch is incorrectly predicted. In a single issue machine the cycle penalty and the instruction penalty are the same; in multiple issue architecture the cycle penalty is multiplied by the issue width to determine the number of instruction slots lost due to misprediction.

Table 2. Branch Misprediction Penalty

Branch Prediction Scheme	**Penalty**	
	Cycles	Instr
Branch Always	1	1
PA-RISC Branch Backward	1	1
Alpha Branch Backward	4	8
Alpha 1-bit	4	8
Pentium 2-bit	3	6
PowerPC 604	3	12
2-Level Adaptive	3	12

4.3. Predication Models

In order to measure the effect of predication on branch prediction, the following two translation schemes were studied:

1. The *Aggressive* scheme translates all short forward branches with a branch distance less than or equal to the *predication distance* to predicate form. This allows the removal of a significant portion of the branch misprediction penalty, and also enables the joining of small basic blocks into much larger ones. This approach will translate the greatest number of branches.
2. The *Restricted* scheme transforms only branches that contain no instructions between the branch in question and the branch's target that may generate exceptions or change control flow (load, store, branch and divide instructions). This is modeled in order to analyze the effectiveness of some existing predication mechanisms (e.g. Alpha).

4.4. Characteristics of Predication Techniques

Predication has the potential to remove all forward branches. However, each time a branch that jumps forward over a number of instructions is transformed into a predicated set of instructions, the total number of instructions that must be executed is increased. For example, if a branch that jumps over 5 instructions is transformed via predication, then the 5 instructions that were skipped if the branch was taken will have to be fetched and executed. We refer to this cost as the *Predication Cost*, which is calculated as the number of times a branch is taken multiplied by the branch distance. Clearly longer branch distances incur much larger predication costs.

Table 3 shows that a small number of branches are dynamically executed a large number of times. In this table, the **Total** column contains a count of the total number of static branches (both

conditional and unconditional) in each benchmark program. The *100%* column shows how many of the static branches in the Total column could be removed via predication using the aggressive (*A*) and Restricted (*R*) strategies. Since the aggressive scheme can predicate all instructions out to the predication distance, the *A* column actually contains a static count of the number of branches that have a branch distance between 0 and 12. The other three columns contain the number of static branches that are responsible for the listed percentage of the total. So, for example, the table shows that (for alvinn) there are 3292 branches that the aggressive predication scheme could remove. However, 80% of the beneficial effect of predicating all 3292 can be achieved by predicating only 13 of them. Similarly, 95% of the effect of predicating all 3292 can be achieved by predicating only 22 of them. Obviously, very few branches need to be predicated to achieve almost complete coverage of the available branch executions. Yet, as seen in the next section, the removal of these branches can provide a substantial reduction in the branch penalty.

From the table we see that *Aggressive* predication is capable of removing approximately 30% of the total branches in the program, while *Restricted* predication is capable of removing a much smaller number (only about 5%). The inability of the Restricted scheme to predicate loads and stores account for about half of the difference, with the remaining restrictions due to the appearance of additional branches (that could not be predicated) between the original branch instruction and its branch target address. The effect of this restriction is remarkably similar for each benchmark.

Table 3: Removal of Branches by Predication

Bench-Mark	Total Branch	80%		95%		99%		100%	
		A	R	A	R	A	R	A	R
alvinn	11150	13	6	22	10	35	11	3292	580
doduc	10129	7	2	49	8	71	12	3346	495
ear	2599	2	0	2	0	2	0	831	121
fpppp	9202	9	8	17	13	26	17	3147	478
hydro2d	10152	17	2	32	11	41	14	3250	494
mdljdp2	10034	8	3	13	6	17	6	3368	498
mdljsp2	10013	8	0	10	0	12	0	3343	482
nasa7	9660	36	1	112	7	175	9	3155	470
ora	8749	3	3	3	3	4	3	3024	433
spice	12736	13	2	22	5	36	5	3648	588
su2cor	10428	4	2	7	3	8	4	3276	492
swm256	8976	2	2	3	3	3	3	3043	438
tomcatv	8060	2	2	2	2	2	2	2747	412
wave5	11150	13	6	22	10	36	12	3292	580
compress	1539	3	1	4	1	5	1	490	66
eqntott	4254	1	0	4	0	15	1	1194	166
espresso	5803	10	0	49	1	90	3	1879	275
gcc	21500	315	17	775	58	1345	99	6188	569
sc	6584	21	1	51	5	110	16	1701	271
xlisp	3391	12	0	27	0	41	0	197	14

5. Analysis

The removal of branches via predication will affect several of aspects of program performance. Primarily, performing predicate transformations will affect two things; the number of cycles spent dealing with a program's branches, and a program's average basic block size. In order to quantify these effects, we investigated the relationship between predication, branch prediction accuracy, the branch penalty, and the average basic block size.

Branch prediction accuracy is an important measure of how well an architecture deals with branches; however, just as cache hit rates are not in and of themselves an adequate measure of cache performance, prediction accuracy by itself ignores some essential components of overall branch handling performance. For example, the number of cycles lost due to a mispredicted branch (the *misprediction penalty*) varies from machine to machine, and grows with pipeline depth. Architectures that issue multiple instructions per cycle also pay a higher misprediction penalty because, even though they may lose fewer cycles, each cycle is capable of doing more work.

Branch prediction accuracy figures also fail to include the total number of branches that the figure is based on. For instance, if half of the branches are removed from the execution stream via predication, maintaining the same prediction rate will in reality yield a 50% decrease in the cycles lost to mispredicted branches. Therefore, any attempt to measure the relationship between predication and branch performance must include an examination of the branch penalty.

Finally (and perhaps most importantly), the effect that predication has on average basic block size is important because it directly relates to the amount of parallelism that can be extracted by the code scheduler. By increasing the number of instructions that are executed between branch decisions, the scheduler (hardware and software) can more easily find independent instruction to fill issue slots in the pipeline.

5.1. Effects of Predication on Branch Prediction

Using predication to remove short forward branches of distances less than 2, 4, 6, 8, 10, and 12 instructions from the instruction stream affects a branch prediction scheme performance in several ways. For example, figure 7 showed that the branches that predication removes exhibit less than average prediction accuracy. In addition, for those schemes that perform table driven dynamic prediction, the reduction in the number of branches encountered can ease contention for branch table access. The impact of predication on branch prediction accuracy can be seen in Figures 8 and 9. The accuracy of branches in the range 0-12 is not displayed in these figures, because the number of branches remaining in that range after predication decreases to the point where meaningful representation is not possible.

For the sake of clarity, these figures show the effects of predication (by distance) on branch *misprediction* rates (the change in misprediction rate relative to branch accuracy before predication). So, for example, a level of 80% indicates that the prediction accuracy has improved because the

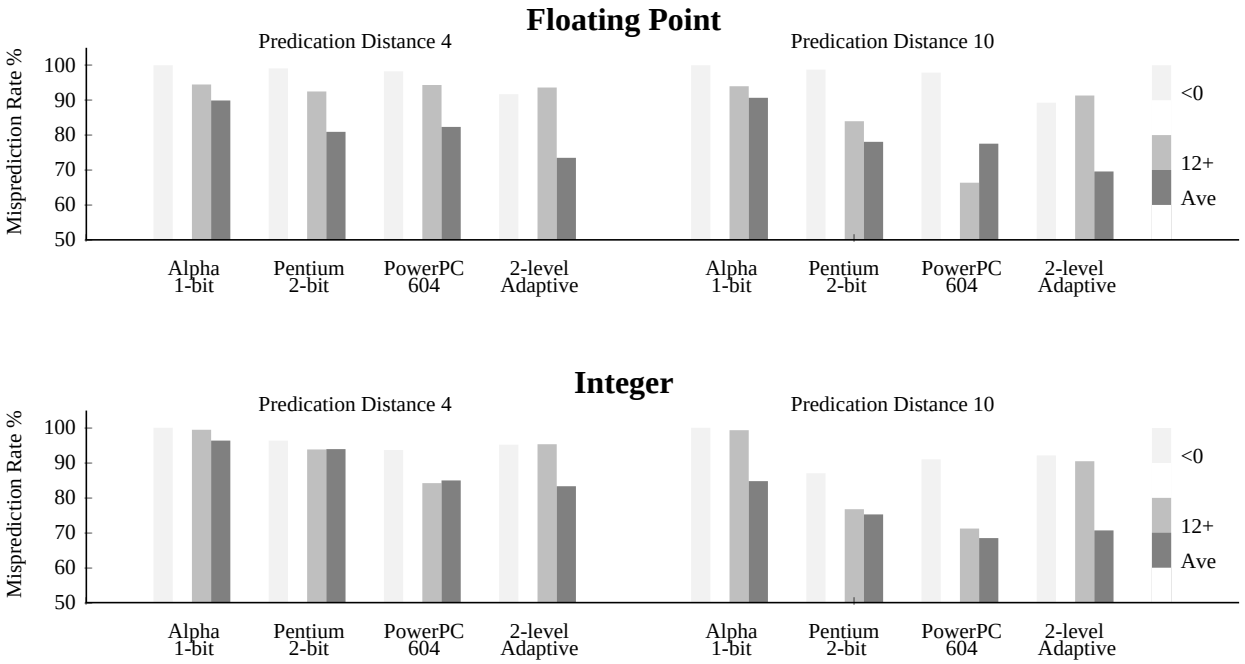

Figure 8. Change in Branch Misprediction Rate for Aggressive Predication

misprediction rate has been reduced by 20%. These figures show that several schemes experience reductions of up to 30% in the misprediction rate. It is interesting to note that many of the schemes do not experience significant accuracy changes. This indicates that many current table driven schemes contain sufficient state information space and do not suffer from resource (table space) contention.

5.2. Effects of Predication on Branch Penalty

A branch handling scheme that has a very high branch prediction rate but a correspondingly high branch penalty may very well perform more poorly than a processor with a shorter pipeline and a simpler scheme. In order to study the relationship between the branch penalty and predication, we define the following two definitions:

$$\textit{branch penalty} = \textit{cost of executing a branch} * \textit{total number of branches executed} + \textit{misprediction penalty} * \textit{number of mispredicted branches}$$

$$\textit{predication cost} = \textit{Branch distance of removed Branch} * \textit{branches taken pre-removal}$$

Calculating the branch penalty is fairly straightforward. Each branch must be issued (using an issue slot), and each time a branch is predicted incorrectly, some number of clock cycles are lost to squashing instructions in progress and redirecting the fetch logic (the *misprediction* penalty). This

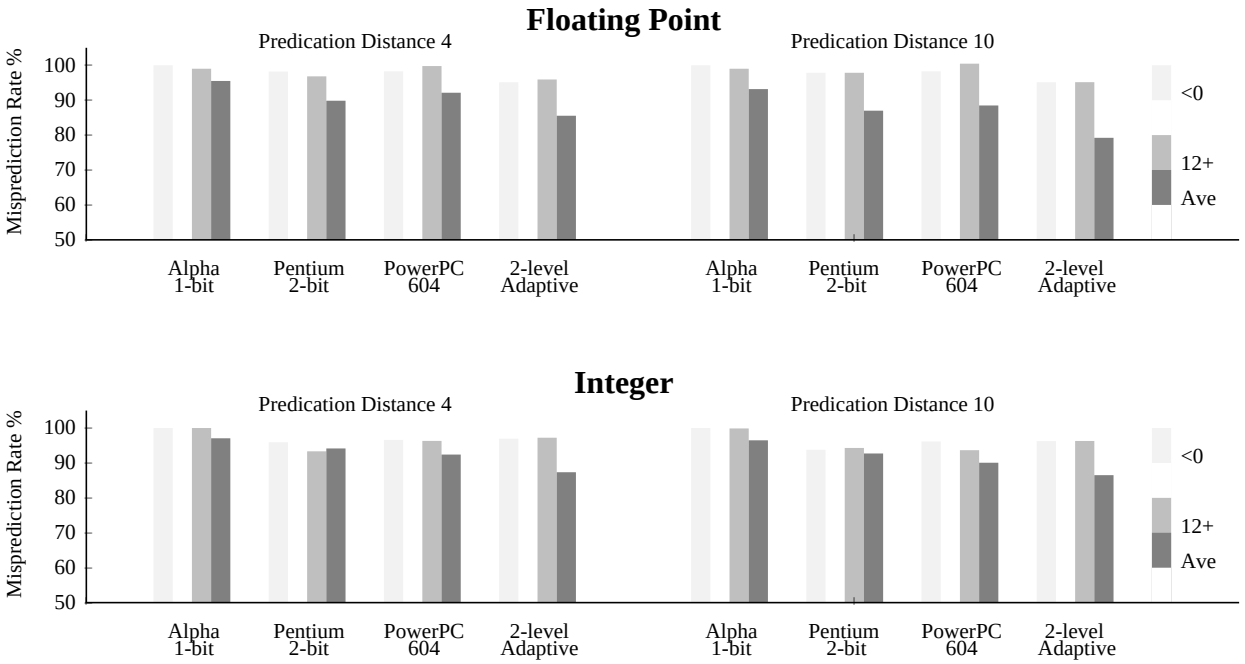

Figure 9. Change in Branch Misprediction Rate for Restricted Predication

misprediction penalty varies from scheme to scheme (see table 2), and is measured in *cycles*, and not instructions. This will clearly impact a multiple-issue machine more than a pipelined machine.

Calculating the predication cost is more complicated and depends heavily on the predication mechanism employed. The basic cost is the number of instructions issued that are not performing useful work. In some predication schemes, an additional instruction is required to specify the predicate value (this is analogous to the branch issue required for each branch execution). Other predication mechanisms avoid the need for inserting additional instructions by utilizing additional source fields in the instruction format to convey the predicate value. Another source of extra cost due to predication is the fact that all predicated instruction are issued regardless of their predicate value. If a forward branch of distance N is taken M times, for example, then N*M instructions are bypassed and not executed. However, if predication is used to remove the branch instruction, then the N instructions will no longer be bypassed, but will have to be fetched and executed. This cost will have to be included.

For each predication scheme, Figure 10 shows the total branch penalty calculated for each predication distance. Note that as the predication distance increases, the number of cycles spent handling branches decreases. This is due to two factors -- predication is removing branches and therefore reducing the first term of the branch penalty equation, and the branch prediction accuracy of the remaining instructions is increased (and the misprediction rate is correspondingly decreased) forcing the second term of the equation down as well.

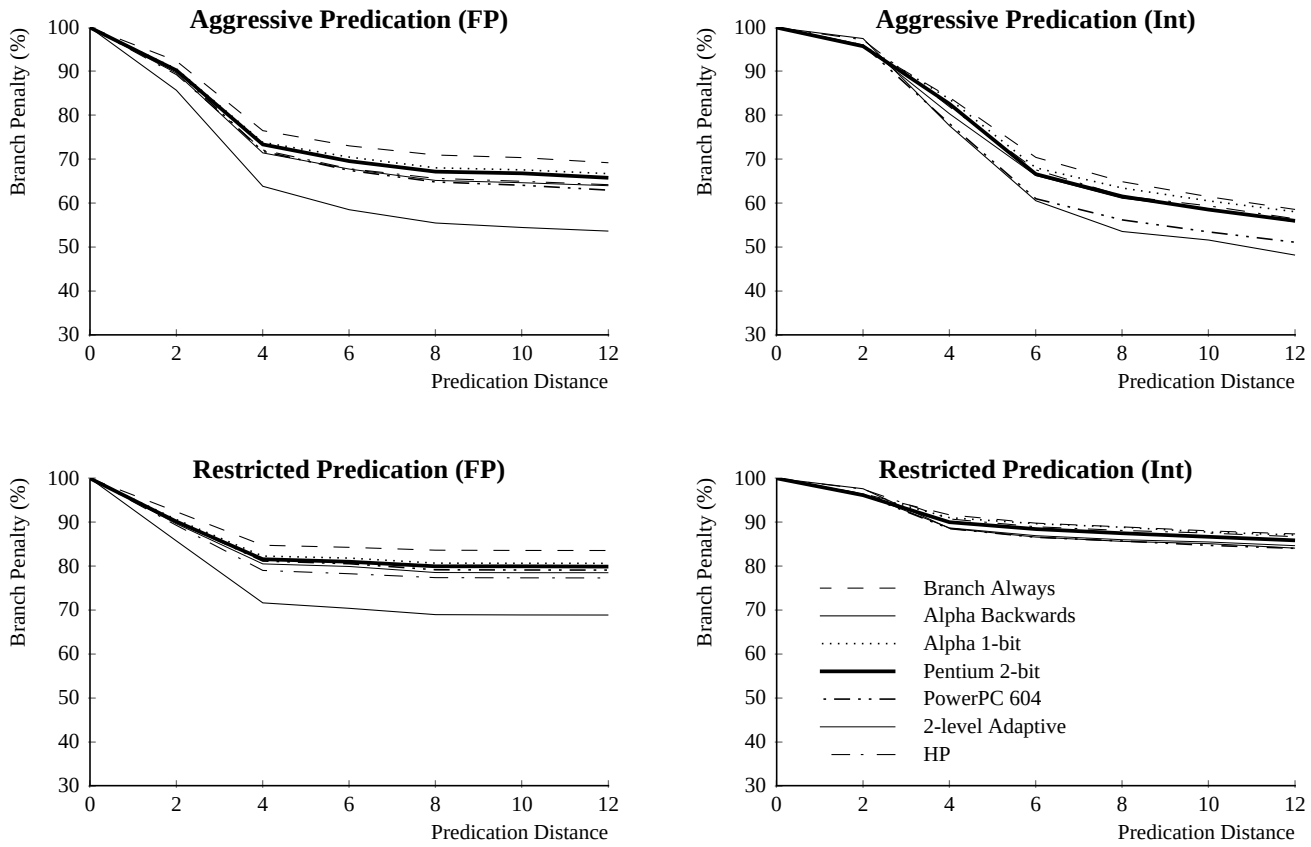

Figure 10. Percent Change in Branch Penalty for Each Predication Distance

Branch penalty accounts for only a portion of the effect of predication; the cost of predication must be factored in as well in order to determine the overall effect on performance. Figure 11 shows what happens when this number is included. We see that the greater the predication distance the less effective predication is in reducing the overall number of instructions issued. This is due to the fact that the predication cost is directly related to the distance of the branch, unlike the branch penalty (which is independent of the branch distance). In addition, each of the prediction schemes can be seen following the same pattern, a drop in the number of instructions issued as very short branches are predicated, followed by a gradual rise as predication cost climbs for greater branch distances.

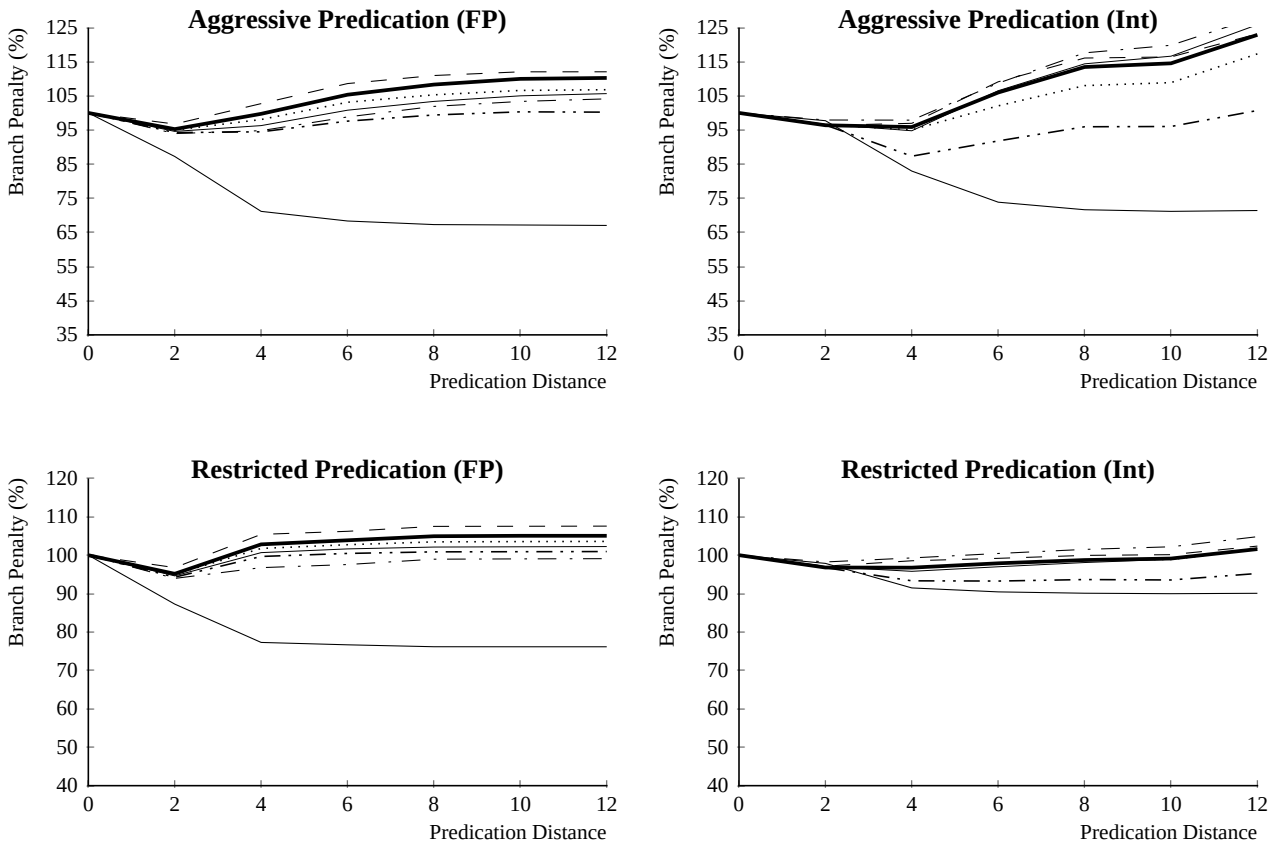

Figure 11. Total Cost of Branches (Branch Penalty + Predication Cost)

There are two particularly interesting things to note in this figure. First, the Alpha utilizing a branch backward prediction scheme clearly behaves differently than the other approaches. In this scheme forward branches are predicted *not* taken, so a misprediction penalty is incurred each time the branch *is* taken. If this misprediction penalty is greater than the predication penalty generated by transforming the branch, performing the transformation will result in improved performance. Since the predicate penalty is smaller than the misprediction penalty for all branch distances less than 8 instructions (4 cycle misprediction penalty * 2 issues), a substantial performance improvement can be made by predicating **ALL** branches up to distance 9 on an Alpha with this configuration.

The second point demonstrated in figure 11 is that newer architectures may benefit from predication more than old designs even though the newer branch prediction schemes are more accurate. Intuitively, it is clear that these multi-issue architecture would receive a greater benefit from predication after scheduling the new bigger basic blocks. However, due to the greater misprediction penalty, this improvement is seen before scheduling as well. In fact the PowerPC

604 shows an overall improvement in instruction issue even in the most aggressive scheme up to branch distance 12, exclusive of the much greater benefit that the removal of 30% of the branches will have on static and dynamic scheduling.

Finally, figure 11 shows that because the *Restricted* scheme is more conservative in approach, the corresponding effectiveness is limited. Very few branches of distance greater than 4 are predicated. This result may be of limited importance, though, since because the newer architectures can tolerate the greater predication cost caused by increasing the branch distance, the current trend is towards more and more aggressive predication.

5.3. Effects of Predication on Basic Block Size

Control dependencies restrict the ability of multi-issue architectures to fill instruction slots. Branch prediction schemes are used to help alleviate this problem by providing a set of candidate instructions with a high probability of execution that can be used to fill vacant issue slots. Unfortunately, since branch points help define the size of a basic block, many compiler transformation are still precluded.

By removing a control dependency entirely via predication, the compiler is provided more flexibility when reordering the code sequence, and can often achieve a more efficient code schedule. We have shown that an average of 30% of branch instructions are short forward branches and are therefore good candidates for predicate transformation. The removal of these branches will leads to a significant increase in basic block size, and thus an increase in the efficiency of the scheduler.

Figure 12 shows the results of applying the *Aggressive* transformation to the integer SPEC benchmarks. This figure shows that as the predication distance increases, the basic block size increases as well, and that applying the predication transform to branch instructions of distance less than or equal to 12 provides a 40% increase in basic block size. This is nearly half of the total increase achievable if **all** forward branches were removed. As usual, the *Restricted* predication scheme provides a much more limited improvement in basic block size (4% to 6% increase).

It is also possible to use predication in conjunction with other transformations to remove even a wider range of branches. For example, a loop unrolling transformation removes backward branches by duplicating the body of the loop and iterating (and therefore branching) fewer times. If the number of iterations can be determined at compile time, then the unrolling is a simple duplication. However, if the number of iterations cannot be determined at compile time, then care must be taken to avoid overshooting the terminating condition of the loop. This can be accomplished by placing conditional branches exiting the loop between the duplicated copies of the loop body, and using predication to transform these conditional branches into predicated

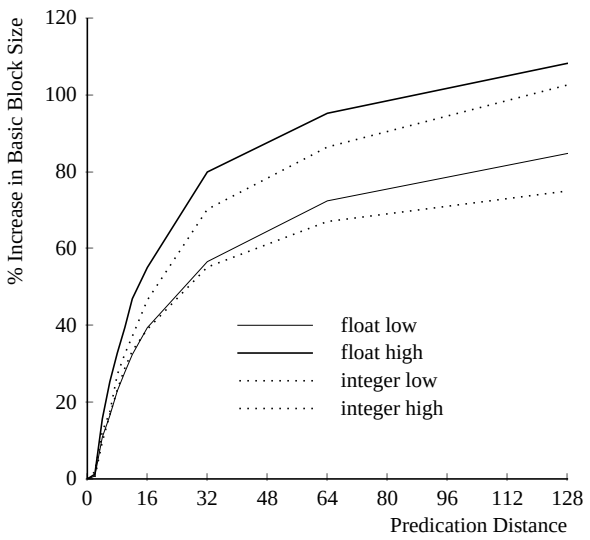

Figure 12. Affects of Predication on Average Block Size

instruction sequences. Using these two transformations together, the compiler can modify a small loop containing a maze of *if-then-else* conditionals into a long sequence of predicated instructions that can then be more efficiently scheduled to fill instruction slots.

5.4. Other Scheduling Issues

In [MHBG94], the effect of predication on branch prediction during hyperblock formation was studied. A hyperblock is a collection of basic blocks which has a single entry point and one or more exit points. When hyperblock formation is applied to an inner loop, predication can be used to eliminate any branches from the loop. Loop structures allow for the removal of even the backward branches; this can be accomplished by inserting a loop-back branch at the end of the loop and then re-targeting all current loop-back branches to this newly created one. This has the effect of translating those backward branches to forward branches (the new target is at the end of the loop); these can then be predicated in the usual manner.

Their results show a similar effect on branch penalty reduction as well as the improvement of branch predictors. This effect is more pronounced in those benchmarks which execute primarily in their inner loops. For example, in *ear* the increased use of predication during hyperblock formation allows for over a 70% reduction in the number of mispredicted branches.

5.5. Predication Without Predicates?

The importance of predication is likely grow as branch penalties increase, because the use of predicates allow for large branch penalties to be removed in favor of much smaller predication costs. However, many future processor implementations will need to execute instruction sets that do not support predication. In [TFRP95] we explore the feasibility of modifying the method used to recover from mispredicted branches to allow the branch penalty to be reduced in much the same manner as predication without the necessity of predicates.

In [TFRP95], a mechanism is presented by which those short forward branches that are incorrectly predicted to be **NOT-TAKEN** can be flushed in a manner which reduces the penalty to that of predication (the number of instructions from the branch to the target). This is done by only flushing those instructions that fall between the branch and the target; all instructions past the target of the branch remain in the pipeline and continue execution normally. This approach can be especially effective in architectures with a large set of pipeline stages prior to the execution or register rename stage (e.g. the 7 stages found before register rename in the P6 architecture).

Branch prediction logic can then be skewed to favor the **NOT-TAKEN** prediction for short forward branches - to allow for a greater application of the improved recovery mechanism. Taken to the extreme, all short forward branches can be predicted **NOT-TAKEN**. These modification lead to a form of predication which is implemented without compiler support and requiring no predicate instructions.

6. Conclusions

Supporting the conditional execution of instructions is a technique that can have a significant impact on the performance of most high-performance architectures. It accomplishes this by providing a mechanism for removing control hazards (branches) which are well-known impediments to achieving greater amounts of instruction level parallelism (ILP). The short forward branches that are amenable to predication also have relatively poor prediction rates. Therefore, their removal can lead to an increase in the overall branch prediction accuracies for even the most sophisticated dynamic branch prediction strategies.

In order to measure these effects, we studied the relationship between predication and branch performance for the SPEC92 benchmark suite. Our study shows that an aggressive approach to predicating branches of distance less than or equal to 12 can reduce the count of instruction slots lost to executing branches by as much as 50%, and that a reduction in instruction slots of 30% to 50% holds true for each of the branch prediction schemes studied. This indicates that the improved branch prediction accuracies exhibited by some newer architectures do not offset their branch misprediction penalty. For example, the PowerPC 604 uses a highly

accurate branch prediction mechanism but pays a greater misprediction penalty (4 cycles) than the PA-RISC, which uses a simpler less accurate branch-backward scheme with a much lower misprediction penalty (1 cycle). Both of these processors receive almost identical benefits from predication.

The instruction slots lost due to unfilled instruction slots in multi-issue architectures can far exceed the instruction slots lost due to branch misprediction. However, aggressive predication also increases average basic block size by up to 45%, providing the code scheduler with more instruction to fill the execution pipeline and significantly reducing pipeline stalls.

In addition to the aggressive predication scheme, we examined two other more restrictive approaches. Results indicate that current predication schemes that allow for expression boosting but not full instruction set predication are far less able to reduce branch misprediction penalty and increase basic block size than the more aggressive approaches. While these restricted schemes can be useful in certain applications (e.g. tuned OS routines), they provide only a limited benefit in improving more general applications.

We feel that adding architectural support for the conditional execution of instructions is going to continue to grow in importance for the following reasons:

1. Branch misprediction penalties will continue rising as pipeline depths and issue widths increase.
2. The predicate cost is in terms of instructions, and therefore not dependent on issue width. As issue widths increase, the performance penalty for branch prediction becomes greater, where the predication cost remains constant.
3. Branches transformed by predication have worse than average prediction accuracy.
4. Predication allows for a significant increase in basic block size.

Current trends in Instruction Set Architecure (ISA) design include a restricted form of predicate operation. Predicate capabilities can be found in the following architectures: Intel P6, DEC Alpha, SUN Sparc, IBM/Motorola PowerPC, and HP PA-RISC. In order to maximize the benefit that predication allows, other architectural modifications will be required. These include the ability to perform speculative loads and to delay exeception on speculative instructions.

7. References

[AlAv93] D. Alpert and D. Avnon, ‘‘Architecture of the Pentium Microprocessor’’, *IEEE Micro*(June 1993), pp. 11-21.

[AADM93] T. Asprey, G. S. Averill, E. DeLano, R. Mason, B. Weiner and J. Yetter, ‘‘Performance Features of the PA7100 Microprocessor’’, *IEEE Micro*(June 1993), pp. 22-35.

[CMCW91] P. P. Chang, S. A. Mahlke, W. Y. Chen, N. J. Warter and W. Hwu, ‘‘IMPACT: An Architectural Framework for Multiple-Instruction-Issue Processors’’, *Proceedings of the Eighteenth Annual International Symposium on Computer Architecture*, Toronto, Canada (May 27-30, 1991), pp. 266-275.

[DeHB89] J. C. Dehnert, P. Y. T. Hsu and J. P. Bratt, ‘‘Overlapped Loop Support for the Cydra 5’’, *Proceedings of the 17th Annual Symposium on Computer Architecture*(May 1989), pp. 26-38.

[FiFr92] J. A. Fisher and S. M. Freudenberger, ‘‘Predicting Conditional Branch Directions from Previous Runs of a Program’’, *Proceedings of the Fifth International Conference on Architectural Support for Programming Languages and Operating Systems*, Boston, MA (October 12-15, 1992), pp. 85-95.

[HePa90] J. Hennessy and D. Patterson, *Computer Architecture: A Quantitative Approach*, Morgan Kaufman, San Mateo, California, (1990).

[HePa96] J. Hennessy and D. Patterson, *Computer Architecture: A Quantitative Approach (second edition)*, Morgan Kaufman, San Francisco, California, (1996).

[LeS84] J. K. L. Lee and A. J. Smith, ‘‘Branch Prediction Strategies and Branch Target Buffer Design’’, *Computer*, vol. 17, no. 1 (January 1984), pp. 6-22.

[MHBG94] S. A. Mahlke, R. E. Hank, R. A. Bringmann, J. C. Gyllenhaal, D. M. Gallagher and W. W. Hwu, ‘‘Characterizing the Impact of Predicated Execution on Branch Prediction’’, *Proceedings of the 27th Annual International Symposium on Microarchitecture*, San Jose, Ca. (November 30 - December 2, 1994), pp. 217-227.

[McLe93] E. McLellan, ‘‘The Alpha AXP Architecture and 21064 Processor’’, *IEEE Micro*(June 1993), pp. 35-47.

[PnSo94] D. N. Pnevmatikatos and G. S. Sohi, ‘‘Guarded Execution and Branch Prediction in Dynamic ILP Processors’’, *Proceedings of the 21st Annual International Symposium on Computer Architecture*, Chicago, Illinois (April 18-21, 1994), pp. 120-129.

[RYY89] B. R. Rau, D. W. L. Yen, W. Yen and R. A. Towle, ‘‘The Cydra 5 Departmental Supercomputer’’, *IEEE Computer*(Jan 1989), pp. 12-35.

[Russ78] R. M. Russell, ‘‘The CRAY-1 Computer System’’, *Communications of the ACM*, vol. 21, no. 1 (January 1978), pp. 63-72.

[Site92] R. L. Sites, ‘‘Alpha Architecture Reference manual’’, *Digital Press*(1992).

[Smit81] J. E. Smith, ‘‘A Study of Branch Prediction Strategies’’, *Proceedings of the Eighth Annual International Symposium on Computer Architecture*, Minneapolis, Minnesota (May 1981), pp. 135-148.

[SrEu94] A. Srivastava and A. Eustace, ‘‘ATOM: A System for Building Customized Program Analysis Tools’’, *Proceedings of the ACM SIGPLAN Notices 1994 Conference on Programming Languages and Implementations*(June 1994), pp. 196-205.

[TFRP95] G. Tyson, M. Farrens, K. Rich and A. Pleszkun, ‘‘Reducing the Branch Penalty of Mispredicted Short Forward Branches’’, Computer Science Department Technical Report CSE-95-7, University of

California - Davis, Davis, Ca 95616 (August 1995).

[YeP91] T. Yeh and Y. Patt, ‘‘Two-Level Adaptive Training Branch Prediction’’, *Proceedings of the 24th Annual International Symposium on Microarchitecture*, Albuquerque, New Mexico (November 18-20, 1991), pp. 51-61.

[YeP93] T. Yeh and Y. Patt, ‘‘A Comparison of Dynamic Branch Predictors that use Two Levels of Branch History’’, *Proceedings of the Twentieth Annual International Symposium on Computer Architecture*, San Diego, CA (May 16-19, 1993), pp. 257-266.

[94] ‘‘PowerPC 601 RISC Microprocessor Users’s Manual Addendum for 604’’, *Motorola / IBM Microelectronics*(1993, 1994).