PAPER

An Approach for Solving SAT/MaxSAT-Encoded Formal Verification Problems on FPGA

Kenji KANAZAWA[†a)] ***and*** **Tsutomu MARUYAMA**[††b)]**,** ***Members***

SUMMARY WalkSAT (WSAT) is one of the best performing stochastic local search algorithms for the Boolean Satisfiability (SAT) and the Maximum Boolean Satisfiability (MaxSAT). WSAT is very suitable for hardware acceleration because of its high inherent parallelism. Formal verification of digital circuits is one of the most important applications of SAT and MaxSAT. Structural knowledge such as logic gates and their dependencies can be derived from SAT/MaxSAT instances generated from formal verification of digital circuits. Such that knowledge is useful to solve these instances efficiently. In this paper, we first discuss a heuristic to utilize the structural knowledge for solving these problems by using WSAT. Then, we show its implementation on FPGA. The problem size of the formal verification is typically very large, and most data have to be placed in off-chip DRAMs. In this situation, the acceleration by FPGA is limited by the throughput and access latency of the DRAMs. In our implementation, data are carefully mapped on the on-chip memory banks and off-chip DRAMs so that most data in the off-chip DRAMs can be continuously accessed using burst-read. Furthermore, a variable-way cache memory comprised of the on-chip memory banks is used in order to hide the DRAM access latency by caching the head portion of the continuous read from the DRAMs and giving them to the circuit till the rest portion is started to be given by the burst-read. We evaluate the performance of our proposed method by changing configuration of the variable-way cache and the processing parallelism, and discuss how much acceleration can be achieved.

key words: *FPGA, SAT, MaxSAT, WalkSAT*

1. Introduction

Given a set of variables and a set of clauses that are disjunctions of the variables and their negations, the goal of the Boolean satisfiability (SAT) problem is to find a truth assignment to the variables in order to satisfy all clauses. The Maximum satisfiability (MaxSAT) problem is a variant of the SAT problem, and its goal is to find a truth assignment that satisfies as many clauses as possible. Many real-world applications can be encoded as SAT or MaxSAT problems. Formal verification, which is one of the most important applications of the SAT problem, is a mathematical method to verify the correctness of digital circuit systems.

WalkSAT (WSAT) and its variants [1], [2] are one of the best performing Stochastic Local Search (SLS) algorithms for SAT and MaxSAT problems. These algorithms begin by considering a random truth assignment and a set of unsatisfied clauses with the assignment. Then, an unsatisfied clause is chosen, and one of its literals is flipped from false to true (a literal is a variable or its negation, and the variable becomes false by this flipping when the literal is a negation of the variable) to satisfy the clause. Clauses that include the negation of the flipped literal are re-evaluated to update the set of unsatisfied clauses. This procedure is repeated until all clauses are satisfied. The heuristic to choose an unsatisfied clause and a literal to be flipped in it is the most critical part of the algorithm. By choosing proper literals, larger problems can be solved efficiently in shorter time. A problem of SAT and MaxSAT is typically given as a propositional conjunctive normal form (CNF), and it is known that the functional dependencies among the clauses (namely, gates in the circuit) can be reconstructed from the given CNF [7], [8], and it helps to solve those problems efficiently [18].

In [22], [23], we proposed a new heuristic for WSAT algorithm that utilizes the structural knowledge in a given CNF, and evaluated its performance using SAT-encoded formal verification problems of digital circuits. The size of formal verification problems is very large, and most of the data have to be placed in off-chip DRAMs. Under this situation, the performance of the implementation is limited by the throughput and access latency of the DRAMs. In [22], we showed an implementation method of our heuristic leveraging the high data transfer rate of DRAMs, and studied how much speedup was possible using the memory throughput and memory access latency as parameters. In [23], we evaluated its performance on SoC in order to get rid of the hardware resource limitation of FPGA and verify the maximum performance of the proposed implementation.

In [24], we introduced a variable-way cache (V-cache) memory on FPGA in order to hide the DRAM access latency by using the limited amount of on-chip memories efficiently. The size of data blocks that are frequently fetched from the DRAMs considerably varies in the WSAT algorithm. When the block size is small enough, all data in the block are cached in V-cache using several entries on the same line according to the block size. When the block size exceeds the total amount of the entries on one line, only a head portion of the block is cached using all the cache entries on the line. In the former case, data in the cache memory are simply sent to the circuit. In the latter case, on the other hand, DRAM access to the rest of the block data is immediately started,

Manuscript received December 9, 2016.
Manuscript revised April 4, 2017.
Manuscript publicized May 12, 2017.

[†]The author is with the Division of Information Engineering, Faculty of Engineering, Information and Systems, University of Tsukuba, Tsukuba-shi, 305–8573 Japan.
[††]The author is with the Division of Intelligent Interaction Technologies, Faculty of Engineering, Information and Systems, University of Tsukuba, Tsukuba-shi, 305–8573 Japan.
a) E-mail: kanazawa@cs.tsukuba.ac.jp
b) E-mail: maruyama@darwin.esys.tsukuba.ac.jp

DOI: 10.1587/transinf.2016EDP7487

and until the first data from the DRAM arrives, the cached data is sent to the circuit.

In this paper, we first show that our heuristic is effective for MaxSAT problems as well as for SAT problems. Then, we describe its FPGA implementation in detail (mainly data mapping on on-chip and off-chip memory banks, and V-cache), and evaluate the performance gain by our implementation using more benchmark problems. We also discuss how much parallelism can be exploited under the resource limitation of current largest FPGAs available and how much acceleration can be achieved under this situation.

This paper is organized as follows. In Sect. 2, we define the SAT and MaxSAT problems, and we introduce the related work in Sect. 3. In Sect. 4, our heuristic is described and its performance is evaluated. Its FPGA implementation is described in Sect. 5 in detail. The performance of the FPGA implementation is shown in Sect. 6. Section 7 gives the conclusions and future work.

2. Satisfiability and Maximum Satisfiability Problems

The SAT problem is a well-known combinatorial problem. An *instance* of the problem can be defined by a given Boolean formula $F(x_1, x_2, \ldots, x_n)$, and the question is to determine if there exists an assignment of binary values to the *variables* $(x_1, x_2, \ldots, x_n)$ that makes the formula true. Typically, F is presented in conjunctive normal form (CNF), which is a conjunction of a number of *clauses*, where a clause is a disjunction of a number of *literals*. Each literal represents either a Boolean variable or its negation. For example, in the following formula in CNF that consists of four clauses, $\{x_1, x_2, x_3\} = \{1, 1, 0\}$ satisfies all clauses:

$F(x_1, x_2, x_3) =$
$(\neg x_1 \vee x_2) \wedge (\neg x_2 \vee \neg x_3) \wedge (x_1 \vee \neg x_2 \vee x_3) \wedge (x_1 \vee \neg x_3)$.

The MaxSAT problem, on the other hand, is an optimization variant of the SAT problem, whose goal is to find an assignment to the variables that maximizes the number of satisfied clauses (i.e. minimizes the number of unsatisfied clauses). For instance, in the following CNF formula, there exists no solution, but $\{x_1, x_2, x_3\} = \{1, 1, 0\}$ minimizes the number of unsatisfied clauses:

$F(x_1, x_2, x_3) = (x_1 \vee x_2) \wedge (\neg x_2 \vee \neg x_3) \wedge (\neg x_1) \wedge (x_1 \vee x_3)$.

3. Related Work

Many algorithms and hardware solvers have been proposed to date. Algorithms for solving the SAT problem can be divided into two major groups: *complete* and *incomplete*. The complete algorithms can always find a solution, or conclude that the problem is unsatisfiable. When no solutions can be found, the incomplete algorithms do not guarantee to find a solution. When a solution can not be found by those algorithms, it is impossible to determine whether the problem is unsatisfiable, or the algorithms could not find the solution. Nevertheless, these algorithms are of particular interest, because they are very effective on many large problems, and can be used to solve the MaxSAT problem.

```
input a CNF formula F, MAX-FLIPS and MAX-TRIES
begin
for i in 1 to MAX-TRIES
    T = randomly generated truth assignment
    for j in 1 to MAX-FLIPS
        if T satisfies F then return T
        c = a random unsatisfied clause
        v = a variable in c chosen by a heuristic
        T = T with v flipped
    end for
end for
    return "no satisfying assignment found"
end
```

Fig. 1 Procedure of WSAT algorithms

WSAT [1], [2] is one of the best performing incomplete algorithms. Figure 1 shows the basic procedure of WSAT algorithms. The procedure begins by considering a random truth assignment to the variables. It searches for a solution by repeatedly selecting an unsatisfied clause at random, and then employing some heuristics to select a variable in that clause to *flip* (change its truth value from true to false or vice-versa). The heuristic to choose a literal to be flipped is the key of the search, and determines the performance of the algorithm. Many heuristics have been proposed for this reason [1]–[4], [19], [20]. Among them, WSAT/SKC [1] is the basis of our proposed method. Here, we introduce the variable selection heuristic in WSAT/SKC:

> For each variable in the randomly selected unsatisfied clause, count the number of clauses that are true in the current truth assignment, but that would become false if the flip were made (called a `break-value`). If variables with break-value of 0 exist, pick any of them. If not, with probability p, pick any variable, otherwise pick a variable that gives the minimum break-value.

Several approaches to accelerate simple WSAT algorithms by hardware systems have been proposed [9]–[11]. However, the size of the problems which can be solved by those hardware solvers are very limited. This is because it is not easy to solve large real-world problems using only simple WSAT algorithms. Sophisticated SLS algorithms such as [3], [4], [19] have been proposed, but their control structures and the required memory access sequences are complex, and it is not easy to implement them in hardware. In [21], an FPGA solver for probSAT [20], one of the latest SLS algorithms, is proposed. In probSAT, selecting next flip variable is based only on probability distribution calculated by an elementary function. Using Xilinx XC7V690T, this FPGA solver achieves up to 99 times speedup over software running on Intel Core-i5 4670K with 32GB main memory. However, the performance of this solver to formal verification is not clear because benchmark problems used in the evaluation are not based on real-world applications, and their size are very small (up to 250 variables and 1065 clauses).

Several approaches based on complete algorithms have been proposed [12], [13]. The performance of recent

complete algorithms [14]–[16] has been significantly improved by introducing several techniques to prune the search space [14]–[17]. However, these techniques also require complicated control structure, which makes it difficult to handle large real-world problems in hardware solvers.

4. Outline of Our Algorithm

In this section, we describe our heuristic for the WSAT algorithm that is designed for formal verification problems of digital circuits.

4.1 Gates and Dependencies

Formal verification is a mathematical method of verifying hardware or software systems. In SAT/MaxSAT-encoded formal verification of hardware systems, a design of the hardware design (typically in gate-level) with its verification specification is translated to a CNF instance. Here, we describe several logic gates mainly used in CNF.

In the following discussion, we follow the terminology used in [7] and [18]. First, we consider the following CNF formula:

$(\neg x_1 \vee y) \wedge \ldots \wedge (\neg x_n \vee y) \wedge (x_1 \vee \ldots \vee x_n \vee \neg y)$

This formula becomes true, when y is true and at least one of $x_1, \ldots, x_n$ is true, or y is false and $x_1, \ldots, x_n$ are all false. Namely, this formula shows an OR gate $y = \vee(x_1, \ldots, x_n)$. In the same way, the following formula:

$(x_1 \vee \neg y) \wedge \ldots \wedge (x_n \vee \neg y) \wedge (\neg x_1 \vee \ldots \vee \neg x_n \vee y)$

means an AND gate $y = \wedge(x_1, \ldots, x_n)$. In the formulas above, y is an output variable of the gate, and $x_1, \ldots, x_n$ are input variables. Another gate which is commonly used is XOR $y = \oplus(x_1, x_2)$, and can be described as follows:

$(\neg y \vee \neg x_1 \vee \neg x_2) \wedge (y \vee x_1 \vee \neg x_2) \wedge (y \vee \neg x_1 \vee x_2) \wedge (\neg y \vee x_1 \vee x_2)$.

An XNOR gate $y = \Leftrightarrow(x_1, x_2)$ can be represented as follows:

$(y \vee x_1 \vee x_2) \wedge (\neg y \vee \neg x_1 \vee x_2) \wedge (\neg y \vee x_1 \vee \neg x_2) \wedge (y \vee \neg x_1 \vee \neg x_2)$

Here, note that it is not possible to determine the output variable from the clauses for XOR and XNOR.

Basically, by finding the sets of clauses corresponding to any of above ones, we can find gates in the CNF (more sophisticated approaches are necessary to detect more gates as described in [7], [8]). An output variable of a gate becomes an input variable of the next gate. In order to make the data dependencies among the gates clear, we define "internal gate", "external gate" and "independent variable". An internal gate is any gate that can be recognized as $\vee, \wedge, \Leftrightarrow, \oplus$, namely a set of clauses which corresponds to any of that described above. An external gate is a clause which is not the part of the internal gates. Any literals included in them is not an input signal to other gates. It is considered that these literals correspond to the output signals of the circuit [18]. Independent variable is a variable that is never found in internal and external gates. Namely, the independent variables can be considered to the input signals to the circuit.

Figure 2 shows an example of gates, and their data dependencies. External gates have no output variables to other parts in the given CNF, and the status of each external gate becomes an output of the circuit. In our implementation, AND/OR-type gates are detected first, and the input/output of XOR and XNOR gates are inferred from the input/output of AND and OR gates, so that no input/output conflicts happen among the gates using a simple back-tracking method (the input/output variables of XOR and XNOR gates cannot be known from only the clauses for the gate, though those of AND and OR can be known). Then, a variable which is not the output variable of any gate becomes an "independent" variable.

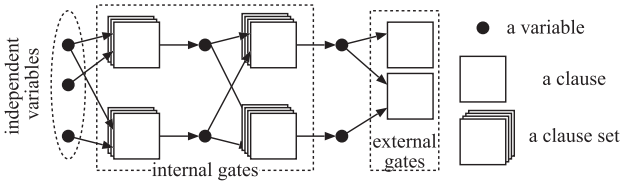

Fig. 2 Gates and their dependencies

4.2 Heuristic for Formal Verification Problems of Digital Circuits

Figure 3 shows the procedure of our algorithm [22]–[24]. In this procedure, the parameters *MAX-TRIES* (the number of new search sequences) and *MAX-FLIPS* (the maximum number of flips per try) are used to control the maximum run-time of the algorithm. Given a random truth assignment to the variables, several clauses become unsatisfied. This unsatisfiability of the clauses moves to the output side of the circuit (namely, to the external gates) by flipping the output variables (*forward search*). On the other hand, by flipping the input variables of the gates, the unsatisfiability moves to the input side (to the independent variables) (*backward search*). The scenario of our search is as follows:

1. Flipping the output variables of the gates preferentially (forward search).
2. All gates except for some external gates are satisfied.
3. Starting from the unsatisfied external gates, continue to flip one of the input variables of the gates until some of the independent variables are flipped (a series of clauses to one of the independent variables which make the external gate false are tracked) (backward search).
4. Then, repeat Steps 1 to 3.

According to our observations, by flipping the literals that correspond to the output signals of the gates preferentially, the search will converge very quickly to a local minimum. When the search is stuck in the local minimum, it is possible to get out of the minimum by flipping the input signals of the gates preferentially. Thus, by repeating these two phases, better local minima can be found efficiently. In Fig. 3, p decides the probability to choose the output signals to be flipped, which is automatically adjusted by a noise parameter tuning mechanism [3] considering the period of be-

Table 1 The Number of Instances in benchmark suite for which solutions could be found

	LIVENESS-SAT-1.0						VLIW-UNSAT-4.0				DLX-IQ-UNSAT-1.0	
	20/20	19/20 - 15/20	14/20 - 10/20	9/20 - 5/20	4/20 - 1/20	0/20	20/20	19/20 - 15/20	4/20 - 1/20	0/20	10/10	0/10
Ours	1	4	2	2	1	0	4	0	0	0	32	0
RSAPS	0	0	0	0	0	10	4	0	0	0	32	0
Sparrow	0	0	0	0	0	10	0	1	1	2	0	32
WSAT/SKC	0	0	0	0	0	10	0	0	3	1	0	32

```
WSAT-ex(F, MAX-TRIES, MAX-FLIPS)
  /* Preprocessing */
  Detect logic gates and their input/output signals();
  /* Main Procedure */
  for (i = 0; i < MAXTRIES; i++) {
    T = randomly generated truth assignment;
    for (j = 0; j < MAX-FLIPS; j++) {
      if T satisfies F return T;
      c = an unsatisfied clause selected at random;
      v = a variable in c chosen by the Heuristic(c);
      T = T with v flipped;
      Noise parameter tuning();
    }
  }
}
the Heuristic(c) {
  List = φ;
  for each literal l_i in c {
    count break-value;
    if (break-value = 0) List=List ∪ l_i;
  }
  if (List is not empty) {
    flip any of variables in List randomly;
  } else {
    with probability p, flip any variable randomly;
    with probability 1-p, flip the variable that
        corresponds to the output signal of c;
  }
}
Noise parameter tuning() {
  if (there is no improvement) {
    step = step + 1;
    if (step > N_c × θ) {
      p = p + (1 − p) × φ; step = 0;
    }
  } else {
    p = p − p × φ/2; step = 0;
  }
}
```

Fig. 3 Procedure in our algorithm

ing stuck in a local minimum.

4.3 Performance of the Proposed Heuristic

We compare the performance of our heuristic with four algorithms: (1) Sparrow [19], one of the latest SLS algorithms, (3) RSAPS [4], one of the best performing WSAT variants for real-world problems, and (4) original WSAT/SKC [1], using one satisfiable benchmark suite in [25] named LIVENESS-SAT-1.0 (10 benchmarks), and two unsatisfiable ones named VLIW-UNSAT-4.0 (4 benchmarks) and DLX-IQ-UNSAT-1.0 (32 benchmarks). These suites are from the formal verification of microprocessors. Each instance in LIVENESS-SAT-1.0 and VLIW-UNSAT-4.0 is evaluated 20 times, that in DLX-IQ-UNSAT-1.0 10 times, and each try is finished in 7200 seconds. All programs were executed on Intel Core i7-3770 3.4 GHz with 8 GB of main memory. The size of the benchmarks ranges from 96K variables and 1.8M clauses to 773K variables and 12.0M clauses.

Table 1 shows the number of instances in each benchmark suite for which correct solutions could be found. For example, '1' in '20/20' column in LIVENESS-SAT-1.0 means that the correct solutions could be found for one instance in all the 20 tries, and '4' in '19/20-15/20' means that the correct solution could be found for 4 instances in 15 to 19 tries out of the 20. For the two unsatisfiable suites, solutions which can not satisfy only one clause are considered as their correct solutions. As shown in this table, our heuristic is comparable to RSAPS in the MaxSAT problems and superior to all other heuristics in the SAT problems.

5. Hardware Architecture

In our system, first, (1) data arrays are generated from the given CNF, (2) data dependencies in it are analyzed, and (3) a random truth assignment for the variables and the initial set of unsatisfied clauses are generated on the host computer. Then, the data arrays, the random truth assignment and the set of unsatisfied clauses are downloaded to the FPGA, and the solution of the problems is searched on FPGA.

5.1 Data Arrays

Figure 4 shows tables used in our algorithm. Each entry of the clause table (`c_tbl[]`) points to the element in `c_arg_tbl[]`. c_arg_tbl[] holds the literals of clauses (called argument literals). `c_sat_tbl[]` holds the number of *true* literals in each clause. If its value is zero, it means that the clause is not satisfied with the current assignment (otherwise, it means that the clause is satisfied). `usc_buf[]` holds the clause numbers of the unsatisfied clauses. `ref_tbl[]` is a two-word width array, and both words point to the element in `c_list_tbl[]`. ref_tbl[] is accessed using a variable number, and the first word is used for the variable, and the other for its negation. c_list_tbl[] holds the list of clauses include a variable x (called `clause list(`x`)`), or its negation (called `clause list(`$\neg x$`)`), separately. `v_tbl[]` holds the truth values of the variables (1 for true, and 0 for false).

5.2 Processing Sequence

On the host computer,

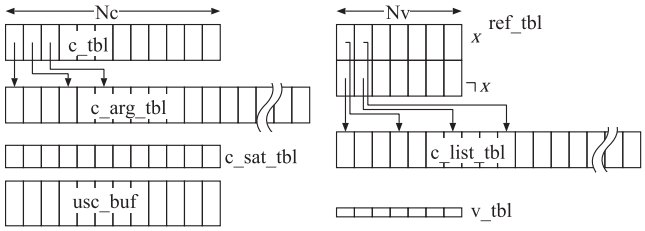

Fig. 4 Tables used in our algorithm

1. generate c_tbl[] and c_arg_tbl[] from a CNF file;
2. scan c_tbl[] and c_arg_tbl[], and
   a. find the output signals,
   b. make the clause list(x) and the clause list($\neg x$),
   c. store them in c_list_tbl[] and the references to them in ref_tbl[].
3. generate a truth assignment to variables at random;
4. evaluate all clauses using the truth assignment, and
   a. count the number of true literals in each clause, and store them in c_sat_tbl[],
   b. put the clause number of the unsatisfied clauses in usc_buf[];
5. download the tables to the circuit.

The circuit on the FPGA executes the following steps:

1. choose an unsatisfied clause c from usc_buf[] (if c is satisfied, discard it and choose another one);
2. read argument literals of c from c_arg_tbl[] using c_tbl[c];
3. $List \leftarrow \phi$;
4. for each literal l in c (here, all literals in c are false),
   a. read out clause list($\neg l$) from c_list_tbl[] using ref_tbl[$\neg l$] ($\neg l$ becomes false if l is flipped)
   b. count the number of clauses whose c_sat_tbl[] = 1 in clause list($\neg l$) (these clauses become false if l is flipped)
   c. if the number of these clauses is zero, add l to $List$
5. if $List$ is not empty, l_f = any of literals in $List$;
   otherwise,
   with probability p, l_f = any of literals in c;
   with probability $1 - p$, l_f = the literal that represents the output signal of c;
6. flip l_f (v_tbl[variable_number(l_f)] = $\neg$ v_tbl[variable_number(l_f)]);
7. read out clause list($\neg l_f$) from c_list_tbl[] using ref_tbl[$\neg l_f$], and decrement their c_sat_tbl[]. If c_sat_tbl[] becomes zero, add its clause number in usc_buf[];
8. read out clause list(l_f) from c_list_tbl[] using ref_tbl[l_f], and increment their c_sat_tbl[];
9. for the SAT problems, repeat Steps 1 − 8 until usc_buf[] becomes empty;
   for the MaxSAT problems, repeat them until reaching a solution with the target solution quality (the target number of unsatisfied clauses);

5.3 Parallelism in the Circuit

The parallelism in the algorithm is very high. However, throughput of off-chip DRAMs, namely, the number of words (L) that are given in parallel by the off-chip DRAM interface limits the circuit parallelism because most parts of the tables have to be placed in the off-chip DRAMs (in the following discussion, we call the L words given by the DRAM interface in parallel "L-words").

Suppose that the circuit on FPGA runs at f_c MHz. The databus of the DDR3-SDRAM operates at $4 \times f_c$ MHz and transfers the data with double data rate operation. Therefore, one DDR3-SDRAM bank of $32b$ word provides the data of $8 \times 32b$ to the circuit in parallel. Then, up to 32 words can be given to the circuit in parallel when it has four DRAM banks of $32b$ width (many FPGA boards e.g. Xilinx VC709 have the DRAM interface of the same configuration). In our circuit, most of the entries of the tables can be represented by up to $26b$, and it is reasonable to use a $32b$ word for each entry of the tables except for some special cases ($64b$ for usc_buf[], up to $18b$ for c_sat_tbl[] and $1b$ for v_tbl[]).

5.4 Data Mapping and System Architecture

Next, we describe the mapping of the tables onto on-chip and off-chip memory banks.

The width of v_tbl[] is 1b, and the access to it is random. With 64 on-chip memory banks (block RAMs) configured as $16\text{K} \times 1b$, we can store truth values of 1M variables. With this size of v_tbl[], we can process all the benchmark problems evaluated in Sect. 4.3.

c_sat_tbl[] is most frequently accessed, and the access to this table is also random. The size of this table is propotional to the number of clauses, and it is too large to map all of the entries onto on-chip memory banks. In order to reduce the size of c_sat_tbl[] and to map it onto on-chip memory banks, the following two methods are used:

I. For clauses with two and three argument literals (called `2-` and `3-arg clauses`), the set of the argument literals are used instead of clause numbers. For example, in c_list_tbl[], the set of argument literals in 2- and 3-arg clauses excluding literal l are placed in clause list(l) instead of the clause numbers (l does not appear in clause list(l) because it is obvious that all the clauses in clause list(l) contain l). The truth of those clauses are evaluated by the truth of the argument literals, and the truth of the argument literals in clause list(l) are obtained by referencing v_tbl[], not by c_sat_tbl[]. The truth of l is known in advance as mentioned in Sect. 5.2. With this method, no entries are prepared for 2- and 3-arg clauses in c_sat_tbl[].

II. For the clauses with more than three literals (called `k-arg clauses`), the clause numbers are used, and

Table 2 Statistical information of benchmark problems

	#args		length of clause list	
	2	3	average	max
LIVENESS	88.5 ~ 91.7%	3.5 ~ 6.0%	21.9~27.4	33161
VLIW	62.0 ~ 65.0%	30.9 ~ 34.5%	27.5~35.7	75980
DLX	87.5 ~ 93.4%	4.0 ~ 8.2%	19.0~34.2	19777

c_sat_tbl[] is used only for these clauses. c_sat_tbl[] has different width according to the number of literals in the clauses in order to implement c_sat_tbl[] with less block RAMs. For this approach, all clauses are sorted according to the number of their literals before giving them their clause numbers, and the data width of the first portion of c_sat_tbl[] for less number of arguments is narrower, and that of the last portion for more arguments is wider.

Table 2 shows some statistic information about the number of argument literals per clause and the length of the clause lists of our tested benchmark suites (their names are abbreviated). #args 2 and 3 show the percentage of the clauses with two and three literals. As shown in this table, the percentage of the clauses with two and three literals account for about 97% of all the clauses. By applying the two methods above, we can reduce the size of c_sat_tbl[] drastically and make c_sat_tbl[] small enough to implement on the on-chip memory banks.

The size of the other tables is very large, and these tables have to be placed in off-chip memory banks. Among them, the accesses to c_list_tbl[] can be continuous because the processing of a clause list is repeated for #args (the number of argument literals) + 2 times, and many elements are read from c_list_tbl[] continuously (19 to 36 on average, 75980 at a maximum) in each repetition. Thus, the burst-read of DRAMs can work efficiently for c_list_tbl[].

c_tbl[], c_arg_tbl[], and ref_tbl[] are also mapped on off-chip memory banks, and these tables are accessed at random. However, the overhead caused by the accesses to these tables are relatively small, because

1) for each access to ref_tbl[], c_list_tbl[] is also accessed following the access to ref_tbl[], and
2) for each access to c_arg_tbl[] using c_tbl[c], 1) is repeated for each argument literal in c_arg_tbl[c_tbl[c]], where c is an unsatisfied clause fetched from usc_buf[].

A small part of usc_buf[] is placed in the on-chip memory banks and is accessed preferentially than the other part placed in the off chip memory banks. When the usc_buf[] on on-chip memory banks is empty, a part of unsatisfied clauses are moved from usc_buf[] in off-chip memory banks to that in on-chip memory banks by single burst read access. For 2- and 3-arg clauses in usc_buf[], the set of the argument literals is used instead of the clause number. Unlike c_list_tbl[], all the argument literals are included in the set.

Figure 5 shows a block diagram of our circuit when we have four off-chip DDR3-SDRAM banks of 32b word. The circuit runs at 1/8 of the DRAM data transfer rate. By deciding how to map the tables into the DRAMs, the architecture

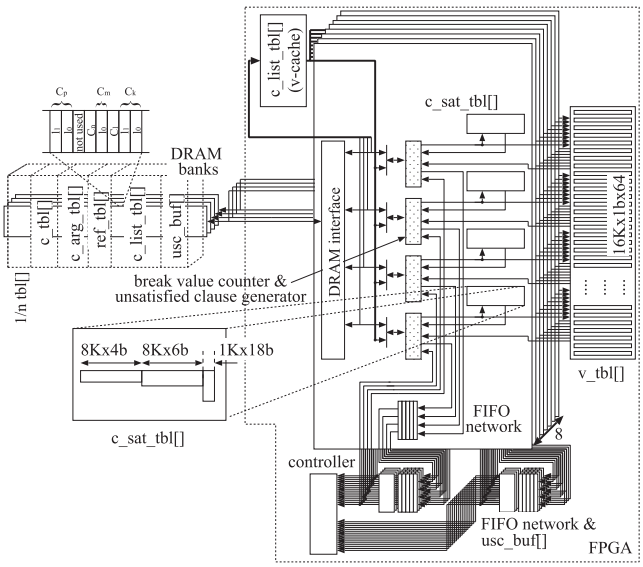

Fig. 5 System overview

of the circuit is almost fixed. In the mapping in Fig. 5, four DRAM banks work as one large bank, and 32(= 4×8) words are given from the DRAM interface to the circuit in parallel. As shown in Fig. 5, The width of c_sat_tbl[] is changed from 4b to 18b (4b, 6b and 18b for clauses with up to 15, 63 and more literals respectively) in order to use the block RAMs efficiently. In our tested benchmarks, the maximum number of k-arg clauses is 408055 (for iq54_a). By this variable width approach, entries for up to 544K k-arg clauses can be supported with 192 18Kb block RAMs, which is enough for our target FPGA.

On this architecture, up to 32 unsatisfied clauses may be generated at the same time (though much fewer on average). These unsatisfied clauses are first put in the FIFOs (depth = 32), and then moved to the usc_buf[] on on-chip memory banks (called cached usc_buf) via FIFOs connected as a baseline network. If the cached usc_buf[] are already fulfilled, all of its entries are flushed into the usc_buf[] on off-chip memory banks before moving the entries in the FIFO-network to the cached usc_buf[].

5.5 Parallel Evaluation of the Clauses

In our circuit, off-chip DRAMs are accessed as one larger memory bank for simplifying the memory access control, and up to L words are given to the circuit in parallel. As described above, the processing of a clause list is repeated for #args + 2 times, and occupies the most of the computation time. Therefore, the parallelism in our circuit mainly exists in the processing of the clause lists given from c_list_tbl[].

Each clause list consists of three parts: (1) 2-arg clauses, (2) 3-arg clauses, and (3) k-arg clauses. As described in Sect. 5.4, v_tbl[] is used for evaluating 2- and 3-arg clauses, while c_sat_tbl[] is used for evaluating k-arg clauses. These three types of clauses are arranged in each L-words in a clause list so that the maximum parallelism can

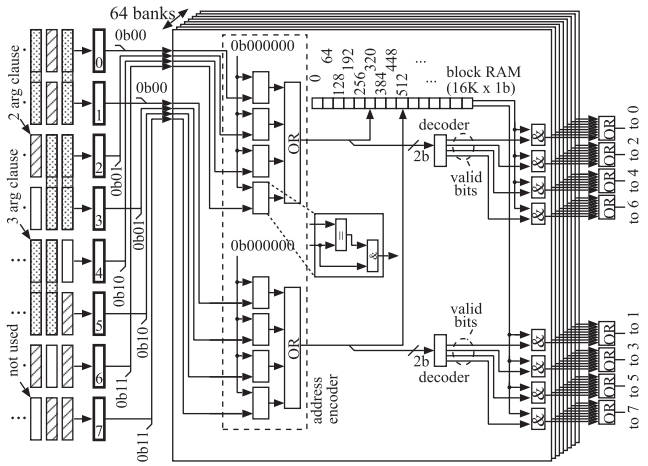

Fig. 6 v_tbl[]

be achieved. In c_list_tbl[], one word is required for representing one argument literal or one clause number. Therefore, 3-arg clauses have to be placed on double-word boundary, while 2- and k-arg clauses can be placed any slots in the L-words. Thus, up to $L/2$ clauses can be evaluated in parallel for 3-arg clauses, while up to L for 2- and k-arg clauses.

In order to evaluate argument literals in 2- or 3-arg clauses, we need to refer v_tbl[] to obtain the truth values of the variables used as the argument literals. Figure 6 shows the details of v_tbl[] ($L = 8$ for simplicity). v_tbl[] consists of 64 banks, each of which includes a block RAM, an address encoder and two decoders. The truth values of the variables are stored in one of the 64 block RAMs. To read the truth value of a variable, the bank number (to choose one of the block RAMs) and the address of the block RAM are required. In our implementation, the least-significant six bits of the variable number are used as the bank number (called `bank-index`), and the rest bits are used as the address of the block RAM (called `bank-address`).

Argument literals of 2- or 3-arg clauses in L-words given from c_list_tbl[] are divided into two groups according to their positions in L-words (*even* and *odd*). We call these groups `literal-group`s. The two argument literals of each 3-arg clause are always assigned to the different literal-groups, and the values of the two argument literals can be accessed at the same time utilizing the dual port access of the block RAMs. Clauses with argument literals for which the accesses to the same block RAM are required can not be assigned to the same L-words, and they are placed in different L-words to avoid the memory access conflict.

The variable numbers of the argument literals in the two literal-group are broadcasted to all of the banks along with their positions in the literal-groups (called `source-position`s) as shown in Fig. 6. Each bank has its own bank number (for example, 0b000000 is shown in Fig. 6), and in each bank, the bank-indexes of the broadcasted variables are compared with its bank number in the the address encoder. Through this comparison, the bank-indexes and source positions of the variables that do not match the bank number are masked to zero, and those in the same literal-group are ORed. Then, the bank-address of the ORed is used to access the block RAM. At the same time, the source position of the ORed is decoded. The truth value from the block RAM is masked by the decoder outputs. Its results (4 results for each literal-group in Fig. 5) from 64 block RAMs are ORed, and delivered back to the source positions. With this implementation, we can obtain up to L truth values in parallel from 64 block RAMs.

k-arg clauses are evaluated by reading out the number of their satisfied argument literals from c_sat_tbl[] as described in Sect. 5.2. k-arg clauses have to be placed in the fixed positions of L-words, because each entry of c_sat_tbl[] is placed in the fixed position.

The scheduling of these three kinds of clauses for avoiding bank conflict is executed on the host computer in advance. There exists no data dependency in this scheduling, and most of the L-words can be filled.

5.6 Variable-Way Cache Memory

As mentioned in the previous section, parallel processing of the clause lists is the main source of performance gain. It can be considered that $L = 32$ gives a good balance point of the performance and the hardware resource usage because the average length of the clause list is 19 to 36. However, for each access to the clause list, the idle time by DRAM access latency becomes the main factor that limits the system performance.

Let f be the operational frequency of the FPGA, the number of the cycles of row address to column address latency *RCD* (the cycles required between activation of a row and reading the first data from that row), its time T_{RCD}, the number of the cycles of CAS latency CL, its time T_{CL}, and the latency by DRAM interface T_{IF}. The total access latency is given by

$$T_d = T_{IF} + T_{RCD} + T_{CL} \simeq T_{IF} + 2 \times T_{CL}.$$

Here, note that T_{RCD} equals T_{CL}. The number of clock cycles in T_d is given by

$$C_d = T_{IF} \times f + 2 \times T_{CL} \times f = T_{IF} \times f + CL/2$$

because T_{CL} equals $CL/(4 \times f)$ (I/O bus clock frequency of DDR3-SDRAM is $4 \times f$). If we can hold the first $L \times C_d$ words of all clause lists on FPGA, we can completely hide the access latency to c_list_tbl[]. Here, let $L = 32$, $T_{IF} = 50$ ns, $f = 100$ MHz and $CL = 6$ (for DDR3-800). Then $32 \times (50 \times 10^{-9} \times (100 \times 10^6) + 6/2) = 256$ words have to be cached for each clause list to hide its access latency. The total number of the clause lists is $2 \times N_v$ (the number of variables), and it is impossible to cache all clause lists even with the current largest FPGAs.

Figure 7 shows a block diagram of a simple direct map cache memory for c_list_tbl[] (called F-cache). The F-cache consists of D blocks, each of which holds k lines. The data width of each line is L, and these L words can be read out

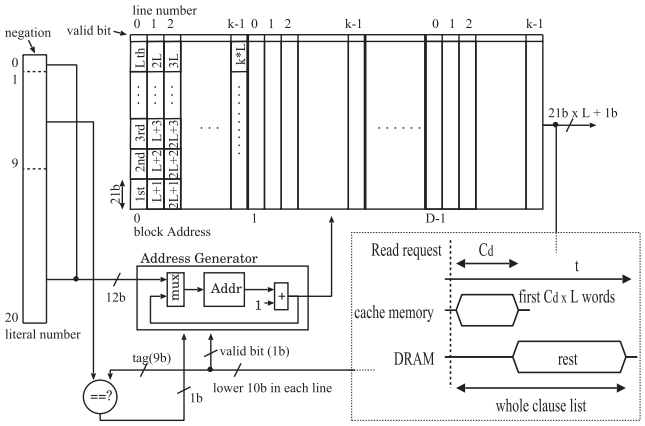

Fig. 7 F-cache (direct map)

Table 3 Clock cycles to read whole clause list

		1	≤ 16	≤ 32
SAT	bug1	92.2%	99.5%	99.8%
	bug7	91.1%	99.4%	99.7%
MaxSAT	iq3_C1	84.8%	99.7%	99.8%
	iq54_a	80.8%	99.3%	99.5%

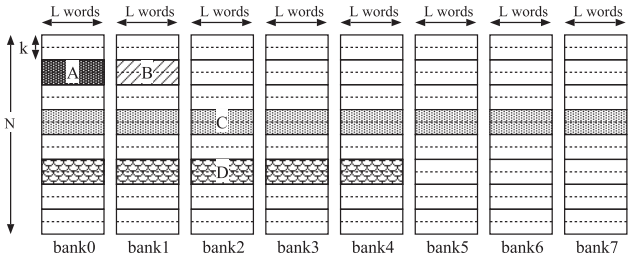

Fig. 8 Data mapping onto Variable-way cache (V-cache)

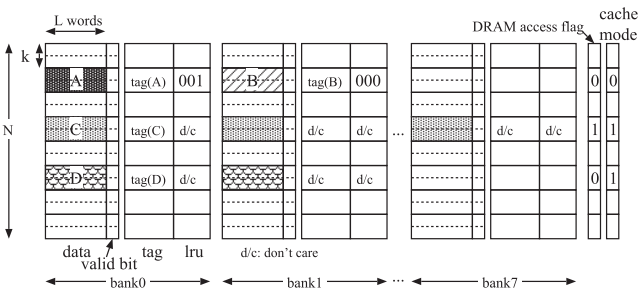

Fig. 9 Details of V-cache

in parallel. In Fig. 7, data width of one word is $21b$, which is wide enough to represent literal numbers and clause numbers for our target benchmarks. By holding C_d lines in each block ($k = C_d$), the access latency of DRAMs can be completely hidden as shown at the lower right corner in Fig. 7. To realize this data access sequence, first, the F-cache is looked up. If it hits, the clause list in the DRAMs are read from $(L \times C_d)$th words to read out the uncached part. Otherwise, it is read from the first to read out the whole clause list. The F-cache is looked up using $\log_2 D - 1$ bits of the variable number and its negation bit as the cache index, and the rest bits as the tag. The reason for using the negation bit as a part of the address is to avoid the cache conflict caused by x and $\neg x$. This F-cache can be easily extended to a set-associative cache.

This simple approach, however, does not work well. Table 3 shows the ratio of FPGA clock cycles to read the clause list excluding the access latency in four benchmarks (see Table 4 for their problem size). In Table 3, for the benchmark 'bug1', 92.2% of the clause lists can be given to the FPGA in only one FPGA clock cycle, and 99.5% of the clause lists can be given within 16 FPGA clock cycles. This means that most of the clause list can be read within the DRAM access latency. Furthermore, 80 to 90% of them can be stored in one line of the F-cache in Fig. 7, which means that the rest $k - 1$ lines in the block are wasted.

Figure 8 shows our approach (called V-cache) against this problem. In Fig. 8, the cache memory is constructed using 8 banks (8-way set associative). Each bank consists of N blocks, and each block has k lines of L-word width. Like F-cache, $\log_2 N - 1$ bits of the variable number are used as the cache index (block address), and rest bits as the tag. As shown in Fig. 8, if the clause list 'A' and 'B' are short enough, only one block of the V-cache is assigned to cache whole of them. In this case, the set of the blocks that store 'A' and 'B' and other blocks on the same block address works as 8-way set associative. If a clause list is very long, on the other hand, up to first $L \times C_d$ words of that have to be cached. When the clause list is longer than the size of a block (like 'C' and 'D'), all blocks of on the same block address are used to cache it, and this set of the blocks works as direct map.

In V-cache, first, the target clause list is looked up, and if cache hits, a flag in the cache block (DRAM access flag shown in Fig. 9) is checked to decide whether to start the DRAM access or not. When the whole clause list is cached, the DRAM access is not started, and otherwise, the DRAM access is started to read the uncached part.

5.6.1 Cache Block Replacement

Figure 9 shows the detail structure of V-cache (the maximum associativity = 8). In Fig. 9, 'data' indicates the data caching field. 'cache mode' represents whether the cache blocks are used as direct map or set associative. 'lru' contains the usage history of the corresponding data. All of the lrus are initialized by zero. 'valid bit' represents if the cached data on the corresponding line is valid or not.

The behavior of the cache replacement depends on the cache mode. In direct map, blocks for the same index are simply overwritten by the new cached data (before caching the new data, all of the valid bits for the same index are set to zero in order to invalidate the old data. Whenever changing the cache mode, all of the valid bits for the same index are set to zero before caching new data). In set associative, on the other hand, the cache replacement is executed in LRU policy. When a read or write access to a block is occurred, lrus whose values are smaller than that

for the read/written block are incremented, and that for the read/written one is set to zero. Hence, it follows that the block with the biggest value of lru is the least-recent-used among those for the same index. The maximum value of lru is at most $N_a - 1$, where N_a is the maximum associativity. Therefore, data width of lru is $\log_2 N_a$. When the replacement of the block becomes necessary, the block with the biggest value of lru is selected. Then, the lru for the replaced block is set to zero, and these for other blocks are incremented.

6. Performance Evaluation

The circuit size for $L = 32$ without F- or V-cache is 90K LUTs, 42K flipflops (FFs) and 303 18Kb block RAMs. These block RAMs are used mainly for v_tbl[], c_sat_tbl[], and usc_buf[]. The size of the control logic for the cache is 17K LUTs, 2.6K FFs and 54 18Kb block RAMs (mainly used for the tags and the lrus) when its maximum associativity, k and L are 16, 2 and 32, respectively. In total, 106.5K LUTs, 45K FFs and 387 block RAMs are required, which is feasible for all of the devices in Xilinx Virtex-7 series FPGA. The size for the data field of F- or V-cache are limited by the amount of the block RAMs on FPGA. XC7V1140T, the largest FPGA in Virtex-7 series, has 3760 18Kb block RAMs. Therefore, more than 3000 18Kb block RAMs can be still utilized. In the following evaluation, 3072 18Kb block RAMs are used for the data field in F- or V-cache.

A Hardware simulator is used for the following performance evaluation in order to facilitate changing the configuration of F-/V-cache and the circuit parallelism. This simulator simulates the hardware in logic circuit level and counts the number of the FPGA clock cycles. It also counts the number of the access to the off-chip DRAMs, calculates the total amount of the DRAM access latency and converts that to the number of the FPGA clock cycles. In the simulation, four DRAM banks of 32b word are used as one bank as described in Sect. 5. We consider the DRAM throughput to be that of DDR3-2133 (11-11-11) which is the fastest speed grade in JEDEC's standard. Each DRAM has internal memory banks in it, and data with continuous addresses can be read out by burst-read. The operational frequency of the FPGA is 1/8 of the memory data transfer rate of the DRAMs. DRAM interface latency is assumed to be 100 nsec.

6.1 Performance Comparison Over Software

First, we evaluate the system performance over software using the 20 benchmarks used in Sect. 4.3. In this experiment, L, k and the maximum associativity in V-cache are fixed to 32, 2 and 16, respectively based on the experiment described in the next subsection.

Table 4 shows the results of this evaluation (the averages of 20 runs). N_v and N_c are the number of variables and clauses respectively. 'ratio' shows the ratio that the correct solutions could be found in 20 runs with 2^{27} flips at a maximum. Two ratios on each line should be same because the same algorithm is used, but they are different. Because they depend on random numbers. #flips is the number of flips required to find the solutions. t_{ave} is the average of the total execution time (the processing time on the host computer and the time for data downloading are included), and tf_{ave} is the average execution time per flip. X_f shows the speedup over our algorithm on CPU (Intel Core i7-3770 3.4 GHz with 8GB main memory) per flip, and X_t shows the speedup of total execution time (X_t does not necessarily give the true performance because it depends on random numbers). The execution time of our algorithm on CPU is used as the base

Table 4 Performance gain over CPU

instance		N_v	N_c	Our algorithm on Core i7-3770 3.4 GHz				FPGA							
				ratio (%)	#flips(M)	t_{ave} (sec)	tf_{ave} (usec/flip)	ratio (%)	#flips(M)	no cache		16-way (V-Cache)			hdn (%)
										X_t	X_f	X_t	X_f	X_{VC}	
SAT	bug1	171648	2614464	85	22.5	301.8	13.4	60	5.68	10.2	2.58	12.8	3.24	1.26	58.8
	bug2	196655	3068742	65	35.2	574.01	16.3	80	14.5	6.87	2.82	8.44	3.47	1.23	52.1
	bug3	224920	3596474	80	15.8	298.5	18.9	85	4.19	11.2	2.98	13.6	3.59	1.20	54.7
	bug4	256697	4205986	75	5.35	141.1	26.4	85	18.1	1.18	3.99	1.41	4.79	1.20	49.7
	bug5	292249	4906188	70	9.48	304.5	32.1	85	18.8	2.45	4.86	2.90	5.74	1.18	47.0
	bug6	331848	5706594	75	13.5	482.9	35.8	55	10.1	6.40	4.78	7.48	5.59	1.17	48.4
	bug7	375775	6617342	55	7.83	377.2	48.2	70	34.1	1.54	6.72	1.78	7.75	1.15	44.0
	bug8	424320	7649214	45	16.7	1084.2	57.4	45	25.8	5.44	7.63	5.75	8.07	1.06	40.6
	bug9	477782	8813656	20	30.0	2048.7	68.4	5	74.6	2.54	6.32	2.82	7.01	1.11	32.8
	bug10	536469	10122798	40	59.2	5067.6	85.6	5	29.6	13.8	6.87	15.0	7.50	1.09	30.2
MaxSAT	9vliw_C1	96177	1814189	100	0.0724	2.11	29.2	100	0.066	1.25	1.14	1.27	1.16	1.02	46.5
	iq3_C1	333336	8122058	100	0.358	53.5	149.4	100	0.309	4.08	3.52	4.14	3.57	1.01	39.1
	iq33_a	143519	1877765	100	0.555	3.55	6.39	100	0.338	1.96	1.87	2.18	2.08	1.11	56.2
	iq37_a	245646	4568332	100	0.799	17.4	21.8	100	0.487	4.35	2.94	4.64	3.14	1.07	50.4
	iq48_a	365189	5250970	100	1.54	20.7	13.5	100	0.836	3.86	2.72	4.17	2.93	1.08	49.3
	iq53_a	471429	6952455	100	2.02	32.3	16.0	100	1.12	4.33	2.84	4.66	3.06	1.08	47.2
	iq54_a	663574	15489863	100	2.24	113.45	50.6	100	1.24	7.59	4.33	7.89	4.50	1.04	43.4
	iq59_a	623700	9457051	100	2.73	53.2	19.5	100	1.48	5.06	3.00	5.43	3.22	1.07	45.0
	iq63_a	720715	11606548	100	3.53	110.7	31.4	100	1.87	7.44	4.69	8.04	5.07	1.08	48.2
	iq64_a	773005	11974186	100	3.34	72.1	21.6	100	1.78	5.45	3.07	5.81	3.27	1.07	43.7

Table 5 Speedup with variable-way cache over CPU

		fixed-way cache (F-cache)												variable-way cache (V-cache)											
		4-way			8-way			16-way			32-way			4-way			8-way			16-way			32-way		
	k	hit	hdn	X_f	hit	hdn	X_f	hit	hdn	X_f	hit	hdn	X_f	hit	hdn	X_f	hit	hdn	X_f	hit	hdn	X_f	hit	hdn	X_f
bug1	1	86.2	30.3	2.88	86.9	30.6	2.88	87.2	30.8	2.99	87.5	30.9	2.99	74.7	44.7	3.05	70.2	46.6	3.07	65.5	50.6	3.12	60.5	54.3	3.17
	2	78.1	40.6	3.00	78.7	41.0	3.00	79.0	41.2	3.01	79.1	41.3	3.03	71.8	48.1	3.09	69.0	53.8	**3.17**	65.5	58.8	**3.24**	62.0	56.3	**3.19**
	8	63.0	39.1	2.98	63.5	39.5	2.99	63.7	39.6	2.99	63.7	39.6	2.99	56.8	50.9	**3.13**	54.3	49.3	3.11	51.7	46.8	3.08	48.9	44.3	3.04
	32	48.4	43.3	**3.03**	48.8	43.6	**3.03**	48.9	43.7	**3.04**	48.9	43.7	**3.04**	44.8	40.6	3.00	43.2	39.2	2.98	41.5	37.6	2.96	39.7	36.0	2.94
	64	42.3	38.3	2.97	42.6	38.7	2.98	42.8	38.8	2.98	42.8	38.8	2.98	39.7	35.9	2.94	38.7	35.1	2.93	37.5	34.0	2.92	36.6	33.2	2.91
bug7	1	70.3	20.3	7.16	70.7	20.5	7.17	71.0	20.6	7.17	71.1	20.6	7.17	58.9	31.4	7.43	55.0	33.0	7.47	51.5	37.3	7.58	48.0	42.8	**7.72**
	2	61.1	26.4	7.31	61.5	26.6	7.31	61.7	26.7	7.31	61.7	26.7	7.31	54.7	32.7	7.46	52.1	37.9	**7.59**	49.4	44.0	**7.75**	46.6	42.3	7.71
	8	47.8	25.4	7.28	48.1	25.5	7.29	48.2	25.6	7.29	48.3	25.6	7.29	41.8	37.2	**7.58**	39.7	36.0	7.54	37.2	33.7	7.49	34.6	31.3	7.43
	32	38.2	34.0	**7.49**	38.5	34.2	**7.50**	38.7	34.3	**7.50**	38.7	34.4	**7.50**	34.1	30.9	7.42	32.4	29.3	7.38	30.4	27.6	7.33	28.4	25.8	7.29
	64	34.4	31.2	7.42	34.7	31.5	7.43	34.9	31.6	7.43	34.9	31.7	7.44	31.4	28.5	7.36	30.2	27.4	7.33	29.0	26.3	7.30	27.9	25.3	7.28
iq3_C1	1	53.9	16.1	3.54	53.9	16.1	3.54	53.8	16.0	3.54	53.8	16.0	3.54	48.9	24.0	3.55	48.2	27.5	3.56	47.1	32.7	3.56	46.0	40.6	**3.57**
	2	48.7	19.6	3.55	48.8	19.6	3.55	48.8	19.6	3.55	48.8	19.5	3.55	46.2	25.7	3.55	45.3	31.0	3.56	44.3	39.1	**3.57**	42.9	38.9	**3.57**
	8	43.5	22.9	3.55	43.6	23.0	3.55	43.7	23.1	3.55	43.8	23.1	3.55	41.5	36.6	**3.57**	40.4	36.7	**3.57**	38.2	34.6	**3.57**	35.9	32.5	3.56
	32	39.7	34.9	**3.57**	39.9	35.1	**3.57**	40.0	35.2	**3.57**	40.0	35.2	**3.57**	36.2	32.9	3.56	34.4	31.1	3.56	30.4	27.6	3.56	28.3	25.7	3.55
	64	37.8	34.2	**3.57**	37.9	34.3	**3.57**	38.0	34.5	**3.57**	38.1	34.5	**3.57**	33.4	30.2	3.56	29.6	26.9	3.56	27.7	25.1	3.55	25.8	23.4	3.55
iq54_a	1	73.0	21.8	4.41	73.6	22.0	4.41	73.9	22.2	4.41	74.1	22.2	4.41	60.0	31.3	4.45	55.7	35.3	4.47	51.7	38.6	4.48	48.4	43.1	**4.50**
	2	64.7	26.3	4.43	65.9	26.5	4.43	66.5	26.6	4.43	66.8	26.7	4.43	55.1	34.8	4.46	51.7	38.6	**4.48**	48.7	43.4	**4.50**	46.3	42.0	4.49
	8	52.7	31.4	4.45	53.1	31.6	4.45	53.3	31.6	4.45	53.4	31.7	4.45	45.6	40.7	**4.49**	43.9	39.8	**4.48**	42.2	38.2	4.48	39.7	36.0	4.47
	32	42.7	37.9	**4.48**	42.7	38.0	**4.48**	42.8	38.0	**4.48**	42.8	38.0	**4.48**	38.6	35.0	4.47	36.3	32.9	4.46	34.4	31.2	4.45	33.7	30.1	4.45
	64	46.7	35.7	4.47	46.8	35.7	4.47	46.9	35.8	4.47	46.9	35.8	4.47	35.2	32.0	4.45	33.4	30.2	4.45	33.2	30.1	4.44	33.0	30.0	4.44

of this comparison, because it finds the solutions faster and more frequently than the other WSAT variants as described in Sect. 4.3. X_{VC} is the speedup by V-cache (the ratio of X_f with no cache and with V-cache). Hdn is the ratio that the idle time could be hidden by the V-cache. In these values, those of the runs that failed to find the solutions are not included.

As shown in Table 4, our system with V-cache achieves 1.16 to 8.07 times of speedup over CPU in the execution time per flip, and the enhancement by V-cache is up to 26% (X_{VC}). The two lowest speedup, 1.16 and 2.08 are given by the two smallest benchmarks, and the speedup for other problems is 3 to 8 approximately, which is fast enough considering that the performance is limited by DRAM throughput. The idle time hidden by V-cache is about 50%. This is not so high, but it is reasonable considering the size ratio of V-cache and c_list_tbl[].

For bug9 and bug10, 'ratio's seem to be inferior in our FPGA solver than those in the software. We are now trying to make the reason clear, but it probably comes from the selection method of an unsatisfied clause and the accuracy of the random number in our solver, and the problem size. When selecting an unsatisfied clauses, our FPGA solver selects from usc_buf[] on the block RAMs, and gradually moves unsatisfied clauses from the DRAMs when usc_buf[] on the block RAMs becomes empty. This may cause unfair selection of an unsatisfied clause. In addition, the lower accuracy of the random number generation may cause the degradation (our solver uses a linear congruential generator whereas the software solver uses Mersenne twister). When the problem size becomes larger, the influence of these factors may not be able to negligible.

6.2 Performance by Changing V-Cache Configuration

For deciding the optimal configuration of V-cache (k and the associativity), we have evaluated the performance of F-cache and V-cache using four benchmarks used in Table 2. Table 5 shows the cache hit ratio (hit), ratio of the idle time that could be hidden (hdn), and the performance gain (X_f, which is same as X_f in Table 4). The values in Table 5 are the average of 20 runs with 2^{27} flips at a maximum.

The left half of Table 5 shows the results of the F-cache. The hit ratio decreases as k is increased (this is because the number of entries in the cache decreases as k is increased). The performance, however, becomes better as k is increased. The peak (shown by bold font) is given when $k = 32$ in most cases, and it does not depend on the associativity. In the F-cache, for example, when $k = 1$, only one line (up to 32 words) of a clause list can be cached. This works well for short clause lists that can be stored in one line, but does not work well for long clause lists, because their access latency can be hidden by only one clock cycle. When $k = 32$, the access latency is hidden by 32 clock cycles, and it showed the best balance of the cache hit ratio and access latency hiding.

The right half of Table 5 shows the results of V-cache. Larger k shows lower hit ratio as with F-cache. Unlike F-cache, higher associativity also brings lower hit ratio. In the V-cache, several entries on the same horizontal line in the cache memory are used to cache a long clause list as shown in Fig. 8. With higher associativity, more entries are used to cache a longer clause list, and all short clause lists that have already been cached in these entries are wiped out. This is the reason of lower hit ratio for higher associativity. However, the lower hit ratio does not mean the lower performance gain as shown in the table (for the same value of k).

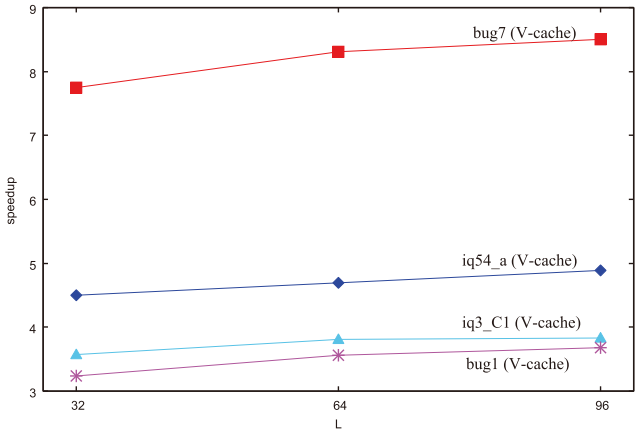

Fig. 10 Performance gain by enhancing parallelism

The best performance is given by V-cache when $k = 2$ and the associativity is 16. In this case, 32 lines ($32 \times L$ words) can be cached for each long clause list, and it is enough to hide the DRAM access latency.

The interesting point here is that in both F- and V-cache, the maximum number of lines to be cached to achieve the highest performance is the same (32 in this experiment). In this situation, hit ratio and hdn of V-cache are always 5 to 10% higher than those of F-cache, and then the performance of V-cache always outperforms that of the F-cache. This indicates that V-cache can utilize the same space of caching more efficiently than F-cache.

6.3 Performance with More Parallelism

The latest FPGAs support more hardware resources than Virtex-7 series FPGAs. Therefore, we can increase the performance by the following two approaches: executing different problems in parallel by copying the circuit or accelerating the same problem by enlarging L. Here, we consider the second approach. Xilinx XCVU13P, the largest FPGA in Virtex Ultrascale+ series, has about three times of logic cells and six times of on-chip memory banks than XC7V1140T. Its number of High Performance I/Os (HP I/Os), which are required for high speed transfer between DDR3-SDRAMs and FPGA, are a bit less than that of XC7V1140T, but still large enough to support $L = 96$.

Figure 10 shows the speedup of the four benchmarks by changing L from 32 to 96. The improvement by enlarging L is not so high, because the average length of the clause list is at most 36 in the tested benchmarks as shown in Table 2. However, the performance continues to be improved by enlarging L except for iq3_C1. This is probably because the length of the frequently fetched clause lists is longer than the average. To make it clear, we need to analyze the length of the clause lists which are actually fetched in the search in detail. The performance improvement by using larger FPGA is not so high, but it can be effective when it is allowed to use larger FPGA.

7. Conclusions and Future Work

In this paper, we have shown an FPGA solver for large SAT/MaxSAT-encoded formal verification problems of digital circuits. To solve large real-world problems efficiently on FPGA, first, we have proposed a new heuristic for WSAT algorithms that is simple enough to realize it on a hardware circuit, and we have showed its implementation on the FPGA that uses off-chip DRAM banks to hold the main data. The performance gain by our system is approximately 3 to 8 times for large problems. This speedup is not so drastic, but it is fast enough when we consider that it is limited by the throughput and access latency of the off-chip DRAMs and that the search time on CPU that is sometimes longer than one hour can be reduced to 1/3 to 1/8. In this system, all the on-chip memory banks that are not used to store data arrays are used to configure the specialized cache memory for the search, and the performance can be improved up to 26%. We have also evaluated the system performance when the parallelism is enlarged in order to clarify how much speedup could be possible by using largest FPGA available.

It will be possible to improve the performance of our heuristics by analyzing the relation between the unsatisfied external gates and the values of the independent variables when the search is stuck, so that the search can get out from the local minima more easily. The search can be also improved by managing the history of the flipped variables, and by choosing the next variable to be flipped using the history. These improvements requires run-time analysis and management of the relation of the variables. To implement these improvements on hardware is our future work.

References

[1] B. Selman, H. Kautz, and B. Cohen, "Noise Strategies for Improving Local Search," AAAI-94, pp.337–343, 1994.
[2] D. McAllester, B. Selman, and H. Kautz, "Evidence for invariants in local search," AAAI-97, pp.321–326, 1997.
[3] H.H. Hoos, "An adaptive noise mechanism for WalkSAT," AAAI-02, pp.655–660, 2002.
[4] F. Hutter, D.A.D. Tompkins, and H.H. Hoos, "Scaling and probabilistic smoothing: Efficient dynamic local search for SAT," CP-02, vol.2470, pp.233–248, 2002.
[5] C.M. Li, W. Wei, and H. Zhang, "Combining adaptive noise and look-ahead in local search for SAT," SAT-07, pp.121–133, 2007.
[6] A. Belov and Z. Stachniak, "Improving Variable Selection Process in Stochastic Local Search for Propositional Satisfiability," SAT-09, vol.5584, pp.258–264, 2009.
[7] R. Ostrowski, É. Grégoire, B. Mazure, and L. Sais, "Recovering and exploiting structural knowledge from CNF formulas," CP-02, vol.2470, pp.185–199, 2002.
[8] É. Grégoire, R. Ostrowski, B. Mazure, and L. Sais, "Automatic Extraction of Functional Dependencies," SAT-05, vol.3542, pp.122–132, 2005.
[9] P.H.W. Leong, C.W. Sham, W.C. Wong, H.Y. Wong, Y.S. Yuen, and M.P. Leong, "A Bitstream reconfigurable FPGA implementation of the WSAT algorithm," IEEE Transaction on Very Large Scale Integration Systems, vol.9. no.1, pp.197–201, 2001.
[10] R. Yap, S. Wang, and M. Henz, "Real-time Reconfigurable Hardware WSAT Variants," FPL-03, pp.488–496, 2003.

[11] K. Kanazawa and T. Maruyama, "An Approach for Solving Large SAT Problems on FPGA," Trans. ACM TRETS, vol.4, no.1, Article No.10, 2010.
[12] M. Safer, M.W. El-Kharashi, M. Shalan, and A. Salem, "A Reconfigurable, Pipelined, Conflict Directed Jumping Search SAT Solver," DATE-11, pp.1–6, 2011.
[13] T. Ivan and E.M. Aboulhamid, "An Efficient Hardware Implementation of a SAT Problem Solver on FPGA," DSD-13, pp.209–216, 2013.
[14] J.P. Marques-Silva and K. Sakallah, "GRASP: A search algorithm for propositional satisfiability," IEEE Transactions on Computers, vol.48, no.5, pp.506–521, 1999.
[15] M.W. Moskewicz, C.F. Madigan, Y. Zhao, L. Zhang, and S. Malik, "Chaff: Engineering an efficient SAT solver," DAC-01, pp.530–535
[16] N. Eén and N. Sörensson, "An extensible SAT-solver," SAT-04, vol.2919, pp.502–518, 2004.
[17] L. Zhang, C.F. Madigan, M.H. Moskewicz, and S. Malik, "Efficient conflict driven learning in a boolean satisfiability solver," ICCAD-01, pp.279–285, 2001.
[18] D.N. Pham, J. Thornton, and A. Sattar, "Building structure into local search for SAT," IJCAI-07, pp.2359–2364, 2007.
[19] A. Balint and A. Fröhlich, "Improving Stochastic Local Search for SAT with A New Probability Distribution," SAT-10, vol.6175, pp.10–15, 2010.
[20] A. Balint and U. Schöning, "Choosing Probability Distributions for Stochastic Local Search and the Role of Make Versus Break," SAT-2012, vol.7317, pp.16–29, 2012.
[21] A.A. Sohanghpurwala and P. Athanas, "An Effective Probability Distribution SAT Solver on Reconfigurable Hardware," ReConFig-16, pp.1–6, 2016.
[22] K. Kanazawa and T. Maruyama, "An FPGA Solver for SAT-encoded Formal Verification Problems," FPL-11, pp.38–43, 2011.
[23] K. Kanazawa and T. Maruyama, "Solving SAT-encoded Formal Verification Problems on SoC Based on a WSAT Algorithm with a New Heuristic for Hardware Acceleration," MCSoC-13, pp.101–106, 2013.
[24] K. Kanazawa and T. Maruyama, "FPGA Acceleration of SAT/Max-SAT Solving using Variable-way Cache," FPL-14, pp.1–4, 2014.
[25] http://www.miroslav-velev.com/sat_benchmarks.html

Kenji Kanazawa received his PhD from University of Tsukuba in 2012. He is currently an assistant professor at Faculty of Engineering, Information and Systems, University of Tsukuba. His research interests are reconfigurable accelerator and hardware algorithms for hard computation problems including combinatorial optimization.

Tsutomu Maruyama received his PhD from University of Tokyo in 1987. He is currently a professor at Faculty of Engineering, Information and Systems, University of Tsukuba. His research interest is reconfigurable accelerator for applications including image processing, bioinformatics and combinatorial problems.