Fast & Accurate Instruction Fetch and Branch Prediction

Brad Calder and Dirk Grunwald*
Department of Computer Science,
Campus Box 430, University of Colorado,
Boulder, CO 80309-0430
(Email:{calder,grunwald}@cs.colorado.edu)

Abstract

Accurate branch prediction is critical to performance; mispredicted branches mean that ten's of cycles may be wasted in superscalar architectures. Architectures combining very effective branch prediction mechanisms coupled with modified branch target buffers (BTB's) have been proposed for wide-issue processors. These mechanisms require considerable processor resources. Concurrently, the larger address space of 64-bit architectures introduce new obstacles and opportunities. A larger address space means branch target buffers become more expensive. In this paper, we show how a combination of less expensive mechanisms can achieve better performance than BTB's. This combination relies on a number of design choices described in the paper. We used trace-driven simulation to show that our proposed design, which uses fewer resources, offers better performance than previously proposed alternatives for most programs, and indicate how to further improve this design.

1 Introduction

During a related study on the architectural features used by object-oriented languages, such as C++, we examined several branch architectures to see how many optimizations a compiler would have to perform to efficiently execute object-oriented programs [4, 5]. We assumed future processors would be pipelined superscalar architectures and would need to drastically reduce the occurence of pipeline stalls for efficient execution.

Conventional processor architectures, particularly superscalar designs, are extremely sensitive to control flow changes. A simplified processor pipeline can be divided into **fetch**, **decode**, **execution**, **memory access** and **write** stages. Changes in control flow, be they conditional or unconditional branches, direct or indirect function calls or returns, are not detected until those instructions are decoded. To keep the pipeline fully utilized, processors typically fetch the address following the most recent address. If the decoded instruction is a break in control flow, the previously fetched instruction can not be used, and a new instruction must be fetched, introducing a "pipeline bubble" or unused pipeline step. This is called an *instruction misfetch penalty*.

The final destination for conditional branches, indirect function calls and returns are typically not available until the **memory access** stage of the pipeline is completed. At this point the branch has been completely evaluated in the **execution** stage. The program counter is updated after the **memory access** stage. As with instruction fetch, the processor may elect to fetch and decode instructions on the assumption that the eventual branch target can be accurately predicted. If the processor mispredicts the branch destination, those instructions fetched from the incorrect instruction stream must be discarded, leading to several "pipeline bubbles." In practice, pipeline bubbles due to mispredicted breaks in control flow degrade a programs' performance more than the misfetch penalty. For example, the combined branch mispredict penalty for both pipelines of the Digital AXP 21064 processor is 10 cycles. By comparison, the AXP 21064 would lose only two instruction issues from instruction misfetches. As processors issue more instructions concurrently, these penalties increase, and the instruction fetch penalty becomes increasingly important. It is more likely that a branch will occur as more instructions are fetched per cycle, decreasing the likelyhood that the "fall through" instruction will be executed. Historically, processor design has focused on correctly predicting conditional control flow changes, because it is simple to implement and results in considerable savings. There are a number of mechanisms to ameliorate the effect of uncertain control flow changes, including static and dynamic branch prediction, branch target buffers, delayed branches, prefetching both targets, early branch resolution, branch bypassing and prepare-to-branch mechanisms [11].

Likewise, there are a variety of mechanisms to reduce the instruction mispredict penalty, including delayed branches, where the instruction following a branch is either always executed or conditionally executed ("squashed") depending on the branch target or a condition code, and branch target buffers. A branch target buffer (BTB) is a cache storing the branch address and likely target address. When an instruction is fetched, the same address is offered to the BTB; if there's a match in the BTB, the next in-

* This paper appears in the 1994 Intl. Symp. on Computer Architecture, Chicago, Il. April, 1994

struction is fetched using the target address specified in the BTB. Originally, BTB's were used as a mechanism for branch prediction, effectively predicting the prior behavior of a branch – even small BTB's were found to be very effective [10, 15, 17].

More recently, there has been considerable interest in using BTB's to reduce instruction misfetch penalties; for example Yeh *et al* [21] propose using a very large BTB to improve prediction accuracy and reduce misfetch penalties. In fact, their BTB records a multitude of useful information to support wide-issue processors. Wide-issue processors fetch multiple instructions, roughly the size of a basic block. If a basic block address is in the BTB, then the basic block contains a break in control flow; Yeh & Patts' design includes additional information indicating whether the break is a conditional branch, unconditional jump, indirect jump or a return instruction. Each BTB entry also contains a per-basic block pattern history register, used to index into a 2-level branch history table [20, 22].

Architectures using BTB's can issue a large number of instructions per cycle because of accurate branch and fetch prediction. However, BTB's lead to a complex architecture. In this paper, we show how to achieve the same or better performance using simpler techniques. We do this by:

- Decoupling branch prediction from the branch target buffer. This allows us to accurately predict a conditional branch's direction even when a BTB miss occurs on the branch's address.
- Changing the BTB allocate policy. We measure the effect of not storing "fall through" branches in the BTB. Architectures usually have a 'default' rule for branches (e.g., backwards taken, forward not taken). By exploiting this default behavior, we can make more effective use of the BTB. Furthermore, various profile-guided code transformations can make this modification very effective.
- Dispensing with the BTB all together. We propose a branch architecture that selects the branch destination address from the instruction.

As always, there are caveats to our paper architecture; we discuss them at the end of the paper.

2 Background

In order to contrast our instruction fetch architecture to a proposed aggressively designed architecture, we describe the instruction fetch architecture proposed by Yeh *et al*, briefly describing their branch prediction mechanism and the structure of their branch target buffer (BTB). Their mechanism is a logical continuation of currently proposed and implemented designs.

There are two sources of pipeline stalls we want to remove. The first is the instruction misfetch penalty. This can be done a number of ways; e.g, by using branch delay slots [13] or branch target buffers [10, 11, 15, 17]. A BTB can eliminate misfetch stalls by storing the branch destination. For unconditional branches, indirect jumps or functions calls, this destination can be immediately fetched. For conditional branches, either the "fall-through" or the destination stored in the BTB is selected; obviously, some form of branch prediction is needed to select between the fall-through and taken address. To differentiate between actions for the different branch types, we need to be able to identify the branch type; thus, some BTB designs store the branch type in the BTB. For function calls (either direct or indirect), the previous function address is stored in the 'destination' field of the BTB. This can also be done for return instructions, but a return stack [8] is much more accurate. When using a return stack the BTB provides no useful information for returns, but it does indicate the instruction is a return instruction so the return stack can be used, avoiding the misfetch penalty.

The other component of most branch architectures is some mechanism to predict whether conditional branches are 'taken' or 'not taken' (i.e., the fall-through address is executed). Branch prediction techniques are classified as *static* or *dynamic*. Static branch prediction information does not change during the execution of a program, while dynamic prediction may change, reflecting the time-varying activity of the program. Static methods range from compile-time hueristics [1, 10, 13, 17] to profile-based methods [7, 13, 19]. In general, profile based prediction techniques outperform compile-time prediction techniques or techniques that use hueristics based on the direction of the branch target (forward or backward) or instruction opcode.

While static prediction mechanisms, particularly profile-based methods, accurately predict 80-90% of branches, modern computer architectures increasingly depend on mechanisms that estimate future control flow decisions to increase performance, requiring more accurate branch prediction mechanisms. Some architectures use *dynamic prediction*. BTBs and branch history tables, either alone or in combination, are two examples of dynamic prediction mechanisms.

A BTB can be used to predict conditional branches by storing a destination address and predicting that instruction is executed. The destination address can either be updated on each branch or two-bit saturating counters [10] can be used to improve prediction accuracy. By *coupling* the branch prediction information with the BTB, we avoid both misfetch and misprediction penalties; however, we can only do this for branches that have been entered in the BTB. Typically, a BTB contains from 32 to 512 entries with varying degrees of associativity. A BTB requires a lot of storage, because it stores the address of the branch *and* the address of the probable destination. Some BTB's also include additional storage to encode the branch type and prediction information.

In the absence of a BTB, conditional branches can be predicted using much simpler mechanisms. A *pattern history table* eliminates the site and target addresses from the table; hence the table only predicts the direction for conditional branches. These designs use the branch site address as an index into a table of *prediction bits*. Since different branch addresses can index into the same table entry, several conditional branches may share the same prediction information. For example, in a 4096 entry table, branches at addresses 0, 16384 and 32768 all map to the same entry in the table. When a conditional branch at these addresses is executed, the information for entry '0' is used to predict the

branch direction, even if that information was recorded for one of the other branches. The most common variants of this design are 1-bit techniques that indicate the direction of the most recent branch mapping to a given prediction bit, and 2-bit techniques that yield much better performance for programs with loops [10, 13, 17]. The advantage of the pattern history tables is that they keep track of very little information per conditional branch site and are very effective in practice.

More recently Pan *et al* [14] and Yeh and Patt [20, 22] have proposed *branch-correlation* or *two-level* branch prediction mechanisms. Although there are a number of variants, these mechanisms generally combine the history of several recent branches to predict the outcome of an incipient branch. The simplest example is the so-called *degenerate method* of Pan *et al*. When using a 4096 entry table, the processor maintains a 12-bit shift register (the pattern history register) that records the outcome of previous branches. If the previous 12 branches that executed were a sequence of three taken branches, six non-taken branches and three more taken branches (TTTNNNNNNTTT), the register might store the value 111000000111_2, or 3591. This is used as an index into the 4096-entry table, much as the program counter is used in the previous method. This provides contextual information about particular patterns of branches. Other methods combine the pattern register with other information. McFarling [12] `xor`'s the program counter with the history register, scattering the table references and slightly improving performance.

Yeh and Patt [22] propose a number of alternatives. We focus on their 'PAs' method, since they found it to be most effective. Each branch in the BTB has a unique history register. In the PAs(6,16) method, history registers are six-bits, holding the history of the previous six branch decisions *for that specific branch*; thus, for a 512-entry BTB, 3072 bits are used for the pattern history registers. When predicting the outcome of a particular branch, bits `<5:2>` of the program counter and bits `<5:0>` of the history register form a ten-bit index into the 1024-entry history table; the history table contains 2048 bits.

In this paper, we are primarily concerned with 2-level prediction methods; see [14, 20, 22] for details on their design. The problem with using only a 2-level prediction method is that one cannot avoid the fetch penalty associated with identifying what type of break has occurred and computing its target address. Also, the BTB can store the history of each branch, making branch prediction more accurate. This is why BTB's are useful for eliminating instruction misfetch penalties, and why some architectures combine both BTB's and these accurate prediction mechanisms.

3 A BTB-based instruction Fetch Architecture

Figure 1 is a schematic representation of the branch prediction and instruction fetch architecture suggested by Yeh and Patt [21]. The current instruction address is concurrently offered to the instruction cache (not shown), providing the actual instruction, and to the BTB. A 32-entry return address stack handles return instructions. There are three important types of branches: direct or indirect branches, conditional branches and function returns. Depending on the branch type and the prefetched branch prediction information (all stored in the BTB) the destination, fall-through or return stack address is selected as the next instruction fetch. For example, the first time a particular 'return' is entered into the BTB, the BTB entry records that the new entry contains a return instruction. When the return is encountered again, the BTB immediately indicates this is a 'return', and selects the next branch destination using the return stack. In this case, the 'destination' field of the BTB is not used, because the return stack is more accurate for procedure returns. Similar activity occurs for a direct or indirect branch, except the 'destination' field of the BTB is used to fetch the next instruction.

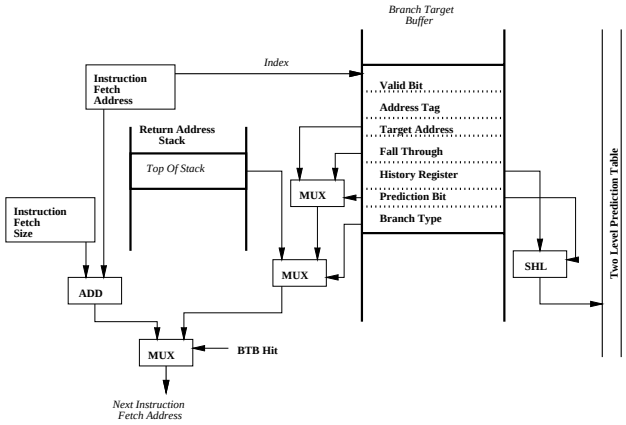


Figure 1: A Schematic Representation of the Branch Prediction Using Two-Level Prediction and Branch Target Buffers, As Proposed By Yeh and Patt

Again, conditional branches have similar actions; however, the 'prediction' field in the BTB is used to predict the likely outcome of conditional branch. Depending on the predicted outcome, the stored 'destination' (which is always the 'taken' address) or the fall-through address is used to fetch the next instruction. Then, the history table is updated; this can occur several cycles later with little penalty – see [21] for more details.

The critical path in this architecture is for conditional branches. The processor must offer the PC to the BTB, extract the destination and prediction fields and use this to select the appropriate destination address. Concurrently, the prediction table is indexed, and the resulting prediction bit (which will be used for the next branch) is stored in the BTB.

The BTB in all of our simulations has 512 entries, organized as a 128 by 4-way set-associative cache using LRU replacement. Each BTB entry corresponds to a single branch, and contains a 6-bit branch prediction history register specific to that branch. This branch history register is used to index the 1024 2-Bit pattern history table (PHT), using the PAs(6,16)[22] mechanism described at the end of §2.

4 Experimental Methodology

We will pose several questions concerning branch architectures and answer those questions using information from trace-based

Program	# Insn's Traced	# Breaks Traced	Conditional Branches				Percentage of Breaks during Tracing				
			Traced	In Program	%Fall	%Taken	**%CBr**	**%IJ**	**%Br**	**%Call**	**%Ret**
alvinn	5,240,969,586	476,254,227	430	1,622	2.23	97.77	98.30	0.02	0.40	0.64	0.64
compress	92,629,658	12,882,149	230	1,124	31.75	68.25	88.51	0.00	7.59	1.95	1.95
eqntott	1,810,540,418	208,877,319	466	1,536	9.70	90.30	93.47	1.70	1.90	0.70	2.24
espresso	513,008,174	87,798,840	1,737	4,568	38.10	61.90	93.25	0.20	1.88	2.29	2.39
gcc	143,737,915	22,960,184	7,640	16,294	40.58	59.42	78.85	2.86	5.75	6.04	6.49
li	1,355,059,387	239,416,514	556	2,428	52.70	47.30	63.94	2.24	7.74	12.92	13.16
sc	1,450,134,411	303,554,400	1,471	4,478	33.12	66.88	85.96	0.98	2.62	5.18	5.26
cfront	19,001,390	3,056,060	5,783	15,509	46.82	53.18	73.45	2.17	6.40	8.72	9.26
db++	86,457,511	15,178,598	421	1,639	43.14	56.86	54.43	15.04	2.03	6.77	21.73
idl	21,138,201	4,145,007	1,001	3,839	53.30	46.70	50.00	12.31	7.55	9.07	21.07
groff	41,522,284	6,687,063	2,511	7,434	45.83	54.17	66.12	4.80	7.80	8.77	12.51

Table 1: Measured attributes of the traced programs. Columns marked 'Traced' are measured during execution of the program.

simulation.

We instrumented the programs from the SPECint92 benchmark suite and object-oriented programs written in C++. Other studies have noted that FORTRAN programs have very predictable branches, and there is little one can do to improve that prediction; we simulated the SPECfp92 benchmarks and found that was true. We omit the results due to space. We used ATOM [18] to instrument the programs; due to the structure of ATOM, we did not need to record traces and could trace very long-running programs. The programs were compiled on a DEC 3000-400 using either the DEC C compiler or DEC C++ compiler. All programs were compiled with standard optimization (`-O`). We constructed several simulators to analyze the program. Typically the simulator was run once to collect information on call and branch targets, and a second time if we needed to use profile information from the prior run. For the SPECint92 programs, we used the largest input distributed with the SPECint92 suite.

The alternate programs include: `cfront`, version 3.0.1 of the AT&T C++ language preprocessor written in C++, `groff`, a version of the `ditroff` text formater written in C++, `idl`, a C++ parser for the CORBA interface description language, and `db++`, a version of the 'deltablue' constraint solution system written in C++. We selected these programs because we found that the SPECint92 suite did not typify the behavior seen in C++ programs [5], and our original goal was to understand the impact of branch architectures on C++ programs. For these alternate programs, we used sizable inputs we hoped would exercise a large part of the program.

Table 1 shows the basic statistics for the programs we instrumented. The first columns lists the number of instructions traced and the second column indicates the number of breaks in control flow that were simulated. The third column indicates the number of unique conditional branches in the program that we actually executed during the trace; the fourth column shows the total number of conditional branches in each program. The fifth and sixth columns show the percentage of conditional branches that are 'fall through' (not taken) or 'taken', respectively. The last five columns break down the number of breaks in control flow encountered during tracing into five classes: conditional branches (**CBr**), indirect jumps (**IJ**), unconditional branches (**Br**), procedure calls (**Call**) and procedure returns (**Ret**).

Note that the C++ programs execute fewer conditional branches than C programs. In part, this is caused by the increased number of procedure calls in the C++ programs. In the compiler we used, indirect jumps are used both to implement indirect function calls and some `switch` statements.

5 Improvements to BTB Architectures

Our goal is to understand the performance improvement of various branch architectures; this requires a metric to compare one architecture to another. In [2], Bray and Flynn state:

> *Past attention in BTB design focused on hit rate to describe the performance, but hit rate is not all that important. How often the instruction fetch unit predicts the correct address is the important performance issue. A branch can miss in the BTB and still be predicted correctly, since the default is to go inline. Because miss rate does not accurately show the performance of the BTB, we use* predict incorrectly *as a measure of performance.*

While this is true, we believe more accurate metrics are still needed. There are two forms of pipeline penalties we are concerned with: misfetching and misprediction. Each branch type can be misfetched; but only conditional branches, indirect function calls and returns can be mispredicted. The penalty for misfetching is less than the penalty for misprediction. We may be willing to misfetch more branches if it means we can reduce the number of mispredicted branches. Thus, we record the percentage of misfetched branches (%MfB) *and* the percentage of mispredicted branches (%MpB). It is often difficult to understand how these metrics influence processor performance. Yeh & Patt defined a formula for the *branch execution penalty*

$$\text{BEP} = \frac{\%\text{MfB} \times \text{misfetch penalty} + \%\text{MpB} \times \text{misprediction penalty}}{100},$$

	PAs(6,16)			GAg(11)			GAg(12)		
Program	%MfB	%MpB	BEP	%MfB	%MpB	BEP	%MfB	%MpB	BEP
alvinn	0.03	0.19	0.01	0.06	0.25	0.01	0.06	0.21	0.01
compress	0.00	10.13	0.51	0.00	10.36	0.52	0.00	9.86	0.49
eqntott	0.00	1.39	0.07	0.00	1.29	0.06	0.00	1.28	0.06
espresso	0.11	5.70	0.29	0.13	5.69	0.29	0.13	5.07	0.25
gcc	3.24	14.73	0.77	5.62	13.96	0.75	5.75	12.38	0.68
li	0.29	4.62	0.23	0.33	4.79	0.24	0.33	4.01	0.20
sc	0.16	3.14	0.16	0.25	3.37	0.17	0.26	3.09	0.16
cfront	6.40	16.22	0.87	10.71	14.06	0.81	11.03	11.49	0.68
db++	0.26	1.02	0.05	0.46	0.98	0.05	0.47	0.77	0.04
idl	0.89	2.09	0.11	1.16	1.86	0.10	1.16	1.61	0.09
groff	3.59	7.89	0.43	5.19	7.39	0.42	5.35	5.82	0.34

Table 2: Effects of Decoupling the Pattern History Table from the Branch Target Buffer

	PAs(6,16)			GAg(11)			GAg(12)		
Program	%MfB	%MpB	BEP	%MfB	%MpB	BEP	%MfB	%MpB	BEP
alvinn	0.00	0.24	0.01	0.00	0.25	0.01	0.00	0.21	0.01
compress	0.00	10.13	0.51	0.00	10.36	0.52	0.00	9.86	0.49
eqntott	0.00	1.40	0.07	0.00	1.29	0.06	0.00	1.28	0.06
espresso	0.08	7.55	0.38	0.10	5.69	0.29	0.10	5.07	0.25
gcc	2.24	13.43	0.69	3.81	13.93	0.73	3.90	12.35	0.66
li	0.03	4.59	0.23	0.04	4.79	0.24	0.04	4.01	0.20
sc	0.08	3.10	0.16	0.13	3.36	0.17	0.14	3.09	0.16
cfront	4.40	13.48	0.72	7.49	13.99	0.77	7.74	11.42	0.65
db++	0.07	1.06	0.05	0.12	0.98	0.05	0.12	0.77	0.04
idl	0.65	1.81	0.10	0.85	1.78	0.10	0.85	1.53	0.09
groff	2.29	6.59	0.35	3.45	7.06	0.39	3.59	5.49	0.31

Table 3: Effects of Only Storing 'Taken' Branches in the BTB

which reflects the average penalty suffered by a branch due to misfetch and misprediction. So, a BEP of 0.5 means that, on average, each branch takes an extra half cycle to execute; values close to zero are desirable. We use this metric to provide a more intuitive understanding of how the two penalties interact. However, this binds us to a specific misfetch and misprediction penalty – we have assumed a one cycle misfetch penalty and a five cycle misprediction penalty.

We simulated the BTB-based architecture as proposed by Yeh & Patt [21]. We choose their model because it has been described clearly and in depth, making it easier to duplicate their simulations. Despite that we are simulating a different base architecture and used different compilers than used in [21], our results for their architecture reflect the performance noted in [21].

5.1 Decoupled Prediction and Fall Throughs

One of the disadvantages of a coupled pattern history register, as used in the BTB-based architecture, is that a branch may not be in the BTB. The branch may suffer a misfetch penalty and the outcome of the branch must be predicted using less accurate static prediction methods. However, the information in the PHT could have been used to predict the branch with more accuracy, avoiding some branch mispredict penalties. If we used a single pattern history register, as originally proposed by Pan [14], we can use the PHT to predict the branch whether it is in the BTB or not. In a comparison of prediction methods [22],Yeh *et al* compared this method (which they termed the 'GAg' method) and other prediction methods. They found that storing prediction registers in the BTB gave a higher *prediction accuracy* [22]; however, they did not account for the differences between misfetch and misprediction.

In Table 2 we show more detailed metrics for the organization found to have the best prediction accuracy in [22] (PAs(6,16)) and the simpler method that can use the pattern history register even when the branch is not located in the BTB (GAg). In the PAs method, if a branch is not in the BTB, we use a static backward-taken/forward-not-taken prediction. We simulated the GAg method using the same history table size (GAg(11), 2048 entries) as the PAs method, and one with a larger table (GAg(12), 4096 entries). Although the sum of the misfetched and mispredicted branches is higher for the GAg methods, making them look worse, the GAg methods *misfetch* more often than they *mispredict* – and mispredicting is more expensive than simply misfetching. Thus, the branch execution penalty for our reference architecture is actually *smaller* for the GAg methods.

This is not true for all programs; for example, `compress` has slightly worse performance using the GAg methods. When `compress` is compiled on the Alpha, a handful of branches account for most of the executed branches – predicting these branches is extremely important for good performance.

5.2 Only Taken Allocate

We can further increase the effectiveness of the BTB by only storing 'taken' branches in the BTB. If a branch is not in the BTB and it is "not taken," it is not entered in the BTB. If the 'not taken' branch is already in the BTB, the prediction information is updated, but the LRU reference information is not – thus, 'not taken' branches will be displaced more frequently than 'taken' branches. Taken branches are always entered in the BTB. By not entering fall-through branches, we avoid displacing prediction information for other branches, and 'not taken' branches don't really benefit from the BTB, since they fetch the following instruction. If a branch is not found in the BTB, the architecture uses a static prediction mechanism (backwards taken, forward not taken) that is fairly accurate.[1]

Table 3 shows the performance of this method – this improves the PAs method more than the GAg method, in part because the PAs is more sensitive to capacity misses in the BTB; however, the GAg methods still have a lower overall BEP. Using PAs, all programs except `espresso` have improved performance. `espresso` has many branches and the static prediction for branches not found in the BTB is not very effective. We see less improvement for the GAg methods than the PAs method when we only store taken branches. However, note that `espresso` benefits from the improved branch prediction available in the GAg method. With this type of BTB organization, it becomes very useful to have programs with many 'not taken' branches. In a related paper [3], we show how programs can be restructured to accomplish this and the impact it has on branch architectures. We profile the program behavior, recording the most likely direction for each branch and then modify the program binary to favor 'not taken' branches. Pettis [16] and Bray [2] have performed similar transformations, but for different goals and with different outcomes.

Before leaving the BTB based architecture, a final observation regarding the C++ programs we traced is in order. Programs using the object-oriented paradigm, which are becoming increasingly common, appear to substitute indirect procedure calls and returns for conditional branches – programmers use subclasses to specialize behavior rather than conditionals. With a BTB, an indirect procedure call can be predicted about as accurately as a conditional branch, and a return-stack [8] predicts returns very accurately without using the BTB. Thus, programs such as `db++` and `idl`, which perform a tremendous number of function calls have the lowest BEP of our sample programs. We should note that although `cfront` is written in C++, it is structured more like a conventional C program internally, using many conditional expression, and has a larger BEP due to this.

[1] We could have only stored branches that violate the static prediction rule into the BTB – however, this introduces many more misfetchx penalties for programs with a large number of 'taken' branches, such as `alvinn` and `eqnttot`.

6 An Alternative Architecture

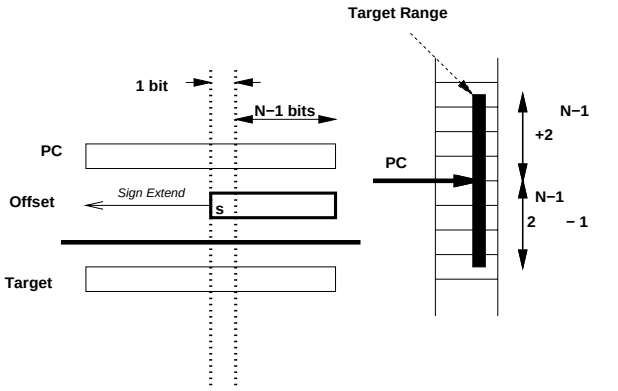
In the previous section, we used a realistic performance metric to show that it's better to remove some functionality from the BTB. This leaves only the branch tag and likely destination in the BTB – everything except the instruction type information can be removed. This means the BTB is only saving us from misfetch penalties and mispredictions due to indirect jumps. There are three reasons for using a BTB: (1) By virtue of an instructions address being in the BTB, we know the instruction is a branch; (2) Accurate prediction information can be associated with each BTB entry and (3) The BTB provides pre-computed destination and fall-through addresses for unconditional and conditional branches. The destination of return instructions can be predicted using a return stack. We have argued that item (2) is not really needed. If we can determine the instruction type (1) and the destination address *via* other means, we may be able to dispense with the BTB.

6.1 Computing the Branch Target

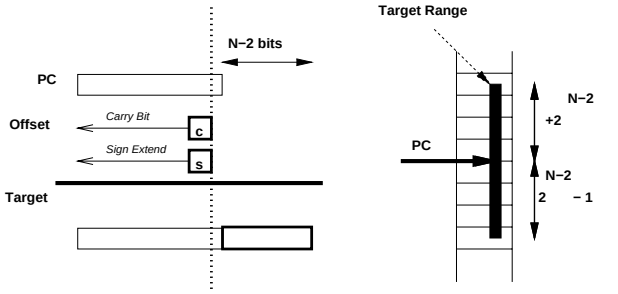
Traditional branch architectures use a PC-relative displacement; Figure 2(a), modeled after the diagrams in [9], schematically illustrates the process. In the encodings, information in lightly outlined boxes is provided or computed at execution time; for example, in Figure 2(a), the PC is available during execution. Heavily-outlined boxes show the information provided by the branch instruction – in Figure 2(a), the instruction provides $n+1$ bits. On the right-hand side, the solid boxes show the range of instructions that can be addressed. A displacement stored in the branch instruction is sign-extended to the size of the program counter and added to the program counter. Each branch can directly address instructions at address $PC - 2^{n-1} - 1 \ldots PC + 2^{n-1}$. For simplicity, we assume the program counter is always aligned on instruction boundaries, since we are chiefly concerned with architectures with fixed-width instructions.

Katevenis [9] proposed several branch encodings where the branch displacement field contains the least significant bits of the branch target address. Figure 2(b), shows one such encoding. Here, the sign bit for the offset and the carry for the addition of the lower bits are computed by the compiler (or linker) and encoded in the instruction. The lower bits can be immediately used to index a cache; concurrent with the cache fetch, the higher order bits are computed and matched against the address tags when the cache fetch returns. If the tags are mismatched with the actual PC, an instruction-cache miss occurs and the pipeline is stalled. During the stall, the program counter is corrected. Since the instruction must include both the carry and the sign bit, an n-bit displacement can only address $PC - 2^{n-2} - 1 \ldots PC + 2^{n-2}$.

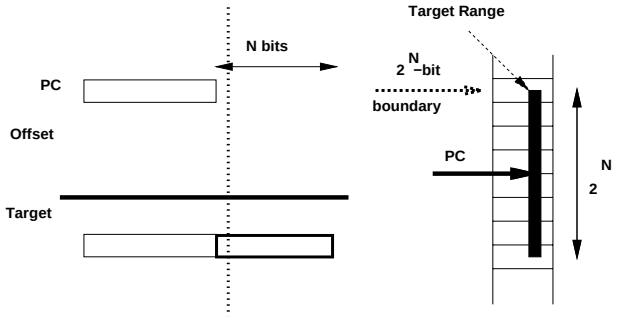
Figure 2(c) diagrams our proposed branch encoding. We use an *explicit displacement* instead of a PC-relative displacement because we need to calculate target addresses in time to use them for the next instruction fetch and, as Katevenis noted, an adder is usually too complex for this purpose. The n-bit displacement is used as the lower order part of the destination address. Each branch can then jump within a span of 2^n instructions. Every direct branch within that 2^n instruction span can only branch



(a) Traditional Sign-Extended PC-Relative Branch



(b) Compiler-Assisted Sign-Extended Branch



(c) Latched Branches (Not Sign-Extended)

Figure 2: Alternate Branch Methods

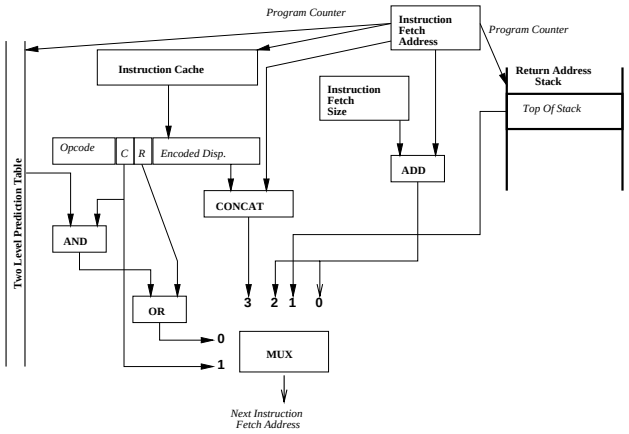
within that span. To branch outside that span, a indirect jump must be used. This idea is not new – a similar mechanism is used in the Crisp processor [6]. In that architecture a branch destination is included in every decoded instruction, resulting in very large instructions – 192 bits. However, the decoded instruction cache must updated on each cycle; this may limit the processor cycle time.

We rely on the program linker to compensate for the limited branching – after a fashion, we are applying the "RISC design philosophy" to branch architectures - we let the software (linker) share the burden of making the hardware efficient and inexpensive. The chief complication with an explicit displacement encoding is that PC-relative code relocation, important for shared program libraries, is no longer possible without dynamic relinking. However, consider using an architecture such as the DEC Alpha AXP, which uses a 21-bit branch displacement for word-aligned instructions in a 64-bit instruction address space. The instruction space is broken into $2^{64-21-2}$, or $\approx$ 2 trillion 'segments' of 8MB each. Branches within each 8MB segment use a single explicit displacement. Each segment can be relocated to $\approx$ 2 trillion different locations without modification. Such large segments would address almost all programs we have encountered. We can use simple profile-based optimizations to reduce the frequency of inter-segment branches if a single program is larger than the branch displacement. For smaller programs, inter-segment branches will occur most frequently when using shared code libraries, where the runtime destination is not known until execution time. These function calls typically used indirect jumps in existing operating systems.

In the past, segment architectures have been greeted with less than overwhelming enthusiasm, due to limited segment sizes. However, do don't have segments – we just have limited branches. It seems unlikely that the size of programs will grow without limit, in part because additional software implies additional complexity. Beyond some point, monolithic programs are difficult to maintain. Programs larger than 64KB are common; whether this will be true with programs larger than 8MB remains to be seen.

Figure 3 shows our proposed instruction fetch architecture. We assume that the instruction cache or the instruction itself contains two instruction type bits, encoded as shown. The displacement indicates the branch target displacement within the same 8MB code segment. This completely eliminates the misfetch penalty as long as the branch target is predicted correctly for conditional branches, indirect branches and returns. For indirect branches, profile-based information can used to predict the next instruction. If the program has not been profiled, or this is an inter-segment branch, the program will always mispredict indirect branches. Profile-based prediction of indirect function calls has been shown to be effective and important for the C++ programming language [4], where such branchs occur frequently. We assume return instructions use a 32-entry return address stack for predicting the branch address.

In the BTB approach, the address is offered to the instruction cache and the BTB. Both the BTB and the instruction cache must respond in a single cycle. The pattern history table must also respond in a single cycle, but the results of the BTB lookup are



C	R	Instruction Type
0	0	Non-branch instruction
0	1	Return instruction
1	0	Conditional branch
1	1	Unconditional branch or indirect branch

Figure 3: A Schematic Representation of the Proposed Branch Prediction Using Two-Level Prediction And Explicit Displacement Encoding

needed to predict the branch destination; thus, additional information is stored in the BTB to 'cache' the prediction result for the next branch (see Figure 1).

By comparison, in the proposed architecture, the instruction address is concurrently offered to the instruction cache, the pattern history table and an adder that computes the 'fall-through' address (PC+4). All of these must respond by the end of the cycle, but no additional updating or forwarding of the BTB or PHT is needed. In most processors, instruction fetch limits the processor cycle time. However, we interpose a single gate layer (OR) and a multiplexor between the instruction cache and the next instruction fetch address. We also eliminate the need for the BTB; this may provide enough space for a two-level on-chip cache, which may improve overall performance. We are not familiar enough with the trade-off's between the BTB and instruction cache access time to affirm that the proposed architecture wouldn't have a longer cycle time. More detailed design and experimentation is needed.

6.2 Figuring out it's a Branch

Recall that we need to know that an instruction is a branch with an explicit displacement (e.g., call/branch/conditional branch) in order to fetch using the explicit displacement. We assumed there is sufficient flexibility in the instruction set architecture to encode information directly into the instruction. If this is not possible, the additional information may be recorded with the instruction cache; i.e., when a branch instruction is decoded, bits associated with that instruction are set in the cache line. This would let the processor use information about the decoded instruction the next time the instruction is executed. Similar mechanisms are used in extant architectures; e.g., the Alpha AXP 21064 stores branch prediction information with words in the on-chip instruction cache. Due to lack of space, we do not explore this option further in this paper.

We *did* simulate an *instruction type prediction table* (ITPT) to see if we could reduce the number of instruction bits needed to predict branches. If two bits are available in the instruction, we say the instruction type is **known** for branch prediction. If a single unique bit is available in the instruction, we use the bit to indicate that the instruction is a branch, and use a table to determine the **branch type**. We use this table only when we encounter a branch, as indicated by the unique bit in the instruction. If no unique bits are available we use a table to predict the **instruction type**. We use this table to predict the type of *each* instruction, since we have no way of knowing whether it is a branch or not.

The ITPT is similar to the branch pattern history table (PHT). We used a direct-mapped table indexed by the program counter. Each entry contains two bits, encoding the instruction type as shown in Figure 3. In the **branch type** method, the table is only updated after branches are decoded; in the **instruction type** method, the table is updated after each instruction is decoded. As expected, the more information encoded in the instruction, the more accurately the instruction type can be determined, decreasing the misfetch penalty. If the **instruction type** method can accurately predict branches, it may be more useful than associating information with the instruction cache, because the information can be fetched for future instructions. The **instruction type** table only relies on the program address.

7 Performance of Alternative Architecture

We considered three mechanisms to predict the type of an instruction, with corresponding performance shown in Table 4. When using the **instruction type** method, some non-branch instructions will be erroneously predicted to be branches. Thus, an address other than the next address will be fetched, causing a pipeline bubble. To simplify comparison, we assign all of these delays, shown by the column labeled **%MfI**, to the branches in the program. For example, if a program executes 90 `load` and ten **branch** instructions, and we erroneously predict four `load`s to be `branch`s, the **%MfI** would be 40%. As before, the **%MfB** column shows the number of misfetches due to branches, and the **%MpB** columns shows the number of mispredicted branch destinations. The **BEP** column gives the branch execution penalty for a processor with 1-cycle misfetch and 5-cycle mispredict penalties. We used the same 4096-entry two-bit table to predict the outcome of conditional branches as was used in the GAg(12) method in §5.

Using the **branch type**, we know that a branch *is* a branch; we just don't know what kind of branch it is. Thus, non-branch instructions are properly fetched, and the **%MfI** column would always be zero. Using the **known** type, we can detect *all* branches and properly determine the appropriate type. The only source of penalties is mispredicting the outcome of conditional branches, indirect jumps and returns. The return stack avoids most return mispredictions.

	16K-entry Instruction Type				4K-entry Branch Type			Known	
Program	%MfI	%MfB	%MpB	BEP	%MfB	%MpB	BEP	%MpB	BEP
alvinn	0.00	0.00	0.23	0.01	0.14	0.23	0.01	0.23	0.01
compress	0.00	0.00	9.86	0.49	0.00	9.86	0.49	9.86	0.49
eqntott	0.00	0.00	2.97	0.15	0.00	2.97	0.15	2.97	0.15
espresso	0.36	0.36	5.25	0.27	0.06	5.25	0.26	5.25	0.26
gcc	3.24	2.95	13.42	0.73	1.63	13.42	0.69	13.42	0.67
li	0.00	0.00	5.08	0.25	0.25	5.08	0.26	5.08	0.25
sc	0.16	0.16	3.59	0.18	0.15	3.59	0.18	3.59	0.18
cfront	4.12	3.84	12.25	0.69	2.06	12.25	0.63	12.25	0.61
db++	0.00	0.01	15.71	0.79	0.19	15.71	0.79	15.71	0.79
idl	1.09	0.95	13.35	0.69	0.21	13.35	0.67	13.35	0.67
groff	3.25	3.14	9.03	0.52	1.88	9.03	0.47	9.03	0.45

Table 4: Misprediction and Branch Execution Penalties for Different Instruction Encoddings (**No static profiling**)

	16K-entry Instruction Type				4K-entry Branch Type			Known	
Program	%MfI	%MfB	%MpB	BEP	%MfB	%MpB	BEP	%MpB	BEP
alvinn	0.00	0.00	0.21	0.01	0.14	0.21	0.01	0.21	0.01
compress	0.00	0.00	9.86	0.49	0.00	9.86	0.49	9.86	0.49
eqntott	0.00	0.00	1.41	0.07	0.00	1.41	0.07	1.41	0.07
espresso	0.36	0.36	5.09	0.26	0.06	5.09	0.26	5.09	0.25
gcc	3.24	3.00	11.91	0.66	1.68	11.91	0.61	11.91	0.60
li	0.00	0.00	3.68	0.18	0.25	3.68	0.19	3.68	0.18
sc	0.16	0.16	3.01	0.15	0.15	3.01	0.15	3.01	0.15
cfront	4.12	3.90	11.09	0.63	2.14	11.09	0.58	11.09	0.55
db++	0.00	0.01	3.05	0.15	0.25	3.05	0.16	3.05	0.15
idl	1.09	1.13	1.26	0.09	0.26	1.26	0.07	1.26	0.06
groff	3.25	3.20	4.89	0.31	1.94	4.89	0.26	4.89	0.24

Table 5: Misprediction and Branch Execution Penalties for Different Instruction Encoddings (**With static profiling**)

If we have previously executed the program, we can use profile based indirect jump prediction [4]. In Table 5, we use the profile information from the same input to predict the indirect jumps. The BTB-based architectures can predict indirect jumps without profiling. Profiling mainly benefits the C++ programs because they execute many indirect jumps, although `eqntott` also benefits due to the structure of the quick sort routine used in that program.

In Tables 4 and 5, we varied the size of the instruction type prediction tables used in the different methods to have the performance be roughly equivalent. Each entry in the table requires two bit; thus, the 16K-entry table requires 32K bits (4K byte) and the 4K-entry table requires 8K bits (1K byte) of memory. Using the BEP metric, the **branch type** method is almost as useful as the **known** method using a fairly small table. We were very surprised that the **instruction type** method performed as well as it did. This encourages us to investigate the possibility of associating this information with the entries in the instruction cache.

There are some penalties that we did not model, because we don't know how expensive they would be. If a branch is predicted to be a `return` and we pop the return stack, we may need to place the popped address back on the return stack. More importantly, by mispredicting instructions to be branches, we may issue instruction fetches from random parts of memory – these should not cause faults until we actually determine if the instruction was a branch. However, all processors that speculativly fetch instructions may encounter this problem to some degree due to mispredicted conditional branches.

7.1 Performance Comparison

In general, the BTB-based architecture has slightly better performance for most programs unless we use profiles to predict indirect jumps. This is particularly true for the C++ programs and `eqntott` since those programs have many indirect jumps. After profiling, all programs except `eqntott` and `db++` have better performance; these programs have indirect function calls that are very difficult to predict using static methods. For programs with many conditional branches, (`gcc`, `cfront` and `groff`), the proposed architecture performs better than the BTB architecture – the 512-entry BTB suffers from many capacity misses and conflicts for these programs. However, even for these programs, the small number of indirect function calls degrades performance unless profiling is performed.

These results suggest that using a small BTB in conjunction with the other mechanisms would be fruitful. The BTB would

only store indirect jumps, and could be very small, particularly if we could include only indirect jumps that are difficult to predict statically. The smaller BTB will probably not increase the overall processor cycle time. In related work [4], we found that indirect jumps in many programs can be accurately predicted using profile information. For example, we encountered 547 unique indirect jumps when tracing `idl`, but all of these are easy to predict. By comparison, `eqntott` and `db++` each had less that thirty indirect jumps, but they are very difficult to predict statically.

8 Conclusions

Using metrics that more accurately reflect pipeline delays, we have shown that the 'GAg' branch architecture is as effective as the 'PAs' architecture, and the GAg method uses less on-chip memory. When using slightly more on-chip memory, the 'GAg' can be more effective than PAs, while still being simpler to implement.

We have also shown that dispensing with the BTB entirely and using an explicit branch displacement can be as effective as having a large BTB. We believe that combining a small BTB devoted to indirect jumps would result in a branch architecture that uses few resources and has excellent performance, particularly for programs with a large number of branches.

We believe our results apply to the current wide issue processors being developed, where several instructions are fetched concurrently; the information needed by the proposed architecture can be directly extracted from the instructions, or can be stored in the instruction cache for an entire basic block. The use of explicit branch displacements pose no serious problems for current software in 64-bit architectures. Lastly, there is another advantage to the ITPT-based designs – the proposed combination of the branch prediction mechanisms do not depend on the size of the address range. By comparison, the size of a non-direct mapped BTB would increase as the instruction address range increases.

This work was funded in part by NSF grant No. ASC-9217394. We would like to thank Andy Glew and John Feehrer for comments on earlier versions of the paper, and Alan Eustace and Amitabh Srivastava for providing ATOM, which greatly simplified our work.

References

[1] T. Ball and J. R. Larus. Branch prediction for free. In *1993 SIGPLAN Confernce on Programming Language Design and Implementation*. ACM, June 1993.

[2] Brian Bray and M. J. Flynn. Strategies for branch target buffers. In *24th Workshop on Microprogramming and Microarchitecture*, pages 42–49. ACM, ACM, 1991.

[3] Brad Calder and Dirk Grunwald. Branch alignment. Technical report, Univ. of Colorado, 1994. (In Preperation).

[4] Brad Calder and Dirk Grunwald. Reducing indirect function call overhead in c++ programs. In *Proceedings of the 21st Annual ACM Symposium on Principles of Programming Languages*, January 1994.

[5] Brad Calder, Dirk Grunwald, and Benjamin Zorn. Quantifying behavioral differences between C and C++ programs. Technical Report CU-CS-698, Univ. of Colorado-Boulder, January 1994.

[6] David R. Ditzel and Hubert R. McLellan. Branch folding in the CRISP microprocessor: Reducing branch delay to zero. In *14th Annual International Symposium of Computer Architecture*, pages 2–9. ACM, ACM, June 1987.

[7] J. A. Fisher and S. M. Freudenberger. Predicting conditional branch directions from previous runs of a program. In *Fifth International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 85–95, Boston, Mass., October 1992. ACM.

[8] David R. Kaeli and Philip G. Emma. Branch history table prediction of moving target branches due to subroutine returns. In *18th Annual International Symposium of Computer Architecture*, pages 34–42. ACM, May 1991.

[9] Manolis G. H. Katevenis. *Reduced Instruction Set Computer Architecture for VLSI*. ACM Doctoral Dissertation Award Series. MIT Press, 1985.

[10] Johnny K. F. Lee and Alan Jay Smith. Branch prediction strategies and branch target buffer design. *IEEE Computer*, 21(7):6–22, January 1984.

[11] David J. Lilja. Reducing the branch penalty in pipelined processors. *IEEE Computer*, pages 47–55, July 1988.

[12] Scott McFarling. Combining branch predictors. TN 36, DEC-WRL, June 1993.

[13] Scott McFarling and John Hennessy. Reducing the cost of branches. In *13th Annual International Symposium of Computer Architecture*, pages 396–403. ACM, 1986.

[14] S.-T. Pan, K. So, and J. T. Rahmeh. Improving the accuracy of dynamic branch prediction using branch correlation. In *Fifth International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 76–84, Boston, Mass., October 1992. ACM.

[15] Chris Perleberg and Alan Jay Smith. Branch target buffer design and optimization. *IEEE Transactions on Computers*, 42(4):396–412, April 1993.

[16] Karl Pettis and Robert C. Hansen. Profile guided code positioning. In *Proceedings of the ACM SIGPLAN '90 Conference on Programming Language Design and Implementation*, pages 16–27. ACM, ACM, June 1990.

[17] J. E. Smith. A study of branch prediction strategies. In *8th Annual International Symposium of Computer Architecture*. ACM, 1981.

[18] Amitabh Srivastava and Alan Eustace. Atom: A system for building customized program analysis tools. TN 41, DEC-WRL, January 1994. (To appear in PLDI'94).

[19] D. W. Wall. Limits of instruction-level parallelism. In *Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 176–188, Boston, Mass., 1991.

[20] Tse-Yu Yeh and Yale N. Patt. Alternative implementations of two-level adaptive branch predictions. In *19th Annual International Symposium of Computer Architecture*, pages 124–134, Gold Coast, Australia, May 1992. ACM.

[21] Tse-Yu Yeh and Yale N. Patt. A comprehensive instruction fetch mechanism for a processor supporting speculative execution. In *25th Workshop on Microprogramming and Microarchitecture*, pages 129–139, Portland, Or, December 1992. ACM.

[22] Tse-Yu Yeh and Yale N. Patt. A comparison of dynamic branch predictors that use two levels of branch history. In *20th Annual International Symposium of Computer Architecture*, pages 257–266, San Diego, CA, May 1993. ACM.