Formal Analysis of Branch Prediction Algorithms *

Somesh Jha, Yuan Lu, and Edmund M. Clarke
Department of Computer Science
Carnegie Mellon University
Pittsburgh, PA 15213, U.S.A.
{sjha,yuanlu,emc}@cs.cmu.edu

Abstract

In modern microprocessors deep pipelines are widely used to speed up execution. The importance of a good branch predictor to the effectiveness of a deep pipeline in the presence of conditional branches is well-known. There are several branch prediction schemes described in literature. The common way to evaluate the performance of branch prediction algorithms is simulation. In this paper we present a formal methodology for evaluating branch prediction algorithms. Our goal is to provide a technique for analyzing branch prediction which is based on rigorous mathematical foundations and thus provide a deeper insight into the analysis and design of branch prediction algorithms. Our technique hinges on constructing a Markov model or a probabilistic automaton corresponding to the test suite. We also show the effectiveness of our method by experimental results.

Keywords: Formal methods, branch prediction.

1 Introduction

In modern microprocessors, pipelining is one of the most effective ways to improve performance. Branches affect the performance of pipelined microprocessors by causing pipeline stalls due to unresolved branches. Branch prediction is a way to reduce the execution penalty due to branches by predicting, prefetching and initiating execution of the branch target before the branch is resolved. Due to the usefulness of branch prediction schemes several algorithms have been proposed and implemented in modern microprocessors [MRe95]. There are two

*This research is sponsored by the Semiconductor Research Corporation (SRC) under Contract No. 97-DJ-294, the National Science Foundation (NSF) under Grant No. CCR-9505472, and the Defense Advanced Research Projects Agency (DARPA) under Contract No. DABT63-96-C-0071. Any opinions, findings and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of SRC, NSF, DARPA, or the United States Government.

types of branch prediction schemes: *static* and *dynamic*. In static schemes the strategy does not change during execution. A static branch prediction scheme can be as simple as assuming that all branches are taken or that all branches are not taken. The accuracy of various static schemes are reported in Lee and Smith [LS84]. On the other hand, dynamic schemes take advantage of the knowledge of branches' run-time behavior to make predictions. A dynamic branch prediction algorithm is described in [TP95]. The scheme described in [TP95] keeps a history register (indexed by the address of the branch) and a pattern table. The history register records the history of a particular branch and the pattern table keeps the logic for branch prediction. An overview of this scheme is provided in Section 2 and a detailed account can be found in [TP95].

The traditional way to evaluate branch prediction algorithms is simulation. Effectiveness of branch prediction schemes is measured by *missRate*, i.e., a measure of how many times an algorithm fails to correctly predict an outcome of the branch. Next we describe the concept of *missRate* in the context of one branch B. The extension of the idea to multiple branches is fairly straight forward but tedious. For purposes of branch prediction an execution of the program can be viewed as a string over the alphabet $\{Y, N\}$. For example, $(NNYY)$ describes an execution where branch B is not taken the first two times and taken the next two times. Hence a test-suite T (a collection of programs) is a set of strings over the alphabet $\{Y, N\}$. Consider a test-suite T. Given a branch prediction scheme $\mathcal{A}$, let $miss(\sigma, \mathcal{A})$ denote the number of times the prediction of $\mathcal{A}$ misses when the input to the algorithm is the string/execution σ in the test-suite T , or in other words $miss(\sigma, \mathcal{A})$ is the number of times $\mathcal{A}$ fails to correctly predict the outcome of the branch on the execution σ. *MissRate* of the test suite T is denoted by the following fraction:

$$\frac{\sum_{\sigma \in T} miss(\sigma, \mathcal{A})}{\sum_{\sigma \in T} length(\sigma)}$$

MissRate is simply the ratio of the number of misses and the length of the test-suite. We provide a formal methodology to compute the *missRate* of a branch prediction algorithm $\mathcal{A}$. We provide a rough overview of our method here. Using the test-suite T we build a probabilistic automaton or a Markov model PA_T. PA_T induces a probability on each string over the alphabet $\{Y, N\}$. Intuitively, the probability of a string is its frequency of occurring in the test suite T. Now using the description of the branch-prediction algorithm $\mathcal{A}$ we build a Moore automaton MA. Using the Markov model and the model of the branch-prediction algorithm we compute the *missRate*. We explain the basic idea behind our definition. Let Σ_n be the set of strings of length less than or equal to n over the alphabet $\{Y, N\}$. Each string $\sigma \in \Sigma_n$ has a probability $P(\sigma)$ given by the Markov model. Let $miss(\sigma)$ be the number of times the branch-prediction algorithm misses on the execution corresponding to σ. The miss-rate of all the strings of length n is given by the following quantity:

$$missRate_n = \sum_{\sigma \in \Sigma_n} P(\sigma) \frac{miss(\sigma)}{length(\sigma)}$$

The *missRate* in our technique is given by the limit (as $n \to \infty$) of the quantity given above. Please see Figure 1 for a pictorial description of our method. Besides being formal

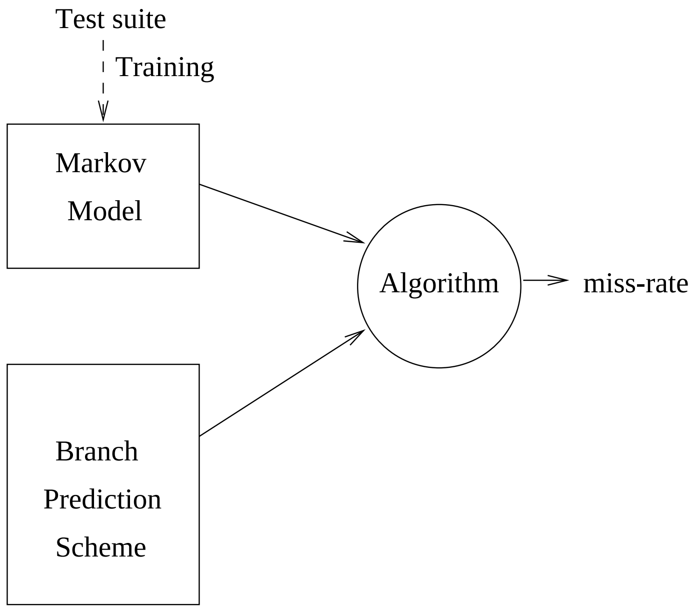

Figure 1: A pictorial description.

and precise, we offer the following advantages over the traditional simulation techniques to evaluate branch-prediction schemes:

- Suppose we are interested in evaluating performance of m branch prediction schemes $A_1, \cdots, A_m$ on a test-suite T. In traditional simulation techniques the test-suite T has to be executed using each of the branch prediction algorithms A_i. In our method once we build the probabilistic automaton PA_T corresponding to the test-suite T, we can reuse PA_T each time we compute the $missRate$ corresponding to the algorithm A_i. We also found that building the Markov model provided important information in itself. We also believe that because our method is more formal it can provide additional insight into branch prediction algorithms. For example, we can compute the lowest miss-rate possible for a specific test-suite given that we are allowed to only keep history of specified length.

- Simulation computes the miss-rate of a branch prediction algorithm on finite executions of a test-suite of programs. We derive a Markov model of the test-suite and then compute the miss-rate of the branch prediction algorithm. The difference from simulation is that we do not put *any bound* on the length of executions. In an abstract sense, we are computing the limit of the miss-rate and hence consider executions of arbitrary length. This can reveal important information about the branch prediction algorithm. For example, if the miss-rate does not converge, it might signify a deeper problem in the algorithm or the nature of the test-suite.

The paper is arranged in the following manner: Section 2 provides a brief overview of two branch prediction algorithms. Section 3 describes a step by step procedure for using our methodology. Mathematical foundations behind our technique are discussed appendix A. Extension of our technique to the case of multiple branches is also discussed in section 3.

We provide experimental results in Section 4. Related work and conclusion are discussed in Section 5 and Section 6 respectively.

2 Overview of Branch Prediction

The purpose of branch prediction algorithms is to predict whether a certain branch will be taken or not. In this section we provide a brief description of two branch prediction algorithms.

First, we briefly describe an algorithm known as *two-level adaptive training scheme*. A more detailed account can be found in [TP95]. Since the *one-level adaptive scheme* is a restriction of the two-level scheme we describe it at the end of the section. The two major data-structures in this scheme are the *branch history register (HR)* and the *branch history pattern table (PT)*. The history register is a shift register that records the most recent history for a particular branch. All the history registers are contained in a *history register table (HRT)*. The history register table is indexed by branch instruction addresses and the pattern table is global, i.e., shared by all history registers. The structure of the *two-level adaptive training scheme* is shown in Figure 2.

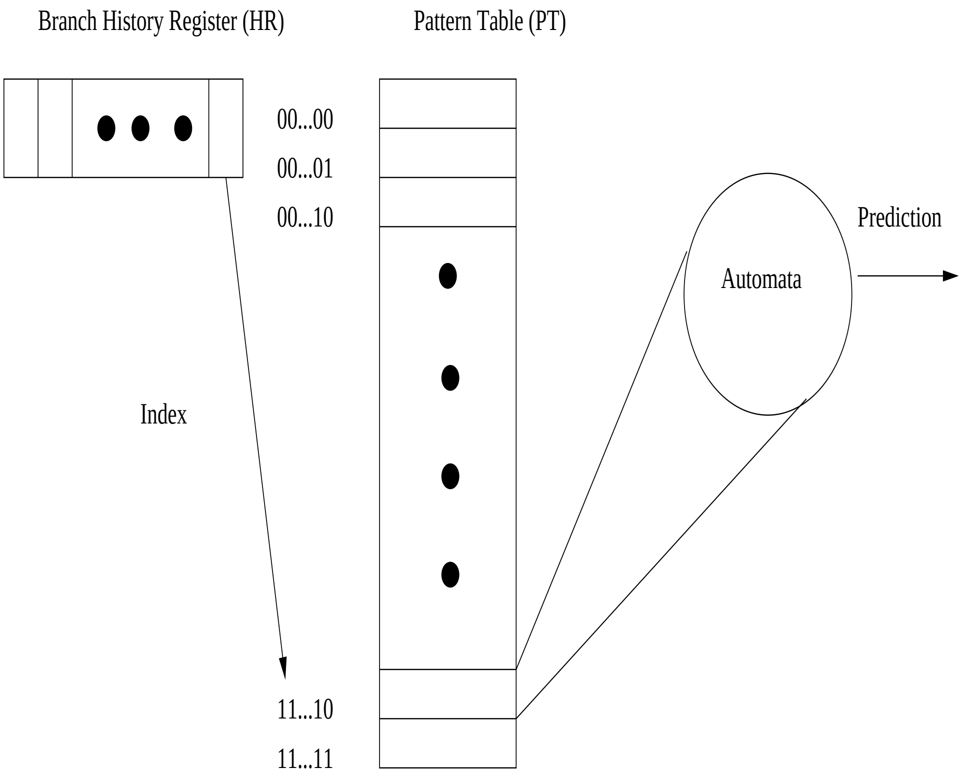

Figure 2: A schematic diagram

Suppose we are interested in predicting the outcome of a conditional branch B_i. Assume that the history register HR_i corresponding to B_i has n bits. Each time if branch B_i is taken a 1 is recorded in HR_i otherwise a 0 is recorded. Assume that we encounter branch

B_i and the contents of the history register HR_i are $(b_1, \cdots, b_n)$. Next, we find the entry with index $(b_1, \cdots, b_n)$ in the pattern table PT. The entry of the pattern table contains a state of a Moore automaton MA. Let us say the entry in the pattern table corresponding to $(b_1, \cdots, b_n)$ is s (state of the Moore automata MA). The output $\lambda(s) \in \{Y, N\}$ of the Moore automaton is the prediction whether the branch is taken or not. Suppose the branch is actually encountered and the real outcome of the conditional branch is x ($x = Y$ if and only if the branch is taken). In this case the entry in the pattern table PT indexed by $(b_1, \cdots, b_n)$ is changed to $\delta(s, x)$, where δ is the transition function of the Moore automaton MA and the outcome of the branch is recorded in the history register. Notice that the algorithm depends on the underlying Moore automaton MA. Several choices for the Moore automaton are investigated in [TP95]. A particular choice of the automata is shown in Figure 3.

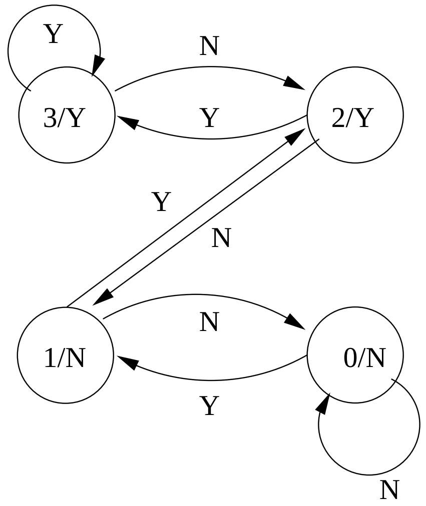

Figure 3: Automaton $A2$

Moore automaton $A2$ shown in Figure 3 is called the *saturated up-down counter*. The counter is incremented when the branch is taken and is decremented when the branch is not taken. The next execution of the branch will be predicted as taken when the counter value is greater than or equal to two; otherwise the branch will be predicted as not taken.

It is not feasible to have enough history registers so that each static branch has its own history register. We describe one approach that allows sharing of history registers between multiple branches. Some other schemes for sharing history registers are described in [TP95]. Recall that the set of history registers is called a history register table. A history register table can be implemented as a hash table, where the address of a conditional branch is used for hashing in the table. So if two branches have the same hash value, they share a history register. This implementation causes interference between branches and hence does not perform as well as the ideal scheme where each static branch has its own history register.

In the *two-level adaptive training scheme* the prediction of a particular branch B uses the following two steps:

- Use the address of B to find its history register HR_B.

- Use the contents of HR_B as index into the pattern table PT. Let us say the entry in the pattern table is s. The prediction for the branch is $\lambda(s)$, where λ is the output function of the underlying Moore machine MA.

- After the actual outcome of the branch is known, the entry the pattern table whose index is given by the contents of the history register HR_B is changed. Contents of the history register HR_B are changed to record the outcome of the branch.

In the *one-level adaptive scheme* described in [Smi81] history registers are not present. The last n bits of the address of the branch are used as an index into the pattern table. The rest of the algorithm works the same as in the *two-level adaptive training scheme*. For example, suppose a branch B with address $(b_1, \cdots, b_n)$ is encountered while executing a program. The address of branch B will be used to index into the pattern table. Suppose state s is stored in the pattern table at address $(b_1, \cdots, b_n)$. If the output of the Moore automaton is Y, branch B is predicted as taken. If the actual outcome of branch is N, the content of the pattern table at address $(b_1, \cdots, b_n)$ is changed to $\delta(s, N)$, where $\delta(s, N)$ denotes the successor of state s with label N in the Moore automaton.

3 Formal Analysis of Branch Prediction

This section explains each step in our methodology for evaluating branch-prediction algorithms. We explain our methodology by considering the case of a single branch. Later in the section we explain how to extend the methodology to multiple branches. Mathematical underpinnings behind our methodology are explained in Appendix A. The exposition in this section is informal.

3.1 Extracting Traces

Outcome of a specific branch in an execution of a program can be considered as string of *taken* or *not-taken*. For example, one might run the program `compress` (to compress files) on a specific file and track the outcome of a specific branch. A *test suite* is simply a collection of program executions, and hence can be modeled as a set of strings. We will view a test suite simply as a subset of strings over the alphabet $\Sigma = \{Y, N\}$, where Y denotes the branch being *taken* and N as the branch *non-taken*. This view simplifies our exposition. In the rest of the section we assume that the set of traces corresponding to the test-suite has been extracted.

3.2 Building the Markov model

Recall that a test-suite T can be regarded as a set of strings from the alphabet Σ (or $T \subseteq \Sigma^\star$), where $\Sigma = \{Y, N\}$. A string $v \in T$ denotes the sequence of outcome of a specific branch B in an execution of a certain program in the test suite. This section describes how to build a Markov model or a probabilistic automaton corresponding to the test-suite T. A precise and mathematical exposition of probabilistic automaton is given in Appendix A.2.

First, we decide on the length of the history (denoted by m) for building the Markov model. For example, if $m = 2$, we will keep the last two outcomes of the branch B. Suppose we have an initial state s_0. For the sake of concreteness, let s_0 correspond to string of all Ns (this amounts to having a history where the branch B is not taken). For each state s in the Markov model, we keep a counter (denoted by $N(s)$) which counts how many times state s was encountered while traversing the strings/executions in the test-suite T. For each transition $s \rightarrow s'$ we also maintain a counter $N(s, s')$ which counts how many times the transition $s \rightarrow s'$ is taken. These counters are initially set to 0. For each string $\sigma = v_1, \cdots, v_k$ in the test-suite we perform the following steps:

1. Let $s_0, s_1, \cdots, s_k$ be the sequence of states corresponding to the string $v_1, \cdots, v_k$. The history changes from s_{i-1} to s_i on the input v_i. For example, suppose we maintain a history of length two. Assume that the initial state is $[NN]$. The sequence corresponding to the string YNY is:
$$[NN] \rightarrow [NY] \rightarrow [YN] \rightarrow [YY]$$

2. For each state s_i $(0 \leq i \leq k)$, we increment the counter $N(s_i)$.

3. For each transition $s_i \rightarrow s_{i+1}$ $(0 \leq i < k)$ we increment the counter $N(s_i, s_{i+1})$.

Assume that we have performed the steps outlined above for all the strings/executions in the test-suite T. The probability of the transition $s \rightarrow s'$ is $\frac{N(s,s')}{N(s)}$, i.e., the ratio of the number of times transition $s \rightarrow s'$ is taken and the number of times state s is visited or encountered. Intuitively, $\frac{N(s,s')}{N(s)}$ is the frequency with which the transition $s \rightarrow s'$ is taken by programs in the test-suite T. We illustrate our method using a small example.

Example 3.1 Let the test-suite T be:

$$\{NN, NNNY, NNNYY, NNNYNY, NNNYYY\}$$

Suppose we use a 2-bit history in the probabilistic automaton. The automaton PA is shown in Figure 4. Suppose the automaton PA is run on all the strings in the test-suite T. State 00 is encountered 18 times. Transition $00 \xrightarrow{Y} 01$ is taken 4 times. Hence the probability of the transition $00 \xrightarrow{Y} 01$ is $\frac{4}{18}$. Probabilities of the rest of the transitions are computed in a similar manner.

3.3 Modeling the branch prediction algorithm

We model the branch prediction algorithm as a *Moore automaton*. A *Moore automaton* has the following components:

- A set of states.
- An initial state in which the Moore automaton starts.
- An output of $\{Y, N\}$ in each state. If output in a state is Y, it corresponds to predicting that the branch will be taken.

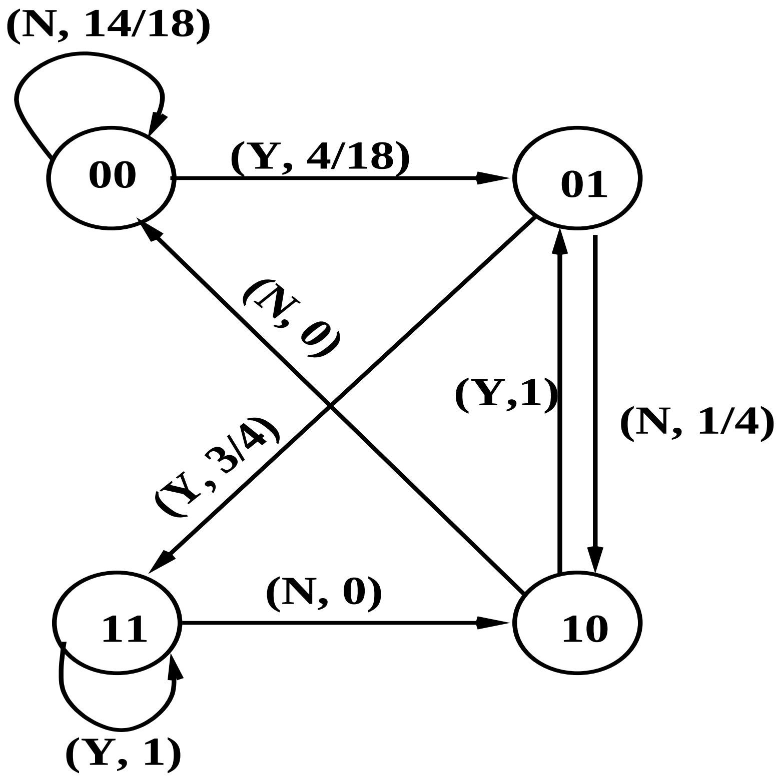

Figure 4: Probabilistic Automaton

- *Transition function*, which describes transitions of a state on inputs Y and N. The successor of state s corresponding to input I (where I is Y or N) will be denoted by $next(s, I)$. Recall that Y corresponds to the branch being taken and N represents the branch being not taken.

Consider a simple branch-prediction scheme which predicts the outcome of a branch to be the same as the last outcome of the branch. For example, if previously the branch was taken, we will predict the branch is taken. The Moore automaton corresponding to this is shown in Figure 5. State `s0` and `s1` correspond to the case when the branch was `not-taken` and `taken` respectively. The outputs or predictions in the state are shown in the figure. We are assuming that the branch-prediction scheme initial starts in the state `s0`.

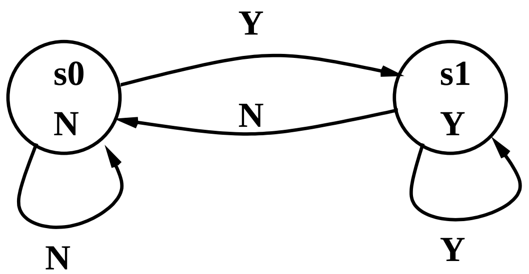

Figure 5: A simple prediction scheme

Next, we describe the structure of the Moore automaton corresponding to the *two-level adaptive training scheme*. For the sake of convenience let us call this automaton M_{2L}. Each state s of the automaton M_{2L} has two components: s_H (the state of the history register) and s_P (state of the pattern table). Output in a state $s = (s_H, s_P)$ can be obtained by looking at the state of the pattern table s_P. The transition function of the Moore automaton M_{2L} should be clear from the description of the algorithm.

3.4 Composing the models

Suppose we have a Markov or a probabilistic automaton model for the test-suite T. For notational convenience we call this model P_T. Let M be the Moore automaton corresponding to the branch-prediction algorithm. Next we describe how to *compose* the two models P_T and M (we denote this composed model as $P_T \| M$). We provide an informal description of the composition in this subsection. For a formal description, please see appendix A.2. A state s in the composed model can be written as (s_1, s_2), where s_1 is a state in P_T and s_2 is a state in M. There is a transition from state (s_1, s_2) to (s'_1, s'_2) if and only if there is a transition from s_1 to s_2 in P_T and s'_1 to s'_2 in M. The probability assigned to the transition $(s_1, s_2) \rightarrow (s'_1, s'_2)$ is the probability of the transition $s_1 \rightarrow s'_1$ in P_T and the output of the state (s_1, s_2) is the output of the state s_2 in M. Loosely speaking, the probabilities are derived from P_T and the outputs or predictions are derived from M. Since the state space of the Moore automaton corresponding to the branch-prediction algorithm can be very large, the naive construction of the combined model can take enormous amount of time. We have devised a heuristic which builds the combined model incrementally starting from the initial state and only considers states reachable from the initial state. Consider the Markov model shown in Figure 4 and the branch-prediction algorithm in Figure 5. Assume that the initial states of the Markov model (Figure 4) and the Moore automaton (Figure 5) are `00` and `s0` respectively. The composite model is shown in Figure 6. Our algorithm works on this composite model and computes the *missRate*. The algorithm is quite technical and the details can be found in appendix A.1.

3.5 Best Predictor

Suppose we are interested in the following question: *what is the best achievable miss-rate by keeping a history of m bits?*. This sub-section addresses this question. The answer is surprising simple and can be computed by using the following steps:

- First, we build a Markov model of the test-suite using m bits of history. This was covered before.

- Next for each state in the Markov model we need to decide what to predict in that state. At a state s let s' be the state such that the probability of the transition from state s to s' is maximal among all the transitions from state s. The *best predictor* at state s predicts the label of the transition from state s to s'. It can be formally proved that the best predictor will always predict the next state which corresponds to the transition with maximal probability.

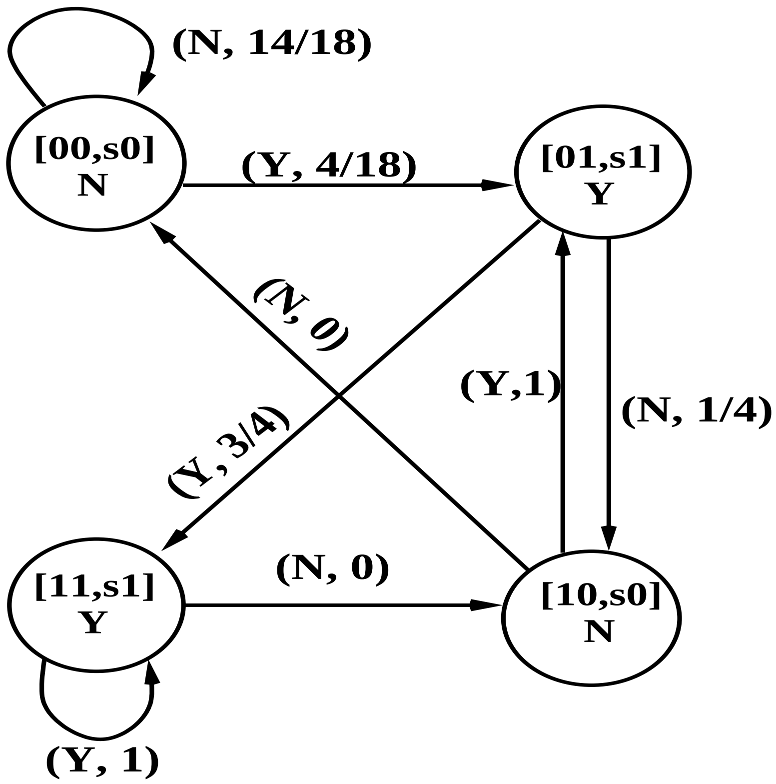

Figure 6: Composite model

- Using the algorithm given in appendix A.1 we find the miss-rate of the best predictor constructed in the previous step.

For example, take the Markov model shown in Figure 4. The best predictor corresponding to the Markov model is shown in the table in Figure 7. We explain the first entry in the

state	prediction
00	N
01	Y
10	Y
11	Y

Figure 7: Best Predictor corresponding to Markov model shown in Figure 4

table. Consider the state `00`. The successor of this state with maximum probability is `00`. Therefore, we predict N in the state `00`. Similar reasoning can be used for the other states. Hence the best 2-bit predictor for the test-suite T considered in subsection 3.2 corresponds to the following simple rule:

> If the outcome for the branch is N last two times, predict N; otherwise predict Y.

3.6 Multiple Branches

Next, we describe how our algorithm can be extended to handle multiple branches. Assume that we are predicting the performance or miss-rate of a branch-prediction algorithm $\mathcal{A}$ on k branches $B_1, \cdots, B_k$. Let us say we are given a test-suite T. Recall that while analyzing one branch we used $\Sigma = \{Y, N\}$ as our alphabet, where Y denoted the fact that the branch was taken and N denoted that the branch was not taken. In the case of k branches $B_1, \cdots, B_k$ the alphabet is $\Sigma = \{Y, N\} \times \{1, 2, \cdots, k\}$, where (Y, i) means that the i-th branch B_i was encountered and taken. In this framework a test-suite T is set of strings from the alphabet Σ or $T \subseteq \Sigma^\star$. For example, $(Y, 1)(N, 3)(Y, 2)$ denotes trace of program where branch B_1 is taken, branch B_3 is not taken, and B_2 is taken. Therefore, probabilistic automaton PA or Markov model representing T is over the alphabet $\{Y, N\} \times \{1, 2, \cdots, k\}$. The Moore automaton MA (representing the branch prediction algorithm) is also over the same alphabet. The definition of miss-rate does not change at all. Hence our framework readily extends to multiple branches.

4 Experimental Results

In this section, we present experimental results for various benchmarks and branch prediction algorithms. All the tables show the *Hit-Ratio* which is one minus the *missRate*. *Hit-Ratio* is shown in percentages. First, we computed the hit-ratio of various algorithms using simulation and then our new method. These methods are not directly comparable because as we pointed out earlier our method conceptually computes the *limit* of the miss-rate, but we wanted to make a comparison between our technique and simulation. Results from simulation are shown in Figure 8. First column shows the name of the benchmark. All the programs are part of the `SPEC95` test-suite. We explain the terminology used in the tables. Consider the first row labeled with `cmpr[1,1]`. This benchmark corresponds to the run of the `compress` program. The first 1 shows that we only considered one branch. The second 1 signifies that we tracked the outcome of a specific branch in the `compress` program. Now we explain columns 2, 3, and 4. These columns correspond to different branch-prediction algorithms. The different branch prediction algorithms are:
`1L2`: one-level branch prediction scheme with history register of length 2.
`2L3`: two-level adapative scheme with history register of length 2.
`2L4`: two-level adapative scheme with history register of length 4.
`2L6`: two-level adapative scheme with history register of length 6.
Description of these algorithms can be found in section 2.

We also computed the hit-ratio for the best-predictor for various history lengths. Table 9 shows the miss-rate of the best-predictor for various history lengths. We consider history lengths of 2,5,6,and 8. Tables 10 and 11 shows the results for benchmarks `cmpr[3,1]` and `cmpr[5,1]`. Recall that these benchmarks correspond to tracking 3 and 5 branches respectively.

Table 12 shows the hit-rate for various algorithms and benchmarks using our new technique. The hit-ratios are slightly different from the one computed using simulation which is not surprising because our method computes the hit-rate in the limit. The column marked

benchmark	1L2	2L3	2L4	2L6
cmpr[1,1]	47.80	71.13	70.45	69.78
cmpr[1,2]	76.87	77.30	77.08	80.15
gcc[1,1]	50.40	57.95	66.34	78.01
go[1,2]	97.67	97.86	97.58	97.26
ksim[1,1]	95.96	95.24	94.97	94.43
ksim[1,2]	99.99	99.99	99.99	99.99
vortx[1,1]	99.35	99.35	99.32	99.36
cmpr[3,1]	87.16	88.83	92.23	93.25
cmpr[5,1]	86.67	82.28	85.28	89.19

Figure 8: Hit-ratio using simulation

# hist	2	5	6	8
cmpr[1,1]	69.95	71.96	73.26	74.25
cmpr[1,2]	78.95	77.24	83.93	87.69
gcc[1,1]	60.66	70.84	78.10	80.97
go[1,2]	98.16	98.26	98.21	98.36
ksim[1,1]	96.09	95.98	95.93	95.82
ksim[1,2]	100.0	100.0	100.0	100.0
vortx[1,1]	99.38	99.38	99.38	99.40

Figure 9: Hit-ratio for best predictor for one branch

# hist	6	9	12	18
cmpr[3,1]	87.59	89.66	92.96	95.25

Figure 10: Hit-ratio for best predictor for three branches

# hist	10	15	20	30
cmpr[5,1]	94.05	95.17	96.62	97.49

Figure 11: Hit-ratio for best predictor for five branches

benchmark	TL	HL	1L2		2L3		2L4		2L6	
			hit	time(s)	hit	time(s)	hit	time(s)	hit	time(s)
cmpr[1,1]	253225	55	48.90	268	69.86	36	69.36	36	70.31	36
cmpr[1,2]	840003	48	73.80	302	75.71	667	78.62	115	80.33	142
gcc[1,1]	48594	35	50.30	141	54.33	163	51.40	*73	48.01	*67
go[1,2]	6528	41	97.94	18	95.32	228	86.82	*205	80.20	*200
ksim[1,1]	1113	20	95.37	15	93.41	17	92.74	18	91.43	20
ksim[1,2]	462226	60	99.23	23	99.05	56	98.89	65	98.56	84
vortx[1,1]	149626	40	99.42	35	99.44	54	99.44	49	99.45	119
cmpr[3,1]	333428	41	87.23	45	84.51	68	86.69	227	80.61	170
cmpr[5,1]	840003	52	84.62	406	79.70	183	82.01	601	80.70	*568

Figure 12: Hit-ratio using new methodology ("*" means that the method does not converge)

`HL` shows the history length to build the Markov model. The length of the trace corresponding to a benchmark is shown in the column marked `TL`. Besides showing the hit-ratio we also show the time used to compute the hit-ratio.

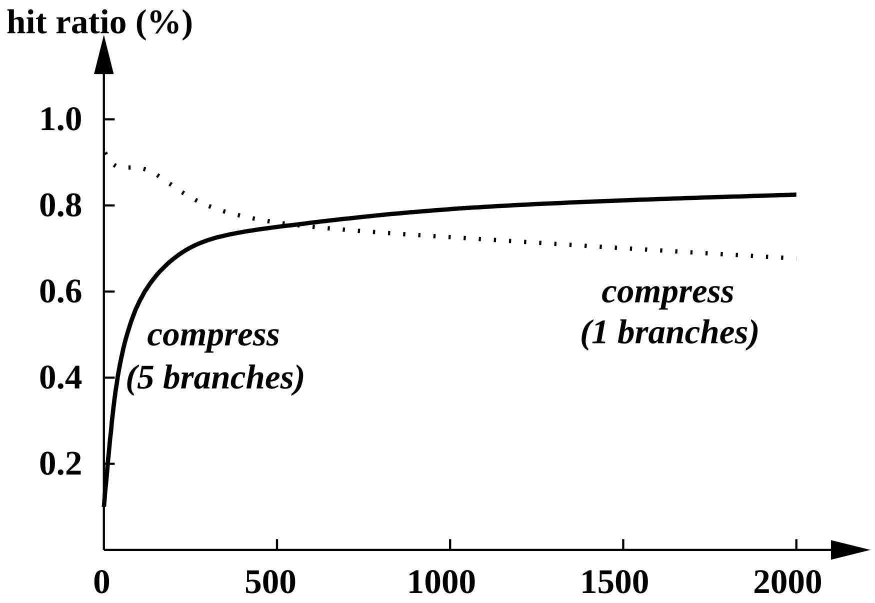

Figure 13: Illustration of convergence

The two curves shown in figure 13 shows the convergence of hit-ratio for benchmarks `compress[1,1]` and `compress[5,1]`. The horizontal axis shows the number of iterations (upto 2,000) used in our calculations and the vertical axis shows the hit-ratio computed using the specified number of iterations. As can be clearly seen from the figure the hit-ratio converges. The hit-ratio depicted in figure 13 corresponds to the `2L4` branch prediction scheme (two-level adaptive scheme with history register of length 4).

5 Related Work

The literature on branch prediction algorithms is vast. We will not try to summarize the literature here. To the best of our knowledge we are the first ones to identify a formal algorithm for computing the miss-rate for branch prediction algorithms. The work that has some similarity with our work is [CCM96]. In [CCM96] authors draw parallel between branch prediction schemes and algorithms for data compression. Based on this analogy they prove optimality results for some specific branch prediction schemes. Due to this similarity we plan to take a closer look at the literature on compression. In an abstract sense there are connections between our method and probabilistic model-checking [ACD91, CY95, PZ93]. In probabilistic model-checking one is interested in proving temporal properties of a labeled Markov chain. On the other hand, we compute a metric (in this case miss-rate) on a Markov chain. Perhaps, some techniques for improving efficiency of probabilistic model-checking could also be used in our algorithm. We will investigate this direction in the future.

6 Conclusion

In this paper we presented a formal methodology for analyzing branch prediction algorithms. Our method was based on extracting a Markov model of the test-suite under question. There are several directions for future work. First, the space requirements for our algorithm in the case of several branches becomes very large. We are currently addressing the issue and have made significant progress in this direction. We will report the results of this new method in a future paper. In some isolated cases, we encounter very slow convergence for the *missRate*. We are currently investigating this problem. We also want to see whether our methodology extends to other methods based on value prediction [LWS96, Lip96]. We believe that this opens a rich new application area for formal methods. Notice that the probabilistic automaton or the model of the test-suite is basically a Markov chain [Dur95, Bil95, Wil92]. We want to investigate whether any techniques from the theory of Markov chains can be used in our algorithm to compute the miss-rate.

Acknowledgements: We thank John P. Shen for his encouragement and feedback. Subash Shankar and Carsten Schuermann provided useful comments on an earlier draft of the paper. Brian Black assisted us with the experimental section.

References

[ACD91] R. Alur, C. Courcoubetis, and D. Dill. Verifying automata specifications of probabilistic real-time systems. In *Real-Time: Theory and Practice*, volume 600 of *Lecture Notes in Computer Science*. Springer-Verlag, Berlin, 1991.

[Bil95] Patrick Billingsley. *Probability and Measure*. John Wiley & Sons, 3rd edition, 1995.

[CCM96] I.K. Chen, J.T. Coffey, and T.N. Mudge. Analysis of branch prediction via data compression. In *Proceedings of the Seventh International Conference on Architec-*

tural Support for Programming Languages and Operating Systems (ASPLOS-VII), October 1996.

[CY95] C. Courcoubetis and M. Yannakakis. The complexity of probabilistic verification. *Journal of ACM*, 42(4):857–907, 1995.

[Dur95] Richard Durrett. *Probability: Theory and Examples.* Duxbury Press, 2nd edition, 1995.

[Lip96] M.H. Lipasti. *Value Locality and Speculative Execution.* PhD thesis, Carnegie Mellon University, 1996.

[LS84] J. Lee and A.J. Smith. Branch prediction strategies and branch target buffer design. *IEEE Computer*, pages 6–22, January 1984.

[LWS96] M.H. Lipasti, C.B. Wilkerson, and J.P. Shen. Value locality and load value prediction. In *Proceedings of the Seventh International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS-VII)*, October 1996.

[MRe95] Microprocessor report, sebastopol, ca. Technical report, MicroDesign Resources, March 1995.

[PZ93] A. Pnueli and L. Zuck. Probabilistic verfication. *Information and Computation*, pages 1–29, 1993.

[Smi81] J.E. Smith. A study of branch prediction strategies. In *Proc. Eight Annual Symposium on Computer Architecture*, pages 135–148, May 1981.

[TP95] T.Yeh and Y. Patt. Two-level adaptive branch prediction. In *Proc 28th Annual International Symposium on Microarchitecture*, pages 313–325, November 1995.

[Wil92] David Williams. *Probability with Martingales.* Cambridge Univ. Press, 1992.

A Mathematical Foundations

Given an alphabet Σ, Σ^i represents all strings of length i over that alphabet. Σ^+ is equal to $\bigcup_{i=1}^{\infty} \Sigma^i$ and $\Sigma^\star = \{\epsilon\} \cup \Sigma^+$, where ϵ is the empty string.

Definition A.1 A *probabilistic Moore automaton* (PMA for short) PM is a 6-tuple $(S, s_0, \Sigma, \delta, \lambda, P)$ where

- S is the set of states.
- $s_0 \in S$ is the initial state.
- Σ is the alphabet.
- $\delta \subseteq S \times \Sigma \times S$ is the transition relation. $s \xrightarrow{x} s'$ is used as a shorthand for $(s, x, s') \in \delta$.

- $\lambda : S \to \Sigma$ is the output function.

- $P : (S \times \Sigma \times S) \to \Re^+$ assigns probabilities to transitions such that for all $s \in S$, $\sum_{s' \in Im(s)} P(s \stackrel{x}{\to} s') = 1$, where $Im(s)$ is the image of the state s under the transition relation δ.

If for all $x \in \Sigma$ and $s \in S$ there exists one state s' such that $s \stackrel{x}{\to} s' \in \delta$, then the probabilistic Moore automaton is called *deterministic*. Throughout the paper we assume that the probabilistic Moore automaton is deterministic.

Given a probabilistic Moore automaton $PM = (S, s_0, \Sigma, \delta, \lambda, P)$ and a string $v \in \Sigma^\star$, let $path(v, PM)$ be the unique path taken by the automaton from the initial state on string v. Given a string v and a PMA PM we define $P(v, PM)$ and $miss(v, PM)$ in the following way: let $s_0 \stackrel{x_0}{\to} s_1 \stackrel{x_1}{\to} \cdots s_{i-1} \stackrel{x_{i-1}}{\to} s_i$ be $path(v, PM)$. Notice that $x_0 x_1 \cdots x_{i-1} = v$. Define the following quantities:

$$
\begin{aligned}
P(v, PM) &= \prod_{k=0}^{i-1} P(s_k \stackrel{x_k}{\to} s_{k+1}) \\
miss(v, PM) &= \sum_{k=0}^{i-1} 1_{(\lambda(s_k) \neq x_k)}
\end{aligned}
$$

where $1_{x \neq y}$ is equal to 1 if $x \neq y$ and 0 otherwise. Intuitively, $P(v, PM)$ is the probability of the unique path traced out by the string v, $miss(v, PM)$ represents the number of times the output of the PMA PM is different from the input or, in other words, the prediction of PM is incorrect. We define $missRate_n(PM)$ as follows:

$$
missRate_n(PM) = \frac{1}{n} \sum_{v \in \bigcup_{j=1}^{n} \Sigma^j} \frac{P(v, PM) \cdot miss(v, PM)}{|v|} \tag{1}
$$

Notice that the quantity given below is the *expected missRate ratio* for a string v.

$$
\frac{P(v, PM) \cdot miss(v, PM)}{|v|}
$$

$missRate_n(PM)$ is sum of *expected missRate ratios* of all strings v with length less than or equal to n divided by n. The division by n is needed to have a normalized number, i.e. number between 0 and 1. Observe that $miss(v, PM) \leq |v|$, so we have the following series of equations:

$$
\begin{aligned}
\sum_{v \in \bigcup_{j=1}^{n} \Sigma^j} \frac{P(v, PM) \cdot miss(v, PM)}{|v|} &\leq \sum_{v \in \bigcup_{j=1}^{n} \Sigma^j} P(v, PM) \\
&= \sum_{j=1}^{n} \sum_{v \in \Sigma^j} P(v, PM) \\
&= n
\end{aligned}
$$

The last step uses the fact that $\sum_{v \in \Sigma^j} P(v, PM) = 1$. This fact will be used repeatedly in this section. The actual *missRate* is given by the following equation:

$$\lim_{n \to \infty} missRate_n(PM) = missRate(PM) \tag{2}$$

The existence of the limit shown above is proved in lemma A.2. The quantity $missRate_n$ is the miss rate of all strings whose length is less than or equal to n. We give another definition of *missRate*. In our experiments, both definitions of miss-rate provide comparable results, but for expository reasons we will continue to use the first definition. Since $miss(v, PM) \leq v$, we have the following series of equations:

$$\begin{aligned}
\sum_{v \in \bigcup_{j=1}^{n} \Sigma^j} P(v, PM) miss(v, PM) &\leq \sum_{v \in \bigcup_{j=1}^{n} \Sigma^j} P(v, PM)|v| \\
&= \sum_{j=1}^{n} \sum_{v \in \Sigma^j} P(v, PM) j \\
&= \sum_{j=1}^{n} j \\
&= \frac{n(n+1)}{2}
\end{aligned}$$

Based on this we could define $missRate_n(PM)$ as:

$$missRate_n(PM) = \frac{2}{n(n+1)} \sum_{v \in \bigcup_{j=1}^{n} \Sigma^j} P(v, PM) miss(v, PM)$$

Next we proceed to derive recursive equation for $missRate_n$. Given a PMA PM and a state $s \in S$, let $\Sigma(s, n)$ be the set of strings of length n such that the last state on $path(v, PM)$ is s. We will define the following quantities:

$$\begin{aligned}
P(s, PM, n) &= \sum_{v \in \Sigma(s,n)} P(v, PM) &(3) \\
miss_1(s, PM, n) &= \sum_{v \in \Sigma(s,n)} miss(v, PM) P(v, PM) &(4)
\end{aligned}$$

Next we provide recursive equations for $P(s, PM, n)$ and $miss(s, PM, n)$. Let $Pre(s)$ be the pre-image of the state s under the transition relation δ. Given $s' \in Pre(s)$ let $s' \xrightarrow{x} s$ be the unique transition taking s' to s. Define $P(s', s) = P(s' \xrightarrow{x} s)$, $input(s', s) = x$, and $miss(s', s)$ equal to 1 if $\lambda(s') \neq x$ and 0 otherwise. Recursive equations for $P(s, PM, n)$ and $miss(s, PM, n)$ are (assume $P(s, PM, 0) = 1$ and $miss(s, PM, 0) = 0$ for all s):

$$\begin{aligned}
P(s, PM, n) &= \sum_{s' \in Pre(s)} P(s', PM, n-1) \cdot P(s', s) &(5) \\
miss_1(s, PM, n) &= \sum_{s' \in Pre(s)} miss_1(s', PM, n-1) P(s', s) \\
&\quad + \sum_{s' \in Pre(s)} miss(s', s) P(s', PM, n-1) P(s', s) &(6)
\end{aligned}$$

Notice that in the equations given above we overload P, but from the context is should be clear what we mean. $P(s, PM, n)$ is the sum of probabilities of all strings that take the initial state to the state s. The miss-rate of all strings in the set $\Sigma(s, n)$ is denoted by $miss_1(s, PM, n)$. The following lemma proves the correctness of the equations given above.

Lemma A.1 Assume that we have correctly computed $P(s, PM, n-1)$ and $miss_1(s, PM, n-1)$ for all $s \in S$. Equations 5 and 6 correctly compute $P(s, PM, n)$ and $miss(s, PM, n)$.

Proof: We will prove equation 6. The proof of equation 5 is similar in nature, but much simpler. The proof for equation 6 follows from series of equations given below:

$$\begin{aligned}
miss_1(s, PM, n) &= \sum_{v \in \Sigma(s,n)} miss(v, PM) P(v, PM) \\
&= \sum_{s' \in Pre(s)} \sum_{u \in \Sigma(s', n-1)} miss(u \cdot input(s', s), PM) \cdot P(u \cdot input(s', s), PM) \\
&= \sum_{s' \in Pre(s)} \sum_{u \in \Sigma(s', n-1)} [miss(u, PM) + miss(s', s)][P(u, PM) \cdot P(s', s)] \\
&= \sum_{s' \in Pre(s)} P(s', s) \sum_{u \in \Sigma(s', n-1)} miss(u, PM) P(u, PM) \\
&\quad + \sum_{s' \in Pre(s)} miss(s', s) P(s', s) \sum_{u \in \Sigma(s', n-1)} P(u, PM) \\
&= \sum_{s' \in Pre(s)} P(s', s) miss_1(s', PM, n-1) + \\
&\quad + \sum_{s' \in Pre(s)} miss(s', s) P(s', s) P(s', PM, n-1)
\end{aligned}$$

The last equation follows from the definition of $miss(s', PM, n-1)$ and $P(s', PM, n-1)$. Also notice that $v \in \Sigma(s, n)$ iff there exists a string $u \in \Sigma(s', n-1)$ such that $s' \in Pre(s)$ and $v = u \cdot input(s', s)$. This is the basis of the second equation $\Box$.

Define

$$\begin{aligned}
q_n(PM) &= \sum_{s \in S} miss(s, PM, n) && (7) \\
&= \sum_{v \in \Sigma^n} miss(v, PM) P(v, PM) && (8)
\end{aligned}$$

Then (see equation 1),

$$missRate_n(PM) = \frac{1}{n} \sum_{i=1}^{n} \frac{q_i}{i} \quad (9)$$

Lemma A.2 $\lim_{n \to \infty} missRate_n$ exists.

Proof: We have the following series of equations:

$$missRate_{n+1} - missRate_n = \frac{q_{n+1}}{(n+1)^2} - \frac{1}{n(n+1)} \sum_{i=1}^{n} \frac{q_i}{i}$$

$$
\begin{aligned}
|missRate_{n+1} - missRate_n| &\leq \frac{q_{n+1}}{(n+1)^2} + \frac{1}{n(n+1)} \sum_{i=1}^{n} \frac{q_i}{i} \\
&\leq \frac{1}{n+1} + \frac{1}{n+1} \\
&= \frac{2}{n+1}
\end{aligned}
$$

The penultimate step follows from the fact that $q_i \leq i$, which is proven in the following series of equations:

$$
\begin{aligned}
q_i &= \sum_{s\in S} miss(s, PM, n) \\
&= \sum_{s\in S} \sum_{v\in\Sigma(s,i)} miss(v, PM)P(v, PM) \\
&\leq i \sum_{s\in S} \sum_{v\in\Sigma(s,i)} P(v, PM) \\
&= i \sum_{v\in\Sigma^i} P(v, PM) \\
&= i
\end{aligned}
$$

The second step follows from the fact that if v is a string of length i, then $miss(v, PM) \leq i$. □

The following series of equations establishes a recursive equation for q_n.

$$
\begin{aligned}
q_n &= \sum_{s\in S} miss(s, PM, n) \\
&= \sum_{s\in S} \sum_{s'\in Pre(s)} miss(s', PM, n-1)P(s', s) + miss(s', s)P(s', PM, n-1)P(s', s) \\
&= \sum_{s'\in S} \sum_{s\in Im(s')} miss(s', PM, n-1)P(s', s) + miss(s', s)P(s', PM, n-1)P(s', s) \\
&= \sum_{s'\in S} \left(miss(s', PM, n-1) \sum_{s\in Im(s')} P(s's) + \sum_{s\in Im(s')} miss(s', s)P(s', PM, n-1)P(s', s) \right) \\
&= \sum_{s'\in S} miss(s', PM, n-1) \;+\; \sum_{s'\in S} \sum_{s\in Im(s')} miss(s', s)P(s', PM, n-1)P(s', s) \\
&= q_{n-1} + \sum_{s'\in S} \sum_{s\in Im(s')} miss(s', s)P(s', PM, n-1)P(s', s)
\end{aligned}
$$

Taking $q_0 = 0$ we have the following recursive equation for q_n $(n \geq 1)$:

$$
q_n = q_{n-1} + \sum_{s'\in S} \sum_{s\in Im(s')} miss(s', s)P(s', PM, n-1)P(s', s) \tag{10}
$$

A.1 Algorithm

This suggests the following algorithm for computing the miss rate. A PMA PM and a precision parameter α are given as an input. We maintain variables `n`, `q`, `missRate`. For each state s in PM we also maintain the variable $P(s)$. Initially, $\mathtt{n} = 0$, $\mathtt{q} = 0$, and $\mathtt{missRate} = 0$, and for all s, $P(s) = 1$. The variables are updated as follows:

1. $\mathtt{n} \leftarrow \mathtt{n} + 1$.
2. Update $\mathtt{q}$ according to the equation 10.
3. For all s update $P(s)$ according to equation 5.
4. $\mathtt{oldmissRate} \leftarrow \mathtt{missRate}$.
5. $\mathtt{missRate} \leftarrow \frac{\mathtt{n}}{\mathtt{n+1}}\mathtt{oldmissRate} + \frac{\mathtt{q}}{\mathtt{n}^2}$.
6. if $|\mathtt{missRate} - \mathtt{oldmissRate}| < \alpha$, stop the algorithm.

Algorithm keeps updating the various quantities according to equations derived earlier until desired accuracy is achieved.

A.2 Computing the miss-rate

A *Moore automaton* MA is a 5-tuple $(S, s_0, \Sigma, \delta, \lambda)$ where these quantities where described in Definition A.1. A *Probabilistic automaton* PA is a 5-tuple $(S, s_0, \Sigma, \delta, P)$ where these quantities carry the same meaning as in Definition A.1. Basically, a probabilistic automaton is a Markov chain where the transitions are labeled by alphabets. Given a string v, define $P(v, PA)$ and $miss(v, MA)$ as before. Notice that $P(v, PA)$ is defined with respect to the probabilistic automaton and $miss(v, MA)$ is defined with respect to the Moore automaton. Given a Moore automaton MA and probabilistic automaton PA, define $missRate(PA, MA)$ by the following equations:

$$missRate'_n(PA, MA) = \frac{1}{n} \sum_{v \in \bigcup_{i=1}^{n} \Sigma^i} \frac{P(v, PA) \cdot miss(v, MA)}{|v|} \tag{11}$$

$$missRate'(PA, MA) = \lim_{n \to \infty} missRate'_n \tag{12}$$

The definition of deterministic Moore and probabilistic automaton is the same as deterministic PMA, i.e., for all s and $x \in \Sigma$ there exists exactly one s' such that $s \xrightarrow{x} s'$. Throughout the paper we assume that all structures are deterministic.

Definition A.2 Given a Moore automaton $MA = (S, s_0, \Sigma, \delta, \lambda)$ and a probabilistic automaton, $PA = (S', s'_0, \Sigma, \delta', P)$, their product is a probabilistic Moore automaton $PA \times MA = (S \times S', (s_0, s'_0), \Sigma, \delta_1, \lambda_1, P_1)$, where various quantities are defined as follows:

- $((s, s'), x, (s_1, s'_1))$ if and only if $(s, x, s_1) \in \delta$ and $(s', x, s'_1) \in \delta'$.
- $\lambda((s, s')) = \lambda(s)$.
- $P_1((s, s'), x, (s_1, s'_1)) = P((s', x, s'_1))$.

Notice that if MA and PA are deterministic, then $PA \times MA$ is a deterministic PMA as well. The following lemma relates the miss rate of a Moore automaton and probabilistic automaton with the miss rate of the product PMA. Proof of the lemma follows straight from the definitions.

Lemma A.3 Let MA be a deterministic Moore automaton and PA be a deterministic probabilistic automaton. Let $PA \times MA$ be their product. We have the following equation:

$$missRate'(PA, MA) \quad = \quad missRate(PA \times MA)$$

The lemma given above suggests the following algorithm:

- Input a Moore automaton MA and probabilistic automaton PA and a precision parameter α.
- Construct the PMA $PA \times MA$ and then run the algorithm given in Subsection A.1.