Boolean Formula-based Branch Prediction for Future Technologies

Daniel A. Jiménez   Heather L. Hanson   Calvin Lin
Department of Computer Sciences
The University of Texas at Austin
Austin, TX 78712
{djimenez,hhanson,lin}@cs.utexas.edu

Abstract

Accurate branch prediction is essential to sustaining the performance of deeply pipelined wide-issue microarchitectures. However, as clock rates increase and feature sizes decrease, wire delay severely restricts the size of branch prediction tables. In this paper, we present a new method for branch prediction that encodes in the branch instruction a formula, chosen by profiling, that is used to perform history-based branch prediction. Our method replaces the large table of current dynamic branch predictors with a small and fast circuit that interprets the encoded Boolean formula. By using a special class of Boolean formulas, our encoding is extremely concise. Moreover, the prediction accuracy of our method outperforms dynamic schemes. In current technologies, the accuracy of our predictor matches or exceeds that of dynamic schemes. In a projected 70 nm technology and an aggressive clock rate of about 5 GHz, a modest implementation of our method that uses an 8-bit formula encoding has a misprediction rate 60% lower than that of the best gshare *predictor implementable in that technology. Our predictor also consumes much less power than table-based predictors. This paper describes our predictor, our hardware implementation and our profiling algorithm, and presents experimental results using the SPEC 2000 integer benchmarks.*

1 Introduction

As pipelines become deeper and issue widths become wider, the penalty for a mispredicted branch increases. Thus, in the future, accurate branch prediction will become more important to the performance of microprocessors. However, recent studies show that as clock rates increase and feature sizes decrease, wire delay will have an increasingly significant impact on the time to access large structures such as branch prediction tables [1, 12]. For instance, at current 180 nm technology and a moderate clock rate of 1 GHz, an 8K-entry *gshare* predictor with an average accuracy of about 94% can be built. In the future, with 35 nm technology and an aggressive projected clock rate of 9.92 GHz, the largest *gshare* predictor accessible in one cycle will have only 512 entries and an accuracy of about 86%. Thus, predictors must become more accurate to support deeper pipelines, but they must also become smaller because of wire delay and aggressive clocking.

One solution to this problem is to allow the compiler to assist with branch prediction. Existing architectures such as IA-64 allow hint bits in a branch instruction to specify whether to use the dynamic branch predictor or a static prediction, thus filtering the accesses to the dynamic predictor and reducing aliasing (i.e., contention for branch prediction resources). If the static predictions are chosen well, better branch prediction accuracy can be obtained, even with a smaller dynamic branch predictor.

We extend this idea to consider history-based predictors encoded in the branch instruction. In our scheme, a branch instruction encodes a Boolean function, learned through profiling, whose input is the branch history and whose output is a prediction. The key to our solution is a concise encoding of Boolean functions, based on *monotone read-once Boolean formulas*, that is well-suited for branch prediction. For example, Figure 1 shows such a formula as a simple circuit diagram.

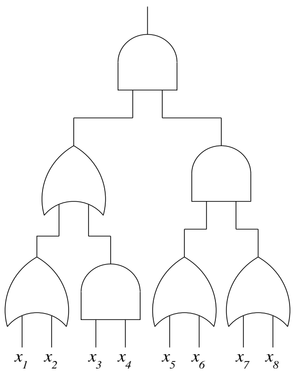

Figure 1: Tree representation of the formula $((x_1 \vee x_2) \vee (x_3 \wedge x_4)) \wedge ((x_5 \vee x_6) \wedge (x_7 \vee x_8))$. Only seven bits, representing the types of the gates, are needed to encode this formula.

For N bits of history, each formula can be encoded using only N bits. We show that using this encoding yields almost the same branch prediction accuracy as the use of arbitrary Boolean formulas, with exponentially fewer bits required for the representation. Decoding and evaluating the Boolean formulas is done quickly using a simple circuit. For instance, with eight bits of history, the formula evaluation circuit is equivalent to 34 NAND gates, at a depth of 9 gates.

At small technologies with aggressive clock speeds, our predictor outperforms purely dynamic schemes. For instance, in a projected 70 nm technology and an aggressive clock rate of about 5 GHz, a modest implementation of our method has a misprediction rate of 5.5%, which is 60% lower than that of the best *gshare* predictor implementable in that technology. Our predictor also uses much less power than table-based methods. For instance, in 70 nm technology, our predictor mentioned above consumes 0.06 mW, while a *gshare* predictor with comparable accuracy consumes 19.7 mW. As another example, a version of our predictor has a 33% lower misprediction rate lower than a 4K-entry *gshare*, while consuming less than 1% of the power of *gshare*.

This paper makes the following contributions:

- We present a new method for branch prediction based on Boolean formulas that mitigates the effects of technology scaling and wire delay. For instance, in one cycle, our predictor can deliver a prediction with the accuracy of a 32K-entry *gshare* predictor in a technology where only a 512-entry *gshare* predictor can be accessed in one cycle.

- We describe the hardware implementation of our predictor, showing that is has less delay and consumes less power than a conventional branch predictor.

- We describe a profiling algorithm for training our predictor.

- We evaluate our method using the SPEC 2000 integer benchmarks, analyze the results, and draw conclusions about why the predictors behave the way they do.

2 Background and Related Work

To provide context for our research, we now review some of the recent work in branch prediction.

2.1 Dynamic Branch Prediction

Recent research on dynamic branch prediction focuses on refining the two-level scheme of Yeh and Patt [26] in which a pattern history table of two-bit saturating counters is indexed by a combination of branch address and global or per-branch history. The high bit of the counter is taken as the prediction. Once the branch outcome is known, the counter is incremented if the branch is taken, and decremented otherwise.

An important problem in two-level predictors is aliasing [21], and many of the recently proposed branch predictors seek to reduce aliasing [18, 16, 23, 10] but do not change the basic prediction mechanism. Jiménez and Lin recently introduced the perceptron predictor [13], which uses a different prediction mechanism. It uses perceptron learning rather than indexing a table of saturating counters. As in the research presented here, this technique allows only a limited number of branch prediction functions to be expressed, but still provides good accuracy.

2.2 Static Branch Prediction

A purely static branch predictor always predicts the same outcome for a particular static branch. The prediction can be derived from the structure of the branch, e.g., the "backwards taken/forwards not taken" approach of the Alpha AXP-21064, or encoded into the branch instruction as a bias bit, as in the IA-64 and HP-PA/RISC instruction sets. The compiler, through profiling or static heuristics [4, 7], can provide hints to the microarchitecture about the likely direction of the branch. Static branch predictors are usually less accurate than dynamic branch predictors because they cannot consider changing conditions at run-time.

Lindsay explores the use of decision trees to encode statically-learned Boolean functions [17]. The decision trees are learned by profiling and are encoded in programmable logic arrays (PLAs). By contrast, our encoding is represented only in the branch instruction, requiring little hardware in the CPU itself. Although Lindsay's thesis addresses latency issues, PLAs representing the behavior of large sets of branch instructions will have the same technology scaling issues in future technologies as large banks of SRAM. Similarly, Fern *et al* [11] study the use of decision trees, grown dynamically, for branch prediction. The trees are kept in a large structure in the CPU and would have the same problems with wire delay as other predictors. Thus, our technique is distinctly well-suited to the issues of technology scaling.

2.3 Compiler-Guided Branch Prediction

Several schemes for employing the compiler in branch prediction have been suggested. August *et al* [3] propose placing in each branch instruction hint bits that tell a dynamic predictor what kind of state to examine to make a prediction. The variable length path branch predictor [24] encodes profiling information in branch instructions. This information guides a dynamic predictor, telling it what history length to use and what hash function of past branch addresses to use to form an index into a table of counters.

Other techniques use the compiler to help with branch prediction without changing the prediction mechanism. For instance, *branch alignment* [8, 27] increases instruction fetch bandwidth by minimizing the number of taken branches in a program. *Static correlated branch prediction* [28] is another optimization that introduces duplicate basic blocks, encoding in the program counter information about the path taken to reach a particular static branch and increasing the accuracy of static prediction.

2.4 Wire Delay and Branch Predictors

As clock rates increase and feature sizes shrink, wire delay increases significantly relative to gate delay [1]. As this trend continues, the chip area reachable in a single cycle will decrease. This means that large banks of SRAM, such as caches and branch prediction tables, will have to either decrease in size or increase in delay.

Jiménez *at al* [12] show that a branch predictor must return a prediction in a single cycle, because a highly accurate two-cycle branch predictor yields much lower instruction

Minimum Feature Size (nm)	Predicted Clock Rate (GHz)	Largest *gshare* Table Accessible in One Cycle (# entries)
180	1.92	1024
130	2.67	1024
100	3.47	1024
70	4.96	1024
50	6.94	1024
35	9.92	512

Table 1: Effects of technology scaling on branch predictor size. With an aggressive clock rate, the size of a *gshare* predictor accessible in one clock cycle must decrease as technology moves forward.

throughput than a relatively inaccurate single-cycle predictor. The same study shows that with aggressive clocking, the number of two-bit counters reachable in a single cycle will drop to 1K in 180 nm technology, and down to 512 in the 35 nm technology that is projected to be available in 2012. The study also suggests several mechanisms to mitigate the delay by adding extra hardware. For instance, read access to the branch predictor can be pipelined. Here, our focus is different, as we propose to use *much less* hardware to increase accuracy, in exchange for some extra profiling effort and changes to the instruction set architecture (ISA).

Table 1 shows the maximum size of a *gshare*-like predictor as technology moves forward. These results of ECacti [1, 20] simulations use an aggressive clock rate equivalent to eight times the gate delay of propagating a value from a single inverter to four copies of itself. This "eight fan-outs-of-four" measure was used as the aggressive clock speed for the study by Agarwal *et al* [1], giving a technology-independent projection of future clock rates. Note that these capacities only consider the time to read the branch prediction table. The gate delay involved in acting upon a branch prediction is not included and further exacerbates the problem.

2.5 Combining Static and Dynamic Branch Prediction

Branch prediction accuracy can be increased by combining static and dynamic branch prediction. Some of the branches can be predicted with a static bias bit, while others with less biased behavior can use the dynamic predictor. Since the easily predictable branches are filtered out, aliasing in the dynamic predictor is alleviated and accuracy is improved. We use this technique to evaluate our branch predictor. This technique, along with a methodology for choosing the bias bits, was introduced by Chang *et al* as *branch classification* [9] and refined by Patil and Emer [19].

3 Branch Prediction with Boolean Formulas

3.1 Boolean Formulas as Branch Predictors

History-based branch prediction can be viewed as the problem of learning the Boolean function of the branch history that gives the best prediction. Let $\mathbf{h}$ be a Boolean N-vector containing the outcomes of the last N branches executed. For now, we can think of this branch history as being either global or per-branch. For a static branch B, there exists a Boolean function $f_B(\mathbf{h})$ that best predicts whether B will be taken given the history $\mathbf{h}$. The goal of dynamic branch predictors is to learn this function as quickly as possible to provide accurate prediction [13].

One approach to branch prediction is to learn $f_B(\mathbf{h})$ for each branch in a profiling run, then somehow encode each $f_B(\mathbf{h})$ in the branch instruction and have the hardware use the dynamic history to compute the function and provide a branch prediction. Statically chosen bias bits, such as those available on HP-PA and IA-64, encode a constant Boolean function requiring no history information.

If the behavior of branches is stable across different program inputs, then we would expect branch prediction using these functions to perform very well, even better than dynamic branch predictors, which have the disadvantages of destructive aliasing and training time. In practice, input-dependent behavior, such as loop trip counts that vary from run to run, limits the accuracy of a Boolean function predictor. But as we will see, these functions still provide highly accurate predictions.

One problem with this approach is that of representing a Boolean function within a branch instruction. For instance, with a moderate history length of 10, there are $2^{2^{10}}$ different Boolean functions. To allow encoding all of these functions would require branch instructions over 1000 bits long. Therefore, we consider an extremely compact, but sufficiently expressive, encoding of Boolean formulas.

3.2 Read-Once Monotone Boolean Formulas

We now describe an expressive subset of Boolean formulas that can be compactly represented. The basic idea is to restrict the Boolean formulas such that each variable appears in the formula only once, and the only operations allowed are AND and OR.

Let $\mathbf{x}, \mathbf{y} \in \{0, 1\}^N$, i.e., $\mathbf{x}$ and $\mathbf{y}$ are N-bit vectors of Boolean values. We say that $\mathbf{x} \leq \mathbf{y}$ if, for all i, $\mathbf{x}_i \leq \mathbf{y}_i$. Consider a Boolean function $f\{0, 1\}^N \mapsto \{0, 1\}$, i.e., a function f mapping a vector of N bits to a single bit. We say that f is *monotone* if $\mathbf{x} \leq \mathbf{y}$ implies $f(\mathbf{x}) \leq f(\mathbf{y})$ [15]. A *monotone Boolean formula* is a Boolean formula that uses only AND ($\wedge$) and OR ($\vee$), without NOT, as connectives. It can be shown that the functions induced by these formulas are monotone [15], hence the name.

A *read-once formula* is a formula in which each variable appears exactly once in the formula. Read-once formulas are also known as μ-formulas or Boolean trees [2]. Read-once monotone Boolean formulas have a concise description as a tree whose internal nodes are ANDs and ORs and whose leaves are the Boolean variables. As an example, Figure 1 from the introduction shows the tree representation of the formula $((x_1 \vee x_2) \vee (x_3 \wedge x_4)) \wedge ((x_5 \vee x_6) \wedge (x_7 \vee x_8))$ as a logic diagram.

3.3 Using Monotone Read-Once Formulas for Branch Prediction

A read-once monotone Boolean formula of N variables can be encoded as a bit vector of size $N - 1$, each bit representing a connective in the Boolean tree, with 0 for AND and 1 for OR. Thus, each branch instruction encodes a read-once monotone Boolean formula using $N - 1$ bits. We also store another bit that, if set to 1, causes the value of the function to be inverted, so that we can also represent the complements of monotone read-once formulas. No two different bit patterns represent the same Boolean function, so this encoding is quite efficient. For a history length of N, the formula encoding in the branch instruction takes N bits. Monotone Boolean formulas are incapable of representing Boolean constants, so we allow the formula whose connectives are all ANDs to compute 0 (i.e. *false*). By choosing to invert the output, this formula can also produce 1 (i.e. *true*). These two values are necessary, since they allow us to represent "always predict taken" or "always predict not taken."

For branch prediction, we keep a branch history shift register into which the Boolean outcomes (i.e., 1 for *taken* and 0 for *not taken*) of branches are shifted. We keep a global history, rather than keeping a history for each static branch, but the concept would work either way.

When a branch instruction is fetched, the Boolean formula is sent, along with the contents of the history register, to a circuit that decodes the formula and computes the prediction.

We use a profiling phase to decide which formulas to encode in each branch instruction. The profiling algorithm uses statistics about the behavior of each static branch to choose the best monotone read-once formula for that branch.

3.4 Impact of Encoding

Since each branch instruction encodes a Boolean formula, we must find an efficient way to encode the formula in the instruction without having a negative impact on performance. Some instructions sets already provide extra bits for communicating hints to the microarchitecture. For instance, the Alpha AXP ISA provides 14 bits in each indirect branch instruction for profiling information [22]. In their work on variable length path branch predictor, Stark *et al* [24] use extra bits such as these to communicate to the microarchitecture information on hash functions for a branch predictor.

For architectures in which branch instructions do not set aside enough bits to represent our Boolean formulas, we describe two options in the following subsections.

3.4.1 Changing Branch Instructions

We can change the ISA so that branch instructions encode the formulas. For example, each branch instruction on the Alpha is 32 bits long: six bits indicate the op code of the instruction, five more bits indicate the register to test, and 21 bits are for the branch offset. Our experiments show that only 1.08% of dynamic branches in the SPEC 2000 integer benchmarks require an offset of 17 or more bits. Thus, with a negligible impact on performance, we could redesign the conditional branch instructions to encode a 4-bit formula, and allow those few extra long branches to jump to an unconditional branch instruction whose target is the original branch target. This solution is somewhat unsatisfying, because it allows only small formulas and histories to be represented and, as we will see in Section 4, the accuracy of our predictor increases with history length.

3.4.2 Adding Formula Registers

We can represent the formulas outside of the branch instruction. A naive approach is to have a hint instruction before each branch that gives the formula for that branch, increasing code size and stressing fetch bandwidth. Alternately, we can build a small file of perhaps eight registers, each capable of representing a single formula, and we can provide instructions for loading these registers. Each branch instruction will use three bits to refer to the register containing the formula it needs. With compiler analysis, many redundant formula-loading instructions could be removed. For instance, the compiler can move most of the formula-loading instructions out of loops. It can also minimize the number of times a formula register has to be loaded by doing register coloring and sharing registers for branches that use the same formula. As we will see in Section 4.4, a small number of the Boolean formulas account for a great majority of the branch instructions, so this sharing will greatly reduce the cost of this scheme.

3.5 Profiling Algorithm

We now describe our algorithm for determining which formulas best predict each static branch. Using a trace of each branch address and outcome, we simulate the dynamic contents of the history register. For each static branch, we keep a list of the different histories that lead up to that branch, and the number of times each history led to the branch being taken or not taken. After every dynamic branch has been examined, we check the list for each static branch B and exhaustively test every monotone Boolean formula and its complement to see which one would have yielded the fewest mispredictions given all the histories that led up to B. This best formula is then encoded into the branch instruction.

For branches that are executed fewer than 500 times in the profiled program, we simply use the constant formula (0 or 1) that best predicts that branch, rather than considering all 2^N formulas. We are investigating ways to speed up the algorithm with a more intelligent search. Section 4.5 gives timing results for the profiling algorithm.

3.6 Hardware Implementation

A hardware implementation of a Boolean formula branch predictor is simple. Each Boolean connective (i.e., AND or OR) in the formula is represented by a circuit with three inputs: two data inputs, corresponding to the variables or outputs of other gates, and one control input that specifies whether the Boolean connective should compute AND or OR. Figure 2 shows a logic diagram for this four-NAND circuit. With a history length of N, our predictor is built from $N-1$ connectives and a single XOR gate at the output that acts as an inverter when its input is 1. Figure 3 shows a circuit implementation of the predictor for $N = 8$. For clarity, the extra logic to produce 0 when all the connectives are ANDs is not shown, since this logic requires relatively few gates and is not on the critical timing path.

We simulate a straightforward static CMOS implementation of the Boolean formula predictor with the HSPICE circuit simulator. First, we create a sub-circuit composed of four NAND gates as shown in Figure 2. Then, we instantiate $2 \log N$ of these subcircuits and add an XOR, which is a subcircuit consisting of two inverters and two NAND gates. The connections between the subcircuits are shown in Figure 3. Finally, we add capacitance between the gates to model local interconnect.

Note that although the concept of a read-once monotone Boolean formula is somewhat similar to the actual implementation as a circuit, to avoid confusion, the two should be thought of separately as function vs. implementation. For instance, the circuit is optimized for static CMOS technology with NAND gates and is not a read-once circuit.

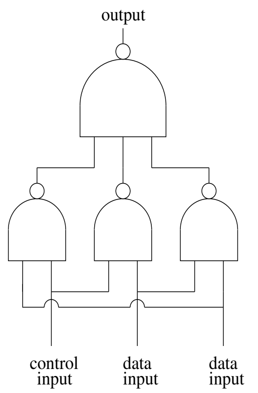

Figure 2: Boolean connective subcircuit. If the control input is 0, then the output is the AND of the two data inputs. Otherwise, the output is the OR of the two data inputs.

3.7 Delay

The depth of the formula evaluation circuit with N inputs is $2 \log N$ plus the final XOR gate. For instance, for $N = 16$, the critical delay path passes through eight NAND gates and one XOR gate. In contrast, the *gshare* predictor looks up values from a table by reading from an SRAM array.

We use circuit simulations and ECacti [20], a tool for cache delay calculation, to determine predictor access times for a range of current and future integrated circuit generations. We estimate the access time of the Boolean formula predictor by simulating the combinational circuit and measuring the delay from the branch instruction and history register inputs to the output of the XOR gate. The delay measurements are the time from the midpoint of the input signal switching to the midpoint of the output signal switching; these delays are shown in Table 2 for $N = 8$ and $N = 16$ inputs. We calculated the lookup time for a *gshare* predictor using the ECacti tool. Table 2 shows the access times for a 4K-entry *gshare* predictor and two sizes of the Boolean formula predictor, $N = 8$ and $N = 16$, for a range of fabrication technologies. We chose the 4K-entry predictor because, as we will see in Section 4, the $N = 8$ version of the Boolean formula predictor only slightly exceeds the accuracy of a 4K-entry *gshare*. Thus, our delay comparisons show that we can achieve higher accuracy with lower latency.

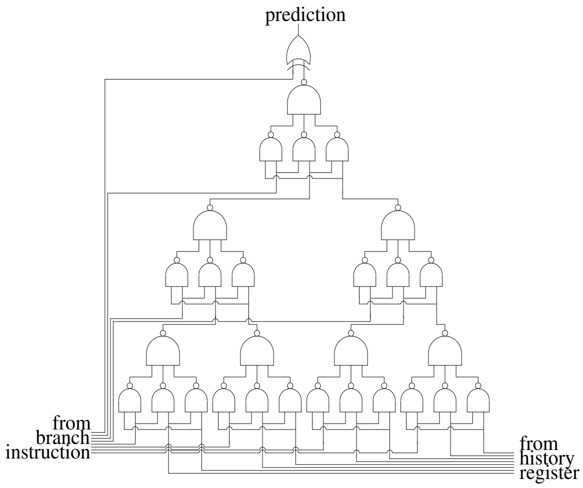

Figure 3: Boolean formula branch predictor circuit. This circuit makes a branch prediction based on a history length of 8 and an 8-bit encoding of a read-once Boolean formula.

As fabrication technology improves, transistors can be made smaller and faster, resulting in higher clock frequencies and faster combinational circuits. As Table 2 shows, access times for each structure improve as the minimum feature size decreases.

The Boolean formula predictor is consistently faster than the 4K-entry *gshare* predictor, allowing more time for communication and computation within a clock cycle. At the projected clock rate of 6.94 Ghz for 50 nm technology from Table 1, the clock period would be 144 picoseconds. A traditional table-lookup predictor such as *gshare* would require most of a cycle—134 picoseconds in this case—for the prediction. In the same technology, the Boolean formula predictor would provide a prediction in 59 picoseconds, leaving over half of the cycle to act upon the prediction.

3.8 Power

Power consumption has become a primary concern in microprocessor design. In this section, we contrast the power consumption of traditional branch predictors with that of the Boolean function predictor.

The Boolean function predictor is a simple combinational circuit, which uses less dynamic power than an SRAM-based

Minimum Feature Size (nm)	Access Time (picoseconds)		
	4K-entry *gshare*	Formula, $N = 8$	Formula, $N = 16$
180	606	211	260
130	437	168	208
100	359	112	138
70	247	85	103
50	134	50	59

Table 2: Access times for a 4K-entry *gshare* predictor vs. the Boolean formula predictor. The delays were obtained using an HSPICE model for the Boolean formula predictor and ECacti for *gshare*.

Minimum Feature Size (nm)	Power (milliwatts)		
	4K-entry *gshare*	Formula, $N = 8$	Formula, $N = 16$
180	44.3	0.61	1.28
130	36.9	0.28	0.58
100	31.3	0.11	0.24
70	19.7	0.06	0.12
50	14.0	0.06	0.13

Table 3: Dynamic power consumption for the Boolean formula predictor and a 4K-entry *gshare*. These figures were obtained from HSPICE for the Boolean formula predictor and the Wattch tool for *gshare*.

predictor. This small predictor has smaller gate and interconnect capacitance than an SRAM structure, which has decoding logic, a memory array, sensing logic, and output logic.

Table 3 shows the Boolean function predictor's dynamic power consumption for $N = 8$ and $N = 16$, as measured with the HSPICE simulator. This table also shows the power of a 4K-entry *gshare* predictor, as measured with the Wattch simulator [5]. These measurements show that the Boolean formula predictor consumes between 0.4% to 2.9% of the power of a *gshare* predictor with comparable accuracy.

With lower transistor threshold voltages in emerging technologies, static power—due to leakage current through transistors—is becoming a sizable percentage of the total power consumed [25]. With fewer transistors in the circuit to leak current, the Boolean predictor circuit will also have less static power than an SRAM structure. Furthermore, the Boolean circuit implementation is amenable to a low static power design technique that takes advantage of the stacked transistors within gates to bias transistors into a low-leakage mode [25].

4 Experimental Results

In this section, we give the results of simulating our branch predictor on the SPEC 2000 integer benchmarks, and we compare our results against both static (i.e. bias bit) and dynamic branch prediction. We also give results for a predictor that combines Boolean formulas with dynamic prediction, and we compare this to similar work that combines static and dynamic prediction.

Benchmark Name	# Instructions Executed	# Static Branches
164.gzip	2,159,450,356	773
175.vpr	1,678,973,402	2,458
176.gcc	2,685,859,608	31,314
181.mcf	1,206,663,723	893
186.crafty	1,779,552,658	3,911
197.parser	1,151,438,421	4,207
252.eon	1,807,574,168	3,693
253.perlbmk	1,363,745,030	9,002
254.gap	3,014,596,275	3,326
255.vortex	1,255,446,737	7,650
256.bzip2	1,653,885,931	910
300.twolf	1,428,388,788	2,818

Table 4: Benchmarks simulated. This table shows the benchmarks used for this study, along with the number of instructions profiled and the number of static branches in each benchmark.

4.1 Methodology

We use the 12 SPEC 2000 integer benchmarks running under SimpleScalar/Alpha [6] to collect traces. For each benchmark, we gather traces giving the branch address and outcome for 200 million branches. Table 4 shows the benchmarks we use, as well as the number of instructions executed to reach 200 million branches, and the number of static branches that are executed at least once. Each benchmark executes over one billion instructions before the simulation ends.

We use the `train` inputs for the profiling runs, and we use the `ref` inputs to evaluate the accuracy of the various predictors. To estimate the delay to access branch prediction structures, we use a version of the ECacti cache simulator [20] that was modified to give delay estimates across a wide range of technologies [1, 12]

4.2 Predictors Simulated

We simulate monotone read-once Boolean formula predictors with $2 \leq N \leq 18$. We use only global history information, i.e., we do not use path or per-branch information. We also simulate the *gshare* [18], bi-mode [16] and agree [23] branch predictors, three global dynamic branch predictors from the literature well-known for their high accuracies. The *gshare* and bi-mode predictors use only dynamic history information. The agree predictor combines static and dynamic information by predicting whether a branch will agree with a bias bit.

History length has been observed to have a significant impact on predictor accuracy [18], so at each hardware budget, we try all possible history lengths on the `train` inputs and keep the one with the lowest average misprediction accuracy.

As a limit study, we also measure the results of using arbitrary Boolean formulas to give a lower-bound on misprediction rates for any Boolean-formula based predictor. Note that although it is not the focus of our research, this arbitrary formula predictor is actually implementable for history lengths of up to four, since the truth table for a Boolean function in four variables can be encoded in 16 bits, and we have argued that this much information could be used by our Boolean formula predictor.

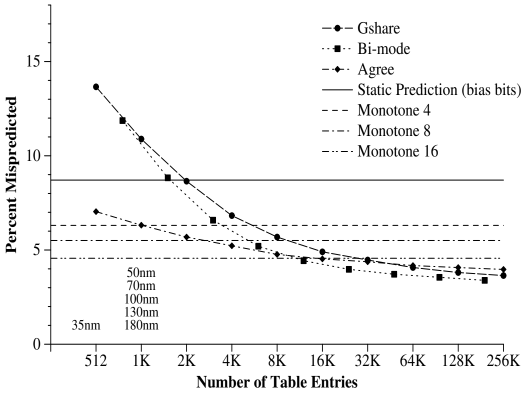

Figure 4: Accuracy of dynamic branch predictors vs. static prediction and the Boolean formula predictor. The numbers above the x-axis show the technologies in which the corresponding hardware budgets are reachable in one cycle with aggressive clocking. Misprediction rates are the harmonic means over the SPEC 2000 integer benchmarks.

4.3 Misprediction Rates

Figure 4 shows misprediction rates for the monotone read-once Boolean formula predictor at history lengths of 4, 8 and 16, compared with *gshare*, bi-mode and agree predictors at hardware budgets from 512 to 256K entries. Labels above the 512 and 1K-entry hardware budgets show the process technologies for which the corresponding budget is reachable in one cycle at the aggressive clock rates listed in Table 1. Even ignoring delay, our predictor remains competitive with dynamic methods that have up to 16K or 32K table entries, which are larger than current branch predictors, such as the hybrid predictor in the Alpha 21264 [14]. Thus, our predictor is a good idea regardless of technology trends, but is especially good for future technologies and aggressive clocks.

At today's 180 nm and 130 nm technologies, for which branch predictors with about 1K to 2K table entries state are available at aggressive clock speeds, the Boolean formula predictor, with a history length of 4 and misprediction rate of 6.3%, roughly matches the accuracy of the agree predictor. With a history length of 16, the Boolean formula predictor has misprediction rate of 4.56%, an improvement of 31% over the 3072-entry bi-mode predictors and 47% over the 2048-entry *gshare*, and 20% over the 2048-entry agree predictor.

At small hardware budgets, both static prediction and Boolean formula prediction perform better than the purely dynamic bi-mode and *gshare* predictors. At the smallest technology of 35 nm, the largest predictor reachable in a single cycle has only 512 entries. At this budget, with a short history of 4, the misprediction rate of the Boolean formula predictor is 28% lower than that of static prediction amd 10% lower than that of the agree predictor. With a modest history length of 8, the monotone read-once Boolean formula predictor has a misprediction rate of 5.5%, which is 60% lower than that of *gshare*, 37% lower than that of static prediction with profiled bias bits, and 22% lower than that of the agree predictor. With a long history of 16, the misprediction rate is 48% lower than that of static prediction, and 35% lower than that of the agree predictor.

To put these figures another way, the Boolean formula predictor with a history length of 8 achieves roughly the same predictive power as an 8K-entry *gshare* predictor, a 6K-entry bi-mode predictor, or a 4K-entry agree predictor. With a history length of 16, the Boolean formula predictor is about as accurate as a 12K-entry bi-mode predictor, a 32K-entry *gshare* predictor, or a 16K-entry agree predictor.

Figure 5 shows misprediction rates for the monotone read-once Boolean formula predictor, as well as for the predictor that uses arbitrary formulas, at history lengths ranging from 2 to 18. For reference, it also shows the misprediction rates for pure static prediction with bias bits, as well as for dynamic prediction with a 1K entry *gshare*, a 1K entry agree predictor, and a 1.5K entry bi-mode predictors; these sizes represent the predictors accessible in a single cycle in 50 through 130 nm technology with aggressive clock rates. As history length increases, the misprediction rate of the Boolean formula predictor decreases and remains close to the performance of the arbitrary formula predictor.

For the same five predictors, Figure 6 shows misprediction rates on each benchmark. The Boolean formula predictor usually has a misprediction rate far lower than that of the dynamic predictors. However, in a few cases, such as `256.bzip2`, the misprediction rate is high, most likely due to input-dependent program behavior that cannot be learned by profiling.

Figure 7 compares static Boolean formula predictors against a hybrid *gshare*/monotone formula predictor. The purely static predictors perform poorly on `256.bzip2`, giving a misprediction rate of about 23%. We see that the arbitrary formulas give the same poor accuracy as the monotone formula predictor, so the problem is not due to the compact encoding. The misprediction rate for `256.bzip2` goes down to 7.5% when the Boolean formula predictor is alloyed with a dynamic predictor.

Figure 8 shows the misprediction rates of hybrid predictors that combine static hints and dynamic prediction using *branch classification* [9], which statically chooses either a dynamic predictor or a static predictor. A static branch uses the static hint if the misprediction rate of *gshare* on that branch is worse than the misprediction rate of the static hint predictor on that branch; otherwise *gshare* is used. Thus, the dynamic predictor is only used on branches for which it is more appropriate, which reduces aliasing by filtering out the other branches. For comparison, results for *gshare* and bi-mode by themselves, without any static hints, are also shown,

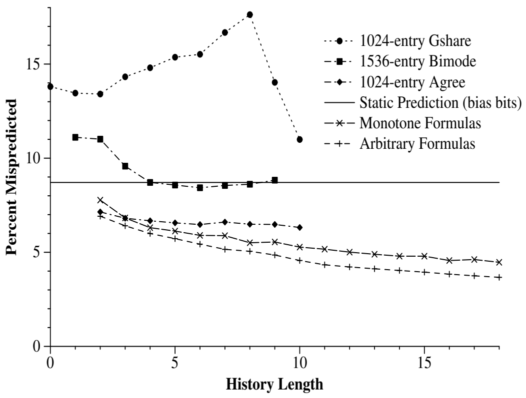

Figure 5: Misprediction rate for the Boolean formula predictor as a function of history length. For comparison, misprediction rates of bias bits and arbitrary Boolean formula predictors are shown, along with *gshare*, agree and bi-mode in deep sub-micron technologies with aggressive clock rates; these curves stop as the maximum history length for each predictor is reached. Misprediction rates are the harmonic means over the SPEC 2000 integer benchmarks.

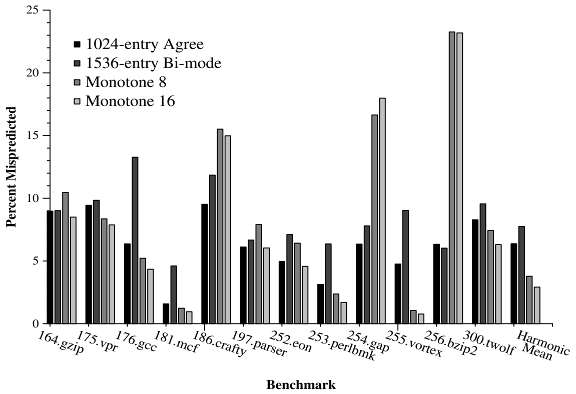

Figure 6: Accuracy of the predictors on each benchmark. This graph compares the Boolean formula predictor at history lengths of 8 and 16 against future-technology implementations of *gshare* and bi-mode.

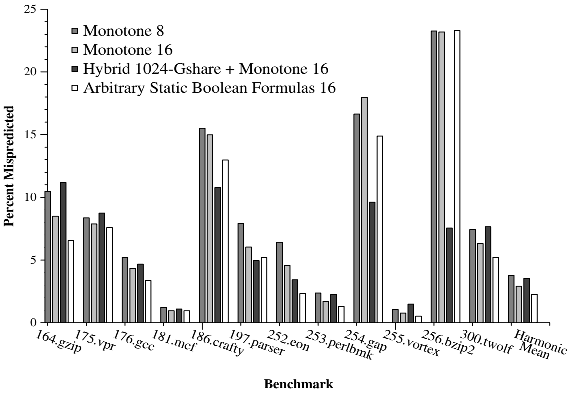

Figure 7: Accuracy of Boolean formula predictors on each benchmark. This graph compares the monotone Boolean formula predictors against a hybrid monotone/gshare predictor and an arbitrary Boolean formula predictor.

as well as results for the agree predictor with bias bits (these are the same curves shown in Figure 4).

The combination of dynamic and static prediction gives us the largest jump in accuracy and is consistent with the results given by Patil *et al* [19]. At a hardware budget of 512 two-bit entries, representing 35, 50, and 70 nm technologies with aggressive clocking, combining static prediction with dynamic prediction decreases the misprediction rate of *gshare* by 59%, from 13.66% to 5.57%. The use of the Boolean formula predictor decreases the misprediction rate of this hybrid static-dynamic predictor by another 13%, down to 4.85%, at a history length of 8, and by 24%, down to 4.24%, at a history length of 18. The advantage of the hybrid dynamic/Boolean formula predictor diminishes as the hardware budget increases.

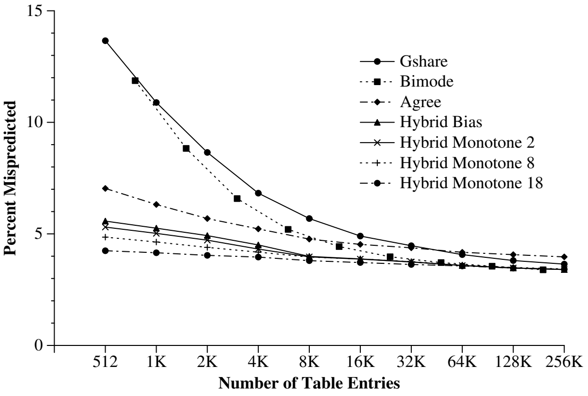

Figure 8: Accuracies of hybrid dynamic/Boolean formula predictors. Misprediction rates are harmonic means over SPEC 2000.

Formula	% Static Frequency	% Dynamic Frequency
1	65.46	56.58
0	29.96	27.88
$\neg(x_0 \wedge x_1 \wedge (x_2 \vee x_3))$	0.63	3.43
$(x_0 \vee x_1) \wedge x_2 \wedge x_3$	0.64	1.83
$x_0 \wedge x_1 \wedge (x_2 \vee x_3)$	0.44	1.72
$\neg((x_0 \vee x_1) \wedge x_2 \wedge x_3)$	0.57	1.69
$(x_0 \vee x_1) \wedge (x_2 \vee x_3)$	0.58	1.57
$\neg((x_0 \vee x_1) \wedge (x_2 \vee x_3))$	0.53	1.55
$(x_0 \wedge x_1) \vee (x_2 \wedge x_3)$	0.17	1.07
$\neg(x_0 \vee x_1 \vee (x_2 \wedge x_3))$	0.23	0.79
$x_0 \wedge x_1 \wedge x_2 \wedge x_3$	0.11	0.59
$(x_0 \wedge x_1) \vee x_2 \vee x_3$	0.13	0.33
$\neg((x_0 \wedge x_1) \vee x_2 \vee x_3)$	0.14	0.31
$\neg(x_0 \wedge x_1 \wedge x_2 \wedge x_3)$	0.08	0.26
$\neg((x_0 \wedge x_1) \vee (x_2 \wedge x_3))$	0.18	0.24
$x_0 \vee x_1 \vee (x_2 \wedge x_3)$	0.14	0.17

Table 5: Distribution of Boolean formulas with a history length of four. The variables are elements of the history register, with x_0 being the outcome of the most recently executed branch, x_1 being the next recent, etc.

4.4 Distribution of Formulas

An analysis of the distribution of Boolean formulas chosen by the profiling algorithm shows that most of the Boolean formulas chosen are the two constant functions, 0 and 1. This dependence on constant formulas decreases as history length increases. For instance, with a history length of 4, 84.5% of static branches in the SPEC 2000 integer benchmarks are best predicted with a constant formula, as opposed to 65.6% for a history length of 16. As history length is increases, the predictive power of the Boolean formula predictor increases, and the constant functions representing "predict taken always" and "predict not taken always" give way to more intelligent choices.

Table 5 shows the dynamic and static frequencies for each formula with a history length of four. For brevity, we omit similar tables for the other history lengths. This table shows that most branches use a constant formula, and the rest of the formulas are chosen according to a non-uniform distribution.

With a history length of 8, the non-constant formula with the highest dynamic frequency is the following one:

$$(x_0 \vee x_1) \wedge x_2 \wedge x_3 \wedge (x_4 \vee x_5 \vee x_6 \vee x_7),$$

accounting for 1.0% of all branches executed. This corresponds to a branch prediction policy of "predict taken if either of the last two branches were taken *and* the third and fourth most recent branches were both taken, *and* any of the other branches in the history were taken."

4.5 Profiling Cost

The cost of determining the best Boolean formula for each branch is an important component of the cost of our branch predictor. Here, we quantify this cost.

Figure 9 shows the amount of time taken for profiling, as a function of history length. The graph shows the arithmetic mean of the time, in seconds, that our algorithm spent for each benchmark. These times were collected by running our program on our network of 733MHz Pentium III computers.

Our implementation takes time exponential in the history length. However, for the small history lengths we consider in this study, the time is not unreasonable. For instance, with a history length of 16, the profiling algorithm takes about 12 minutes. For a history length of 10, the program takes about 2 minutes. For history lengths less than about 12, the time for the program is dominated by activities unrelated to finding the best Boolean function. For instance, much time is spent simply reading the large trace file from the disk and doing other processes that many feedback-directed optimizations would require. Our algorithm is also easy to parallelize. The time-consuming part of the algorithm during which the best Boolean formula is decided for each static branch is embarrassingly parallel. The static branches can be partitioned among many processors. Thus, we feel that our profiling algorithm would be appropriate in a framework in which other optimizations are also being explored by simulation.

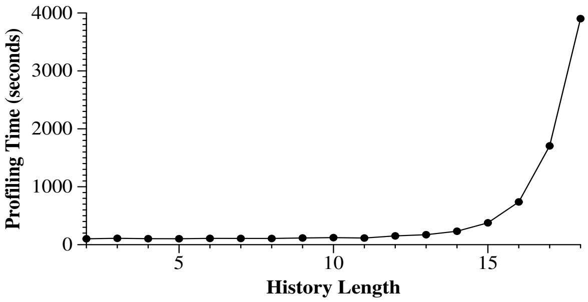

Figure 9: Average, over all benchmarks, of the amount of time spent profiling. The time is reported as a function of history length.

5 Conclusions and Future Work

Our predictor borrows the concept of a read-once monotone Boolean formula from complexity theory to provide a compact encoding for a class of functions that is expressive enough to perform branch prediction and simple enough that an implementation is faster than a traditional branch predictor. By offloading most of the prediction work to the compiler, our Boolean formula branch predictor is small, fast and consumes little power, yet delivers accuracies exceeding those of current large predictors. Even in today's technology, our predictor provides a competitive alternative to dynamic branch prediction. At the low hardware budgets projected for future microprocessors, the benefit of our scheme increases.

We are currently studying ways to improve the training algorithm so that it takes less time at longer history lengths. For instance, we are exploring genetic algorithms as a way to get a near-optimal choice of formula at a fraction of the time the original algorithm takes. We are also looking at ways the compiler can help place branch prediction hints such that they have little to no impact on instruction fetch bandwidth.

6 Acknowledgments

We thank Steve Keckler and Samuel Guyer for their helpful discussions and comments on this paper. Calvin Lin is supported by an NSF CAREER Grant ACI-9984660 and ONR grant N00014-99-1-0402. Heather Hanson is supported by an Intel fellowship.

References

[1] V. Agarwal, M. S. Hrishikesh, S. W. Keckler, and D. Burger. Clock rate versus IPC: The end of the road for conventional microarchitectures. In *The 27th Annual International Symposium on Computer Architecture*, pages 248–259, June 2000.

[2] D. Angluin, L. Hellerstein, and M. Karpinski. Learning read-once formulas with queries. *Journal of the Association for Computing Machinery*, 40(1):185–210, January 1993.

[3] D. I. August, D.A. Connors, J. C. Gyllenhaal, and W. M. Hwu. Architectural support for compiler-synthesized dynamic branch prediction strategies: Rationale and initial results. In *Proceedings of the Third International Symposium on High-Performance Computer Architecture*, February 1997.

[4] T. Ball and J. Larus. Branch prediction for free. In *Proceedings of the SIGPLAN '93 Conference on Programming Language Design and Implementation*, pages 300–313, June 1993.

[5] D. Brooks, V. Tiwari, and M. Martonosi. Wattch: A framework for architectural-level power analysis and optimizations. In *Proceedings of the 27th International Symposium on Computer Architecture*, Vancouver, British Columbia, June 2000.

[6] D. Burger and T. M. Austin. The simplescalar tool set version 2.0. Technical Report 1342, Computer Sciences Department, University of Wisconsin, June 1997.

[7] B. Calder, D. Grunwald, M. Jones, D. Lindsay, J. Martin, M. Mozer, and B. Zorn. Evidence-based static branch prediction using machine learning. *ACM Transactions on Programming Languages and Systems*, 19(1), 1997.

[8] B. Calder and D. Grunwald. Reducing branch costs via branch alignment. In *Proceedings of the Sixth International Conference on Architectural Support for Programming Languages and Operating Systems*, October 1994.

[9] P.-Y. Chang, E. Hao, T.-Y. Yeh, and Y. N. Patt. Branch classification: a new mechanism for improving branch predictor performance. In *Proceedings of the 27th International Symposium on Microarchitecture*, November 1994.

[10] A. N. Eden and T. N. Mudge. The YAGS branch prediction scheme. In *Proceedings of the 31st Annual ACM/IEEE International Symposium on Microarchitecture*, November 1998.

[11] A. Fern, R. Givan, B. Falsafi, and T. N. Vijaykumar. Dynamic feature selection for hardware prediction. Technical Report TR-ECE 00-12, School of Electrical and Computer Engineering, Purdue University, 2000.

[12] D. A. Jiménez, S. W. Keckler, and C. Lin. The impact of delay on the design of branch predictors. In *Proceedings of the 33^{rd} Annual International Symposium on Microarchitecture*, December 2000.

[13] D. A. Jiménez and C. Lin. Dynamic branch prediction with perceptrons. In *Proceedings of the Seventh International Symposium on High Performance Computer Architecture*, January 2001.

[14] R.E. Kessler. The Alpha 21264 microprocessor. *IEEE Micro*, 19(2):24–36, March/April 1999.

[15] E. Kushilevitz and N. Nisan. *Communication Complexity*. Cambridge University Press, 1997.

[16] C.-C. Lee, C. C. Chen, and T. N. Mudge. The bi-mode branch predictor. In *Proceedings of the 30th Annual International Symposium on Microarchitecture*, November 1997.

[17] D. Lindsay. *Static Methods in Branch Prediction*. PhD thesis, University of Colorado, Department of Computer Science, 1998.

[18] S. McFarling. Combining branch predictors. Technical Report TN-36m, Digital Western Research Laboratory, June 1993.

[19] H. Patil and J. Emer. Combining static and dynamic branch prediction to reduce destructive aliasing. In *Proceedings of the Sixth International Symposium on High Performance Computer Architecture*, January 2000.

[20] G. Reinman and N. Jouppi. Extensions to cacti, 1999. Unpublished document.

[21] S. Sechrest, C.-C. Lee, and T.N. Mudge. Correlation and aliasing in dynamic branch predictors. In *Proceedings of the 23rd International Symposium on Computer Architecture*, May 1999.

[22] R. L. Sites. *Alpha Architecture Reference Manual*. Digital Press, Burlington, MA, 1992.

[23] E. Sprangle, R.S. Chappell, M. Alsup, and Y. N. Patt. The Agree predictor: A mechanism for reducing negative branch history interference. In *Proceedings of the 24th International Symposium on Computer Architecture*, June 1997.

[24] J. Stark, M. Evers, and Y. N. Patt. Variable length path branch prediction. In *Proceedings of the 8th International Conference on Architectural Support for Programming Languages and Operating Systems*, October 1998.

[25] Y. Ye, S. Borkar, and V. De. A new technique for standby leakage reduction in high performance circuits. In *Symposium on VLSI Circuits*, June 1998.

[26] T.-Y. Yeh and Y. N. Patt. Two-level adaptive branch prediction. In *Proceedings of the 24^{th} ACM/IEEE International Symposium on Microarchitecture*, November 1991.

[27] C. Young, D. S. Johnson, D. R. Karger, and M. D. Smith. Near-optimal intraprocedural branch alignment. In *Proceedings of the SIGPLAN'97 Conference on Program Language Design and Implementation*, June 1997.

[28] C. Young and M. D. Smith. Static correlated branch prediction. *ACM Transactions on Programming Languages and Systems*, May 1999.