Optimal 2-Bit Branch Predictors

Ravi Nair, *Fellow, IEEE*

Abstract—This paper presents an efficient technique to analyze finite-state machines to determine an optimal one for branch prediction. It also presents results from using this technique to determine optimal 4-state branch predictors for applications in the SPEC89 benchmark suite running on the IBM RS/6000. The paper concludes that the simple 2-bit counter is the only machine that performs consistently well and close to the optimal over all applications.

Index Terms—Branch instructions, dynamic branch prediction, optimal 2-bit predictors, trace analysis, 2-bit counter machine, 2-bit pattern machine.

I. INTRODUCTION

The penalty due to branch instructions in an instruction stream has been well documented in the literature (see, for example [1] and [2]). Pipelined machines suffer a degradation in performance whenever branches cause a disruption in the smooth issue of instructions to the functional units. A disruption occurs when either the condition or the target address of a conditional branch cannot be resolved in time to fetch and issue the target instructions.

Branch prediction is a common technique used to reduce branch penalty. Branch prediction essentially involves a guess of the likely stream of instructions that need to be executed after a branch; disruption in the pipe is thus avoided whenever the guess is accurate. For branches whose target addresses are unknown at compile time, such a guess is hard, but for most common conditional branches the likely path of the program can be guessed by simply guessing whether or not that branch will be taken. If information is available about the likely direction of the branch at compile time, the compiler could indicate this information, either by setting a bit in the branch instruction [3], [4], or by arranging code so that the more likely path is the fall-through path.

In comparison to these static techniques, dynamic branch prediction is implemented in hardware by saving some history of the behavior of the application to facilitate prediction of future branches. A simple technique is to store, for each conditional branch, one bit indicating whether the branch was taken or not taken at last encounter and predict the same behavior at the next encounter.

Studies (e.g., [2]) have indicated that better results can be obtained by using two or more bits to represent the history of each branch, with the cost-effectiveness diminishing rapidly beyond three bits. In order to improve on the prediction percentage beyond that obtained by simply recording the history of each branch in a fixed number of bits, it is necessary to use more sophisticated techniques such as the *two-level branch prediction scheme* [5]. In this scheme, the recent history of each branch is recorded and is used as an index to a table which predicts whether the branch should be taken or not. If this table is static (determined, for example from a profiling run, as suggested in [2]), it is referred to as a *static training predictor*, whereas, if it is dynamically updated, it is referred to as a *two-level adaptive predictor* [5], [6]. As explained in these papers, these schemes, particularly the adaptive ones, tend to have considerably fewer mispredictions compared to the simple *n*-bit history schemes.

In another form of two-level adaptive predictor, called the *correlation-based predictor*, proposed in [7], a global branch history is used to index into a table associated with each branch. A thorough analysis of the relative cost and performance of the two-level schemes appears in [8].

The two-level predictors are more expensive to implement compared to, say, a 2-bit or 3-bit history scheme. In high performance processors which have resources for executing several instructions in a single clock cycle, and where the hardware cost of implementing them can be tolerated, these predictors help significantly in improving performance. When space on a chip is at a premium, however, the simple schemes provide a good cost-effective way of reducing the penalty due to conditional branches. For example, on the IBM RS/6000, for the SPEC benchmark *espresso*, a history table with 256 2-bit entries reduces the penalty due to conditional branches from 0.176 cycles per instruction (cpi) to 0.051 cpi [9]. Reducing the penalty further (perhaps by half) using one of the two-level predictors would more than double the hardware required.

An additional motivation for studying simple 2-bit predictors is that they often form a part of more sophisticated predictors. For example, both in Yeh and Patt's work [5], [6], [8], as well as in the work by Pan, et al. [7], the second level pattern history is kept in a simple 2-bit form.

Given a limited number of bits (say two or three) per branch, it then becomes interesting to ask what the best representation of the history is. Previous studies have simulated some of the more intuitive ways of using two bits to represent the state of a branch. The number of finite-state machines having four states (and hence representable in two bits) is rather large, suggesting that it is impractical to look for the optimal 2-bit representation. In this paper we demonstrate that while this number is indeed large, there exist techniques to reduce the effort in simulating all possible 4-state finite state machines for branch prediction. We also provide results of such an evaluation for various commonly used benchmarks.

II. ENUMERATION OF ALL 2-BIT PREDICTORS

Our principal objective is to determine a 4-state machine which when used to represent the state of each branch in a given application maximizes the success in predicting the outcome of the next occurrence of the branch. Each branch will use the same machine so that the hardware requirement is simply a table of $2n$ bits, where n is the number of distinct branches, in addition to one 4-state sequential machine. We are focusing here on determining the optimal from a large set of machines; we will not consider the effects of limiting the size of the table. The reader is referred to [9] for more results from a study of practical implementations for the IBM RS/6000.

A 2-bit predictor for a conditional branch instruction is essentially a Moore machine having four states, one input and one output. An input value of 0 indicates that the current branch was not taken, and 1 indicates it was taken. Each state is associated with an output value, 1 indicating that the next occurrence of the branch should be predicted taken, and 0 indicating not taken. Each state also has a specified next state for each of its two input values. Simply speaking, there are four ways to assign each of eight next states, and two ways to assign an output to each of the states, giving us $4^8 * 2^4$ or 2^{20} possible machines. Evaluating these million machines for each of the benchmark traces is not practical even by current computing standards.

In the machine of Table I, the next state (NS) is depicted as a function of the present state (PS) and the actual outcome (I) of a

branch. Each state is also associated with a value that predicts the outcome of the next instance (O) of the branch. Also shown is a graphical representation of the machine. The starting state is represented by a bold arrow, states having a dark outline predict "taken" (O = 1), light predict "not-taken," solid arcs represent 1-transitions, broken represent 0-transitions. A convenient representation for the machine would be simply (BCBAADCD : 3), where the letters correspond to the next states for each of the states A, B, C, D in the table, and where the 4-bit binary representation of the number 3 characterizes the output (O) column of the table.

TABLE I
SAMPLE FINITE STATE MACHINE

PS	NS		O
	I = 0	I = 1	
A	B	C	0
B	B	A	0
C	A	D	1
D	C	D	1

The first step in reducing the number of machines simulated is to ignore the output for each state during simulation of a given finite-state machine (FSM). We simply record the percentage of taken branches for each state. Clearly the best among the 2^4 possible machines having the same next-state transitions will be one which assigns a 0 to a state if less than half the branches are taken in that state, and 1 if more than half are taken in that state. Thus, if the taken fractions for each state during simulation were A: 0.8, B: 0.2, C: 0.3, and D: 0.7, then the machine with an O assignment of 1001 would be the best among all machines having the same NS assignments.

Among the 4^8 machines are several that are either trivial, have fewer than four equivalent states, or are equivalent to other FSMs, even ignoring output assignment. We will now show by simple combinatorial techniques that, for a given starting state there are no more than 5,248 distinct machines having exactly four distinct (nonequivalent) states.

Consider an FSM with S states, and hence S rows in its table description. We need to consider only those machines which have the first row representing the starting state, and for which a new state appears in a next state entry only if all previous states (in row order) have already appeared in the table. For all $i < S$, the first i rows must then lead to j states, $j > i$. (If $j \leq i$, the $i + 1$th state will be unreachable). These j states must be represented by the first j rows in the table description. We can now write a recurrence relation for the number of machines $n_{i,j}$ for which the first i rows lead to exactly j states:

$$n_{i,j} = n_{i-1,j-2} + (2j-1)n_{i-1,j-1} + j^2 n_{i-1,j} \text{ for } i < j, \ j < S.$$

The first term in the above equation comes from the fact that if the first $i - 1$ rows lead to $j - 2$ states, then the ith row must have transition on 0 going to state $j - 1$, and transition on 1 leading to state j. The second term indicates that row i must have at least one of its transitions going to state j. The last term comes from the fact that since all j states have been touched already there are j^2 ways to assign the ith row.

The initial conditions are given by:

$$n_{1,1} = 0, n_{1,2} = 3, n_{1,3} = 1, \text{ and } n_{1,j} = 0, j > 3.$$

The total number of machines with S states is obtained as $S^2 * n_{S-1,S}$. Table II shows the various values of $n_{i,j}$ for a 4-state machine.

TABLE II
CALCULATION OF NUMBER OF DISTINCT 2-BIT PREDICTORS

i	j			
	1	2	3	4
1	0	3	1	0
2	0	0	5*3 + 9*1 = 24	1*3 + 7*1 = 10
3	0	0	0	7*24 + 16*10 = 328
4	0	0	0	16*328 = 5248

Even among these machines there are some uninteresting ones, e.g., machines having the final state leading back to itself under both 0 and 1 input. But we will not attempt to prune this number further.

Simulating all machines with eight states (three bits) appears unreasonable since the above formula indicates that there are more than 12 billion such machines. However, it is more practical to simulate all 4-state FSMs for a suite of applications, as we did. We provide below a program which lists all 5,248 distinct 4-state machines.[1]

```
#include <stdio.h>
int S0[5][5][10000][5]; /* next state for I = 0 */
int S1[5][5][10000][5]; /* next state for I = 1 */
/* Array indices: Table 2 row, table 2 col,
    machine #, present state */
void Copy(int ai,int aj,int an,int bi,int bj,int
  bn) {
 /* Copies machine number 'an' from row ai, column
    aj of Table 2 to machine number 'bn' from row bi,
    column bj.
 */
 int i;
 /* Next states are defined only for states 1..ai
    */
 for (i = 1; i <= ai; i++) {
   S0[bi][bj][bn][i] = S0[ai][aj][an][i];
   S1[bi][bj][bn][i] = S1[ai][aj][an][i];
 }
}
void main() {
 int i,j,k,p,q;
 int n[5][5];
 for (j = 0; j <= 4; j++) {
   n[0][j] = 0;
 }
 for (i = 1; i <= 4; i++) {
   for (j = 0; j <= i; j++) {
     n[i][j] = 0;
   }
 }
 /* Initializing machines in (1,2) and (1,3) */
 S0[1][2][0][1] = 1;  S1[1][2][0][1] = 2;
 S0[1][2][1][1] = 2;  S1[1][2][1][1] = 1;
 S0[1][2][2][1] = 2;  S1[1][2][2][1] = 2;
 n[1][2] = 3;
 S0[1][3][0][1] = 2;  S1[1][3][0][1] = 3;
 n[1][3] = 1;
 n[1][4] = 0;

 /* Deriving machines for rows 2 and 3 */
```

---

[1] We believe our number, 5,248, is the right number of distinct 4-state machines compared to the number 5,428 obtained in another independent work [10] brought to our attention by Chriss Stephens of CMU after this work had been completed.

```
for (i = 2; i <= 3; i++) {
  for (j = i+1; j <= 4; j++) {
    n[i][j] = 0;
    for (k = 0; k < n[i-1][j-2]; k++) {
      Copy(i-1,j-2,k,i,j,n[i][j]);
      S0[i][j][n[i][j]][i] = j-1;
      S1[i][j][n[i][j]][i] = j;
      n[i][j]++;
    }
    for (k = 0; k < n[i-1][j-1]; k++) {
      for (q = 1; q <= j-1; q++) {
        Copy(i-1,j-1,k,i,j,n[i][j]);
        S0[i][j][n[i][j]][i] = q;
        S1[i][j][n[i][j]][i] = j;
        n[i][j]++;
      }
      for (q = 1; q <= j; q++) {
        Copy(i-1,j-1,k,i,j,n[i][j]);
        S0[i][j][n[i][j]][i] = j;
        S1[i][j][n[i][j]][i] = q;
        n[i][j]++;
      }
    }
    for (k = 0; k < n[i-1][j]; k++) {
      for (p = 1; p <= j; p++) {
        for (q = 1; q <= j; q++) {
          Copy(i-1,j,k,i,j,n[i][j]);
          S0[i][j][n[i][j]][i] = p;
          S1[i][j][n[i][j]][i] = q;
          n[i][j]++;
        }
      }
    }
  }
}

/* Deriving machines for (4,4) from (3,4) */
i = j = 4;
for (k = 0; k < n[i-1][j]; k++) {
  for (p = 1; p <= 4; p++) {
    for (q = 1; q <= 4; q++) {
      Copy(i-1,j,k,i,i,n[i][j]);
      S0[i][j][n[i][j]][i] = p;
      S1[i][j][n[i][j]][i] = q;
      n[i][j]++;
    }
  }
}
/* Printing out machines */
for (i = 4; i <= 4; i++) {
  for (j = 4; j <= 4; j++) {
    for (k = 0; k < n[i][j]; k++) {
      printf("\n");
      for (q = 1; q <= i; q++) {
        printf("%d%d,"S0[i][j][k][q],
          S1[i][j][k][q]);
      }
    }
  }
}
```

III. EXPERIMENTAL RESULTS

Our experiments were performed on the IBM RS/6000. Details about the architecture of the RS/6000 may be found in [11]. More details about different types of branch instructions on the RS/6000, and about the behavior of branches on SPEC89 benchmarks may be found in [9]. For the present study we gathered representative traces from each of the SPEC benchmarks. Four of these benchmarks, *nasa7*, *matrix300*, *fpppp*, and *tomcatv* appeared to cause negligible disruption due to branches and were ignored. Efficient analysis was facilitated by extracting only the relevant branch information from the traces and storing them in a compressed form. Only conditional
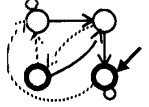
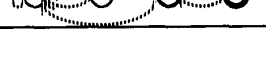branches were considered. Some of the conditional branches, e.g., decrement the count register, and branch according the register value being zero or non-zero, are already handled effectively by special branch-preprocessing hardware, and hence, were ignored. Thus, the branches analyzed were truly only the "difficult" branches. Given in Table III is a summary of the traces.

TABLE III
TRACE STATISTICS

Benchmark	Trace length (instructions)	"Difficult" conditional branches (% instructions)	Branches taken (% "difficult" conditional branches)
spice2g6	38,741,662	6.41	35.03
doduc	36,857,194	6.13	31.13
gcc1.35	19,362,014	14.05	50.37
espresso	24,062,240	15.20	41.55
li	3,108,199	13.81	38.28
eqntott	4,378,967	13.04	20.61

Our experimental results are shown in the Table IV. For each benchmark we show the best prediction percentage and the finite state machine that resulted in that prediction.

TABLE IV
OPTIMAL 2-BIT PREDICTOR FOR EACH APPLICATION

Benchmark	Best prediction %	Best predictor
spice2g6	97.2	
doduc	94.3	
gcc1.35	89.1	
espresso	89.1	
li	87.1	
eqntott	87.9	

For *doduc* and *espresso*, the optimal finite state machines are 2-bit "counters" first proposed in [1]. This machine counts up from state 0 to state 3 when it sees consecutive 1s. It counts down from state 3 to state 0 when it sees consecutive 0s. States 2 and 3 predict the next branch to be taken, while states 0 and 1 predict it to be not taken. The optimal machine for *gcc* is also a counter, except that the starting state is different. The optimal machines for *li* and *eqntott* are slight variations of the counter. For both these machines, there is only one state where the branch is predicted taken.

The only machine which is fundamentally different in structure is the optimal machine for *spice*. A closer examination reveals that this machine is a variation of the "pattern" machine. The pure pattern machine, suggested in [2], simply predicts the next branch to be taken if the last two branches were both taken, or if the last two

branches were "taken" followed by "not-taken." In the last case, the pattern machine is guessing that there is an alternating sequence of "taken" and "not-taken" conditions for the branch.

Table V lists the 20 machines that were consistently close to the optimal for all applications. All these machines are variations of the counter. In fact, the top 16 machines are simply different versions of five machines shown in Table VI with different starting states. The fourth machine in Table VI is the same as the predictor proposed for the S-1 machine as quoted in [2]. All machines in this table predict to within 0.5% success rate of each other.

It is interesting to note that the pattern machine does not appear in our list of top 20 machines. In fact, the best variation of the pattern machine is one which predicts taken only if the last two branches were taken (rank 102). In Table VII we compare the performance of this pattern machine with that of the counter and the predictor used in the S-1 machine. The counter is marginally worse than the pattern machine for *spice* and *eqntott*, but is much better than the pattern machine in the other cases. It is also always better than the S-1 predictor. Indeed, the counter does consistently well, as evidenced by the fact that it performs almost as well as the optimal 2-bit machine in all applications.

TABLE V
LISTING OF THE BEST 20 MACHINES OVER ALL APPLICATIONS

	Machine	Average success rate	Best rank	Worst rank	Average rank
1.	ABACBDCD:3	.9058	1	179	50
2.	BCBAADCD:3	.9056	1	210	50
3.	BCBABDCD:3	.9055	2	166	36
4.	BCBDACBA:10	.9054	3	165	35
5.	BACADBDC:12	.9054	4	182	55
6.	BACACDCB:12	.9052	4	164	38
7.	ABACADCD:3	.9051	5	167	39
8.	BCDAACDB:10	.9049	2	292	71
9.	BCBADCAC:3	.9033	8	195	77
10.	BCDCACDB:10	.9032	8	194	78
11.	ABACDCBC:3	.9030	7	193	78
12.	BACADADC:12	.9028	12	200	84
13.	BCBADCBC:3	.9015	6	318	119
14.	BCBDACBC:10	.9014	7	316	119
15.	ABACDCAC:3	.9012	8	319	121
16.	BCBADDAC:3	.9012	19	220	78
17.	BCDCAADB:10	.9011	19	219	80
18.	BCBABDCC:3	.9011	19	77	48
19.	BCBAADCC:3	.9011	17	300	119
20.	BACACDCA:12	.9011	9	327	126

TABLE VII
COMPARISON OF SUCCESS RATES OF VARIOUS COMMON PREDICTORS

Benchmark	Optimal 2-bit	Counter ABACBDCD:3	S-1 predictor BCBADCBC:3	Pattern BCDABCDA:2	1-bit
spice2g6	97.2	97.0	97.0	97.1	96.2
doduc	94.3	94.3	94.3	93.6	90.2
gcc1.35	89.1	89.1	88.8	87.2	86.0
espresso	89.1	89.1	88.7	87.3	87.2
li	87.1	86.8	85.5	85.7	82.5
eqntott	87.9	87.2	86.6	87.5	82.9

IV CONCLUSIONS

In this paper we have demonstrated a new and efficient technique to analyze finite-state machines to determine an optimal one to be used for branch prediction. The technique employs the fact that many of the possible machines are essentially equivalent, and that one simulation can simultaneously analyze all possible output assignments for a set of next-state assignments. We have used this technique to determine optimal 4-state machines for the SPEC benchmark programs running on the IBM RS/6000. We have determined that the 2-bit counter does consistently well across all applications. The pattern machine never outperformed the counter machine significantly and was usually worse. The widely used S-1 predictor [12], [13], 13th in Table V, was outperformed by the counter in all applications. This is contrary to expectation from results in [2], possibly because our study omitted the often pervasive loop counter type branches. These branches do well on most common predictors. It is generally easy to take care of them by alternative hardware techniques as in the RS/6000. This also allows smaller branch history tables to be used for predicting the more difficult branches. While our results are specific to the IBM RS/6000, we feel they would also apply to other machines provided the normally predictable branches, like loop-counter branches and returns from subroutines, are filtered out.

The results of this paper suggest that the counter as a 2-bit branch predictor provides a good base for comparison of more sophisticated branch predictors. Further, in several recently published two-level adaptive prediction schemes [5], [6], [7], [8], the second level pattern history uses two bits to record the history. The results in this paper justify the use of the 2-bit counter for recording this history.

TABLE VI
BEST FIVE MACHINES IGNORING STARTING STATE

Rank	State diagram for machine
1.	
2.	
3.	
4.	
5.	

REFERENCES

[1] J.E. Smith, "A study of branch prediction strategies," *Proc. 8th Annual Int'l Symp. on Computer Architecture*, pp. 135-147, June 1981.

[2] J.K.F. Lee and A.J. Smith, "Branch prediction strategies and branch target buffer design," *IEEE Computer*, vol. 17, no. 1, January 1984, pp. 6-22.

[3] *i860 64-bit microprocessor programmer's reference manual*, Intel Corporation, Santa Clara, 1989.

[4] *Alpha architecture handbook*, Digital Equipment Corporation, Maynard, MA, 1992.

[5] T.-Y. Yeh and Y.N. Patt, "Two-level adaptive training branch prediction," *Proc. 24th ACM/IEEE Int'l Symp. and Workshop on Microarchitecture*, pp. 51-61, Nov. 1991.

[6] T.-Y. Yeh and Y.N. Patt, "Alternative implementations of two-level adaptive branch prediction," *Proc. 19th Annual Int'l Symp. on Computer Architecture*, pp. 124-134, May 1992.

[7] S.-T. Pan, K. So, and J.T. Rahmeh, "Improving the accuracy of dynamic branch prediction using branch correlation," *Proc. 5th Int'l Conference on Architectural Support for Programming Languages and Operating Systems*, pp. 76-84, Oct. 1992.

[8] T.-Y. Yeh and Y.N. Patt, "A comparison of dynamic branch predictors that use two levels of branch history," *Proc. 20th Annual Int'l Symp. on Computer Architecture*, pp. 257-266, May 1993.

[9] R. Nair, "Branch behavior on the IBM RISC System/6000," *Research Report RC 17859*, IBM T.J. Watson Research Center, Yorktown Hts., NY, Apr. 1992.

[10] C.G. Ponder, "Studies in branch prediction," *Report UCRL-ID-106077*, Lawrence Livermore National Laboratory, Sept. 1990.

[11] G.F. Grohoski, "Machine organization of the IBM RISC System/6000 processor," *IBM Journal of Research and Development*, vol. 34, no. 1, pp. 37-58, Jan. 1990.

[12] S. McFarling and J. Hennessy, "Reducing the cost of branches," *Proc. 13th Annual Int'l Symp. on Computer Architecture*, pp. 396-403, June 1986.

[13] D.J. Lilja, "Reducing the branch penalty in pipelined processors," *IEEE Computer*, pp. 47-55, July 1988.

Safety Levels–An Efficient Mechanism for Achieving Reliable Broadcasting in Hypercubes

Jie Wu

Abstract—We consider a distributed broadcasting algorithm for injured hypercubes using incomplete spanning binomial trees. An injured hypercube is a connected hypercube with faulty nodes. The incomplete spanning binomial tree proposed in this paper is a useful structure for implementing broadcasting in injured hypercubes. It is defined as a subtree of a regular spanning binomial tree that connects all the nonfaulty nodes. We show that in an injured n-dimensional hypercube with m faulty nodes, there are at least $2^n - 2^m$ source nodes (called I-nodes), each of which can generate an incomplete spanning binomial tree. A method is proposed to locate a large subset of the I-node set using the concept of safety level. The safety level of each node in an n-dimensional hypercube can be easily calculated through $n - 1$ rounds of information exchange among neighboring nodes. An optimal broadcast initiated from a safe node is proposed. When a nonfaulty source node is unsafe and there are at most $n - 1$ faulty nodes in an injured n-dimensional hypercube, the proposed broadcasting scheme requires at most $n + 1$ steps.

Index Terms—Binomial trees, broadcasting, fault tolerance, hypercubes.

I. INTRODUCTION

Efficient *broadcasting* [3] of data is one of the keys to the performance of a hypercube system. Basically, broadcasting is the process of transmitting data from one node, called the source node, to all the other nodes once and only once. Broadcasting provides basic

functions to implement distributed agreement, clock synchronization, and broadcast-and-aggregate type of algorithms. We define an *injured hypercube* [2] as a connected hypercube with faulty nodes. Broadcasting in an injured hypercube is defined as successful broadcasting of a datum to all the nonfaulty nodes. The concept of *incomplete spanning binomial tree* is introduced to implement the broadcasting process. An incomplete spanning binomial tree in an n-dimensional injured hypercube is a connected subgraph of an n-level spanning binomial tree with the same root node that connects all the nonfaulty nodes in the cube, and its root node is called I-node.

Lee and Hayes [5] proposed the concept of *safe node* which requires a stronger condition than the one that defines I-nodes. (Therefore, the safe node set is a subset of the I-node set.) The safe node set can be decided in $O(n^2)$ rounds of information exchange among neighboring nodes. However, the broadcasting algorithm based on this definition of safe node is applicable to injured hypercubes with no more than $\lceil \frac{n}{2} \rceil$ node failures. That is, there are cases when no safe nodes exist in an injured hypercube with more than $\lceil \frac{n}{2} \rceil$ faulty nodes. Wu and Fernandez [10] gave a refined definition of safe nodes by relaxing certain conditions and hence increasing the size of the safe node set and raising the degree of fault tolerance. The process that identifies the node status needs fewer rounds than the one in [5] in general. However it still requires $O(n^2)$ rounds in the worst case.

In this paper, we propose the concept of *safety level*, which is an enhancement of the safe node concept by further weakening its definition. Each node in an n-dimensional hypercube is assigned an integer within the range of 0 to n. A node with safety level n is still called safe node. The safety level is an approximate measure of the number and distribution of faulty nodes in the neighborhood, rather than just the number of faulty nodes. We provide a process that identifies the node status in $n - 1$ rounds of information exchange among neighboring nodes. Simulation results show that the safe node set is very close to the I-node set when $m < n$. A broadcasting scheme is proposed which uses the safety level of each node. It is shown that broadcast from a safe node is both *time* and *traffic* optimal [4], where time is measured by the number of hops (or steps) required to complete a broadcasting and traffic is a measure of the total number of messages transmitted from one node to another in the broadcasting process. Moreover, it is proved that, for each nonfaulty but unsafe node, there is at least one safe neighbor when $m < n$. The same broadcasting scheme can be used by selecting a safe neighbor as the source node. A total of $n + 1$ steps is required in this case.

The proposed method differs from the existing fault-tolerant broadcasting methods which are based on either local information [6] or global information ([1], [7], [9]). Local-information-based broadcasting algorithms normally require routing history as part of message to be broadcast, and results are not optimal. Global-information-based broadcasting algorithms, although having their merits of simplicity and optimality, require a process that collects global information. The broadcasting based on limited information is a compromise of the above two schemes. In the proposed method limited global information is captured in the safety level associated with each node. Since this type of information is easy to update and maintain and the optimality is still preserved, this method is more attractive than the existing ones.

The safety level is the first practical model that captures fault information in terms of the distribution and the number of faults, rather than just in terms of the number of faults. In a separate paper, we show that it can also be used to achieve optimization in routing and