システムＬＳＩ設計技術　93－9
(1999. 11. 27)

BDDの規模によらず一定の実記憶の範囲内で動作するストリーム形式BDD処理アルゴリズム

湊 真一　石原晋也

NTT 未来ねっと研究所
〒239-0847 神奈川県横須賀市光の丘1-1
{minato, shinya}@exa.onlab.ntt.co.jp

あらまし　従来のBDD処理系では，限界を超える大規模なBDDを扱うと記憶あふれを起こし動作不能になるという問題があった．本稿では，BDDの規模によらず一定の実記憶の範囲内で動作し仮想記憶にあふれ出すことのないBDD処理アルゴリズムを提案する．本手法は，ストリーム形式のBDDデータを入出力とし，主記憶は一時的なワークエリアとしてのみ使用する．実験結果によれば，従来のBDD処理系が苦手としていた問題ほど本手法は有効であり，従来の100分の1以下の主記憶容量でも効率良く動作する場合がある．さらに複数の処理系をパイプライン接続して並列動作させることも容易である．本手法は，これまでメモリ消費型のアプリケーションと考えられていたBDD処理系をディスク消費型またはネットワーク消費型に転換するものであり，今までにないBDDの応用形態を生み出す可能性がある．

キーワード　BDD, 論理関数, LSI, 最適化, 検証, 並列計算

Stream-type BDD Manipulation Algorithm with Data-Size-Independent Memory Requirement

Shin-ichi Minato　Shinya Ishihara

NTT Network Innovation Laboratories
1-1, Hikarinooka, Yokosuka-shi, 239-0847 Japan.
{minato, shinya}@exa.onlab.ntt.co.jp

Abstract　BDD-based programs commonly have the memory-overflow problem to manipulate large-scale BDDs. In this paper, we propose a new algorithm which never causes overflow nor swap out from the main memory in processing unlimited size of BDDs. The algorithm has stream-type BDD data I/O, and uses the main memory for a temporary working area. Experimental result shows that our new method is especially effective in the cases where usual BDD packages are ineffective. In some cases, the memory requirement is reduced into 100 times smaller in a feasible performance overhead. Our method is easily applied to pipelined parallel computing. BDD-based programs have been regarded as memory-consuming applications, but now they can be disk-consuming or network-consuming ones. It may lead new style of BDD applications.

key words　BDD, Boolean function, LSI, optimization, verification, parallel computing

1 はじめに

VLSIの論理設計や検証を始めとする数理工学分野の多くの問題において、論理関数データをコンパクトに表現し高速に演算処理を行うことは最も重要な基盤技術の1つである．BDD (Binary Decision Diagram, 二分決定グラフ) [3] は、現実の問題に現れる多くの論理関数を効率よく扱えるデータ構造として処理系の研究開発が盛んに行われ [2, 8, 10], 現在では様々な問題で広く実用化されている．

通常のBDD処理系では，BDDデータはランダムアクセス可能な主記憶内に構築される．しかし大規模な論理関数を扱おうとすると，演算処理を繰り返すにつれてBDDの節点数が増大し，遂にはメモリあふれを起こして異常終了するか，さもなければ極端に性能が低下する．BDDの節点数は実際に計算してみるまでわからないことが多いため，実用上の大きな問題となっている．

BDDを計算機の主記憶に格納しなければならない最大の理由は，BDD処理が「ハッシュテーブルによる節点の一意性の確保」という，主記憶への高速なランダムアクセスを基礎に成り立っているからである．これまでに，ランダムアクセス性を減らして2次記憶（仮想主記憶）を効果的に利用する手法 [9, 12] や，BDDを分割して保持し並列処理を行う手法 [6, 14] 等が提案されている．しかし，節点の一意性を保つハッシュテーブルだけはランダムアクセス性を取り除くことができないため，限られた容量の主記憶内に置く必要があり，これが節点数の上限が存在する理由となっている．

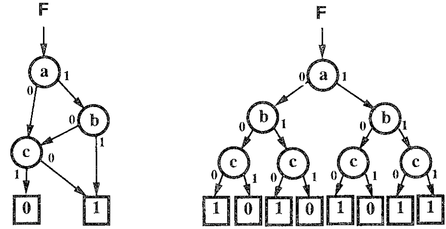
本稿では，BDDの規模によらず一定のハッシュテーブル容量の下で正しく動作するBDD処理アルゴリズムを提案する．すなわち，BDDの規模によらず一定の実記憶の範囲内で動作し仮想記憶にあふれ出すことがない処理系を実現できる．本手法は，従来の処理系のように主記憶内にBDDを構築していくのではなく，ストリーム形式のBDDデータを入力および出力とし，主記憶は一時的なワークエリアとしてのみ使用する．

従来のBDD処理系でも，主記憶内に構築したBDDデータをファイルに書き出して一旦保存し，後ほど再度読み込んで演算処理を続行する機能を持つものがある [10] が，主記憶容量を超える大規模なBDDデータを読み込んで演算処理を続行することはできなかった．これに対し，本処理系では主記憶に格納し切れない大規模なBDDデータであっても，これを読み込みながら同時に演算処理を実行することができる．演算結果のBDDデータは，入力を読み終えるのを待つことなく，演算結果が確定した部分から順次出力される．複数の演算処理をパイプライン接続して並列動作させることも可能である．

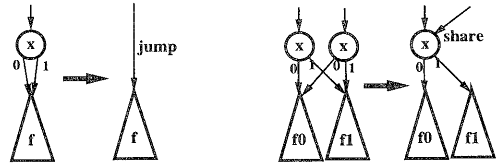
本処理系では，ハッシュテーブル容量が十分ある場合には，従来の処理系と全く同じBDDがストリーム形式で出力される．一方，ハッシュテーブル容量が不足する場合には，冗長な節点が複数回出力されることがあるため，出力データ長は増加する．その場合でも演算処理は正しく行われる．出力データ長はハッシュテーブルの容量不足の度合に応じて連続的に変化する．また計算時間は入出力データ長にほぼ比例する．

本手法は，主記憶容量による不連続な限界を持つ従来のBDD処理アルゴリズムを，容量不足の度合に応じて時間方向へ連続的に展開するものであると言える．さらに，これまでメモリ消費型のアプリケーションと考えられていたBDD処理系を，ディスク消費型あるいはネットワーク消費型のアプリケーションに転換するものであり，今までにないBDDの応用形態を生み出す可能性がある．

以下の本文では，まず2章で従来のBDD処理アルゴリズムとその問題点について述べ，続いて3章で新しく提案するストリーム形式BDD処理アルゴリズムおよびデータ構造を示す．4章で処理系の実装と性能評価について述べ，5章で応用例について述べる．最後に6，7章で関連研究と今後の課題をまとめる．



(a) BDD.　(b) 場合分け二分木.
図 1: $F = (a \wedge b) \vee \overline{c}$ を表す BDD と場合分け二分木.



(a) 冗長節点の削除.　(b) 等価な節点の共有.
図 2: BDD の簡約化規則.

2 従来のBDD処理系とその問題点

BDDは，図1(a) に示すような論理関数のグラフによる表現である．これは，論理関数の値を全ての変数について場合分けした結果を図1(b) のように二分木グラフで表し，これを縮約することにより得られる．このとき，場合分けする変数の順序を固定し，冗長な節点の削除 (図2(a)) と，等価な節点の共有 (図2(b)) を可能な限り行うことにより既約な形が得られ，論理関数をコンパクトかつ一意に表せることが知られている [1]．このようなBDDを特に ROBDD (Reduced Ordered BDD) と呼ぶ．現在実用化されているBDD処理系のほとんどはこのROBDDの技法にもとづいている．以下の本文ではROBDDのことを単にBDDと記す．

通常，計算機上にBDDを生成するには，まず最初に，ただ1つの節点からなる単純なBDDを各入力変数ごとに作り，続いて，与えられた論理式にしたがってBDD同士の論理演算 (AND, OR 等) の結果を表すBDDを生成する手続きを実行し，これを繰り返して，より複雑なBDDを順次構築して行く．BDD同士の論理演算アルゴリズムは，Bryant[3] により示されたもので，その基本的な手続きは以下の通りである．

1. 二項演算 $(f \circ g)$ において，2つのBDDを，ある入力変数 x について場合分けして，2つの部分演算 $(f_{(x=0)} \circ g_{(x=0)})$ と $(f_{(x=1)} \circ g_{(x=1)})$ に分解する．
2. 上記の分解をすべての入力変数について再帰的に繰り返して，最終的に自明な演算になったところで部分演算の結果のBDDを生成する．
3. 各部分演算で得られたBDDを再度組み上げることにより，全体の演算結果を表すBDDを構築する．

上記の分解と再構築を単純に実行すると，冗長な部分演算が多数生成される場合があるが，

- 生成したBDDの節点のデータ（変数と2つの枝の行き先）をすべてハッシュテーブルに登録しておき，同じ節点は重複して生成しない．

- 部分演算の結果を記録しておく「演算結果テーブル」を用意し，同じ部分演算がすでにあれば再帰呼び出しをせずに即座に結果を返す．

という2つの技法により，冗長な部分問題の処理を省いて高速化を行っている．その結果，BDDの節点数にほぼ比例する時間で論理演算を実行することができる．

以上に述べた手続きを計算機プログラムとして実装すると，BDDの節点は配列データとしてランダムアクセス可能な主記憶内に格納され，演算処理を繰り返すにつれて主記憶内でBDDが成長して行く形となる．多くのBDD処理系はCまたはC++のライブラリとして提供され，これを呼び出すアプリケーションプログラムとリンクされて1つのプログラムとして実行される．そして実行時にBDD格納用の主記憶が割り当てられる．計算機の主記憶容量は有限であるため格納可能な節点数の上限値が存在し，実行中にBDDが成長して上限を超えると異常終了するか，または極端に処理速度が低下して実用に適さなくなる．入力変数の順序を工夫することにより節点数を削減して記憶量を節約できるが，その効果にも限度がある．半導体技術の進歩により1度に利用できる主記憶容量は年々増加しているが，より大規模な問題を解きたいという要求はとどまることがないため，節点数の上限の存在は常に意識されることになる．

より大規模なBDDを扱いたいという要求にこたえるため，処理系の大容量化の研究がいくつか報告されている．これらを大きく分類すると，ハードディスク等の大容量の2次記憶を仮想主記憶として利用する方法と，複数の計算機を利用する並列処理法の2つに分けられる．前者に属する手法としては，通常の処理系がBDDの各節点を深さ優先順に処理するのと異なり，同じ変数レベル毎にまとめて節点をたどる「幅優先アルゴリズム」[9]が知られている．これによりハードディスクが苦手とするランダムアクセス性が減少するため，処理データが実記憶から仮想主記憶にあふれ出たときの極端な性能低下を防ぐことができる．さらに，深さ優先と幅優先のアルゴリズムを組み合わせて用い，より大規模なBDDを扱う処理系も報告されている[12]．しかし幅優先アルゴリズムを用いても，同じ変数レベルにある節点同士は一意性を保つために1つのハッシュテーブルに格納する必要があり，このハッシュテーブルだけは実記憶に納まることが前提条件となっている．したがって，BDDの横1列の節点数が実記憶容量を超える場合は有効に機能しないという問題が残っている．

一方，BDDを分割して複数の計算機に格納することで大容量化を図る方法もいくつか提案されている．BDDを上位の変数で場合分けして部分グラフに分割しそれぞれ独立に処理する方法[17]は一定の効果があり実用的であるが，部分グラフ間で節点を共有できないことと均等な分割が難しいという問題がある．それに対して，ハッシュ関数を用いて各節点をほぼランダムに分散させる方法[18, 11]は節点の一意性を保ちながら均等な負荷分散が行えるが，計算機間の通信が激しく発生することが大容量化の際に障害となる．また別の方法として，変数レベル毎に輪切りにして分散処理する方法[6, 14]があり，節点の一意性を保ちながら計算機間の通信コストも比較的小さく抑えられるので現実的であるが，幅優先アルゴリズムと同様に横1列の節点は同じハッシュテーブルに格納する必要があり，それ以上分割できないという問題がある．

以上のように，様々な大容量化の取り組みが行われているが，「ハッシュテーブルによる節点の一意性の確保」を基礎にしている限り，ランダムアクセスを完全に取り除くことはできず，節点数の上限という不連続な制限が存在し続ける要因となっている．

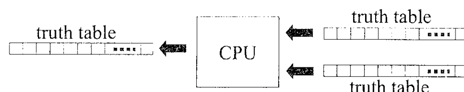


図 3: 真理値表のストリーム計算モデル．

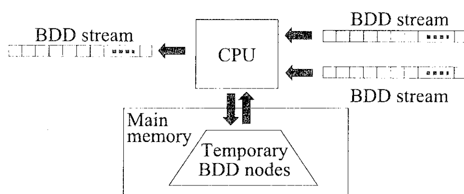


図 4: BDDのストリーム計算モデル．

3 ストリーム形式BDD処理アルゴリズム

本稿で提案する手法は，従来の処理系のように主記憶内にBDDを構築していくのではなく，ストリーム形式のBDDデータを入力および出力とし，主記憶は一時的なワークエリアとしてのみ使用する．本手法により，BDDの規模によらず一定の実記憶の範囲内で動作し仮想記憶にあふれ出すことがない処理系を実現できる．以下にその動作原理を述べる．

3.1 ストリーム計算モデル

まず，図3のように論理関数を真理値表により表現し，所定の順序で並べてビット列のストリーム形式にしたものを考える．これを先頭から順に読み込んでビット同士の論理演算を行い，演算結果を順次ストリームとして出力することによって，データを格納する内部記憶を全く持たなくても演算処理を行うことができる．しかしこの方法では入力変数の個数nに対して常に$O(2^n)$の計算時間を要する．

そこで図4のように，BDDをストリーム形式のデータとして先頭から順次処理を行うことを考える．上記の真理値表のストリームを読むことは，論理空間を所定の順序でたどることに他ならない．これはBDDにおいては，グラフを深さ優先順にたどることにより実現できる．したがって，BDDの節点を深さ優先順にたどって列挙したものをストリームとすれば，真理値表の場合と同様に内部記憶を全く持たなくても演算処理が行えることになる．

BDDが真理値表と異なるのは，部分グラフの共有や冗長節点の削除によりデータを簡約化しているという点である．もしも共有された部分グラフを毎度重複してたどれば，真理値表と同様に指数関数的な計算時間となる．しかし，グラフをたどりながら各節点に一意に番号をつけて行き，同じ節点に再度到達した場合には「以下はk番目の節点と等価」とだけ出力すれば，同じ枝を重複してたどらずに済むため，計算時間はBDDの節点数に比例する長さに抑えられる．これは言い換えれば，過去に処理した部分的論理空間をラベルづけして記憶しておいて，必要なときに参照するということである．一方，0-枝と1-枝が同じ行き先を指すような節点は削除するという簡約化に関しては，等価な節点が2回続けて出現したときに2回目の出力を省略するという操作に対応する．

もしも十分な主記憶容量があれば，すべての節点をハッシュテーブルに登録することができるので，ストリーム入出力の計算モデルでも通常のBDD処理系と同じ演算処理を実行できる．しかし，ハッシュテーブル容量が十分でない場合は，すべての節点を登録することができないため，本来ラベル参照だけで済むはずの等価な部分グラフを見失い，重複して節点をたどる場合が生ずる．これはすなわちストリームデータの圧縮率が低下することを意味する．容量不足の度合が大きい

```
Stream ::= MaxID Inv Node
Inv ::= '~' | /* empty */
Node ::= SavedNode | TempNode
SavedNode ::= '0' | ID
        | '(' SavedNode ')'
        | '(' SavedNode Inv SavedNode '):' ID
TempNode ::= '(' Node Inv Node ')'
MaxID ::= [1-9][0-9]*
ID ::= [1-9][0-9]*
```

図 5: BDD ストリームデータの文法.

```
x1                           (0~0):1
x2                           ((0~0):1)
AND(x1, x2)                  (0(0~0):1):2)
XOR(x1, x2, x3)              (((0~0):1~1):2~2):3.
ITE(x1, x2, x3)              (((0~0):1)(0~0):2):3.
Majority(x1, x2, x3) ((0(0~0):1):2(1~0):3):4.
```

図 6: 基本的な論理関数の記述例.

ほど, 等価な節点を発見する機会が失われ, データの圧縮率は低下して行く. ただし, 容量が最悪ゼロであったとしても, 真理値表のストリーム処理と等価になるだけであるから, 効率は悪いが正しく動作することは保証される. 従来のBDD処理系のように容量限界を超えると不連続的に機能が失われるのではなく, 正しく動作しながら処理効率が連続的に変化するところが大きく異なる点である.

3.2 データフォーマット

ストリーム形式のBDD処理系を実装するために, まず入出力データフォーマットを定義した.

最初に, データを出力する側とそれを読み込む側との間で, 主記憶内のテーブル容量を合わせておく必要がある. 本処理系では, 出力データの先頭に容量値MaxIDを明示することとし, 読み込み側はその容量値を見て主記憶領域を確保する. 各節点は1からMaxIDまでの自然数でラベルづけを行う. 0番の節点は特に0-終端節点を表す. 本処理系では否定枝[8]を使用するので1-終端節点は0の否定で表す.

本フォーマットの文法をBNF風に記述したものを図5に示す. 1つの節点は2つの枝が指す節点の組を括弧で囲んで表し, 括弧の入れ子によりグラフ構造を表現する. 否定枝の記号「~」は直後の節点を修飾する. 節点をテーブルに格納する際は, 右括弧を閉じてから「:」を挟んで節点のIDを定義する. 文法記述において, SavedNodeはテーブルに格納されている節点を表し, TempNodeは格納されずにすぐ失われる節点を表す. SavedNodeの内側の節点はやはりSavedNodeでなければならない. またIDの参照はその定義より後でなければならない.

冗長な節点の飛び越しは, 括弧の間に節点を1つだけ記述することにより表す. 括弧の個数を数えることにより各節点の入力変数のレベルがわかるので, 入力変数は陽には記述しない. 以上の規則にもとづき基本的な論理関数を表した例を図6に示す. ITEは2-to-1データセレクタ, Majorityは多数決論理を表している.

本フォーマットはBDDの全節点を深さ優先順にたどって列挙したものであるから, テーブル容量が十分あれば, 通常のBDDと同様に論理関数を一意に表現できることは明らかである. 一方, テーブル容量が不足する場合には, 同じ論理関数でも容量によっては異なる表現となる. 例えばMCNC benchmarkの "9sym" のBDDストリーム表現を図7に示す. 上から順にテーブル容量が30, 20, 10の場合を表している. この関数のBDDは24個の節点からなるが, テーブル容量がこれより少ないと節点のIDが途中で足りなくなる. この

```
30
((((((0(0(0~0):1):2):3(2(1~0):4):5):6(5(4~0):7)
:8):9(8(7~0):10):11):12(11(10~(0 3):13):14):15)
:16(15(14~(13 6):17):18):19):20(19(18~(17 9):21)
:22):23):24.

20
((((((0(0(0~0):1):2):3(2(1~0):4):5):6(5(4~0):7)
:8):9(8(7~0):10):11):12(11(10~(0 3):13):14):15)
:16(15(14~(13 6):17):18):19):20(19(18~(17 9):20)
:16):12).

10
((((((0(0(0~0):1):2):3(2(1~0):4):5):6(5(4~0):7)
:8):9(8(7~0):10):9)(9(10~(0 3):6):9))(((8 10):9
(10~6):9)(9~(6(3 5):8):9)))(((((5 7):10(7~0):9)
:6(9~(0 3):8):6)(6~(8(3 5):10):6))(((9~8):6~(8
10):6)~(6(10(5 7):9):6)))).
```

図 7: "9sym" のBDDストリーム記述.

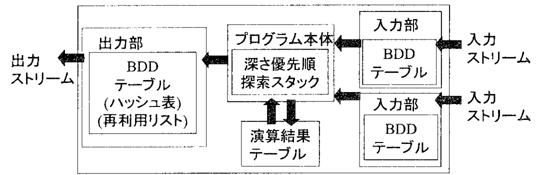


図 8: 論理演算プログラムの内部構成.

とき本処理系では, まだ他から参照されていない「浮いた」節点を見つけ出して, その節点を消去してIDを再利用している. 例えば図7で容量が10のときは, 1番から10番までの節点はそのままテーブルに格納するが, 11番目の節点は格納できないため, まだ誰からも参照されていない9番の節点を消去して上書きしている. その結果, 昔の9番の直下にあった6番の節点が誰からも参照されなくなるので, 次に再利用可能となる. これを繰り返すことにより限られたテーブル容量を有効に使うことができる. 我々の実装では, 再利用可能な節点が複数ある場合は, 最も長い間参照されていない節点から順次取り出すようにしている. プログラムは決定的な動作をするので, ある固定したテーブル容量の下では, BDDストリームデータは論理関数に対して一意な表現となる.

3.3 論理演算アルゴリズム

ストリーム形式のBDDの論理演算は, 図8に示すように, 2本のストリームファイルを順次読み込みながら, 演算結果のBDDを表すストリームファイルを出力する形式となる. その内部データ構造は, 入力される2つののBDDをそれぞれ一時的に格納するテーブルと, 計算結果のBDDを一時的に格納するテーブルからなる. 入力部のテーブル容量は読み込み, 先頭のMaxIDを見て自動的に主記憶上に確保される. 一方, 出力部のテーブル容量は, 次段の処理系の能力に応じて, プログラム起動時にユーザが指定する. このとき2つの入力部と出力部のテーブル容量はそれぞれ異なっていてもよい. 入力部のテーブルは節点のIDをアドレスとする単純な配列構造でよい. それに対して出力部のテーブルには, 生成した節点の一意性を確保するためのハッシュテーブル機構と, 他の節点から参照されている枝の本数を数える参照カウンタ, および最も長い間参照されなかった節点を取り出すためのキューの機構を備える必要がある.

入力部においては, プログラム本体から「1ステップ読み進め」という指令が与えられる毎に, 深さ優先順にBDD節点を1つずつ順にたどる. 初めはストリームファイルから入力される節点を順次読んでたどり, たどった節点のIDが定義されたときにはこれをテーブルに格納する. その後, 格

納した節点が再び参照されたときには，その節点の下の部分グラフについては，ストリームファイルからの読み込みを一時中断し，テーブル内の節点データをたどっていく．部分グラフをたどり終わって元の節点に戻ってきたら，ストリームファイルからの読み込みを再開する．

プログラム本体は，２つの入力部に順次指令を出し，両方のBDDを同期させながら深さ優先順にたどっていく．その過程で，片方または両方のBDDが終端節点に到達して自明な論理演算になったときには演算結果の終端節点を出力する要求を出力部に送る．さらにバックトラックして，0-枝，1-枝側双方の部分グラフをたどり終わったところで，それらの演算結果を表す節点を出力するように出力部に要求を送る．この処理を全ての論理空間をたどり終わるまで繰り返す．

出力部においては，プログラム本体から「ある節点を出力せよ」という要求が送られると，その節点と等価な節点がすでにハッシュテーブルに登録されているかどうかをまずチェックする．もしも新しい節点であればハッシュテーブルに新規登録するとともに，0-枝と1-枝の指す節点IDの組を括弧で囲んで出力し，続けてその節点IDを出力する．一方，等価な節点がすでに存在するならば，より上位の新しい節点がみつかるまで何も出力せずにバックトラックを続行する．

前節で述べたように，生成するBDDの規模に対してハッシュテーブル容量が十分でない場合には，節点の一部を消去して上書きすることとした．そのため，ある節点の0-枝側の部分グラフの情報が，1-枝側の部分グラフをたどっている間に上書きされてしまうという事態が起こり得る．これをチェックするため，すべての節点にタイムスタンプを設けて，上書きされるごとにタイムスタンプを更新することとした．そして0-枝側のタイムスタンプが1-枝側を処理する前後で異なっている場合はテーブルあふれが起きたと判断し，その節点を登録することを禁止する．その節点を直接間接に参照している上位の節点もテーブル登録は禁止され，生成と同時に直ちにストリーム出力に書き出される．さらに，0-枝と1-枝の指す節点が同じ場合は重複せずに１度だけ出力するという簡約化についても，テーブル登録できなかった節点については適用できないため，２度重複して出力しなければならない．したがってテーブル容量が不足するほど圧縮率が低下し出力データ長は長くなる．

上記の論理演算処理においては，従来のBDD処理系と同様に，２章で述べた演算結果テーブルにより冗長な部分演算の処理を省略して高速化を図ることができる．これにより，入出力するBDDストリームのデータ長にほぼ比例する計算時間で演算処理を実行できる．ただし，入力ストリームの読み込みは省略できないので，２つの入力部がともにテーブル内の節点をたどっているときにのみ，部分演算処理の省略が許される．さらに，同じ節点IDでもタイムスタンプが異なっている場合は，ヒットしないようにしておく必要がある．

ここまで二項論理演算について述べたが，入力処理部を３つに増やせば，多数決論理や2-to-1データセレクタのような三項論理演算を一度に処理できるようになる．その場合，入力BDDを格納するテーブルを１つ余計に使うため主記憶を多く必要とするが，二項演算を繰り返して三項演算を実現するのに比べると処理時間やディスク使用量を削減できる．さらに４以上の入力を持つ演算も原理的には可能である[1]．

3.4 代入演算

与えられたBDDに対して，ある変数xに０（または１）を代入したときのBDDを求める計算も，BDD処理系の基本的な機能の１つである．BDDの最上位の変数に代入する場合，通常のBDD処理系では0-枝（または1-枝）が指す節点を返す処理となる．これをストリーム処理系に置き換えると，ストリームの前半部（または後半部）を取り出し，両端を括弧で囲んで出力するという形になる．ただしこれは節点を格納するテーブル容量が十分ある場合の話で，そうでなければ前半部（または後半部）を２回繰り返して出力する必要がある．つまり入力のBDDストリームを２度読みする必要が生じるので，読み終わったデータをすぐ捨てることはできず，ハードディスクのような静的記憶にストリームを一時保存しておく必要がある．

BDDの変数に値を代入する操作は，ストリームのデータ順序の入れ換えを伴うので，ストリーム形式の処理系にとっては負荷の大きい処理である．これはBDDの変数順序を途中で入れ換えたり，代入したBDD同士の論理演算（例えば$\forall x f(x)$など）を行う場合についても同じことが言える．

なお，論理関数の一部または全部の入力変数に0,1の定数値を代入した場合に関数が充足可能(Satisfiable)かどうかを調べるだけであれば，代入演算を使う必要はなく，論理積演算のみを用いて判定することができる．

4 処理系の実装と評価

以上に述べたストリーム形式BDD処理系をUNIX上に実装した．処理系はテキストファイルを入出力とする以下の３つのプログラムからなる．

- BDDprime: 定数関数または１変数関数のBDDストリームを出力する．
- BDDnot: 入力されたBDDの否定のBDDストリームを出力する．
- BDDstrm: ２つまたは３つのBDDストリームを読み込んでその論理演算結果のBDDストリームを出力する．

BDDprimeは与えられた変数番号に応じて適当な個数の括弧で囲んだ１節点を出力する．BDDnotはストリームの先頭の否定記号を反転させる他は入力をそのまま出力する．BDDstrmは本処理系の核心部で，３章で説明した論理演算を行う． プログラムの規模はC言語で約2000行である．BDDprimeで生成した単純なBDDに対してBDDstrmおよびBDDnotを繰り返し適用することにより，より複雑なBDDを構築して行く．これらはUNIXのパイプでつないで並列動作させることも可能である．

BDDstrmを起動する際には，内部テーブル容量（出力する節点IDの最大値）を指定することができ，それに応じて必要な主記憶が割り当てられる．32ビットマシンでは，１ワードで表現できる節点IDは2^{32}を超えられないが，本処理系ではIDの再利用ができるので，2^{32}を上回る数の節点を処理することが可能である．このようにビット幅の壁を越えられることも本処理系の特長の１つである． また，計算の結果，膨大な長さのデータが出力されてハードディスクが使い尽くされるのを防ぐため，起動時に出力データ長の上限を指定して，それを超えると途中で打ち切る機能を持たせた．このとき論理空間全体の何%を処理し終わったかを計算して表示することができる．さらに，生成したBDDストリームのデータ長を非圧縮の真理値表と比較した圧縮率も計算して表示できる．

なお上記のプログラムとは別に，従来のBDD処理系で主記憶内に生成したBDDをたどって，ストリーム形式ファイルに変換して出力するプログラムも合わせて作成した．これにより処理系の動作の確認や，従来手法との性能比較を行うことができた．

[1] プログラムが繁雑になるので，我々の実装では３入力までにとどめている．

10 × 10 ビット乗算器（第11ビット出力）　入力数：20

容量	50000	10000	5000	1000	500	100	50	10
節点数	10573	10573	11286	15203	16082	19010	35613	120665

8-Queens 問題の解集合関数　入力数：64

容量	5,000	1,000	500	100	50	10	5	1	0
節点数	2,450	2,551	2,760	3,402	3,651	3,774	3,830	3,996	4,087

9sym　入力数：9

容量	30	20	10	5	3	2	1	0
節点数	24	24	43	81	112	136	164	219

26 入力パリティ関数

容量	26	24	22	20	18	16	14	12	10
節点数	26	27	37	83	273	1,039	4,109	16,395	65,545

表 1: テーブル容量と出力データ長（節点数）の関係.

4.1 単一マシンでの性能評価

我々が実装したプログラム BDDstrm の性能評価の結果を示す．まず BDD 1 節点当たりの主記憶使用量は，入力部のテーブルで約 12 バイト，出力部のテーブルで約 31 バイトである．二項演算の場合は入力部が 2 つあるのでそれらを加算する必要がある．例えば，主記憶 64MByte の計算機では，入出力部でそれぞれ 80 万〜 100 万節点程度のテーブルを割り当てることができる．

次に BDD の節点数に対する入出力データ長を調べた[2]．節点 ID の桁数や連続する括弧の数[3]により値は変わるが，だいたい 5 〜 15 バイト / 節点となる．実際の例に当てはめると，100 万節点の BDD をストリーム化すると約 10M バイト，1 億節点で約 1G バイトとなる．

最後に演算速度を評価した．本処理系の計算時間は入出力データ長にほぼ比例する．IBM-PC 互換機（Celeron 300A, 64MByte, FreeBSD 2.6）において，種々の BDD ストリームを流してそのスループットを調べたところ，約 300k 〜 500kByte/sec (2.5 〜 4Mbps) という速度であった．これは節点数に直すと，毎秒 3 万〜 5 万節点を処理している計算になり，30 分〜 1 時間程度で 1 億節点に達する．

主記憶内で動作する従来の BDD 処理系と速度を比較した場合，本処理系は数倍〜 10 倍程度低速であると見積もられる．原理的には従来手法と同じ深さ優先アルゴリズムにもとづいているので，単一マシンで実行する限りは従来手法よりも高速になる要素はなく，ストリームの文法解析やスタック処理の複雑化がネックとなって，何倍かのオーバヘッドが生じている．しかし，従来の処理系はハッシュテーブルが実記憶に納まることが必要条件となっており，この条件が満たされなければ異常終了するか，極端な性能低下を起こす．これに対し本処理系は，ハッシュテーブル容量を超える規模の BDD にもある程度耐えられることを考慮すると，用途によっては上記のオーバヘッドは十分許容できる値と言える．

4.2 テーブル容量と出力データ長の関係

3 章で述べたように，本処理系は内部のハッシュテーブル容量を超える規模の BDD を正しく演算処理できるが，容量不足の度合に応じて圧縮率が低下し，出力データ長が増大する．そこで，実際にどの程度データ長が増大するかを調べた．実験では，あらかじめ十分なテーブル容量で生成した BDD ストリームファイルを用意し，同じデータを不十分なテーブル容量値を指定して出力した場合に，データに含まれる節点数がどのように増大するかを調べた．表 1 に結果を示す．10

C432- 第 7 出力関数　入力数：36

容量	100000	10000	1000	300	200	100	50
順序 A	378	378	378	526	26905	1699872	15749782
順序 B	5442	5442	36924	158059	266090	674751	1726289
順序 C	58135	134462	337560	1140000	1807297	3268024	10025073

vg2- 第 5 出力関数　入力数：25

容量	1,000	500	300	200	100	50	30	20	10
順序 A	60	60	60	60	60	151	1,158	22,722	125,319
順序 B	495	495	1,058	2,970	7,273	12,825	21,348	44,421	110,737
順序 C	689	1,216	1,570	1,876	2,332	2,809	3,812	5,812	31,404

表 2: 入力変数順序による節点数増加の違い.

× 10 ビットの乗算器関数の例では，テーブル容量を本来の BDD 節点数 (10,573) のわずか 1% まで減らしても，たかだか 2 倍程度の増加 (19,010) にとどまっている．これは，乗算器の BDD には共有される節点が少ないため，過去にたどった節点を消去して上書きしても影響が非常に小さいことを示している．8-Queens 問題の解を表す関数では，92 通りの解を表す入力以外はすべて 0 を返すため，やはり共有される節点が少なく，テーブル容量を完全に 0 としても，出力データ長は 2 倍にしかならない．一方，9sym は対称関数であり，節点の共有が比較的多い例であるが，この場合はテーブル容量を約半分に減らすとデータ長は 2 倍に増えている．最後のパリティ関数は節点の共有が極端に多い例で，内部テーブルの節点数を 1 減らすごとにデータ長が指数関数的に増大している．以上の観察から，BDD の節点共有が多いほど，容量不足に対する圧縮率の低下が激しいことがわかる．言い換えると，従来 BDD が苦手としていた論理関数ほど，少ない主記憶容量で処理できるということになる．

次に，入力変数の順序づけがデータ圧縮率にどのような影響を与えるかを実験した．結果を表 2 に示す．MCNC ベンチマーク回路 C432 と vg2 について出力関数の BDD ストリームをそれぞれ異なる変数順で生成し，同じようにテーブル容量不足に対するデータ長増大の様子を調べた．実験結果から，同じ論理関数でも変数順により圧縮率低下の仕方が異なることがわかる．その結果，平常時に最適な変数順がメモリ不足時にも良い順序とは限らないという興味深い現象が起こっている．平常時もメモリ不足時も共に有効な変数順序を考える必要がある．

本処理系では，BDD をストリーム化する際に変数順序を固定する必要があるため，演算の途中で動的に変数順序を入れ換えることは困難である．ただし従来の処理系のように，節点数が少しでも上限を超えると続行不可能になるわけではないので，最初に発見的手法で比較的良い順序づけを求めておけば十分な場合も多いと考えられる．一般に，変数順の最適化には多大な計算時間を要するので，コストと効果のトレードオフを考慮する必要がある．

4.3 パイプライン接続による並列実行

本処理系は入力と出力が同時並行的に進行するため，複数の演算処理をパイプライン接続して並列動作させることが可能である．個々の演算処理は単純なストリーム入出力を持つ UNIX コマンドとして実現されているため，特別な並列ソフトウエア環境を用意しなくても，複数ホストにまたがる並列動作を容易に実現できる．例えば，互いに rsh コマンドが利用可能な UNIX マシンが複数台あれば，mkfifo コマンドと cat コマンドを組み合わせて用いることにより，複数ホストにまたがるパイプライン処理を実現することができる．各ホスト間の通信路は単純なストリームを通すだけなので，数 Mbps のバンド幅が確保されていればよい．ストリーム処理では通信遅延は計算時間にほとんど影響しないので，ホスト同士が地理的に離れていても構わない．

[2] 我々の実装では可読性を優先してテキストファイル形式としたが，バイナリ形式ならばデータ長は約半分になる．

[3] 我々の実装では括弧が 100 個連続するような場合には "(*100" のようなランレングス記述を導入して効率化を図っている．

並列動作を調べるため，$n \times n$ ビット乗算器回路を与えて $2n$ ビットの出力を表す BDD ストリームを生成する実験を行った．回路は n^2 個の全加算器を行列型に並べた最も基本的なものである．個々の全加算器は 3 入力 XOR ゲートと 3 入力多数決ゲートで構成した．これを以下の 3 通りの方法で実験を行った．

- 1 台のホストで逐次処理．（1 演算ごとに結果をディスクに保存）
- 2 台のホストで全加算器の内部の XOR ゲートと多数決ゲートを並列に実行．（各加算器ごとに結果をディスクに保存）
- $2n$ 台のホストで 1 列分の部分積の加算を一度に並列実行した．（桁上げ信号を直結．n 回繰り返し）

実験では，複数の IBM-PC 互換機（Celeron 300A, 64MB, FreeBSD 2.6）を 100BaseT の Ether で接続したものを使用した．内部テーブル容量は 70 万節点とした．

$n = 13$ のときに，計算時間 (elapse time) を計ったところ，1 ホスト版で約 23 分，2 ホスト版で約 15 分，$2n$ ホスト版で約 9 分という結果となった．$2n$ ホスト版で台数に比べ速度が伸び悩んでいるが，これは各ホストで処理する BDD の大きさに偏りがあって最も時間がかかるホストに律速されるということと，節点の共有によって BDD データが圧縮されるためにストリームのスループットが一定ではなく，データが完全に円滑に流れていないという 2 つの原因があると考えている．より高い台数効果を得るには，もう一工夫が必要と思われる．それでも，パイプライン並列化により，途中の計算結果を保存するハードディスク容量を節約できることは大きな利点である．

なお，13×13 ビット乗算器の BDD 節点数は数百万に達するため，従来の BDD 処理系では主記憶 64MB の計算機でこれを全て格納することは不可能である．ただし「幅優先アルゴリズム」を用いてハードディスクを仮想記憶として上記と同じ実験を行った報告 [9] があり，本手法と同等またはやや高速な結果が得られている．これは，幅優先アルゴリズムでは横 1 列の BDD 節点が主記憶に納まるという仮定を置いて，この性質を有効利用しているためである．これに対して本手法は，いかなる規模の BDD に対しても主記憶あふれを起こさないという，より厳しい制約の下でほぼ同等の処理性能を実現していると言える．

5 応用例

5.1 論理回路からの BDD 生成

論理回路の等価性判定や記号シミュレーションは BDD の典型的な応用の 1 つである．等価性判定を行う場合は 2 つの回路からそれぞれ BDD を作り，その排他的論理和を調べれば良い．もし 2 つの回路が等価であれば，最終結果の BDD ストリームの節点数は 0 になるはずである．逆に節点が 1 つでも出力されれば，処理が完了しなくても直ちに等価でないことがわかる．

本処理系では演算処理中の BDD の規模が主記憶容量を超える場合でも，ストリーム長が伸びるだけで，途中終了することなく処理を進めることができる．前に示した実験結果（表 1, 表 2）に示すように，従来の BDD 処理系が苦手としていた論理関数ほど本手法は有効であり，従来の 100 分の 1 以下の主記憶容量でも効率良く動作する場合がある．

組合せ回路から BDD を生成する場合，外部入力に近いゲートから順に論理演算を実行して回路の内部論理を表す BDD を生成して行く．このとき，最初のうちは BDD の規模が小さいので，ストリーム形式の処理系を使うよりも従来の主記憶上の処理系を使う方が効率が良い．そこで BDD の規模がある限界値に達するまでは従来の方法で BDD を生成し，限界に達したら内部論理を表す BDD をすべてストリーム形式に変換して，あらかじめ指定されたディレクトリに保存する．さらに残りの論理演算の手順を表す Makefile を合成して一旦終了する．あとはこの Makefile にしたがって make コマンドを実行すれば自動的にストリーム処理系が起動され最終結果の BDD ストリームが生成される．

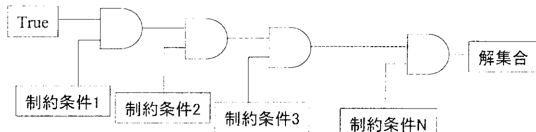


図 9: BDD ストリーム演算による制約充足問題の解法．

5.2 制約充足問題への適用

LSI CAD を含む数理工学上の多くの問題は，複数の制約条件を全て満たすような論理値の組合せを求める「制約充足問題」に帰着される．よく取り上げられる例題としては，N-Queens 問題，巡回セールスマン問題，グラフマッチング問題，最小フロー問題，0-1 整数計画問題，等がある．これらの問題は，複数の制約論理式の集合として記述できる．なお，Boolean Satisfiablity (SAT) 問題は，上記の制約論理式を入力変数の論理和の形式のみに制限したものであり，制約充足問題の特殊な例である．

BDD を使った制約充足問題の解法については文献 [7, 16] に詳しく述べられている．各制約論理式を表す BDD を作り，これら全体の論理積を表す BDD を生成できれば，それがすなわち解集合を表す．しかし大規模な問題に対しては記憶あふれを起こし結果を得ることができないという欠点がある．これまでの研究から，BDD を用いた解法は，すべての解を列挙したり解の総数を求める場合には有効であるが，多数の解の中の 1 つを見つければ良いという問題には不利という結果が得られていた．

しかし，ストリーム形式の BDD 処理系を用いると，全解探索の効率を低下させることなく，早い段階で解の一部を見つけることが可能となる．図 9のように，各制約式を BDD ストリームで表しておいて，これらをパイプライン状に接続して論理積演算を実行することにより，全体の解集合を表す BDD ストリームを生成できる．演算途中の BDD ストリームは，それより上流にある全ての制約式を満たす「解の候補」を表している．言い換えると，各論理積演算は解の候補から制約式を満たさない要素を取り除くフィルタとなっていて，最終的に全ての制約を満たす解が抽出されて順次出力される．もしも解が存在しなければ節点数 0 の BDD となるので，1 つでも節点が出力されればその時点で解が存在することがわかる．計算量の大きい問題を与えると出力 BDD データ長はいくらでも長くなり，何日経っても出力が終わらない場合もあるが，いつ処理を中断してもそれまでに出力された BDD データは有効な部分解を表している．従来の BDD 処理系では処理を中断すると部分解すら得られなかったのに比べると大きな違いである．

例えば文献 [16] によれば，N-Queens 問題の解集合を従来の BDD 処理系で求めようとすると，主記憶 1GB のマシンを用いても $N = 14$ までが限界であった．しかし本処理系を用いれば，主記憶 64MB の普通の PC でも同じ解集合を生成可能である．さらに $N = 17$ とした場合，従来手法では 100GB 以上の非現実的な容量の主記憶が必要になると予想されるが，本処理系では普通の PC を 1 時間程度動かすだけ

で最初の数千個の解を生成することができる．

なお，個々の論理積演算をすべて異なるホストに割り当てて一度に並列実行できれば理想的であるが，それができない場合は途中のBDDストリームをディスクに一時保存してプロセスを終了し，これを読み出して次段以降の演算処理を続行することになる．もしも途中のストリームが長くなり過ぎてディスクに入り切らない場合は，適当な長さで打ち切って次段に進むことができる．その場合，打ち切ったところまでの部分空間の探索となる．

6 関連研究

本処理系は，論理関数データを圧縮してストリーム処理するという点で，既存のデータ圧縮技術と関連が深い．現在広く用いられている compress, gzip, lha 等のファイル圧縮ソフトは，1977年頃に発表された Lempel-Ziv 符号 [13] を基本にしている．その原理は，過去に入力された文字列の履歴を主記憶上のテーブルに格納しておき，入力文字列データを順次読み込みながら，テーブルを検索して同一の部分列を検出し，その格納番地のみを符号として出力することによりデータを圧縮するという方法である．部分列の区切り方や辞書の構造により様々なバリエーションがあり [19]，処理速度や圧縮率が長年に渡って競われてきた．理論的にも各種符号の平均冗長度が詳しく調べられている [15]．

BDDストリームは，BDDの構造にもとづき真理値表データを部分列に分解して辞書に格納するデータ圧縮符号の一種であると言える．ただし通常のデータ圧縮法では，圧縮されたデータを解凍しなければ意味のある演算処理が行えないのに対し，BDD処理系ではデータを圧縮したままで圧縮データ長に比例する時間で論理演算を行うことができるところが大きな特長である．データ圧縮符号の観点からBDDの情報理論的解析を行うことも興味深いテーマと思われる．

7 おわりに

半導体メモリの集積度は限界に近づいており，従来と同じペースで10GB, 100GBと大容量化を続けることは難しいと思われる．しかしランダムアクセス記憶の容量を制限し，それを超える処理はシーケンシャルアクセスだけで実現することにすれば，低コスト大容量なハードディスクの利用や，通信を介した並列処理の可能性が広がる．本手法は，これまでメモリ消費型のアプリケーションと考えられていたBDD処理系をディスク消費型またはネットワーク消費型に転換するものであり，今までにないBDDの応用形態を生み出す可能性がある．

今後の課題としては，ストリーム処理系にとってコストの高い演算（代入演算や変数順序入れ換え等）をどう扱うかが挙げられる．また，単純なBDDに限らず，Multi-Terminal BDD[4] や Edge-Valued BDD[5] のような種々のデータ構造への拡張を考えることも興味深い．

謝辞

討論いただいたNTT未来ねっと研究所の宮崎敏明氏，高原厚氏，谷誠一郎氏に感謝いたします．

参考文献

[1] S. B. Akers. Binary decision diagram. *IEEE Trans. on Computers*, C-27(6):509–516, June 1978.

[2] K. S. Brace, R. L. Rudell, and R. E. Bryant. Efficient implementation of a BDD package. In *Proc. of 27th ACM/IEEE Design Automation Conference (DAC'90)*, pages 40–45, June 1990.

[3] R. E. Bryant. Graph-based algorithms for Boolean function manipulation. *IEEE Trans. on Computers*, C-35(8):677–691, Aug. 1986.

[4] E. M. Clarke, K. L. McMillan, X. Zhao, M. Fujita, and J. Yang. Spectral transforms for large Boolean functions with applications to technology mapping. In *Proc. of 30th ACM/IEEE Design Automation Conference (DAC'93)*, pages 54–60, June 1993.

[5] Y.-T. Lai, M. Pedram, and S. B. Vrudhula. FGILP: An integer linear program solver based on function graphs. In *Proc. of IEEE/ACM International Conference on Computer-Aided Design (ICCAD-93)*, pages 685–689, Nov. 1993.

[6] K. Milvang-Jensen and A. J. Hu. BDDNOW: A parallel BDD package. In *Formal Method in Computer-Aided Design (Proc. of FMCAD-98)*, LNCS-1522, pages 501–507. Springer, June 1998.

[7] S. Minato. BEM-II:an arithmetic Boolean expression manipulator using BDDs. *IEICE Trans. Fundamentals*, E76-A(10):1721–1729, Oct. 1993.

[8] S. Minato, N. Ishiura, and S. Yajima. Shared binary decision diagram with attributed edges for efficient Boolean function manipulation. In *Proc. of 27th ACM/IEEE Design Automation Conference (DAC'90)*, pages 52–57, June 1990.

[9] H. Ochi, Y. Kouichi, and S. Yajima. Breadth-first manipulation of very large binary-decision diagrams. In *Proc. of IEEE/ACM International Conference on Computer-Aided Design (ICCAD-93)*, pages 48–55, Nov. 1993.

[10] F. Somenzi, et al. CUDD: CU decision diagram package. Public Software. http://vlsi.colorad.edu/~fabio/CUDD.

[11] T. Stornetta and F. Brewer. Implementation of an efficient paralell BDD package. In *Proc. of 33th ACM/IEEE Design Automation Conference (DAC'96)*, June 1996.

[12] B. Yang, Y.-A. Chen, R. E. Bryant, and D. R. O'Hallaron. Space- and time-efficient bdd construction via working set control. In *Proc. of Asia and South Pacific Design Autonation Conference (ASPDAC-98)*, pages 423–432, Feb. 1998.

[13] J. Ziv and A. Lempel. A universal algorithm for sequential data compression. *IEEE Trans. Inform. Theory*, IT-23(3):337–343, Mar. 1977.

[14] 井口，笹尾，松浦，伊勢野．ハードウエアを用いた論理関数の評価．情報処理学会 *DA シンポジウム '99*, pages 41–46, July 1999.

[15] 植松友彦．情報源符号化の現状と展望．信学技報, IT98-54:27–35, Dec. 1998.

[16] 奥乃博．二分決定グラフ (BDD) による探索型組合せ最適化問題の解法での組合せ爆発抑制法．情報処理学会論文誌, 35(5):739–753, May 1994.

[17] 木村，松本，羽根田．二分決定グラフの並列処理アルゴリズムについて．情報処理学会 *DA シンポジウム '92*, pages 105–109, Aug. 1992.

[18] 甲村，児玉，山口．データ駆動計算機 EM-4 における共有二分決定グラフの並列処 理について．情報処理学会第 *44* 回全国大会 *3D-5*, pages 6–(43–44), Mar. 1992.

[19] 山本博資．ユニバーサルデータ圧縮アルゴリズム：原理と手法．情報処理, 35(7):600–608, July 1994.