Compiler Synthesized Dynamic Branch Prediction

Scott Mahlke Balas Natarajan
Hewlett-Packard Laboratories
Palo Alto, CA 94304
E-mail: {mahlke,balas}@hpl.hp.com

Abstract

Branch prediction is the predominant approach for minimizing the pipeline breaks caused by branch instructions. Traditionally, branch prediction is accomplished in one of two ways, static prediction at compile-time via compiler analysis or dynamic prediction at run-time via special hardware structures. In this paper, we propose a novel technique that aims to combine the strengths of the two approaches – the lower cost of compile-time analysis with the effectiveness of dynamic prediction. Specifically, we propose that the compiler use profile feedback to define a prediction function for each branch and insert a few explicit instructions per branch into the compiled code to compute the prediction function. These instructions are carefully selected to predict the direction of the branch using any information available during run-time. A strength of this approach is that information beyond branch history can be used to make predictions, such as the contents of the architectural registers. To substantiate our proposal, we present an algorithm for selecting the prediction instructions, and demonstrate the performance of the approach against contemporary static and dynamic branch prediction strategies.

1 Introduction

The performance of pipelined processors depends strongly on the efficiency with which branch instructions are handled. Branches cause lengthy breaks in the pipeline by redirecting instruction flow during program execution. In a simple pipelined processor, instructions proceed sequentially through fetch, decode, execute, and write-back stages of the pipeline. When a branch is fetched, the address of the instruction that will execute next is not immediately known. Hence, the fetch stage must stall and wait for the branch target to be calculated. The target of a branch is generally resolved during the execute stage. Therefore, the fetch unit stalls while the branch advances through the decode and execute stages of the pipeline. After the branch instruction has completed execution, the branch direction is known and instruction fetch can safely resume at the correct target address. Stalling the instruction fetch for each branch introduces a large number of empty cycles, referred to as bubbles, into the pipeline. These bubbles severely limit the performance of pipelined processors by restricting the utilization of the processor resources. The performance problem becomes amplified as pipeline depth and instruction issue width of processors are increased.

Branch prediction is a highly effective approach for dealing with branches in pipelined processors. Branch prediction guesses the targets of branches in order to speculatively continue executing instructions while a branch target is being calculated. In the cases where the prediction is correct, all the speculative instructions are useful instructions, and pipeline bubbles are completely eliminated. On the other hand, incorrect prediction results in both the normal pipeline bubbles while the branch target is resolved, as well as additional delay to remove all instructions that were improperly executed. Clearly, the accuracy of the branch prediction strategy is central to processor performance.

There are two basic classes of branch prediction strategies: static and dynamic. Static branch prediction utilizes information available at compile time to make predictions. In general, the compiler is responsible for static branch prediction. The most common static branch prediction approach is to use profile information [8],[5]. Other static branch prediction approaches include deriving and applying heuristics based on program structure [2],[6], and utilizing machine-learning techniques to determine static feature sets of branches that are correlated with branch direction [3]. Static branch prediction can also be done with simple hardware such as prediction using branch direction (backward taken, forward not taken) [15]. The major advantage of static branch prediction is the low cost. Branch state information is not required since the prediction is explicitly specified by the program itself. The fundamental limitation of static branch prediction is the prediction is fixed at compile-time, thus it cannot vary during program execution. As a result, the accuracy of static branch prediction is inherently limited for unbiased branches.

The second class of branch prediction strategies is dynamic branch prediction. Dynamic branch prediction utilizes run-time behavior to make predictions. In general, a hardware structure is provided to maintain branch history. Based on the current history, a prediction is made for each branch encountered in the program. Example dynamic branch prediction schemes are the branch target buffer (BTB) with a 2-bit saturating counter [15] [11] and two-level adaptive branch prediction [16],[17],[14]. More recently, hybrid branch predictors which allow different dynamic predictors to be selected for different branches, have been proposed [13],[4]. The major advantage of dynamic prediction is the increased accuracy. The use of run-time information and the ability to predict a branch differently during various phases of execution, allows dynamic techniques to enjoy significantly higher accuracy than static techniques. For superscalar and VLIW processors, this increased accuracy translates into large performance gains. The primary disadvantage of the dynamic prediction techniques is the cost. Dynamic branch prediction techniques utilize relatively large amounts of hardware and provide difficult challenges for circuit designers to meet cycle time goals.

In this paper, we propose a new approach for dynamic branch prediction, referred to as compiler synthesized prediction, that is not hardware based. Rather, the compiler is completely responsible for defining and realizing a prediction function for each branch. The technique is based on the following observation. When a branch is executed, the contents of the registers offer complete information on the outcome of the branch. In particular, evaluating the branch comparison on the source register values would perfectly predict the branch. As we go backwards in time from the branch, this information becomes gradually less complete. Thus, the contents of the registers offer a path to predicting the branch, although the prediction may not be perfect eight or sixteen cycles in advance of the branch. With this in mind, we propose that the compiler add a few instructions to the compiled code ahead of each branch to be predicted. The role of these instructions would be to predict the branch. These instructions will be standard operations from the instruction set of the processor, such as arithmetic, comparison, and logical instructions. The only hardware support needed for these instructions is that the instructions be allowed to write their predictions into a “branch predict register” that will later be used by the processor's instruction pipeline.

There are two major advantages to this approach. First, the flexibility of selecting a predictor function is vastly increased. The compiler is free to realize almost any function including standard hardware prediction functions (such as a 2-bit counter) as well as new functions involving the processor register values. These new predictor functions offer the potential to substantially increase the prediction accuracy where branch history is not effective. Furthermore, a specific prediction function can be tailored for each branch which minimizes cost and maximizes performance, as is utilized with the hybrid predictor strategy [4]. However with this approach, the number of predictors is not limited to a preselected group which are available in the hardware. The second advantage of this approach is the reduced hardware overhead. Branch prediction state information is no longer maintained and manipulated by a separate hardware structure. Rather, the compiler uses architecturally visible registers and explicit instructions to compute the branch prediction values.

Two important questions arise with compiler synthesized prediction. How is the compiler to determine the predictor, and how complex should the predictor be? Clearly, there is a tradeoff between the complexity of the predictor and its effectiveness. We give an algorithm that automatically generates a predictor for a branch from profile information — history of the machine register values and the branch outcome. The complexity of the predictor can be explicitly controlled in terms of the number of instructions involved. We close with experimental results obtained by applying our algorithm to some sample benchmarks. For each benchmark, we selected the branches that are most frequently mispredicted by the profile-based static predictor. Our results show that our proposed method can offer significantly improved prediction on these branches. In addition, we compare our results with popular hardware-based dynamic predictors, namely a 2-bit counter BTB and a two-level branch predictor.

2 Branch Architecture

The major architectural feature that is required to support compiler synthesized dynamic branch prediction is a branch prediction mechanism that is visible in the instruction set. By visible, it is meant that the compiler can generate instructions which manipulate the prediction values for a particular branch. In this manner, new prediction functions can be realized by inserting additional instructions into the program to compute prediction values and write them into the appropriate predictor locations. An additional requirement is that there must be a known way to associate branch predictor locations with the branches themselves. This is necessary to be able to selectively modify prediction values for specific branches.

The PlayDoh branch architecture is chosen as the baseline architecture for this paper [10]. PlayDoh is an experimental architecture specification that serves as a vehicle for ILP research. While PlayDoh provides general branch prediction and instruction prefetch mechanisms exposed to the compiler, it is not sufficient in its current form to support compiler synthesized dynamic branch prediction. Therefore, we extend it to support dynamic branch prediction. This section summarizes the baseline PlayDoh branch architecture and our extensions. The application of compiler synthesized branch prediction is not limited in any way to architectures such as PlayDoh. This section closes with a brief discussion of several strategies to incorporate compiler synthesized prediction into more traditional branch architectures.

2.1 PlayDoh branch architecture

In the PlayDoh architecture, a branch is broken down into its basic components [10]. The purpose of this strategy is to allow different pieces of information regarding a branch to be specified as soon as they become available. The processor can then utilize this early information to reduce the performance overhead of the branch. Branches consist of three components in PlayDoh:

1. Specification of the target address - Prepare-to-branch instructions specify the branch target address in advance of the branch point allowing a prefetch of the instructions from the target address to be performed [18]. A static prediction bit is also provided at this step to indicate the direction the compiler believes the branch will go.

2. Computation of the branch condition - The branch condition is computed by a compare-to-predicate instruction and stored in a predicate register. For an unconditional branch, this condition computation is not required. Hence, this component does not exist for unconditional branches.

3. Transfer of control - The actual redirection of control flow occurs if the branch is taken. The true branch in PlayDoh performs this action. The architecture provides several types of branch instructions including conditional, unconditional, branch and link, and special branches to support loop execution.

The PlayDoh instructions and their functionality are best described with an example. For simplicity, the guarding predicates

```
     blt   r2, r3, L1            pbra               b2, L1, 1
                                 cmpp_w_lt_un_un    p2, -, r2, r3
                                 brct               b2, p2

            (a)                             (b)
```

Figure 1: Representation of a compare-and-branch instruction in PlayDoh, (a) conventional branch, (b) PlayDoh equivalent.

```
                    Original branch:    blt   r2, r3, L1

                    p9 = f (···)    cmpp_w_lt_un_un   p2, -, r2, r3
pbra                b2, L1, p9, 0   pbra              b2, L1, p2, 1
cmpp_w_lt_un_un     p2, -, r2, r3   brct              b2, p2
brct                b2, p2

                    (a)                         (b)
```

Figure 2: Representation of a compare-and-branch in PlayDoh with dynamic prediction, (a) dynamic prediction realization, (b) early comparison operand availability realization.

for each PlayDoh instruction are ignored in this discussion. Figure 1 contains an example compare-and-branch instruction for a general RISC architecture along with its realization in the PlayDoh architecture. The original branch, *blt*, branches to label *L1* if $r2$ is less than $r3$. Otherwise, execution falls through to the next sequential instruction. In the PlayDoh architecture, the branch is split up into its 3 components. The first instruction, *pbra*, specifies the potential target of the branch if it is taken, namely label *L1*. In addition, a 1-bit literal is specified to indicate the static prediction direction for the branch. In this example, the compiler expects the branch to be taken, so a '1' is used. The prepare-to-branch instruction writes the computed effective target address and prediction bit into a special set of registers, referred to as the branch target registers (BTRs). For the example, $b2$ is the BTR that is utilized. The BTRs are the medium to communicate the target and prediction information to the subsequent branch as well as the instruction prefetch unit.

The second instruction performs the actual comparison operation, computing $r2 < r3$ and storing the result into a predicate register ($p2$ in the example). The final instruction performs the actual branch. Each branch specifies a source BTR, which contains the branch target address. Furthermore, all branches which are conditional specify a source predicate operand as the condition. In this example, a branch on condition true ($brct$) is utilized, which indicates if p2 contains a 1, the branch should transfer control to the target address in $b2$.

The usage and functionality of the PlayDoh branch instructions is still a subject of open research, but some general statements can be made. Ideally, a prepare-to-branch instruction should be scheduled so that there is enough time between the prepare and the corresponding branch to prefetch instructions from the target address. Therefore, when the actual branch is encountered, instruction fetch can be immediately switched to the predicted target. The compiler must also manage the prefetch resources so they are not over-committed by a number of prepare instructions. There are no ordering restrictions placed between the prepare-to-branch and comparison instructions, so they can be scheduled in any order. However, the comparison instruction normally has limited code motion freedom because it must wait for the comparison operands to become available. In contrast, the prepare instruction uses only immediate operands, so it can be scheduled much earlier to allow the prefetch to complete.

2.2 Extensions to PlayDoh

As it stands, the PlayDoh architecture only supports static branch prediction. To enhance it to support compiler synthesized dynamic branch prediction, two simple extensions are made to the operation of the prepare-to-branch instruction. First, the 1-bit literal field used for static prediction is generalized to be a predicate register operand. In this manner, a prediction value may be computed by an arbitrary set of machine instructions with the final result placed in a predicate register. The prepare-to-branch instruction subsequently reads the particular predicate register to obtain the prediction value.

The second extension is to efficiently handle the case where a prediction is not required. Rather, the exact direction of the branch is known when the prepare-to-branch instruction is issued. For this case, the comparison operands are available very early, which allows the comparison to be computed in advance of the prepare. To support this mode of operation, a new 1-bit literal field is added to the prepare-to-branch instruction which specifies whether the predicate operand is the actual branch direction or simply a prediction. In the case where the predicate is the actual branch condition, the processor can be much more aggressive in terms of fetching and executing instructions after the branch since there is no ambiguity as to whether the prediction may turn out to be wrong. In the opposite case where the predicate utilized by the prepare is a prediction, the normal operation of the prepare is observed.

Instruction sequences illustrating the usage of the enhanced prepare-to-branch instruction are shown in Figure 2. The most common use is prepending the previous three instruction sequence with a set of instructions to compute a dynamic prediction for the branch. In the example (Figure 2a), the prediction value is computed by the function f and the result is stored into $p9$. The prepare-to-branch instruction explicitly sources the predicate register containing the prediction value along with the target of branch. In addition, the literal field specifying whether the predicate is the actual or a predicted value is set to 0, indicating the predicate is only a prediction. The prepare-to-branch instruction writes all this information into its destination BTR, $b2$. The two remaining instructions are the same as the conventional PlayDoh architecture.

The second case is where the branch comparison can be performed before the prepare. For this case (Figure 2b), the predicate operand for the prepare-to-branch is just specified as the destination of the comparison instruction, $p2$ in the example. In addition, the 1-bit field in the prepare is set to 1 to indicate the value in $p2$ is the actual branch direction. Again, the other instructions are unchanged from the baseline PlayDoh architecture.

2.3 Use in traditional branch architectures

The applicability of compiler synthesized dynamic branch prediction is not limited to architectures employing static branch prediction, such as PlayDoh. It can be efficiently incorporated into more conventional architectures that utilize hardware-based dynamic branch prediction. In [1], two schemes to accomplish compiler synthesis are proposed: Predicate Only Prediction (POP) and Predicate Enhanced Prediction (PEP). For these schemes, a traditional BTB is extended with an additional field for each entry. The new field holds the register number for the condition value of the branch. When a branch is allocated an entry in the BTB, the condition register number is recorded. Subsequent predictions of the branch fetch and utilize the contents of the condition register to assist with the prediction. In the POP scheme, the contents of the condition register fully specify the value of the prediction. In contrast for the PEP scheme, the contents of the condition registers are used to select among two potential prediction values.

These schemes offer two important advantages for a processor that utilizes hardware-based dynamic branch prediction. First, in cases where the comparison instruction for a particular branch can be hoisted sufficiently early, all mispredictions associated with the branch are eliminated. The branch predictor effectively communicates the known branch condition value to the instruction pipeline. The second advantage is that they efficiently support compiler synthesis of branch prediction functions. As with the PlayDoh model, the compiler inserts instructions into the program sufficiently in advance of the branch to compute prediction values. The only additional requirement is the condition value be stored in the branch condition register. Additionally with the PEP scheme, the compiler is free to selectively utilize its own synthesized prediction functions for certain branches and rely on the hardware-based predictor for the remaining branches.

3 The Prediction Algorithm

With support for compiler synthesized dynamic branch prediction available, the compiler is free to choose almost any prediction function for a branch. For the purposes of this paper, we have selected one class of predictor functions for investigation. Specifically, prediction functions are synthesized which correlate the values contained in architecture registers with the direction of the branch. Register contents and branch directions are obtained by profiling the target program. Of course, many other classes of prediction functions are interesting and will likely be the subject of future research.

For this section, a brief description of the general approach to obtain the information to be utilized by the compiler is first described. Then, the algorithms used to derive the branch prediction function are presented. The algorithm is best described in its basic form, i.e., in the context of infinite resources, assuming that the predictor can be arbitrarily complex. Later we describe a practical version of the algorithm that takes into account the fact that any real predictor will be restricted to consist of a few instructions. Otherwise, the cost of evaluating the predictor will outweigh the gains from the increased prediction accuracy.

3.1 General approach

The synthesis of branch prediction functions which correlate the values in architecture registers with the branch direction require two sets of information. First, for each branch a record of the direction taken for each execution is needed. This is identical to the information obtained with standard control flow profiling. The second set of information is a dump of the contents of the register file some predetermined distance before the branch. These register values are those that are available at run-time to compute a prediction. For the purposes of this paper, the register values 16 cycles before the branch are used. The procedure for collecting this information fits into the standard three step process used for almost all profile-based compilation:

Phase (1) Precompile: Compile the program without attempting any synthesis of branch prediction functions. Instrumentation code is inserted to collect branch direction information as well as the contents of the architectural registers a specified number of machine cycles prior to each branch.

Phase (2) Profile: Run the compiled code on sample inputs. The program will output the branch profile information as well as the register dump information.

Phase (3) Recompile: The register dumps are analyzed to identify correlations between branch directions and the contents of the registers. Based on the analysis, predictors are constructed for each branch. The operations for these predictors are added back to the intermediate code, and the intermediate code is then rescheduled to generate the final compiled code.

3.2 Basic prediction algorithm

Figure 3 gives our basic algorithm for constructing a predictor for a single branch. The algorithm limits itself to constructing predictors that are based on at most two registers in that the predictor will predict the branch predicate using the value of some two registers, at a predetermined number of cycles prior to the branch. This restriction on the number of registers is not inherent to the algorithm, and is only in the interest of pragmatics. Apart from this restriction, the algorithm pays no attention to resource constraints. Let the processor have n registers $r_1, r_2, ...r_n$. Let us focus on a specific branch b, and assume that the branch predicate value resides in a branch register r_b. At any particular execution of the branch, let $v_1, v_2, ..., v_n$ denote the values of the registers at a fixed number of cycles prior to branch b, and let v_b denote the value of the branch predicate after it is evaluated during that run. In other words, $v_1, v_2, ..., v_n$ and v_b constitute the entries in the register dump, for a particular pass through the branch, with the values $v_1 - v_n$ for the registers $r_1 - r_n$ being a fixed number of cycles prior to the branch, and the value v_b of the branch predicate register r_b just after the branch.

The algorithm maintains integer valued arrays, C_i for each register r_i and $C_{i,j}$ for each pair of registers r_i, r_j. There is one entry in C_i for each of the values r_i could take, and one entry in $C_{i,j}$ for each possible pair of values for r_i and r_j. This is an impracticable demand and is solely in the interest of expository clarity. Later, the algorithm will be modified to respect practical resource constraints. All the entries in the arrays are initialized to zero. For each pass through the branch b, if the branch predicate evaluated

```
for $i = 1, 2, ..., n$ do
    for all register values $v$ do $C_i[v] = 0$;
for $i = 1, 2, ..., n$ do
    for $i = 1, 2, ..., n$ do
        for all register values $u, v$ do $C_{i,j}[u][v] = 0$;
for each pass through branch $b$ do
    if $v_b = 1$ then
        for $i = 1, 2, ..., n$ do increment $C_i[v_i]$;
        for $i, j = 1, 2, ..., n$ do increment $C_{i,j}[v_i][v_j]$;
    else
        for $i = 1, 2, ..., n$ do decrement $C_i[v_i]$;
        for $i, j = 1, 2, ..., n$ do decrement $C_{i,j}[v_i][v_j]$;
for $i = 1, 2, ..., n$ do $S_i = \sum_v |C_i[v]|$;
for $i, j = 1, 2, ..., n$ do $S_{i,j} = \sum_{u,v} |C_{i,j}[u][v]|$;
let $k$ be such that $S_k$ is maximum over all $S_i$.
let $l, m$ be such that $S_{l,m}$ is maximum over all $S_{i,j}$.
if $S_{l,m} > S_k$ then
    output the two-register predictor based on $r_l$ and $r_m$.
else output the one-register predictor based on $r_k$;

One-register predictor based on $r_k$
    predict $(C_k[v_k] > 0)$

Two-register predictor based on $r_l, r_m$
    predict $(C_{l,m}[v_l][v_m] > 0)$
```

Figure 3: The basic algorithm for branch prediction synthesis.

to true (v_b has the value 1), the arrays entries incremented. In particular, if register r_i had value v_i, then entry $C_i[v_i]$ is incremented. Similarly, $C_{i,j}[v_i][v_j]$ is incremented for the pair of registers i, j. If the branch predicate evaluated to false (v_b has the value 0), the corresponding array entries are decremented. In essence, the entries in the array estimate the usefulness of knowing the values of the registers towards predicting the branch. If for a given value v of register r_i, the branch predicate evaluates to true much of the time, the array entry $C_i[v]$ will be strongly positive. Conversely, if the branch predicate evaluates to false much of the time, the array entry $C_i[v]$ will be strongly negative. If the branch predicate is evenly distributed between true and false, $C_i[v]$ will be close to zero.

With this in view, after all the passes through the branch have been processed, the absolute values of the array entries are summed up to assign scores S_i and $S_{i,j}$ to each register r_i and register pair r_i, r_j. The register or register pair with the highest score is the best predictor for the branch.

Once the register or register pair with the highest score has been selected, constructing a predictor is straightforward. Suppose a single register r_k has the highest score. A predictor involving r_k would take the value v_k of r_k, and check to see whether the array entry $C_k[v_k]$ is positive. If so, the branch is predicted to be 1, else not. If a pair of registers r_l, r_m have the highest score, the predictor would be similar. If $C_{l,m}[v_l][v_m]$ is positive, the branch predicate would be predicted to be 1, else 0.

Reg	Pass #						
	1	2	3	4	5	6	7
v_b	1	1	0	1	0	0	1
v_1	17	17	19	13	14	19	17
v_2	15	12	15	12	0	8	8

Table 1: Register dump for 7 passes of a branch, two-register machine of Example 1. Values for r_1 and r_2 are at 16 cycles ahead of the value for branch register r_b.

3.3 Example 1

To illustrate the working of the basic algorithm, we consider a processor with two registers r_1 and r_2, and one-branch register r_b. Suppose that for a sample execution in a certain program, a particular branch was executed seven times. The register dump for these seven passes is show in Table 1, where the values of r_1 and r_2 are at 16 cycles prior to the value of r_b.

Applying the basic algorithm, we get the following values for the array entries.

$C_1[17] = 3; C_1[19] = -2; C_1[13] = 1; C_1[14] = -1;$
$C_2[15] = 0; C_2[12] = 2; C_2[0] = -1; C_2[8] = 0;$
$C_{1,2}[17][15] = 1; C_{1,2}[17][12] = 1; C_{1,2}[19][15] = -1;$
$C_{1,2}[13][12] = 1; C_{1,2}[14][0] = -1;$
$C_{1,2}[19][8] = -1; C_{1,2}[17][8] = 1;$

From the above array values, we get that $S_1 = 7$, $S_2 = 3$ and $S_{1,2} = 7$. Taking S_1 to be the maximum of S_1, S_2 and $S_{1,2}$, the algorithm will output the one-register predictor based on r_1. The predictor is shown below in simplified form.

```
if $v_1 = 17$ or $v_1 = 13$ then predict 1
else predict 0
```

3.4 Practical prediction algorithm

We now discuss modifications to the basic algorithm to make it practical. Two issues must be addressed. (1) The range of the register values must be quantized so that the arrays C_i and $C_{i,j}$ are feasibly small. (2) The complexity of the predictors constructed should be small and controllable. In order to reduce the range of the register values, we quantize them to fit within a predetermined interval. We consider two quantizers, a linear quantizer and a logarithmic quantizer. The linear quantizer would scale the register values linearly to lie between 0 and 256, so that the arrays C_i would have 256 entries and the arrays $C_{i,j}$ would have 65536 entries. The logarithmic quantizer would simply take the logarithm of the register content in base 2, yielding a range from 0 to 32, assuming 32-bit registers. We will use one of these two quantizers for each register, depending on the distribution of values in the register dumps – for values that are widely dispersed we use the logarithmic quantizer and for values that are concentrated, the linear quantizer. A less ad hoc procedure for quantizer design is

```
for $i = 1, 2, ..., n$ do
    for all quantized register values $v$ do $C_i[v] = 0$;
for $i = 1, 2, ..., n$ do
    for $i = 1, 2, ..., n$ do
        for all register values $u, v$ do $C_{i,j}[u][v] = 0$;
for each pass through branch $b$ do
    Quantize $v_1, v_2, ..., v_n$;
    if $v_b = 1$ then
        for $i = 1, 2, ..., n$ do increment $C_i[v_i]$;
        for $i, j = 1, 2, ..., n$ do increment $C_{i,j}[v_i][v_j]$;
    else
        for $i = 1, 2, ..., n$ do decrement $C_i[v_i]$;
        for $i, j = 1, 2, ..., n$ do decrement $C_{i,j}[v_i][v_j]$;
for $i = 1, 2, ..., n$ do
    $S_i = \max_{v^*} \left|\sum_{v<v*} C_i[v]\right| + \left|\sum_{v \geq v^*} C_i[v]\right|$;
for $i, j = 1, 2, ..., n$ do
    $S_{i,j} = \max_{u^*,v^*}$
        $\left|\sum_{u<u^*,v<v^*} C_{i,j}[u][v]\right| + \left|\sum_{u \geq u^*,v<v^*} C_{i,j}[u][v]\right| +$
        $\left|\sum_{u<u^*,v \geq v^*} C_{i,j}[u][v]\right| + \left|\sum_{u \geq u^*,v \geq v^*} C_{i,j}[u][v]\right|$;

let $k$ be such that $S_k$ is maximum over all $S_i$.
let $l, m$ be such that $S_{l,m}$ is maximum over all $S_{i,j}$.

if $S_{l,m} > S_k$ then
    output the two-register predictor based on $r_l$ and $r_m$.
else output the one-register predictor based on $r_k$;


One-register/One-compare predictor based on $r_k$
    if $v_k < v^*$ then predict $(\sum_{v<v*} C_k[v] > 0)$;
    else predict $(\sum_{v \geq v^*} C_k[v] > 0)$;

Two-register/Two-compare predictor based on $r_l, r_m$
    let $u = v_l; v = v_m$;
    if $u < u^*, v < v^*$ then
        predict $(\sum_{u<u^*,v<v^*} C_{l,m}[u][v] > 0)$;
    else if $u \geq u^*, v < v^*$ then
        predict $(\sum_{u \geq u^*,v<v^*} C_{l,m}[u][v] > 0)$;
    else if $u < u^*, v \geq v^*$ then
        predict $(\sum_{u<u^*,v \geq v^*} C_{l,m}[u][v] > 0)$;
    else predict $(\sum_{u \geq u^*,v \geq v^*} C_{l,m}[u][v] > 0)$;
```

Figure 4: The practical algorithm for branch prediction synthesis.

deferred to a later paper. There is a considerable body of literature on optimal quantizer design for prediction, see [12] for instance.

We restrict the complexity of the constructed predictors as follows. A one-register predictor is only allowed to compare the contents of the register with a single literal. If the register value is less than the literal, it predicts the branch predicate to be true (false), else the branch predicate is predicted to be false (true). A three-way comparison involving *less than, equal to, and greater than* can be handled by a simple modification of the algorithm, but is not considered in this paper. Thus, the practical one-register predictor requires one operation for its implementation. Similarly, a two-register predictor is allowed to compare the contents of each of the registers with literals, and combine the results of the comparison with logical operators. There are four possible outcomes for the combinations of these comparisons, and each combination will be awarded an independent prediction. Since the four combinations can be obtained as the logical AND, OR or XOR of the outcomes of the two comparisons, the practical two-register predictor requires three operations for its implementation.

In light of the above, the practical algorithm of Figure 4 works in much the same was as the basic algorithm, with two exceptions. Firstly, the arrays C_i and $C_{i,j}$ are constructed using quantized register values. Secondly, the score values S_i and $S_{i,j}$ are computed in the context of the one-compare predictor. Specifically, for each array C_i, the algorithm finds the best cut-point v^* that optimally partitions the values of r_i, in the sense that predicting the branch predicate to be true (false) for all values that lie below v^* and to be false (true) for all other values is optimal over all cut-points. Similarly, for each array $C_{i,j}$, the algorithm finds the best cut-point (u^*, v^*) yielding the optimal rectilinear partition of the values of registers r_i and r_j.

3.5 Example 2

Let us reexamine Example 1 in the context of the practical algorithm. Since the values of the registers are all small positive integers, using the linear quantizer to scale the values between 0 and 256 will leave them unaltered. Thus, we can ignore the quantization step, and the arrays C_1, C_2 and $C_{1,2}$ will be the same as in Example 1. We now compute the scores S_1, S_2 and $S_{1,2}$. For r_1, $v^* = 18$ is the best cut-point, yielding a score $S_1 = 5$. For r_2, $v^* = 1$ is the best cut-point, yielding a score $S_2 = 3$. For the register pair r_1, r_2, the best cut-point is $u^* = 18, v^* = 1$ with a score $S_{1,2} = 7$. See Figure 5. Since $S_{1,2}$ is the maximum among these scores, the algorithm outputs the two-register/two-compare predictor based on registers r_1 and r_2. The predictor is shown below in simplified form.

```
Two-register/Two-compare predictor based on $r_1, r_2$
    if $v_1 < 18, v_2 \geq 1$ then predict 1
    else predict 0;
```

4 Experimental Results

The effectiveness of compiler synthesized dynamic branch prediction is analyzed in this section. Branch prediction accuracy results are presented and compared to four other branch prediction strategies: static prediction based on profile information, dynamic prediction using a 2-bit counter BTB, and two configurations of dynamic prediction using a two-level adaptive predictor.

4.1 Methodology

In order to accomplish compiler synthesized dynamic branch prediction, the PlayDoh emulation facilities in the IMPACT compiler were modified. The modifications were two fold. First, the

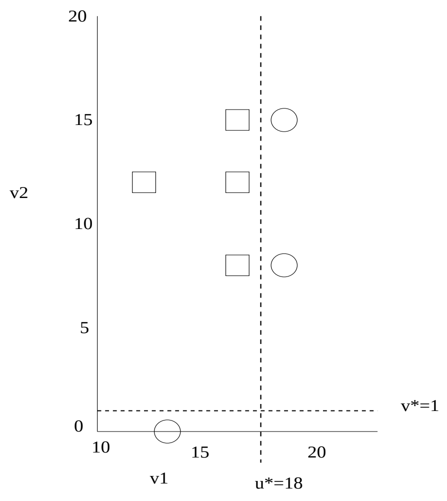

Figure 5: Pictorial view of the optimal cut-point for the two-register case of Example 2. The circles shown are (v_1, v_2) pairs for which $v_b = 0$, and the squares are (v_1, v_2) pairs for which $v_b = 1$. The optimal cut-point is selected so that the four quadrants formed by them best separate the circles from the squares.

synthesis algorithms require dumps of the register file contents to identify correlations. Instrumentation code was added into the PlayDoh emulation facility to provide such dumps a user specified number of cycles before the branch. Thus, when the program is executed for the profiling phase, the emulator maintains a running history of the register file contents for the last several cycles of execution. Then, for each instrumented branch, the register file contents the specified number of cycles before the branch are dumped into the profile database. For all the experiments presented in this paper, a 16 cycle distance is utilized unless otherwise specified.

The second modification was the insertion of new instructions to realize the derived branch prediction functions by the compiler backend. The compiler attempts to place the prediction instructions approximately 16 cycles ahead of the branch. In general, the compiler must insert these instructions along all paths of control leading to the branch to ensure the prediction is always available. For our experiments, this task was simplified by utilizing the superblock compilation approach in the IMPACT compiler [9]. For most cases, superblock construction ensures a single path leading to each important branch. Therefore, a single copy of the prediction instructions is inserted into each superblock with a branch targeted for prediction synthesis. With other scheduling frameworks, multiple copies of the prediction instructions may be required to cover all paths leading to a branch.

The benchmarks studied in this paper were compiled with superblock compilation techniques for the PlayDoh architecture [9]. The benchmarks consist of 11 integer programs collectively from the SPEC-92 suite (*008.espresso*, *023.eqntott*, *026.compress*, and *072.sc*); SPEC-95 suite (*099.go* and *124.m88ksim*); and several common Unix utilities (*grep*, *lex*, *qsort*, *wc* and *yacc*). The processor model chosen is an 8-issue VLIW processor. The processor has no restrictions on the combination of instructions that may be issued each cycle, except for branches. At most one branch per cycle is allowed. Further, perfect caches and instruction latencies for the HP PA-RISC 7100 are assumed [7].

Four alternative branch prediction models are considered in the evaluation. The first is the conventional static branch prediction approach using profile information (“Profile”). The second model is dynamic branch prediction utilizing a BTB with a 2-bit saturating counter (“2bit-Ctr”). The BTB is assumed to be direct-mapped and have 1024 entries. The final models are two alternative implementations of dynamic branch prediction using two-level branch prediction. One is a PAg configuration with a direct-mapped, 1024 entry branch history table with a history length of 8 bits (“2lv-PAg(8)”). The other is a PAs configuration with a direct-mapped, 1024 entry branch history table with a history length of 10 bits and 64 pattern history tables (“2lv-PAs(10,64)”).

As it currently stands, our synthesis procedure is not fully automated. In order to reduce experimentation time, all branches in the benchmarks were not subject to the prediction synthesis. Instead, the branches were ranked according to the mispredictions that are suffered by a static predictor based on profile information. Then, the top fraction of the branches were treated by the synthesis algorithm. To permit an estimate of full performance results for the compiler synthesized approach, branches that are not subject to synthesis utilize the static prediction based on profile information. More details regarding the branches selected are presented in the next section.

An important component of the experiments for any profile-based compilation technique are inputs for each benchmark that are used for profiling (training) and those for measurement (reference). For this paper, the effectiveness of compiler synthesized branch prediction is first evaluated without any profiling uncertainties. Therefore, the reference input is chosen as the training input. After these measurements, a second set of data is presented to assess the profiling sensitivity of the technique. In this experiment, separate training and reference inputs are used.

As previously mentioned, the distance chosen to obtain the register dumps was 16 cycles. The basis for this choice is that we assume the prepare-to-branch instruction must be issued at least 12 cycles before the actual branch to provide sufficient time to communicate the branch prediction value to the processor as well as to initiate the appropriate instruction prefetch requests. Since the prepare communicates the prediction value to the hardware, the prediction computation must be performed prior to this 12 cycle distance. Using register values available 16 cycles before the branch provides a reasonably accurate view of the processor state when the actual prediction must be computed. Note that having register dumps for the exact distance at which prediction instructions will be issued is not required. Although, the prediction functions will likely lose accuracy if the register dumps are not indicative of the register values available when prediction instructions are issued.

A consequence of the 16 cycle distance and the 8-issue processor used in these experiments is that in no case could the actual branch comparison instruction be scheduled ahead of the prepare. As was described in Section 2.2, in situations where the compare can be issued before the prepare, no prediction is required since the true direction of the branch is known. However, for the processor models and branches studied in this paper, this situation never oc-

Benchmark	No. br	Br Cov (%)	Mp Cov (%)
008.espresso	14	25.9	67.8
023.eqntott	4	90.9	97.0
026.compress	6	58.9	92.8
072.sc	9	30.2	79.0
099.go	14	7.2	9.7
124.m88ksim	1	3.7	33.8
grep	5	42.1	97.8
lex	12	46.1	71.4
qsort	6	70.2	93.5
wc	4	44.7	99.9
yacc	12	24.7	62.2

Table 2: Characteristics of the branches selected for compiler synthesized branch prediction.

curred. Consequently, all results reported in the next section are based completely on prediction results.

4.2 Results

Selected branches. Table 2 presents the details of the branches selected for synthesizing branch prediction functions. As previously mentioned, for each benchmark, we treated a select subset of the branches with our algorithm, and used the static predictor to predict the remainder. Column 2 of the table shows the number of branches treated for each benchmark. Columns 3 and 4 show the dynamic fraction of branches and mispredictions with a profile-based static predictor that were accounted for by the treated branches, respectively. For instance, for *008.espresso*, the 14 branches treated accounted for 25.9% of the dynamic branches and 67.8% of the dynamic mispredictions. In general, we selected branches in order of their static misprediction rate, from most mispredicted to least, until we covered 90% of the dynamic total mispredictions. This was difficult to achieve for some of the benchmarks since the number of branches involved was large. For these benchmarks, a smaller coverage of branches predicted by compiler synthesized branch predictions is the consequence. If our technique were fully automated into a compiler, this would have not been an obstacle.

Comparison of branch prediction models. The overall performance of the compiler synthesized dynamic branch prediction method compared with other common branch prediction schemes is presented in Table 3. The branch prediction schemes compared against are: a 2-bit counter BTB, a PAg two-level predictor, a PAs two-level predictor, and a profile-based static predictor. The results in the table only consider the prediction accuracy for conditional branch instructions in the application. The last two columns of the table, marked *CS-Practical* and *CS-Theoretical*, correspond to the misprediction rate of the compiler synthesized practical and theoretical algorithms, respectively. We remind the reader that in both cases, we applied our algorithms to a select subset of the branches as outlined in Table 2, with the remaining branches being treated with the static predictor. Furthermore, the practical variant of our algorithm restricts itself to one-compare per register, with at most two registers being involved in the prediction, requiring one instruction for a one-register predictor and three instructions for a two-register predictor. The theoretical predictor is an upper bound on the performance of the synthesis algorithm, and is shown solely as a reference point.

From Table 3, it is clear that the practical compiler synthesized predictor achieves an extremely competitive prediction rate. For every benchmark tested, the compiler synthesized predictor outperforms both profile-based prediction and the 2-bit counter BTB. The most notable improvements occur for *023.eqntott* and *wc* in which the 2-bit counter misprediction rate is cut in half by the compiler synthesized predictor. The two-level predictors and the compiler synthesized predictor achieve similar performance levels. On six of the eleven benchmarks, the compiler synthesized predictor outperforms the PAg predictor. Similarly, the compiler synthesized predictor outperforms the PAs predictor on five of the eleven benchmarks. For the remaining benchmarks, the two-level predictors achieve a better misprediction rate. The major advantage the two-level and the compiler synthesized predictors have over the other predictors is exploiting correlation information to compute a more accurate prediction value. Neither the 2-bit counter nor the profile-based predictor utilize any correlation information. On the other hand, the correlation information utilized by the two-level and compiler synthesized predictors is quite different. The two-level predictors use branch history information exclusively. In contrast, the compiler synthesized predictor uses the contents of architecture registers.

Comparing the practical and theoretical misprediction rates for the compiler synthesized predictors in Table 3 shows that generally a near upper bound result can be achieved with the practical realization of the prediction function. The one real exception to this statement occurs for *026.compress*, in which a 9.808% miss rate is achieved with the practical while a 7.408% is theoretically feasible. In general, it is believed there is a large room for improvement in both the practical and theoretical performance levels by using more careful quantizer designs, and allowing a broader range of operations within the predictor.

Table 4 is similar to Table 3, but compares the performance of the various predictors only over the branches that were treated by the synthesis algorithm. In general, the performance differences between the branch prediction schemes become more apparent in this experiment. The smoothing effect of utilizing profile-based prediction for the untreated branches is removed. As a result, the performance differences between the compiler synthesized model and the 2-bit counter and profile-based static model becomes quite substantial. On average, the misprediction rate is reduced by approximately 50% with the compiler synthesized model on the selected branches. The largest improvement is observed for *124.m88ksim*, in which the misprediction rate is reduced from approximately 50% for the profile and 2-bit counter models to less than 1% with the compiler synthesized model.

Interestingly, the relative performance distinction between the two-level model and the compiler synthesized model is increased when only the selected branches are considered. The benchmarks in which each model outperforms the other model remain unchanged. However, the performance difference is noticeably larger. For example, with *072.sc* the misprediction difference has substantially increased to favor both two-level prediction schemes

Benchmark	Misprediction Rate (%)					
	2bit-Ctr	2lv-PAg(8)	2lv-PAs(10,64)	Profile	CS-Practical	CS-Theoretical
008.espresso	9.449	5.025	4.497	12.703	6.942	6.833
023.eqntott	17.561	6.115	5.455	13.381	8.214	7.961
026.compress	12.795	11.946	10.947	13.933	9.808	7.408
072.sc	6.783	2.667	1.601	11.000	5.399	5.084
099.go	26.880	28.507	25.157	24.853	24.317	23.996
124.m88ksim	5.840	4.202	2.300	5.463	3.630	3.622
grep	1.611	1.374	1.377	1.333	1.164	0.975
lex	2.268	1.413	0.964	2.660	1.863	1.659
qsort	17.652	16.619	15.568	16.249	14.117	14.011
wc	9.961	10.889	10.621	9.394	4.822	3.534
yacc	5.636	4.118	3.275	7.783	5.620	5.385
A-mean	10.585	8.443	7.433	10.796	7.809	7.315

Table 3: Comparison of the overall dynamic branch misprediction rate for all of the prediction schemes.

Benchmark	Misprediction Rate (%)					
	2bit-Ctr	2lv-PAg(8)	2lv-PAs(10,64)	Profile	CS-Practical	CS-Theoretical
008.espresso	22.472	9.580	8.752	33.302	11.036	10.614
023.eqntott	18.834	6.062	5.567	14.279	8.595	8.316
026.compress	20.393	19.363	18.092	21.958	14.955	10.880
072.sc	17.030	2.871	1.771	28.748	10.213	9.170
099.go	35.015	36.610	29.998	33.238	25.827	21.391
124.m88ksim	49.891	0.528	0.513	49.861	0.386	0.174
grep	3.748	3.173	3.130	3.097	2.695	2.245
lex	3.611	2.053	1.298	4.124	2.395	1.952
qsort	23.269	21.080	19.811	21.650	18.612	18.459
wc	22.255	24.018	23.641	20.993	10.767	7.884
yacc	12.280	8.228	6.485	19.627	10.857	9.902
A-mean	20.800	12.142	10.823	22.807	10.576	9.181

Table 4: Comparison of the dynamic branch misprediction rate across all of the prediction schemes over just the branches treated by the prediction synthesis algorithm.

for only the selected branches. In contrast, for *099.go* and *wc*, the misprediction difference is increased to distinctly favor the compiler synthesized model. Clearly, for *072.sc* branch directions correlate very strongly with branch history. In contrast, for *099.go* and *wc* the correlation of the branch directions with branch history is rather weak. Branch prediction using correlation with the contents of the register file provides distinct performance improvements. One potential strength of compiler synthesized branch prediction is the ability to tailor branch prediction functions to distinct branches to take advantage of the strongest correlations.

Effect of prediction distance. One important parameter that has been kept constant up to this point is the distance ahead of the branch in which the prediction is computed. Figure 6 shows the misprediction rate achieved with compiler synthesized dynamic branch prediction for the most frequently executed branch in *026.compress* as the branch prediction distance is varied from 1 to 64 cycles. Recall, that a 16 cycle distance has been assumed throughout all the previous experiments. The figure also contains data points for the four alternative branch prediction models for reference purposes. For these models, the specified prediction distance is not applicable. From the figure, the prediction accuracy is perfect for the compiler synthesized model for up to two cycles before the branch. At this distance, the branch comparison operands are available and can be used directly to determine the outcome of the branch.

As the distance is increased, the misprediction rate of the practical compiler synthesized model increases steadily, crossing both two-level misprediction rates between 16 and 32 cycles. This result reflects the information loss in the register file as earlier and earlier points in time are considered. However, even at 64 cycles before the branch, the compiler synthesized model achieves nearly the performance of the PAg two-level predictor and out performs both the 2-bit counter and profile models. Its encouraging to observe that the theoretical compiler synthesized predictor maintains

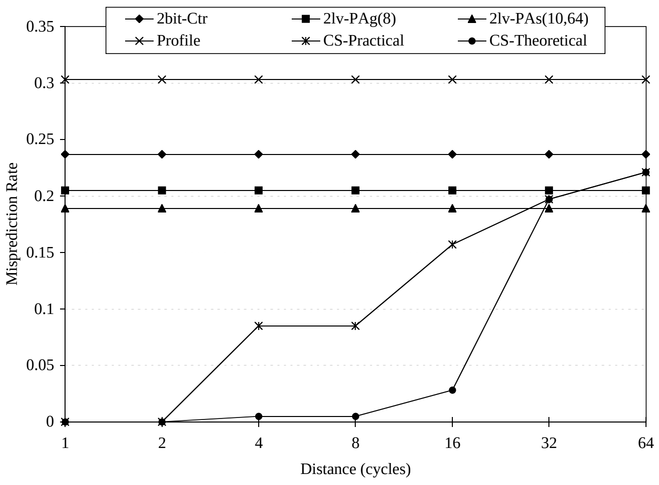

Figure 6: Effect of prediction distance on the accuracy of compiler synthesized branch prediction.

Benchmark	Static (%)	Dynamic (%)
008.espresso	0.03	8.06
023.eqntott	0.05	25.53
026.compress	0.17	14.33
072.sc	0.04	8.94
099.go	0.03	2.33
124.m88ksim	0.00	1.43
grep	0.25	9.14
lex	0.12	10.40
qsort	2.44	7.32
wc	2.15	14.65
yacc	0.16	8.69
A-mean	0.49	10.07

Table 5: Static and dynamic instruction overhead to accomplish compiler synthesized branch prediction.

nearly a 0% misprediction rate through 16 cycles for this branch. With a more sophisticated algorithm for computing practical predictors, it is expected the real performance can more closely track the theoretical performance.

Overhead of prediction synthesis. The primary overhead associated with compiler synthesized dynamic branch prediction is the additional instructions that the compiler inserts to realize the prediction functions. Table 5 presents the static and dynamic instruction overhead incurred with prediction synthesis. From the table, the static code size is not appreciably effected by the prediction instructions. On average, the prediction instructions increase the static code size by only 0.49%. The dynamic instruction count is more noticeably increased by the prediction instructions. The average increase across the benchmarks is 10.07% with the largest increases occurring for *023.eqntott*, *026.compress*, and *wc*. These large increases generally occur in cases where the benchmark contains a small loop which dominates the dynamic execution. Inserting several instructions into this loop to realize a branch prediction function substantially increases the dynamic instruction count. For larger benchmarks with a more dispersed execution pattern, a more modest increase in dynamic instruction count is observed.

Benchmark	Misprediction Rate (%)			
	Training = Ref		Training ≠ Ref	
	Overall	Selected	Overall	Selected
008.espresso	6.942	11.036	7.808	12.821
023.eqntott	8.214	8.595	8.300	8.678
026.compress	9.808	14.955	10.642	16.042
072.sc	5.399	10.213	8.784	16.121
099.go	24.317	25.827	24.752	27.521
124.m88ksim	3.630	0.386	3.720	0.386
grep	1.164	2.695	1.139	2.635
lex	1.863	2.395	2.153	2.911
qsort	14.117	18.612	14.136	18.639
wc	4.822	10.767	4.988	11.136
yacc	5.620	10.857	6.365	11.475
A-mean	7.809	10.576	8.435	11.669

Table 6: Effect of the training input on the dynamic branch misprediction rate for compiler synthesized branch prediction.

The relationship between the increase in the dynamic instruction count and the associated increase in the dynamic cycle count is highly dependent on the processor model. For our experiments, sufficient empty instruction slots were generally available to place the prediction instructions. Therefore, only negligible increases in dynamic execution cycles were observed. For a processor with fewer available instruction slots, larger increases are possible. Note that the prediction functions exclusively utilize ALU instructions. Thus, the dynamic execution overhead will be highly dependent on the available ALU execution bandwidth.

Profile sensitivity. Up to this point, all of the results presented in this section utilize the same input for reference and training. Clearly, this approach provides optimistic results for any profile-driven compilation technique. To provide a better understanding of the profile sensitivity of the synthesized prediction functions, the branch prediction accuracy results are regenerated using different inputs for reference and training. The misprediction rates for the compiler synthesized predictors are compared for different training inputs in Table 6. The first two columns of the table are the previously presented misprediction rates for the practical prediction algorithm over the entire application and over just the selected branches (column 5 of Tables 3 and 4). The last two columns correspond to the same data with an input different from the reference input used for training.

Across the benchmarks, the use of a different training input produces only small losses of accuracy for the synthesized predictor functions. As an example with *008.espresso*, the misprediction rate for the selected branch increases from 11.0% to 12.8% with the different training input. In correspondence, the overall misprediction rate increases from 6.9% to 7.8%. The loss of accuracy occurs for several reasons. First, some of the branches may behave differently across different runs of the application. Second, the register values that are correlated with branch direction in one run, may have a weaker correlation in a different run. However, in general, the synthesized prediction functions appear to be rather stable with a different training input. In fact, for all cases where the compiler synthesized predictor achieved better performance than the two-level predictors in the original experiment, this relation still holds in this experiment.

The one notable exception is *072.sc*. For this benchmark, rather large increases in the misprediction rates are observed. The prediction functions synthesized from the training input were not effective for predicting the branches with the reference input. Clearly, a more suitable training input or set of training inputs are required for this benchmark. It is important to note that by no means is this profile sensitivity experiment complete. More detailed analysis of the profile sensitivity of compiler synthesized branch prediction is required to make any definitive conclusions.

5 Concluding Remarks

A novel branch prediction technique called *compiler synthesized dynamic branch prediction* has been proposed in this paper. Our technique combines the lower cost of static compile-time prediction with the greater accuracy of dynamic prediction. Specifically, we proposed that the compiler use profile feedback to define a prediction function for each branch and insert a few explicit instructions per branch into the compiled code to compute the prediction function. These instructions are carefully selected to predict the direction of the branch using any information available during run-time. To substantiate our proposal, we presented an algorithm for selecting the prediction instructions. Using simulation experiments on a range of benchmark programs, we demonstrated the performance of the approach against contemporary static and dynamic branch prediction strategies. Our experiments showed that the performance of our algorithm is significantly better than that of the 2-bit counter BTB, and is comparable to that of the more sophisticated two-level hardware predictors.

These encouraging early results point us towards further research to improve the effectiveness of the technique. One important issue that warrants investigation is more general branch prediction functions. Currently, the algorithms are limited to a specific class of prediction functions. Extensions to support other classes may provide significant increases in prediction accuracy.

Acknowledgments

The authors thank Rick Hank and Dan Connors for all their effort developing and debugging the PlayDoh emulation tools in the IMPACT compiler; Richard Johnson for all the discussions he took part in where the concepts for this paper were developed; David August, Bob Rau, and the anonymous referees whose comments and suggestions substantially improved this paper.

References

[1] 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. To Appear *Proceedings of HPCA-3*, February 1997.

[2] T. Ball and J. R. Larus. Branch prediction for free. In *Proceedings of the ACM SIGPLAN 1993 Conference on Programming Language Design and Implementation*, pages 300–313, June 1993.

[3] B. Calder, D. Grunwald, D. Lindsay, J. Martin, M. Mozer, and B. Zorn. Corpus-based static branch prediction. In *Proceedings of the ACM SIGPLAN 1995 Conference on Programming Language Design and Implementation*, pages 79–92, June 1995.

[4] 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 Annual International Symposium on Microarchitecture*, November 1994.

[5] J. A. Fisher and S. M. Freudenberger. Predicting conditional branch directions from previous runs of a program. In *Proceedings of ASPLOS-V*, pages 85–95, October 1992.

[6] R. E. Hank, S. A. Mahlke, R. A. Bringmann, J. C. Gyllenhaal, and W. W. Hwu. Superblock formation using static program analysis. In *Proceedings of the 26th Annual International Symposium on Microarchitecture*, December 1993.

[7] Hewlett-Packard Company, Cupertino, CA. *PA-RISC 1.1 Architecture and Instruction Set Reference Manual*, 1990.

[8] W. W. Hwu, T. M. Conte, and P. P. Chang. Comparing software and hardware schemes for reducing the cost of branches. In *Proceedings of the 16th International Symposium on Computer Architecture*, pages 224–233, May 1989.

[9] W. W. Hwu et al. The Superblock: An effective technique for VLIW and superscalar compilation. *The Journal of Supercomputing*, 7(1):229–248, January 1993.

[10] V. Kathail, M. S. Schlansker, and B. R. Rau. HPL playdoh architecture specification: Version 1.0. Technical Report HPL-93-80, Hewlett-Packard Laboratories, Palo Alto, CA 94303, February 1994.

[11] J. Lee and A. J. Smith. Branch prediction strategies and branch target buffer design. *IEEE Computer*, pages 6–22, January 1984.

[12] S. P. Lloyd. Least squares quantization in PCM. *IEEE Transactions on Information Theory*, 28(3):129–137, March 1982.

[13] S. McFarling. Combining branch predictors. Technical Report TN-36, Digital Equipment Corporation WRL, Palo Alto, CA, June 1993.

[14] S. Pan, K. So, and J. T. Rahmeh. Improving the accuracy of dynamic branch prediction using branch correlation. In *Proceedings of ASPLOS-V*, pages 76–84, October 1992.

[15] J. E. Smith. A study of branch prediction strategies. In *Proceedings of the 8th International Symposium on Computer Architecture*, pages 135–148, May 1981.

[16] T. Y. Yeh and Y. N. Patt. Two-level adaptive training branch prediction. In *Proceedings of the 24th Annual International Symposium on Microarchitecture*, pages 51–61, November 1991.

[17] T. Y. Yeh and Y. N. Patt. Alternative implementations of two-level adaptive branch prediction. In *Proceedings of the 19th Annual International Symposium on Computer Architecture*, pages 124–134, May 1992.

[18] H. C. Young and J. R. Goodman. A simulation study of architectural data queues and prepare-to-branch instructions. In *Proceedings of the IEEE International Conference on Computer Design: VLSI in Computers ICCD '84*, pages 544–549, 1984.