Improving Static Branch Prediction in a Compiler

Brian L. Deitrich

Computing Systems Research Laboratory
Chicago Corporate Research Laboratories
Motorola
Schaumburg, IL 60196
Email: Brian_Deitrich-ABD022@email.mot.com

Ben-Chung Cheng* Wen-mei W. Hwu†

*Department of Computer Science
†Dept. of Electrical and Computer Engineering
and Coordinated Science Laboratory
University of Illinois, Urbana, IL 61801
Email: {bccheng, hwu}@crhc.uiuc.edu

Abstract

An ILP (Instruction-Level Parallelism) compiler uses aggressive optimizations to reduce a program's running time. These optimizations have been shown to be effective when profile information is available. Unfortunately, users are not always willing or able to profile their programs. A method of overcoming this issue is for an ILP compiler to statically infer the information normally obtained from profiling. This paper investigates one aspect of this inference: the static prediction of conditional-branch direction. The goals of this work are to utilize the source-level information available in a compiler when performing static branch prediction, to identify static-branch-prediction cases in which there is a high confidence that a branch will go in one direction at run time, to gain an intuitive understanding into the reasons why the static-branch-prediction heuristics are effective, and ultimately to improve the accuracy of the static branch prediction. The effectiveness of the static-branch-prediction heuristics developed in this paper is demonstrated on a set of programs from SPEC CINT92, SPEC CINT95, and the IMPACT compiler.

1 Introduction

An ILP (Instruction-Level Parallelism) compiler uses aggressive optimizations to reduce a program's running time. These optimizations have been shown to be effective when profile information is available [7, 10, 12, 5, 3]. Unfortunately, there are several drawbacks to profiling. First, profiling can be time consuming. This makes it unacceptable for many users. Second, profiling may not be feasible in some environments, such as real-time applications. Third, profiling accuracy relies on the behavior of the program remaining relatively constant for all possible inputs. If the program's behavior varies, poor performance after compilation may occur for some inputs. Finally, profiling must ensure that all important sections of code are executed. Otherwise, the compiler may not optimize the unexercised code sections well, if at all.

In this paper, the problem of compiling when no profiling is performed — the first and second profiling drawbacks — is dealt with by predicting the direction of conditional branches through the use of static analysis. This is termed *program-based* branch prediction because the predictions are based solely on the program's structure. If the compiler can do a good job of statically predicting conditional-branch directions, the effective application of many aggressive ILP optimizations is possible even when profiling is not performed. A description on how a compiler can deal with the third and fourth profiling drawbacks from the previous paragraph is presented in [8].

This paper addresses the following items for static branch prediction in a compiler. First, a compiler has source-level information available to it, and the static-branch-prediction heuristics should use this information. A large amount of previous work has performed static branch prediction on program traces [1, 4], but they have not had access to information from the source code. Source-level information is used extensively in this paper's loop header and pointer heuristics. Second, a compiler can benefit from knowledge about highly-confident predictions that allow strong assertions about branch directions at run time to be made. The application of control speculation is an example of a compiler optimization that can benefit from this information. The more confidence the compiler has that a branch will always go in a certain direction, the more aggressive the compiler can be when speculating instructions over that branch. Third, it is important to understand why the static-branch-prediction heuristics work well. This knowledge provides confidence that the heuristics will perform well for other benchmarks and can be used to disable

certain heuristics if the intuition is not valid. For example, it is desirable to know that a heuristic that is valid for C programs may not perform well for another programming language. All of these items allow the state of the art in static branch prediction to be improved.

The remainder of this paper is organized as follows. The next section presents the previous work that has been done in static branch prediction. Section 3 describes this paper's static-branch-prediction heuristics. Section 4 presents the results showing the effectiveness of the static-branch-prediction heuristics. Concluding remarks are given in Section 5.

2 Previous work

Early static branch prediction techniques were aimed at providing branch predictions for use by the hardware. These techniques employed several different yet simple heuristics that included assuming all branches were taken, assuming backward branches were taken and forward branches were not taken, and assuming certain branch opcodes were taken while other opcodes were not taken [14, 11]. Bandypadhyay et al., in a C compiler for the CRISP microprocessor, assumed branches in loop expressions were resolved so that the program stays in loops, and used a table lookup based on the branch opcode and operand types to determine the direction for non-loop branches [2]. They claimed that they could produce good results, but provided few details.

A different approach to static branch prediction was proposed by Ball and Larus [1]. In addition to predicting that loop branches were resolved so that the program stays in loops, they developed new heuristics to handle non-loop branches. Their major contribution was analyzing the code contained in the two targets of a branch, taken and fall-thru, in order to predict a branch direction. For example, they determined that a subroutine call located in only one branch target was usually avoided.

Several other papers have built upon the work of Ball and Larus. Hank et al. used static branch prediction to guide superblock formation [9]. Since static superblock formation was their focus, they were actually more concerned with the avoidance of hazards than predicting branch directions. Wagner et al. used a subset of the Ball and Larus heuristics, but applied them to an abstract-syntax tree representation [15]. Calder et al. trained a neural net to perform static branch prediction and used neural net inputs that were similar to those used in the Ball and Larus heuristics [4]. Finally, Patterson extended static branch prediction by propagating the possible values that operands could have at any time [13]. This allowed accurate branch prediction to occur for branches located in a `for` loop with known loop bounds.

The static branch prediction work in this paper also builds upon the work of Ball and Larus. This paper provides new intuitive understanding into the reasons why certain heuristics perform well. In addition, this work is geared toward use in a compiler. Most of the previous work has been performed on program traces, and it has not emphasized the use of static branch prediction in the compiler. Finally, this work improves the state of the art in static branch prediction by improving several heuristics and proposing other new heuristics.

3 Heuristics

This section describes the heuristics for static branch prediction that are used in this paper. These heuristics have been developed through analysis of branch behavior for a set of programs from the UNIX utilities and SPEC CINT92 benchmarks and understanding the original location of branches in the source code.

The static-branch-prediction heuristics benefit from source-level and variable-type information. This information includes knowledge of the location of a branch in the source code (e.g., is a branch derived from a conditional expression in an `if` statement or loop?), and knowledge of the branch operand types (e.g., is one of the branch operands a pointer?). For the most part, this information was not used in previous static-branch-prediction work. It was not available to many of the previous researchers because they were performing static branch prediction on program traces, and they did not have access to source-level information. The work in this paper uses the IMPACT compiler infrastructure to perform static branch prediction.

The static-branch-prediction heuristics presented in this paper are used to predict the branch direction of non-loop, non-switch branches. *Loop* branches are predicted so that loops continue to iterate. Conditional branches associated with `switch` statements, *switch* branches, are predicted so that each `case` is equally likely. More details on the prediction of loop branches and switch branches can be found in [8].

The following subsections describe the heuristics of this paper. Each subsection starts with the definition of the heuristic, and is followed by a discussion of the heuristic. For each heuristic, a comparison to the similar Ball and Larus heuristic is made.

3.1 Call

> If the successor contains a subroutine call and does not postdominate, predict that the successor is not taken. If the subroutine call is associated with I/O buffering, exiting, error processing, memory allocation, or printing, a much higher confidence in the prediction is possible.

The Ball and Larus call heuristic predicts that a successor containing any subroutine call that does not postdominate is not taken [1]. The reason they gave for this heuristic working well is that many conditional calls are used to handle exceptional conditions. While this is often true, not all subroutines executed conditionally are involved in error processing.

A study was performed to understand which subroutine calls are avoided after conditional branches and which ones are actually taken after conditional branches. This was done to determine if there are any subroutine types that allow a branch prediction to be made on a conditional branch that guards the subroutine's execution. There did not appear to be any subroutine types that are useful when determining whether a conditional branch will force the execution of a subroutine call. However, there were five subroutine types that were avoided, and therefore allow a strong prediction to be made on the conditional branch guarding their execution. These subroutine types are as follows:

I/O Buffering – The subroutine calls of *filbuf*, used with the *getchar* macro, and *flsbuf*, used with the *putchar* macro, are avoided. These subroutines are only called when the I/O buffer is empty or full, and this is a rare occurrence. These subroutine calls allow the branch associated with the `?` operator in their corresponding macro to be predicted accurately.

Exiting – Any subroutine that forces the termination of program execution is avoided. The conditional exit of a program is usually only done when an exception has occurred. In addition, when an exit occurs, the program stops executing. Therefore, the compiler should assume that the program will continue running because this is the only case where the compiler can greatly improve the program's performance.

Error Processing – Error processing is a rare event, so any subroutine associated with error processing is avoided.

Memory Allocation – Many times the conditional execution of subroutines that allocate memory is avoided. While the application of the call heuristic in this case works well, the confidence the compiler can have in the heuristic being correct is less than in the previous three cases.

Printing – Printing that can be executed conditionally, even when it is not associated with `stderr`, is avoided. As in the case with the conditional execution of memory allocation, the confidence the compiler can have in the call heuristic being correct in this case is less than for the first three subroutine types described.

The enhancement made to the call heuristic is to understand which subroutine call types are forcing the application of the call heuristic. If the subroutine call is one of the five subroutine types described above, a strong assertion about the predicted branch direction can be made. The determination of subroutine call types is accomplished by running a preprocessor on the source code to determine which functions force the unconditional execution of a subroutine call found in one of the five subroutine types. Only subroutines found in one of the lists are used in the stronger prediction of conditional branches.

The preprocessor starts off with a list of seed functions for each of the five subroutine types. If a function in the benchmark allows a subroutine located in one of the lists to be executed unconditionally, that function is added to the same list. The preprocessor runs iteratively until all the subroutines that force an unconditional execution of one of the functions in each list have been identified. The seed functions are shown in Table 1.

Table 1. Seed functions used to determine the special subroutines that are used with the call heuristic.

Subroutine Type	Subroutine Names
I/O Buffering	filbuf flsbuf
Exiting	exit abort
Error Processing	yyerror error perror fprintf (to stderr) vfprintf (to stderr)
Memory Allocation	malloc calloc realloc
Printing	printf vprintf fprintf vfprintf fflush

Unfortunately, the preprocessor's requirement that a subroutine execute unconditionally does not capture all the useful cases. This can occur when all the seed functions in user-defined error-processing functions are guarded (e.g., printing to `stderr` only occurs if a flag is set). This prevents user-defined error-processing routines from becoming new members of the error-processing function list. This case is prevalent in the benchmark *li*. In *li*, almost 20% of the non-loop, non-switch branches seen during the program's execution would be covered by the call heuristic's error-processing subroutine type if this issue was overcome. This classification is desirable because the compiler can be very aggressive with its optimizations when a classification is determined. An area of future work is to detect these situations, and allow more functions to be recognized as one of the five special subroutine types.

3.2 Loop header

> When a branch is derived from a conditional loop expression, predict that the successor is taken when it is a loop header or a loop preheader that does not postdominate the branch.

The Ball and Larus loop header heuristic is the same as the loop header heuristic used in this paper, except that it is applied to all branches guarding the execution of a loop.

```
if (j) {
      i = init;
      if (init < loop_bound)
            do {
                  do_something;
                  i++;
            } while (i < loop_bound);
}
```

(a)

```
if (j) {
      i = C1;
      if (C1 < loop_bound)
            do {
                  do_something;
                  i++;
            } while (i < loop_bound);
}
```

(b)

```
if (j) {
      i = C1;
      do {
            do_something;
            i++;
      } while (i < C2);
}
```

(c)

Figure 1. Different situations covered by Ball and Larus loop header heuristic: (a) variable initialization, (b) constant initialization and variable loop bound, (c) constant initialization and constant loop bound.

There is no check to ensure that a branch is derived from a conditional loop expression.

In order to understand this distinction, a loop transformation performed by many compilers must be analyzed. Many compilers transform `while` and `for` loops by replicating the conditional loop expression in a newly created `if` statement, using the `if` statement to guard the loop's execution, and transforming the original `while` or `for` loop into a `do-while` loop. The advantage of this approach is that an unconditional branch can be removed from the loop.

The Ball and Larus loop header heuristic gets applied to either a branch that originally preceded the loop or a branch in an `if` statement that is derived from the original conditional loop expression. The different cases covered by the Ball and Larus loop header heuristic are shown in Figure 1. This figure has three different cases that are mutually exclusive, and cover all the cases caught by the original Ball and Larus loop header heuristic. The branches that have the Ball and Larus loop header heuristic applied to them are in bold in the figure.

Figure 1(a) shows the case where the `for` loop initialization is variable. In this case, the derived `if` statement, "`if (init < loop_bound)`," cannot be optimized away. Figure 1(b) is similar to case (a) because the derived `if` statement, "`if (C1 < loop_bound)`," is not optimized away. The difference is that the `for` loop initialization is a constant. Case (b) is distinguished from case (a) because case (b) can have a stronger assertion made about the loop header branch. Since the initialization is constant, the loop is more likely to be executed at least once.

Figure 1(c) is different from the previous two cases because the derived `if` statement can be optimized away. In this case, another `if` statement, "`if (j)`," located before the original loop in the source code, has the Ball and Larus loop header heuristic applied to it. However, this branch does not have to be related to the loop at all, and it may not be biased so that the loop is entered.

In this paper, only cases (a) and (b) have the loop header heuristic applied to them. In addition, cases (a) and (b) are distinguished because case (b) has a better chance of being correct. This is useful when determining how aggressive a compiler optimization can be with code located near a branch where the loop header heuristic is applied.

3.3 Pointer

> If a branch compares a pointer to NULL or compares two pointers, predict that the pointers are not equal as long as these pointers are not part of an array.

The Ball and Larus pointer heuristic is the same as the pointer heuristic used in this paper, except that a pointer operand that is part of an array does not get the pointer heuristic applied to its branch in this paper. This exception is made because an element in an array of pointers has a higher likelihood of being NULL than a pointer that is not part of an array. This is definitely true if the array of pointers is sparsely populated and if the array contains only a few non-NULL members.

An advantage that the pointer heuristic used in this paper has over most previous branch prediction work is that source-level information is available when applying the heuristic. Ball and Larus, along with Calder et al., used binary instrumentation to gather statistics and applied the branch prediction heuristics to these program traces. Because of this, they did not have access to source-code information to know which branch operands were pointers. Therefore, they looked for code patterns to determine whether branch operands were pointers, and these patterns may have captured some spurious cases that were not associated with a pointer comparison. The source-level information used in this paper allows the pointer heuristic to perform better.

The pointer heuristic works because pointers do not usually equal NULL or one other. While this is definitely not always the case, our results indicate it to be true more often than it is false.

3.4 Return

> Predict that the successor that contains a return is not taken.

The Ball and Larus return heuristic is identical to the return heuristic used in this paper. This heuristic seems to work well because of the way that most programmers use conditional return statements. Returns are often used to exit a function early when an error or boundary case is detected. In these cases, the return statement is not likely to be taken.

3.5 Opcode

> Predict that a comparison of an integer less than zero, less than or equal to zero, or two floating point values being equal fails. Predict that a comparison of an integer greater than zero or greater than or equal to zero succeeds.

The Ball and Larus opcode heuristic is identical to the opcode heuristic used in this paper. This heuristic seems to work well because integers normally contain positive numbers. Many times this heuristic can be applied simultaneously with the loop header heuristic, return heuristic, and call heuristic. The overlap with the loop header heuristic occurs in counting loops that decrement. In this situation, a loop header branch that does a comparison against zero is usually present. The return heuristic and call heuristic overlap occurs when negative numbers are used to denote error conditions.

3.6 Character comparison

> If a branch is an integer comparison against a character constant, predict that the comparison fails.

The character comparison heuristic is a new heuristic defined in this paper that is based on the fact that ASCII characters are usually not equal to one particular constant character value. For example, in English text, the most common character is a space, and it occurs less than half of the time.

The heuristic is applied to *branch if equals* (`BEQ`) and *branch if not equals* (`BNE`) branches that contain one operand that is a character constant. Unfortunately, the EDG front end used in IMPACT converts all character constants to integer constants. This makes it impossible to distinguish constant character comparisons and regular constant integer or enumerated type comparisons. The integer constants checked are –1 (EOF), 9 ('\t'), 10 ('\n'), and 32 (' ') thru 126 ('~'). This implementation ends up capturing some extraneous cases that are not involved with characters, but that share constant values that overlap with these common ASCII character values. The check against –1 does well even when a program is not working with ASCII characters because it is a common method of marking an error.

3.7 Restricted opcode AND

> If a comparison is made against one bit in a bit field, predict that the bit is not set.

The restricted opcode AND heuristic is a new heuristic defined in this paper that works by assuming that one particular bit in a bit field is normally not set. The intuition is that many times when bit fields are used, they are set in the minority case. This heuristic is one of the weakest heuristics used, but our results show that it works reasonably well.

The heuristic looks for a `BEQ` or `BNE` branch that contains a constant zero as one of its operands. For the heuristic to apply, the other operand must have resulted from a preceding AND operation, and one operand of that AND must be a constant, power-of-two value.

3.8 Other Ball and Larus heuristics

Two heuristics described by Ball and Larus are not used in the branch prediction heuristics of this paper. These heuristics are the guard and store heuristics. These heuristics are not used because an analysis of the cases where these heuristics applied did not reveal any intuition that explained the success of the heuristics.

The Ball and Larus guard heuristic states that if one of the branch's operands is used in a branch's target, assume the target containing the operand is taken. This heuristic assumes that a condition is being tested for, and the normal case allows the operations using the operand to execute. The most common situation for this occurs with a pointer check against NULL. Only if the pointer is non-NULL, is a load from the pointer allowed. However, this case is already captured by the pointer heuristic. Since the most common case for this heuristic ends up being a subset of the pointer heuristic, this heuristic is not applied in this paper.

The Ball and Larus store heuristic states that if one of the branch's targets contains a store, assume that the other target is taken. No intuition has been provided on why this heuristic works, and no potential intuitive reason for the success was detected when examining the cases where this heuristic performed well. Calder et al. found that this heuristic was wrong more than it was correct for the C programs evaluated in their paper [4].

4 Results

This section presents the results of static branch prediction. The results include an in-depth analysis of the individual heuristics in order to provide insights into the decisions made when defining the heuristics used in this paper. For this work, both the heuristics used in this paper and the heuristics described by Ball and Larus are implemented in the IMPACT compiler [6].

The IMPACT intermediate-representation files that are used for prediction have gone through a traditional compilation path. The C source code is read in by IMPACT,

translated into the intermediate representation, and optimized using IMPACT's classical optimizations. After optimization, the code is profiled. The results of profiling do not affect the actual static branch predictions, but profiling does allow the success of the branch prediction to be easily measured.

The branches in a program can be broken down into three categories: loop branches, switch branches, and non-loop, non-switch branches. Loop branches are loop back-edge and loop exit branches that control the iteration of loops. Switch branches are branches that are derived from `switch` statements in the source code. The remaining branches are non-loop, non-switch branches. The non-loop, non-switch branches are evaluated in this analysis. These branches are difficult to predict, and they represent a large portion of the branches that are dynamically executed.

The benchmarks used in this analysis consist of *espresso* and *sc* from SPEC CINT92, all the benchmarks from SPEC CINT95, and *Pcode* and *Lhppa* from the IMPACT compiler.[1] Even though the results are presented for these benchmarks, most of the analysis performed in developing the static-branch-prediction heuristics was performed on a different set of benchmarks. These benchmarks included nine Unix utilities (*cccp*, *cmp*, *eqn*, *grep*, *lex*, *qsort*, *tbl*, *wc*, and *yacc*), the six benchmarks present in SPEC CINT92, and *alvinn* and *ear* from SPEC CFP92. These benchmarks represent the training set used to understand and evolve the static branch prediction heuristics. There is some overlap in the benchmarks used during training and the benchmarks used for generating the results, but a majority of the benchmarks used for the results were not present in the training set. A training set was purposely used when developing these heuristics in order to not develop a set of "tuned" heuristics that perform well for the benchmarks that are analyzed, but do not perform well for other programs.

Because of space constraints, the success of the heuristics in predicting a branch direction is not evaluated. Instead, an in-depth analysis of the individual heuristics is presented. An analysis of the effectiveness of the Ball and Larus heuristics and this paper's heuristics to predict branch directions is presented in [8].

An evaluation of the static-branch-prediction heuristics that are enhanced in this paper is shown in Table 2. For the three enhanced heuristics, the results of different subcases are shown. The results are based on the dynamic behavior observed when the code is actually executed. A branch gets counted every time that it is executed during a program run. The results of each subcase are shown with three numbers. The number on the left of each column represents the percentage of non-loop, non-switch branches executed dynamically that have the heuristic subcase applied to them. If a subcase is applied to less than 1% of the non-loop, non-switch branches, the entire column entry is blank. The two numbers on the right of each column represent the effectiveness of the predictions. The number to the left of the slash contains the miss rate for a perfect predictor, which always predicts the most-frequently-taken branch direction, and the number to the right of the slash contains the miss rate based on the heuristic decisions. A heuristic miss rate less than 50% indicates that the heuristic predicted a majority of the branches correctly. A good heuristic has a heuristic miss rate that approaches the perfect miss rate, which is the best possible miss rate.

The call heuristic is evaluated first. The three subcases are mutually exclusive and represent all the cases in which the call heuristic is applicable. The *error* column represents the combined results of the I/O buffering, exiting, and error processing subcases; the *special* column represents the cases in which the memory allocation and printing subcases apply; and *other* represents all other call heuristic cases. The cases present in the *error* and *special* columns are obtained using the techniques described in Section 3.1, and they are broken out so more confident predictions can be made. As seen in the table, the *error* case is very biased and always correctly predicted. The *special* case is also biased, but not as much, and it is predicted correctly most of the time. The *other* case is not as biased and is correctly predicted less often than the two other subcases.

The benchmark *li* would benefit from a more sophisticated preprocessor, as is described at the end of Section 3.1, that enables more error processing functions to be determined. If a better preprocessor was built, at least 18% of the cases in the *other* column could be put into the *error* column. The behavior of the heuristic for the branches that would move is very biased — both the perfect and heuristic miss rates would be 0. The behavior of the remaining branches in the *other* column would be worsened — the perfect miss rate would be 13 and the heuristic miss rate would be 43.

The loop header heuristic has three subcases that are mutually exclusive and represent all the cases in which the Ball and Larus loop header heuristic apply. The *V-I* column represents the variable-initialization subcase; the *C-I* & *V-LB* column represents the constant-initialization, variable-loop-bound subcase; and *C-I* & *C-LB* represents the constant-initialization, constant-loop-bound subcase. The definition of each of these subcases is found in Section 3.2. The *V-I* and *C-I* & *V-LB* subcases are captured by both the Ball and Larus and this paper's loop header heuristic. The cases are distinguished in this paper because the *C-I* & *V-LB* subcase is predicted correctly more often than the *V-I* subcase. The *C-I* & *C-LB* subcase is captured by the Ball and Larus heuristics, but ignored by this paper's heuristics. This subcase is excluded in this paper because the branches associated with this subcase are mispredicted more often than they are predicted correctly. As seen in the table, the *C-I* & *C-LB* subcase is mispredicted a majority of the time in five out of eight benchmarks.

Finally, the pointer heuristic is evaluated. The two subcases are mutually exclusive and represent all the cases in which the Ball and Larus pointer heuristic apply. The *non-array* column represents the cases in which the pointer is not obtained from an array, and this accounts for all the

[1] The IMPACT code comes from the version 3.0 beta release dated June 10, 1997, which was made available to several universities outside of the University of Illinois.

Table 2. Evaluation of the static-branch-prediction-heuristic enhancement cases.

	Call (Section 3.1)						Loop Header[a] (Section 3.2)						Pointer (Section 3.3)			
Benchmark	Error[b]		Special[c]		Other		V-I		C-I & V-LB		C-I & C-LB		Non-Array		Array	
espresso					2	5/44	5	1/2	10	0/0	5	10/84	3	2/3		
sc			1	5/9	19	12/21	8	1/9					5	32/56	12	31/47
go					15	20/46	9	11/15	1	16/23	6	29/34				
m88ksim					12	10/36	1	4/21			2	9/11	4	5/12		
gcc	2	0/0			19	13/40	9	10/48	3	8/14	2	22/67	10	11/38	3	18/57
compress					25	25/44					4	11/77				
li					44	8/27	21	0/10			10	30/59	32	3/17		
ijpeg					14	7/14	8	24/68	1	0/0	2	3/85				
perl	2	0/0	3	0/0	24	3/14	2	32/65			2	0/0	17	0/46		
vortex					40	0/20							1	1/12	2	0/0
Pcode	7	0/0	5	7/20	20	5/28	7	16/31					22	21/48		
Lhppa	23	0/0	10	3/8	31	15/36	7	6/17	4	5/6			32	7/18	12	24/58

[a] C-I = constant initialization, V-I = variable initialization, C-LB = constant loop bound, V-LB = variable loop bound

[b] Includes instances of call heuristic associated with I/O buffering, exiting, and error processing.

[c] Includes instances of call heuristic associated with memory allocation and printing.

applications of this paper's pointer heuristic. The *array* column represents the cases in which a pointer is obtained from an array. More details on the pointer heuristic are present in Section 3.3. As seen in the table, the non-array pointer subcase is predicted correctly a majority of the time in eight out of nine benchmarks. The array pointer subcase does not perform as well as the non-array pointer subcase. It is mispredicted a majority of the time in two out of four benchmarks.

An evaluation of the heuristics unique to one set of the heuristics — either the Ball and Larus or this paper's heuristic sets — is shown in Table 3. The numbers in each column have the same meanings as the numbers found in Table 2.

The character comparison and restricted opcode AND heuristics are present in only in this paper's heuristics. They are described in Sections 3.6 and 3.7. The character comparison heuristic is predicted correctly most of the time in all of the benchmarks. The restricted opcode AND heuristic is predicted correctly in six out of seven benchmarks.

The guard and store heuristics are present in only the Ball and Larus heuristics. These heuristics are described in Section 3.8. The guard heuristic is broken down into two subcases: *pointer* (when the pointer heuristic also applies) and *other* (when the pointer heuristic does not apply). The pointer heuristic already captures the cases present in the *pointer* column. The *other* column subcase does not perform as well as the *pointer* subcase. The *other* subcase is mispredicted in 6 out of the 12 benchmarks. The store heuristic does not perform well. It is predicted correctly in 7 out of 12 benchmarks, but it has an average heuristic miss rate of close to 50%.

5 Conclusions and Future Work

This paper has investigated static branch prediction from the compiler's perspective. This work used source-level information to help with the static-branch-prediction decisions and allowed subcases to be determined that give the compiler a high confidence that a branch will go a certain direction at run time. These subcases enable a compiler to be aggressive with its optimizations. In addition, this work provided insights into the reasons for the effectiveness of the static-branch-prediction heuristics. Finally, this work created two new heuristics, character comparison and restricted opcode AND, and enhanced several others.

A major area of future work involves the integration of the static-branch-prediction decisions into the compiler's heuristics. This information should be useful when profiling is not performed. How this information is used and how the high-confidence subcases can be exploited needs to be investigated. The use of this information should allow the compiler to generate code that is more efficient than would otherwise be possible.

Another area of future work is to improve the preprocessor that determines which functions should be included in each of the subroutine types of the call heuristic. The current implementation only allows the unconditional execution of functions already contained in a category to add the caller function to the same category. The improvement of this preprocessor is important because the subroutine categories allow the compiler to have a high confidence in the branch's run-time behavior.

6 Acknowledgments

The authors would like to thank Brian Deitrich's dissertation committee (Wen-mei Hwu, Scott Mahlke, Bill Sanders, and Ben Wah) and Carole Dulong from Intel

Table 3. Evaluation of heuristics included in only one heuristic set.

Benchmark	Character Comparison (Section 3.6)		Restricted Opcode AND (Section 3.7)		Guard (Section 3.8)				Store (Section 3.8)	
					Pointer		Other			
espresso	14	0/0	39	18/31			21	10/20	21	21/31
sc			7	5/32	9	25/34	20	25/53	11	4/59
go	2	25/43					33	22/31	31	20/45
m88ksim	3	3/4	2	16/30			19	8/26	11	10/45
gcc	25	11/34			4	14/28	32	11/37	20	10/43
compress							42	15/20	36	6/70
li			11	37/45	20	0/2	4	28/58	34	14/50
ijpeg							28	11/64	11	3/46
perl			15	4/26	5	0/0	29	9/68	29	11/70
vortex			9	0/100			28	1/74	40	0/30
Pcode	13	8/10			13	2/8	30	12/34	23	7/36
Lhppa	4	2/16	2	14/39	10	2/11	23	16/52	20	5/54

whose input during Brian's PhD work was invaluable. The authors would also like to thank the members of the IMPACT compiler team and the anonymous referees whose comments and suggestions helped to improve the quality of this paper. This research has been supported by the National Science Foundation (under grant CCR96-29948), Intel, Advanced Micro Devices, Hewlett-Packard, and NCR.

References

[1] 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.

[2] S. Bandyopadhyay, V. S. Begwani, and R. B. Murray. Compiling for the crisp microprocessor. In *Proceedings of the IEEE Spring Compcon 87*, pages 96–100, February 1987.

[3] R. A. Bringmann. *Compiler-Controlled Speculation*. PhD thesis, Department of Computer Science, University of Illinois, Urbana, IL, 1995.

[4] 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.

[5] P. P. Chang, S. A. Mahlke, W. Y. Chen, and W. W. Hwu. Profile-guided automatic inline expansion for C programs. *Software Practice and Experience*, 22(5):349–370, May 1992.

[6] P. P. Chang, S. A. Mahlke, W. Y. Chen, N. J. Warter, and W. W. Hwu. IMPACT: An architectural framework for multiple-instruction-issue processors. In *Proceedings of the 18th International Symposium on Computer Architecture*, pages 266–275, May 1991.

[7] P. P. Chang, S. A. Mahlke, and W. W. Hwu. Using profile information to assist classic code optimizations. *Software Practice and Experience*, 21(12):1301–1321, December 1991.

[8] B. L. Deitrich. *Static Program Analysis to Enhance Profile Independence in Instruction-Level Parallelism Compilation*. PhD thesis, Department of Electrical and Computer Engineering, University of Illinois, Urbana, IL, 1998.

[9] 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*, pages 247–255, December 1993.

[10] W. W. Hwu, S. A. Mahlke, W. Y. Chen, P. P. Chang, N. J. Warter, R. A. Bringmann, R. G. Ouellette, R. E. Hank, T. Kiyohara, G. E. Haab, J. G. Holm, and D. M. Lavery. The Superblock: An effective technique for VLIW and superscalar compilation. *The Journal of Supercomputing*, 7(1):229–248, January 1993.

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

[12] S. A. Mahlke, D. C. Lin, W. Y. Chen, R. E. Hank, and R. A. Bringmann. Effective compiler support for predicated execution using the hyperblock. In *Proceedings of the 25th International Symposium on Microarchitecture*, pages 45–54, December 1992.

[13] J. R. C. Patterson. Accurate static branch prediction by value range propagation. In *Proceedings of the ACM SIGPLAN 1995 Conference on Programming Language Design and Implementation*, pages 67–78, June 1995.

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

[15] T. A. Wagner, V. Maverick, S. L. Graham, and M. A. Harrison. Accurate static estimators for program optimization. In *Proceedings of the ACM SIGPLAN 1994 Conference on Programming Language Design and Implementation*, pages 85–96, June 1994.