Contract MDA-903-83-C-0335

Reducing the Cost of Branches

Scott McFarling and John Hennessy
Computer Systems Laboratory
Stanford University

Abstract

Pipelining is the major organizational technique that computers use to reach higher single-processor performance. A fundamental disadvantage of pipelining is the loss incurred due to branches that require stalling or flushing the pipeline. Both hardware solutions and architectural changes have been proposed to overcome these problems. This paper examines a range of schemes for reducing branch cost focusing on both static (compile-time) and dynamic (hardware-assisted) prediction of branches. These schemes are investigated from quantitative performance and implementation viewpoints.[1]

Introduction

Branches constitute anywhere from 15-30% of the instructions executed on typical machines. On higher performance pipelined machines, such instructions consume a larger fraction of time because they cause pipeline stalls and pipeline flushes. On machines with powerful instruction sets, the frequency of branches tends to be very high. RISC-style machines emphasize very high instruction execution rates, and although branch frequencies may be lower, the branch penalties must also be kept lower. In a RISC machine, branches are the most significant barrier to achieving single-cycle execution (i.e. initiation of an instruction on every machine cycle).

There are numerous approaches to dealing with branches. This paper examines a set of compile-time and run-time branching schemes, evaluates their effectiveness, and makes some observations about their implementation cost. Procedure call and return jumps are excluded since different tradeoffs exist and special optimizations may be appropriate. We start by defining a machine model that we will use for evaluating the various branch schemes.

[1]The MIPS-X project has been supported by the Defense Advanced Research Projects Agency under contract # MDA 903-83-C-0335.

A Model of Branch Cost

Pipeline structure significantly affects the cost of a branch. We will examine some alternative pipeline structures, determine why they impact performance, and give a method of assigning branch cost. We will conclude with a pipeline structure that will be the basis for evaluating the different branch schemes.

We start with a five-stage pipeline that we will later change to reduce the overall branch penalty. For simplicity, we assume a register-register machine; this simplifies the pipeline and also makes it easier to quantify execution time. The evaluation in this paper can be extended to machines with more complex pipelines using the data contained in this paper.

Our initial pipeline has the following structure:

IF	ID	ADDR	ALU	WB
	IF	ID	ADDR	ALU
		IF	ID	ADDR

The function of these stages are as follows:

IF	Fetch instruction
ID	Decode instruction.
ADDR	Fetch registers and compute the effective address.
ALU	Access data memory or do an ALU operation.
WB	Write into register file either loaded data or ALU result.

An instruction fetch is assumed to take a single cycle. For a high performance machine, an instruction cache may be required to provide instructions at this rate. We also assume that only one instruction can be fetched per cycle. Specifically, we ignore schemes that require multiple instruction cache ports or multiword busses that allow instruction decode to get ahead of execution.

Let's first consider a branch instruction that includes a register-register compare, i.e., a compare and branch instruction. The results of the compare are available at the end of the ALU phase, and the two possible new PCs (the branch target and the sequential branch successor) can easily be computed before then. However, the pipeline requires that the new PC must be sent to the instruction cache three stages earlier. Hence, a three cycle penalty is required whenever the branch is

taken. Furthermore, to avoid a penalty on the untaken case, we must be careful not to commit any state during the the first three stages of the pipeline. If this is not possible, we must be able to back-out any state changes. Additionally, the hardware must be capable of disabling the three instructions in the pipe when the branch is taken.

Alternatively, consider a condition-code based machine. In such a machine, the branch can be done as soon as the address is evaluated, since the branch condition is based on simple masking of the condition code. This removes one delay cycle, leaving a branch delay of two. However, the setting of the condition code must precede the branch instruction. If we assume that the condition code must be stable by the beginning of the ADDR cycle of the branch, then the immediately preceding instruction can set the condition code. For some other pipeline structures, this may not be possible and delays will be needed between the condition code setting and its use by the branch.

In an earlier paper[1], condition codes were shown to rarely set for free i.e. by ALU operations needed for another purpose. Thus, the branch cost for a condition code machine with this pipeline structure is two instructions (the condition code setting instruction plus the actual branch) and two delay slots. Since delay slots can be filled more often than the condition code can be set for free, the condition code approach is more expensive.

To compare a variety of condition evaluation mechanisms effectively, we propose that *branch cost* be based on the average cycle count between the start of condition evaluation and the start of the corresponding stage of the instruction executed after the branch. This average cycle count includes the cycles for the register-register comparison, plus the branch itself, plus the delay slots that are idle or do not advance the state of the computation. For the above pipeline, the cost of a branch for both the condition-code and compare-and-branch models, assuming that all the delay cycles are wasted, is four, although the cycle breakdown is different in the two cases.

The branch cost reduction schemes we will present all aim at using the branch delay slots to achieve less costly branches. Since less than 100% of the delay slots can be effectively used, we would first like to try to reduce the number of delay slots. The length of the branch delay is determined by the position of certain operations in the pipeline, namely evaluating the branch condition and computing the destination PCs. For a complex instruction set with multiple addressing modes for data items and branch destinations, it would be difficult to improve on this pipeline. However, for a simple instruction encoding, the base address computation that is done during ADDR could be accomplished during ID. If we attempt to compute the branch-taken destination during ID, we will need to know the position and size of the branch displacement without much decode time (perhaps a half-cycle). Moving this address calculation will also cost an additional adder, because the main ALU and effective address adder are still required by the preceding two instructions.

Just moving the branch destination calculation does not reduce the branch delay, because either the condition code setting instruction or the condition evaluation in the compare-and-branch instruction force a total branch delay of three cycles. If we make some further assumptions about our instruction encoding, we can reduce this delay to two cycles with either condition evaluation mechanism. To do this, we need to move the ALU cycle of the pipeline up to the position of the ADDR cycle; the resulting pipeline is identical to the one used in MIPS-X[2], a high performance successor of the MIPS architecture.

IF	ID	ALU	MEM	WB
	IF	ID	ALU	MEM
		IF	ID	ALU

This pipeline requires a register fetch during ID, implying that the instruction format is simple enough to access the registers without decoding the instruction type. While this is possible for a load/store machine, it is very difficult for a machine with complex datatypes and addressing modes. The decode time of the more complex instruction format can be pipelined away only if one disregards branches. For simplicity and consistency, we will assume that we are dealing with this streamlined pipeline structure and its worst case branch cost of three cycles.

Branch Schemes

A longer, more aggressive pipeline requires a more aggressive branch strategy. On the first generation Berkeley and Stanford RISC machines, the pipeline required a branch delay of one instruction and had a branch cost of 1.3 to 2.5. With a 20% branch frequency, those machines saw a loss of 6% to 30% of the machine compared to a machine with a single cycle branch instruction. The effect of a deeper pipeline with its longer branch delays is considerable: a two cycle branch penalty (which equals a three cycle branch) implies 40% of the machine is wasted in branch delays, and a three cycle penalty wastes 60%! These performance losses could easily wipe out the advantage of deeper pipelining. We would like to keep the branch cost comparable to that incurred on a less deep pipeline.

In the remainder of this paper, we will describe several techniques to reduce branch cost and evaluate their performance on our example pipeline. To make a quantitative comparison feasible, we have measured performance on a set of benchmarks. These benchmarks are:

Bigfm	Fiduccia-Mattheyses graph partitioning algorithm; 500 lines.
Dnf	converts logic equations to disjunctive normal form; 1500 lines.
Hopt	a simple global optimizer for Pascal; 2200 lines.

All these programs are written in Pascal, and while they are not huge, we believe they are fairly typical of non-numeric code. For numeric applications the predictability of branches should be better. Each program was compiled with an optimizing compiler[3]. Optimization tends to increase the relative frequency of branches, because branch instructions are rarely eliminated, while loads, stores, and arithmetic instructions often are. Also, removing redundant instructions prevents them from being scheduled into the branch delay slots, artificially improving the performance of the scheduling schemes we will explore later. Hence, using optimized code as the basis for a study of branches is important.

Beyond efforts to shorten the branch delay, schemes to alleviate branch cost focus on the use of the branch delay slots. To use those slots effectively, we must predict the outcome of the branch; we will examine techniques that attempt branch prediction both in hardware and in software. We first consider three hardware-oriented schemes, assume the branch is not taken, dynamic branch prediction, and a branch target buffer. These schemes require increasing amounts of hardware. We then consider more software-oriented schemes, increasing both the hardware support requirements and the accuracy of software prediction.

Predict Branch Not Taken

The first scheme we consider is what many architectures do: continue fetching instructions ignoring the branch. The viability of this approach depends intimately on the depth of the pipeline and the arrangement of activities in the pipeline. Most machines with a pipeline depth of three can easily use this mechanism with a single-cycle delay, since the only activity that will occur before the branch outcome is known is to prefetch the sequential successor. The 68020 and the VAX 11/780 use this scheme. However, even when the branch delay is only one, complications can arise. For example, fetching the instruction following the branch may cause a page fault or a protection violation. To prevent this from occurring, the VAX 11/780 will halt its prefetching if it detects such an event, until the processor is sure that the instruction should be attempted.

As the branch delay increases, these complications get more severe. In many complex architectures, machine state may be changed early in the computation. For example, auto-increment/decrement addressing modes will cause problems if the register update is allowed to execute before the branch is determined. Our example pipeline can start the two instructions after the branch because no state is committed in the first two stages of the pipeline.

Execution proceeds without penalty if the branch is not taken because the sequentially following instructions have been fetched and initiated. If the branch is taken, the sequential instructions must be *squashed* and the target fetched. Squashing refers to disabling instructions in the pipeline such that the instructions do not change the program state. Even when no state has been changed, squashing can be difficult. Several actual processors suffer an extra penalty to squash instructions in the pipeline. This overhead must be eliminated if the predict-not-taken approach is to have much value.

Assuming squashing problems can be solved, performance of predict-not-taken is limited by the percentage of branches taken. We assume that there is no penalty for not taken branches (the branch cost is one), and that taken branches cost a full 3 cycles (one branch instruction and two aborted delay slots). Taken branches are more common: as Table 1 shows, 63% of branches are taken on our example benchmarks; other studies have shown numbers that are slightly higher. For example, Clark[4] reported that 67% of all branches are taken. Table 1 also shows performance for a predict taken scheme, which would always fetch the target as soon as a branch is recognized. This means that taken branches cost two cycles and not-taken branches cost three cycles. Overall, predict-taken is slightly slower than predict-not-taken, and it is also more complex to implement.

Benchmark	Branches Taken	Predict Not Taken	Predict Taken
Bigfm	.67	2.34	2.34
Dnf	.54	2.08	2.46
Hopt	.67	2.34	2.33
Average	.63	2.26	2.38

Table 1: Assume Taken or Not-Taken Performance

Dynamic Prediction

The extra cost of branches relative to ALU instructions can be broken down into 2 parts: condition evaluation and target fetch. The target fetch must be delayed until the branch direction is known. If the direction could be predicted, the branch penalty of a taken branch could be reduced by one cycle. Dynamic prediction attempts to predict the direction of a branch from its past behavior. Lee and Smith[5] evaluated several hardware prediction methods. We will examine the technique they found most accurate. In this strategy, the lower-order bits of the branch address are used to access a table, yielding two prediction bits. These bits specify the prediction to be used for all branches whose addresses are identical modulo the size of the table. Since no tags are needed, the table can be quite small.

The table entry for a branch is updated whenever that branch is executed. The update is done according to the finite state machine shown in Figure 1. If the last two mapped branches have gone in the same direction, the FSM predicts this direction. If a branch goes the opposite way, the FSM will continue predicting its usual direction on the next mapped branch. Thus, if a branch goes an unusual direction one time, the prediction will only be wrong one time. With a single prediction bit, the prediction would be wrong twice. Note, that 2-bit prediction correctly handles loop branches that are almost always taken but have occasional, single changes in their behavior.

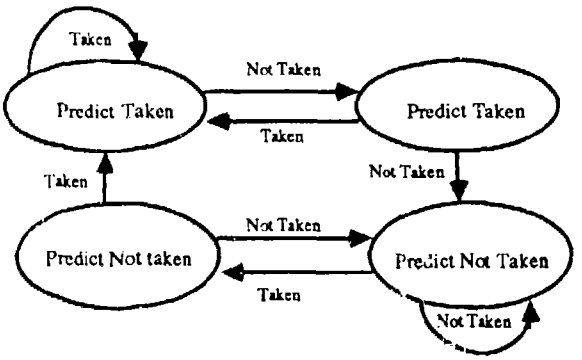

Figure 1: Branch Prediction State Diagram

For the purposes of comparison we will assume a prediction table of 128 entries. Larger prediction tables do not generate significant improvements. We will also assume that squashing is available, so that the next sequential instruction can be started in case the branch is not taken. This assumption leaves us with the branch cost matrix shown below:

	Actual Branch	
Prediction	Taken	Not Taken
Taken	2	3
Not Taken	3	1

Table 2 shows the prediction accuracy of this scheme on our benchmarks together with overall performance.

Benchmark	Prediction Accuracy	Cycles/ Branch
Bigfm	.85	1.92
Dnf	.81	1.97
Hopt	.85	1.96
Average	.83	1.95

Table 2: Dynamic Branch Performance

Branch Target Buffer

Hardware prediction successfully reduces branch cost because it accurately predicts branch direction. However, there is still almost a full cycle wasted per branch, largely because a correctly predicted, taken branch wastes a cycle waiting for the target to be fetched. To drive branch cost down further, a branch target buffer(BTB) can be used to get the target instruction early. A BTB acts as a cache of branch targets: given the address of a branch, it returns the actual target instruction. If the BTB is small and fast, it can be accessed during the ID phase of the branch instruction (before the branch is actually decoded). With our pipeline and its delay of two cycles, we need to retrieve two instructions. Rather than store the second instruction in the BTB, it can be fetched as in the previous section if a separate prediction table is maintained. Because the BTB entries are wider than prediction table entries, this will save hardware. Also, we can miss in the BTB, but correctly predict with the prediction table, and still lose only one cycle.

There are several variations on the branch target buffer. For example, we could cache the addresses of the predicted instruction and access the BTB during IF; this would be especially attractive for a machine with multiword instructions. Another variation, which we will not examine, is to keep the branch target successor instruction in the BTB, and access the BTB during IF. This could yield zero cost unconditional branches, since we would know from the BTB whether the instruction is a branch, and the next instruction to be executed, both at the end of IF. The BTB could just return the next instruction, eliminating the unconditional branch execution completely. This last variation has some implementation challenges that arise from the need to keep more instructions in the cache and from the need to update the BTB on a miss.

In the chart below we give the branch costs for the BTB scheme:

	Buffer Hit		Buffer Miss	
Prediction	Taken	Not Taken	Taken	Not Taken
Taken	1	3	2	3
Not Taken	3	1	3	1

If we can both predict the branch correctly and find it in the buffer, the branch costs only a single cycle. However, it takes a fairly large buffer to obtain a good hit rate. For example, the MU-5[5, 6] had an eight entry buffer and was able to start instructions only 40-60% of the time, *including* the instructions following branches that were predicted not taken. A BTB is basically a small, selective, instruction cache. A BTB has an advantage because it only needs to store target instructions; however, the spatial locality of a BTB is low, and the amount of tag storage is larger (only one-word blocks make sense). In our simulations, we noticed that a direct mapped BTB and an instruction cache of the same size had about the same hit ratio.

Note also that a collision in a BTB is more damaging than a collision in a branch prediction table, since a BTB miss yields nothing of value, while prediction bits still have some relevance. In fact, since most branches are taken, a prediction table entry has a greater than 50% chance of predicting correctly, even if the entry were for a different branch.

Simulation results for the BTB scheme are given in Tables 3 and 4. We show a variety of sizes for the BTB, all assuming a prediction table of 128 entries. The BTB is assumed to be direct mapped; higher associativity could lead to better hit rates. By comparison, the hit rates we found are still more optimistic than the hit rates Lee and Smith obtained with higher associativity, probably because their test programs were larger. For large BTB sizes, the average branch cost is quite good. However, the buffer size is significant, must be close to the processor, and may be complex to implement.

Benchmark	16 Entries	64 Entries	256 Entries
Bigfm	.54	.83	.94
Dnf	.49	.80	.92
Hopt	.42	.85	.94
Average	.47	.83	.93

Table 3: BTB Hit Rates

Benchmark	16 Entries	64 Entries	256 Entries
Bigfm	1.56	1.36	1.28
Dnf	1.66	1.46	1.40
Hopt	1.65	1.32	1.26
Average	1.62	1.38	1.31

Table 4: Branch Performance with BTB

Delayed Branch

From a hardware point of view, the simplest way to optimize branches is the delayed branch. The machine continues executing instructions after the branch until the condition is determined. The compiler tries to schedule useful instructions into the slots after the branch from one of the three locations shown in Figure 2. Instructions from before (a) can not be used if the branch depends on them. Instructions from after the branch (b) or at the target (c) must be safe to execute whether the branch is taken or not. No live state may be destroyed and no illegal operations may be done, such as loading from a null address. Also, (b) and (c) only reduce branch cost if the scheduled instructions would have been executed anyway, i.e. the branch went the favorable direction. Thus, where there is a choice, strategy (a) should be used rather than (b) or (c).

Delayed branches have been used successfully on several RISC machines, including the IBM 801, RISC II[7], and MIPS[8,9]. All three machines had a one-cycle branch delay. For these machines, branch costs are in the range of 1.3 to 1.5. The lower number was achieved on the MIPS design through the use of compare-and-branch, which requires only one branch-related instruction. However, the MIPS number must be taken with some reservation since each instruction took two clock cycles.

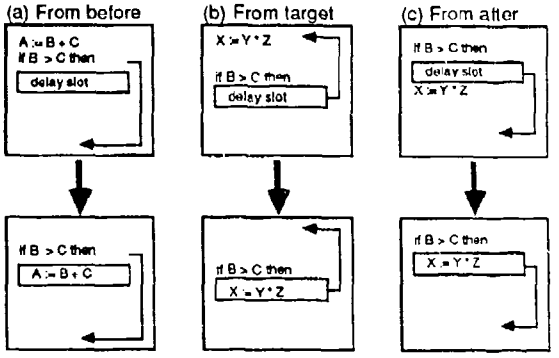

Figure 2: Scheduling a Delayed Branch

The MIPS 1.3 cycle branch cost means that the single delay slot could only be used about 70% of the time. Unfortunately, the second slot of our example pipeline is much more difficult to fill. If both slots could be used 70% of the time, we would predict an average branch cost of 1.6 cycles. Table 5 shows a branch penalty of over 2.2 cycles. The second slot could not be used over 75% of the time.

Benchmark	Cycles/Branch
Bigfm	2.41
Dnf	2.27
Hopt	1.96
Average	2.21

Table 5: Delayed Branch Performance

Fast Compare

The delayed branch scheme incurs a delay of two with our pipeline structure because of the need to wait until the branch condition is known. The condition requires a full cycle for a full ALU operation. Katevenis[10] observed that a general ALU operation is not needed for most compares; these *fast* compares include tests for equality, inequality, and any relation with zero. Other compares can often be converted to the fast type. For example, the C loop:

```
for (i=0;i<10;i++) a[i] = b[i];
```

can be converted to:

```
for (i=0;i!=10;i++) a[i] = b[i];
```

Those compares that can not be simplified can be split: one instruction can do the compare and put the result in a general register which can then be compared against zero by the fast compare-and-branch instruction.

The fast compare can be done during the ID phase of the instruction as soon as the register operands are

fetched. Thus, the branch delay for a fast compare is reduced to one cycle, which can be used as a delayed branch. Since only one instruction is needed, we expect better filling than for the 2-cycle delayed branch in the previous section. These assumptions imply the following cost matrix:

	Slot Filled	Slot Not Filled
No Compare Needed	1	2
Compare Needed	2	3

Table 6 shows the breakdown of comparisons used in our benchmarks; compares are classified as full compares requiring a separate instruction, compares against zero, or equal/not equal compares. It is interesting to note the variance in the distribution: Dnf and Bigfm make heavy use of compare against zero, while Hopt relies heavily on equal/not equal compares. In all cases, the compiler has attempted to use fast compares whenever possible. Table 7 shows the overall branch cost including the number of full compares and the number of wasted delay cycles.

Benchmark	Equal/ Not Equal	Compare Against 0	Full Compare
Bigfm	.22	.69	.09
Dnf	.07	.84	.09
Hopt	.79	.14	.07
Average	.36	.56	.08

Table 6: Compare Types Needed

Benchmark	Wasted Slots	Compares Needed	Cycles/ Branch
Bigfm	.54	.09	1.63
Dnf	.43	.09	1.52
Hopt	.36	.07	1.44
Average	.44	.08	1.53

Table 7: Fast Branch Performance

Cycle count performance of the fast compare scheme is encouraging. However, the compare must be done in the same cycle as the register fetch, which may present problems on machines with complex instruction encodings. Also the timing of the simple compare is a concern, because it must complete in time to change the instruction address going out in the next cycle. This could easily end up as the critical path controlling instruction cache access and, hence, cycle time.

Delayed Branches with Squashing

The major limitation of delayed branches is the difficulty of filling the branch delay slots with safe instructions. An alternative to the fast compare or simple delayed branch schemes is to use delayed branches with controlled squashing of the instructions in the delay slot. There are four varieties of delayed branch with squashing that are generated by varying two characteristics:

Squashing direction: the branch indicates whether it is likely to be taken and, hence, when the delay slots should be squashed (i.e. when the prediction is wrong). Of course, setting this bit depends on predicting the branch at compile-time. Without extra analysis, the compiler would always guess that branches were taken. The value of this bit comes when the compiler has accurate information on taken versus not-taken frequencies. We will examine the issue of accurate prediction later; for now, we assume that squashing is never done when branches are taken.

Squashing control: the branch instruction contains a bit specifying whether instructions immediately after the branch may be squashed or not. This bit allows the use of delayed branches if both slots can be filled with instructions from before the branch, i.e. where a delayed branch would be faster. MIPS-X supports branches with a squash control bit that squashes only when they are not taken and the control bit is on. The value of the control bit is rather limited, since it is usually on. However, for comparison purposes we will assume it is available.

Delayed branch with squashing provides the best of the delayed branch, predict-taken, and predict-not-taken schemes. If for a particular branch, a delayed branch is best, it can be used by not setting the squash control bit. Otherwise, a delayed branch with squashing has the advantage of predict-taken, in that the instructions most likely to follow the branch are started. However, like predict-not-taken the started instructions can begin immediately because they are positioned sequentially after the branch. Table 8 shows the resulting performance. By combining the best features of the other three schemes, nearly one half cycle per branch is saved over the best of the three alone. Cycle count performance is not quite as good as for fast-compare, but cycle time impacts may well overshadow the difference.

Benchmark	Probability Taken	Cycles/ Branch
Bigfm	.67	1.64
Dnf	.54	1.98
Hopt	.67	1.70
Average	.63	1.77

Table 8: Squashed Branch Performance

Profiled Branches

The performance of squashing branches is limited by the assumption that all branches are usually taken. Performance could be increased further if the compiler knew which branches are usually not taken. Such information could be supplied by an execution profiler. Table 9 compares the prediction accuracy of a profile with that of dynamic hardware prediction. The profile predicts slightly better than a 128-entry predictor and doesn't require any hardware. Most branches usually go one way or the other throughout the execution of a program. Additionally, software prediction does not suffer any loss of accuracy due to collisions in the

branch prediction hardware.

Benchmark	Profile Prediction	Hardware Prediction
Bigfm	.83	.85
Dnf	.82	.81
Hopt	.89	.85
Average	.85	.83

Table 9: Branch Prediction Accuracy

Profile information can be used to optimize branches in several ways. Delayed branch performance can be improved by filling slots from the predicted successor basic block. For higher performance, we need to handle the case where the likely successor is not safe to always execute. Ideally, the instruction set would contain branches with the squashing direction bit described in the previous section. Alternatively, the compiler can attempt to increase the fraction of taken branches by modifying the control flow graph. We will examine two alternatives:

- **If-then-else** restructuring,
- branch insertion

If-then-else control structures can be optimized by rearranging the flow graph. If the **then** clause is more likely to be executed, the branch corresponding to the **if** will usually not be taken. The **if**-branch can be changed to usually taken by swapping the **then** and **else** clauses and inverting the branch condition. For a predict-not-taken machine, this technique could be used to *decrease* the number of branches taken. However, swapping the **then** and **else** clauses solves only part of the problem: on our Pascal benchmarks, only 44% of **if**'s have **else** clauses. Plus, the technique does nothing for loop branches.

The cost of *any* usually not-taken branch can be improved without requiring a squash direction bit. We simply insert an additional branch before the predicted not-taken branch. The new branch has the inverse of the original condition and thus is usually taken. The original branch is made unconditional and the slots for both branches can be filled with instructions from their respective targets (requiring only squashing when the branch is not taken). For example:

```
        bgt r1,r2,label1
```

becomes:

```
        ble r1,r2,lnew
        bra label1
lnew:
```

If the squash control bit is on and if all squash slots can be filled, branch costs are as follows:

Prediction	Actual Branch: Taken	Actual Branch: Not Taken
Taken	1	3
Not Taken	4	1

If profile prediction is correct, the branch costs only a single cycle. If the profile predicts not taken and the branch is actually taken, there is an *extra* cycle delay for the added branch. Thus, the profile prediction must be fairly accurate or branch insertion could increase branch cost.

The results of profile optimization are given in Table 10. Performance of the fast compare scheme with profile information is provided as well. Here, the profile is used to fill the delay slots with instructions more likely to be useful.

Benchmark	Fast Branch	Squashing Branch
Bigfm	1.59	1.54
Dnf	1.35	1.46
Hopt	1.38	1.30
Average	1.44	1.43

Table 10: Branch Performance with Profile

As Table 10 shows, profile prediction significantly reduces branch cost. However, the data in the table assumes that the program will be run on only one set of inputs. To be useful, the prediction must be accurate for other inputs as well. To measure how severe this effect is, we changed the inputs to our set of benchmarks and measured how much slower they were relative to the same program recompiled using a profile derived from the new input. Over 98% of the savings due to profiling was preserved. This high value is quite encouraging. For other programs the number may not be this high, but we still expect most of the gain to be maintained.

Conclusion

We have presented several schemes for improving branch cost. The performance results are summarized in Tables 11 and 12. Table 11 shows performance in terms of cycles per branch, while Table 12 shows overall machine speed relative to a hypothetical machine with single cycle branches. The overall performance is based on the instruction set of MIPS-X. Since MIPS-X is a load/store machine, branch frequencies are somewhat lower than on machines with fuller instruction sets. The impact of branch cycle count will be even higher on a machine where branches are more frequent.

Scheme	Performance Average
Delayed Branch	2.21
Predict Not Taken	2.26
Branch Target Buffer(256)	1.31
Fast Compare	1.53
Profiled Fast Compare	1.44
Squashing Branch	1.77
Profiled Squashing Branch	1.43

Table 11: Branch Performance Summary: Cycles per Branch