FPGAを用いた
SATプリプロセッサの高速化

２０１６年3月

鈴木将之

FPGAを用いた
SATプリプロセッサの高速化

鈴木将之

システム情報工学研究科
筑波大学

２０１６年3月

目 次

第 1 章　序論 **5**

第 2 章　SAT 問題と SAT プリプロセッサ **8**

2.1 充足可能性 (SAT) 問題 **8**

2.2 SAT ソルバ **10**

2.3 SAT プリプロセッサ **11**

第 3 章　FPGA を用いた SAT プリプロセッサ **13**

3.1 FPGA **13**

3.2 SAT ソルバ **16**

3.3 SAT プリプロセッサ **17**

3.3.1 本システム概要 17

3.3.2 SAT プリプロセッサにおけるメモリアクセス 18

第 4 章　プリプロセッサ SatELite の高速化 **21**

4.1 SatELite **21**

4.1.1 節の削除 23

4.1.2 リテラルの削除 23

4.1.3 変数の削除 23

4.1.3.1 単位伝播 24

4.1.3.2 Resolution 24

4.1.3.3 Variable Elimination by Substitution 25

4.1.3.4 Hyper-unary-resolution 26

4.2 FPGA への実装 **26**

4.2.1 データ構造 27

4.2.2 Clause Buffer 29
4.2.3 処理手順 29
4.2.3.1 節の包含関係の判定 30
4.2.3.2 Resolution と Hyper-unary-resolution 31
4.2.4 本システムにおける削減率 34

4.3 性能評価 **36**

第 5 章　プリプロセッサ Unhiding の高速化 **44**

5.1 Unhiding **44**
5.1.1 スタンピング 46
5.1.1.1 Failed literal elimination 49
5.1.1.2 Transitive reduction 50
5.1.1.3 Equivalent literal substitution 50
5.1.2 単純化 51
5.1.2.1 Unhiding Tautology Elimination 51
5.1.2.2 Unhiding Literal Elimination 52

5.2 FPGA への実装 **52**
5.2.1 データ構造 52
5.2.2 処理手順 53
5.2.3 Stamping unit 55
5.2.4 Simplification unit 59

5.3 性能評価 **61**

第 6 章　まとめと今後の展望 **66**

6.1 まとめ **66**

6.2 今後の課題 **67**

図 目 次

1 Xilinx 製 FPGA 概要 13
2 FPGA 開発フロー 14
3 FPGA を用いた SAT プリプロセッサ概要 17
4 DRAM からの読み込み手順 19
5 システム概要 27
6 Occurrence table，occurrence list と clause table 28
7 Inclusion detection unit 32
8 Resolution unit 35
9 DRAM へのアクセスシーケンス 37
10 Inclusion detection unit の並列度と計算時間 39
11 Resolution unit の並列度と計算時間 40
12 メモリアクセス遅延と計算時間の変化 41
13 Binary Implication Graph 46
14 タイムスタンプ例 49
15 Failed literal の例 49
16 Transitive edge の例 50
17 ELS の例 50
18 システム概要 53
19 データ構造 54
20 Stamping unit 概要 55
21 Simplification unit 概要 59
22 Simplification unit における並列性と計算時間 63
23 メモリアクセス遅延と計算時間 65

表 目 次

1 DDR3 転送速度 18
2 SatELite の適用結果 22
3 L に対する削減率の変化 31
4 R_t に対する削減率の変化 33
5 $|C_d|$ に対する削減率の変化 34
6 本システムの削減率 36
7 計算時間 (秒) と高速化率 38
8 各フェーズにおける連続メモリアクセスの比率 40
9 リテラル l を含む平均節数とそれらの節に含まれるリテラルの総数 43
10 ベンチマークと Unhiding 適用後のベンチマークの変数の数，節数，リテラル数と計算時間 45
11 Stack と S の深さ 56
12 L_s に対するリテラルの削減率 60
13 計算時間と高速化率 62
14 各フェーズにおける高速化率 62
15 各フェーズにおけるメモリアクセスの比率 64

第1章
序論

充足可能性問題 (SAT 問題) は論理式を真にする変数への真偽の割り当てを求める問題である．SAT 問題は形式的検証問題 [11] やスケジューリング [12] などに幅広く応用されるが，その実応用問題は数百万変数，数千万節を含む．これまでに SAT 問題を高速に解くためにさまざまな手法が提案されてきた ([16] [17] [18] [19]．ここでは一部のみを挙げる).

大規模な問題をより高速に解くための一般的な手法として，1. 処理手順の工夫による計算量の削減，2. アルゴリズムの並列化，3. 外部メモリアクセスの最適化などがある．1. の処理手順の工夫による計算量の削減は，各処理に対しより効率の良いアルゴリズムを採用することや，与えられたハードウェア資源のもと途中結果の一時保持や再利用などを最大限行うことなどである．2. のアルゴリズムの並列化はスーパーコンピュータやクラスタなどのマルチプロセッサを使用する方法とシングルプロセッサ (マイクロプロセッサ，GPU，FPGA など) のマルチコアを使用する方法とに分けられる．各プロセッサに対して外部メモリが付与されており，マルチプロセッサにおいては複数のプロセッサがそれぞれの外部メモリに独自にアクセスすることができるのに対し，シングルプロセッサ内のマルチコアでは一つのメモリが共有される．マルチプロセッサにおける高速化においても，シングルプロセッサの性能を最大限使用するための並列化手法は有効である．3. のメモリアクセスの最適化は，メモリアクセスがボトルネックとなりやすい大規模な問題における高速化において重要である．なぜならば，データ転送幅がプロセッサの処理性能に比べ低く，またランダムアクセス時にメモリアクセス遅延による待機状態が発生するためである．

本論文では大規模な SAT 問題の実応用問題に対する，FPGA を用いた SAT プリプロセッサの高速化手法について述べる．SAT ソルバが実応用問題を高速に解きやすくなるよう問題規模を削減するなど，問題を改変するために SAT プリプロセッサ [23] [39] [38] が用いられる．最新の SAT ソルバ [17] [18] [19] において，SAT プリプロセッサが SAT ソルバに組み込まれており，その重要度は高まりつつある．SAT プリプロセッサを高速化することにより，全体の計算時間を減らすことができ，またより多くのアルゴリズムを適用することができる．SAT プリプロセッサで用い

られる多くのアルゴリズムは高い並列性を持つため，専用ハードウェアを構築しそれらのアルゴリズムを並列に適用することで大きく高速化することができる．しかし，それらのアルゴリズムは複雑なメモリアクセスを必要とし，単純な実装やGPUでは十分な高速化を行うことができない．大規模な実応用問題に対するSATプリプロセッサの高速化のためには，アルゴリズムの並列化と共にメモリアクセスの最適化が必須である．本論文ではアルゴリズムの並列化の他いくつかのメモリアクセス最適化手法を適用し，SATプリプロセッサの高速化を目指すとともに，ハードウェアを用いたSATプリプロセッサの高速化におけるメモリアクセス最適化の有効性を評価することを目的とする．

本論文では二つのSATプリプロセッサSatELiteとUnhidingのFPGAを用いた高速化手法について述べる．

SatELite [38] は現在広く使われているSATプリプロセッサである．節の包含関係による節とリテラルの削除と，節のマージによる変数の削除を行う．このアルゴリズムは高い並列性を持ち，FPGAを用いて高速化を行いやすい．またデレファレンスを除き多くのメモリアクセスを連続で行うことができる．このため，節を格納するキャッシュメモリやレジスタを用意することで，メモリアクセス遅延の発生を減らすことができ，非常に高い高速化率を達成することが期待できる．

Unhiding [39] は比較的最近開発されたSATプリプロセッサであり，リテラルの依存関係を元に節とリテラルを削除する．このアルゴリズムはリテラルの依存関係の検出と単純化に分けることができる．リテラルの依存関係の検出においては，リテラルの依存関係を木構造として扱うため，その木構造を構築するために多くのメモリアクセスがランダムとなる．この処理では次に読み込むべきリテラルを事前に決定することができないため，読み込み順序を最適化することができない．またFPGAの内部メモリは小さく，探索を並列に行うこともできない．そのため，リテラルの依存関係の検出の高速化はそれほど期待することができないと考えられる．単純化においても，節に含まれるリテラルのタイムスタンプを読み込む必要があり，ランダムアクセスを必要とする．しかし，この処理では次に読み込むべきタイムスタンプを事前に決定することができるため，読み込み順序を最適化することができる．読み込み順序の最適化としてバンクインタリーブを行い，アクセス遅延による待機時間の発生を減らすことにより十分な高速化を行うことが期待できる．

この二つのプリプロセッサの高速化を通して，これまでFPGAによる

高速化がほとんど行われていない，外部メモリへの参照が頻繁に行われるような大規模な問題の高速化に対しても，問題の特性に応じた並列化，メモリアクセスの最適化を行うことで，数倍から数十倍の高速化が実現可能であることを示す．

本論文の構成はまず第2章にてSAT問題とSAT問題を解くためのアルゴリズムを紹介し，SATプリプロセッサについて解説する．第3章ではハードウェアSATソルバの先行研究と本SATプリプロセッサシステムの概要を述べる．第4章ではSatELiteの高速化手法について，第5章ではUnhidingの高速化手法について述べる．最後に第6章にて本論文のまとめと今後の展望を述べる．

第2章

SAT問題とSATプリプロセッサ

本章ではまずSAT問題について解説し，次にSAT問題を解くために用いられるSATソルバと，SATソルバにおいて用いられるSATプリプロセッサについて述べる．これまでに多くのSATソルバとそのアルゴリズムが開発されてきた．近年これに加え，SAT問題をより高速に解くためにSATプリプロセッサの研究が進んでいる．

2.1 充足可能性(SAT)問題

充足可能性問題 (*satisfiability problem*，*SAT* **問題**) は組合せ最適化問題としてよく知られる問題であり，論理式 $F(x_1, x_2, ..., x_n)$ を真とするような**変数** (*variable*) への真偽の割り当てを求める問題である．一般に F は**乗法標準形** (*conjunctive normal form*，*CNF*) で表される．CNFは**節** (*clause*) 同士の連言であり，節は**リテラル** (*literal*) 同士の選言である．リテラルは変数の肯定か否定である．以下のCNFで表現された論理式 (C_i は節であり C_1，C_2 と C_4 は2つのリテラルを含み，C_3 は3つのリテラルを含む) において，$\{x_1, x_2, x_3\} = \{1, 1, 0\}$ とすると F は真となる．F を真とする変数への割り当てが存在する時 F は**充足可能**であるといい，変数への割り当てが存在しない時 F は**充足不能**であるという．

$$
\begin{aligned}
F(x_1, x_2, x_3) &= C_1(x_1, x_2, x_3) \wedge C_2(x_1, x_2, x_3) \wedge \\
&\qquad C_3(x_1, x_2, x_3) \wedge C_4(x_1, x_2, x_3) \\
C_1(x_1, x_2, x_3) &= \overline{x}_1 \vee x_2 \\
C_2(x_1, x_2, x_3) &= \overline{x}_2 \vee \overline{x}_3 \\
C_3(x_1, x_2, x_3) &= x_1 \vee \overline{x}_2 \vee x_3 \\
C_4(x_1, x_2, x_3) &= x_1 \vee \overline{x}_3
\end{aligned}
$$

節に含まれるリテラル数を**節の長さ**と呼ぶ．長さ L の節を L-**リテラル節**と呼び，特に長さ1の節を**単位節** (*unit clause*) と呼ぶ．

SAT問題は形式的検証 [11] [13]，プラニング問題やスケジューリングの最適化 [12] などに応用することができる．

形式的検証は形式手法を使用して，ハードウェアやソフトウェアが仕様を満たすかどうかを検証することでその信頼性を高める手法である．SAT 問題を利用した代表的な形式的検証手法として，有界モデル検査 (Bounded Model Checking) [13] がある．有界モデル検証はそのシステムの各ステップを論理式として表し，その充足可能性を判定することである性質 (例えば [11] の 2 ビットカウンタ例における，カウンタが 3 の状態になるかどうか) を満たすことができるか検証する手法である．

その状態を $x_1, x_2, ..., x_n$ のビット列で表すことができるシステムにおいて，初期状態を $I(s)$，状態遷移を $T(s, s')$ とすると，k ステップ目における状態は $I(s_0) \wedge \bigwedge_{i=0}^{k-1} T(s_i, s_{i+1})$ と表すことができる．それに性質 $P(s_i)$ の否定命題 $\neg P(s_i)$ を加えた論理式 $\varphi_k = I(s_0) \wedge \bigwedge_{i=0}^{k-1} T(s_i, s_{i+1}) \wedge \bigvee_{i=0}^{k} \neg P(s_i)$ の充足可能性を SAT ソルバなどを用いて判定する．φ_k が充足不能であるならば，システムは k ステップ目においてその性質を満たすことができることを意味する．φ_k が充足可能ならばその性質の反例があることが分かる．これを任意のステップまで繰り返す．

プランニング問題は初期状態と目標状態が与えられた時に，目標状態に到達する実行可能なプランが存在するか，実行可能なプランが存在するならばステップ数またはアクション数が少ないプランを求める問題である [12] [14] [15]．プランニング問題においては，以下の手順で処理が行われる．

1. ステップ数 l を $l = 1$ とする．
2. l までに目標状態に到達する可能性のあるアクションを列挙する．目標状態に到達するアクションが存在しない場合，l を 1 増やす．
3. プランニング問題を SAT 問題 P に変換する [15]．
4. SAT ソルバを用いて P を解く．充足可能であるならば実行可能なプランを抽出する．充足不可能であるならば 2. に戻る．

スケジューリング問題は最小コストを求めるプランニング問題に近い．スケジューリング問題の一つであるショップスケジューリング問題は，処理時間が割り当てられているオペレーションの集合とオペレーションを一つ実行できるマシンの集合に対し，全てのオペレーションが完了するまでの総処理時間が最小であるスケジューリングを求める問題である．

総処理ステップ数を l と仮定し，ショップスケジューリング問題を SAT 問題 P_l へ変換しそれを解く．P_l が充足可能であるならば処理時間 l のス

ケジュールが存在する．充足不能であるならば，l より大きな処理ステップ数を必要とする．すなわち P_k が充足可能かつ P_{k-1} が充足不能であるならば，k で完了するスケジューリングが最小の処理時間を与える．

SAT 問題は NP 完全問題であり [7]，その計算量のオーダーは $O(n^2)$ であると思われている．しかし，実応用問題より生成された SAT 問題はより短い時間で解くことができるものも多い．これは実応用問題から生成された SAT 問題は規則的な構造を持ち，SAT プリプロセッサや学習などにより探索空間を限定することができるためである．本論文においては実応用問題より生成された SAT 問題として SAT competition [9] の application 部門よりいくつかの問題を抜粋し用いた (SAT competition は SAT 学会 [8] が開く SAT ソルバの競技会であり，application 部門は実問題から生成されたベンチマークに対する SAT ソルバの計算時間を競う部門である)．

2.2 SATソルバ

SAT 問題を解くためのアプリケーションを *SAT* **ソルバ**と呼ぶ [9] [16] [17] [18] [19]．SAT ソルバは**系統的** *SAT* **ソルバ**と**確率的** *SAT* **ソルバ**に分けることができる．

系統的 SAT ソルバは解空間を全探索するアルゴリズムに基づくものである．確率的 SAT ソルバに比べ一般により多くの計算時間を必要とする．系統的 SAT ソルバの代表的なアルゴリズムである DPLL アルゴリズム [10] の概要を以下に示す．

1. 真偽を割り当てる変数 x をひとつ選ぶ．

2. x に真偽を割り当てる．

3. 2. の変数割り当てにより生成された単位節が真になるように，その単位節に含まれる変数に真偽を割り当てる．例えば，論理式 $(\overline{x} \vee y) \wedge (\overline{x} \vee \overline{y} \vee z)$ に対し，x を真に割り当てた場合，$(\overline{x} \vee y)$ より単位節 (y) が生成される．(y) を真にするために y に真を割り当てる．この変数への真偽割り当てにより $(\overline{x} \vee \overline{y} \vee z)$ からさらに単位節 (z) が生成され，変数 z へ真を割り当てることができる．このような連鎖を**単位伝播** (*boolean constraint propagation*，*BCP*) と呼ぶ．

4. 3. により空節が生成された (充足不能である) ならば，バックトラックする．

5. 論理式を真にする全ての変数への割り当てを見つけるか，全ての真偽の割り当ての組み合わせを試すまで 1. から 4. を繰り返す．

確率的 SAT ソルバは真偽の割り当てをランダムに変更し解を探索するアルゴリズムである．確率的 SAT ソルバは系統的 SAT ソルバに比べ一般に短い計算時間で解に到達することが期待できるが，解に到達しない場合その問題が充足不能であるか判別することができない．確率的 SAT ソルバの代表的なアルゴリズムである WalkSAT [10] の概要を以下に示す．

1. 全ての変数に対しランダムに真偽を割り当てる．

2. 充足していない節をランダムに選択する．その中から一つ変数を選択し真偽を反転させる (真が割り当てられているならば偽，偽なら真を割り当てる)．これにより，その節を充足することができるが，他の充足されていた節が非充足になることもある．

3. 2. を全ての節が真になるまで繰り返す．

最近の SAT ソルバではこれらのアルゴリズムに加え学習，リスタート，SAT プリプロセッサなどが適用される．学習は矛盾の原因となる割り当ての否定を学習節として論理式に追加する手法である [16] [18]．学習節を追加することにより，同じ矛盾の発生を抑えることができる．リスタートは定期的に学習節を追加したまま探索を最初からやり直すことである [17]．リスタートにより変数の選択順序などの影響を緩和することができる．SAT プリプロセッサについては第 2.3 章にて述べる．

2.3 SAT プリプロセッサ

SAT **プリプロセッサ** (*SAT preprocessor*) は SAT ソルバが問題を解く前に以下のことを行い，探索空間を削減するアプリケーションである [20] [21] [22] [23] [38] [39]．

1. 変数や節を削除することにより問題規模を削減する

2. 制約となる節とリテラルを追加することにより探索空間を削減する

SAT 問題の実応用問題は数百万変数，数千万節を含むため，SAT プリプロセッサによる探索空間の削減が近年重要視されつつある [39].

上記 1. の代表的なアルゴリズムは NiVER [20]，SatELite と Unhiding である．NiVER は SatELite の基となった SAT プリプロセッサであり，resolution(第 4.1.3.2 章) による節のマージにより，変数を削除し問題規模を削減する．SatELite については第 4 章，Unhiding については第 5 章で詳しく述べる．

上記 2. の代表的なアルゴリズムは Hypre [21] である．Hypre は論理式 F に節 $(l_0 \vee l_1 \vee ... \vee l_n)$ と $n-1$ 個の 2-リテラル節 $(\overline{l_i} \vee l)$ が含まれている場合，節 $(l_n \vee l)$ を F に追加する処理 hyper-resolution (hyper-binary-resolution) を行う ($F = (l_0 \vee l_1 \vee l_2) \wedge (\overline{l_0} \vee l) \wedge (\overline{l_1} \vee l)$ に対し $(l_2 \vee l)$ が追加される)．このような 2-リテラル節を追加することにより，論理式中の節数とリテラル数が増える代わりに，より多くの単位伝播を適用することが可能となり，計算時間を減らすことができることがある．ただし，このタイプのアルゴリズムは，管理すべき節が増えるため逆に問題を解く時間が増えることがある．

近年では SAT プリプロセッサを SAT ソルバの処理中に適用する研究もある (このような手法は inprocessing と呼ばれる) [24]．本来 DPLL アルゴリズムでは BCP のほか純リテラル規則 ($\bar{l}$ が論理式に含まれていない時，l は真であると決定することができる) や分割規則 (論理式 $F = (l \vee A) \wedge (\bar{l} \vee B) \wedge P$ に対し $F' = A \wedge P$ と $F'' = B \wedge P$ の二つの論理式を生成する) を適用する [10]．しかし多くの系統的 SAT ソルバでは処理中に BCP のみを行う．これは他のアルゴリズムが BCP ほど効果的でないことと，BCP より多くの計算時間を要するためである．BCP に加え効率よく設計された SAT プリプロセッサのアルゴリズムを適用することにより，より計算時間を減らすことが期待できる．

SAT ソルバ lingeling [40] において，全計算時間の内そのプリプロセッサ SatELite の変数削除アルゴリズムの適用時間が 16%，Unhiding の適用時間が 7%ほどであり [39]，決して短いとは言えない．SAT プリプロセッサを高速化することにより，SAT 問題を解くための全計算時間を短くすること，より多くの SAT プリプロセッサのアルゴリズムを適用することや inprocessing を行うことなどが可能となる．

第3章
FPGAを用いたSATプリプロセッサ

本章ではまず先行研究である FPGA を用いた SAT ソルバを紹介する．次に FPGA を用いた SAT プリプロセッサの概要について述べる．

3.1 FPGA

Field Programmable Gate Array(*FPGA*) はその場で書き換え可能な LSI である．FPGA は起動時に外部 RAM から回路情報を読み込み，回路を構築することができる．外部 RAM 中の回路情報を変更することのみで，FPGA 上に構築される回路を変更することができる．

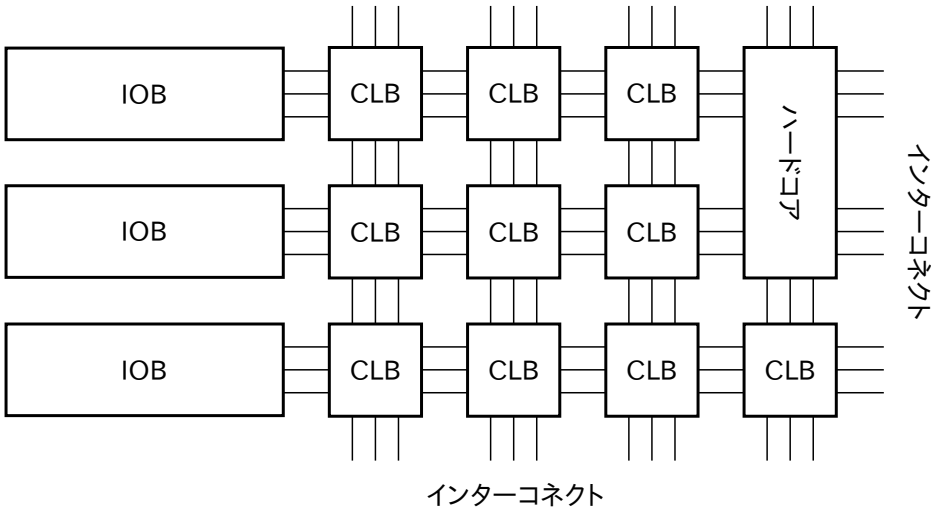

図 1: Xilinx 製 FPGA 概要

Xilinx [1] 製 FPGA の構造の概略を図 1 に示す．Xilinx 製 FPGA は以下の単位で構成される．

コンフィギャブルロジックブロック (Configurable logic block，CLB)

ルックアップテーブル，レジスタや乗算器などから構成される論

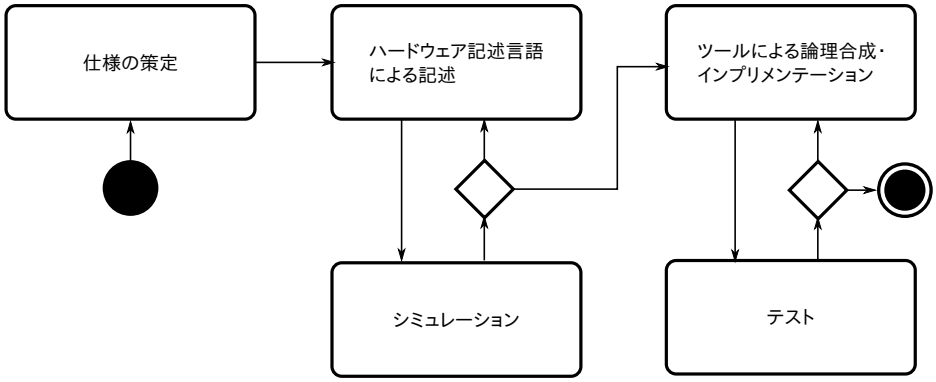

図 2: FPGA 開発フロー

理単位．**ルックアップテーブル** (*lookup table*，*LUT*) は 4 または 6 入力 1 出力の RAM である．LUT の値を書き換えることにより論理素子などを表現することができる．

インターコネクト (Interconnect)
CLB 間や CLB と I/O 間を結ぶ接続単位．インターコネクトの設定により，CLB で表現された論理単位同士の接続を変更することができる．

SelectIO (IOB)
外部 I/O 単位．

ハードコア (Hard core)
外部インターフェース (JTAG，PCI Express や DDR3 など) やメモリなどの機能単位．そのうち内部 SRAM メモリのハードコアを**ブロック** *RAM*(*block RAM*) と呼ぶ．

本論文においては LUT とブロック RAM の使用個数を回路規模とする．

FPGA の代表的な開発フローを図 2 に示す．FPGA の回路は**ハードウェア記述言語**を用いて記述することができる．ハードウェア記述言語を用いて記述された回路は，各ベンダーのツールなどを用いて FPGA の回路データに変換される．ハードウェア記述言語による記述や配線データなどが要求と仕様を満たすかどうか検証するためにシミュレーションが行われる．本論文ではハードウェア記述言語 Verilog HDL [5] を用いて回路を記述し，Xilinx Vivado 2014.2 [1] を用いて回路を生成した．

FPGA は主に以下の用途に使われる．

ASIC のプロトタイピング

FPGA 上に回路を構築しそれをテストすることで，ASIC の開発コストを抑えることができる [2].

IP コアの外部インターフェース

インターフェースに FPGA を用いることにより，いかなる外部回路とも IP コアを接続することができる．

厳しい環境下における修復可能な回路

宇宙線などにより回路が損傷した場合，再度コンフィギュレーションを行うことで回路を修復することができる [3] [4].

専用回路による高速計算

FPGA 上に専用ハードウェアを構築することにより，アルゴリズムを並列化，パイプライン化し，高速にアルゴリズムを適用することができる．

本論文では専用回路による高速計算を実現するために FPGA を用いる．FPGA を用いて SAT プリプロセッサのアルゴリズムの並列化，パイプライン化を行う．他のプロセッサに対し，以下の点において FPGA は SAT プリプロセッサの高速化に向いたプロセッサであるといえる．

1. マイクロプロセッサが実現可能な並列度より，SAT プリプロセッサのアルゴリズムは高い並列度を持つ．FPGA であればその回路規模が許す範囲で並列化を行うことができる．

2. GPU に比べメモリアクセスに関する制約が少ない．GPU のデータ転送幅を最大限使用するためには連続メモリアクセスを行うことと，各コアに対応してメモリ上に連続してデータが配置されていること，各データが特定のビット長にアライメントされていることが必要である (コアレッシング) [6]．しかし，SatELite と Unhiding のデータ構造 (第 4.2.1 章と第 5.2.1 章) はデレファレンスを必要とする．そのため，必ずしも連続的にメモリアクセスを行うことはできず，また事前にコアレッシングを行うことができるようデータを配置することもできない．すなわち，この二つの SAT プリプロセッサのデータ構造は GPU には不向きである．

3. 処理後ハードウェア SAT ソルバへ回路を書き換えることができる．

3.2 SATソルバ

FPGAを用いたSATソルバはこれまで多く研究されてきた ([26] など). 本章ではそのいくつかの例について紹介する.

[27] [28] はDPLLアルゴリズムを回路としてFPGA上に構築し，その回路を動作させることにより問題を解くSATソルバである．この手法では条件分岐全てを回路として展開することにより，DPLLにおける解空間の探索を行う．この手法の欠点は，問題によってはFPGAのリソースを効率よく使用することができないこと，解くことができる問題規模が小さい (400変数程度) ことと，与えられた問題ごとに回路を生成する必要があり，その生成に多大な時間を必要とすることである．

[29] [31] はホストコンピュータとFPGAボードを用いたソフトウェア/ハードウェア混在型SATソルバである．FPGA上でDPLLにおけるBCPと変数決定を行い，それ以外 (節の管理，バックトラッキングや学習など) をホストコンピュータのソフトウェア上で行う．[29] はこのようなシステムを構築した最初の例である．[31] は単純なBCPに加えenhanced BCP ($(\overline{a} \vee \overline{b} \vee \overline{c}) \wedge (\overline{a} \vee b \vee c) \wedge (a \vee \overline{b} \vee c) \wedge (a \vee b \vee \overline{c}) \equiv (b = c)$) を行い，BCPの効果を高めている．この手法の長所はそれぞれのプロセッサが得意とするアルゴリズムを実行することができる (マイクロプロセッサはバックトラッキングや学習などを得意とし，FPGAはBCPを得意とする) ことと，既に開発されている信頼性のあるソフトウェアを活用することができることである．欠点はFPGAボードとホストコンピュータとの通信 (この場合はPCI Expressバスの帯域) がボトルネックとなりやすいことである．

[30] はマイクロプロセッサを搭載したFPGAボードを用いて，DPLLにおけるBCPフェーズと学習をFPGAで行い，それ以外の部分 (変数選択，バックトラッキング) などをマイクロプロセッサで行うSATソルバである．

[32] [33] [34] [35] [36] [37] はFPGAボードとFPGAボード上のDRAMを用いてWalkSATの高速化を行うSATソルバである．FPGAを使うことにより，WalkSATにおける節の充足可能性を並列に判定することができ，さらにパイプラインによるマルチスレッド処理により複数の試行を並列に行うことができる．

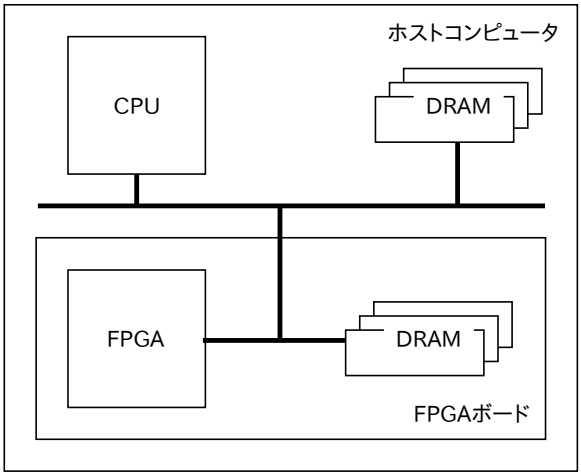

図 3: FPGA を用いた SAT プリプロセッサ概要

3.3 SAT プリプロセッサ

3.3.1 本システム概要

SAT ソルバを高速化するために多くのハードウェアシステムが構築されてきた (第 3.2 章). しかし，我々の知る限りハードウェアを用いた SAT プリプロセッサは開発されていない．これはおそらく，従来のハードウェアソルバでは解くことのできる問題規模が小さく，小規模な問題では SAT プリプロセッサを用いても大きな高速化を期待することができないためである．

本論文では SatELite と Unhiding の二つの SAT プリプロセッサの高速化手法について述べる．SatELite は現在最も広く使われる SAT プリプロセッサであるため，その高速化は非常に有意義である．Unhiding は比較的新しい SAT プリプロセッサであり，SatELite と組み合わせることでより探索空間を削減することができる．

図 3 に本システムの概要を示す．本システムではホストコンピュータと FPGA ボードを用いる．ホストコンピュータと FPGA ボードは PCI Express などのバスで接続されており，そのバスを介して FPGA ボード上の DRAM へデータを送受信することができる．

本システムの処理手順は以下のとおりである．

1. FPGA 上に SAT プリプロセッサの回路を構築する．
2. ホストコンピュータ上で問題ごとのデータ構造を生成する．

3. FPGA ボード上の DRAM へデータ構造を転送する．

4. FPGA 上の SAT プリプロセッサを用いて問題を改変する．改変された問題とデータ構造は FPGA ボード上の DRAM に格納される．

問題を改変した後，ホストコンピュータ上へ転送し，ホストコンピュータ上の SAT ソルバを用いて解を求めるか， FPGA を SAT ソルバに書き換え FPGA 上の SAT ソルバを用いて解を求める．

3.3.2 SAT プリプロセッサにおけるメモリアクセス

本論文において，FPGA 上に実装される各アルゴリズムは十分に並列化されている．それに伴い，本システムにおいてボトルネックとなるのが DRAM へのアクセスである．本論文の多くのデータ構造は FPGA ボード上の DRAM へ格納されており，このデータ構造を頻繁に参照するため，メモリアクセスが多く発生する．そのため高速化においてデータ転送幅の制限とメモリアクセス遅延の発生が問題となる．

表 1: DDR3 転送速度

チップ規格	パスクロック (MHz)(データ転送速度 (MHz))	最大転送速度 (GB/s)
DDR3-1333	667(1333)	10.7
DDR3-1600	800(1600)	12.8
DDR3-1866	933(1866)	14.9
DDR3-2133	1066(2133)	17.1

DDR3(DRAM のインターフェース) [25] の転送速度 (転送幅) を表 1 に示す．表 1 より DDR3 の最大転送速度は FPGA の処理能力と比べ低い (例えば SatELite の高速化手法における節の包含関係の判定 (第 4.2.3.1 章) において，本システムは最大 2458GB/s の処理能力を持つ)．すなわちこの転送速度を最大限活用しなければ，高い高速化率を達成することができない．転送速度を最大限活用するためには，連続したメモリアクセスを行うことや先読みにより常に帯域を使用することが必要である．

また，DRAM へのアクセスには遅延が発生する．DRAM からの読み込み手順を図 4 に示す．DRAM からデータを読み込むためには読み込み命令 (図 4 における READ) とアドレス (図 4 における Address0，Address1，

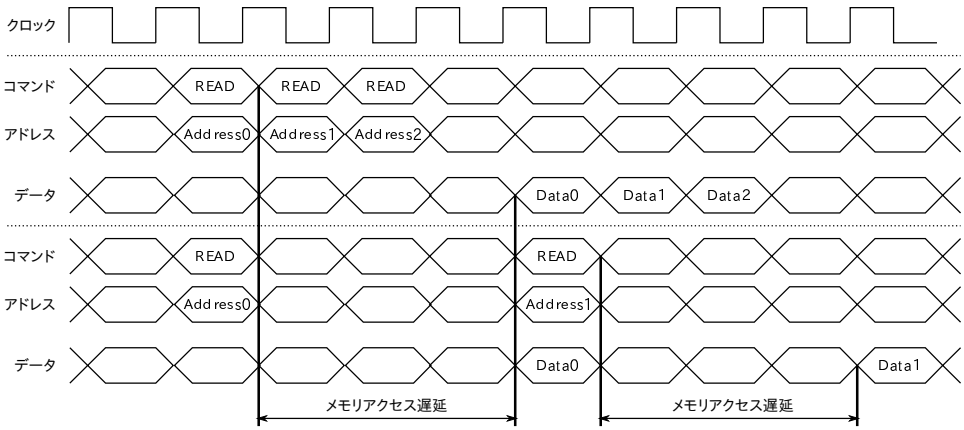

図 4: DRAM からの読み込み手順

Address2) を指定する．その数クロック後データ (図 4 における Data0, Data1，Data2) が送信される．この遅延は DRAM の構造による遅延と，DDR3 インターフェースハードコア (本システムは DDR3 インターフェスハードコアを使用することを想定している) のデータのバッファやオーダーの変更などによる遅延によるものである．本論文において，DRAM のアクセス遅延+DDR3 インターフェースハードコア由来の遅延を**メモリアクセス遅延**とする．メモリアクセス遅延は一回のアクセスごとに発生する．図 4(上) において，Address0 から Address2 へ連続アクセスを行った場合は 1 回のみメモリアクセス遅延が発生する．図 4(下) において，Address0 と Address1 へ非連続アクセスを行った場合はそれぞれメモリアクセス遅延が発生する．すなわち，メモリアクセス遅延を減らすためには，メモリアクセス自体を減らすか，連続したアクセスを行えばよい．

帯域を最大限活用しメモリアクセス遅延を隠蔽するために，一般に以下の手法が取られる．

メモリアクセスの削減 アルゴリズムを工夫することによりメモリアクセス自体を減らす．

キャッシング プロセッサの内部メモリ内にデータを格納し再利用することにより，メモリアクセスを減らす．

先読み アルゴリズムを適用中，次に必要とされるデータが事前にわかっている場合データを必要となる前に読み込む．これにより常に帯域を使用することができる．

本論文におけるシステムにおいて先読みを行うことは一般的に難しい．また FPGA の内部メモリは小さく，先読みを行ったとしても読み込んだデータを十分に格納することができない．

バーストリードの活用 なるべく連続メモリアクセスを行うようにし，メモリアクセス遅延を減らすことができる．

バンクインターリーブ DRAM の複数バンクから順番にデータを読み込むことにより，メモリの帯域を最大限に活用することができる．

本論文において以下のメモリアクセス最適化手法を適用した．

プリプロセッサ SatELite の高速化
SatELite のアルゴリズムは変数ごとに，その変数を含む節に対して適用される．変数を含む節をキャッシュし再利用することでメモリアクセスの回数を減らすことができる (キャッシング)．またある変数を含む全ての節を一度に読み込むため，多くの連続アクセスを行うことができる (バーストリードの活用)．

プリプロセッサ Unhiding の高速化
深さ優先探索において子ノードをキャッシュすることによりメモリアクセスの回数を減らすことができる (キャッシング)．また読み込んだデータをアソシエイティブキャッシュにキャッシュすることにより，データを再利用することができる (キャッシング)．さらに一部アルゴリズムでは，データのプリフェッチが可能であるため, 単純な実装ではランダムアクセスとなるメモリアクセスに対し，バンクインタリーブを行うことでメモリの帯域を最大限に活用することができる．

第4章
プリプロセッサSatELiteの高速化

本章ではプリプロセッサSatELiteの高速化について述べる [41] [42].

SatELiteは広く使われているSATプリプロセッサであり，高い並列性を持つためハードウェアを用いて高速化しやすい．本システムではアルゴリズムの並列化のほか，キャッシングの最適化などを行いメモリアクセス遅延の隠蔽を試みた．その結果，本システムは基のソフトウェアと比べ3.4から54倍の高速化を達成した．

4.1 SatELite

SatELite [38] はNiklas EènやArmin Biereらによって開発されたプリプロセッサである．SatELiteは節をマージすることにより変数を削除し，節の包含関係の判定により節とリテラルを削除することにより問題規模を削減する．SatELiteは現在最も広く使われるSATプリプロセッサである ([17] [18] [40]といった代表的なSATソルバにSatELite自体かSatELiteの一部のアルゴリズムが使われている).

表2にSAT competition 2013から抜粋したベンチマークに対しSatELiteを適用し，問題規模を削減した結果を示す．表2において#varは変数の数，#clsは節数，#litはリテラル数を示す．表2の実行環境はCPUがIntel Core i7-2600，メモリ容量が8GBである．表2より，問題規模の大きな問題に対して問題規模をより削減することができるが，大きな計算時間を必要とすることが分かる．

表 2: SatELite の適用結果

	適用前			適用後			
ベンチマーク	#var	#cls	#lit	#var	#cls	#lit	実行期間 (sec)
AProVE07-11	1004933	4426834	11901940	85059	518554	1661277	853.23
velev-vliw-uns-4.0-9	96177	1814189	5280501	82887	1775308	5435667	140.25
11pipe_q0_k	104244	3007883	8917203	61450	2916361	9452973	91.53
maxor128	200308	598619	1396775	64849	340725	1162948	69.17
8pipe_k	35065	1332773	3936683	29031	1232440	3859607	65.40
blocks-blocks-36-0.130-NOTKNOWN	530375	9511460	20734360	377415	6570270	14369954	63.06

SatELiteで実行されるアルゴリズムは以下の3つに分けることができる．

節の削除

subsumption

リテラルの削除

self-subsumption

変数の削除

単位伝播，Resolution, Variable elimination by substitution, Hyper-unary-resolution

4.1.1 節の削除

二つの節 $C_0 = (a \vee b), C_1 = (a \vee b \vee c)$ が論理式 F に含まれているとする．C_0 が真であるならば C_1 は c の値に関係なく真となるため，C_1 を F より削除することができる．このような包含関係を基に節を削除するアルゴリズムを *subsumption* と呼ぶ．

4.1.2 リテラルの削除

リテラルを削除するために *self-subsumption* と呼ばれるアルゴリズムを用いる．二つの節 $C_x = (x \vee a \vee b), C_{\overline{x}} = (\overline{x} \vee a)$ が F に含まれているとする．二つの節にはそれぞれリテラル x とその否定 $\overline{x}$ が含まれる．C_x に含まれる x を除いたリテラル (a, b) は，$C_{\overline{x}}$ に含まれる $\overline{x}$ を除いたリテラル (a) のサブセットとなっている．この場合，下記式1より C_x を $(a \vee b)$ で置換することができ，C_x に含まれていた x を削除することができる．

$$
\begin{aligned}
C_x \wedge C_{\overline{x}} &\equiv (x \vee a \vee b) \wedge (\overline{x} \vee a) \\
&\equiv (\overline{x} \wedge a) \vee a \vee (\overline{x} \wedge b) \vee (a \wedge b) \\
&\equiv (a \vee b) \wedge (\overline{x} \vee a) \\
&\equiv (a \vee b) \wedge C_{\overline{x}} \qquad (1)
\end{aligned}
$$

4.1.3 変数の削除

変数を削除するために以下の4つのアルゴリズムが用いられる．

1. 単位伝播
2. Resolution
3. Variable elimination by substitution
4. Hyper-unary-resolution

4.1.3.1 単位伝播

(l) のように一つのリテラル l のみを含む節を**単位節**と呼ぶ．単位節 (l) が F に含まれていた場合，F が真であるためには l が真である必要がある．そのため，l が x であるときは変数 x は真であり，l が $\overline{x}$ であるときは x は偽であると決定することができる．単位節 (l) が含まれていた場合，F から以下のようにして変数 l を削除することができる．

1. l を含む全ての節を削除する．
2. $\overline{l}$ を含む全ての節から $\overline{l}$ を削除する．

例えば論理式 $(\overline{a} \vee b) \wedge (a)$ において (a) は単位節であり，a は真となる．その結果新たな単位節 (b) が生成される．このように単位節を次々と生成し，変数を削除するアルゴリズムを**単位伝播**と呼ぶ．

4.1.3.2 Resolution

リテラル x とその否定 $\overline{x}$ を含む二つの節 $C_0 = (x \vee a_0 \vee ... \vee a_n), C_1 = (\overline{x} \vee b_0 \vee ... \vee b_n)$ から *resolvent* と呼ばれる新しい節 $(a_0 \vee ... \vee a_n \vee b_0 \vee ... \vee b_n)$ を生成するアルゴリズムを *resolution* と呼び，resolvent を $C_1 \otimes C_2$ と記す．Resolvent の生成において，リテラルの集合 $A = \{a_0, ..., a_n\}$ と $B = \{b_0, ..., b_n\}$ の真偽によって以下の 4 つのケースが考えられる．

1. $A = true, B = true$ の場合，resolvent は真であり，$C_0 \wedge C_1$ は x の真偽に関わらず真となる．
2. $A = true, B = false$ の場合，resolvent は真であり，$C_0 \wedge C_1$ を真とするための x の値は偽である．
3. $A = false, B = true$ の場合，resolvent は真であり，$C_0 \wedge C_1$ を真とするための x の値は真である．

4. $A = false, B = false$ の場合，resolvent と $C_0 \wedge C_1$ は共に偽である．

x と $\overline{x}$ がそれぞれ C_0 と C_1 にのみ含まれるとして，F 中の $C_0 \wedge C_1$ をその resolvent で置き換え，F を真とする変数の割り当てを求める．この時 $A = \{a_0, ..., a_n\}$ と $B = \{b_0, ..., b_n\}$ の真偽値から，上記 1. から 4. に従い F が x の真偽に関らず充足不能であるか (上記 4.)，F を真にするための x の真偽が分かる (上記 1. から 3.).

x と $\overline{x}$ を含む節がそれぞれ N_x，$N_{\overline{x}}$ 個 F に含まれているとすると，N_x，$N_{\overline{x}}$ 個の節全ての組み合わせに対して resolvent を生成する必要がある．すなわち $N_x \times N_{\overline{x}}$ 個の resolvent を生成しなければならない．一般に resolvent の数 $N_x \times N_{\overline{x}}$ は元の節の数 $N_x + N_{\overline{x}}$ より大きい．しかし，$(x \vee a \vee b)$，$(\overline{x} \vee \overline{a})$ といった節が含まれていた場合，その resolvent は $(a \vee b \vee \overline{a})$ となり，明らかに真である．そのため，この resolvent を F に追加する必要はない．

SatELite において resolution は以下の場合に適用される．

1. N_x と $N_{\overline{x}}$ がともに 10 より小さい (第 4.2.3.2 章で詳しく述べるが，本システムでは $N_x + N_{\overline{x}} \leq 64$ であるときに適用される).

2. 真である自明な resolvent を除いた resolvent の数が $N_x + N_{\overline{x}}$ 以下である．

4.1.3.3 Variable Elimination by Substitution

Variable elimination by substitution は特殊なケースにおける resolution であり，生成される resolvent の数をより抑えることができる． 論理回路から生成された論理式 F には，$(x \vee \overline{a} \vee \overline{b})$，$(\overline{x} \vee a)$，$(\overline{x} \vee b)$ のような節のセットが含まれていることが多い．この節のセットは AND ゲートまたは OR ゲートに対応するものである ($x = a \wedge b$ または $\overline{x} = a \vee b$)．この節の集合を x についての *definition* と呼ぶ．S を x と $\overline{x}$ を含む節の集合，G_x を x についての definition のうち x を含む節の集合，$G_{\overline{x}}$ を x についての definition のうち $\overline{x}$ を含む節の集合，R_x を definition 以外の節の内 x を含む節の集合，R_x を definition 以外の節の内 $\overline{x}$ を含む節の集合とする．

$$S = \underbrace{\overset{1}{(x \vee c)} \wedge \overset{2}{(x \vee d \vee e)}}_{R_x} \wedge \underbrace{\overset{3}{(x \vee \overline{a} \vee \overline{b})}}_{G_x} \wedge \underbrace{\overset{4}{(\overline{x} \vee a)} \wedge \overset{5}{(\overline{x} \vee b)}}_{G_{\overline{x}}} \wedge \wedge \underbrace{\overset{6}{(\overline{x} \vee \overline{f})}}_{R_{\overline{x}}}$$

この時 S の resolvent のセット S' は以下の式で与えられる．

$$
\begin{aligned}
S' &= S'' \cup G' \cup R' \\
S'' &= (R_x \otimes G_{\overline{x}}) \cup (R_{\overline{x}} \otimes G_x) \\
&= \overset{1\otimes 4}{(a \vee c)}, \overset{1\otimes 5}{(b \vee c)}, \overset{2\otimes 4}{(a \vee d \vee e)}, \overset{2\otimes 5}{(b \vee d \vee e)}, \overset{3\otimes 6}{(\overline{a} \vee \overline{b} \vee \overline{f})} \\
G' &= G_x \otimes G_{\overline{x}} = \overset{3\otimes 4}{(\overline{a} \vee \overline{b} \vee a)}, \overset{3\otimes 5}{(\overline{a} \vee \overline{b} \vee b)} \\
R' &= R_x \otimes R_{\overline{x}} = \overset{1\otimes 6}{(c \vee \overline{f})}, \overset{2\otimes 6}{(d \vee e \vee \overline{f})}
\end{aligned}
$$

ここで G' に含まれる全ての節は definition の定義より常に真となる．さらに R' に含まれる全ての節は S'' に含まれる節より導出することができる [38]．すなわち x についての resolvent としては S'' のみ用いればよい．このような特殊なケースの resolution を variable elimination by substitution と呼び，通常の resolution に比べ resolvent の数を抑えることができる．

4.1.3.4 Hyper-unary-resolution

$S = (\overline{a} \vee \overline{b}), (\overline{x} \vee a), (\overline{x} \vee b)$ といった節が F に含まれているとする．この時変数 x を真とすると S は偽となり，x を偽とすると S は真となる．そのため x の値を偽と決定することができる．このような節の集合を見つけ変数の値を決定する処理を *hyper-unary-resolution* と呼ぶ．

4.2 FPGA への実装

図 5 に回路の概要を示す．全てのデータは DRAM に格納されており，data read unit より読み込まれ各ユニットに供給される．読み込まれた節は同時に clause buffer にキャッシュされる．キャッシュされた節を再利用することで，DRAM へのアクセス回数を減らすことができる．

Unit propagation unit では単位節の検出を行う．Inclusion check unit では二つの節を比較しその包含関係を判定する．Definition detection unit では definition を検出しかつ hyper-unary-resolution を適用することができるか判定する．Resolution unit では resolvent を生成する．これらのユニットで変更が加えられた節は queue に格納される．Occurrence list update unit は DRAM に格納されているデータ構造の occurrence table と occurrence list の更新を行う．

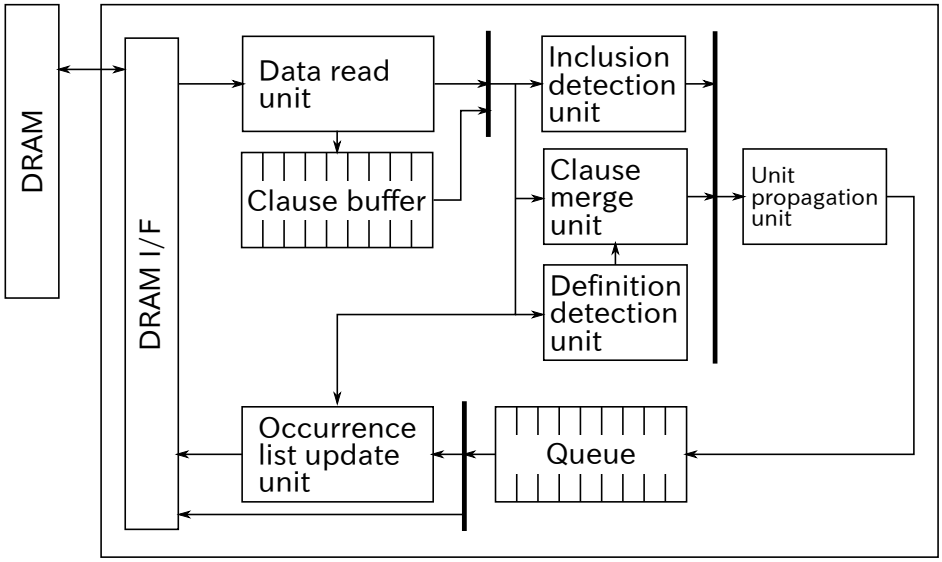

図 5: システム概要

4.2.1 データ構造

本システムは以下のデータ構造を用いる．

- 節に含まれるリテラルを格納するための clause table.
- 特定のリテラルを含む節に高速にアクセスするための occurrence table と occurrence list.
- 変更された変数に対してのみアルゴリズムを適用するための touched list．変数 v を含む節が変更された時 v が touched list に追加される．本システムでは 4 つの touched list を用いる．
- 単位伝播を適用するリテラルを格納するためのリテラルのスタック *single*．初期状態は空である．

図 6 に節を格納するためのデータ構造 (occurrence table， occurrence list，clause table) を示す． Occurrence table は変数の番号により参照される．Occurrence table の各エントリはリテラル x についての occurrence list の先頭アドレス，終端アドレスを格納する．x についての occurrence list は x を含む全て節へのアドレスを保持する．Clause table は各節の本体 (各節に含まれるリテラル) を格納する．

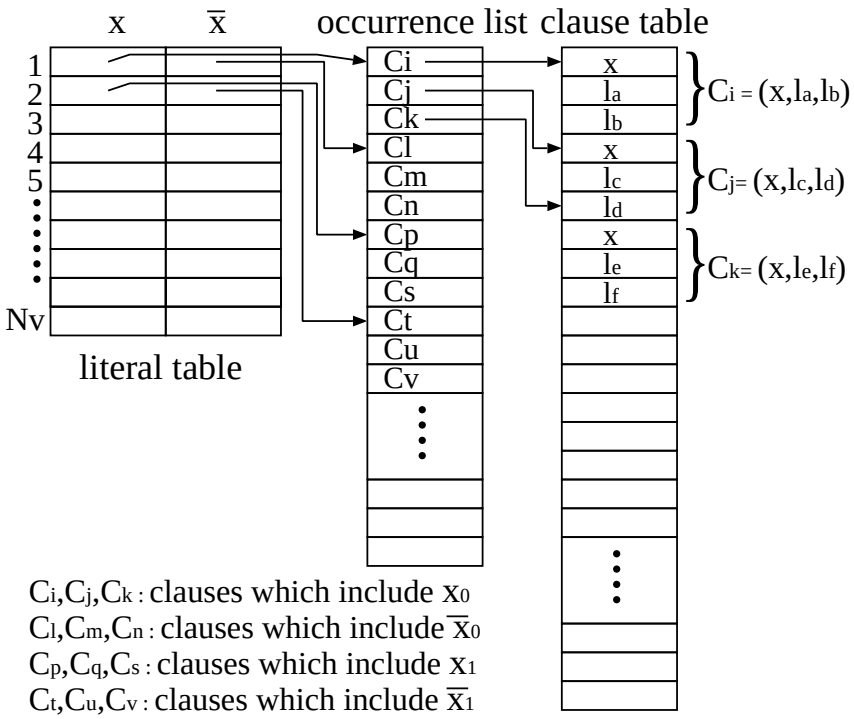

図 6: Occurrence table，occurrence list と clause table

以下 S_x をリテラル x を含む全ての節の集合とする．外部メモリから S_x を読み込む手順は以下の通りである．

1. Occurrence table から x についての occurrence list の先頭アドレスと終端アドレスを読み込む．

2. x についての occurrence list を読み，各節の先頭アドレスと終端アドレスを取得する．

3. Clause table から各節を読み込む．

4. 読み込まれた節は，clause buffer にキャッシュ可能な場合，キャッシュされ再利用される．

4.2.2 Clause Buffer

S_x に含まれる節は，DRAM より読みこまれると同時に clause buffer にキャッシュされる．キャッシング可能な節の数は回路構成と FPGA の規模に依存するが，現在の実装では 440 個の block RAM を clause buffer に用いており，およそ 225K 個の節をキャッシングすることが可能である (節に含まれるリテラルの数によってキャッシング可能な節の数は異なるが，平均 3 個として計算した)．S_x の節数が多く clause buffer にキャッシュすることができない場合は clause buffer を使用しない (今回評価に用いたベンチマークでは，全リテラルの 5% に対する S_x を clause buffer に格納することができなかった)．これは回路の設計を簡略化するための処置であり，一部の節のみを clause buffer にキャッシュすることで，さらに大規模な問題に対してもある程度の高速化を行うことができるが，それは今後の課題である．Clause buffer のサイズは，今回使用した FPGA に実装可能な最大サイズとした．回路から生成された実応用問題の特性から，現在 clause buffer にキャッシュすることができない S_x の節数は非常に多く，ある程度 clause buffer を大きくしてもそれをキャッシュすることはできない．このため，ほとんどの問題においては，これより大きな clause buffer (より大きな FPGA) を用いても，大きな速度改善を期待することはできない．

Hyper-unary-resolution 以外の処理においては，処理対象となるリテラル l (またはその否定 $\bar{l}$) を含む節のみが必要とされる．しかし，hyper-unary-resolution においては，あるリテラル l に関する処理を実行中に，その処理状況に応じて，異なるリテラル m を含む全ての節を読み込む必要が生じる．この時，clause buffer の容量および実装上の複雑さの問題から，l と m の両方を含む節をキャッシングすることができず，l を含む節は，m を含む節によって上書きされてしまう．このため，m に関する処理が終了したのち，l を含む節を再度メモリから読み込む必要がある．

4.2.3 処理手順

本システムにおける処理手順はソフトウェアとほぼ同様である [38] [41]．本システムにおける処理手順を以下に示す．これらの処理は touched list を用いて，変更が加えられた変数のみに対して行われる．

1. 全てのリテラルを touched list に入れる．

2. Touched list からリテラルをひとつ取り出す．そのリテラルに対し節の包含関係を判定する (subsumption と self-subsumption)．変更が行われたリテラルを touched list に入れる．

3. Touched list から変数をひとつ取り出す．その変数に対し resolution, variable elimination by substitution ，hyper-unary-resolution を適用する．変更が行われたリテラルを touched list に入れる．

4. Touched list が空になるまで 2. と 3. を繰り返す．

またこれらの処理中に単位節が生成された場合，単位伝播を適用する．

本システムでは，各変数は順次 (逐次的に) 処理され，各変数に対してアルゴリズムが並列に適用される．各変数に対して各アルゴリズムを適用した結果，節が更新された場合には，それらの節に応じて各テーブルが更新され，その後，次の変数の処理が開始される．また，各変数の処理中に，clause buffer にキャッシングされている節の更新を行わなくても，ほぼ同一の結果を得ることができるため clause buffer 中の節の更新は行わない．

4.2.3.1 節の包含関係の判定

$C_0 = (a \vee b)$ と $C_1 = (a \vee b \vee c)$ といった節の組を検出した場合，subsumption を適用することができる (第 4.1.1 章)． $C_0 = (x \vee a \vee b)$ と $C_1 = (\overline{x} \vee a \vee b \vee c)$ といった節の組を検出した場合，self-subsumption を適用することができる (第 4.1.2 章)．Self-subsumption においてリテラル x と $\overline{x}$ を無視すると，どちらの処理も節の包含関係の判定であり，同じユニットを用いて処理することができる．節同士の比較を並列に行うことで，節の包含関係の検出を高速化することができる．さらに節同士の比較において，リテラル同士の比較を並列に行うことができる．

表 3 に subsumption と self-subsumption を，包含される節のリテラル数が L 個以下 ($L = 8, 16, 32$) の場合にのみ適用した時の削減率を示す．表 3 において，#var は変数の数，#cls は節数，#lit はリテラル数の割合を表す．また 1.00 はソフトウェアと同等の性能を示し 0.30 はソフトウェアに比べ 70%削減率が低下したことを示す (各アルゴリズムの適用順番の差異により，本システムの削減率はソフトウェアを上回ることも下回ることもある)．表 3 より L が 8 以上であれば削減率は変わらないことが分かる．本システムでは L を 16 とした．

表 3: L に対する削減率の変化

	L					
	8		16		32	
ベンチマーク	#var	#cls	#var	#cls	#var	#cls
AProVE07-11	0.30	0.30	0.30	0.30	0.30	0.30
velev-vliw-uns-4.0-9	1.01	1.00	1.01	1.00	1.01	1.00
11pipe_q0_k	1.11	1.03	1.11	1.03	1.11	1.03
maxor128	1.21	1.00	1.21	1.00	1.21	1.00
8pipe_k	1.09	1.15	1.09	1.15	1.09	1.15
blocks-blocks-36-0.130-NOTKNOWN	0.77	0.75	0.77	0.75	0.77	0.75

図 7 に inclusion detection unit の概要を示す．Literal inclusion checker は clause register (current) に格納された節が，Data read unit から読み込まれたリテラルに含まれるか判定する．Clause register (next) は次の探索において clause register (current) にセットされる節を格納する．節 C のリテラル数を $|C|$ とすると，L 個のリテラルを同時に比較することで (本システムは 16 個のリテラルを同時に比較する)，C の包含関係を $|C|/L$ クロックサイクルで判定することができる．

4.2.3.2 Resolution と Hyper-unary-resolution

計算量を減らし，回路を単純化するために hyper-unary-resolution，variable elimination by substitution と resolution は，S_x と $S_{\overline{x}}$ ($S_{\overline{x}}$ はリテラル $\overline{x}$ を含む全ての節の集合) に含まれる節の数 N_x と $N_{\overline{x}}$ が閾値 R_t 以下であり ($N_x + N_{\overline{x}} \leq R_t$)，かつ生成された resolvent の数が $N_x + N_{\overline{x}}$ 以下である場合に適用される．表 4 に $R_t = 32, 64, 128$ とした場合の削減率を示す (0.94 は 6%削減率が低下したことを示す)．表 4 より，R_t が大きいほど削減率が大きくなるが，その向上率は小さいことが分かる．本論文では R_t を 64 とする．

Variable elimination by substitution または resolution を適用する前に，definition detection unit を用いて x についての definition を探索し，hyper-unary-resolution が適用できるかを判定する．

Definition は $(x \vee \overline{a} \vee \overline{b}), (\overline{x} \vee a), (\overline{x} \vee b)$ といった節のセットであるため，長

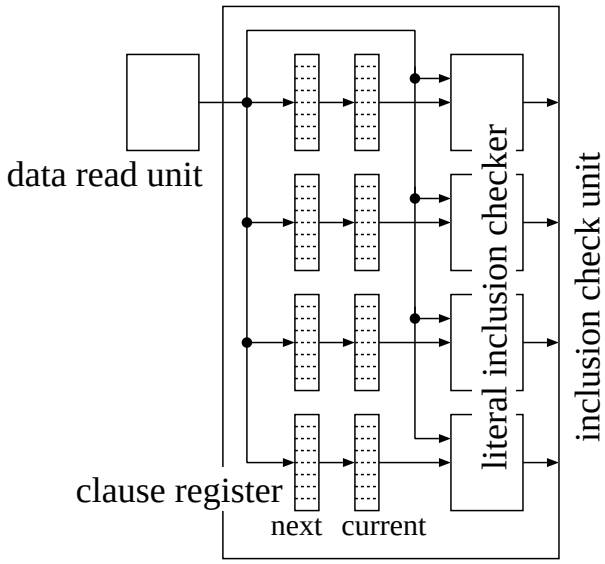

図 7: Inclusion detection unit

さが 2 の節に含まれる全てのリテラルの否定を含む節を見つけることで検出することができる．一方，hyper-unary-resolution は $(\overline{a}\vee\overline{b}), (\overline{x}\vee a), (\overline{x}\vee b)$ といった節が含まれていた場合，$\overline{x}$ は真であると決定するアルゴリズムである．Hyper-unary-resolution の判定は definition の判定とほぼ同様であるため，definition detection unit を用いて definition の探索中に行うことができる．

$(x \vee \overline{a} \vee \overline{b}), (\overline{x} \vee a), (\overline{x} \vee b)$ といった definition が含まれていた場合，一番長い節 $C_d = (x \vee \overline{a} \vee \overline{b})$ が definition detection unit に保持される．この間 $(\overline{a} \vee \overline{b}), (\overline{x} \vee a), (\overline{x} \vee b)$ といった節が含まれ，hyper-unary-resolution を適用することができると判定された場合，変数 x が F より削除され，次の変数の処理へ移行する．

Definition の探索が終了し hyper-unary-resolution が適用されなかった場合，resolution unit によって resolvent の生成が開始される．

C_d に含まれるリテラルの否定と読み込まれた節に含まれるリテラルは並列に比較することができる．節の包含関係の判定と違い C_d の長さ以下の節のみを対象とするため，一つの節につき 1 クロックサイクルで判定を行うことができる．

本論文において C_d の最大長は 8 とした．表 5 に C_d の最大長を $3, 8, 16$ とした場合の削減率を示す (0.97 は 3%削減率が低下したことを示す)．表 5 より C_d の最大長が 8 以上であれば十分な削減率を達成することができ

表 4: R_t に対する削減率の変化

	R_t					
	32		64		128	
ベンチマーク	#var	#cls	#var	#cls	#var	#cls
AProVE07-11	0.30	0.30	0.30	0.30	0.30	0.30
velev-vliw-uns-4.0-9	1.01	1.00	1.01	1.00	1.01	1.00
11pipe_q0_k	1.08	1.01	1.12	1.02	1.12	1.02
maxor128	1.21	1.00	1.21	1.00	1.21	1.00
8pipe_k	0.94	0.94	1.00	0.99	1.24	1.36
blocks-blocks-36-0.130-NOTKNOWN	0.73	0.70	0.73	0.71	0.80	0.78

ることが分かる．

図 8 に resolution unit の概要を示す．Definition が見つかった場合 variable elimination by substitution を適用し，見つからなかった場合は resolution を適用する．

各 clause merger は，例えば $C_1 = (x \vee l_0 \vee l_1 \vee l_2)$ と $C_2 = (\overline{x} \vee l'_0 \vee l'_1 \vee l'_2)$ から resolvent $C_1 \otimes C_2 = (l_0 \vee l_1 \vee l_2 \vee l'_0 \vee l'_1 \vee l'_2)$ を生成する． この時例えば $l_0 = v$ かつ $l'_2 = \overline{v}$ であるならば $C_1 \otimes C_2$ は常に真となり，消去することができる．本論文においては，

1. 事前に各節の全てのリテラルを変数番号順にソートしておく．
2. C_1 と C_2 から一つずつリテラルを変数番号順に順次取り出すことで，C_1 と C_2 の resolvent を生成する．
3. 2. において，例えば C_1 と C_2 内に v と $\overline{v}$ が含まれていた場合，resolvent の生成を停止し，その resolvent を破棄する (C_1 と C_2 中のリテラルは 1. において変数番号順にソートされているため，v と $\overline{v}$ が C_1 と C_2 中に含まれることを容易に検出することができる)．

resolvent の数が $N_x + N_{\overline{x}}$ を超えた場合，resolvent の生成を停止し，次の変数へ移行する．この時生成した resolvent は全て破棄される．

Variable elimination by substitution においては，resolvent $G_x \otimes G_{\overline{x}}$ と $R_x \otimes R_{\overline{x}}$ の生成をする必要がない (第 4.1.3.3 章)．これらをスキップするた

表 5: $|C_d|$ に対する削減率の変化

	$\|C_d\|$					
	3		8		16	
ベンチマーク	#var	#cls	#var	#cls	#var	#cls
AProVE07-11	0.17	0.15	0.22	0.21	0.30	0.30
velev-vliw-uns-4.0-9	1.00	1.00	1.01	1.00	1.01	1.00
11pipe_q0_k	1.12	1.02	1.12	1.02	1.12	1.02
maxor128	1.21	1.00	1.21	1.00	1.21	1.00
8pipe_k	0.97	0.98	0.97	0.98	0.97	0.98
blocks-blocks-36-0.130-NOTKNOWN	0.78	0.76	0.79	0.77	0.79	0.77

めに，節が clause merger へ供給される前に definition detection unit を用いて definition の一部であるか判定する．Clause merger へ供給される二つの節が共に definition の一部であるか，共に definition の一部ではない場合，その resolvent の生成は行われず次の resolvent の生成に移行する．

4.2.4 本システムにおける削減率

表 6 に，様々な規模の問題に対する本システムの問題規模の削減率とソフトウェアにおける削減率との比較を示す．最初の 3 列は変数 (#var)，節 (#cls) とリテラル (#lit) の削減率を示す (例えば，0.38 は 62%の変数を削除することができたことを意味する)．次の 3 列はソフトウェアとの比較である．1.00 はソフトウェアと同等の削減率を意味し，1.01 はより高い削減率を，0.22 は低い削減率を意味する．表 6 に示したように，本システムとソフトウェアによる削減率はベンチマークによって異なる (良い場合もあれば，悪い場合もある)．これは，本システムとソフトウェアでは，同じアルゴリズムを用いてはいるものの，どのリテラルから各アルゴリズムが適用されるかという順番が異なるからである (本システムでは計算時間が最短となるようにリテラルが選択される)．AProVE07-11 を除けば，平均値としては，ほぼ同じ削減率を達成している．AProVE07-11 において本システムの削減率が低い理由は，hyper-unary-resolution におけるリテラルの選択順序の影響が特に大きいことと，表 6 の比較ではソ

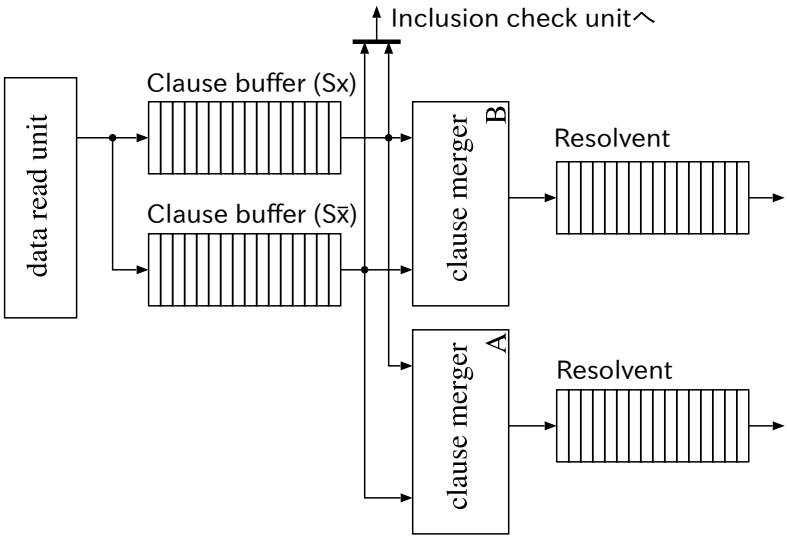

図 8: Resolution unit

フトウェア (SatElite) をデフォルトの状態で実行しているが，その状態では本論文で紹介したアルゴリズム ([38] に記述されているもの) 以外も用いられているためである (ソースコードが公開されておらず，マニュアルも完備されていないため詳細は不明である)．ソフトウェアに対して本システムと同様のアルゴリズムのみが適用されるようオプションを指定した場合と，本システムにおいて hyper-unary-resolution におけるリテラルの処理順を [38] と同じにした場合の，変数/節/リテラルの削減率は，それぞれ 0.24/0.34/0.44 (ソフトウェア)，0.29/0.41/0.46 (本システム) と，かなり近い値となる．

プリプロセッサの真の性能は，単なる削減率ではなく，ソルバと組み合わせた時の総計算時間によって評価されるべきであるが，ここではプリプロセッサにおける削減率のみを示した．これは，非常に多くのソルバが提案されており，どのソルバーと組み合わせるかにより，有効なアルゴリズムの適用順が異なること，また，一つのソルバに限定しても，問題毎に有効な適用順が異なると考えられるためである．

表 6: 本システムの削減率

	削減率			ソフトウェアとの比較		
ベンチマーク	#vars	#cls	#lits	#vars	#cls	#lits
11pipe_q0_k	0.54	0.96	1.18	1.10	1.01	0.9
8pipe_k	0.87	0.99	1.1	0.95	0.94	0.89
9vliw_m_9stages_iq3_C1_bug9	0.91	0.99	1.11	1.01	1.00	0.92
9dlx_vliw_at_b_iq4	0.68	0.89	1.00	1.06	1.03	1.00
aaai10-planning-ipc5-TPP-30-step11	0.96	0.97	1.00	0.89	0.85	0.88
AProVE07-11	0.38	0.57	0.62	0.22	0.21	0.22
blocks-blocks-36-0.130-NOTKNOWN	0.97	0.98	1.00	0.73	0.71	0.69
E02F22	0.86	1.00	1.00	0.98	0.97	0.91
maxor128	0.27	0.57	0.80	1.21	1.00	1.04
velev-vliw-uns-4.0-9	0.86	0.98	1.12	1.01	1.00	0.92

4.3 性能評価

本論文では回路を Xilinx Kintex-7 XC7K325T に実装した．回路規模は 40K FFs (利用可能な FF の 10%)，53K LUTs (利用可能な LUT の 26%)，445 36Kb Block RAMs (利用可能な Block RAM の 100%) である．本システムでは各モジュールの並列度を以下のようにした．

1. Inclusion check unit において，16 個の節を同時に比較し (図 7 の inclusion detection unit を 16 個同時に動作させる)，さらに各ユニットにおいて 16 個のリテラルを同時に比較する．

2. Resolution と variable elimination by substitution を $N_x + N_{\overline{x}} \leq 64$ の場合に適用し，clause merger を 2 個用いる．

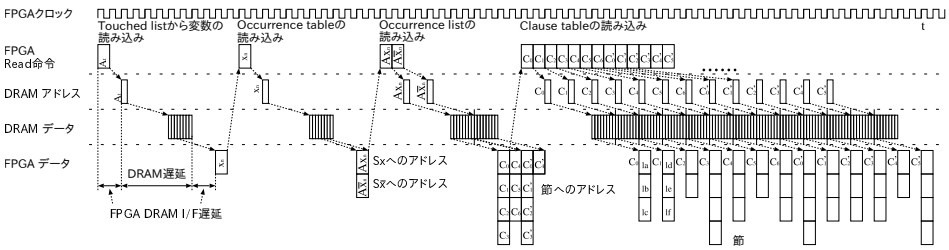

図 9: DRAM へのアクセスシーケンス

本論文の実装では，評価に用いた問題の規模に合わせてリテラルのビット幅を 24 ビットとした (2^{23} 個の変数まで対応可能)．DRAM のデータ幅は 64 ビットであるため，64 ビットに 2 リテラルを格納することとなる．各テーブル中に格納されるポインタ (DRAM アドレス) としては，アラインメントを考慮して，64 ビットを用いた．この回路に基づいて，本システムをハードウェアシミュレータを用いて性能評価を行った．シミュレータを用いることで，同時に動作するユニット数や DRAM の遅延を変更させた場合の高速化率を評価することができる．

図 9 にシミュレーションにおける DRAM へのアクセスタイミングを示す．FPGA は 200MHz で動作する．DRAM I/F と DRAM は 400MHz で動作し，バースト転送は 800MHz(400MHz の立ち上がり 立ち下がり) で行われる．データ転送長は 8 ワード固定である (図 9 の灰色領域は不必要なデータで，DRAM I/F により無視される)．図 9 において，まず touched list が読み込まれる． DRAM 中の touched list に保持される変数は FPGA DRAM I/F の遅延 + DRAM のアクセス遅延だけ遅れて FPGA に読み込まれる．シミュレーションではこのメモリアクセス遅延を FPGA のクロックサイクルで 50 クロックとした (本システムにおいては 250 ナノ秒であり，DRAM アクセスの遅延時間と DRAM インタフェースの遅延時間の合計としては適切な値であると考えている) ．次いで occurrence table と occurrence list が読み込まれた後，clause table より節が読み込まれる．全ての節のリテラル数が十分に小さい時，これらの節は clause buffer に格納され，その後は 1 クロックサイクル毎に節を読むことができる．

表 7 に本システムとソフトウェアの計算時間との比較を示す．読み込んだ節を clause buffer にキャッシュすることによって，ソフトウェアに比べ 3.4 倍から 54.2 倍の高速化を実現することができる．読み込んだ節のキャッシュを行わない場合，高速化は最大 41.6 倍に留まる．表 7 より

表 7: 計算時間 (秒) と高速化率

	Clause buffer 使用時		Clause buffer 未使用時	
benchmark	t(sec)	speedup	t(sec)	speedup
11pipe_q0_k	5.59	16.41	159.24	0.58
8pipe_k	3.81	16.88	129.71	0.50
9dlx_vliw_at_b_iq4	12.29	19.30	40.30	5.89
9vliw_m_9stages_iq3_C1_bug9	586.22	5.43	2556.71	1.24
aaai10-planning-ipc5-TPP-30-step11	2.44	22.12	2.66	20.28
AProVE07-11	69.92	12.63	285.03	3.10
blocks-blocks-36-0.130-NOTKNOWN	18.07	3.44	21.23	2.93
E02F22	4.67	37.44	184.99	0.95
maxor128	1.24	54.15	1.62	41.57
velev-vliw-uns-4.0-9	23.3	5.88	66.40	2.06

clause buffer によるキャッシングを行わない場合，ソフトウェアに比べ計算時間が増える場合がある．これはキャッシングを行わない場合，節の包含関係の判定において節を繰り返し読み込むため，DRAM へのアクセスが大幅に増えるためである (S_x と $S_{\overline{x}}$ に含まれる L 個以下のリテラルを含む節数を $|S_L|$ とすると S_x と $S_{\overline{x}}$ を $|S_L|/L$ 回読み込む)．

図 10 に inclusion detection unit において同時に比較する節数を変更した場合の計算時間を示す．図 10 より，節の包含関係の判定において並列に動作するユニット数を増やすことでより大きな高速化を行うことができるが，16 より並列度を増やした場合計算時間の変化は小さくなることが分かる．これは次の 2 つの理由によるものと考えられる．まず，評価に用いたベンチマークでは S_x の平均数が 26 程度であるため，処理の並列度も平均 26 程度にとどまる．また，ユニット数が 16 個の時の velev-vliw-uns-4.0-9 における処理の内訳を見ると，約 60%の計算時間がメモリアクセスに費やされている．これは，一旦読み込まれた節は clause buffer に格納され，再利用されるが，ユニット数がある程度以上になると FPGA 内での計算時間が十分に高速化され，最初の読み込みに要する時間が支

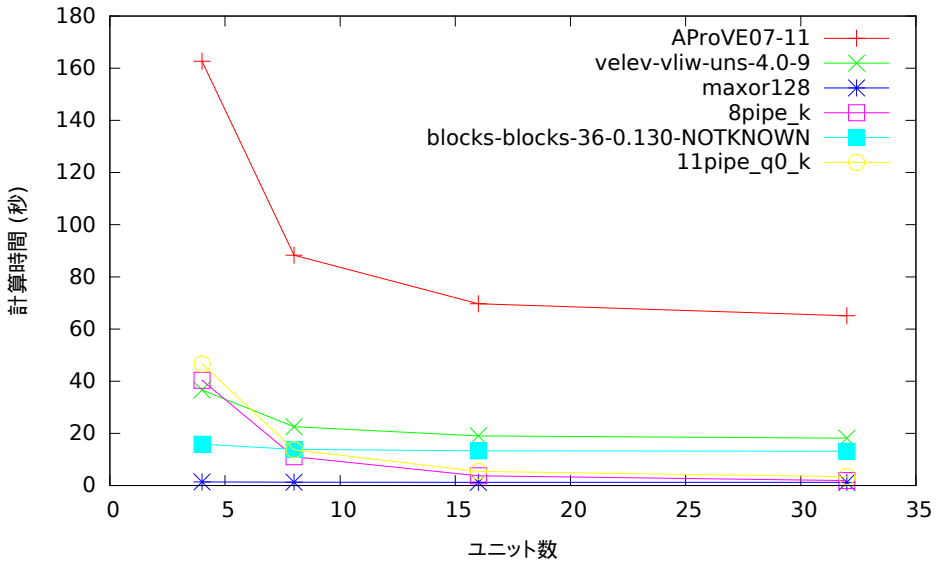

図 10: Inclusion detection unit の並列度と計算時間

配的となり始めるからである．より高速な処理を実現するためには，より大規模な FPGA を用いて clause buffer を 2 個実装し，あるリテラルに関する処理を実行している間に，次のリテラルに対する節を先行読み出しすることが有効だと考えている．

図 11 に resolution unit において並列に動作する clause merger のユニット数を 2，4，9 と変更した場合の計算時間を示す．これらのユニット数は block RAM の dual port access を用いることで各 clause merger に 2 系統のリテラルを並列に供給することができる数である．9 より多くの clause merger を動作させるためには clause merger と data read unit 間にネットワークを構築する必要があり，複雑な回路と制御が必要となる．図 11 において並列に動作するユニット数をある程度以上増やしても，処理速度の向上が得られていないが，これは以下の理由によるものである．まず，本システムでは，変数 x に対する resolution において，$N_x + N_{\overline{x}}$ 個の resolvent が生成された場合，x に対する resolution の処理を強制的に終了し，次の変数の処理に移行する．もし，新たに生成された全ての resolvent が明らかに真ではない場合 (削除可能ではない場合)，1 回の節のスキャンにより 2 個のユニットから生成される resolvent の数は，ほぼ $N_x + N_{\overline{x}}$ 個となる ($N_x \simeq N_{\overline{x}}$ の場合)．このため，resolution が有効に機能しない変数に対しては，ユニット数を 2 以上としても，それ程の速度向上

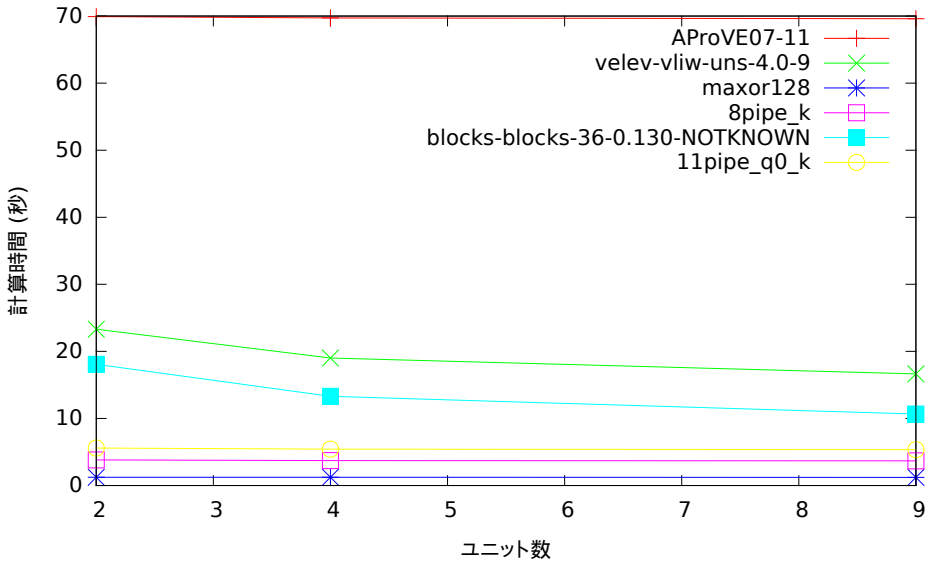

図 11: Resolution unit の並列度と計算時間

は望めない．また，このような場合，上記の inclusion detection unit の並列度と計算時間に議論の場合と同様に，約 60%の計算時間がメモリアクセスに費やされている (velev-vliw-uns-4.0-9 の場合)．より高速な処理を実現するためには，上記と同様に，より大規模な FPGA を用いて clause buffer を 2 個実装し，あるリテラルに関する処理を実行している間に，次のリテラルに対する節を先行読み出しすることが有効である．

表 8: 各フェーズにおける連続メモリアクセスの比率

ベンチマーク	全体	節の包含関係の判定	節のマージ
AProVE07-11	0.89	0.12	0.92
velev-vliw-uns-4.0-9	0.93	0.86	0.96
maxor128	0.16	0.09	0.19
8pipe_k	0.81	0.82	0.80
blocks-blocks-36-0.130-NOTKNOWN	0.72	0.73	0.72
11pipe_q0_k	0.80	0.84	0.70

表 8 に各フェーズにおける連続メモリアクセスの比率を示す．第二列における 0.89 は全メモリアクセスのうち 89%を，第三列における 0.12 は節

の包含関係の判定におけるメモリアクセスのうち 12%を，第四列における 0.92 は節のマージにおけるメモリアクセスのうち 92%を連続メモリアクセスが占めることを意味する．表 8 より，ほとんどのベンチマークに対し本システムは多くの連続メモリアクセスを行っており，メモリ転送幅を活用することができていることがわかる．一部のベンチマークのフェーズで連続メモリアクセスを行うことができない理由は，そもそもそのフェーズで適用が試みられる各アルゴリズムが適用可能となる割合が少なく連続メモリアクセスを行う回数がないことと (例えば AProVE07-11 における節の包含関係の判定フェーズは全計算時間の 9.4%のみを占める)，問題の特徴によるものであるためである (maxor128 は以下に詳しく述べるように，メモリから読み出されるデータ長が短く，連続メモリアクセスが有効に機能しない)．

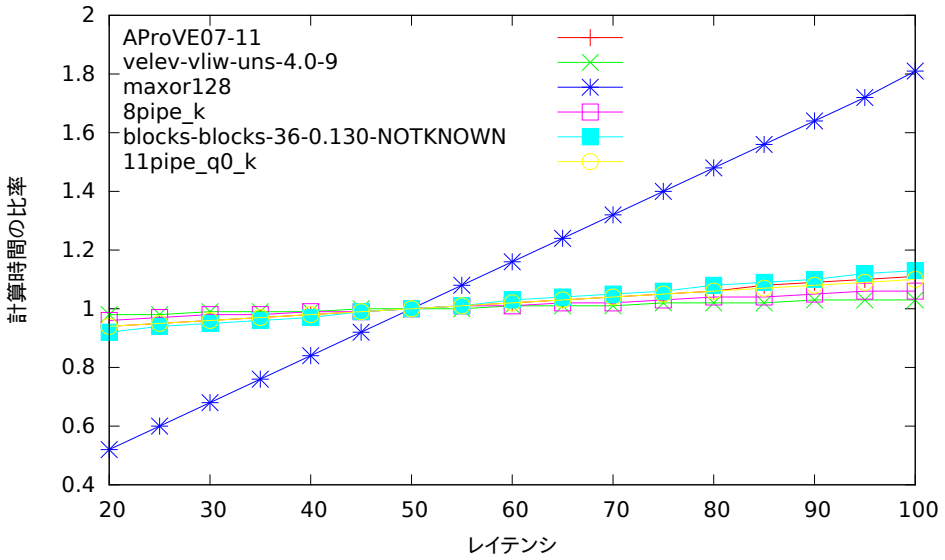

図 12: メモリアクセス遅延と計算時間の変化

図 12 にメモリアクセス遅延を変更した場合の計算時間の変化を示す．図 12 では，メモリアクセス遅延が 50 クロックサイクルである時の計算時間を 1 としている．図 12 において，ベンチマーク maxor128 を除き計算時間の変化は 15%未満であり，本システムがメモリアクセス遅延を隠蔽することができていることが分かる．他のベンチマークと異なり maxor128 の計算時間は，メモリアクセス遅延の影響を受けている．表 9 に，幾つかのベンチマークにおける，あるリテラル l を含む節の数の平均値と，l を

含む節に含まれるリテラルの総数の平均値と最大値を示す．この表から分かるように，maxor128 では，他のベンチマークと比べて節の数も，リテラルの総数も著しく少ない．このため，外部メモリからの各データ読み出しにおいて読み出されるデータ量が少なく，メモリアクセス遅延の占める割合が相対的に大きくなる．また本システムが実現している並列度が節の数に比べて十分大きいため，各アルゴリズムの処理も十分に高速化されている (表 6 に示したように maxor128 の高速化率は最も高い)．その結果，相対的にメモリアクセス遅延が全実行時間において大きな割合を占めるようになる．このような問題の比率はベンチマークにおいては少なく，また，本実装による高速化率も他のベンチマークと比べて高いことから，メモリアクセス遅延を隠蔽しているとは言い難いが，本実装で十分であると考えている．このような問題に対しては，

1. 表 9 に示した数値を事前に調べることにより，このような問題の検出を行い，

2. 小さな clause buffer を複数持つ回路を別途用意し (maxor128 においては表 9 の最大値に示したように高々 645/2 個 (各節は少なくとも 2 個のリテラルを持つ) の節をキャッシングすればよい)，次に処理する変数が決定している場合 (hyper-unary-resolution 以外の場合がこれに該当する)，それらに対する先行読み出しを行う回路を別途用意し，その回路を用いる

ことにより，ある程度のメモリアクセス遅延の隠蔽が可能となると考えている．

表 9: リテラル l を含む平均節数とそれらの節に含まれるリテラルの総数

ベンチマーク	あるリテラル l を含む節の数の平均	あるリテラル l を含む節に含まれるリテラルの数の総和	
		平均	最大
AProVE07-11	5.9	30.1	599402
velev-vliw-uns-4.0-9	27.4	401.3	248959
maxor128	3.5	8.5	645
8pipe_k	56.1	1271.7	641454
blocks-blocks-36-0.130-NOTKNOWN	19.5	535.4	7996
11pipe_q0_k	42.8	1227.5	553247

第5章
プリプロセッサUnhidingの高速化

本章ではプリプロセッサUnhidingの高速化について述べる [43]．Unhidingはリテラルの依存関係を基に節とリテラルを削除するプリプロセッサであり，高い並列性を持つ．Unhidingのアルゴリズムはスタンピングと単純化の二つのフェーズからなる．スタンピングにおいてはキャッシング，単純化においてはキャシングとバンクインタリーブによるメモリアクセスの最適化を行った．本システム全体の高速化率は1.3から2.7倍程度にとどまったが，単純化フェーズのみでは3倍程度(シングルスレッドとの比較においては4.8倍から7.3倍)でありバンクインタリーブによるメモリアクセスの効率化が本システムにおいて有効であることを確認した．

5.1 Unhiding

UnhidingはSATソルバLingeling [40]のSATプリプロセッサの一つである．Unhidingはリテラルの依存関係を検出し，それを元にリテラルと節を削除する．表10にSAT competition 2014 [9]のベンチマークに対しUnhidingを適用後の変数，節とリテラル数を示す(表10の実行環境はCPUがIntel Core i7-2600，メモリ容量が8GBであり4コア8スレッド下で実行している)．u.c.%はベンチマークに含まれる単位節の割合を示す．これらの単位節とそれらに含まれる変数はSATソルバか他のプリプロセッサのBCPフェーズにて削除される．表10よりUnhidingを用いることでリテラル数を大きく削減することができ，単位節を大きく増やすことができることがわかる．

表 10: ベンチマークと Unhiding 適用後のベンチマークの変数の数，節数，リテラル数と計算時間

	original			preprocessed(ratio)			
ベンチマーク	変数の数 ($\times 10^6$)	節　数 ($\times 10^6$) (u.c.%)	リテラル数 ($\times 10^6$)	変数の数	節　数 (u.c.%)	リテラル数	計算時間 (秒)
SAT_dat.k95-24_1_rule_2	1.54	6.04(0.79%)	15.2	0.95	0.97(49%)	0.60	5.03
9vliw_m_9stages_iq3_C1_bug9	0.52	13.4(0%)	39.2	0.99	1.00(54%)	0.50	7.98
blocks-blocks-36-0.120-NOTKNOWN	0.49	8.78 (0.016%)	19.1	1.00	1.00(51%)	0.68	5.75
SAT_dat.k90.debugged	1.46	5.72(0.79%)	14.3	0.95	0.97(50%)	0.60	4.77
grid-strips-grid-y-3.065-SAT	0.48	3.97(0.016%)	9.59	1.00	0.99(23%)	0.73	2.99

Unhiding のアルゴリズムは以下の二つのフェーズからなる．

スタンピング
タイムスタンプを各リテラルにつけることにより，リテラルの依存関係を検出する．

単純化
リテラルの依存関係を元に節とリテラルを削除する．

5.1.1 スタンピング

リテラルの依存関係「 a ならば b」を「 $a \to b$」と表記し，式 2 のように定義する．すなわち，リテラルの依存関係「 a ならば b」は 2-リテラル節で表される．実応用問題にはこのようなリテラルの依存関係が多く含まれている．

$$(a \to b) \equiv (\overline{a} \vee b) \tag{2}$$

論理式 $F = (\overline{a} \vee c) \wedge (\overline{b} \vee c) \wedge (\overline{b} \vee d) \wedge (\overline{c} \vee e) \wedge (\overline{c} \vee f) \wedge (\overline{d} \vee f)$ には図 13 に示すリテラルの依存関係が含まれている．このようなリテラルの依存関係のグラフを *binary implication graph* (*BIG*) と呼ぶ．

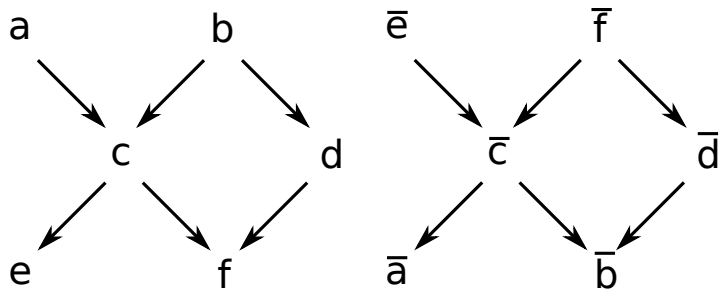

図 13: Binary Implication Graph

BIG を構築するために，Unhiding では深さ優先探索を行い、**タイムスタンプ**を各リテラルにつける．アルゴリズム 1 に各リテラルにタイムスタンプをつけるための擬似コードを示す (0 はスタンプが与えられていないことを示す)．リテラル l のタイムスタンプは以下のスタンプのセットである．

discovered ($dsc(l)$)
深さ優先探索において，l に最初に訪れた時に付けられるタイムスタンプ (アルゴリズム 1 の 8 行目).

finished ($fin(l)$)
深さ優先探索において，l から親リテラルに戻る時に付けられるタイムスタンプ (アルゴリズム 1 の 17 行目).

observed ($obs(l)$)
l にアクセスしたときに更新されるスタンプ．l が複数の親リテラルを持つとき，$dsc(l)$ は一度のみ付けられるのに対し，$obs(l)$ は他の親リテラルからアクセスされた時にも更新される (アルゴリズム 1 の 14，18 行目)．スタンピング中の節の削除などに使用される．

discovered of the root literal ($root_dsc(l)$)
ルートリテラルの discovered．スタンピング中の節の削除などに使用される．

parent($prt(l)$) l の親リテラル.

論理式 F に対し，アルゴリズム 1 に従いタイムスタンプ $[dsc(l) : fin(l) : obs(l) : root_dsc(l)]$ を各リテラルに割り当てた例を図 14 に示す．図 14 においては，まずルートリテラルの一つ a が選択され discovered がつけられる (**ルートリテラル**は親リテラルを持たないリテラル)．次に c，e とたどり discovered がつけられる．e には子リテラルが存在しないため finished がつけられ，c へ戻り次の子リテラル f に移動する．a の子リテラルをすべて探索した後，他のルートリテラルを選択し (図 14 では $\overline{f}$)，同様の処理を繰り返す．

タイムスタンプからリテラルの依存関係を検出するために，*parenthesis theorem* を用いる．この定理より，リテラル u，v が $dsc(u) < dsc(v)$ かつ $fin(u) > fin(v)$ を満たす時，v は u の子リテラルである．図 14 において実線で表された依存関係が，parenthesis theorem により検出された BIG である．Parenthesis theorem により全てのリテラルの依存関係を検出することはできないが (例えば図 14 における破線で表された依存関係)，タイムスタンプの比較のみで高速にリテラルの依存関係を検出することができる．

深さ優先探索によりリテラルにタイムスタンプをつけている間，以下のアルゴリズムを適用し，タイムスタンプのつけ直しと，節と変数の削

アルゴリズム 1 タイムスタンプの付与手順

```
List root literals (RTS)
stamp := 0
for all root ∈ RTS do
    Stamp(stamp, l, stamp)
end for
function STAMP(stamp, l, root_dsc)
    stamp := stamp + 1
    dsc(l) := stamp
    root_dsc(l) := root_dsc
    for all (l ∨ l′) ∈ F do
        if dsc(l′) = 0 then
            Stamp(stamp, l′, root_dsc)
        end if
        obs(l′) := stamp
    end for
    stamp := stamp + 1
    fin(l) := stamp
    obs(l) := stamp
end function
```

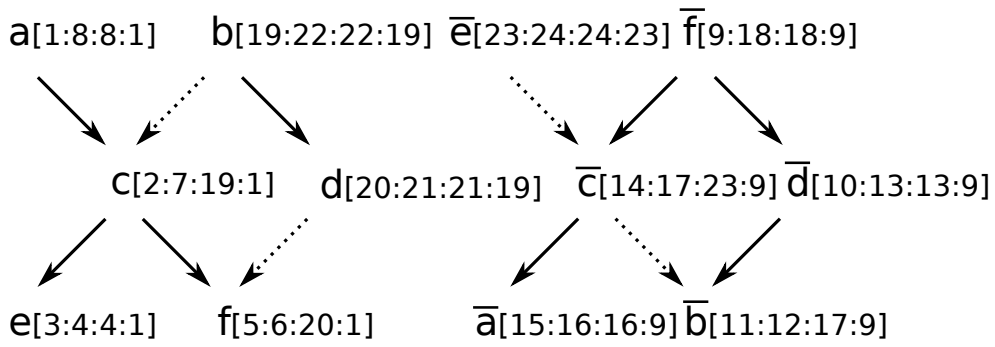

図 14: タイムスタンプ例

除を行う．

1. Failed literal elimination (FLE)
2. Transitive reduction (TRD)
3. Equivalent literal substitution (ELS)

5.1.1.1 Failed literal elimination

単位節 (x) を論理式 F に追加すると F が偽になる時，リテラル x を *failed literal* と呼ぶ．$(l \to l')$ に対し，l' と $\overline{l'}$ が共通する親リテラル x を持つとき，x は真でなければならず $\overline{x}$ は failed literal となる．$(l \to l')$ が $root_dsc(l) \leq obs(\overline{l'})$ を満たす時，l' と $\overline{l'}$ は共通する親リテラルを持つ．これは図 15 に示すような $(root(l) \to x) \wedge (x \to \overline{l'}) \wedge (x \to l) \wedge (l \to l')$ といった依存関係が含まれていることを意味する ($root(l)$ は l のルートリテラル).

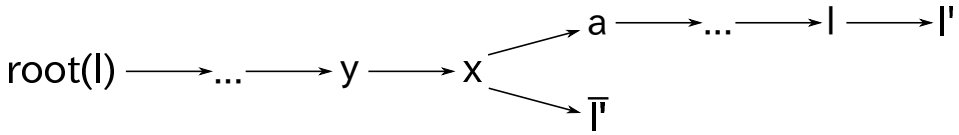

図 15: Failed literal の例

$(l \to l')$ のタイムスタンプが上記の条件を満たす時，Unhiding はスタックを遡り $dsc(x) \leq obs(\overline{l'})$ を満たす x を探索する．x が $dsc(x) \leq obs(\overline{l'})$ を満たすならば，$\overline{x}$ は failed literal である．$\overline{x}$ が failed literal であると判定

されたならば，単位節 (x) を F へ追加する (変数 x は SAT ソルバまたは他の SAT プリプロセッサの BCP フェーズにて削除される)．このような単位節の追加を *failed literal elimination* (*FLE*) と呼ぶ．

5.1.1.2 Transitive reduction

論理式 $F = (u \rightarrow a) \wedge (a \rightarrow v) \wedge (u \rightarrow v)$ において $(u \rightarrow v)$ は $(u \rightarrow a) \wedge (a \rightarrow v)$ より導出することができる (図 16)．BIG においてこのような他のパスから導出できるパスを *transitive edge* と呼ぶ．Transitive edge は他のパスから導出できるため，F より削除しても F の真偽は変わらない．

F より trasitive edge を削除する処理を *transitive reduction*(*TRD*) と呼ぶ．

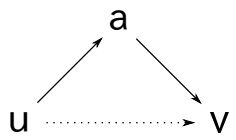

図 16: Transitive edge の例

5.1.1.3 Equivalent literal substitution

節 $(l \rightarrow l')$ が $fin(l') = 0$ (l' の finished にスタンプが与えられていない) かつ $dsc(l') < dsc(l)$ を満たす時，$(l \rightarrow l') \wedge (l' \rightarrow l)$ といった環状の依存関係が存在する．

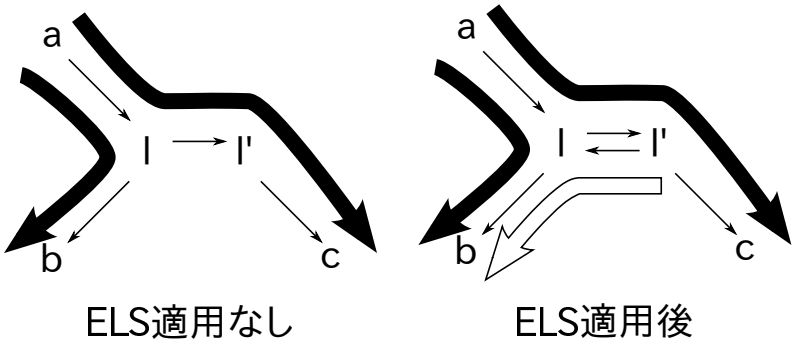

図 17: ELS の例

この時 l と l' に同じタイムスタンプをつけることでより多くのリテラルの依存関係を検出することができる．論理式 $F = (a \to l) \wedge (l \to l') \wedge (l' \to l) \wedge (l \to b) \wedge (l' \to c)$ に対し，アルゴリズム 1 のみ処理した場合 $(l \to b)$，$(l \to c)$，$(l' \to c)$ のみを検出することができる (図 17 における黒矢印)．l と l' に同じタイムスタンプをつけることで，さらに $(l' \to b)$ を検出することができる (図 17 における白矢印)．

l と l' のようなリテラルを *equivalent literal* と呼び，equivalent literal に同じタイムスタンプをつける処理を *equivalent literal substitution*(*ELS*) と呼ぶ．

5.1.2 単純化

スタンピングフェーズにより付けられたタイムスタンプを元に，リテラルの依存関係を検出し，以下のアルゴリズムを適用することで節とリテラルを削除する．

1. Unhiding tautology elimination (UHTE)
2. Unhiding literal elimination (UHLE)

5.1.2.1 Unhiding Tautology Elimination

論理式 $F = (a \vee b \vee c) \wedge (\overline{a} \to x) \wedge (x \to b)$ において，式 3 より節 $(a \vee b \vee c)$ を削除することができる．

$$
\begin{aligned}
&(a \vee b \vee c) \wedge (\overline{a} \to x) \wedge (x \to b) \\
&\equiv ((a \vee b \vee c) \wedge (\overline{a} \to b)) \wedge (\overline{a} \to x) \wedge (x \to b) \\
&\Rightarrow (\overline{a} \to x) \wedge (x \to b) \qquad (3)
\end{aligned}
$$

F における $(a \vee b \vee c)$ を *hidden tautology* と呼び，hidden tautology を削除する処理を *hidden tautology elimination*(*UHTE*) と呼ぶ．

タイムスタンプの比較により節 C が hidden tautology であるか判定することができる．C に含まれるリテラル l_{pos} と C に含まれるリテラルの否定 l_{neg}(l_{neg} は C に含まれない) が $dsc(l_{neg}) < dsc(l_{pos})$ かつ $fin(l_{neg}) > fin(l_{pos})$ を満たす時，C は hidden tautology である．

ただし $|C| = 2$ であり，$l_{pos} = \overline{l}_{neg}$ または $prt(l_{pos}) = l_{neg}$ である時，C は hidden tautology ではない．$l_{pos} = \overline{l}_{neg}$ である場合，l_{neg} は failed literal

であるため $((\bar{l} \rightarrow l) \equiv (l))$ ，この依存関係は残したほうがよい．また $prt(l_{pos}) = l_{neg}$ である場合，$(prt(l_{pos}) \rightarrow l_{pos})$ であり，この依存関係は削除することができない．

5.1.2.2 Unhiding Literal Elimination

論理式 $F = (a \vee b \vee c \vee d) \wedge (a \rightarrow b) \wedge (b \rightarrow c)$ において，式 4 より $(a \vee b \vee c \vee d)$ から a と b を削除することができる．式 4 は self-subsumption(第 4.1.2 章) と逆の式変形である．

$$
\begin{aligned}
&(a \vee b \vee c \vee d) \wedge (a \rightarrow b) \wedge (b \rightarrow c) \\
&\equiv ((a \vee b \vee c \vee d) \wedge (\overline{a} \vee b)) \wedge (b \rightarrow c) \\
&\equiv ((b \vee c \vee d) \wedge (\overline{b} \vee c)) \wedge (a \rightarrow b) \\
&\equiv (c \vee d) \wedge (a \rightarrow b) \wedge (b \rightarrow c) \qquad (4)
\end{aligned}
$$

このようにしてリテラルを削除する処理を *unhiding literal elimination* (*UHLE*) と呼び，この処理によって削除されるリテラルを *hidden literal* と呼ぶ．

タイムスタンプの比較により，hidden literal を検出することができる．節 C に含まれるリテラル l と l' が $dsc(l) < dsc(l')$ かつ $fin(l) > fin(l')$，または $dsc(\bar{l}) > dsc(\overline{l'})$ かつ $fin(\bar{l}) < fin(\overline{l'})$ を満たす時，l は hidden literal である．

5.2 FPGA への実装

図 18 に本システムの概要を示す．Stamping unit は深さ優先探索を行い，各リテラルにタイムスタンプを付ける．同時に *TRD*，*ELS*，*FLE* を適用する．Simplification unit は *UHTE* と *UHLE* を適用し，リテラルと節を削除する．Time stamp cache はタイムスタンプのアソシエイティブキャッシュである．一度読み込まれたタイムスタンプは time stamp cache にキャッシュされ再利用される．

5.2.1 データ構造

図 19 に本システムで用いるデータ構造を示す．Short clause table は長さ 32 以下の節の先頭アドレスと節の長さを格納する．本システムにおい

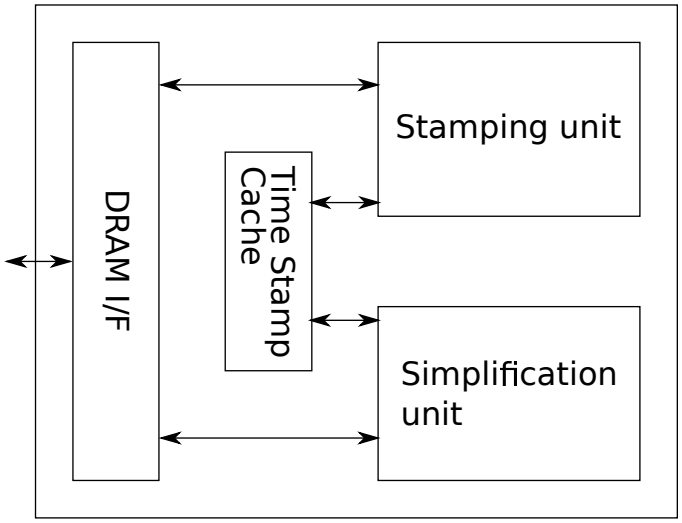

図 18: システム概要

て short clause table は一度構築された後は変更されない．Clause list は第 4.2.1 章における clause table と同様であり，節に含まれるリテラルを格納する．

Binary occurrence table はリテラル l の binary occurrence list の先頭アドレスと終端アドレスを格納する．Binary occurrence list は l を含む 2-リテラル節の，l でないリテラルを格納する (第 4.2.1 章における occurrence list が節へのアドレスを格納するのに対し，binary occurrence list はリテラルを直接格納する)．本システムにおいて，binary occurrence table 及び binary occurrence list は一度構築された後は変更されない．

Time stamp map は変数からタイムスタンプを取得することができる連想配列である．Time stamp map はスタンピングにおいて構築と更新がされ，単純化においては変更されない．

これらのデータ構造は DRAM 上に格納される．

5.2.2 処理手順

本システムにおける処理手順はソフトウェアと同様である．

1. Stamping unit にてタイムスタンプを各リテラルにつける．同時に TRD，ELS，FLE を適用する．

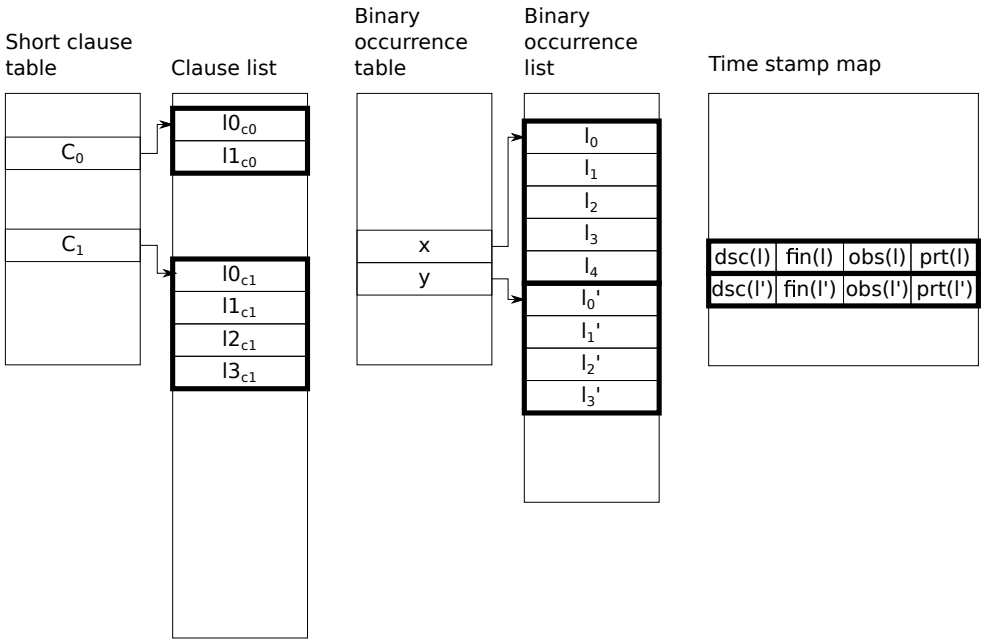

図 19: データ構造

2. Simplification unit にて上記 1. で構築した BIG を元に UHTE と UTLE を適用する．

本システムのメモリアクセスは以下の 3 つに大別することができる．

Binary occurrence list のリニアスキャン

ルートリテラルをリストするために，binary occurrence list をスキャンする．これは全て連続アクセスであり，バーストリードが可能である．

スタンピングにおける深さ優先探索

このメモリアクセスはランダムであり，メモリアクセス遅延が大きく性能に影響を与える．本システムでは専用のキャッシュによりメモリアクセス遅延を隠蔽することを目指す．

単純化における各節のタイムスタンプの読み込み

このメモリアクセスはランダムであるが，読み込みの順序が事前にわかる．本システムでは 16 個の節を同時に比較するために，まず 8 個の DRAM バンクに対応した 8 個のキューに各リテラルのアドレスをソートし格納する．その後 8 個の DRAM バンクに順々に読み

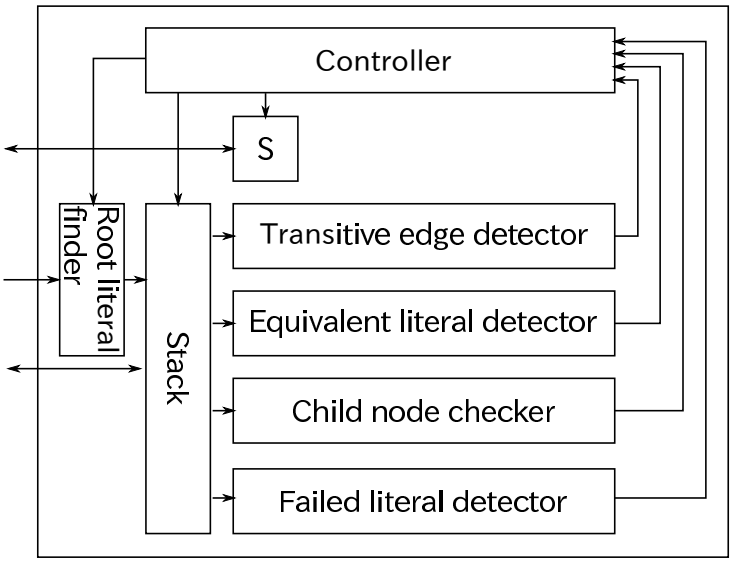

図 20: Stamping unit 概要

込みをかけることで，メモリ転送幅を最大限活用する．また，専用のキャッシュによりメモリアクセス遅延を隠蔽する．

5.2.3 Stamping unit

図 20 に stamping unit の概要を示す．Stamping unit において，まず root literal finder を用いてルートリテラルをリストする．リテラル l が論理式に含まれていない時，l はルートリテラルである ($(x \to l)$ が論理式に含まれていないことを意味する)．すなわち binary occurrence list をスキャンし，l が含まれているかどうか調べることでルートリテラルを検出することができる．

深さ優先探索によるタイムスタンプの付加において，二つのスタック (図 20 における Stack と S) が使用される．Stack には以下のデータがスタックされる．

1. リテラル l.
2. l の binary occurrence list.
3. l と $\bar{l}$ のタイムスタンプ.
4. FLE と TRD が適用された時にセットされるフラグ.

表 11: Stack と S の深さ

benchmark	Stack の深さ	S の深さ
E00N23	23	23
blocks-blocks-36-0.120-NOTKNOWN	4	4
blocks-blocks-37-1.120-NOTKNOWN	4	4
aaai10-planning-ipc5-pipesworld-27-step14	10	12
traffic_r_uc_sat	119	119

5. l が equivalent literal である時にセットされるフラグ.

S は親リテラルに戻る際に $fin()$ がつけられるリテラルのスタックである.

表 11 に各ベンチマークにおいて必要とされる Stack と S の深さを示す. 表 11 より本論文で用いたベンチマークでは Stack と S の深さは 128 あれば十分であり，block RAM を用いることで実装可能である．本システムでは Stack と S の深さを 256 とした.

アルゴリズム 2 にタイムスタンプの処理手順を擬似コードで示す．擬似コードにおいて，*parallel*，*parallel if* と *parallel else if* は並列に実行される.

Stamping unit では failed literal detector，transitive edge detector，equivalent literal detector と child node checker の 4 モジュールが同時に動作する (アルゴリズム 2 における行 34，39，44，56)．Failed literal detector，transitive edge detector と equivalent literal detector はそれぞれ FLE，TRD と ELS を適用する．Child node checker は探索すべき子リテラルが存在するか判定する．これらの結果は controller に送られ，Stack と S が更新される.

読み込まれた子リテラルは Stack 内にキャッシュされ, 再利用される．読み込まれたタイムスタンプは time stamp cache にキャッシュされ，再利用される．スタンピングにおいて $obs(l)$ は頻繁に更新されるため (アルゴリズム 2 の 4 行目, アルゴリズム 2 の 48,62 行目)，l のタイムスタンプが time stamp cache にキャッシュされているならば，DRAM 上と time stamp cache 上の $obs(l)$ を同時に更新する.

アルゴリズム 2 スタンピング処理手順

```
 1: List root literals (RTS)
 2: stamp := 0
 3: while root ∈ RTS do
 4:     Push root to Stack
 5:     while Stack is not empty do
 6:         l is the head of Stack
 7:         if l does not have children then
 8:             Processed on Stack
 9:                 if observed != 0 then                                  ▷ FLE
10:                     if dsc(l_failed) > observed then
11:                         l_failed := l
12:                     else
13:                         parallel
14:                             Write {l̄_failed}; observed := 0
15:                         end
16:                     end if
17:                 end if
18:
19:             Processed on S
20:                 if is_equivalent(l) == false then                      ▷ ELS
21:                     stamp := stamp + 1
22:                     repeat
23:                         parallel
24:                             l' := S.pop()
25:                             dsc(l') := dsc(l)
26:                             fin(l') := stamp
27:                         end
28:                         Write l' time stamp to DRAM
                            and Time stamp cache
29:                     until l'! = l
30:                 end if
31:
```

```
        else                                                          ▷ l has children
            Read a time stamp of l′ (one of a
            l child) from DRAM or Time stamp
            cache
                                is_visited        ==
            parallel if         false&&
                                dsc(root(l)) ≤ obs(l̄′)
                parallel                                              ▷ FLE
                    l_failed := l; observed := obs(l̄′)
                    is_visited(l) := true
                end
            parallel else if    is_visited(l) == false
                                && dsc(l) < obs(l′)
                parallel                                              ▷ TRD
                    Remove {l̄ ∨ l′} from Clause table
                    is_visited(l) := true
                end
            parallel else if    is_visited(l) == false
                                && dsc(l) == 0
                parallel
                    Processed on Stack                                ▷ Visit l′
                        stamp := stamp + 1
                        dsc(l′) := stamp; obs(l′) := stamp
                        is_visited(l′) := false
                        is_equivalent(l′) := false
                        Read l′ children
                                                                      ▷ Update l time stamp
                        is_visited(l) := true

                    Push l′ to S
                end
            parallel else if    fin(l′) == 0&&
                                dsc(l′) < dsc(l)
                parallel                                              ▷ ELS
                    dsc(l) := dsc(l′)
                    is_equivalent(l) := true
                end
            end if
            obs(l′) := stamp
        end if
    end while
end while
```

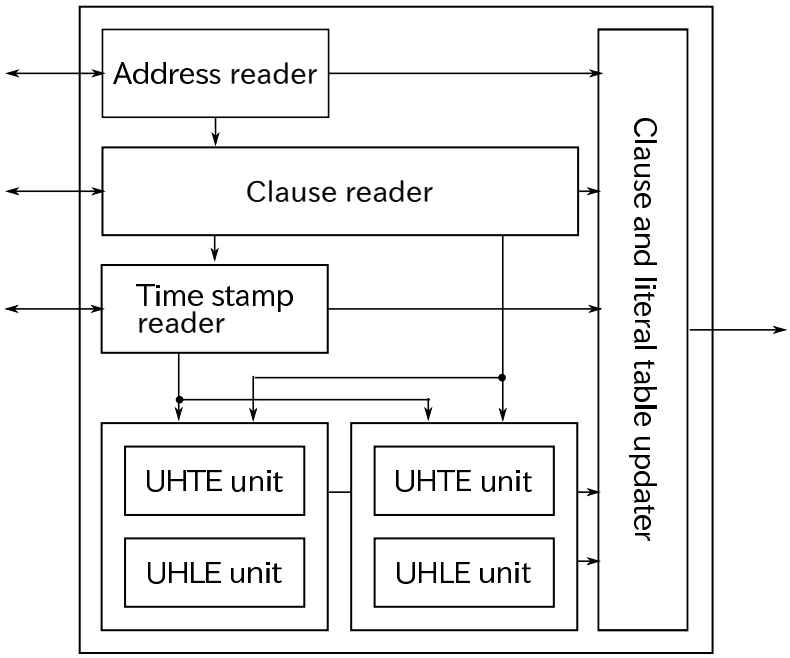

図 21: Simplification unit 概要

5.2.4 Simplification unit

図 21 に simplification unit の概要を示す．スタンピングの後，simplification unit にて UHTE と UHLE を適用する．

UHTE と UHLE では $dsc(l_{neg}) < dsc(l_{pos})$ かつ $fin(l_{neg}) > fin(l_{pos})$，または $dsc(l) < dsc(l')$ かつ $fin(l) > fin(l')$ を満たす節内のリテラルの探索が行われる (第 5.1.2 章)．どちらの処理も節内のタイムスタンプのみを比較することで行うことができる．このため，全ての節に対しアルゴリズムを並列に適用することができる．本システムでは 16 個の長さ $L_s = 32$ 以下の節を並列に処理するとともに，各節に対して UHTE と UHLE を並列に適用する．

Simplification unit の処理手順は以下のとおりである．

1. Address reader が short clause table より L_s(本システムでは 32) 以下の長さの節のアドレスを N(本システムでは 16) 個読み込む．

2. Clause reader が Clause list より N 個の節に含まれるリテラルを読み込む．

3. Time stamp reader が各リテラルのタイムスタンプを読み込む．

表 12: L_s に対するリテラルの削減率

	L_s			ソフトウェアと $L_s = 32$ の時との比較
ベンチマーク	16	**32**	64	
SAT_dat.k95-24_1_rule_2	0.39	**0.40**	0.40	1.00
9vliw_m_9stages_iq3_C1_bug9	0.34	**0.41**	0.44	0.86
blocks-blocks-36-0.120-NOTKNOWN	0.25	**0.25**	0.27	0.91
SAT_dat.k90.debugged	0.39	**0.40**	0.40	1.00
grid-strips-grid-y-3.065-SAT	0.19	**0.19**	0.19	0.89

4. UHTE ユニットと UHLE ユニットが UHTE と UHLE を N 個の節に並列に適用する．

5. Clause and literal table updater がリテラルと節を消去する．

6. 1. から 5. を長さ L_s 以下の全ての節に対し適用する．

L_s を変化させた時のリテラルの削減率の変化を表 12 に示す．表 12 において，0.41 は 41%のリテラルを削除したことを意味する．表 12 より L_s が 32 以上であるならば，ソフトウェアとほぼ同等の削減率に達することがわかる (ソフトウェアでは UHTE が非常に長い節を削除することがある．節数は少ないが，これがリテラルの削減率に影響している)．本システムでは L_s を 32 とした．

単純化において，ボトルネックとなるのが上記 3. のタイムスタンプの読み込み時のメモリアクセス遅延である．これは単純に読み込みを行おうとするとランダムアクセスを必要とするためである．ランダムアクセスによる待機時間の発生を防ぐために，バンクインタリーブを行う．論理式 $F = (a \vee b \vee c) \wedge (b \vee c \vee d) \wedge (c \vee d \vee e)$ に対し $K = 2$ 個のバンクを持つ DRAM へバンクインタリーブを行う例を示す．

1. $K = 2$ 個のキュー ($Q0$ と $Q1$) に F 中のリテラルを格納する．この時 $Q0$ には DRAM のバンク 0 に格納されているタイムスタンプに対応するリテラルが，$Q1$ には DRAM のバンク 1 に格納されているタイムスタンプに対応するリテラルが格納される．論理式 F に対し

DRAM バンク 0 には $\{a, c, e\}$ が，バンク 1 には $\{b, d\}$ が格納されている場合，$Q0$ には $\{a, c, c, c, e\}$，$Q1$ には $\{b, b, d, d\}$ が格納される．

2. $Q0$ と $Q1$ に格納されているリテラルの読み込み命令を各 DRAM バンクに交互に送る．この例の場合，読み込み命令の発行順番は $\{a, b, c, b, c, d, c, d, e\}$ となる．タイムスタンプが time stamp cache にキャッシュされている場合は time stamp cache からタイムスタンプを読み込む．

この手法では，全ての DRAM アクセスをインターリーブすることはできないが，単純な実装で効率的なメモリアクセスを実現することができる．また N 個の節を並列に処理するために十分に強力な手法である．

この処理において，リテラルは節に格納された順番に読み込まれるとは限らない．このため，読み込まれたタイムスタンプは time stamp reader によって再配置される．

5.3 性能評価

本システムを Xilinx Kintex-7 XC7VLX325T に実装した．この回路は DRAM I/F を除いて 32 K LUTs と 340 個の block RAM を使用する．動作周波数は 200MHz である (これは DRAM I/F に依存する)．このシステムにおいて，最大 8192 個のタイムスタンプが time stamp cache にキャッシュされる．また UHTE と UHLE は長さ 32 以下の節 16 個に対し並列に適用される．

本システムの性能評価は全てハードウェアシミュレータによって行った．ハードウェアシミュレータを用いることで，最適な並列度を決めたりメモリアクセス遅延を変更した時の性能の変化を評価したりすることができる．

表 13 に本システムの計算時間とソフトウェアに対する高速化率を示す．また表 14 にスタンピングと単純化それぞれにおける，ソフトウェアに対する高速化率を示す．表 13 と表 14 における 1.29 は 1.29 倍ソフトウェアに比べ高速化したことを意味する．表 13 に示したとおり，本システムの高速化率は 1.3 から 2.3 倍程度とそれ程大きいとは言えない．これは以下の理由による．

1. ソフトウェアは 4 コア 8 スレッドを使用する (表 13 と表 14 において Intel Core i7-2600 3.4GHz を使用している)．表 13 より，1 コア

表 13: 計算時間と高速化率

		高速化率	
ベンチマーク	計算時間	4コア 8スレッド	1コア 1スレッド
SAT_dat.k95-24_1_rule_2	3.91	1.29	1.84
9vliw_m_9stages_iq3_C1_bug9	3.52	2.27	4.34
blocks-blocks- 36-0.120- NOTKNOWN	2.96	1.94	3.08
SAT_dat.k90.debugged	3.70	1.29	1.82
grid-strips-grid-y-3.065-SAT	1.81	1.65	2.76

表 14: 各フェーズにおける高速化率

	4コア8スレッド		1コア 1スレッド
ベンチマーク	スタンピング	単純化	単純化
SAT_dat.k95-24_1_rule_2	0.69	3.68	4.85
9vliw_m_9stages_iq3_C1_bug9	1.30	3.49	7.31
blocks-blocks- 36-0.120- NOTKNOWN	1.21	3.37	5.71
SAT_dat.k90.debugged	0.69	3.68	4.83
grid-strips-grid-y-3.065-SAT	0.91	3.74	6.74

1スレッドのソフトウェアに対し，本システムは1.8から4.3倍の高速化を行うことができることが分かる．

2. マイクロプロセッサは本システムより大きなキャッシュメモリ (8MB) を持つ．

3. 本システムでは，メモリアクセスの複雑さからスタンピングが十分に高速化されていない．表14に示すとおり，スタンピングにおける高速化率は0.69から1.30倍にとどまる．これはタイムスタンプのキャッシングによるメモリアクセス回数の削除しか行われていないためである．一方，単純化における高速化率は3.4から3.7倍 (1コア1スレッド下においては4.8から7.3倍) であり，十分な高速化率を達成したといえる．これは単純化においてタイムスタンプのキャッシングによるメモリアクセス回数の削除やバンクインタリーブを活

用した並列処理が行われているためである．

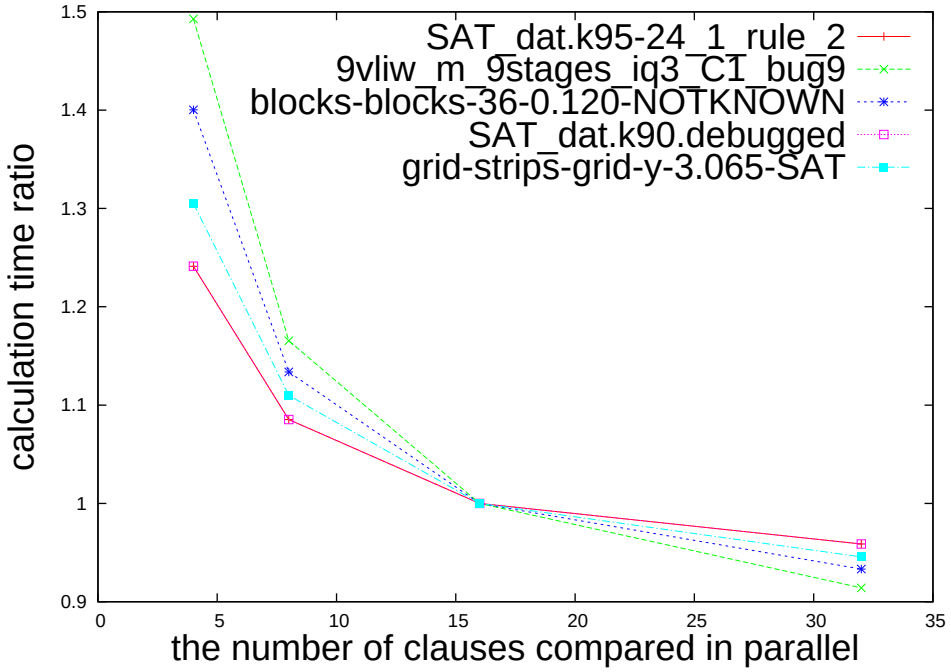

図 22: Simplification unit における並列性と計算時間

図 22 に並列に UHTE と UHLE を適用する節数を変更した際の，計算時間の変化を示す (並列処理する節数が 16 の時の計算時間を 1 とする)．図 22 より，並列に処理を行う節数を増やすほど計算時間は短くなるが，その減少幅は小さくなる．本システムでは 16 個の節に対し並列に UHTE と UHLE を適用する．

表 15 に各フェーズにおけるメモリアクセスの種類の比率を示す．第二列における 0.017 はスタンピングにおける全メモリアクセスに対し，連続アクセスが 1.7%を占めることを意味する．第三列における 0.55 は単純化におけるメモリアクセスに対し連続アクセスが 55%を，第四列における 0.18 はバンクインタリーブが 18%を占めることを意味する．表 15 よりスタンピングにおいて連続メモリアクセスはほとんど行われていないことがわかる．これは子リテラルのタイムスタンプの読み込みがランダムアクセスであることと，連続アクセスを行うことができる子リテラルの読み込みは stack を用いたキャッシングによりその回数が抑えられていることによるものである．単純化におけるメモリアクセスは連続アクセスとバンクインタリーブが多くを占める．すなわち，本システムにおいては

表 15: 各フェーズにおけるメモリアクセスの比率

	スタンピング	単純化	
ベンチマーク	連続アクセス	連続アクセス	バンクインタリーブ
SAT_dat.k95-24_1_rule_2	0.017	0.55	0.18
9vliw_m_9stages_iq3_C1_bug9	0.046	0.63	0.073
blocks-blocks-36-0.120-NOTKNOWN	0.043	0.63	0.067
SAT_dat.k90.debugged	0.017	0.55	0.18
grid-strips-grid-y-3.065-SAT	0.026	0.60	0.12

単純化においてメモリアクセスを著しく効率化することができたことを意味する．表 15 において，単純化における連続アクセスとバンクインタリーブの比率が大きく異なる理由は，バンクインタリーブが行われるタイムスタンプの読み込みの際には，time stamp cache によりキャッシュされたタイムスタンプが再使用されることがあり，これがメモリアクセスの回数を抑えているためである (タイムスタンプの読み込みにおいてキャッシングが有効である理由は，リテラルか変数をキーとして問題中の節がソートされていることがあるためである)．

表 15 より単純化においてはリテラルやタイムスタンプのキャッシングが有効に働いていること，また本システムのボトルネックはスタンピングにおける子リテラルのタイムスタンプの読み込みであることがわかる．

図 23 にメモリアクセス遅延を変更した際の計算時間の変化を示す．図 23 においては本システムの 50 クロックサイクル (250 ナノ秒) を 1 として計算時間を正規化している．図 23 よりメモリアクセス遅延に対する計算時間の変化は小さく，本システムがメモリアクセス遅延を隠蔽することができていることを示す．図 23 において，SAT_dat.k95-24_1_rule_2 と SAT_dat.k90.debugged は他のベンチマークに比べメモリアクセス遅延の影響を大きく受けている．これはこの二つのベンチマークでは総計算時間に対しスタンピングに要する時間の比率が大きいためである．

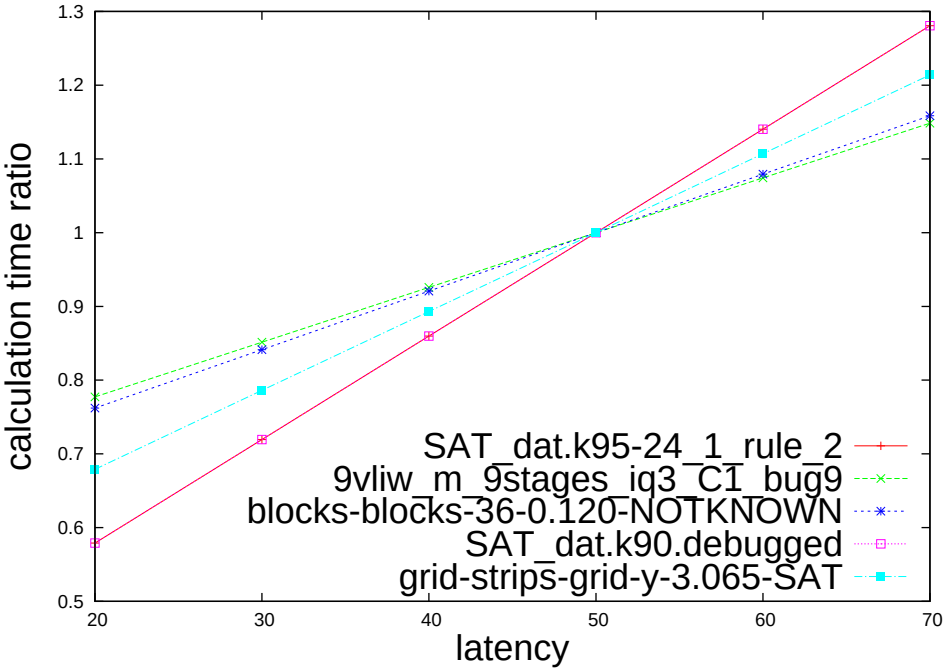

図 23: メモリアクセス遅延と計算時間

第6章
まとめと今後の展望

本章では本論文のまとめと，今後の展望について述べる．

6.1 まとめ

本論文では2つのSATプリプロセッサSatELiteとUnhidingの高速化手法について述べた．本論文ではそれぞれのSATプリプロセッサのアルゴリズムのFPGAへの実装方法，メモリアクセスの効率化手法とアクセス遅延や並列度を変えた際の性能の変化を調べた．

SatELiteは高い並列性(節の包含関係におけるリテラルの比較とresolventの生成等)を持ち，ハードウェアを用いて高速化しやすいアルゴリズムからなる．しかし，並列化したときボトルネックとなるのがメモリアクセスである．本システムはキャッシングによりメモリアクセスの回数を抑えることで，メモリアクセス遅延を隠蔽した．さらに並列化とメモリアクセス最適化を行うことにより，3.4倍から54倍の高速化を行うことができた．しかし，問題によっては，高速な処理速度は実現できてはいるものの，メモリアクセス遅延を隠蔽できていない．これは，これらの問題においては 一回の処理の対象となる節の数が少ないため，あるリテラルの計算時間と，そのリテラルの処理に必要な節の読み込み時間を比べたときに，リテラルの計算時間の方が短くなってしまっているためである．このような問題に対しては節数が少ないことを利用して，複数のリテラルに対する節の保持が可能となるようにclause bufferを改良し，複数のリテラルに対する節の先読みを行うことにより，ある程度まで遅延を隠蔽することができると考えているが，この実装は今後の課題である．

Unhidingのアルゴリズムも基本的に高い並列性を持つ．スタンピングにおいては各アルゴリズムを並列に適用することができるとともに，複数のルートリテラルから探索を行うことができるが，FPGAの内部メモリ量の制約から今回の実装では複数のルートリテラルからの並列探索は実現できていない．単純化においては複数の節，複数のリテラル並列に処理するためにバンクインタリーブを行った．ランダムに発生する8個のDRAMバンクに対するメモリアクセスを，8個のブロックRAMを用いてソートし，各DRAMバンクへのアクセスを順繰りに行うことによりほ

ぼ最大限にメモリ転送幅を使用することができた．これらの工夫により単純化において十分な高速化を行うことができ，この手法が大規模な問題に対し有効であることを示すことができた．本システムにおいて，スタンピングに関しては十分に高速化することができなかった．スタンピングも単純化と同様にバンクインタリーブを活用することにより高速化することが可能である．これについては今後の課題である．

6.2 今後の課題

本論文においては二つの SAT プリプロセッサの FPGA への実装を行い，高速処理を実現するために並列化とメモリアクセスの最適化を行った．並列化については並列に動作するユニット数を変化させることで評価を行った．しかし，メモリアクセス最適化についての十分な評価を行うことができなかった．これは以下の理由による．

1. ハードウェアにおいて，シミュレータによりどのようなメモリアクセスが行われているか調べることができる．しかし，ソフトウェアにおいては，どのようなメモリアクセスが行われているか調べることは困難である．そのため，比較を行うことが難しい．

2. メモリアクセス最適化が一切働いていない場合の評価を行うことが難しい．これはその評価を行うためにバーストリードなどが行われないように，別途データ構造やアルゴリズムを再構築する必要があるためである．

メモリアクセス最適化手法の有効性を評価するためには，メモリアクセス最適化手法を評価するためのデータ構造やシミュレータの作成，ソフトウェアにおけるメモリアクセスのプロファイリングなど，メモリアクセスの有効性を評価する手法を構築する必要がある．

本論文では 2 つの SAT プリプロセッサを扱ったが，他にも優秀な SAT プリプロセッサが存在し [23] [24]，また今後も開発されていくものと考えられる．それらの高速化においてこれらのメモリアクセス最適化手法が有効であるか，検討・調査する必要がある．

本システムはハードウェア SAT ソルバまたはソフトウェア SAT ソルバと共に用いられるものである．そのため，本システムとそれら SAT ソルバを組み合わせて評価する必要がある．しかし，SAT プリプロセッサの

有効性は問題の種類，変数の適用順序やSATソルバのアルゴリズムなどに大きく依存する．そのため，実際に使用する際にはベンチマークなどを用いてSATプリプロセッサの組み合わせを試し，チューニングを行う必要がある．

これらは今後の課題である．

謝辞

筑波大学システム情報工学研究科知能機能システム専攻の丸山勉教授には，大学四年次の研究室配属時より研究全般に渡りご指導とご教鞭をいただきました．先生にはFPGAとFPGAを用いたSATプリプロセッサという興味深い研究テーマを与えていただきました．私がこれまで研究を続けてこられたのも先生のご指導あってこそだと思います．ここに深く感謝申し上げます．

本論文の審査を快くお引き受けいただくと共に本研究についての貴重なご助言をしてくださった，筑波大学システム情報工学研究科知能機能システム専攻の鬼沢武久教授と延原肇准教授，同コンピュータサイエンス専攻の安永守利教授と山口佳樹准教授に深く感謝申し上げます．

筑波大学システム情報工学研究科知能機能システム専攻丸山研の皆様とOBの方々に大変お世話になりました．ここに感謝を申し上げます．

最後にこれまで支えてくださった祖父母，両親と妹に深く感謝いたします．

参考文献

[1] Xilinx Inc. Xilinx. 5 Feb. 2016. 10 Feb. 2016.

[2] すすたわり . FPGA 入門 回路図と HDL によるディジタル回路設計. 秀和システム, 2012.

[3] 高原卓也, et al. “SRAM 型 FPGA を用いた高信頼性衛星搭載計算機システム.” 電子情報通信学会論文誌 B 88.1 (2005): 90-98.

[4] Iwakiri, Naohiko, Hirobumi Saito, and Shinichi Nakasuka. “Hardware implementation and laboratory performances of X-band nano satellite high-speed downlink receiver based on turbo equalizer/decoder (ICSANE 2013 (International Conference on Space, Aeronautical and Navigational Electronics).” 電子情報通信学会技術研究報告. SANE, 宇宙・航行エレクトロニクス 113.335 (2013): 127-132.

[5] 1364-2005 - IEEE Standard for Verilog Hardware Description Language. 18 Apr. 2006. 10 Feb. 2016.

[6] CUDA C Programming Guide. 17 Jul. 2015. 10 Feb. 2016.

[7] Cook, Stephen A. “The complexity of theorem-proving procedures.” Proceedings of the third annual ACM symposium on Theory of computing, ACM (1971): 151-158.

[8] The International Conferences on Theory and Applications of Satisfiability Testing (SAT). 23 Oct. 2015. 10 Feb. 2016.

[9] SAT competition. 20 Jul. 2014. 10 Feb. 2016.

[10] Davis, Martin, and Hilary Putnam. “A computing procedure for quantification theory.” Journal of the ACM (JACM) 7.3 (1960): 201-215.

[11] 番原, et al. “SAT によるシステム検証.” 人工知能学会誌 25.1 (2010): 122-129.

[12] 鍋島英知. “SAT によるプランニングとスケジューリング (⟨特集⟩最近の SAT 技術の発展).” 人工知能学会誌 25.1 (2010): 114-121.

[13] Biere, Armin, et al. “Symbolic model checking without BDDs.” Springer Berlin Heidelberg (1999): 193-207.

[14] Kautz, Henry A., and Bart Selman. “Planning as Satisfiability.” ECAI. Vol. 92 (1992): 359-363.

[15] Kautz, Henry, and Bart Selman. “Pushing the envelope: Planning, propositional logic, and stochastic search.” Proceedings of the National Conference on Artificial Intelligence (1996): 1194-1201.

[16] SAT Research Group, Princeton University. zChaff. 20 Feb. 2009. 10 Feb. 2016.

[17] Niklas En, Niklas Srensson, MiniSAT2. 29 Nov. 2015. 10 Feb. 2016.

[18] Laurent Simon. Glucose’s home page. 10 Feb. 2016. 10 Feb. 2016.

[19] The SAT Solving Package Riss. 10 Feb. 2016. 10 Feb. 2016.

[20] Subbarayan, Sathiamoorthy, and Dhiraj K. Pradhan. “NiVER: Non-increasing variable elimination resolution for preprocessing SAT instances.” Theory and Applications of Satisfiability Testing. Springer Berlin Heidelberg (2005): 276-291.

[21] Bacchus, Fahiem, and Jonathan Winter. “Effective preprocessing with hyper-resolution and equality reduction.” Theory and Applications of Satisfiability Testing. Springer Berlin Heidelberg (2004): 341-355.

[22] Alfredsson, Johan. “The SAT Solver kw.” The SAT race 2008: Solver descriptions (2008).

[23] Piette, Cdric, Youssef Hamadi, and Lakhdar Sas. “Vivifying Propositional Clausal Formulae.” ECAI. Vol. 178 (2008): 525-529.

[24] Jrvisalo, Matti, Marijn JH Heule, and Armin Biere. “Inprocessing rules.” Automated Reasoning. Springer Berlin Heidelberg (2012): 355-370.

[25] JEDEC. 10 Feb. 2016. 10 Feb. 2016.

[26] Skliarova, Iouliia, and Antonio de Brito Ferrari. "Reconfigurable hardware SAT solvers: a survey of systems." Computers, IEEE Transactions on 53.11 (2004): 1449-1461.

[27] Suyama, Takayuki, Makoto Yokoo, and Hiroshi Sawada. "Solving satisfiability problems using logic synthesis and reconfigurable hardware." System Sciences, 1998., Proceedings of the Thirty-First Hawaii International Conference on. Vol. 7. IEEE (1998): 179-186.

[28] Suyama, Takayuki, et al. "Solving satisfiability problems using reconfigurable computing." Very Large Scale Integration (VLSI) Systems, IEEE Transactions on 9.1 (2001): 109-116.

[29] de Sousa, Jos T., J. M. Da Silva, and Miron Abramovici. "A configurable hardware/software approach to SAT solving." Field-Programmable Custom Computing Machines, 2001. FCCM'01. The 9th Annual IEEE Symposium on. IEEE (2001): 239-248.

[30] Gulati, Kanupriya, et al. "FPGA-based hardware acceleration for Boolean satisfiability." ACM Transactions on Design Automation of Electronic Systems (TODAES) 14.2 (2009): 33.

[31] Thong, Jason, and Nicola Nicolici. "FPGA acceleration of enhanced boolean constraint propagation for SAT solvers." Proceedings of the International Conference on Computer-Aided Design. IEEE Press (2013): 234-241.

[32] Kanazawa, Kenji, and Tsutomu Maruyama. "An FPGA solver for WSAT algorithms." Field Programmable Logic and Applications, 2005. International Conference on. IEEE (2005): 83-88.

[33] Kanazawa, Kenji, and Tsutomu Maruyama. "An FPGA solver for large SAT problems." Field Programmable Logic and Applications, 2006. FPL'06. International Conference on. IEEE (2006): 1-6.

[34] Kanazawa, Kenji, and Tsutomu Maruyama. "An FPGA solver for very large SAT problems." Field Programmable Logic and Applications, 2007. FPL 2007. International Conference on. IEEE (2007): 493-496.

[35] Kanazawa, Kenji, and Tsutomu Maruyama. “An Approach for Solving Large SAT Problems on FPGA.” ACM Transactions on Reconfigurable Technology and Systems (TRETS) 4.1 (2010): 10.

[36] Kanazawa, Kenji, and Tsutomu Maruyama. “An FPGA Solver for SAT-Encoded Formal Verification Problems.” Field Programmable Logic and Applications (FPL), 2011 International Conference on. IEEE (2011): 38-43.

[37] Kanazawa, Kenji, and Tetsuhiro Maruyama. “FPGA acceleration of SAT/Max-SAT solving using variable-way cache.” Field Programmable Logic and Applications (FPL), 2014 24th International Conference on. IEEE (2014): 1-4.

[38] Eén, Niklas, and Armin Biere. “Effective preprocessing in SAT through variable and clause elimination.” Theory and Applications of Satisfiability Testing. Springer Berlin Heidelberg (2005): 61-75.

[39] Heule, Marijn JH, Matti Jrvisalo, and Armin Biere. “Efficient CNF simplification based on binary implication graphs.” Theory and Applications of Satisfiability Testing-SAT 2011. Springer Berlin Heidelberg (2011). 201-215.

[40] Armin Biere. Lingeling, Plingeling and Treengeling. 9 Feb. 2016. 10 Feb. 2016.

[41] Suzuki Masayuki, and Tsutomu Maruyama. “Variable and clause elimination in SAT problems using an FPGA.” Field-Programmable Technology (FPT), 2011 International Conference on. IEEE (2011): 1-8.

[42] 鈴木将之, 丸山勉. “FPGA による SAT 問題のプリプロセッサの実現.” 情報処理学会論文誌 57.1 (2016): 343-352.

[43] Suzuki Masayuki, and Tsutomu Maruyama. “FPGA Acceleration of SAT Preprocessor.” ParCo2015 (2015).