情報処理学会論文誌 Vol.57 No.1 343–352 (Jan. 2016)

FPGAによるSAT問題のプリプロセッサの実現

鈴木 将之[1,a)] 丸山 勉[1,b)]

受付日 2014年10月9日, 採録日 2015年10月2日

概要：充足可能性問題（SAT 問題）は，論理式全体を真とするような変数への真偽の割当てを求める問題である．スケジューリングや回路検証など多くの問題が SAT に変換できるため，数多くの SAT ソルバが構築されてきた．しかし，SAT 問題は NP 完全であり，その計算量は非常に大きい．計算量を減らす 1 つの方法として，プリプロセッサによる問題規模の削減がある．本論文では FPGA を用いたプリプロセッサ（SatELite）の構築について述べる．論理式中の項や節の依存関係を調べることにより項や節を削除することができ，問題の探索空間を小さくすることができる．SatELite で用いられているアルゴリズムは並列化しやすく，ハードウェア化に向いている．しかし，実際の回路から生成された SAT 問題は，非常に規模が大きく，外部メモリとのデータ転送により，性能が制約される．本システムでは，FPGA 内にいくつかの節をキャッシュし，それらと他の節とを並列かつ連続的に比較する．また，比較された節は外部メモリのアクセス遅延を隠蔽するために再利用される．本システムの性能は問題規模に依存するが，大規模な問題においては高い高速化を達成することができた．

キーワード：SAT 問題，プリプロセッサ，FPGA，アクセラレータ

Variable and Clause Elimination in SAT Problems Using an FPGA

Masayuki Suzuki[1,a)] Tsutomu Maruyama[1,b)]

Received: October 9, 2014, Accepted: October 2, 2015

Abstract: The satisfiability (SAT) problem is to find an assignment of binary values to the variables which satisfy a given clausal normal form (CNF). Many practical application problems can be transformed to SAT problems, and many SAT solvers have been developed. SAT problem is, however, NP-complete and its computational cost is very high. In order to reduce the computational cost, preprocessors are widely used in SAT solvers. In this paper, we describe an approach for implementing a preprocessor (SatELite) on FPGA. In SatELite, the variables and clauses whose values can be uniquely determined from other variables and clauses are eliminated to reduce the search space of the given SAT problem. The algorithms used in SatELite have inherent parallelism, but the data size of the SAT problems is very large, and the performance of the system is limited by the throughput of the off-chip DRAM banks. In our implementation, several clauses are held on the FPGA, and are compared in parallel with a sequence of new clauses. The sequence is cached on the FPGA, and reused in order to hide the access delay to the DRAM banks. The speedup by our system depends on the problem size, however it becomes higher for larger problems.

Keywords: satisfiability problems, SAT preprocessor, FPGA, accelerator

1. はじめに

充足可能性問題（Satisfiability problem，SAT）は，選言のみから構成される節の連言で与えられる論理式に対し，式全体を真とする変数への真偽の割当てを求める問題である．すべての節に k 個のリテラルが含まれる問題は k-SAT 問題と呼ばれる．$k \leqq 3$ の問題は NP 完全であることが初めて証明された問題である．多くの問題が k-SAT 問題に変換できるため，SAT 問題を解くために数多くの SAT ソルバが構築されてきた [1]．形式的検証問題はハードウェアシステムを検証するための数理的手法であり，SAT 問題の重要な応用の 1 つである．形式的検証問題において，システムの正当性は問題が恒偽であるか否かで判定される．これらの問題は非常に問題規模が大きく，SAT ソルバのた

[1] 筑波大学
University of Tsukuba, Tsukuba Ibaraki, 305–8577, Japan
[a)] suzuki@darwin.esys.tsukuba.ac.jp
[b)] maruyama@darwin.esys.tsukuba.ac.jp

めのベンチマークとしても用いられている．

SAT 問題をより効率良く解くために，プリプロセッサが使われることがある [3], [4]．プリプロセッサには様々な種類があるが [3], [4], [5]，本論文の対象とする SatELite [6] は問題規模を削減することを目的とする．問題規模の削減は以下の手順で行われる．

(1) 冗長な節や，依存関係によって一意に値が決定される変数を削除する

(2) 節を変形したり置換したりすることによって，節の数を削減する

10^6 から 10^8 個の変数を含む大規模な問題に対してプリプロセスの計算時間はけっして短いとはいえない．

本プリプロセッサの高速化にはいくつかのハードウェアプラットフォームが考えられるが，データの単純な分割が困難なアルゴリズムであることから，より高速な処理を実現するためには，単一のメモリシステムにおいて高速化が可能な GPU および FPGA が候補となる．GPU も非常に多数のコアを有しており，一見プリプロセッサの高速化に適しているように思われる．しかし，以下の理由で複数のコアが効率的に機能せず，プリプロセッサを高速化することは難しい．

(1) 複数のリテラルに対する処理を並列に行おうとしても，それらの処理に必要なデータは一般的に DRAM 上で不連続に配置されている．

(2) あるリテラル l の処理における並列度は，l を含む節の数に依存するが，その平均値は最大 60 程度（問題により大きく異なるが 3〜60 程度）と必要とされるスレッド数に比べ少なく，GPU には十分な並列度とはならない．

(3) 各節の処理は，そのリテラルの状態に依存するため，複数の節を並列に処理しても，節ごとに分岐条件が異なる．

Field programmable gate array（FPGA）は SAT ソルバの高速化に適したデバイスである．まずプリプロセッサを FPGA 上に構築し，次に問題規模が削減された問題を解くためのソルバを構築することで，ハードウェアリソースを最大限活用することができるからである．しかし，すでに述べたように実際の問題規模は非常に大きく，FPGA の内部メモリに問題を格納するのは不可能である．このため外部 DRAM が必要であるが，DRAM のレイテンシやデータ転送幅の制限により，FPGA を用いた高速化は限られたものとなってしまう．高い性能を出すためには，レイテンシと限られたデータ転送幅を隠蔽する実装方法を考えなければならない．

本論文では SatELite [6] を基にしたプリプロセッサの実装手法について述べる．大規模な問題に対し，並列度や DRAM の遅延を変更した場合にどれだけ高速化できるのかハードウェアシミュレータを用いて評価した．SatELite は主にハードウェアの形式的検証問題から生成された問題に対し効果を発揮するが，それ以外の問題に対しても適用されることがある．本論文では，形式的検証問題から生成された問題のみを対象として，それらを用いてパラメータの設定を行い，それらの問題に対して評価を行った．本論文では，我々の従来研究 [10] では未実装であった hyper-unary-resolution の実装を新たに行った．また，さらに大規模な問題を用いて評価を行い，性能に関する解析をより詳細に行った．これにより，外部メモリへの参照が頻発する大規模問題に対するソフトウェアをほぼ完全な形で FPGA に実装したとき，それらの問題に対して，どの程度の性能向上が実現可能かを明らかにすることを目指す．このために，本論文では，できるだけ多くのメモリ参照がバーストリードとなるようにデータ構造，アクセス方法を工夫し，また，いったん読み込んだデータを再利用するための専用キャッシュメモリを用いることにする．

本論文の構成は次のとおりである．2 章では SAT について，3 章では SatELite について述べる．4 章では FPGA へのアルゴリズムの実装について，5 章ではその性能評価について述べる．6 章では考察と今後の展望を述べる．

2. 充足可能性問題

充足可能性問題（satisfiability problem，SAT）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 $F(x_1, x_2, \ldots, x_n)$ を真とするような変数（*variable*）への真偽の割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）として表現される．CNF は節（*clause*）どうしの連言であり，節はリテラル（*literal*）どうしの選言である．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．

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

これまでに多くの SAT ソルバが開発されてきたが，SAT 問題の規模は大きく（形式的検証問題から生成された問題の規模は特に大きい），プリプロセッサを用いた探索空間の削減が有効である．

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（文献 [7], [8], [9]．ここでは 3 つのみ参照する）．しかし，我々の知る限りハードウェアを用いた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

表 1 SatELite の適用結果
Table 1 Variable, clause and literal elimination by SatELite.

	適用前			適用後			
ベンチマーク	#variables	#clauses	#literals	#variables	#clauses	#literals	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

小さく，小規模な問題では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．

本研究室では大規模な問題に対する FPGA を用いた SAT ソルバを開発した [8], [9], [10]．本システムを用いることでこの FPGA を用いたソルバの性能を高めることができる．

3. SatELite

SatELite [6] は Eén らによって開発されたプリプロセッサである．SatELite は式全体の真偽が変わらないよう変数や節を削除し，問題の探索空間を削減する．**表 1** 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 1 の実行環境は CPU が Intel Core i7-2600，メモリ容量が 8 GB である．表 1 より，問題規模の大きな問題に対して問題規模を大きく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．

SatELite では大きく分けて以下のアルゴリズムが用いられている．

(1) clause elimination
(2) literal elimination
(3) variable elimination

3.1 Clause Elimination

2 つの節 $C_1 = \{a, b\}$，$C_2 = \{a, b, c\}$ が F に含まれているとする．C_1 が真であるならば C_2 は c の値に関係なく真となり，C_2 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．

3.2 Literal Elimination

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．2 つの節 $C_1 = \{x, a, b\}$，$C_2 = \{\overline{x}, a\}$ が F に含まれているとする．2 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．$\overline{x}$ を除く C_2 に含まれるリテラルは，x を除く C_1 のリテラルのサブセットとなっている．この場合，下式より C_1 を $\{a, b\}$ と置換することができ，C_1 に含まれていた x を削除することができる．

$$\begin{aligned} C_1 \wedge C_2 &\equiv (x \vee a \vee b) \wedge (\overline{x} \vee a) \\ &\equiv (\overline{x} \wedge a) \vee a \vee (\overline{x} \wedge b) \vee (a \wedge b) \\ &\equiv (a \vee b) \wedge (\overline{x} \vee a) \\ &\equiv (a \vee b) \wedge C_2 \end{aligned}$$

3.3 Variable Elimination

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) unit propagation
(2) resolution
(3) variable elimination by substitution
(4) hyper-unary-resolution

3.3.1 Unit Propagation

$C = \{l\}$ のように，1 つのリテラル l（l は変数 x またはその否定 $\overline{x}$）のみを含む節を *unit clause* と呼び，unit clause が F に含まれていた場合，明らかに l を真と決定することができる．Unit clause が含まれていた場合，与えられた F から以下のようにして x を削除することができる．

(1) l を含むすべての節を削除する．
(2) $\overline{l}$ を含むすべての節から $\overline{l}$ を削除する．

たとえば論理式 $\{\overline{a}, b\} \wedge \{a\}$ において $\{a\}$ は unit clause であり，a は真となる．その結果新たな unit clause $\{b\}$ が生成される．このように unit clause を次々と生成し，変数を削除するアルゴリズムを *unit propagation* と呼ぶ．

3.3.2 Resolution

リテラル x とその否定（$\overline{x}$）を含む 2 つの節 $C_0 = \{x, a_0, a_1, \ldots, a_n\}$，$C_1 = \{\overline{x}, b_0, b_1, \ldots, b_n\}$ から *resolvent* $\{a_0, a_1, \ldots, a_n, b_0, b_1, \ldots b_n\}$ 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．$A = \{a_0, a_1, \ldots, a_n\}$ と $B = \{b_0, b_1, \ldots, b_n\}$ の真偽によって以下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 にかかわらず真となる．
(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 $C_0 \wedge C_1$ はともに偽である．

x と $\overline{x}$ が C_0 と C_1 にのみ含まれるとして，$C_0 \wedge C_1$ をその resolvent で置き換え，F を真とする変数の割当てを求める．このとき $A = \{a_0, a_1, \ldots, a_n\}$ と $B = \{b_0, b_1, \ldots, b_n\}$ の真偽値から，上記 (1) から (4) に従い F が x の真偽にかかわらず恒偽であるか (4)，F を真にするための x の真偽が分かる．

x と $\overline{x}$ を含む節がそれぞれ N_x, $N_{\overline{x}}$ 個含まれているとすると，N_x，$N_{\overline{x}}$ 個の節すべての組合せに対して 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，$\{x, a, b\}$，$\{\overline{x}, \overline{a}\}$ といった節が含まれていた場合，その resolvent は $\{a, b, \overline{a}\}$ 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．
(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．

3.3.3 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．回路から生成された F には，$\{x, \overline{a}, \overline{b}\}$，$\{\overline{x}, a\}$，$\{\overline{x}, b\}$ 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 を x についての definition，R を S に含まれる definition 以外の節の部分集合とする．

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

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

$$S' = S'' \cup G' \cup R'$$
$$S'' = \overset{1\otimes4}{\{a, c\}}, \overset{1\otimes5}{\{b, c\}}, \overset{2\otimes4}{\{a, d, e\}}, \overset{2\otimes5}{\{b, d, e\}}, \overset{3\otimes6}{\{\overline{a}, \overline{b}, \overline{f}\}}$$
$$G' = \overset{3\otimes4}{\{\overline{a}, \overline{b}, a\}}, \overset{3\otimes5}{\{\overline{a}, \overline{b}, b\}}$$
$$R' = \overset{1\otimes6}{\{c, \overline{f}\}}, \overset{2\otimes6}{\{d, e, \overline{f}\}}$$

ここで G' に含まれるすべての節は definition の定義よりつねに真となる．さらに R' に含まれるすべての節は S'' に含まれる節より導出することができる．すなわち x についての resolvent としては S'' のみ用いればよい [6]．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．

3.3.4 Hyper-unary-resolution

$S_h = \{\overline{a}, \overline{b}\}$，$\{\overline{x}, a\}$，$\{\overline{x}, b\}$ といった節が F に含まれているとする．変数 x を真とすると S_h は偽となり，x を偽とすると S_h 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．この処理を *hyper-unary-resolution* と呼ぶ．

4. FPGA への実装

図 1 に回路の概要図を示す．すべてのデータは 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 へのアクセスを減らすことができる．

Unit propagation unit では 1 つのリテラルのみを含む節を検出する．Inclusion check unit では 2 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 occurrence list の更新を行う．

4.1 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 list.
- 特定のリテラルを含む節に高速にアクセスするための literal 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とき 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．
- Unit propagation 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 2 に節を格納するためのデータ構造（literal table, occurrence list，clause list）を示す．Literal table は変数の番号により参照される．Literal 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 を含むすべて節へのアドレスを保持する．Clause table は各

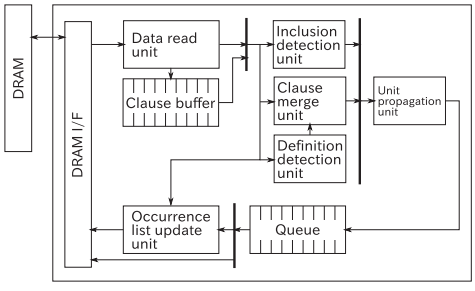

図 1 システム概要
Fig. 1 A block diagram of the FPGA system.

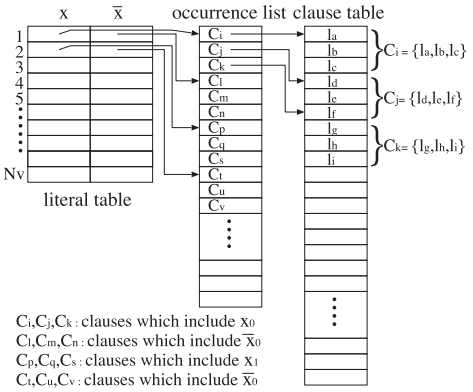

図 2 Literal table，occurrence list と clause table
Fig. 2 Literal table, occurrence list and clause table.

節の本体（各節に含まれるリテラル）を格納する．

外部メモリからリテラル x を含むすべての節 S_x を読み込む手順は以下のとおりである．

(1) Literal 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 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問題に依存するが，現在の実装では 440 個の block RAM を clause buffer に用いており，およそ 225 K 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 が，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で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が，ほぼすべての節をキャッシュすることができている．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 を含むすべての節を読み込む必要が生じる．このとき，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．

4.3 処理手順

本システムの処理手順はソフトウェアとほぼ同様である [6], [10]．本システムの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行う．

(1) 節の包含関係を判定する（subsumption と self-subsumption）．
(2) Resolution，variable elimination by substitution，hyper-unary-resolution を適用する．
(3) 変更が加えられなくなるまで (1) と (2) を繰り返す．

これらの処理中に unit clause が生成された場合，unit propagation 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，各変数に対するアルゴリズムの適用が並列に実行される．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．

4.4 節の包含関係の判定

$C_0 = \{a, b\}$ と $C_1 = \{a, b, c\}$ 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．$C_0 = \{x, a, b\}$ と $C_1 = \{\bar{x}, a, b, c\}$ 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（3 章）．Self-subsumption においてリテラル x と $\bar{x}$ 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．節どうし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．さらに節どうしの比較において，リテラルどうしの比較を並列に行うことができる．

表 2 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）の場合のみ適用したときの削減率を示す．表 2 において，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 2 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 2 L に対する削減率の変化
Table 2 The reduction ratio when L is changed.

	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

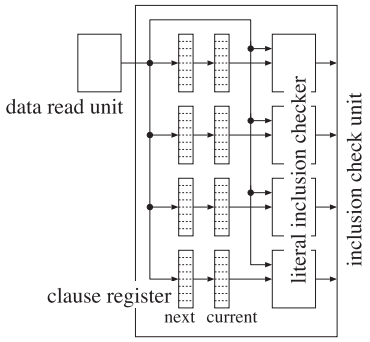

図 3 Inclusion check unit
Fig. 3 Inclusion check unit.

図 3 に inclusion check unit の詳細を示す．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 [10]．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．

4.5 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 $\overline{x}$ を含むすべての節の集合）に含まれる節の数 N_x と $N_{\overline{x}}$ が閾値 R_t（$N_x + N_{\overline{x}} \leqq R_t$）以下であり，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 3** に $R_t = 32$，64，128 の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 3 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 が適用できるか判定する．

表 3 R_t に対する削減率の変化
Table 3 The reduction ratio when R_t is changed.

	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

表 4 $|C_d|$ に対する削減率の変化
Table 4 The reduction ratio when $|C_d|$ is changed.

	$\|C_d\|$					
	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

Definition は $\{x, \overline{a}, \overline{b}\}$，$\{\overline{x}, a\}$，$\{\overline{x}, b\}$ といった節のセットであるため，長さが 2 の節に含まれるすべ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 $\{\overline{a}, \overline{b}\}$，$\{\overline{x}, a\}$，$\{\overline{x}, b\}$ 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り，definition detection unit を用いて definition の探索中に行われる．

$\{x, \overline{a}, \overline{b}\}$，$\{\overline{x}, a\}$，$\{\overline{x}, b\}$ といった definition が含まれていた場合，一番長い節 $C_d = \{x, \overline{a}, \overline{b}\}$ を definition detection unit に保持する．$\{\overline{a}, \overline{b}\}$，$\{\overline{x}, a\}$，$\{\overline{x}, b\}$ 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，次の変数へ移行する．

Definition の探索が終了し hyper-unary-resolution が適用されない場合，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，1 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 である．**表 4** に C_d の最大長が 3，8，16 の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 4 より C_d の最大長が 8 以

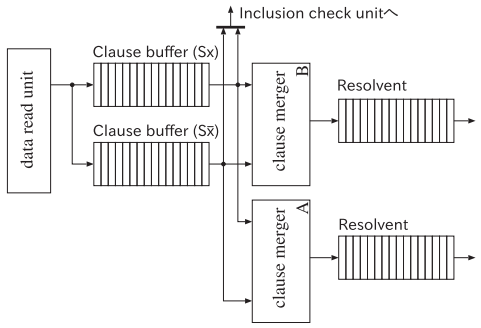

図 4 Resolution unit
Fig. 4 Resolution unit.

上であれば十分な削減率を達成することができることが分かる．

図 4 に resolution unit の詳細を示す．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，たとえば $C_1 = \{x, l_0, l_1, l_2\}$ と $C_2 = \{\overline{x}, l'_0, l'_1, l'_2\}$ から resolvent $C_1 \otimes C_2 = \{l_0, l_1, l_2, l'_0, l'_1, l'_2\}$ を生成する．このとき $l_0 = v$ かつ $l'_2 = \overline{v}$ である場合 $C_1 \otimes C_2$ はつねに真となり，消去することができる．本論文においては，

(1) 事前に各節のすべ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 1 つずつリテラルを変数番号順に取り出すことで，C_1 と C_2 の resolvent を生成する．

(3)（2）において，たとえば 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．

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．生成した resolvent は破棄される．

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（3 章）．これらをスキップするために，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 2 つの節がともに definition の一部であるか，ともに definition の一部ではない場合，次の resolvent の生成へ移行する．

4.6 本システムにおける削減率

表 5 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（たとえば，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 5 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（本システムでは処理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のは，以下の 2 つの理由によるものである．AProVE07-11 においては，(1) 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．(2) 表 5 の比較ではソフトウェア（SatELite）をデフォルトの状態で実行しているが，デフォルトの状態では，文献 [6] に記述されていないアルゴリズムも適用されている（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）．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文献 [6] 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．

表 5 本システムの削減率
Table 5 The reduction ratio by our FPGA system.

	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

プリプロセッサの本当の性能は，単なる削減率ではなく，ソルバと組み合わせたときの総処理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，1 つのソルバに限定しても，問題ごとに有効な適用順が異なると考えられるためである．

5. 性能評価

本論文では回路を Xilinx Kintex-7 XC7K325T に実装し

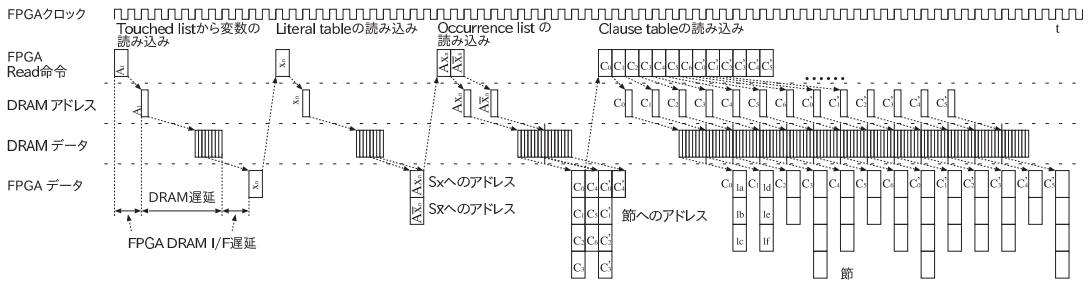

図 5 DRAM へのアクセスシーケンス
Fig. 5 Memory access sequence to DRAM banks.

た（ツールは Xilinx Vivado v2014.2 を使用した）．回路規模は 40 k FFs（利用可能な FF の 10%），53 k LUTs（利用可能な LUT の 26%），445 36 kb 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 3 の inclusion check 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 $N_x + N_{\bar{x}} \leqq 64$ の場合に適用し，clause merger を 2 個用いる．

現在の実装では，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 5 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 200 MHz で動作する．DRAM I/F と DRAM は 400 MHz で動作し，バースト転送は 800 MHz（400 MHz 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 5 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 5 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．この遅延を以下レイテンシと呼ぶ．シミュレーションではこのレイテンシを 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 literal table と occurrence list が読み込まれた後，clause table より節が読み込まれる．すべての節のリテラル数が十分に小さいとき，これらの節は clause buffer に格納され，その後は 1 クロックサイクルごとに節を読むことができる．

表 6 実行時間（秒）と高速化の度合い
Table 6 Execution time (sec) and the speedup.

	キャッシュ使用時		キャッシュ未使用時	
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

表 6 に本システムとソフトウェアの実行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 倍にとどまる．表 6 よりキャッシングを行わない場合，ソフトウェアに比べ実行時間が増える場合がある．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 $S_{\bar{x}}$ 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 $S_{\bar{x}}$ を $|S_L|/L$ 回読み込む）．

図 6 に inclusion check unit において同時に比較する節数を変更した場合の実行時間を示す．図 6 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実行時間の変化は小さくなることが分

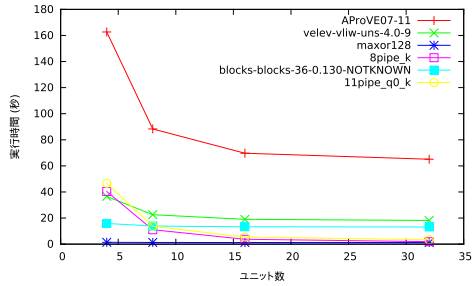

図 6 Inclusion detection unit の並列度と実行時間

Fig. 6 Performance when the number of the inclusion detection units is changed.

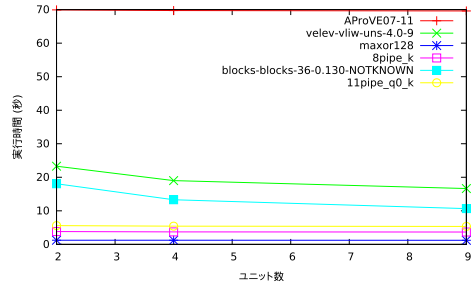

図 7 Resolution unit の並列度と実行時間

Fig. 7 Performance when the number of the resolution units is changed.

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 個のときの velev-vliw-uns-4.0-9 における処理の内訳を見ると，約 60%の処理時間がメモリアクセスに費やされている．これは，いったん読み込まれた節は 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 内での処理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 7 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 と変更した場合の実行時間を示す．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 にリテラルを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 7 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，新たに生成されたすべての 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 の並列度と処理時間を議論の場合と同様に，約 60%の処理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だと考えている．

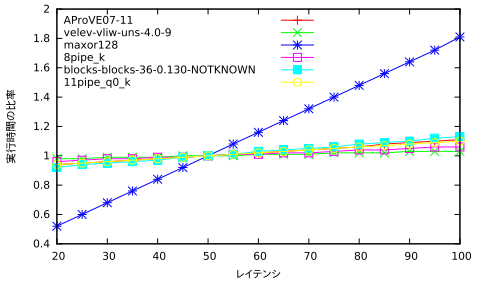

図 8 レイテンシと実行時間の変化

Fig. 8 Execution time when the memory access delay is changed.

図 8 にレイテンシを変更した場合の実行時間の変化を示す．図 8 では，レイテンシが 50 クロックサイクルであるときの実行時間を 1 としている．図 8 において，ベンチマーク maxor128 を除き実行時間の変化は 15%未満であり，本システムがレイテンシ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 の処理時間は，レイテンシの影響を受けている．**表 7** に，いく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，(1) 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，レイテンシの占める割合が相対的に大きくなり，また，(2) 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，相対的にレイテンシ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，レイテンシ

表 7 あるリテラル 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

Table 7 The average number of clauses that include literal l, and the total number of literals in those clauses.

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

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 7 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 7 の最大値に示したようにたかだか 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，ある程度のレイテンシの隠蔽が可能となると考えている．

6. おわりに

本論文において，プリプロセッサ SatELite の FPGA への実装方法と，FPGA 上の回路の並列度や DRAM のアクセス遅延を変更した場合の性能について述べた．DRAM を必要とする多くの大規模な問題に対して，遅延を隠蔽することによって，本システムは大きな高速化を行うことができることを示した．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，遅延を隠蔽できていない（図 8 における問題 maxor128）．これは，これらの問題においては S_x 中の節の数が少ないため，あるリテラル l の処理時間と，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 l の処理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，S_x の節数が少ないことを利用して，複数のリテラルに対する S_x 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 S_x 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．

また，今後 clause elimination と variable elimination のバランスをとるために，SAT ソルバを含めた性能の評価を行う必要がある．

参考文献

[1] SAT, available from ⟨http://www.satisfiability.org/⟩.

[2] The international SAT Competitions web page, available from ⟨http://www.satcompetition.org⟩.

[3] MiniSAT2, available from ⟨http://minisat.se/MiniSat.html⟩.

[4] Alfredsson, J. and Consulting, O.: The SAT Solver KW, *SAT 2009 Competition* (2009).

[5] Piette, C., Hamadi, Y. and Sais, L.: Vivifying propositional clausal formulae, *ECAI 2008* (2008).

[6] Eén, N. and Biere: Effective Preprocessing in SAT through Variable and Clause Elimination, *Proc. 8th Intl. Conf. on Theory and Applications of Satisfiability Testing* (*SAT'05*) (2005).

[7] Gulati, K., Paul, S., Khatri, S.P., Patil, S. and Jas, A.: FPGA-based hardware acceleration for Boolean satisfiability, *ACM Trans. Design Automation of Electronic Systems*, Vol.14, No.2 (2009).

[8] Kanazawa, K. and Maruyama, T.: An Approach for Solving Large SAT Problems on FPGA, *TRETS*, Vol.4, No.1, p.10 (2010).

[9] Kanazawa, K. and Maruyama, T.: An FPGA Solver for SAT-encoded Formal Verification Problems, *FPL 2011* (2011).

[10] M. Suzuki and Maruyama, T.: Variable and clause elimination in SAT problems using an FPGA, *FPT 2011*, pp.1–8 (2011).

鈴木 将之 （学生会員）

平成 24 年筑波大学大学院システム情報工学研究科博士課程前期修了．現在，同研究科博士課程後期在学中．主としてハードウェアを用いた組合せ問題の高速計算の研究に従事．

丸山 勉 （正会員）

昭和 62 年東京大学大学院工学系研究科情報工学専門課程博士課程修了（工博）．平成 9 年より筑波大学．主として書き換え可能なハードウェアを用いた高速計算システムの研究に従事．