Branch Prediction Architectures for 64-bit Address Space

Brad Calder Dirk Grunwald

CU-CS-690-93 Nov 1993

University of Colorado at Boulder

Technical Report CU-CS-690-93
Department of Computer Science
Campus Box 430
University of Colorado
Boulder, Colorado 80309

Branch Prediction Architectures for 64-bit Address Space

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)

Nov 1993

Abstract

Processor architectures will increasingly rely on issuing multiple instructions to make full use of available processor resources. When issuing multiple instructions on conventional processors, accurate branch prediction is critical to performance; mispredicted branches may mean that ten's of cycles may be wasted. 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; proposals commonly suggest that 16 kilobytes of cache be devoted to branch history and prediction information.

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, but other branch prediction techniques become more applicable. 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 offers $\approx$ 21% better performance than previously proposed alternatives. Our design requires few hardware resources, and is oriented towards 64-bit architectures.

1 Introduction

Conventional processor architectures, particularly superscalar designs issuing several instructions concurrently, are extremely sensitive to control flow changes. For example, 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 the *instruction misfetch penalty*.

*This work was funded in part by NSF grant No. ASC-9217394.

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, and the **memory access** stage updates the program counter to reflect the target address of the branch destination. 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 misfetched penalty due to incorrectly fetched instructions. For example, the branch mispredict penalty for each pipeline of the Digital AXP 21064 processor is 5 cycles. Since there are two different pipelines, incorrectly predicting conditional control flow implies the processor lost the opportunity to execute 10 instructions. By comparison, the AXP 21064 would lose only two instructions from an instruction misfetch penalty. As processors issue more instructions concurrently, these costs increase, and the instruction fetch penalty becomes increasingly important because 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 [9].

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 instruction 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 [13, 8, 12].

More recently, there has been considerable interest in using BTB's to reduce instruction misfetch penalties; for example Yeh *et al* [16] propose using a 16KB 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 prcoessors 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 and each BTB entry contains a per-basic block prediction history register, used to index into a 2-level branch history table [15, 17].

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 using:

- A code transformation called *branch alignment* [3] that increases the effectiveness of common branch prediction mechanisms. This transformation can either be directed by the compiler or by directly transforming the program binary using profile information.
- A decoupled branch prediction mechanism that allows us to accurately predict a conditional branch's direction even when a BTB miss occurs on the branch's address. This is done by removing the prediction history registers from each BTB entry and using one global history register.

- We propose a new architecture that quickly determines the instruction type, either using information from the instruction or an *Instruction Type Prediction Table* (ITPT). This is done to predict what instruction should be fetched in the next cycle, reducing misfetch delays.

2 Prior Work

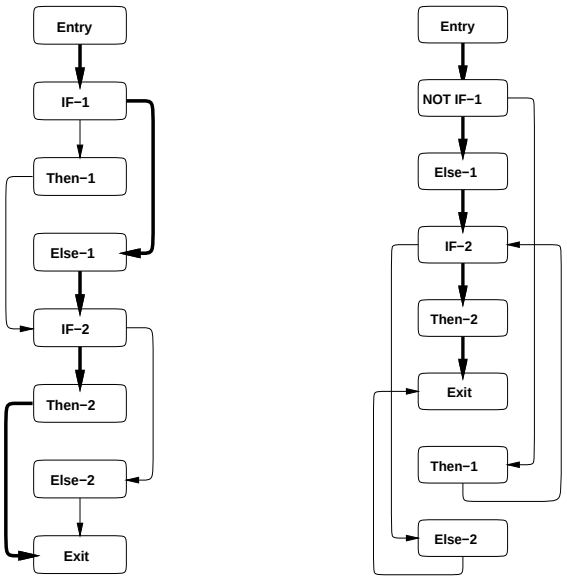
In order to contrast our instruction fetch architecture to a proposed aggressively designed architecture, we describe the instruction fetch architecture proposed by Yeh *et al*, briefly describing their branch prediction mechanism and the structure of their branch target buffer (BTB). Their mechanism is a logical continuation of currently proposed and implemented designs. In order to understand how branch alignment improves branch prediction, we briefly review proposed branch prediction methods. We mainly consider *tagless* branch prediction methods that simply predict whether conditional branches are taken or not-taken. There are a number of other prediction mechanisms that also provide the address of the branch target if the branch is accurately predicted, such as the previously mentioned branch target buffers [8, 12].

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 [13, 8, 10, 1] to profile-based methods [10, 14, 5]. 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*, either using tables or explicit branch registers as in the case of a BTB. There are myriad variations on the general idea; typically, the cache contains from 32 to 512 entries with varying degrees of associativity. The address of a branch site is used as a tag in the BTB, and matching data is used to predict the branch. As mentioned, Yeh [16] also uses this information to direct the instruction fetch for returns, unconditional jumps and indirect jumps.

Other designs take the BTB and eliminate 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. 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 [13, 8, 10]. The advantage of these bit-table techniques is that they keep track of very little information per conditional branch site and are very effective in practice. More recently Pan *et al* [11] and Yeh and Patt [15, 17] 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. In this paper, we are primarily concerned with 2-level prediction methods; see [11, 15, 17] 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. This is why BTB's are still very useful for eliminating instruction misfetch penalties.



(a) Original Program (b) After Branch Alignment

Figure 1: Illustration of Branch Aligning Code Transformations

3 Branch Alignment

In the architecture designs that we study, we have omited details on updating the prediction history table for clarity; the structure is essentially as proposed by Pan [11], using a degenerate prediction history register and two-bit saturating counters. Other two level adaptive prediction history mechanisms could also be used [17] to achieve the same performance. Bit-based branch prediction mechanisms commonly map different branch locations to the same prediction entry. Several researchers, including Pan [11] and Yeh [17], have noted that such *table conflicts* reduce the accuracy of their prediction mechanisms. There are two alternative solutions to decrease table conflicts: increase the size of the table (and hopefully decrease the conflicts) or decrease the importance of conflicts.

In the paper [3], we show how bit-table mechanisms, such as 2-bit prediction or branch-correlation, can be improved by modifying the program. Rather than increase the size of prediction tables, we reduce the occurrence or importance of conflicts. One method for this is *branch alignment*; we profile the program behavior, recording the most likely direction for each branch. We then modify the program binary to "align" the most likely branch direction across different branches. Consider the example in Figure 1, representing a sequence of two `If..then...else` constructs. The dark lines indicate the 'most likely' path, based on profile information from prior executions. The program in 1(a) contains two branches, where the first branch is more likely to be 'taken' and the second branch is more likely to be 'not taken.' Figure 1(b) shows the transformed program, where the condition codes have been inverted so that *both* branches are most likely to be 'not taken'. Consider the impact of this modification in a computer with a single bit for dynamic branch prediction. Both branches would be mapped to the same prediction bit. In the original program, the prediction state would typically alternate between the two branches, providing little predictive

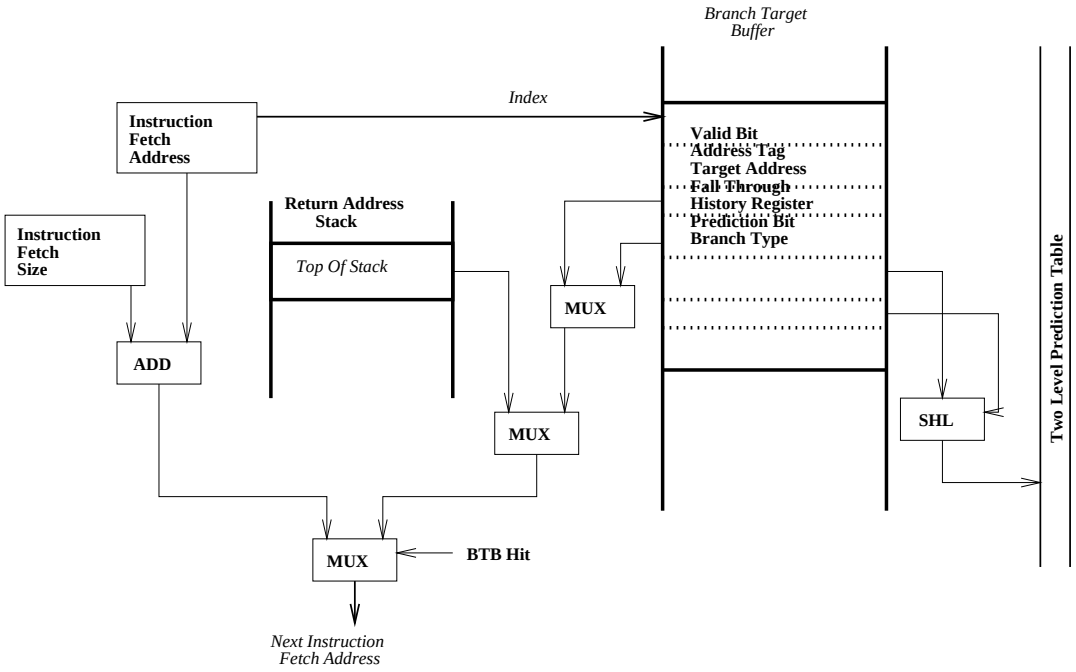


Figure 2: A Schematic Representation of the Branch Prediction Using Two-Level Prediction and Branch Target Buffers, As Proposed By Yeh and Patt

information the next time the branch is taken. By aligning the branches, the prediction bit would more frequently indicate the correct branch direction.

Branch alignment is shown in [3] to improve the accuracy of common 2-bit and branch-correlation techniques, allowing the size of prediction tables to be reduced while maintaining accurate prediction rates. Branch alignment can be implemented by existing compilers and binary transformation tools. Branch alignment relies on the anonymity of the predicted target, simply predicting 'taken' or 'not taken' rather than the target address. Branch alignment reorders the basic blocks of a program, inverting branch conditions to maintain the program semantics. Occasionally, this requires adding unconditional jumps in infrequently executed portions of the program. In this paper, we do not include this overhead from the inserted jumps; however, they would affect each prediction mechanism similarly.

The work described in this paper was started prior to extensive experience with branch alignment, and we simulated *maximal branch alignment*. In practice, not all branches can be profitably aligned. In this paper, we assumed $\approx$ 90% of branches could be aligned; in later work [3], we show that $\approx 80 - 75\%$ alignment is more realistic. A later version of this paper will include the effects of proper alignment values. Note that we only show how current architectures benefit from alignment. Our final proposed architecture does not benefit from alignment. Thus, by assuming a greater degree of branch alignment, we are over-estimating the improvement possible in current architectures.

4 An instruction fetch and branch prediction architecture

Figure 2 is a schematic representation of the branch prediction and instruction fetch architecture suggested by Yeh and Patt [16]. The current instruction address is concurrently offered to the instruction cache (not

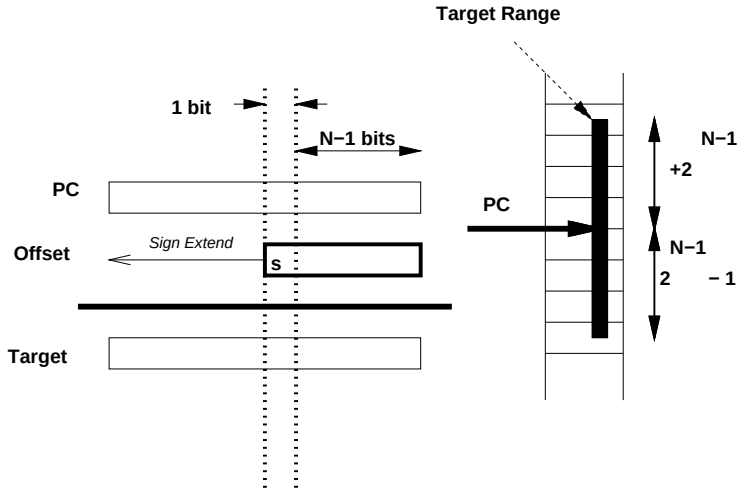
shown), providing the actual instruction, and to the BTB. They use 32-entry return address stack to handle return instructions. Depending on the branch type and the prefetched branch prediction information, all stored in the BTB, the destination, fall-through or return stack address is selected as the next instruction fetch. Following this, the prediction history table is updated; this can occur several cycles later with little penalty – see [16] for more details. Our proposed design differs in several ways, described below.

Branch destination decoding: Conditional branches, unconditional jumps and statically predicted indirect jumps use the displacement from the fetched instruction. We encode branch displacements using an explicit displacement; this is done to rapidly extract a branch destination from the instruction cache rather than from a BTB. Traditional branch architectures use a PC-relative displacement; Figure 3(a) schematically illustrates the process. In the encodings, information in lightly outlined boxes is provided or computed at execution time; for example, in Figure 3(a), the PC is available during execution. Heavily-outlined boxes show the information provided by the branch instruction – in Figure 3(a), the instruction provides $n+1$ bits. On the right-hand side, the solid boxes show the range of instructions that can be addressed. For simplicity 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 $PC - 2^{n-1} - 1 \ldots PC + 2^{n-1}$. The left side of each diagram in Figure 3 shows the branch encoding, while the right hand side shows the instructions that can be addressed by each encoding. For simplicity, we assume the program counter is always aligned on instruction boundaries, since we are chiefly concerned with architectures with fixed-width instructions.

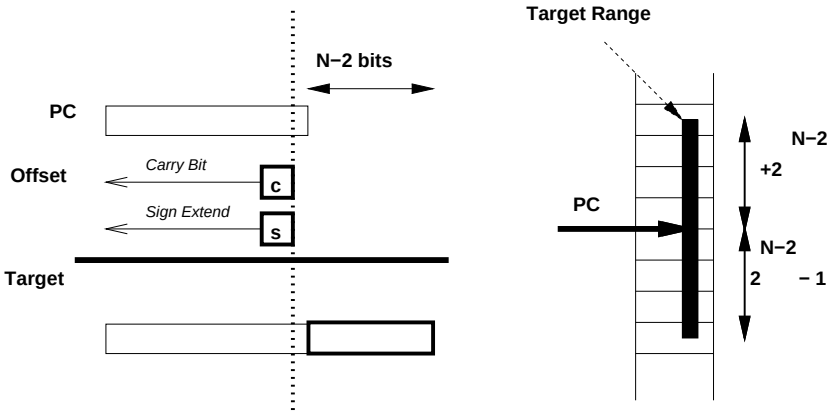
Katevenis [7] proposed several branch encodings where the branch displacement field contains the least significant bits of the branch target address. Figure 3(b), modelled after the diagrams in [7], shows one such encoding. Here, the sign bit for the offset and the carry for the addition of the lower 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 $PC - 2^{n-2} - 1 \ldots PC + 2^{n-2}$. This is sufficient for most subroutines.

We also 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 may be too complex for this purpose. Figure 3(c) diagrams our branch encoding. 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 within that span. To branch outside that span, a register-indirect jump must be used.

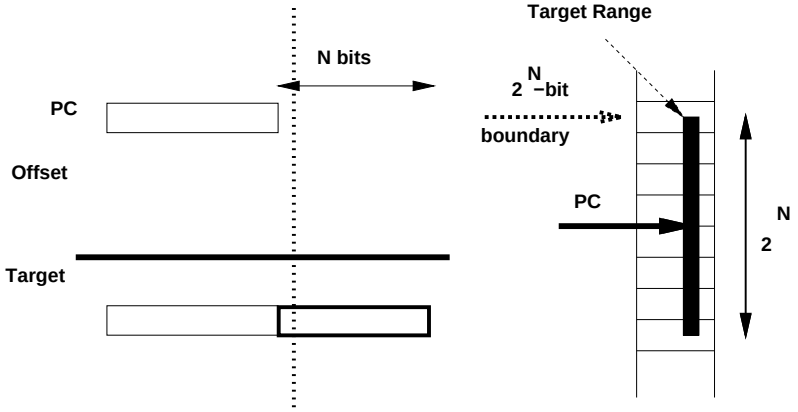
We rely on the program linker to compensate for the limit branching – after a fashion, we are applying the "RISC design philosophy" to branch architectures - we let the software (linker) share the burden of making the hardware efficient and inexpensive. The chief complication with an explicit displacement 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; for example, the largest file distributed with the 64-bit OSF/1 operating system is $\approx 5MB$. With more compact and complex



(a) Traditional Sign-Extended PC-Relative Branch



(b) Compiler-Assisted Sign-Extended Branch



(c) Latched Branches (Not Sign-Extended)

Figure 3: Alternate Branch Methods

instruction encodings, the branch offset may be increased; later, we consider an *instruction type prediction table* to rapidly predict the type of complex instructions. We can use simple profile-based optimizations to reduce the frequency of inter-segment branches if a single program is larger than the 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 register-indirect in existing architectures.

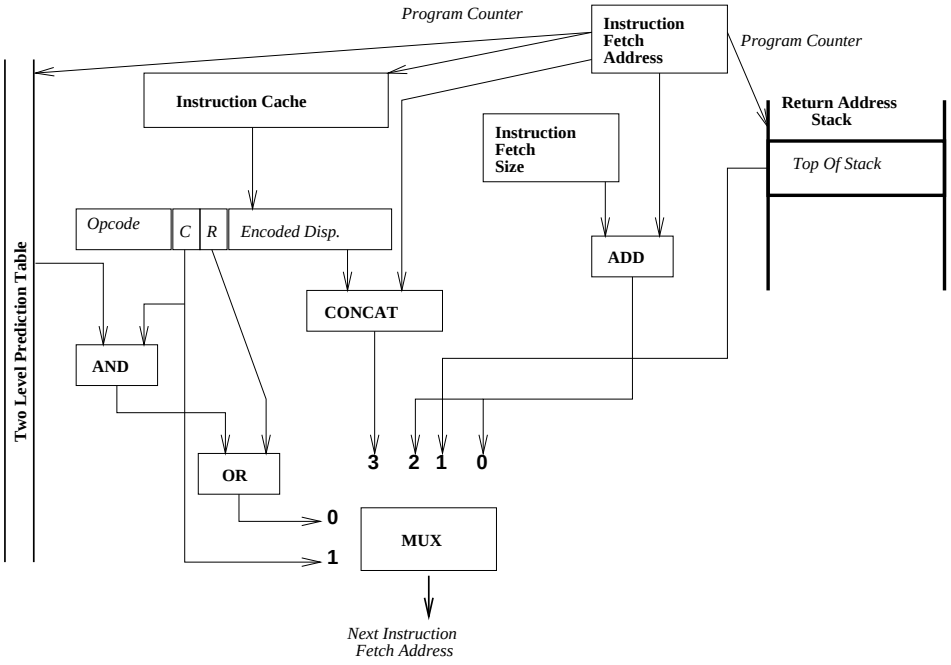
In the past, segment architectures have been greeted with less than overwhelming enthusiasm, due to limited segment sizes. It seems unlikely that the size of programs will grow without limit, in part because additional software implies additional complexity. Beyond some point, monolithic programs are difficult to maintain. Programs larger than 64KB are common; whether this will be true with programs larger than 8MB remains to be seen. If we reduce the size of the reachable code segment, we can increase the number of different code segments reachable via a single branch. For example, we may wish to reach 4 distinct 2MB code segments. For example, consider a C program running on UNIX that uses the standard C library, the math library and the X11 library. These libraries are usually less than 1-2MB in size, but the program must call all three libraries frequently. By specifying the relocation mappings at link-time, the libraries can be organized so all inter-segment calls use the displacement information. These choices are driven by software architecture rather than hardware architecture; our only requirement is that the linker be able to select between inter-segment and intra-segment addresses during program linking.

Figure 4 shows our proposed instruction fetch architecture. We assume that the instruction cache or the instruction itself contains two bits, encoded as shown. Indirect branches and inter-segment branches rely on static prediction in order to avoid pipeline bubbles. The displacement indicates the likely indirect 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 is used. If the program has not been profiled, or this is an inter-segment branch, the program will always mispredict indirect branchs. Profile-based prediction of indirect function calls has been shown to be effective and important for a commonly-used object-oriented programming language [4]. We did not record the occurrence of intersegment branches attributable to shared libraries, because our code instrumentation facility only worked on statically bound executables.

We assume return instructions use a 32-entry return address stack for predicting the branch address; this has been shown to be very effective [6].

Instruction Type Prediction Table: We assume there is sufficient flexibility in the instruction set architecture to encode information directly into the instruction. If this is not possible, the additional information can be recorded with the instruction cache. We examine other encodings that requires less information in the instruction at the cost of another prediction mechanism. Here, we use a single bit stored in the instruction to indicate if the instruction is a branch, and use an *Instruction Type Prediction Table* (ITPT) to predict the type of the branch instruction (direct branch, conditional direct branch, indirect branch or no branch). This mechanism may be useful if we wish to expand the size of the branch displacement field because a more compact instruction encoding may double the size of addressable code segments.

The ITPT is similar to the prediction history table (PHT); it is a 2-bit table directly indexed using the current program counter. Alternatively, we also consider a degenerate case where we use no control bits from the instruction to determine the instruction type; we predict the type of *each* instruction using only the ITPT. In the latter case, the table is updated after each instruction is decoded, while in the former case, the table is only updated after breaks are decoded. As expected, the more information encoded in the instruction, the more accurately the instruction type can be determined, decreasing the misfetch penalty.



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

Program	# of Instructions	# Dynamic Breaks	# unique CB's Encountered	Percentage of Breaks				
				%CB	**%IJ**	**%UB**	**%PC**	**%Ret**
compress	94,345,542	14,689,565	222	79.92	0.00	16.66	1.71	1.71
eqntott	426,520,806	104,614,829	357	95.59	1.02	2.10	0.14	1.16
espresso	604,811,988	89,958,042	1,633	94.04	0.09	1.73	2.02	2.12
gcc	137,821,273	22,666,313	7,481	78.89	0.36	6.77	5.62	5.98
li	58416,738	10,264,842	474	65.32	0.39	6.86	13.15	13.35
sc	657,613,954	119,723,123	1,157	83.53	0.16	3.33	5.59	5.75
ditroff	53,132,806	8,863,360	1,033	71.01	0.10	12.92	7.93	8.02
xdvi	17,402,519	4,466,390	3,669	89.41	0.03	7.16	1.64	1.68
xfig	22,601,237	5,502,761	4,245	88.05	0.11	6.88	2.37	2.49
xtex	21,898,978	5,313,169	5,165	89.55	0.07	6.49	1.86	1.94
cfront	11,495,967	2,198,380	4,894	69.80	0.00	14.53	7.10	7.10
doc	406,673,898	70,039,897	5,391	69.30	7.58	3.62	5.95	13.53
groff	83,938,463	15,791,074	2,364	65.13	2.81	9.73	9.26	12.08
idl	151,419,127	28,409,247	1,006	41.57	10.53	17.73	9.76	20.29
idraw	184,930,700	28,811,665	6,447	64.90	5.17	5.02	9.78	14.95
morpher	52,131,648	9,205,708	4,044	71.84	4.62	4.66	7.09	11.71
rtsh	822,053,075	88,131,470	1,389	59.41	6.29	3.13	12.43	18.73

Table 1: Measured Attributes of the Traced Programs

5 Experimental Methodology

Our comparison used trace-based simulation. We instrumented the integer programs from the SPEC92 benchmark suite and other sizable programs, including object-oriented programs written in C++. We used a modified version of QPT[2] to trace the program execution. The programs were compiled on a DECstation 5000 using either version 2.4.5 of either the Gnu C compiler or Gnu G++ V2.4.5. All programs were compiled with standard optimization (`-O`). We constructed a simulator to analyze the program traces. Typically the simulator was run once to collect information on call and branch targets, and a second time to use prediction information from the prior run. We modified QPT to indicate the cause of a basic-block transition: a conditional or unconditional direct branch, indirect branch or fall-through. Because we used execution-driven simulation, we were able to simulate the complete program execution in most cases. For `sc` and `eqntott`, we cut off the simulation after encountering 100 million conditional branches. For the SPEC92 programs, we used the largest input distributed with the SPEC92 suite.

The alternate programs include: `ditroff`, a common text processing program, `xdvi` and `xtex`, two different previewers for TEX output, `xfig`, a figure-drawing program, `cfront`, version 2.1 of the AT&T C++ language preprocessor, `doc`, a WYSIWYG document editor written in C++ using the InterViews 3.0.1 class library, `groff`, a version of the `ditroff` text formater written in C++, `idl`, a C++ parser for the CORBA interface description language, `idraw`, a drawing program using C++ and InterViews, `morpher`, a sample program from the InterViews distribution that "morphes" figures, and `rtsh`, a ray-tracing program written in C++. For the alternate programs, we used sizable inputs we hoped would exercise a large part of the program; for example, we formated a long man page with `groff`, previewed a sizable document with `xtex`, and briefly edited an existing document with `doc`.

Table 1 shows the basic program statistics for the programs we instrumented. The first columns lists the number of instructions simulated, the second column indicates the number of breaks in control flow that were simulated. The third column indicates the number of conditional branches encountered during the simulation; the programs may contain additional conditional branches, but they were not executed during our tracing. The last four columns break down the number of branches into five classes: conditional branches (**CB**), indirect jumps (**IJ**), unconditional branches (**UB**), procedure calls (**PC**) and procedure returns (**RET**). Note that only 63% of the branches in C++ programs arise from conditional branches. In comparison, conditional branches constitute 83% of the breaks in the SPEC92 C programs. In part, this is caused by the increased number of procedure calls in the non-SPEC92 programs, including indirect procedure calls in the C++ programs.

6 Experimental Results

In order to compare our proposed architecture to one using BTB's, we simulated both our system and Yeh & Patts model [16]; 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 different programs than considered in [16], and that we're using a different base architecture and different compiler, our results for their architecture reflect the performance noted in [16].

Yeh & Patt defined a formula for a *branch execution penalty* (BEP), reflecting time between a branch being issued and the time the next useful instruction is issued. Values close to zero are desirable. Using this metric, they estimated the number of instructions per cycle (IPC) when executing N instructions assuming a given "ideal instructions per cycle" (*i.e.* the issue width, denoted IIPC). The IPC is specified as $\text{IPC} = N / \left(N/IIPC + \text{BEP} \times N \times \%Br\right)$

Program	Before Branch Allignment		After Branch Allignment	
	Not Taken	Taken	Not Taken	Taken
compress	50.73	49.27	86.07	13.93
eqntott	34.36	65.64	88.10	11.90
espresso	39.74	60.26	85.05	14.95
gcc	41.41	58.59	88.33	11.67
li	53.60	46.40	87.47	12.53
sc	42.92	57.08	87.95	12.05
ditroff	26.59	73.41	94.86	5.14
xdvi	39.92	60.08	90.85	9.15
xfig	40.10	59.90	90.49	9.51
xtex	37.80	62.20	91.07	8.93
cfront	52.67	47.33	89.38	10.62
doc	39.65	60.35	95.10	4.90
groff	47.17	52.83	94.03	5.97
idl	50.89	49.11	97.65	2.35
idraw	42.93	57.07	93.83	6.17
morpher	43.77	56.23	91.29	8.71
rtsh	40.04	59.96	89.32	10.68
Mean	42.61	57.39	90.64	9.36
C++ Mean	45.30	54.70	92.94	7.06
Spec Mean	43.79	56.21	87.16	12.84

Table 2: Branch Alignment Capability

All of our results for the simulations are give in terms of instructions per cycle (IPC) using this metric. There are some problems with the underlying assumptions in this model, which we will describe later. We assumed a branch mispredict penalty of 10 cycles and misfetch penalties of one and two cycles. To facilitate comparison to [16], we assume an ideal IPC (IIPC) of five instructions per cycle. As in [16], we assumed there were no instruction cache misses.

Details of the BTB-based Architecture: The BTB (called a BHT in [16]) 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 prediction history table (PHT), using variants of the 2-level branch prediction techniques. When an entry is added to the BTB, its history register normally is initialized to all ones. When simulating branch alignment, we initialize history registers to zero since "not taken" will be more common. Unless otherwise noted, the 1024-entry PHT uses a PAs(6,16)[17] conditional branch prediction table to predict branch direction. We term their implementation as having a *coupled* PHT and BTB; in order to correctly predict a branch, the branch must be recorded in the BTB and the PHT prediction must be correct.

Program	Fall Through		Static Pred		Branch Align	
	One	Two	One	Two	One	Two
compress	2.94	2.94	2.94	2.94	2.95	2.95
eqntott	3.13	3.13	3.13	3.13	3.12	3.12
espresso	3.34	3.33	3.39	3.35	3.47	3.46
gcc	1.80	1.76	2.22	2.03	2.65	2.57
li	3.56	3.55	3.60	3.58	3.63	3.63
sc	3.10	3.09	3.36	3.28	3.41	3.39
ditroff	3.28	3.23	3.82	3.63	4.06	3.98
xdvi	3.96	3.94	4.18	4.10	4.29	4.26
xfig	2.90	2.85	3.33	3.15	3.71	3.63
xtex	3.40	3.36	3.69	3.56	3.95	3.90
cfront	1.75	1.70	2.10	1.89	2.70	2.57
doc	2.85	2.78	3.42	3.18	3.78	3.65
groff	1.90	1.82	2.59	2.26	3.10	2.88
idl	4.39	4.35	4.47	4.42	4.55	4.52
idraw	2.45	2.33	2.91	2.61	3.25	3.03
morpher	2.66	2.56	3.08	2.82	3.38	3.22
rtsh	4.03	4.02	4.07	4.06	4.08	4.07
Mean	2.82	2.77	3.18	3.01	3.45	3.37
C++ Mean	2.59	2.50	3.05	2.80	3.45	3.31
Spec Mean	2.83	2.81	3.03	2.95	3.17	3.14

Table 3: Effects of Alignment

We defer particulars of our proposed architecture until the last subsection, where the model is actually discussed. The following subsections compare the effects of branch alignment and decoupling the branch prediction register from the BTB.

6.1 Branch Alignment and Misprediction

Table 3 show the percentage of taken and not taken branches for each program, before and after branch alignment. Table 3 shows the impact of branch alignment on the previously described BTB-based architecture. The columns list IPC for the original BTB-based architecture (Fall Through), the original BTB-architecture with static branch prediction (Static Pred), and the original BTB architecture using branch alignment (Branch Align). In the "Static Pred" method, branches not in the BTB are resolved using static profile-based hints. For each column, results are shown assuming one and two instruction misfetch penalties. Branch alignment offers a 22% improvement over the basic BTB, and an 8% improvement over BTB with static prediction. It is clear that branch alignment does more than provide simple profile information; the improvement over static prediction can be attributed to reducing the effect of conflicts in the prediction history table (PHT). Note that branch alignment improves all programs except `eqntott`. There are few static branches in `eqntott`, and the BTB has a 0.001% miss rate; thus, the history information in the BTB entry provides accurate prediction (94.89%) with or without alignment. With branch alignment, the PHT prediction rate falls to 94.87%.

Program	Static Pred		2048, **N**, **F**		2048, **A**, **F**		2048, **A**, **No F**		4096, **A**, **No F**	
	One	Two	One	Two	One	Two	One	Two	One	Two
compress	2.94	2.94	2.90	2.90	2.96	2.96	2.96	2.96	2.98	2.98
eqntott	3.13	3.13	3.11	3.11	3.05	3.05	3.05	3.05	3.28	3.28
espresso	3.39	3.35	3.52	3.50	3.58	3.57	3.58	3.57	3.65	3.64
gcc	2.22	2.03	2.21	2.09	2.76	2.66	2.81	2.74	2.88	2.81
li	3.60	3.58	3.60	3.59	3.62	3.61	3.62	3.62	3.69	3.69
sc	3.36	3.28	3.35	3.30	3.49	3.47	3.51	3.50	3.58	3.57
ditroff	3.82	3.63	3.57	3.45	4.10	4.02	4.16	4.13	4.25	4.22
xdvi	4.18	4.10	4.08	4.02	4.24	4.21	4.26	4.24	4.25	4.23
xfig	3.33	3.15	3.41	3.29	3.63	3.48	3.82	3.76	3.95	3.89
xtex	3.69	3.56	3.72	3.64	3.99	3.94	4.02	3.98	4.10	4.06
cfront	2.10	1.89	2.02	1.90	2.75	2.61	2.81	2.72	2.88	2.78
doc	3.42	3.18	3.65	3.44	3.88	3.74	4.03	3.92	4.06	3.95
groff	2.59	2.26	2.64	2.39	3.17	2.94	3.30	3.14	3.35	3.18
idl	4.47	4.42	4.47	4.42	4.57	4.53	4.59	4.57	4.62	4.60
idraw	2.91	2.61	3.10	2.84	3.32	3.09	3.44	3.23	3.48	3.26
morpher	3.08	2.82	3.16	2.96	3.39	3.22	3.51	3.36	3.54	3.39
rtsh	4.07	4.06	3.96	3.95	4.01	4.00	3.96	3.96	4.07	4.07
Mean	3.18	3.01	3.19	3.07	3.49	3.39	3.54	3.48	3.62	3.55
C++ Mean	3.05	2.80	3.10	2.91	3.50	3.34	3.59	3.47	3.64	3.51
Spec Mean	3.03	2.95	3.03	2.98	3.21	3.18	3.22	3.20	3.31	3.29

Table 4: Effects of Decoupling the PHT from the BTB

6.2 Decoupled Prediction and Fall Throughs

One of the disadvantages of a *coupled* prediction history register, as used in the BTB-based architecture, is that a branch may suffer a misfetch penalty *and* a branch misprediction penalty if it is not in the BTB. However, the PHT may accurately predict the branch, avoiding the branch mispredict penalty. We examined a *decoupled* BTB and PHT, meaning prediction history registers are not stored in the BTB. This allows branches to be predicted on BTB misses. Even though we get a BTB miss, we believe that the decoupled PHT would still contain useful prediction information, especially since the PHT would probably hold 4 to 8 times the number of entries that the BTB would hold. We use a single prediction history register (the "degenerate" configuration of Pan [11]). By not including the 6-bit prediction registers in the BTB, we can afford to increase the PHT size from 1024 2-bit entries to 2048 2-bit entries. Once the prediction information is decoupled from the BTB, we can also increase the effectiveness of the BTB by not storing *fall through* or "not 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. Taken branches are always entered in to the BTB. By not entering fall-through branches, we avoid displacing prediction information for other branches.

Table 4 shows the effect of decoupling the branch prediction information and not storing fall-through branches in the BTB for five different configurations. The left most column uses the BTB with static prediction. The remaining columns use decoupled PHT's, using either normal ('**N**') or aligned ('**A**') branches and either including fall-through branches ('**F**') or excluding fall-through branches ('**No-F**') in the

BTB. The instructions per cycle are shown assuming an instruction misfetch penalty of one and two cycles. Note that the second column, (2048,N,F), often performs worse than the first column (Static Pred). The static prediction method uses the BTB with a private PAs(6,16) prediction register for each branch. The remaining methods simulated used a degenerate global history register, which is 1% less accurate than the PAs(6,16) method.

The misprediction rate for the degenerate PHT can be improved using branch alignment, as shown by comparing the second and third columns, (2048, N, F) and (2048, A, F). This results in a $\approx$ 10% improvement over the BTB with static prediction. We can further improve on this configuration by excluding the fall-through branches from the BTB; column four shows the results when we no longer update the BTB for fall-through branches. This has a sizable improvement on programs with a lot of unconditional branches, indirect jumps, procedure calls and returns. As noted earlier, the SPEC92 programs have fewer procedure calls than the other programs we measured. Thus, this improvement probably would not be apparent unless programs other than those in the SPEC92 suite were simulated.

Table 4 shows that `eqntott` and `rtsh` perform better with a coupled BTB and static prediction; every decoupled mechanism using a 2048-entry PHT has worse performance. Again, this is due to increased PHT conflicts, especially in eqntott. If we increase the PHT to 4096 entries, shown by the (4096, A, no F) column, the performance surpasses the coupled implementation. Since we no longer store the 32-bit fall-through address in the BTB, we may well afford doubling the prediction history table, since it gives a 14% improvement over the basic BTB design with static prediction.

6.3 Instruction Fetch Prediction

The previous designs all used BTB-based designs. There are three reasons for using a BTB:

- By virtue of an instructions address being in the BTB, we know the instruction is a branch.
- Accurate prediction information can be associated with each BTB entry.
- 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.

In the previous section, we have shown that branch alignment and decoupled prediction performs better than the coupled prediction information from a BTB. Comparing Tables 3 and 4 shows decoupled prediction is no worse than the BTB even when profiling or branch alignment is not used. The remaining attributes of BTB's reduces instruction misfetches. Most architectures compute a branch fall-through address in parallel with instruction cache lookup. We believe that a fast encoding, such as the explicit displacement encoding we proposed, will allow the 'taken' destination address to be computed using the instruction from the instruction cache. However, before using the lower bits of the instruction to compute the next fetch address, we must be reasonably certain the instruction is a branch, or we may fetch incorrect addresses for the non-branch instructions that constitute 80% of most programs. If we could predict the instruction type, we could completely eliminate the BTB.

As previously described, we considered three mechanisms for this, with the performance shown in Table 5. The first column shows the performance of the best BTB-based method, (4096, A, No F), using a large PHT and decoupled prediction. The second column uses only an *Instruction Type Prediction Table* (ITPT); a fetched addressed is used to index the ITPT which contains two bits (R & C) predicting the instruction type. In this model, the ITPT is updated after each instruction is decoded. The third column assumes we can use part of the instruction opcode to determine whether instructions are branches. This

Program	4096, A, No F		No Insn Info		1Bit Insn Info		2Bit Insn Info	
	One	Two	One	Two	One	Two	One	Two
compress	2.98	2.98	2.11	1.63	2.98	2.98	2.98	2.98
eqntott	3.28	3.28	2.08	1.54	3.20	3.20	3.20	3.20
espresso	3.65	3.64	2.53	1.93	3.66	3.66	3.66	3.66
gcc	2.88	2.81	2.17	1.65	3.22	3.19	3.24	3.24
li	3.69	3.69	2.15	1.50	3.77	3.75	3.78	3.78
sc	3.58	3.57	2.34	1.68	3.87	3.86	3.87	3.87
ditroff	4.25	4.22	2.78	2.06	4.27	4.25	4.29	4.29
xdvi	4.25	4.23	2.31	1.58	4.28	4.27	4.28	4.28
xfig	3.95	3.89	2.26	1.57	4.03	4.02	4.05	4.05
xtex	4.10	4.06	2.36	1.65	4.15	4.14	4.16	4.16
cfront	2.88	2.78	1.94	1.39	3.19	3.14	3.24	3.24
doc	4.06	3.95	2.47	1.71	4.43	4.42	4.45	4.45
groff	3.35	3.18	2.20	1.53	3.92	3.86	3.97	3.97
idl	4.62	4.60	2.30	1.52	4.72	4.72	4.73	4.73
idraw	3.48	3.26	2.59	1.84	4.36	4.33	4.39	4.39
morpher	3.54	3.39	2.42	1.70	4.17	4.15	4.20	4.20
rtsh	4.07	4.07	2.75	2.08	4.05	4.05	4.06	4.06
Mean	3.62	3.55	2.32	1.66	3.83	3.82	3.85	3.85
C++ Mean	3.64	3.51	2.36	1.66	4.07	4.04	4.10	4.10
Spec Mean	3.31	3.29	2.22	1.65	3.42	3.41	3.42	3.42

Table 5: Instruction Type Prediction Table

removes non-branch instructions from the ITPT, greatly improving the prediction accuracy; however, the ITPT is still consulted to determine the type of the branch. The second column in Table 5 uses a degenerate global ITPT with 8192 2-bit entries and branch alignment. The third column uses 4096 2-bit ITPT entries and branch alignment. The program counter is used as a direct index into the ITPT. The last column assumes we can extract the branch type directly from the instruction encoding; no ITPT is needed. The latter method is completely accurate in predicting the type of the instruction.

Recall that the BTB-based results shown are *the best* of the prior modifications to the BTB design, including the overly-optimistic branch alignment model used in this paper. The degenerate ITPT column, using no information from the instruction, performs poorly compared to the BTB. However, the 1-bit ITPT and the instruction-encoded variants offer a 5.8% and 6.3% improvement over the best BTB design, using significantly less hardware resources. This represents a 21% improvement over the basic BTB design proposed by Yeh *et al* that uses profile-based static prediction in addition to the BTB. Note that `eqntott` and `rtsh` are not improved using the ITPT method. In the ITPT architecture, the only way to predict indirect jumps is by static profiling of the programs. For these programs, static prediction has worse performance than the prediction offered by the BTB. When using a BTB for `eqntott`, 14 indirect jumps are mispredicted, and 285,357 indirect jumps are mispredicted for `rtsh`. Using the ITPT method and static indirect jump prediction, `eqntott` has 320,400 mispredicted indirect jumps while rtsh has 357,446 mispredicted indirect jumps.

Recall that the proposed BTB-based architectures are designed to support wide-issue instruction-level parallelism. We believe the performance of the degenerate ITPT mechanism will improve when issuing multiple instructions, but this is not reflected in the BEP and IPC metrics we used. In the degenerate ITPT method, every instruction is offered to the ITPT to determine the instruction type. For single-issue processors, this causes a large number of conflicts for the ITPT table entries. However, if multiple instructions were fetched concurrently, the *basic block address* would be used to index the ITPT; each basic block would likely contain one and only one branch. Thus, we have reason to suspect the degenerate ITPT performance would be closer to the 1-bit ITPT performance for wide-issue processors; however, we have not empirically verified this.

7 Conclusions

We have shown that instruction fetch and branch prediction architecture proposed in this paper has an average 21% improvement over currently proposed BTB-based designs. Our ITPT-based design, coupled with a return stack and profile-based indirect jump prediction minimizes the instruction misfetch penalty. Decoupling the PHT and using branch alignment greatly increases conditional branch prediction accuracy.

We believe our results apply to the current wide issue processors being developed, where 5-8 instructions are fetched concurrently; the information needed by the ITPT methods 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 could be eliminated if fast carry-adders can be implemented, but the explicit 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 BTB would nearly double when going from 32 to 64 bit addresses.

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] T. Ball and J.R. Larus. Optimally profiling and tracing programs. In *Conference Record of the 19th Annual ACM Symposium on Principles of Programming Languages*, pages 59–70, Albuquerque, NM, January 1992.

[3] Brad Calder and Dirk Grunwald. Branch alignment. Technical report, Univ. of Colorado, 1993. (In Preperation).

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

[5] 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.

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

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

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

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

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

[11] 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.

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

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

[14] 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.

[15] 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.

[16] Tse-Yu Yeh and Yale N. Patt. A comprehensive instruction fetch mechanism for a processor supporting speculative execution. In *19th Annual International Symposium on Microarchitecture*, pages 129–139, Portland, Or, December 1992. ACM.

[17] 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.