修士論文

組み合わせ回路における
信頼性の効率的計算手法

氏名：津島 雅俊
学籍番号：6611130040-3
指導教員：山下 茂 教授

提出日：2015 年 2 月 6 日

立命館大学大学院　情報理工学研究科
博士課程前期課程　情報理工学専攻

内容梗概

近い将来，トランジスタによって構成される組み合わせ回路では，動作中に発生する可能性のあるソフトエラーが深刻な問題となることが予想されている．したがって，論理回路の信頼性を効率的に評価する手法の開発は，回路の設計において非常に重要となってきている．

論理回路の信頼性を評価するための既存の手法としては，Probabilistic Transfer Matrix (PTM) を用いた手法が存在する．しかし，この手法は，最悪の場合，非常に大きなメモリが必要となる．そこで，本研究では，既存手法とは異なるアプローチを用いてメモリの使用量を削減する．このアルゴリズムは，各入力に対して出力が正しく得られる確率を用いて，ゲート数に比例した空間計算量で計算する．計算途中に発生する同じ計算を減らすことによって，計算時間を減らす工夫も行われている．

本手法は，最新の PTM を用いた手法の結果と比較実験を行った．その結果，最も効果的なケースでは，メモリの使用量を 2 万分の 1 程度に削減することに成功した．

目 次

図 目 次

表 目 次

第1章 はじめに

集積回路の縮小や動作電圧の低下に伴い，放射線がゲートに与える影響は，無視できないものとなっている．このような影響によって一時的に発生するエラーは，ソフトエラーと呼ばれる [1]．近年，ソフトエラーの発生が顕著になってきたことで，このエラーの発生を考慮した設計が求められるようになった．そのため，回路の正確な信頼性を，できるだけ効率的に評価する手法が必要となってきている．

ソフトエラーの発生は，必ずしも回路の外部出力に影響を与えるとは限らない．例えば，2 入力の AND ゲートに本来 $(00)_2$ の入力を与えるべき状況において，二つのうち一つの入力が反転したとしても最終的な結果が変化することはない．このように，ゲートの論理によってソフトエラーの伝搬を防ぎ止める効果を論理マスクと呼ぶ．一方で，入力が $(11)_2$ の際には，どちらの入力も反転することは許されない．この例のように，論理マスクが発生するかどうかは，ゲートの論理とその入力によって決定される．したがって，回路の出力が最終的に誤りとなる確率は，ゲートの組み合わせとその回路に与えられる入力に依存する．すなわち，回路の入力数 n に対して 2^n パターンの場合の回路の故障確率を考える必要がある．そこで，n 入力の回路を評価する際には，2^n パターンについての信頼性を考慮しなくてはならない．このような計算は，n が大きくなると困難であるため，通常はモンテカルロ法を用いたフォールト挿入テストが用いられる．あるいは解析的な手法として，Probabilistic Transfer Matrix (PTM) [2] を用いた手法が存在する．一方，これらとは異なる考え方として，論理マスクについて見積る手法も提案されている [3].

PTM を用いた手法では，ゲートや部分回路の入出力の対応を行列で表し，その行列同士の積やクロネッカー積を用いて目的とする回路に対応した行列を得る．この時，行列のサイズは，クロネッカー積によって指数的に増加する．そこで，既存手法では Algebraic Decision Diagram (ADD) [4] を用いた実装を行い，空間計算量の問題に対処している．ADD は，論理関数を表現する Binary Decision Diagram [5] から派生したデータ構造の一つであり，行列やベクトルをコンパクトに管理しながら，演算を行うことができる．しかしながら， ADD の終端ノードは，行列中の保持しなくてはならない値の種類数だけ存在する．したがって，最悪の場合は指数的な量のメモリが必要となり，大規模な回路に対する PTM を生成するためには十分ではない．

そこで本論文では，空間計算量を削減した評価手法を提案する．提案手法では，

既存手法の最終的な評価式を変形し，各入力に対する出力が正しく得られる確率を，無閉路有向グラフとして計算する．この手法によって，ゲートの数に比例した量のメモリで計算することができる．特に大幅にメモリの使用量を削減したケースとして，PTM を使った手法と比べて約 2 万分の 1 となったケースも存在する．この手法は，再収斂が存在する回路では正しい結果が得られないが，その誤差は，疑似乱数を用いたシミュレーション結果と比較して 10% 程度に抑えられている．さらに，時間計算量と正確性の向上に向け，本論文では数式処理によって計算する手法も提案する．この手法は，ファンアウト・ポイントで分割し，一つの部分回路を関数とみなして計算を行う．このとき，同一の部分回路を表す関数同士の計算には，特殊な演算規則を適用するようにすることで，再収斂に対する問題を回避する．

以下では，まず，第 2 章で既存手法として PTM について解説する．次に，第 3 章では空間計算量を削減した評価手法とその改良について提案する．その後，第 4 章で実験と考察を述べ，第 5 章で本研究のまとめと今後の課題について述べる．

第2章 Probabilistic Transfer Matrix (PTM)

PTM は，ゲートや回路の入力に対する出力の確率を表す行列である．この章では，回路の信頼性を計算する既存手法として，PTM を用いた手法について説明する．回路全体に対応する PTM は，基本的なゲートや結線の分岐に対応する PTM からいくつかの演算を用いて構成される．この計算は Algebraic Decision Diagram [4] を用いることで，効率的に行うことができる．

2.1 ゲートや回路と対応した PTM の構築

m 入力， n 出力の部分回路を考える．この回路に，$\boldsymbol{i}$ を 2 進数で表現した長さ m のビット列 $bin(\boldsymbol{i}, m)$ を入力し，出力が長さ n のビット列 $bin(\boldsymbol{o}, n)$ となる確率を p とする．この部分回路に対応する PTM は 2^m 行 2^n 列で，M_p の $\boldsymbol{i}+1$ 行 $\boldsymbol{o}+1$ 列目の値は p とする．

$$M_p(\boldsymbol{i}+1, \boldsymbol{o}+1) = p$$

PTM の 1 行は，ある入力パターンに対して，各出力パターンが発生する確率を表現し，その総和は 1 となる．したがって，行列 M の行と列は，それぞれ入力と出力のパターン数だけ存在する．

例えば，出力が $p = 0.05$ で反転するような AND ゲートは，以下の様な行列で表現できる．

$$AND_p = \begin{bmatrix} 1-p & p \\ 1-p & p \\ 1-p & p \\ p & 1-p \end{bmatrix} = \begin{bmatrix} 0.95 & 0.05 \\ 0.95 & 0.05 \\ 0.95 & 0.05 \\ 0.05 & 0.95 \end{bmatrix} \tag{2.1}$$

この例の 1 行目は，入力が $bin(\boldsymbol{i}) = (00)_2$ となる時，出力は 0.95 の確率で $bin(\boldsymbol{o}) = (0)_2$ となるが，0.05 の確率で反転して $bin(\boldsymbol{o}) = (1)_2$ となることを表している．一方 4 行目は，入力が $bin(\boldsymbol{i}) = (11)_2$ となる時，出力は 0.95 の確率で $bin(\boldsymbol{o}) = (1)_2$ となるが，0.05 の確率で反転して $bin(\boldsymbol{o}) = (0)_2$ となることを表している．また，図 2.1 の回路に存在するような 2 個に分岐する部分は以下のように表す．

$$F_2 = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

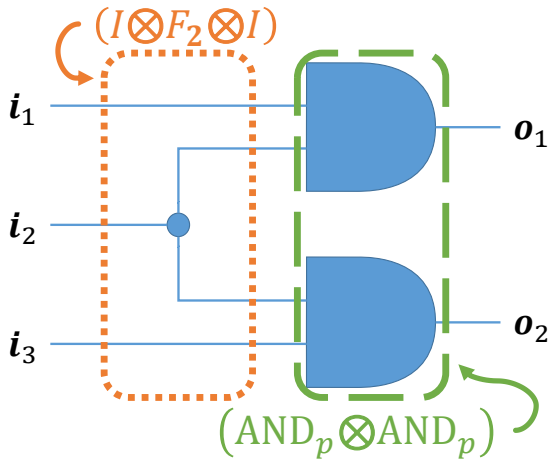

図 2.1：組み合わせ回路の例

ゲートの直列接続に対応する PTM は，対応する PTM 同士の積に対応する．一方，並行したゲートに対応する PTM は，対応する PTM 同士のテンソル積（クロネッカー積，$\otimes$）に対応する．これらの計算を組み合わせることで，目的とする回路の PTM を得ることができる．

例えば，確率 0.05 で反転する AND ゲートを用いて図 2.1 の回路を構成した時，この回路に対応する PTM は式 (2.2) のように求められる．

$$
\begin{aligned}
M_p &= (I \otimes F_2 \otimes I)(AND_p \otimes AND_p) \qquad (2.2)\\
&= \left(\begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix} \otimes \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \otimes \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix} \right) \left(\begin{bmatrix} 0.95 & 0.05 \\ 0.95 & 0.05 \\ 0.95 & 0.05 \\ 0.05 & 0.95 \end{bmatrix} \otimes \begin{bmatrix} 0.95 & 0.05 \\ 0.95 & 0.05 \\ 0.95 & 0.05 \\ 0.05 & 0.95 \end{bmatrix} \right) \\
&= \begin{bmatrix}
0.9025 & 0.0475 & 0.0475 & 0.0025 \\
0.9025 & 0.0475 & 0.0475 & 0.0025 \\
0.9025 & 0.0475 & 0.0475 & 0.0025 \\
0.0475 & 0.9025 & 0.0025 & 0.0475 \\
0.9025 & 0.0475 & 0.0475 & 0.0025 \\
0.9025 & 0.0475 & 0.0475 & 0.0025 \\
0.0475 & 0.0025 & 0.9025 & 0.0475 \\
0.0025 & 0.0475 & 0.0475 & 0.9025
\end{bmatrix}
\end{aligned}
$$

最終的に得られた 8 行 4 列の行列から，各入力に対する出力の発生確率がわかる．例えば，$M_p(4,2) = 0.9025$ からは，$\boldsymbol{i} = 4 - 1 = (011)_2$: $i_1 = 0, i_2 = 1, i_3 = 1$ を入力した時，0.9025 の確率で $\boldsymbol{o} = 2 - 1 = (01)_2$: $o_1 = 0, o_2 = 1$ が出力されることがわかる．

回路に故障が発生しない理想的な状況に対して PTM を計算すると，確率 p は 0 もしくは 1 となる．このような PTM を，特に Ideal Transfer Matrix (ITM) と

呼ぶ．図 2.1 の回路に対応する ITM は，式 (2.3) のようになる．

$$
\begin{aligned}
M &= (I \otimes F \otimes I)(AND \otimes AND) \qquad (2.3)\\
&= \left(\begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix} \otimes \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \otimes \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix}\right) \left(\begin{bmatrix} 1 & 0 \\ 1 & 0 \\ 1 & 0 \\ 0 & 1 \end{bmatrix} \otimes \begin{bmatrix} 1 & 0 \\ 1 & 0 \\ 1 & 0 \\ 0 & 1 \end{bmatrix}\right)\\
&= \begin{bmatrix} 1 & 0 & 0 & 0 \\ 1 & 0 & 0 & 0 \\ 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 1 & 0 & 0 & 0 \\ 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}
\end{aligned}
$$

回路の最終的な信頼性は，式 (2.4) のように，正しい出力が得られる確率を，入力パターンが与えられる確率によって重み付き平均をとることによって評価する．入力パターンが与えられる確率はベクトル $\boldsymbol{v}$ で表現し，$\boldsymbol{i}+1$ 番目の要素は，$\boldsymbol{i}$ を 2 進数としたときの入力パターンが与えられる確率を表している．

$$
\mathit{fidelity}(\boldsymbol{v}, M, M_p) = |\boldsymbol{v}(M_p. * M)|_1 \qquad (2.4)
$$

ここで，‘.∗’ は要素ごとの乗算を表し，$|\boldsymbol{v}|_1$ は，ベクトル $\boldsymbol{v}$ の l_1-ノルム を表す．

図 2.1 の回路に，全ての入力パターンが等確率で与えられる場合，$\boldsymbol{v}$ は，各要素は 0.125 で要素数 8 のベクトルとなる．ここまでの例で計算した M_p, M を用いて式 (2.4) を計算すると，0.9025 という値が得られる．

$$
M_p. * M = \begin{bmatrix} 0.9025 & 0 & 0 & 0 \\ 0.9025 & 0 & 0 & 0 \\ 0.9025 & 0 & 0 & 0 \\ 0 & 0.9025 & 0 & 0 \\ 0.9025 & 0 & 0 & 0 \\ 0.9025 & 0 & 0 & 0 \\ 0 & 0 & 0.9025 & 0 \\ 0 & 0 & 0 & 0.9025 \end{bmatrix}
$$

$$
\begin{aligned}
\mathit{fidelity}(\boldsymbol{v}, M, M_p) &= \left|\begin{bmatrix} 0.125 & \ldots & 0.125 \end{bmatrix} (M_p. * M)\right|_1\\
&= \left|\begin{bmatrix} 0.5640625 & 0.1128125 & 0.1128125 & 0.1128125 \end{bmatrix}\right|_1\\
&= 0.9025
\end{aligned}
$$

この結果は，ソフトエラーによって 0.05 の確率で反転する AND ゲートを用いて図 2.1 の回路を構成した時，全ての入力パターンが等確率で与えられることを仮定すると， 0.9025 の確率で正常に動作することを意味している．

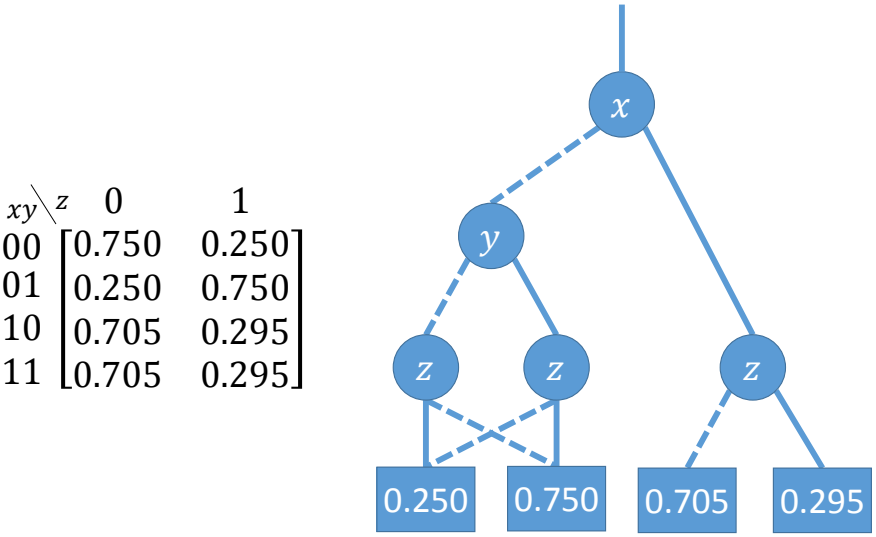

図 2.2：ADD によって行列を表現する例

2.2 Algebraic Decision Diagram (ADD) を用いた計算

2.1 節の手法において，計算の途中で発生する PTM のサイズは，指数的に大きくなる可能性がある．与えられた 2 個の行列 M_1, M_2 のサイズが $k \times l$ と $m \times n$ のとき，$M_1 \otimes M_2$ のサイズは $km \times ln$ となる．そのため，回路の中で並行した部分の入力と出力の数を n', m' とすると，$2^{n'} \times 2^{m'}$ のサイズの PTM が生成される．この課題に対して， Algebraic Decision Diagram (ADD) [4] を用いた手法 [2] が提案された．ADD は Binary Decision Diagram (BDD) [5] から派生したデータ構造で，行列内の部分的に共通した情報を省略して保持しながら計算することができる．

ADD は，無閉路有向グラフ $(\Phi \cup V \cup T, E)$ として $\{0,1\}^n \mapsto S$ の関数の集合を表す．ここで，S は定数の集合である．Φ は入次数 0，出次数 1 の開始節点であり，その ADD で表現したい関数と 1 対 1 で対応する．中間節点 V は，関数が依存する変数を表しているため，出次数は 2 となる．この 2 個の辺は，1-枝と 0-枝と呼ばれる．関数をの結果を得る際には，節点の変数に 0 を割り当てた場合は 0-枝，1 を割り当てた場合は 1-枝をたどり，終端節点 T を探索する．終端節点は，変数割り当ての結果として S に含まれる定数でラベル付けされている．BDD では論理関数を再帰的に表現し，終端節点は定数関数を表す 1 か 0 となる．一方，ADD では，終端節点の扱い方を変更し，0, 1 に限らない任意の終端節点を持つことができる．

行列を ADD で表現した際のグラフ構造の例を図 2.2 に示す．1-枝と 0-枝は，それぞれ実線の矢印と破線の矢印で表されている．各節点で場合分けを行う決定変数が 1 の場合は 1-枝，0 の場合は 0-枝に対応し，根節点から終端節点への 1 本のパスは， $\{0,1\}^n$ と定数の対応を表している．すなわち，各非終端節点は決定変数

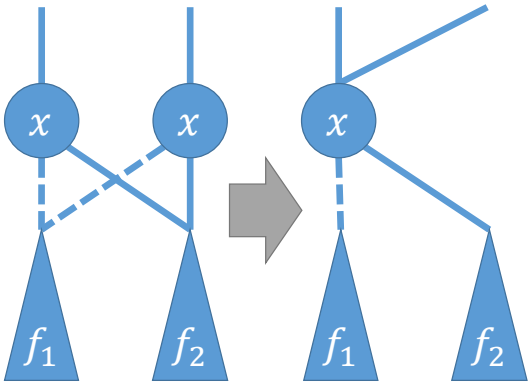

図 2.3：等価な節点の共有

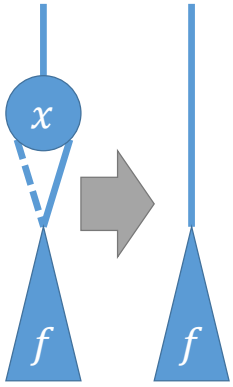

図 2.4：冗長な節点の削除

でラベル付けされているため，決定変数の割り当て通りに枝をたどることで，対応する定数を得ることができる．

ADD は変数の順序を固定し，等価な節点の共有と冗長な節点を削除することによって，コンパクトかつ一意な表現を得ることができる．2 つの節点 v_i, v_j が以下を満すとき，この 2 つの節点は等価であり，図 2.3 のように共有することができる．

- v_i と v_j にラベル付けされた決定変数が同じ
- v_i と v_j の 0-枝が同じ節点を指す
- v_i と v_j の 1-枝が同じ節点を指す

v_i, v_j が終端節点の場合には，ラベル付けされた定数のみを比較して等価性を判断する．また，ある節点の 0-枝と 1-枝が同じ節点を指すとき，この節点は冗長なので，図 2.4 のように削除することができる．図中の f_1, f_2, f は ADD の部分グラフである．

また，2 個の ADD を入力し，それらの二項演算の結果を直接生成するアルゴリズムが考案されている．これらのアルゴリズムは，一意性についてのキャッシュテーブルと演算についてのキャッシュテーブルを利用することで効率的に計算することができる．ADD を用いて PTM を計算する手法では，必要な演算を追加で定義し，同様の効率性を持ったまま計算することができる．

第3章 信頼性の効率的計算手法

ADD を用いると，等価な節点の共有や冗長な節点の削除によって，行列内の部分的に共通した情報を省略して表現できる．しかし，行列はテンソル積によって急激に増加するため，ADD による圧縮では不十分であると考えられる．ADD の終端節点は，行列に含まれる値の種類数だけ存在するため，行列のサイズが増加すれば，終端節点の数も増加しやすくなるからである．終端節点の数が増加すると，それらを区別するための節点も増加し，ADD のサイズは行列のサイズと相関する可能性がある．

また，各ゲートに与えられる p の値は，全て同じ値とは限らない．既存手法では，全てのゲートで $p = 0.05$ 用いて実験を行ったため，同じ部分行列が発生しやすく，ADD に適した状況となっている．逆に，ゲートの出力が反転する確率として，いくつかの異なる値を設定した場合は，節点の共有が発生しづらく，ADD では対処できない状況となることが予想される．大規模で多様な回路を評価するためには，より効率的な計算方法が必要と考えられる．

3.1 評価式の変形による効率化手法

既存手法では，行列 M_p, M のサイズが問題となる．そこで，提案手法ではこの行列を用いずに， $M_p . * M$ と等価な値を得る別のアプローチを検討する．

$M_p . * M$ は，正しい出力以外を 0 にする計算とみなすことができる．$M_p . * M$ から，行ごとに 0 以外の要素を抜き出したベクトル $\boldsymbol{s}$ を用いると，式 (2.4) の値は，内積を用いて以下のように求めることができる．

$$\mathit{fidelity}(\boldsymbol{v}, M, M_p) = \boldsymbol{v} \cdot \boldsymbol{s} \tag{3.1}$$

$M_p . * M$ が正しい出力を抜き出す処理と考えると， $\boldsymbol{s}$ の $\boldsymbol{i}+1$ 番目の要素は，2 進数として表したビット列 $\boldsymbol{i}$ を入力したときに正しい出力が得られる確率である．2.1 節の例と同じ状況（図 2.1 の回路を，0.05 の確率で反転するゲートを用いて構成し，入力パターンが等確率で与えられる場合）ならば， $\boldsymbol{s}$ は各要素が 0.9025 で要素数 8 のベクトルとなり，行列を計算することなく， $\boldsymbol{v} \cdot \boldsymbol{s} = 0.9025$ が得られる．

3.1.1 正しい出力が得られる確率の計算

$\boldsymbol{s}$ の各要素は，組み合わせ回路 C を無閉路有効グラフ $G_C = (PI \cup PO \cup V_C, E_C)$ として表し，再帰的に計算することで求められる．PI, PO は，それぞれ外部入力と外部出力を表す頂点集合であり，V_C は，各ゲートの頂点集合である．ゲート $g \in V_C$ が確率 p で反転するとき，$err(g) = p$ とする．ある頂点 g' の出力が，頂点 g に入力されている時，2 個の頂点は入力される側から出力する側への有向辺 $(g, g') \in E_C$ で結ばれる．また，頂点 g のファンインを $FI(g)$，ファンアウトを $FO(g)$ で表す．g の入次数は $|FI(g)|$，出次数は $|FO(g)|$ となる．$g \in PI$ の場合には $|FI(g)| = 0$ であり，$g \in PO$ の場合には $|FO(g)| = 0$ である．

全ての外部出力が正しく出力されれば，正しい出力が得られている．回路に $\boldsymbol{i}$ を入力した時のゲート g が誤りとなる確率 $fail(\boldsymbol{i}, g)$ とすると，$\boldsymbol{s}$ の各要素は以下のようになる．

$$\boldsymbol{s}(\boldsymbol{i}+1) = \prod_{g \in PO} (1 - fail(\boldsymbol{i}, g)) \tag{3.2}$$

$\boldsymbol{i}$ を入力した時にゲート g が誤りとなる確率 $fail(\boldsymbol{i}, g)$ は，ゲートのファンインで考えられる全てのエラーパターンに分けて考える．エラーのパターンはビット列 $\boldsymbol{e}$ を用いて表す．例えば，$|FI(g)| = 3$ のゲートに $(011)_2$ を入力し，$\boldsymbol{e} = (010)_2$ のエラーが発生した時，ビット同士の XOR によって $(010)_2$ が入力されると考える．全てのエラーパターン集合は，$\{0,1\}^{|FI(g)|}$ で得ることができる．すなわち，頂点 g でエラーパターン $\boldsymbol{e}$ が発生し，同時に反転の状態を出力が誤りとなるような確率を求める．

$$fail(\boldsymbol{i}, g) = \sum_{\boldsymbol{e} \in \{0,1\}^{|FI(g)|}} O(\boldsymbol{i}, g, \boldsymbol{e}) \times toFail(\boldsymbol{i}, g, \boldsymbol{e}) \tag{3.3}$$

ただし，$g \in PI$ の時，反転が発生することはないので，

$$fail(\boldsymbol{i}, g) = 0 \tag{3.4}$$

とする．

エラーパターン $\boldsymbol{e}$ の生起確率は，前段の頂点 g' から計算される．

$$O(\boldsymbol{i}, g, \boldsymbol{e}) = \prod_{\substack{g' \in FI(g), \\ e_{g'} \in \boldsymbol{e}}} \begin{cases} fail(\boldsymbol{i}, g') & : e_{g'} = 1 \\ 1 - fail(\boldsymbol{i}, g') & : e_{g'} = 0 \end{cases} \tag{3.5}$$

$\boldsymbol{e}$ の i 番目の要素が 1 の時，g の i 番目の出力が誤って入力される確率が与えられる．すなわち，式 (3.5) は，全てのファンインが同時に $\boldsymbol{e}$ の表す状態になる確率を求めている．

$toFail(\boldsymbol{i}, g, \boldsymbol{e})$ は，ゲート g が最終的に誤った出力となるように反転する確率である．回路に $\boldsymbol{i}$ を入力した時のゲート g の正しい出力を $out(\boldsymbol{i}, g)$ とし，$\boldsymbol{i}$ を入力してエラーパターン $\boldsymbol{e}$ が発生した時のゲート g の出力を $act(\boldsymbol{i}, g, \boldsymbol{e})$ とする．これらの結果から，このゲートが誤った出力となるためには，反転した状態になるべきかどうかを判断し，式 (3.6) のように計算する．

$$toFail(\boldsymbol{i}, g, \boldsymbol{e}) = \begin{cases} err(g) & : out(\boldsymbol{i}, g) = act(\boldsymbol{i}, g, \boldsymbol{e}) \\ 1 - err(g) & : out(\boldsymbol{i}, g) \neq act(\boldsymbol{i}, g, \boldsymbol{e}) \end{cases} \tag{3.6}$$

入力の誤りに関わらず，論理マスクの影響によって正しい出力が得られる場合，ゲート g は出力が誤りになるように反転しなくてはならない．一方，入力の誤りが出力に影響を与え，最終的に誤った出力となる場合には，ゲート g はその誤った出力を次のゲートへ入力する必要がある．例えば，2 入力の AND ゲート g が存在する回路の外部入力に $bin(\boldsymbol{i})$ を与えた結果，回路内に存在する AND ゲート g に $(00)_2$ が入力されたとする．このとき，正しいゲートの出力は $out(\boldsymbol{i}, g) = 0$ となる．ゲート g の出力が本来とは異なるようになるためには，$bin(\boldsymbol{e}) = (11)_2$ の場合に反転せず，それ以外では反転しなくてはならない．

式 (3.3) の $O(\boldsymbol{i}, g, \boldsymbol{e}) \times toFail(\boldsymbol{i}, g, \boldsymbol{e})$ は，エラーパターン $\boldsymbol{e}$ が発生し，かつ，そのエラーパターンの時に最終的なゲート g の出力が正しい値から反転する確率を求めている．全てのエラーパターンについて場合分けを行い，それぞれの状態についての確率の和を求めることで，そのゲートの出力が誤りとなる確率を得る．

実際の計算では，各入力を終端ノードに設定してから Algorithm 1 のように再帰的に計算を行う．9-16 行目が式 3.5 の計算に相当する．*toFail* の計算に用いる *out* の値は，入力の設定時に事前に計算することができる．また，*act* の値は， *out* と $\boldsymbol{e}$ を用いて計算できる．この計算は，全ての出力線から開始しなくてはならない．

関数 *fail* の結果は，入力パターンとノードの組に対して一意に定まるので，一度求めた結果は再利用することができる．具体的には，Algorithm 1 に加えて， g に対する値を計算していないか調べ，存在すればその値を返すような処理行なえばよい．また，入力パターンを変更するごとに計算結果を消去している．したがって，1 個の入力パターンに対しては，式 (3.2) のように $|PO|$ 回だけ再帰を開始しなくてはならないが，パターンを設定してから 2 個目以降の外部出力に対する結果は，過去に計算した値を利用できる場合があるため比較的少ない時間で求められると期待できる．

Algorithm 1 の 8-18 行目のループは，一度の呼び出しで $2^{|FI(g)|} \times |FI(g)|$ 回実行される可能性がある．そこで，4 入力以上を持つゲート ($|FI(g)| \geq 4$) は，2 入力のゲートに分割して計算しなくてはならない．このとき，元のゲートの出力が反転する確率と，分割後の最終段のゲートの出力が反転する確率は等しく，それ以外は反転を発生しないゲートとする．このように回路を変形させることで，$2^{|FI(g)|} \times |FI(g)|$ 回のループを，2^3 回のループを含んだ $|FI(g)| - 1$ 回の関数呼び出しに置き換えることができる．

Algorithm 1 fail: 与えられたノードの出力が反転する確率を求める

Require: g: 出力のエラー率を求めるノード

1: **if** $g \in PI$ **then**
2: **return** 0
3: **end if**
4: **if** g はすでに計算済み **then**
5: **return** $fail\_cache[g]$
6: **end if**
7: $res = 0$
8: **for all** $\boldsymbol{e} \in \{0,1\}^{|FI(g)|}$ **do**
9: $O \leftarrow 1$
10: **for all** $g' \in FI(g), e_{g'} \in \boldsymbol{e}$ **do**
11: **if** $e_{g'} = 1$ **then**
12: $O \leftarrow O \times fail(g')$
13: **else**
14: $O \leftarrow O \times (1 - fail(g'))$
15: **end if**
16: **end for**
17: $res \leftarrow res + O \times toFail(g, \boldsymbol{e})$
18: **end for**
19: $fail\_cache[g] \leftarrow res$
20: **return** res

3.1.2 異なる入力間での計算結果の再利用

関数 *fail* は，外部入力側のノードからトポロジカル順に計算することもできる．このとき，最終的に出力が誤る確率は入力に依存するため，変化した入力に依存したノードのみを更新すれば十分である．図 3.1 は入力が変化した際に，再計算を行うノードを示している．この例では，入力が変化した $\boldsymbol{i}_1, \boldsymbol{i}_2$ から開始して，$a, \boldsymbol{o}_1, \boldsymbol{o}_2$ の順に更新する．$\boldsymbol{i}_3, \boldsymbol{i}_4$ の入力は変化しないため，これらのみに依存するノード b は更新する必要がない．このようにすると，変化した入力に依存するノードのみを再計算できる．

計算順序の変更に加えて，入力パターンの割り当てにグレイコードを用いることで，さらに計算時間を短縮できる．グレイコードは，1 ビットのみを変更させながら，全ての入力パターンを列挙できるため，再計算が開始されるノードを減らすことができるからである．入力パターンを単純な 2 進数を用いて 0 から順番に割り当てると，全てのビットが変化してしまうケースが存在する．例えば，$(011)_2 \rightarrow (100)_2$ のような変化は，3 箇所のが変化し，3 個のノードから開始しなくてはならない．それに対し，グレイコードを用いると，2 つ目以降の入力パターンについては，常

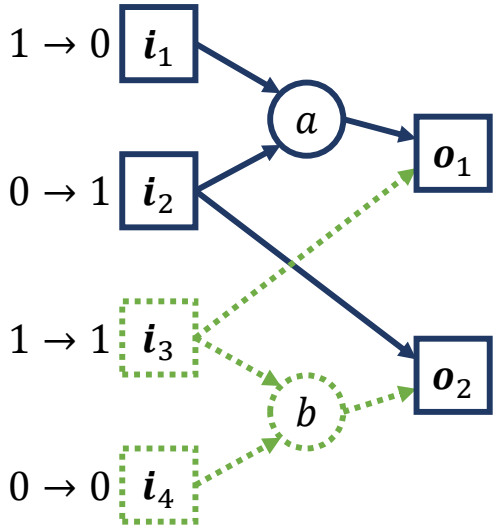

図 3.1：4 入力 2 出力の回路ににおける入力パターンが変更した際の再計算

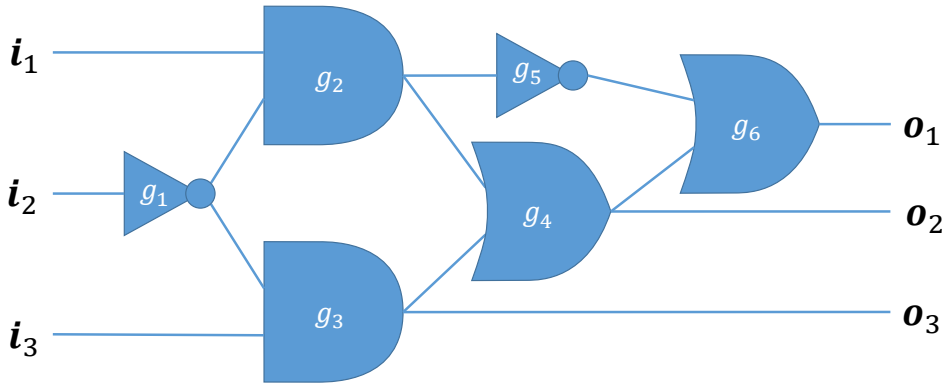

図 3.2：2 箇所の再収斂によって正確な結果が得られない例

に 1 個のノードから再計算をするだけで十分となる．

3.1.3 正しい確率が得られないケース

ここまで提案してきた手法では，回路に再収斂を含むケースに対して正確な結果を得ることができない．回路に再収斂が含まれている場合，計算結果には，同一のゲートに対して異なるソフトエラー発生の状態を仮定した計算が含まれてしまうからである．例えば，図 3.2 のようなケースにおいて，g_1 から分岐した出力は g_4 へ 2 通りの経路から入力される．この時，ゲート g_4 の結果は，ゲート g_1 でエラーが起きた場合と起きない場合の矛盾した仮定を用いているため，ここまで述べてきた手法では，誤った値を計算していることになる．また，ゲート g_3 の結果も両方の仮定を用いている．したがって，g_4 での計算は，g_2 の計算で g_1 にエ

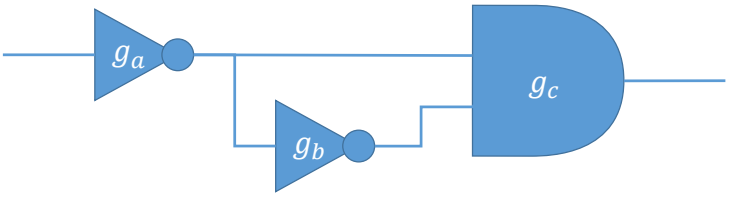

図 3.3：再収斂が 1 箇所存在するケース

ラーが発生しないと仮定したにもかかわらず，g_4 の計算で g_1 にエラーが発生すると仮定して計算した場合を含んでしまう．ゲート g_2 の分岐でも，同様に矛盾した仮定を用いて計算してしまう．

3.2 数式処理による計算手法

関数 $fail$ の計算において誤った結果となってしまうのは，同一のゲートについて，実際には発生しない状態を仮定をしてしまうからである．そこで，再収斂の元となるファンアウト・ポイントに対し変数を定義することで，この状態の計算を修正することを考える．

3.2.1 補正ルールの検討

再収斂が問題となるケースとして，図 3.3 のような回路を考える．ここで，ゲート g_a の出力が異なる状態を仮定してしまうのが問題である．そこで，ゲート g_a の出力が最終的に 1 となる確率を $P(g_a)$ とおく．他のゲートについて出力が 1 となる確率を求めると，以下のようになる．

$$\begin{aligned} P(g_b) &= P(g_a)err(g_b) + (1 - P(g_a))(1 - err(g_b)) \\ P(g_c) &= P(g_a \cap g_b)(1 - err(g_c)) + (1 - P(g_a \cap g_b))err(g_c) \end{aligned}$$

g_b は g_a に依存しているため，ゲート g_a, g_b の出力が同時に 1 になる確率は，$P(g_a)P(g_b)$ では正しく計算出来ない．

$$P(g_a)P(g_b) = P(g_a)^2 err(g_b) + P(g_a)(1 - P(g_a))(1 - err(g_b))$$

ゲート g_a, g_b の出力が同時に 1 になる正しい確率は，

$$P(g_a \cap g_b) = P(g_a)err(g_b)$$

である．そこで，3 種類の状況について，以下のように補正する．

補正ルール 1 $P(g_i)^2 \rightarrow P(g_i)$

補正ルール 2 $(1-P(g_i))^2 \to (1-P(g_i))$

補正ルール 3 $P(g_i)(1-P(g_i)) \to 0$

補正ルール 2 は，出力が 0 となる場合について **補正ルール 1** と同様に補正することを示している．また，同一のゲートであるため，反転が発生する状態と発生しない状態は同時に起こり得ない．そこで **補正ルール 3** のように 0 に置き換える．

ここで，**補正ルール 2**，**補正ルール 3** の左辺を展開し，**補正ルール 1** を適用した場合，

補正ルール 2 $(1-P(g_i))^2 = 1-2P(g_i)+P(g_i)^2 \to (1-P(g_i))$

補正ルール 3 $P(g_i)(1-P(g_i)) = P(g_i)-P(g_i)^2 \to 0$

となるため，**補正ルール 1** のみを適用するだけで十分であるということがわかる．

3.2.2 補正ルールを考慮した数式処理システム

補正ルール 1 を適用するためには，計算途中の情報を数式として保持する方法が考えられる．再収斂されるノードでは，$P(g_i)$ を区別できないため，単に数値として確率を表現している場合には，補正ルールを適用することができないからである．そこで，補正ルールを考慮した数式処理システムを構築する．

$P(g_i)$ を記号として保持し，乗算の演算を行う際に補正ルールを適用させながら数式を処理する．この時，$P(g_i)^2 \to P(g_i)$ のルールから，2 以上の整数を t として $P(g_i)^t \to P(g_i)$ が考えられる．したがって，現れる数式の各変数に関して 2 次以上の項を考える必要が無く，係数と変数の集合だけを管理すればよい．

のちに再収斂が発生するファンアウト・ポイントの集合を R で表す．すなわち，$g \in R$ ならば，$P(g)$ は数式中で変数として表現しなくてはならない．2 以上の指数を考えない場合，係数と変数の組み合わせの組で項を表し，その集合によって以下のように数式を表現することができる．

$$F_e = \{(coeff, vars) | vars \in \{0,1\}^{|R|}, coeff = F(vars)\} \tag{3.7}$$

例えば，$R = \{a, b\}$ において，$F(a,b) = 3ab + b$ を表す場合を考えると，

$$F(0,0) = F(1,0) = 0$$

$$F(0,1) = 1, F(1,1) = 3$$

$$F_e = \{(0,(0,0)),(1,(0,1)),(0,(1,0)),(3,(1,1))\}$$

となる．提案手法に必要な演算は，加減算と乗算である．上記のように数式を表現した場合，これらの演算は集合の操作によって実現することができる．

Algorithm 2 operator $+$: 数式同士の加算

Require: F_e, G_e: 加算を行う多項式を表す集合

1: $result \leftarrow \{$ ハッシュテーブル $\}\varnothing$
2: **for all** $(coeff, vars) \in F_e \cup G_e$ **do**
3: $\quad result[vars] \leftarrow (result[vars]\ or\ 0) + coeff$
4: **end for**
5: **return** *result*

Algorithm 3 operator $\times$: 数式同士の **補正ルール 1** を適用した乗算

Require: F_e, G_e: 乗算を行う多項式を表す集合

1: $result \leftarrow \{$ ハッシュテーブル $\}\varnothing$
2: **for all** $(t_F, t_G) \in F_e \times G_e$ **do**
3: $\quad vars \leftarrow (t_F.vars \mid t_G.vars)$
4: $\quad coeff \leftarrow (t_F.coeff \mid t_G.coeff)$
5: $\quad result[vars] \leftarrow (result[vars]\ or\ 0) + coeff$
6: **end for**
7: **return** *result*

加算は，項の変数の組み合わせごとに分類し，係数の合計を求めれば良い．加算のアルゴリズムを Algorithm 2 に示す．このアルゴリズムによって，*vars* と *coeff* の対応が得られるので，そこから $(coeff, vars)$ の集合を生成することができる．同様のアルゴリズムを用いて，減算も実現することができる．

乗算は， 2 個の数式に含まれる項について乗算を行い，加算と同様に分類する．乗算のアルゴリズムを Algorithm 3 に示す．ここで， '|' は要素同士の OR を表す．項同士の乗算は，係数部分と変数部分に分けて行われ，変数部分については，要素同士の OR で計算できる．

Algorithm 2, 3 で示された通り，数式の演算にはハッシュテーブルが必要となる．したがって，*vars* についてはハッシュ値を計算しなくてはならない．*vars* がビット列で表されている場合には，ビット数に比例する時間でハッシュ値を計算できる．あるいは，変数の組み合わせを，単一の組み合わせのみを含む特殊な Zero-suppressed BDD (ZDD) [6] として表すこともできる．同一の組み合わせかどうかは ZDD の節点番号を用いて判別でき，ハッシュ値もこの節点番号を用いれば良い．要素同士の OR は， ZDD の Join 演算 [7] を用いて実現できる．

3.2.3 数式の生成

数式は，後に再収斂するファンアウト・ポイントで分割し，各部分回路ごとに生成する．各ゲートの出力が 1 になる確率を計算する数式を生成するアルゴリズムは Algorithm 4 のようになる．*fail* の計算と同様，入力パターンの生起確率と，

Algorithm 4 $P(g)$: 与えられたノードの出力が 1 になる確率についての数式を求める

Require: g: 出力のエラー率を求めるノード

if $g \in PI$ もしくは，g はファンアウト・ポイント **then**
  return { 記号としての } $P(g)$
end if
if g はすでに計算済み **then**
  return $one\_ex\_cache[g]$
end if
$res = 0$
for all $\boldsymbol{b} \in \{0,1\}^{|FI(g)|}$ **do**
  $O \leftarrow 1$
  for all $g' \in FI(g), b_{g'} \in \boldsymbol{b}$ **do**
    if $b_{g'} = 1$ **then**
      $O \leftarrow O \times P(g')$ { ルールを適用した数式としての計算 }
    else
      $O \leftarrow O \times (1 - P(g'))$ { ルールを適用した数式としての計算 }
    end if
  end for
  $res \leftarrow res + O \times toBeOne(g, \boldsymbol{b})$ { ルールを適用した数式としての計算 }
end for
$one\_ex\_cache[g] \leftarrow res$
return res

その入力の際に 1 になる確率によって計算できる．$toBeOne$ はゲートにビット列 $bin(\boldsymbol{b})$ を入力した際に，ゲートの出力が 1 になるための確率を返す．この関数は式 (3.8) の様に計算できる．

$$toBeOne(g, \boldsymbol{b}) = \begin{cases} err(g) & : logic(g, \boldsymbol{b}) = 0 \\ 1 - err(g) & : logic(g, \boldsymbol{b}) = 1 \end{cases} \tag{3.8}$$

このとき，$logic(g, \boldsymbol{b})$ は，ゲート g にビット列 $\boldsymbol{b}$ を入力した際の出力を返す．

ある入力について構成された数式は，他の入力においても再利用できる．これは，入力を 0 から 1 に変えることと，外部入力が反転する確率が 0 から 1 になることが等価であるからである．そこで，のちに再収斂が発生するファンアウト・ポイントの集合 R に加え，外部入力 PI に含まれるノードについても変数として扱う．したがって，$P(g), g \in \{R \cup PI\}$ を変数とした数式が生成され，各外部入力が 1 になる確率を与えることで，外部入力の信頼性を計算できる．

$\boldsymbol{s}$ の導出は，外部入力からトポロジカル順に，$P(g)$ へ値を割り当てることで求めることができる．$P(g)$ は，1 となる確率であるから，$g_i \in PI$ の場合は，1 を入力

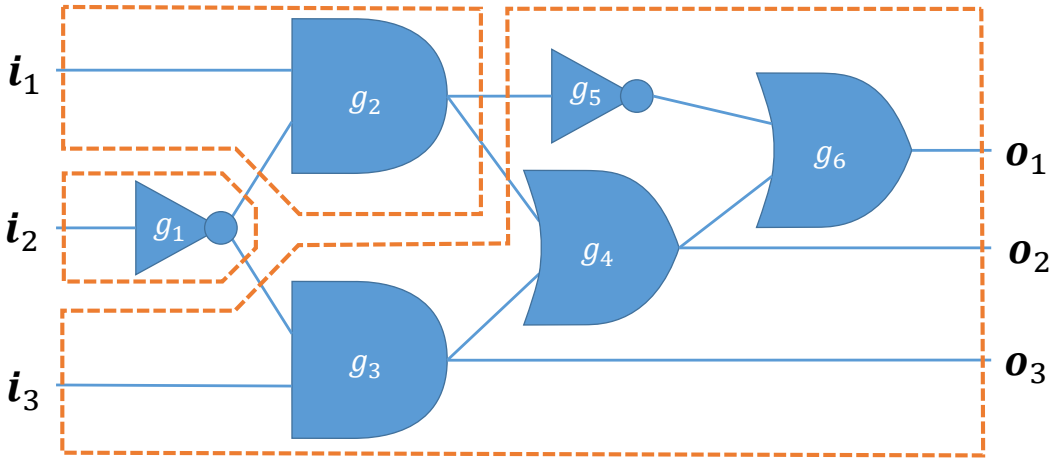

図 3.4：図 3.2 の回路を分割する例

する場合は $P(g_i) = 1$，0 を入力する場合は $P(g_i) = 0$ とすればよい．Algorithm 4 によって，部分回路の出力が 1 となる確率を得ることができるので，その結果を，対応するファンアウト・ポイントの変数へ代入することで，さらに別な部分回路の出力について計算できる．全ての部分回路について計算すると，外部出力が 1 となる確率を得ることができ，別に計算した $out(\boldsymbol{i}, v)$ の結果と合わせて，正しい出力が得られる確率 $\boldsymbol{s}(\boldsymbol{i}+1)$ を得る．

例えば，図 3.2 の回路は，図 3.4 のように分割する．$P(\boldsymbol{i}_2)$ を用いて $P(g_1)$ を表し，$P(\boldsymbol{i}_1), P(g_1)$ を用いて $P(g_2)$ を表し，$P(g_1), P(g_2), P(\boldsymbol{i}_3)$ を用いて $P(\boldsymbol{o}_1), P(\boldsymbol{o}_2), P(\boldsymbol{o}_3)$ を表す．この時，数式は **補正ルール 1** に従って生成される．$\boldsymbol{s}(0)$ の場合は，$P(\boldsymbol{i}_1) = P(\boldsymbol{i}_2) = P(\boldsymbol{i}_3) = 0$ として計算し，それ以外の要素についても同様に計算することで，正確な結果を求めることができる．

第4章 実験結果と考察

3.1 節で提案した手法についてプログラムを実装し，実験を行った．表 4.1 に PTM を用いた既存手法 [2] との比較結果を示す．表の各列は，以下のように定義される．

回路 ベンチマーク回路のファイル名
入力 外部入力の数 $|PI|$
時間 計算時間（秒）
メモリ メモリの使用量 (MB)
PTM 計算順序を改善した PTM の結果
提案 グレイコードを用いて改善した提案手法の結果
比率 (PTM の結果)/(提案手法の結果)

なお， 0.0005 に満たない値は，0.000 として表記している．また，比率の計算において結果が 0.000 の場合は， 0.001 として計算している．

本実験のベンチマーク回路には Berkeley Logic Interchange Format (BLIF) フォーマットで記述された LGSynth'93 [8] に含まれる組み合わせ回路を用いてお

表 4.1：PTM を用いた既存手法との計算時間とメモリ使用量の比較

		時間 (秒)			メモリ (MB)		
回路	入力	PTM	提案	比率	PTM	提案	比率
C17	5	0.212	0.000	212.0	0.000	0.003	0.3
decod	5	5.132	0.000	5132.0	1.02	0.007	145.7
xor5	5	3.589	0.000	3589.0	0.227	0.008	28.4
z4ml	7	3.849	0.006	641.5	1.004	0.015	66.9
9symml	9	89.145	0.035	2547.0	6.668	0.025	266.7
x2	10	11.015	0.006	1835.8	2.344	0.008	293.0
cu	14	23.7	0.195	121.5	2.215	0.013	170.4
parity	16	1.06	0.309	3.4	0.113	0.015	7.5
pm1	16	72.384	0.168	430.9	3.734	0.016	233.4
pcle	19	28.81	4.411	6.5	3.309	0.014	236.4
cc	21	57.4	18.326	3.1	4.839	0.019	254.7
mux	21	18.052	22.765	0.8	2.051	0.017	120.6
c8	28	35559.5	5538.1	6.4	930.023	0.041	22683.5

表 4.2：実験環境

CPU	Intel® Core™ i5-3570 CPU (3.40GHz)
メモリ	8 GB
OS	Linux (Fedora 20)
言語	C++
コンパイラ	g++ 4.8.3

表 4.3：計算時間の改善

回路	改善なし	Dist	Dist-Gray
xor5	0.000	0.001	0.000
9symml	0.065	0.035	0.035
x2	0.024	0.018	0.006
cu	0.529	0.315	0.195
parity	1.45	0.456	0.309
pm1	2.034	0.339	0.168
pcle	14.055	5.862	4.411
cc	72.297	32.845	18.326
mux	103.562	31.852	22.765
c8	30990.6	8417.5	5538.1

り，各ゲートが反転する確率は，$err(g) = 0.05$ を設定した．提案手法の実験環境は，表 4.2 の通りである．PTM を用いた既存手法の実験結果は，Pentium 4 Xeon (2GHz) を用いている．したがって，CPU の FLOPS [9] の差を考慮すると，計算時間が 100 倍以上改善されたケースについては，提案手法の方が十分に優位であると考えられる．

提案手法のメモリ使用量はゲート数に比例するため，既存手法に比べて少なくなっている．一方，計算時間が大きく改善されないのは，既存手法が ADD によって同一の部分回路の再計算を回避しているのに対し，提案手法は，外部入力に与えられるの入力のパターン数がボトルネックになっているからであると考えられる．

表 4.3 は，3.1 節の手法について，計算時間の改善を行った結果を示している．単位は全て秒で表されている．“改善なし” は，Algorithm 1 を用いて計算した結果である．“Dist” は，異なる入力間での計算結果の再利用を行った結果で，“Dist-Gray” は，さらにグレイコードを用いて再計算を開始するノードを減らした結果である．“Dist-Gray” では，実験を行ったほぼ全てのケースで計算時間が改善されている．xor5 の “Dist” が改善しない場合より遅いのは，トポロジカル順序の計算がオーバーヘッドになっているからであると考えられる．

表 4.4 は，3.1 節の手法について，正確性の検証を行った結果である．疑似乱数を用いて，1000 回のシミュレーションを行った結果と，提案手法によって計算した確率を比較している．“絶対差の平均” の列は各パターンごとの確率 $\boldsymbol{s}$ の絶対差の平均値である．また，“絶対差の最大” は，$\boldsymbol{s}$ の絶対差の最大値を示している．

表 4.4：正確性の検証

回路	絶対差の平均	絶対差の最大
C17	0.034	0.091
decod	0.011	0.028
xor5	0.011	0.030
z4ml	0.006	0.021
9symml	0.010	0.035
x2	0.033	0.130
cu	0.036	0.090
parity	0.043	0.111
pm1	0.011	0.060
pcle	0.021	0.081

この結果から，今回実験を行ったほとんどのケースで，10% 以内の誤差の範囲で計算できていることがわかる．

数式を用いた手法は，現在までの実装では，全てのゲートに記号を割り当て，既存の数式処理システムである GiNaC [10] を用いている．この実装の場合は，非常に小さなケースでしか計算することはできなかった．既存の数式処理システムを用いる場合，**補正ルール 3** より，回路中の全ての g について $P(g)(1-P(g_i))$ で割った剰余を求めればよい．しかし，計算中に補正ルールを適用しないため，矛盾した 2 個の状態を保持しながら計算を行うことになってしまう．後に再収斂するゲート g が $P(g)$ の確率で 1 を出力する場合，補正ルールを適用しない計算では，一方のルートによって計算された $xP(g)$ と，もう一方のルートによって計算された $yP(g)$ が発生する．実際には，$P(g), x, y$ は，多項式で表されている．したがって，項の数は再収斂によって指数的に増加し，現実的な時間で剰余を求めることができなかったと考えられる．

3.2 節の手法では，のちに再収斂が発生するファンアウトポイントを 1 つの記号で保持するため，全てのゲートに記号を割り当てた場合のような指数的な増加は発生しない．さらに，**補正ルール 1** を計算中に適用することによって項の数が減少するため，計算時間やメモリの使用量の現象も見込むことができる．また，数式に対しても，3.1.2 と同様に，異なる入力間のでの計算結果の再利用ができる可能性がある．

第5章 おわりに

本研究では，組み合わせ回路における信頼性の評価方法について提案した．提案手法は，無閉路有向グラフの利用によって，メモリの使用量を削減し，グレイコードによって計算時間の面でも効率化を行った．ただし，回路内に再収斂が存在する場合，この手法では正確な確率を計算することはできない．そこでさらに，数式として計算し補正する手法と，そのための数式処理システムを提案した．

実験結果の結果，数値として計算を行う提案手法は，Probabilistic Transfer Matrix (PTM) を用いた既存手法に比べて，メモリの使用量を 2 万分の 1 に抑えられるケースも存在する．また，再収斂による誤差は，シミュレーションとの比較によれば，ほとんどのケースで 10% 以内に抑えられた．したがって，信頼性の評価に対して極めて深刻な誤差が発生するとは限らない．

既存手法では，指数的に増加する計算時間に対処するために，いくつかの入力をサンプリングしたヒューリスティクスも開発されている．提案手法についても，同様のヒューリスティクスが考えられる．このとき，3.1.2 を考慮すると，グレイコードによって，1 ビット違いの連続した入力をサンプリングした方が効率が良い．しかし，このサンプリング方法は，ランダムな入力を用いた場合に比べて偏りが生じるかもしれない．サンプリングの方法によって，どの程度の偏りが生じるかは，検討すべき課題である．

提案手法は，メモリの使用量を削減したことで，分散コンピューティングにも容易に応用できる．並列計算は，一般的にデータの転送時間がボトルネックとなりやすい．しかし，提案手法の計算を入力パターンで分割し，ぞれぞれの入力パターンについて同時に計算する手法を考えると，グラフ構造や数式のデータ構造を，各ユニットへ配布するだけでベクトル $\boldsymbol{s}$ を計算することができる．さらに，*fidelity* の l_1-ノルムは，いわゆる並列リダクションで処理できる．したがって，式 (3.1) の $\boldsymbol{v} \cdot \boldsymbol{s}$ の計算は，MapReduce [11] と呼ばれるプログラミングモデルを適用することができると言える．ベクトル $\boldsymbol{s}$ の計算と，ベクトル $\boldsymbol{v}$ との各要素同士の積を求めるまでが Map 操作に相当し，それらの合計を求めるのが Reduce 操作に相当する．分散コンピューティングで成功したモデルの一つである MapReduce との組み合わせによって，設計時における信頼性の計算はさらなる高速化が期待できる．

謝辞

「このロボットを原発事故の現場に持っていくことはできないんですか？」
「あぁ，それはね，電磁波や放射線の影響で誤作動しちゃうんだよ…」
何年か前のうろ覚えな記憶ではありますが，RoboCup Japan Open の会場でこのような会話を耳にしました．こうしてソフトエラーをテーマにした修士論文を書いてから思い返すと，僅かながら運命的なつながりを感じる部分があります．本研究のきっかけを与え，多大なるご指導をして頂きました，立命館大学 情報理工学部 山下茂教授に深く感謝しお礼申し上げます．

次世代コンピューティング研究室の皆様に深く感謝申し上げます．ユニークなイベントの数々は，他では味わえない貴重な経験になりました．あらゆる立場から接していただき，将来の人生を豊かにするための様々なヒント得ることができたと思います．共に過した多くの時間は，決して忘れることないでしょう．

情報理工学部プロジェクト団体や，立命館大学コンピュータクラブの皆様にもお礼申し上げます．志の高い先輩や後輩，そして同期の方々に出会い，成長の機会と経験を得ることができました．今後とも活躍を期待し，応援したいと思います．

最後に，大学生活を支えていただき，育てていただいた両親にお礼申し上げます．

参考文献

[1] Robert Baumann. Soft errors in advanced computer systems. *IEEE Des. Test*, Vol. 22, No. 3, pp. 258–266, May 2005.

[2] Smita Krishnaswamy, George F. Viamontes, Igor L. Markov, and John P. Hayes. Probabilistic transfer matrices in symbolic reliability analysis of logic circuits. *ACM Trans. Des. Autom. Electron. Syst.*, Vol. 13, No. 1, pp. 8:1–8:35, February 2008.

[3] Yusuke Matsunaga. On exact estimation of logic masking effect for soft erros on combinational circuits. *IEICE Technical Report. Image Engineering.*, Vol. 108, No. 229, pp. 53–58, sep 2008.

[4] R.I. Bahar, E.A. Frohm, C.M. Gaona, G.D. Hachtel, E. Macii, A. Pardo, and F. Somenzi. Algebraic decision diagrams and their applications. In *Computer-Aided Design, 1993. ICCAD-93. Digest of Technical Papers., 1993 IEEE/ACM International Conference on*, pp. 188–191, 1993.

[5] C. Y. Lee. Representation of switching circuits by binary-decision programs. *Bell System Technical Journal*, Vol. 38, No. 4, pp. 985–999, 1959.

[6] Shin-ichi Minato. Zero-suppressed bdds and their applications. *International Journal on Software Tools for Technology Transfer*, Vol. 3, No. 2, pp. 156–170, 2001.

[7] D.E. Knuth. *The Art of Computer Programming, Volume 4A: Combinatorial Algorithms*, p. 273. No. Part 1. Pearson Education, 2014.

[8] McElvain Ken and Graphics Mentor. LGSynth93 Benchmark Set: Version 4.0 ACM/SIGDA benchmarks (archived by Franc Brglez). `http://www.cbl.ncsu.edu:16080/benchmarks/LGSynth93/`.

[9] Intel Corporation. Support for processors. `http://www.intel.com/p/en_US/support/processors`.

[10] Christian Bauer, Alexander Frink, and Richard Kreckel. Introduction to the GiNaC framework for symbolic computation within the c++ programming language. *J. Symb. Comput.*, Vol. 33, No. 1, pp. 1–12, 2002.

[11] Jeffrey Dean and Sanjay Ghemawat. Mapreduce: simplified data processing on large clusters. OSDI ' 04 Proceedings of the 6th Conference on Symposium on Operating Systems Design and Implementation. USENIX Association, 2004.