Evaluating the Effects of Predicated Execution on Branch Prediction

Gary Tyson

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

Matthew Farrens

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

Abstract

High performance architectures have always had to deal with the performance-limiting impact of branch operations. Microprocessor designs are going to have to deal with this problem as well, as they move towards deeper pipelines and support for multiple instruction issue. Branch prediction schemes are often used to alleviate the negative impact of branch operations by allowing the speculative execution of instructions after an unresolved branch. Another technique is to eliminate branch instructions altogether. Predication can remove forward branch instructions by translating the instructions following the branch into *predicate* form.

This paper analyzes a variety of existing predication models for eliminating branch operations, and the effect that this elimination has on the branch prediction schemes in existing processors, including single issue architectures with simple prediction mechanisms, to the newer multi-issue designs with correspondingly more sophisticated branch predictors. The effect on branch prediction accuracy, branch penalty and basic block size is studied.

This work was supported by National Science Foundation Grants CCR-8706722 and CCR-90-11535.

Keywords: Predication, Branch Prediction, HP-RISC, Alpha, ATOM, Pentuim, PowerPC

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 critical. Most of the problem can be eliminated if conditional branches can either be correctly predicted or removed entirely. This paper will focus on two techniques to achieve this goal, *Branch Prediction* and *Predication*.

Branch prediction eliminates much of the branch delay by predicting the direction the branch will take before the actual direction is known, and continuing instruction fetch along that instruction pathway. This has been an active area of research for many years, and continues to be to this day [Smit81] [LeS84] [FiFr92] [YeP93]. There are two main approaches to branch prediction - 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. (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 much more recent area of research [CMCW91, DeHB89] although vector machines like the CRAY [Russ78] have long supported a type of predicated execution with their vector masks. This is a very promising area of research because, in addition to removing the branch itself from instruction stream, it provides the additional benefit of improving scheduling capability.

Branch Prediction and Predication can be used together in a complimentary fashion to minimize performance impacts of branches. For example, the number of 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 will examine the effect of augmenting branch prediction schemes found in a number of existing processors with the ability to do predicated execution. The focus will be on how predication affects the accuracy and branch penalty of the branch prediction schemes and the increase in basic block size. Different predication models will also be examined to determine their effectiveness in removing branches. Emphasis will be placed on whether the newer architectural designs, with their more accurate predictors, will show the same degree of improvement from removing short forward branches. Section 2 presents the different branch prediction techniques we studied, which is followed by 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 existing commercially available products, and two others included for the sake of completeness. Each approach has an associated branch misprediction penalty, indicating the number of cycles lost when a branch is incorrectly predicted.

The simplest of schemes predicts that all branches will be taken - thus the processor will always attempt to fetch instructions from the target of the branch.

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 forward branches are predicted **not** to be taken and all backward branches are predicted to be taken. This scheme performs better than branch always when applications contain forward branches that are not taken more often than they are taken.

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 used in the Alpha operates the same way as in the HP-RISC. The one-bit branch history table approach used in the Alpha 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.

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] uses 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** from that address; 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).

The final branch prediction strategy modeled is the two level adaptive strategy developed by Yeh and Patt [YeP91]. This strategy requires considerably more hardware resources than the other methods, but provides greater branch prediction performance than any of the other methods.

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 contains 512 13-bit entries (which limits the branch history pattern table to 8192 entries) and each branch history pattern table entry is a 2 bit history counter.

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. They 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. This is achieved by moving the statement in question up to before the branch, modifying it to write to a free register instead of a program variable, and then using one of the special predicate operations to conditionally update the active program variables. This is the approach used in the Alpha and HP-RISC processors.

In the unrestricted predication model, *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. Another way is to introduce a special instruction which controls the conditional execution of following (non-predicated) instructions. An example of this approach is seen in the *guarded execution model* proposed by Pnevmatikatos and Sohi [PnSo94], which includes special instructions whose execution specify whether following instructions should or should not be executed.

This section will present 4 different existing predication models, those used in the Alpha processor, the HP-RISC, guarded execution model and the Cydra 5.

The Alpha processor supports the restricted model of predication via a *conditional move* instruction. A standard move instruction only requires 2 operands (source and destination), leaving one field free to specify the conditional value in the conditional move. If the condition is satisfied then the register movement is allowed, otherwise a state change is prevented. This approach toward partial predication has increased in popularity amoung new implementations; the Intel P6 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 moved up to before a conditional branch, and is modified to write to a free register instead of a live program variable. A conditional move is then used in place of the original branch instruction to transfer the temporary value into the live register. If the condition is satisfied, the original destination of the expression will contain the result; if the condition is not satisfied, then the conditional move operation is not performed, and the active state of the application being executed is unchanged.

There are several restrictions on this form of predication. First, the compiler must ensure that no exceptions will be generated by boosting the expression code (e.g. division by zero must be excluded). Other instructions that cannot be predicated include some memory access instructions and flow control operations. Second, 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.

The HP Precision Architecture uses *instruction nullification* to provide a less restricted form of predication. In this architecture, control flow (branch) and arithmetic instructions can specify whether the *following* instruction 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.

Pnevmatikatos and Sohi 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 guarded branch instruction can be handled or what effect guarding will have on fetch latency.

The Cydra 5 system 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 (writeback) 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.

Table 1 shows how each approach would transform the following segment of code.

```
if (R5 > R6) { R7++; R5 = R5 * 10 - R6; } R4++;
```

Table 1: Predicated Code Examples

Alpha	HP
Guard	Full

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 only look at total branch distance, not directed distance. In order to examine the relationship between predication and branch prediction accuracy, it was necessary to begin by focusing on the characteristics of short forward branches (backward branches cannot be -directly- transformed by predication) and to calculate the prediction accuracy of the different schemes before predication.

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 [SrWa94] toolkit was used to generate and help analyze the data gathered for this study.
Branch instructions were instrumented to simulate the branch prediction schemes outlined in section 2. Predicate transformation was then implemented on the instrumented code to account for the removal of some branches.

As mentioned earlier, the Alpha compiler 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 shows the benchmarks programs and 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	*	3554909199	167524380	30274280
doduc	doducin.in	1149864381	84622229	13463476
ear	args.ref	17005800990	896667195	481185420
fpppp	natoms	4333190502	115692876	6508987
hydro2d	hydro2d.in	5682547494	347131563	9876849
mdljdp2	input.file	6856424748	320407455	71051
mdljsp2	input.file	2898940578	352481636	1024674
nasa7	*	6128388226	165573273	23201959
ora	params	6036097727	365260354	87096763
spice2g6	greycode.in	16148172367	1933779718	93549064
su2cor	su2cor.in	4776761988	178137872	30247847
swm256	swm256.in	11037397686	182031528	407016
tomcatv	N/A	899655110	30183243	27571
wave5	N/A	3554909199	167524380	30274280
compress	in	92628682	12379188	502775
espresso	bca.in	424397814	74371674	3603641
espresso	cps.in	513006475	83519772	4051876
espresso	ti.in	568262371	89791260	5083215
espresso	tial.in	983531458	167375187	12731193
gcc	cexp.i	23535507	3098517	506777
gcc	jump.i	143737833	6901206	1063385
gcc	stmt.i	51359229	19426084	2947663
xlisp	li-input.lsp	6856424748	867585743	316747153
eqntott	int_pri_3.eqn	1810542679	199198053	9669793
sc	loada1	1450169424	277556127	29920923
sc	loada2	1634276007	328405132	46912615
sc	loada3	412097081	91802679	6375876

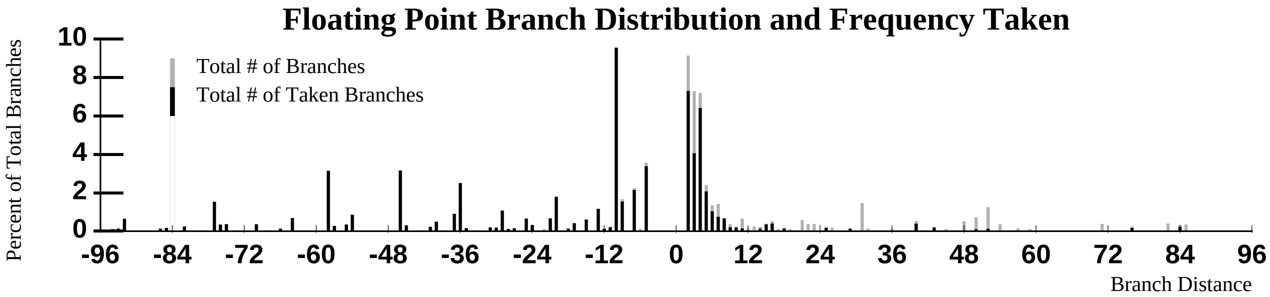
4.2.

Branch Characteristics of SPEC Benchmarks

Examining the branch characteristics of these programs in more detail shows how predication may help performance.

Figure 1 shows the total number of branches taken by the distance (in instructions) between the branch instruction and the target for the integer and floating point benchmarks.

These figures include both conditional and unconditional branches,



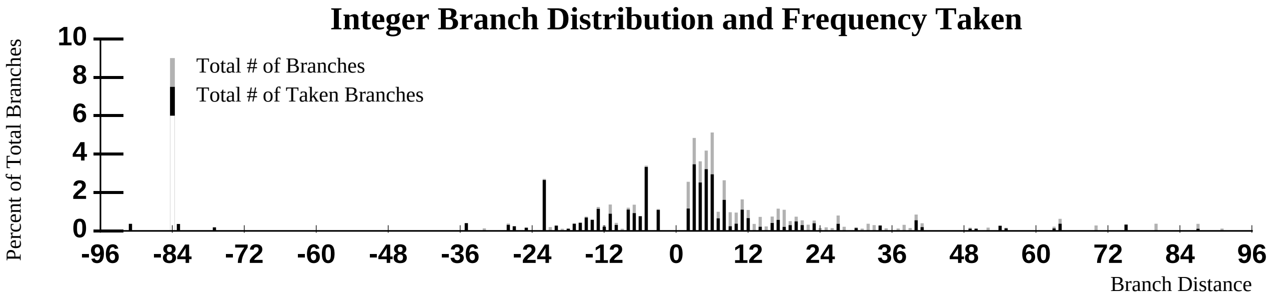


Figure 1: Branch Distribution

but exclude subroutine calls.

Notice that in both integer and floating point applications there is a high percentage of short forward branches. These 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.

Of the forward branches, those between distance 1 and 12 show the greatest frequency of execution.

Figure 2 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.

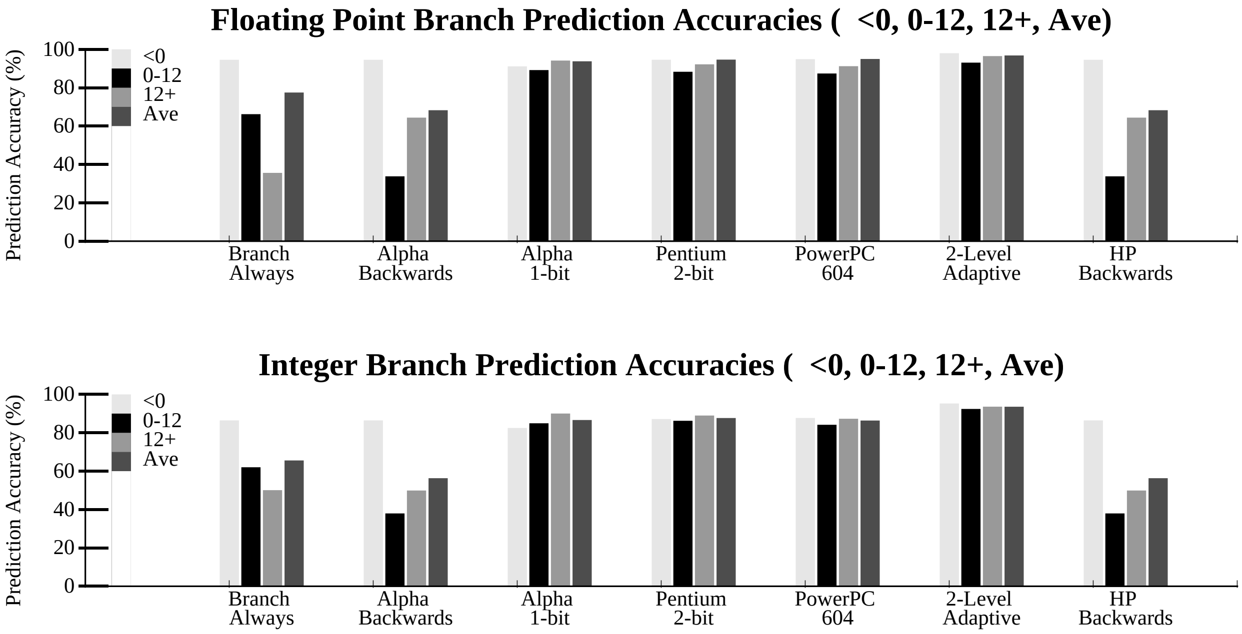


Figure 2: Branch Accuracies Before Predication

This figure shows that the prediction schemes are all able to predict backward branches accurately.

This is because backward branches are almost always taken (figure 1), which is what the static schemes assume and the table based schemes quickly determine.

In addition, for most prediction schemes the accuracy is lowest for branch distances in the range 0-12.

This is because these branches have the highest rate of changing 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.

This 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 would appear that removing these branches

may lead to an overall improvement in branch prediction accuracy.

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 shows the different branch prediction schemes and the number of cycles lost each time a branch is incorrectly predicted.

In addition the number of instruction slots lost is provided. 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.

4.3.

Predication Models

In order to measure the effect of predication on branch prediction,

Table 2. Branch Misprediction Penalty

Branch Prediction Scheme	**Penalty**	
	Cycles	Instr
Branch Always	1	1
HP-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

two translation schemes are 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 is referred to as *aggressive* because it 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 division instructions). This is modeled to analyze the effectiveness of some current predication mechanisms (e.g. Alpha).

4.4. Predicating Backward Branches

It is also possible to predicate some backward branch instructions; this may seem impossible at first glance, but in certain circumstances it is both feasible and useful.

loop example

So we augment the predication model to account for these branches as well...

4.5. Characteristics of Predication Approaches

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. We refer to this cost as the *Predication Cost*, and is calculated as the number of times a branch is taken multiplied by the branch distance. For example, if a branch that jumps over 5 instructions is transformed via predication, then the 5 instructions that were previously skipped will now have to be executed. In the event that the branch would have been taken, these 5 instructions do not need to be executed but are issued due to predication. Clearly longer branch distances incur much larger predication costs.

Table 3 shows the number of branches in each application and how many of those branches were removed by the predication schemes. The *Total* column shows the number of branch instructions (both conditional and unconditional) that are in the application. The *100%* column shows how many of those branches could be predicated using the aggressive (*A*) and Restricted (*R*) strategies. Three other columns show how many of the forward branches must be predicated to account for 80%, 95% and 99% of all branch executions. The *A* field shows the number of branches required and the *R* field show how many of those could be predicated using a restricted predication scheme. 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 achieve a substantial reduction in the branch penalty.

Aggressive predication is capable of removing approximately 30% of the total branches in the program. *Restricted* predication is capable of removing only about 5% of the branches. The inability to predicate loads and stores account for about half of the restricted branches. The remaining restrictions are 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.

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.

Table 3: Removal of Branches by Predication

Bench-Mark	Total Branch	80% A	80% R	95% A	95% R	99% A	99% R	100% A	100% 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

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 high misprediction penalty because, even though they may lose fewer cycles, each cycle is capable of doing more work.

Straight branch prediction accuracies also do not include the total number of branches that the figure is based on. For instance, if half of the branches are removed from the execution 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

In this study, the effect of removing short forward branches of distances 2, 4, 6, 8, 10 and 12 instructions was examined. Removing these branch instructions from the instruction stream affects the prediction schemes in two ways:

(1) Predication removes branches that have less than average prediction accuracy.

(2) For those schemes that perform table driven dynamic prediction, the reduction in the number of branches encountered can ease contention for branch table access.

Figures 3 and 4 show the the effects of predication (by distance) on branch misprediction rates. The figures show the change in misprediction rate relative to branch accuracy before predication. So, a level of 80% indicates that the prediction accuracy has improved by reducing the misprediction rate 20%. Misprediction rates are presented instead of accuracy rates to highlight the improvement in accuracy. The accuracy of branches in the range 0-12 is not displayed, because the number of branches remaining in that range after predication decreases to the point where meaningful representation is not possible.

This figure shows that several schemes experience reductions of up to 30% in the misprediction rate. The fact that many of the schemes do not experience significant accuracy changes also indicates that many current table driven schemes contain enough state information space and do not suffer from resource (table space) contention. The majority of the reduction in branch penality comes from the removal of a significant number of branches int the 0-12 instruction range.

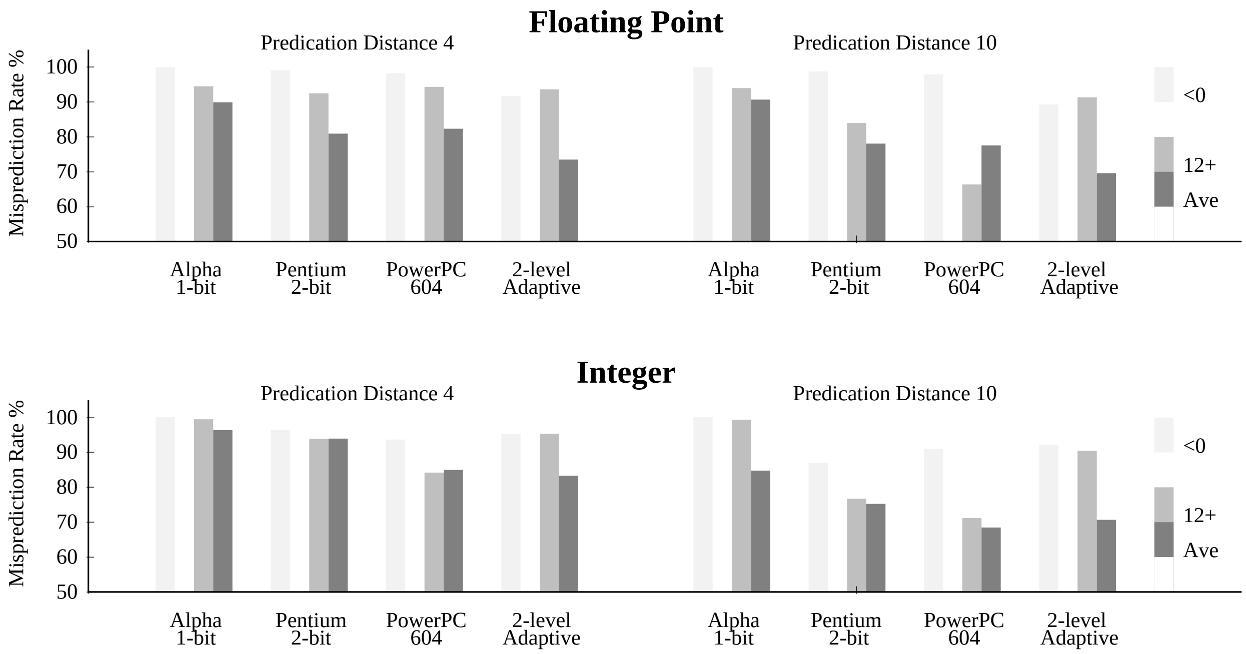


Figure 3. Change in Branch Misprediction Rate for Aggressive Predication

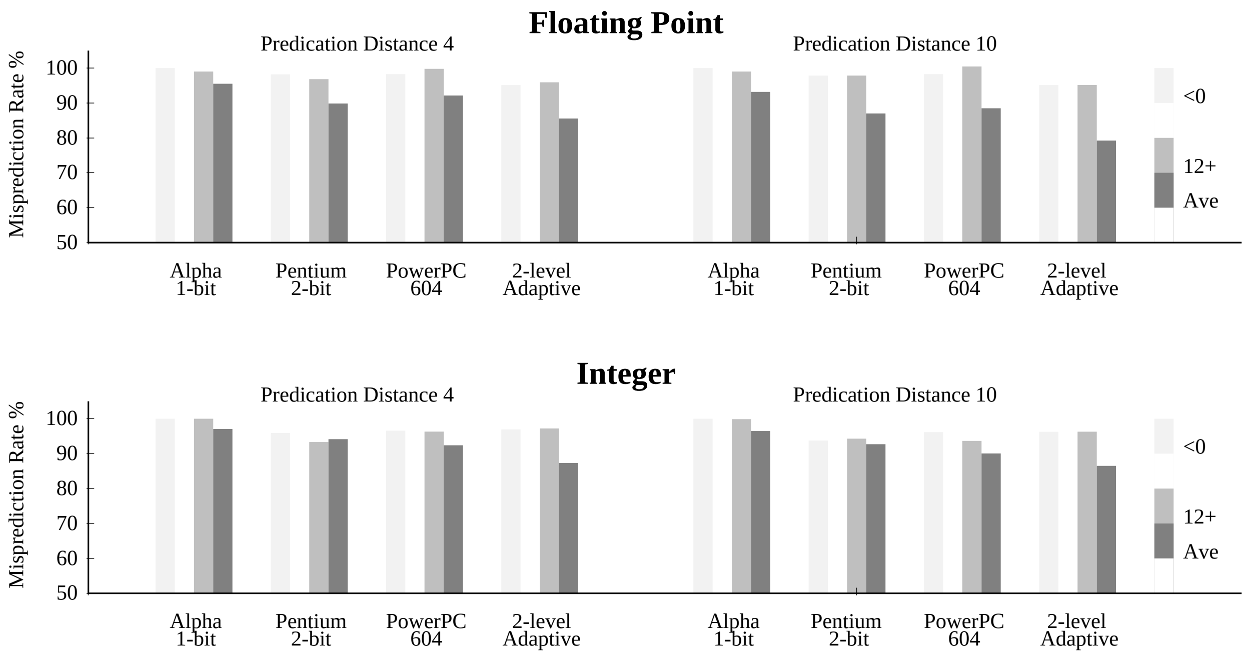


Figure 4. Change in Branch Misprediction Rate for Restricted Predication

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 cost more cycles than a simple scheme on a processor with a shorter pipeline. Measuring the relationship between the branch penalty and predication requires some definitions. We made the following:

branch penalty =cost of executing a branch (1)
**total number of branches executed*
+misprediction penalty
**number of mispredicted branches*

predication cost =Branch distance of removed Branch
**branches taken taken pre-removal*

The branch penalty equation is composed of two fairly straightforward components- each branch must be issued (using an issue slot), and each time a branch is predicted incorrectly, some number of cycles are lost to squashing instructions in progress and redirecting the fetch logic. This *misprediction penalty* varies from scheme to scheme (see table 2). (Note that the misprediction penalty is in *cycles*, and not instructions. This will clearly impact a multiple-issue machine more than a pipelined machine.)

The predication cost equation is a bit 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. The first consideration is whether an additional instruction is required to specify the predicate value (if so, then this is analogous to the branch issue required for each branch execution). Some predication mechanisms do not require the insertion of additional instructions, utilizing additional source fields in the instruction format to convey the predicate value instead. A second 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. Once predication removes the branch instruction, then the N instructions will no longer be bypassed, but will have to be executed and this cost will have to be included.

Figure 5 shows the total branch penalty calculated for each predication distance for each predication scheme. 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 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.

Branch penalty accounts for only a portion of the effect of predication; the cost of predication should be factored in to determine the overall effect on performance. Figure 6 shows what happens when this number is included. There are several things to note about this figure. For example, as you would expect, the greater the predication distance the less effective predication is in reducing the overall number of instructions issued. This is due to the direct relation the predication cost has to the distance of the branch (the branch penalty is independent of the branch distance). Each of the prediction schemes follows the same pattern, a drop in the number of instructions issued as very short branches are predicated, followed by a gradual rise in instruction issues as predication cost rises for greater branch distances.

Two facts of note about this figure are the performance of the Alpha Backward scheme and relative performance of the newer multi-issue architectures to the single-issue architectures using simple prediction schemes but having smaller misprediction penalties.

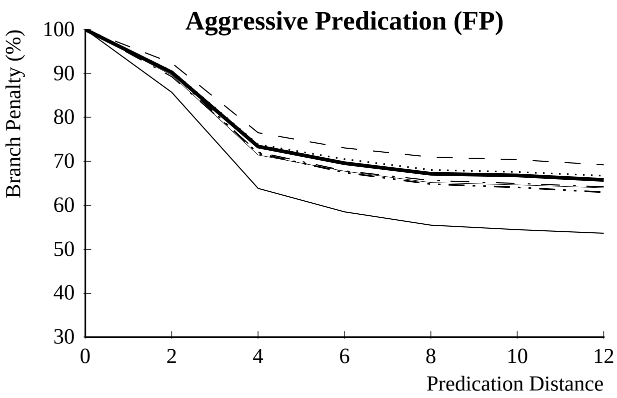
First, the Alpha utilizing a branch backward prediction scheme is an unusual case and worth further examination. In this scheme forward branches are predicted not taken, so a misprediction penalty is incurred each time the branch *is* taken. If this misprediction penalty is greater than the predication penalty generated by transforming the branch, it is a win to perform the transformation.

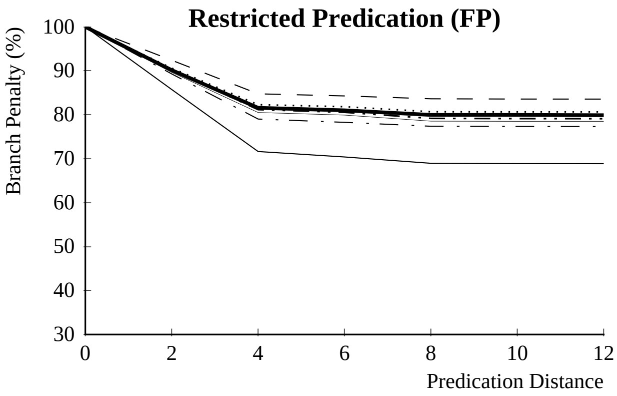
Since the predicate penalty is smaller than the misprediction penalty for all branches under 8 instructions (4 cycle misprediction penalty * 2 issues), a substantial improvement can be made by predicating **ALL** branches up to distance 9 on an Alpha with this configuration.

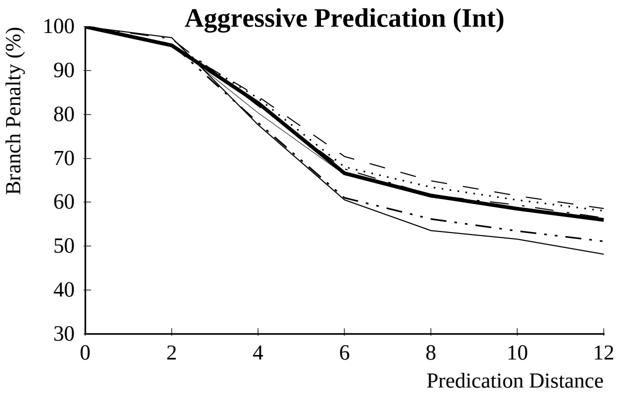
The second point demonstrated in figure 5 is that the newer architectures benefit from predication more than old designs even though the newer branch prediction schemes are better. Intuitively, it is clear that after scheduling these multi-issue architecture would receive a greater benefit from predication, but 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. This is excluding the much greater benefit that the removal of 30% of the branches will have on static and dynamic scheduling.

The *Restricted* scheme also shows improvement in total instructions issued, but it is more conservative in approach and the corresponding limit in effectiveness is proportional. Very few branches of distance greater than 4 are predicated.

The trend is towards more and more aggressive predication. The newer architectures can tolerate the greater predication cost caused by increasing the branch distance.


Aggressive Predication (FP)
Branch Penalty (%)
100
90
80
70
60
50
40
30
0
2
4
6
8
10
12
Predication Distance


Restricted Predication (FP)
Branch Penalty (%)
100
90
80
70
60
50
40
30
0
2
4
6
8
10
12
Predication Distance


Aggressive Predication (Int)
Branch Penalty (%)
100
90
80
70
60
50
40
30
0
2
4
6
8
10
12
Predication Distance

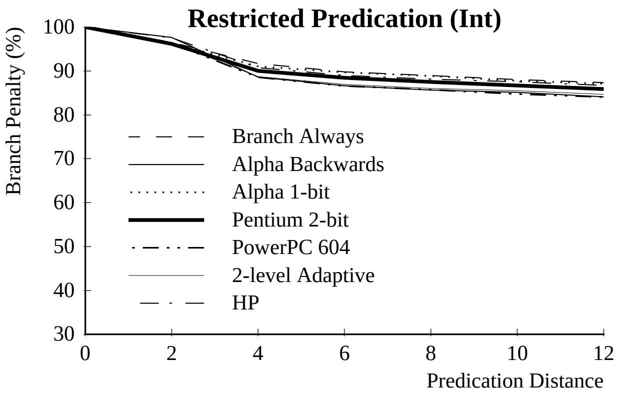
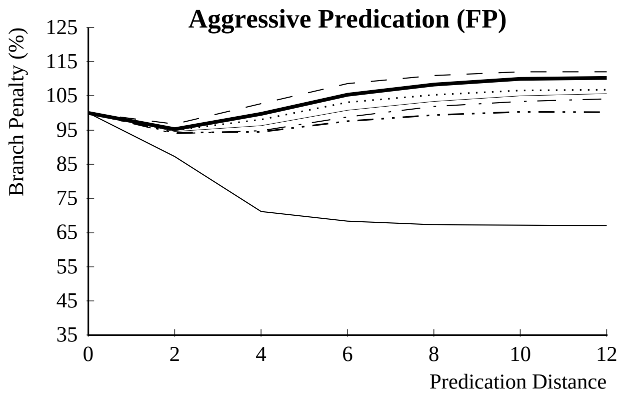
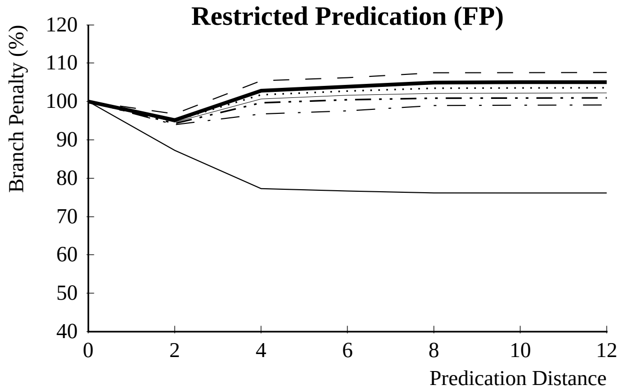

Restricted Predication (Int)
Branch Penalty (%)
100
90
80
70
60
50
40
30
0
2
4
6
8
10
12
Predication Distance
Branch Always
Alpha Backwards
Alpha 1-bit
Pentium 2-bit
PowerPC 604
2-level Adaptive
HP

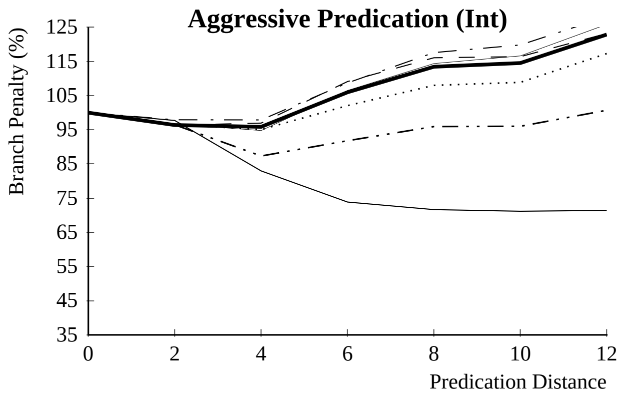
Figure 5. Percent of Branch Penalty per Predication Dist



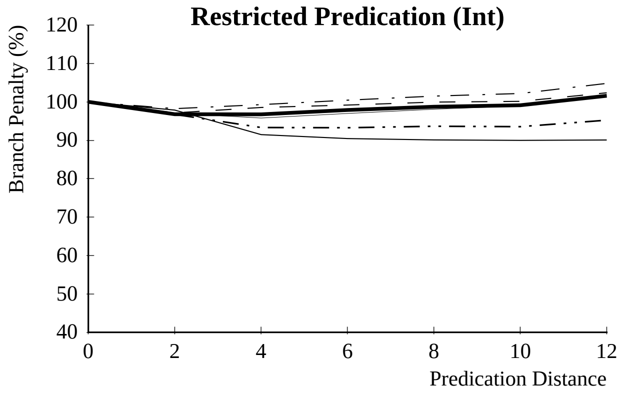
Aggressive Predication (FP)
Branch Penalty (%)
125
115
105
95
85
75
65
55
45
35
0
2
4
6
8
10
12
Predication Distance



Restricted Predication (FP)
Branch Penalty (%)
120
110
100
90
80
70
60
50
40
0
2
4
6
8
10
12
Predication Distance



Aggressive Predication (Int)
Branch Penalty (%)
125
115
105
95
85
75
65
55
45
35
0
2
4
6
8
10
12
Predication Distance



Restricted Predication (Int)
Branch Penalty (%)
120
110
100
90
80
70
60
50
40
0
2
4
6
8
10
12
Predication Distance

Figure 6. Total Cost of Branches

5.3. Scheduling Issues

intro

diffs VLIW and SS (HW and SW scheduling)

reference urbana stuff here

results shown for there stuff

lead into our analysis of BB

5.3.1. 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 slots. Unfortunately, many compiler transformation are still precluded.

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

Figure 7 shows the results of the *Aggressive* transformation on the integer SPEC benchmarks. This figure shows that as the predication distance increases, the basic block size increases as well, and that a branch distance of 12 provides a 40% increase in basic block size. This is almost half of the total increase achievable if **all** forward branches were removed. 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

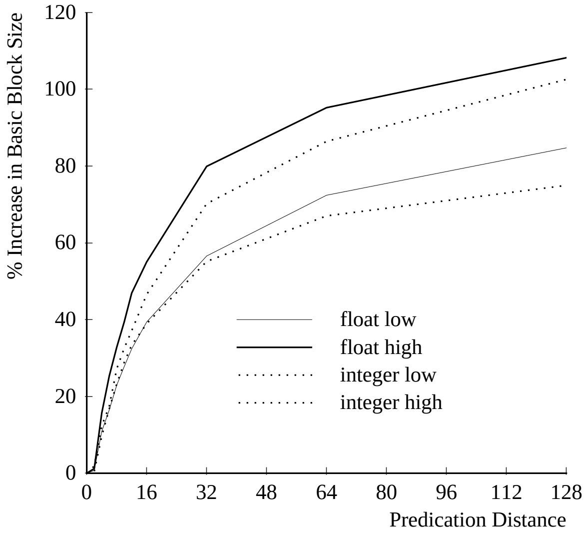


Figure 7. Affects of Predication on Average Block Size

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 not overshoot 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 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. Alternate Schemes

Modified branch predictors (skewed)

Modified misprediction logic

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 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 some of the above 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%. A reduction in instruction slots of 30% to 50% holds true for each of the branch prediction schemes studied. This indicates that 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 HP-RISC, which uses a simple 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 show dramatically less ability to reduce branch misprediction penalty and increase basic block size than the more aggressive approach. While these 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 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 support this conclusion. Predicate capabilities found in the following architectures: P6, Alpha, Sparc, ...

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).

[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.

[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 21th Annual Symposium on Computer Architecture*, Chicago, Illinois (April 18-21, 1994), pp. 120-129.

[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.

[SrWa94] A. Srivastava and D. W. Wall, ‘‘Atom: A system for building custonized program analysis tools’’, *Proceedings of the ACM SIGPLAN Notices 1994 Conference on Programming Languages and Implementations*(June 1994), pp. 196-205.

[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).