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 SPEC92 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, the 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 Pat'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, 0 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 $4^8 \times 2^8$ or $2^8$ 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 (O) 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**

<table>
<thead>
<tr>
<th>PS</th>
<th>NS</th>
<th>O</th>
</tr>
</thead>
<tbody>
<tr>
<td>i = 0</td>
<td>A</td>
<td>B</td>
</tr>
<tr>
<td>i = 1</td>
<td>B</td>
<td>C</td>
</tr>
<tr>
<td>PS</td>
<td>NS</td>
<td>O</td>
</tr>
<tr>
<td>i = 0</td>
<td>C</td>
<td>A</td>
</tr>
<tr>
<td>i = 1</td>
<td>D</td>
<td>D</td>
</tr>
</tbody>
</table>

![Diagram of a FSM](image)

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 2⁴ 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 4⁴ 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 (non-equivalent) states.

Consider an FSM with 5 states, and hence 5 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 = i, the i-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_j, for which the first i rows lead to exactly j states:  

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

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 j-th row must have transition 0 going to state j − 1, and transition 1 leading to state j. The second term indicates that row j 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 j² ways to assign the j-th row.

The initial conditions are given by:  

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

The total number of machines with 5 states is obtained as 5 * n_5,1. Table II shows the various values of n_j for a 4-state machine.

**TABLE II**

**CALCULATION OF NUMBER OF DISTINCT 2-BIT PREDICTORS**

<table>
<thead>
<tr>
<th>i</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>0</td>
<td>3</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>2</td>
<td>0</td>
<td>0</td>
<td>5 * 3 + 9 * 1 = 24</td>
<td>1 * 3 + 7 * 1 = 10</td>
</tr>
<tr>
<td>3</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>7 * 24 + 16 * 10 = 328</td>
</tr>
<tr>
<td>4</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>16 * 328 = 5248</td>
</tr>
</tbody>
</table>

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.

```c
#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
int a0[3][5]; //next state for i = 2
int a1[3][5]; //next state for i = 3

/* Copies machine number 'an' from row ai, column aj of Table 2 to machine number 'bn' from row bi, column bj. */
void Copy(int ai, int aj, int ai, 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];
    }
}

/* Initializing machines in (1,2) and (1,3) */
int n[5][5];
for (i = 5; j <= 4; j++) {
    n[0][j] = 0;
}
for (i = 1; i <= 4; i++) {
    for (j = 0; j <= 1; j++) {
        n[i][j] = 0;
    }
}
*/
```

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 Chris 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][i-2]; k++) {
    Copy(i-1,j-2,k,i,j,n[i][j]);
    S0[i][j][n[i][j]][1] = 0;
    S1[i][j][n[i][j]][1] = 1;
    m[i][j]++;
}

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

/* Deriving machines for (4,4) from (3,3) */
for (i = 3; i <= 4; i++) {
    for (k = 0; k < n[i-1][i-1]; k++) {
        for (p = 1; p <= 4; p++) {
            for (q = 1; q <= 4; q++) {
                Copy(i-1,j,k,i,j,n[i][j]);
                S0[i][j][n[i][j]][1] = p;
                S1[i][j][n[i][j]][1] = 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", 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 SPEC99 benchmarks may be found in [9]. For the present study we gathered representative traces from each of the SPEC benchmarks. Four of these benchmarks, macs7, matrix300, fpypq, and toms304 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>
<thead>
<tr>
<th>Benchmark</th>
<th>Trace length (instructions)</th>
<th>&quot;Difficult&quot; conditional branches (%)</th>
<th>Branches taken (&quot;difficult&quot; conditional branches)</th>
</tr>
</thead>
<tbody>
<tr>
<td>spice2.g6</td>
<td>38,941,662</td>
<td>6.41</td>
<td>35.04</td>
</tr>
<tr>
<td>doduc</td>
<td>36,857,194</td>
<td>6.13</td>
<td>31.13</td>
</tr>
<tr>
<td>gcc1.35</td>
<td>19,362,014</td>
<td>14.05</td>
<td>50.37</td>
</tr>
<tr>
<td>espresso</td>
<td>24,062,240</td>
<td>15.20</td>
<td>41.55</td>
</tr>
<tr>
<td>li</td>
<td>3,108,199</td>
<td>13.81</td>
<td>38.28</td>
</tr>
<tr>
<td>eqntott</td>
<td>4,378,067</td>
<td>13.04</td>
<td>20.61</td>
</tr>
</tbody>
</table>

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>
<thead>
<tr>
<th>Benchmark</th>
<th>Best prediction %</th>
<th>Best predictor</th>
</tr>
</thead>
<tbody>
<tr>
<td>spice2.g6</td>
<td>97.2</td>
<td>Best predictor</td>
</tr>
<tr>
<td>doduc</td>
<td>94.3</td>
<td>Best predictor</td>
</tr>
<tr>
<td>gcc1.35</td>
<td>89.1</td>
<td>Best predictor</td>
</tr>
<tr>
<td>espresso</td>
<td>89.1</td>
<td>Best predictor</td>
</tr>
<tr>
<td>li</td>
<td>87.1</td>
<td>Best predictor</td>
</tr>
<tr>
<td>eqntott</td>
<td>87.9</td>
<td>Best predictor</td>
</tr>
</tbody>
</table>

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.

**TABLE V**

<table>
<thead>
<tr>
<th>Machine</th>
<th>Average success rate</th>
<th>Best rank</th>
<th>Worst rank</th>
<th>Average rank</th>
</tr>
</thead>
<tbody>
<tr>
<td>1. ABACBCDCD.3</td>
<td>9058</td>
<td>1</td>
<td>170</td>
<td>50</td>
</tr>
<tr>
<td>2. BCBAAACDCD.3</td>
<td>9056</td>
<td>1</td>
<td>210</td>
<td>50</td>
</tr>
<tr>
<td>3. BCDABCBCD.3</td>
<td>9055</td>
<td>2</td>
<td>166</td>
<td>36</td>
</tr>
<tr>
<td>4. BCBCABCBCACD.4</td>
<td>9054</td>
<td>3</td>
<td>165</td>
<td>35</td>
</tr>
<tr>
<td>5. BACBCADBCD.12</td>
<td>9054</td>
<td>4</td>
<td>182</td>
<td>55</td>
</tr>
<tr>
<td>6. BACBCACD.12</td>
<td>9052</td>
<td>4</td>
<td>164</td>
<td>38</td>
</tr>
<tr>
<td>7. ABACBCACD.3</td>
<td>9051</td>
<td>5</td>
<td>167</td>
<td>39</td>
</tr>
<tr>
<td>8. BCBCACD.12</td>
<td>9049</td>
<td>2</td>
<td>292</td>
<td>71</td>
</tr>
<tr>
<td>9. BCBCADACD.3</td>
<td>9033</td>
<td>8</td>
<td>395</td>
<td>77</td>
</tr>
<tr>
<td>10. BCBCACD.10</td>
<td>9032</td>
<td>8</td>
<td>194</td>
<td>78</td>
</tr>
<tr>
<td>11. ABACBCACD.3</td>
<td>9030</td>
<td>7</td>
<td>193</td>
<td>78</td>
</tr>
<tr>
<td>12. BACBCABCACD.4</td>
<td>9028</td>
<td>12</td>
<td>200</td>
<td>84</td>
</tr>
<tr>
<td>13. BCBCADACD.3</td>
<td>9015</td>
<td>6</td>
<td>318</td>
<td>119</td>
</tr>
<tr>
<td>14. BCBCADBCD.3</td>
<td>9014</td>
<td>7</td>
<td>316</td>
<td>119</td>
</tr>
<tr>
<td>15. ABACBCACD.3</td>
<td>9012</td>
<td>8</td>
<td>319</td>
<td>121</td>
</tr>
<tr>
<td>16. BCBCAABCD.3</td>
<td>9012</td>
<td>19</td>
<td>220</td>
<td>78</td>
</tr>
<tr>
<td>17. BCBCADBCD.3</td>
<td>9011</td>
<td>19</td>
<td>219</td>
<td>92</td>
</tr>
<tr>
<td>18. BCBCAABCD.3</td>
<td>9011</td>
<td>17</td>
<td>300</td>
<td>119</td>
</tr>
<tr>
<td>19. BCBCACD.12</td>
<td>9011</td>
<td>9</td>
<td>327</td>
<td>126</td>
</tr>
</tbody>
</table>

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 spic and egnot, 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.

**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], [13] 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.

**REFERENCES**


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 broad-
casting of a datum to all the nonfaulty nodes. The concept of incom-
plete spanning binomial tree is introduced to implement the broad-
casting 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 l-node.

Lee and Hayes [5] proposed the concept of safe node which requires
a stronger condition than the one that defines l-nodes. (Therefore, the safe node set is a subset of the l-node set.) The safe node
set can be decided in O(n) rounds of information exchange among
neighboring nodes. However, the broadcasting algorithm based
on this definition of safe node is applicable to injured hyper-
cubes with no more than \(\frac{n}{2}\) node failures. That is, there are cases
when no safe nodes exist in an injured hypercube with more than
\(\frac{n}{2}\) 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 l-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 com-
plete 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 faulty 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 broad-
casting algorithms normally require routing history as part of mes-
sage to be broadcast, and results are not optimal. Global-information-
based broadcasting algorithms, although having their merits of simp-
licity and optimality, require a process that collects global informa-
tion. 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 in-
formation 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

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

Jie Wu

Abstract—We consider a distributed broadcasting algorithm for in-
jured 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 sub-
tree 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 \(2^m\) source nodes (called l-nodes), each
of which can generate an incomplete spanning binomial tree. A method
is proposed to locate a large subset of the l-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, hyper-
cubes.

I. INTRODUCTION

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

Manuscript received Mar. 5, 1993; revised May 2, 1994.

The author is with the Dept. of Computer Science and Engineering, Florida
Atlantic University, Boca Raton, FL 33431; e-mail jie@cs.fau.edu.
IEEECS Log Number C95040.

0018-9340/95$04.00 © 1995 IEEE

IEEE TRANSACTIONS ON COMPUTERS, VOL. 44, NO. 5, MAY 1995