Boolean Formula-based Branch Prediction for Future Technologies

Daniel A. Jiménez[†] Heather L. Hanson[‡] Calvin Lin[†]

[†]*Dept. of Computer Sciences*
The University of Texas at Austin
Austin, TX 78712

[‡]*Dept. of Electrical & Computer Engineering*
The University of Texas at Austin
Austin, TX 78712

{djimenez,hhanson,lin}@cs.utexas.edu

Abstract

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 prediction. By using a special class of Boolean formulas, our encoding is extremely concise. By replacing the large tables found in current predictors with a small, fast circuit, our scheme is ideally suited to future technologies that will have large wire delays. In a projected 70 nm technology and an aggressive clock rate of about 5 GHz, an implementation of our method that uses an 8-bit formula encoding has a misprediction rate of 6.0%, 42% lower than that of the best gshare *predictor implementable in that same technology. In today's technology, a 16-bit version of our predictor can replace bias bits in an 8K-entry* agree *predictor to achieve a 2.86% misprediction rate, which is slightly lower than the 2.93% misprediction rate of the Alpha 21264 hybrid predictor, even though the Alpha predictor has almost twice the hardware budget. Our predictor also consumes much less power than table-based predictors. This paper describes our predictor, explains 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 95% 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.

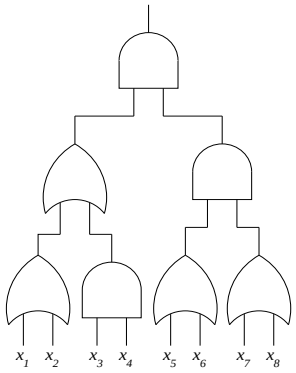

Figure 1. Tree representation of the formula $((x_1 \vee x_2) \vee (x_3 \wedge x_4)) \wedge ((x_5 \vee x_6) \wedge (x_7 \vee x_8))$. Only seven bits, representing the types of the gates, are needed to encode this formula.

We have seen this problem even in current microprocessors. The AMD K6 line of microprocessors featured a highly accurate 8K-entry GAs branch predictor; however, the newer AMD Athlon microprocessor has a scaled-back 2K-entry branch predictor [9]. This change reduces the delay and real-estate costs of the branch predictor, and might be one reason why the Athlon has been able to achieve aggressive clock rates. In exchange for the higher clock rates, AMD has sacrificed branch prediction accuracy in a processor whose longer pipeline increases the misprediction penalty.

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. Whereas an arbitrary Boolean function in N variables requires $O(2^N)$ bits to encode, monotone read-once Boolean formulas only require N bits. Figure 1 shows such a formula as a logic diagram.

We show that the use of 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 small 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 6.0%, which is 42% lower than that of the best *gshare* predictor implementable in that technology. Our predictor also uses much less power than table-based methods. For example, in 70 nm technology, an 8-bit Boolean formula predictor consumes 0.06 mW, while a *gshare* predictor with comparable accuracy consumes 12.9 mW. As another example, a 16-bit Boolean formula predictor has a 33% lower misprediction rate lower than a 4K-entry *gshare*, while consuming less than 1% of the power of *gshare*.

The primary contribution of this paper is a new branch prediction scheme that encodes into branch instructions a predictor in the form of a Boolean formula. Our method is particularly attractive in light of trends in technology scaling and wire delays. Secondary contributions include the following: (1) We describe the hardware implementation of our predictor and analyze it in terms of delay and power; (2) we describe a profiling algorithm for training our predictor; (3) we describe hybrid versions of our predictor that combine our technique with dynamic predictors; and (4) we evaluate the accuracy of our method using the SPEC 2000 integer benchmarks.

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 (PHT) of two-bit saturating counters is indexed by a combination of branch address and global or per-branch history. The most significant 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 [19, 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. Instead of indexing a table of saturating counters, this predictor uses a prediction mechanism that is based on perceptron learning. 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 it can be 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 [5, 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 respond to dynamic changes in program behavior.

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 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 enlist the compiler to assist in branch prediction. Mahlke and Natarajan [18] and August *et al.* [4] 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] increases the accuracy of static prediction by introducing duplicate basic blocks and encoding in the program counter information about the path taken to reach a particular static branch.

2.4 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. Table 1 shows the maximum size of a *gshare*-like predictor as technology moves forward (see Section 3.6 for methodological details).

Jiménez *et 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 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 in exchange for some extra profiling effort and changes to the instruction set architecture (ISA).

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 single-cycle *gshare* must decrease as technology moves forward.

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/RISC and IA-64, encode constant Boolean functions, which require 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 formula 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. Branch instructions would need to have over 1000 bits to allow all of these functions to be encoded. Therefore, we consider an extremely compact, but sufficiently expressive, encoding of Boolean formulas.

3.2 Read-Once Monotone Boolean Formulas

We now describe a 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. The functions induced by these formulas are monotone [15], hence the name.

In a *read-once formula* each variable appears exactly once in the formula. Read-once formulas are also known as μ-formulas or Boolean trees [3]. 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" and "always predict not taken," which are the most common Boolean functions for branch prediction.

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, using the same shift register for all branches. 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.

The following formula is an example of a monotone read-once Boolean formula used for branch prediction with a history length of 8:

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

This formula 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."

3.4 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, along with the number of times each history leads 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 and argues that the cost is reasonable for history lengths up to 16.

3.5 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. Coincidentally, this function is equivalent to the carry-out computed by a full adder. 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 $2 \log_2 N$ of these subcircuits and add an XOR, which is a sub-circuit 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. In particular, the circuit is optimized for static CMOS technology with NAND gates and is not a read-once circuit.

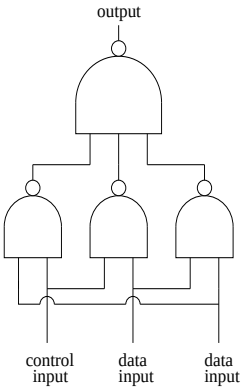

Figure 2. Boolean connective subcircuit. If the control input is 0, then the output is the AND of the two data inputs. Otherwise, the output is the OR of the two data inputs.

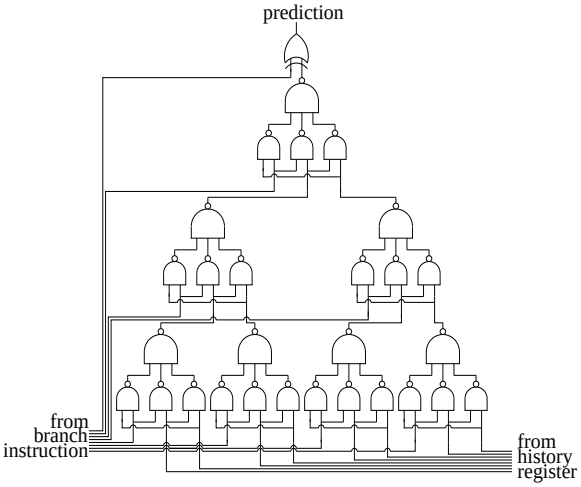

Figure 3. Boolean formula branch predictor circuit. This circuit makes a branch prediction based on a history length of 8 and an 8-bit encoding of a read-once Boolean formula.

3.6 Delay

The depth of the formula evaluation circuit with N inputs is $2 \log_2 N$ 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.

To estimate predictor access times for a range of current and future integrated circuit generations, we use circuit simulations and a modified version of the CACTI 2.0 tool for simulating cache delay. This modified version of CACTI is more accurate in several ways [2]. First, while the original version of CACTI 2.0 [20] uses a simplistic linear scaling for delay estimates, the modified simulator uses separate wire models to account for the physical layout of wire interconnects: thin local interconnect, taller and wider wires for longer distances, and the widest and tallest metal traces for global interconnect. Second, wire resistance is based on copper rather than aluminum material properties. Third, all capacitance values are derived from three-dimensional electric field equations. Fourth, bit-lines are placed in the middle layer metal, where resistance is lower. Finally, bit-addressing is allowed instead of byte-addressing.

Our results for projected technologies, including those given in Table 1, 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.

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. We calculated the lookup time for a *gshare* predictor using our modified CACTI 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.

As fabrication technology improves, transistors can be made smaller and faster, resulting in higher clock frequen-

Minimum Feature Size (nm)	Access Time (picoseconds)		
	4K-entry *gshare*	Formula, $N = 8$	Formula, $N = 16$
180	551	211	260
130	402	168	208
100	321	112	138
70	228	85	103
50	167	50	59

Table 2. Access times for a 4K-entry *gshare* predictor vs. two versions of the Boolean formula predictor. The 8-bit Boolean formula predictor and *gshare* achieve similar accuracies. The delays were obtained using an HSPICE model for the Boolean formula predictor and a modified version of CACTI 2.0 for *gshare*.

Minimum Feature Size (nm)	Power (milliwatts)		
	4K-entry *gshare*	Formula, $N = 8$	Formula, $N = 16$
180	51.4	0.61	1.28
130	31.0	0.28	0.58
100	27.4	0.11	0.24
70	12.9	0.06	0.12
50	8.40	0.06	0.13

Table 3. Dynamic power consumption for two versions of the Boolean formula predictor and a 4K-entry *gshare*. These figures were obtained from HSPICE for the Boolean formula predictor and CACTI for *gshare*.

cies 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 more than a single cycle—167 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 prepare for and act upon the prediction.

One concern with our predictor is that the contents of the branch opcode are on the critical path to making a prediction; the Boolean formula must be read before it can be evaluated. However, this delay is common to any branch predictor that uses bias bits or any other type of information from the branch instruction, such as the agree predictor used on the HP-PA/RISC or the static/dynamic and bias bits provided by IA-64. One solution is to provide pre-decode bits in the instruction cache that provide the opcode information quickly.

3.7 Power

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

The Boolean formula predictor is a combinational circuit that uses less dynamic power than an SRAM-based 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 formula 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, measure using the modified CACTI 2.0. The $N = 8$ results 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].

3.8 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 prediction, Stark *et al.* [24] use extra bits such as these to communicate to the microarchitecture information on hash functions for a branch predictor.

We propose changing 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. For a Boolean-formula based branch predictor requiring N bits in a branch instruction, we propose to reallocate N of the offset bits to the formula. Some long branches will need to be split into a branch followed by a jump to the target, increasing the number of

instructions executed.

We measure the harmonic mean over the SPEC 2000 integer benchmarks of the percentage of extra instructions executed on the Alpha when offset bits are reallocated to Boolean formula predictors. With formulas of up to 9 bits, the number of extra instructions is negligible. With 12-bit formulas, only 0.2% more instructions are executed. With 14-bit formulas, 1.0% more instructions are executed. As history length increases beyond 16 bits, this encoding technique becomes less feasible. For longer histories, we have developed a more sophisticated technique that exploits the fact that most of the functions are constant.

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 bits) 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.

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 up to 300 million branches. We use the `train` inputs for the profiling runs, and we use the `ref` inputs to evaluate the accuracy of the various predictors. To better capture the steady-state performance of the branch predictors, our evaluation runs skip the first 50 million branches, as several of the benchmarks have an initialization period (lasting fewer than 50 million branches), during which branch prediction accuracy is unusually high. Each benchmark executes at least 300 million branches and over one billion instructions on the `test` inputs before the simulation ends.

4.2 Predictors Simulated

We simulate monotone read-once Boolean formula predictors for $2 \leq N \leq 18$. We use only global history information, i.e., we do not use path or per-branch information. We also simulate the *gshare* [19], bi-mode [16] and agree [23] branch predictors, three well-known global dynamic branch predictors from the literature. 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 [19], so for each predictor and each hardware budget, we try all possible history lengths on the `train` inputs and keep the one with the lowest average misprediction accuracy.

To give a lower-bound on misprediction rates for any Boolean-formula based predictor, we also measure the results of using *arbitrary* Boolean formulas. To find the best arbitrary Boolean formula for a particular static branch, we measure the number of taken versus not-taken branches for each history leading up to that branch in the training set, then assign to each history the prediction yielding the most correctly predicted dynamic branches. Out of all the possible histories leading to a branch, only a small fraction will actually be observed; all other histories are assigned the bias bit for that branch. The arbitrary predictor is represented by the profiling algorithm as a set of rows in a truth table where the inputs are the histories and the output is the prediction.

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 only 16 bits.

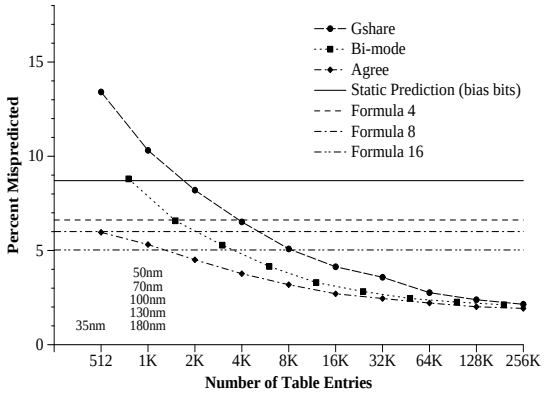

Figure 4. Accuracy of dynamic branch predictors vs. static prediction and the Boolean formula predictor. The numbers above the x-axis show the technologies in which the corresponding hardware budgets are reachable in one cycle with aggressive clocking. Misprediction rates are the harmonic means over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

At today's 180 nm and 130 nm technologies, for which branch predictors with only about 1K to 2K table entries state are available at more aggressive clock speeds, a 4-bit Boolean formula predictor with a misprediction rate of 6.6% roughly matches the accuracy of the bi-mode predictor. With a history length of 16, the Boolean formula predictor has a misprediction rate of 5.02%, an improvement of 24% over the 1.5K-entry bi-mode predictor.

To put these figures another way, a 4-bit Boolean formula predictor achieves roughly the same predictive power as a 4K-entry *gshare* predictor. A 16-bit Boolean formula predictor is about as accurate as an 8K-entry gshare predictor, a 3K-entry bi-mode predictor, or a 2K-entry agree predictor.

Figure 5 shows, for history lengths ranging from 2 to 18, misprediction rates for the monotone read-once Boolean formula predictor, as well as for the predictor that uses arbitrary formulas. 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 predictor; these table 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 lower than that of the dynamic predictors. However, in a few cases, such as `256.bzip2`, the formula predictor's misprediction rate is high, most likely due to input-dependent program behavior that cannot be learned by profiling.

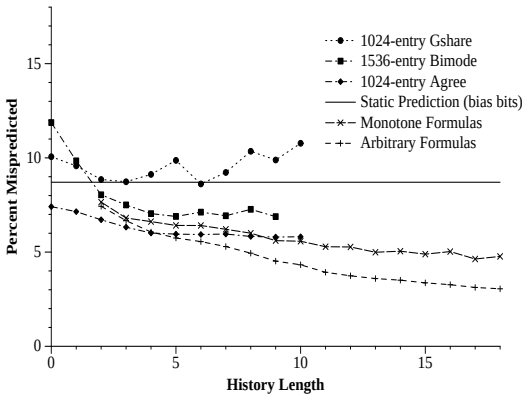

Figure 5. Misprediction rate for the Boolean formula predictor as a function of history length.

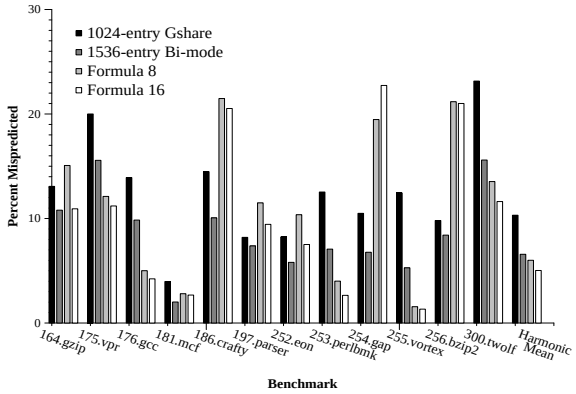

Figure 6. Accuracy of the predictors on each benchmark. This graph compares the Boolean formula predictor at history lengths of 8 and 16 against aggressively clocked implementations of *gshare* and bi-mode.

Figure 7 shows the misprediction rates of predictors using the agree mechanism combined with our formula predictor. An agree predictor predicts whether a branch outcome will agree with a bias bit, turning destructive aliasing into constructive aliasing. Our combined agree/formula predictors use a PHT to predict whether the branch outcome will agree with the output of a Boolean formula, rather than a bias bit. With a 1K-entry PHT, the agree predictor with bias bits yields a misprediction rate of 5.3%. The 8-bit version of our agree/formula predictor decreases this rate to 4.4%, an improvement of 17%. The 16-bit version of our predictor has a misprediction rate of 3.9%, an improvement of 25%.

For reference, we compare our predictor with the Alpha 21264 hybrid branch predictor, which is the most accurate existing predictor for which implementation details are readily available [14]. This predictor uses a 4K-entry global history predictor and a 1K-entry per-branch history predictor combined with a 4K-entry chooser, consuming roughly 4KB of state. The Alpha 21264 predictor achieves a misprediction rate of 2.93% on the traces we gathered. At the same hardware budget, the agree predictor, when enhanced with the 16-bit version of our Boolean formula predictor, achieves a misprediction rate of 2.55%. Even at half the hardware budget of the Alpha 21264 predictor, an 8K-entry version of our agree/formula hybrid achieves a misprediction rate of 2.86%, narrowly better than the Alpha hybrid. Using our aggressive clock modeling, the largest hybrid agree/formula predictor available in a single cycle will achieve a misprediction rate of 3.97%, which is 35% higher than that of the Alpha predictor. However, an important point of our research is that complex predictors such as the Alpha's are infeasible at higher clock rates. Even today's Alpha must employ an overriding mechanism [14], in which branch predictions that don't agree with the less sophisticated cache line predictor introduce a single-cycle

bubble into the pipeline, reducing the performance advantage of the more accurate hybrid predictor.

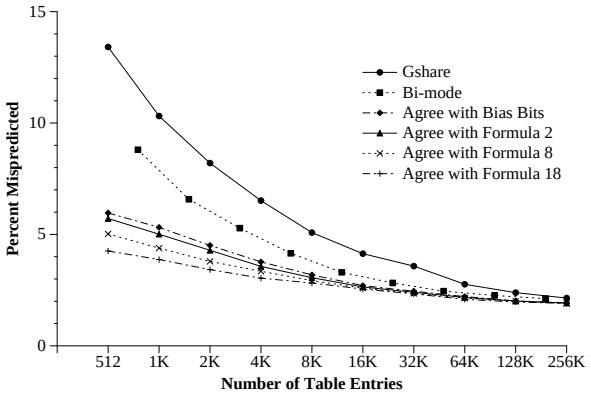

Figure 7. Accuracies of Boolean formula predictors using the agree mechanism. Misprediction rates are harmonic means over SPEC 2000.

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, 78% of static branches in the SPEC 2000 integer benchmarks are best predicted with a constant formula, as opposed to only 49% for a history length of 16. As history length 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 4 shows the dynamic frequencies for each formula with a history length of four, along with the misprediction rate for each formula using a 4-bit Boolean formula predictor and for bias bits. For brevity, we omit similar tables for the other history lengths.

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.

Our current implementation takes time exponential in the history length. However, for the small history lengths that 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 on a 733MHz Pentium III. 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 performing other tasks that any typical feedback-directed optimization 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, as the various 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.

Formula	% Dyn. Freq.	% Mispredicted	
		Formula	Bias
1	40.84	9.4	9.4
0	37.14	10.0	10.0
$(x_0 \vee x_1) \wedge (x_2 \vee x_3)$	3.15	21.8	36.3
$\neg((x_0 \vee x_1) \wedge (x_2 \vee x_3))$	2.36	24.6	36.6
$(x_0 \vee x_1) \vee (x_2 \wedge x_3)$	2.06	21.5	29.3
$\neg((x_0 \vee x_1) \vee (x_2 \wedge x_3))$	1.73	14.4	24.5
$\neg((x_0 \wedge x_1) \wedge (x_2 \vee x_3))$	1.64	20.1	26.8
$(x_0 \wedge x_1) \wedge (x_2 \vee x_3)$	1.60	15.8	22.0
$\neg((x_0 \wedge x_1) \vee (x_2 \vee x_3))$	1.54	16.3	23.7
$(x_0 \wedge x_1) \vee (x_2 \vee x_3)$	1.49	14.1	18.6
$(x_0 \vee x_1) \wedge (x_2 \wedge x_3)$	1.30	26.4	34.9
$(x_0 \wedge x_1) \vee (x_2 \wedge x_3)$	1.23	20.3	38.7
$\neg((x_0 \wedge x_1) \vee (x_2 \wedge x_3))$	1.16	35.6	42.1
$\neg((x_0 \vee x_1) \wedge (x_2 \wedge x_3))$	1.09	26.2	36.1
$\neg((x_0 \wedge x_1) \wedge (x_2 \wedge x_3))$	0.99	21.6	18.5
$(x_0 \wedge x_1) \wedge (x_2 \wedge x_3)$	0.66	5.3	10.3

Table 4. 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.

5 Conclusions

We have introduced and evaluated a new branch prediction scheme that borrows from complexity theory the concept of a read-once monotone Boolean formula. These Boolean formulas provide a compact encoding of a class of functions that is expressive enough to perform branch prediction yet concise enough to be encoded in branch instructions. By offloading most of the prediction work to the compiler, our Boolean formula predictor is small, fast and consumes little power. While our scheme provides a competitive alternative to existing dynamic branch predictors, the real benefit of our scheme lies in the future, as our scheme is significantly less sensitive to the impending technology scaling issues caused by increased wire delays. Our predictor can also form a valuable component of an agree

or hybrid predictor, decreasing misprediction rates by providing better estimates of branch outcomes than bias bits.

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 of our brute-force algorithm.

6 Acknowledgments

We thank Vikas Agarwal for explaining the details of his changes to CACTI 2.0. We also thank Steve Keckler, Samuel Guyer, and the anonymous referees for their helpful discussions and comments on this paper. Calvin Lin is supported by 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 Int'l Symp. on Computer Architecture*, pp. 248–259, May 2000.

[2] V. Agarwal, S. W. Keckler, and D. Burger. Scaling of microarchitectural structures in future process technologies. Technical Report TR2000-02, Dept. of Computer Sciences, The Univ. of Texas at Austin, Feb. 2000.

[3] D. Angluin, L. Hellerstein, and M. Karpinski. Learning read-once formulas with queries. *J. of the ACM*, 40(1):185–210, Jan. 1993.

[4] D. I. August, D. A. Connors, J. C. Gyllenhaal, and W. W. Hwu. Architectural support for compiler-synthesized dynamic branch prediction strategies: Rationale and initial results. In *the Third Int'l Symp. on High-Performance Computer Architecture*, Feb. 1997.

[5] T. Ball and J. Larus. Branch prediction for free. In *Conf. on Programming Language Design and Implementation*, pp. 300–313, June 1993.

[6] D. Burger and T. M. Austin. The SimpleScalar tool set version 2.0. Technical Report 1342, Computer Sciences Dept., Univ. 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 Trans. on Programming Languages and Systems*, 19(1), 1997.

[8] B. Calder and D. Grunwald. Reducing branch costs via branch alignment. In *the Sixth Int'l Conf. on Architectural Support for Programming Languages and Operating Systems*, Oct. 1994.

[9] K. Diefendorff. K7 challenges Intel. *Microprocessor Report*, 12(14), Oct. 1998.

[10] A.N. Eden and T.N. Mudge. The YAGS branch prediction scheme. In *the 31st Int'l Symp. on Microarchitecture*, Nov. 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 Univ., 2000.

[12] D. A. Jiménez, S. W. Keckler, and C. Lin. The impact of delay on the design of branch predictors. In *33rd Int'l Symp. on Microarchitecture*, pp. 67–76, Dec. 2000.

[13] D. A. Jiménez and C. Lin. Dynamic branch prediction with perceptrons. In *the 7^{th} Int'l Symp. on High Performance Computer Architecture*, ppl. 197–206, Jan. 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 Univ. Press, 1997.

[16] C.-C. Lee, C.C. Chen, and T.N. Mudge. The bi-mode branch predictor. In *the 30th Int'l Symp. on Microarchitecture*, Nov. 1997.

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

[18] S. Mahlke and B. Natarajan. Compiler synthesized dynamic branch prediction. In *the 29th Int'l Symp. on Microarchitecture*, Dec. 1996.

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

[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 *the 23rd Int'l Symp. 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 *the 24th Int'l Symp. on Computer Architecture*, June 1997.

[24] J. Stark, M. Evers, and Y. N. Patt. Variable length path branch prediction. In *the 8th Int'l Conf. 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 *Symp. on VLSI Circuits*, June 1998.

[26] T.-Y. Yeh and Y. Patt. Two-level adaptive branch prediction. In *the 24^{th} Int'l Symp. on Microarchitecture*, Nov. 1991.

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

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