One Flip per Clock Cycle

Martin Henz, Edgar Tan, and Roland Yap

School of Computing
National University of Singapore
Singapore
{henz,tanedgar,ryap}@comp.nus.edu.sg

Abstract. Stochastic Local Search (SLS) methods have proven to be successful for solving propositional satisfiability problems (SAT). In this paper, we show a hardware implementation of the greedy local search proceduce GSAT. With the use of field programmable gate arrays (FPGAs), our implementation achieves one flip per clock cycle by exploiting maximal parallelism and at the same time avoiding excessive hardware cost. Experimental evaluation of our prototype design shows a speedup of two orders of magnitude over optimized software implementations and at least one order of magnitude over existing hardware schemes. As far as we are aware, this is the fastest known implementation of GSAT. We also introduce a high level algorithmic notation which is convenient for describing the implementation of such algorithms in hardware, as well as an appropriate performance measure for SLS implementations in hardware.

1 Introduction

Local search has been used successfully for finding models for propositional satisfiability problems given in conjunctive normal form (CNF), after seminal work by Selman, Levesque, and Mitchell [SLM92] and Gu [Gu92]. A family of algorithms has been studied extensively over the last 10 years, all of which are instances of the algorithm scheme given in Program 1.

The algorithm repeatedly tries to turn an initial assignment of variables occurring in the given set of clauses *cnf* into a satisfying assignment by performing flips, which inverts the truth value of a chosen variable. The instances of GenSAT differ in their choice of `INIT_ASSIGN` and `CHOOSE_FLIP`. Note that `INIT_ASSIGN` and `CHOOSE_FLIP` are place-holders for code in the sense of macros, which will be explained later. In all instances of GenSAT, the concept of the score for a variable plays a crucial role. The function $score(i, cnf, V)$ returns the number of clauses in *cnf* that are satisfied by the assignment V modified by inverting the truth value of variable i. For simplicity of discussion, we concentrate on the most basic variant, GSAT [SLM92], where `INIT_ASSIGN` randomly assigns truth values to the components of V and `CHOOSE_FLIP` assigns to f a randomly chosen variable i that produces maximal $score(i, cnf, V)$. Variants of this algorithm, random walk [SKC94], history and tabu mechanisms [MSK97], are presented systematically in [HS00].

Program 1 The GenSAT Algorithm Family

procedure `GenSAT`(cnf, $maxtries$, $maxflips$)
    output: satisfying assignment satisfying cnf
    for $i = 1$ **to** $maxtries$ **do** /* *outer loop* */
        `INIT_ASSIGN`(V);
        for $j = 1$ **to** $maxflips$ **do** /* *inner loop* */
            if V satisfies cnf **then return** V
            else
                `CHOOSE_FLIP`(f);
                $V := V$ with variable f flipped;
end end end end

The speed of GSAT is determined by the cost of checking and flipping a variable. Its time complexity is $\mathcal{O}(maxtries\ maxflips\ m\ n)$, where m is the number of clauses and n is the number of variables. In this paper, our goal is to make this flipping step as fast as possible. Given the simplicity of the GSAT algorithm and that boolean formula can be directly represented as digital logic, the best way of meeting this objective is an implementation of GSAT in hardware. The advantage of hardware is of course speed and fine-grained parallelism which is to be balanced against the difficulty and complexity of realization in hardware. For maximum flexibility and ease of implementation, we use the Xilinx Virtex family of Field Programmable Gate Arrays (FPGAs).

The potential of FPGAs for solving SAT was realized by Hamadi and Merceron [HM97] and Yung, Seung, Lee and Leong [YSLL99]. Hamadi and Merceron describe an implementation of GSAT on FPGAs where the inner loop is done in hardware with n cycles per flip, hence the time complexity for GSAT is $\mathcal{O}(maxtries\ maxflips\ n)$ since the clause checking and the computation of the score is done within one cycle. However, the results in Hamadi and Merceron are sketchy and appear to be estimates based on cycle time rather than results of actual implementation and measurement. Hamadi and Merceron claim a speedup over software of two orders of magnitude, but the software timings which are presented seem to be particularly slow and appear to be using an unoptimized implementation of GSAT.[1] In the work by Yung et al., the implementation in FPGAs is similar, but their results are slower than GSAT in pure software. We shall show in Section 3, why this is not surprising.

After introducing a notation for parallel programs in Section 2 that allows for asymptotic complexity analysis, we state and discuss existing hardware-based GSAT implementations in Section 3, and suggest several improvements. Section 4 further optimizes the algorithm through aggressive parallelization. The details for our GSAT implementation are given in Section 5. Section 6 reports the results of an initial experimental evaluation of the described approach.

[1] Possibly one which does not take advantage of the $\mathcal{O}(1)$ implementation of flipping given size assumptions on clause length and variable occurrences.

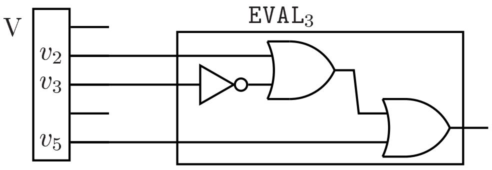
2 Notation

In order to analyse the parallel complexity of GSAT algorithms, we adapt the notation used in [BM99], which in turn adopts central constructs of the parallel functional language NESL [BHSZ95]. We adapt the work-depth model of [BM99] so that we can asymptotically determine the two factors that determine the cost of running a program on an FPGA. The number of gates needed for running the program P is denoted by $g(P)$, which reflects the total size of the FPGA. The depth of a program P is the number of time units required to execute it, and is denoted by $d(P)$ which contributes both to the maximum gate delay within a clock cycle as well as the total number of clock cycles required for execution.

The most basic construct is an assignment such as $P : x := y + z$, where x, y and z are integers. As usual, we assume that integers are represented by a constant number of bits, and thus a constant number of gates suffices to perform integer arithmetic and logical operations, and such operations require only constant time. Thus, $g(P) = \mathcal{O}(1)$ and $d(P) = \mathcal{O}(1)$. Sequential composition $P; Q$ of programs P and Q has the obvious depth $d(P; Q) = d(P) + d(Q)$. The number of gates accumulates in a similar way $g(P; Q) = g(P) + g(Q)$. Note that in some cases the number of gates could be reduced by reusing P's gates for Q. For a sequential loop P : **for** $i = 1$ **to** n **do** Q **end** , we have $g(P) = g(Q)$, since the gates are reused by sequential runs, and $d(P) = n \cdot d(Q)$.

A central feature of the notation is support for sequences (one-dimensional arrays of integers). For example, the assignment $V := [0, 1, 0, 0, 1]$ assigns the sequence of boolean values $[0, 1, 0, 0, 1]$ to a variable V, which can represent an assignment of boolean variables $V_1, \ldots, V_5$. Such sequences are accessed using the usual array notation ($V[3]$ returns 0). Assignment of a field in a sequence is done by $V[3] := 1$, which updates V to $[0, 1, 1, 0, 1]$. A non-destructive substitution expression of the form $V[i \leftarrow x]$ denotes a new sequence that is different by one slot where index i in the sequence has x substituted without affecting V, for example $V[3 \leftarrow 1]$. These sequences are implemented in hardware by arrays of flip-flops. Thus, the depth of both sequence assignments and substitution is $\mathcal{O}(1)$ and the number of gates needed is $O(n)$, where n is the size of the sequence. Note that the implementation of sequences requires that their size must be compile time constant, which is the case for all programs given in this paper.

Since we are constructing a gate array to solve an individual SAT problem, we can encode a clause directly in circuitry. For example, if the third clause of the SAT problem has the form $v_2 \vee \neg v_3 \vee v_5$, we can assume a circuit $\texttt{EVAL}_3(V)$ that evaluates the clause. The circuitry is depicted to the right. Considering that the OR-gates can be arranged into a binary tree structure, for clauses of size n, we have $d(\texttt{EVAL}_i(V)) = \mathcal{O}(\log n)$ and $g(\texttt{EVAL}_i(V)) = \mathcal{O}(n)$. Throughout the paper, log denotes the logarithm function with the base of 2.



The most interesting feature of the notation is with the parallel processing of sequences. This is done using a set like notation. The following expression P evaluates all m clauses of a given SAT problem with n variables in parallel with respect to a given assignment V, $P : \{\mathtt{EVAL}_i(V) : i \in [1..m]\}$. The depth of such a parallel construct is the maximal depth of its parallel components and the number of gates is the sum of the numbers of all component gates. Thus, under the assumptions above, we have $g(P) = \mathcal{O}(mn)$ and $d(P) = \mathcal{O}(\log n)$. Usually there are more variables than clauses in SAT problems, therefore we set $n < m$ for complexity analysis.

The sum of all integers in a given sequence of statically known length n can be computed with the following divide-and-conquer `SUM` program. For simplicity, we assume that n is a power of 2.

macro $\mathtt{SUM}(S, n)$:
 if $n = 1$ **then** $S[0]$
 else $\mathtt{SUM}(\{A[2i] + A[2i+1] : i \in [0..n/2-1]\}, n/2)$

Note that we call `SUM` a macro. We refrain from using runtime functions or procedures in this paper in order to avoid issues regarding parallel calls in the FPGA implementation, which cannot in general map directly to gates. Such macros can be recursive, as long as static macro expansion terminates. This is the case for `SUM`, since the size n of the sequence S is statically known. Consequently, the macro `SUM` creates a binary tree of adders. Thus for a given sequence S of size n, we have $g(\mathtt{SUM}(S, n)) = \mathcal{O}(n)$ and $d(\mathtt{SUM}(S, n)) = \mathcal{O}(\log n)$.

3 Naive GSAT in Hardware

Current Implementations of GSAT

In this section, we will review the implementation of GSAT given in Hamadi and Merceron [HM97]. The work in Yung et al. [YSLL99], is essentially the same but allows clauses with a fixed number of variables to be reconfigured on the FPGA without the need for resynthesis. This is possible because the particular FPGA used, Xilinx XC6216, documents the configuration file for reconfiguring the FPGA. This is not the case with most FPGAs where changing the design requires re-synthesis of the FPGA. As we will be describing both parallel algorithms and the associated hardware, we will in this paper interchangeably use the terms design, implementation, circuit and algorithm where appropriate.

Here, we describe the algorithm sketched in [HM97] in more detail using our notation. This allows for a complexity analysis and comparison. For reasons which we will see later, we will refer to this algorithm as *Naive GSAT*. In Naive GSAT, the inner loop from Figure 1 is implemented in hardware. Meanwhile, the outer loop is implemented in software which is used to make the initial assignment (`INIT_ASSIGN`) and for communication and control to and from the FPGA. The design for `CHOOSE_FLIP` is given in Program 2.

In Program 2, the gate size is primarily bounded by the clause evaluation `EVAL`, therefore, $g(\mathtt{CHOOSE\_FLIP}) = \mathcal{O}(nm)$. The rationale in the design for both

Program 2 CHOOSE_FLIP of Naive GSAT

```
macro CHOOSE_FLIP(f):
    max := −1; f := RANDOM_VARIABLE(n);
    for i = 1 to n do
        score := SUM({EVAL_j(V[i ← ¬V[i]]) : j ∈ [1 . . . m]});
        if (score > max) ∨ (score = max ∧ RANDOM_BIT()) then
            max := score; f := i
        end
    end
```

[HM97,YSLL99] is to make use of the data independence of all calls to EVAL for checking the clauses. This observation and the use of SUM for counting the satisfied clauses yields a depth of $d(\texttt{CHOOSE\_FLIP}) = n * (\mathcal{O}(\log m) + \mathcal{O}(\log n)) = \mathcal{O}(n \ \log m)$. The overall depth of Naive GSAT is $\mathcal{O}(\mathit{maxtries}\ \mathit{maxflips}\ n\ \log m)$.

The experimental results from [YSLL99] show the hardware implementation to be slower than the pure software implementtion of GSAT. A GSAT version 41 from Selman and Kautz, which we refer to as GSAT41, is an optimized software implementation, which usually serves as a reference benchmark implementation, also in this paper. The results from [HM97] are unclear as they appear to be estimates. The software results seem to stem from an unoptimized implementation of GSAT rather than GSAT41, because the flip rate (flips/s) is relatively low. It is however not surprising that neither hardware implementations in [HM97,YSLL99] are particularly fast, as both are based in the GSAT algorithm as given in the paper [SLM92] as opposed to the implementation GSAT41. Furthermore, they assume the bottleneck is in clause evaluation and only parallelize that portion of the algorithm.

Optimized software implementations such as GSAT41 recognize that the basic algorithm of [SLM92] can be greatly improved in practice given two observations: (i) the maximum number of variables in a clause is typically bounded, eg. 3-SAT; and (ii) the maximum number of clauses where a variable occurs in is also bounded. While this does not improve the worst case time complexity in general, it does mean a substantial improvement for many benchmarks and examples occurring in practice, where either one or both of these observations hold. As an example, for a uniform 3-SAT problem, the number of gates for the optimized software becomes $\mathcal{O}(1)$.

This is the reason why we refer to the implementation from [HM97,YSLL99] as Naive GSAT. A detailed description of GSAT41 together with a complexity analysis is given in [Hoo96]. We conclude that it is necessary to parallelize GSAT more agressively in order to significantly improve over GSAT41 running on fast CPUs.

Improving Naive GSAT

A problem of Naive GSAT is that the selection process for the selection of moves is not fair. Sequential calls to the macro RANDOM_BIT generate a bias towards variables that appear earlier in the variable sequence V. Since RANDOM_BIT only

Program 3 CHOOSE_FLIP for Naive GSAT with random selection

```
macro CHOOSE_FLIP(f):
    max := −1; f := RANDOM_VARIABLE(n);
    MaxV := {0 : k ∈ [1 . . . n]};
    for i := 1 to n do
        score := SUM({EVAL_j(V[i ← ¬V[i]]) : j ∈ [1 . . . m]});
        if score > max then
            max := score;
            MaxV := {0 : k ∈ [1 . . . n]}[i ← 1]
        else if score = max then
            MaxV := MaxV[i ← 1]
        end
    end
    f := CHOOSE_ONE(MaxV)
```

produces a stream of 0/1s without knowledge of the underlying V, it is impossible to make a fair variable selection. An improved version of Naive GSAT that avoids this problem is given in Program 3, which also allows the implementation of various variable choice strategies. This version uses a macro CHOOSE_ONE for randomly choosing a value out of a given sequence. This macro is discussed in detail in Section 5. The complexity of gates and depth is unchanged, considering a depth $d(\texttt{CHOOSE\_ONE}) = \mathcal{O}(log\ n)$ and number of gates $g(\texttt{CHOOSE\_ONE}) = \mathcal{O}(n)$.

Parallelism can be increased by using the classical hardware technique of pipelining. The block diagrams in [HM97] show a pipelined implementation, as opposed to [YSLL99] which uses a sequential design. Pipelining can be applied to parallelize operations that multiplies performance with only a minimal increase in the circuit size. The use of pipelining is restricted by data dependencies between operations. In Programs 2 and 3, we can see that only the comparison with *max* is dependent on the results of the previous loop iteration. By making use of an additional queue that ensures data consistency, these designs can be pipelined. Note that while pipelining does not change the asymptotic depth, it can reduce the depth by a constant factor s, where s is the number of stages in the pipeline.

4 A Fully Parallel Design

The speed of the Naive GSAT implementation in the previous section is limited, because only clause evaluation is parallelized and not the variable scoring, hence the minimal depth of CHOOSE_FLIP after applying pipelining is still $\mathcal{O}(n)$.

In Program 2, there is no data dependency between the score computations for the variables. Program 4 improves over Program 2 by exploiting this obvious parallelization opportunity using parallel score computation.

The depth of Program 4 is $\mathcal{O}(\log m)$, since the *Scores* computation is bounded by $\mathcal{O}(\log m+\log n)$ and the CHOOSE_MAX computation is bounded by $\mathcal{O}(\log n)$ (see Section 5), and we assumed $n < m$. While this design comes closer to our goal,

Program 4 Basic CHOOSE_FLIP Design with Parallelized Variable Scoring

macro CHOOSE_FLIP(f):

$\quad Scores := \{\mathtt{SUM}(\{\mathtt{EVAL}_j(V[i \leftarrow \neg V[i]]) : j \in [1 \dots m]\}) : i \in [1 \dots n]\};$

$\quad f := \mathtt{CHOOSE\_MAX}(Scores);$

its drawback lies in an increase of the circuit size by a factor of n to $\mathcal{O}(mn^2)$. With the exception of small problems, this design is not practical.

Selective Parallel Score Computation

To alleviate this problem, we turn to an alternative hardware design. The idea is related to the software optimizations in GSAT41, but here the rationale is to decrease the circuit size while keeping parallel score evaluation. The key observations are:

- The selection of the flip variable can be done on the basis of relative contribution to the score of that variable when flipped.
- The number of clauses which will be affected by a change to one variable is small and typically bounded.

The new optimized design is given in Program 5. As we need to refer to only the affected clauses, we will use the notation $\mathtt{EVAL}_j^{\mathcal{C}(i)}$ to denote the j-th clause from the set of clauses which contain variable i and can be thought of as a fixed boolean function for a particular SAT problem. $NCl[i]$ is a constant and denotes the number of clauses containing variable i.

The total number of $\mathtt{EVAL}_j^{\mathcal{C}(i)}$ needed for Program 5 is bounded by the number of instances of variable i for all clauses. We will denote the bound on the maximal number of clauses per variable as *MaxClauses*. In practice, most problems have also a bound on the number of variables per clause, which we denote by *MaxVars*. For example, for 3-SAT, *MaxVars* is 3. Thus, the number of gates for Program 5 is $\mathcal{O}(MaxVars\ MaxClauses\ n)$. The depth for Program 5 is $\mathcal{O}(\log MaxClauses + \log MaxVars)$, which for practical SAT problems is much smaller than $\mathcal{O}(\log m)$. We remark that one more advantage of this design is that the circuit for SUM is smaller now, because the numbers to be added require fewer bits.

Multi-Try Pipelining

The last step taken for achieving one flip per clock cycle is to push pipelining to its limits. With Program 5 the innermost loop of GSAT is now operating over

Program 5 Parallel CHOOSE_FLIP with relative scoring

macro CHOOSE_FLIP(f):

s1: $\quad NewS := \{\mathtt{SUM}(\{\mathtt{EVAL}_j^{\mathcal{C}(i)}(V[i \leftarrow \neg\, V[i]]) : j \in [1 \dots\ NCl[i]]\}) : i \in [1 \dots\ n]\};$

s1: $\quad OldS := \{\mathtt{SUM}(\{\mathtt{EVAL}_j^{\mathcal{C}(i)}(V) : j \in [1 \dots\ NCl[i]]\}) : i \in [1 \dots\ n]\};$

s2: $\quad Diff := \{NewS[i] - OldS[i] : i \in [1 \dots\ n]\};$

s3: $\quad MaxDiff := \mathtt{OBTAIN\_MAX}(Diff);$

s4: $\quad MaxVars := \{Diff[i] = MaxDiff : i \in [1 \dots\ n]\};$

s5: $\quad f := \mathtt{CHOOSE\_ONE}(MaxVars);$

each flip. Unfortunately, it is not possible to pipeline the different flip iterations of `CHOOSE_FLIP`, since each iteration is data dependent on the flip of the previous iteration. Instead, we pipeline the outer loop of Program 1; we call this multi-try pipelining. Since there is no dependency between different tries in GSAT, essentially one can parallelize each try independently. Each pipeline stage deals in parallel with the work for a different try. For simplicity, *maxtries* should be a multiple of the number of stages in the pipeline.

In practice, for the actual implementation it is feasible in one clock cycle to accommodate the evaluation of every $\texttt{EVAL}_j^{\mathcal{C}(i)}$ and the computation of `SUM`. Therefore, we only need to allocate each design block in Program 5 to a pipeline of five stages. The five stages, list as *s1* to *s5*, can be found in Program 5 is illustrated below.

Tries	Time1	Time2	Time3	Time4	Time5	Time6	Time7	Time8	...
Try1	*s1*	*s2*	*s3*	*s4*	*s5*	*s1*	*s2*	*s3*	...
Try2		*s1*	*s2*	*s3*	*s4*	*s5*	*s1*	*s2*	...
Try3			*s1*	*s2*	*s3*	*s4*	*s5*	*s1*	...
Try4				*s1*	*s2*	*s3*	*s4*	*s5*	...
Try5					*s1*	*s2*	*s3*	*s4*	...

5 GSAT on FPGA Implementation

In this section, we describe further refinements of the design, which result in our final implementation of GSAT on an FPGA. Specific implementation details are discussed for each stage of the design.

In Program 5 stage *s1*, the relative contribution of a variable to the score is computed twice; once for the current value of the variable and once for the flipped value. The corresponding circuits for clause evaluation and summation are essentially duplicated. In a sequential implementation one could reuse the clause evaluation and summation. However given either the use of pipelining or parallel evaluation of the two sequences, reuse of the circuits is prohibited by resource dependency, and duplication of the circuits is necessary.

We therefore propose a refinement to the circuits ofr clause evaluation and summation that reduces the overall circuit size. We first introduce some notation. Instead of working with the original form of the clauses, we use a reduced form. Let $\mathcal{C}(v^+)$ denote the a new set of clauses where variable v occurs positively in the original clauses, and where in each clause, v itself has been deleted. Similarly, $\mathcal{C}(v^-)$ contains those clauses where variable v occurs negated, and where in each clause, v has been deleted. These new clauses are smaller by one variable. We use the term $\texttt{EVAL}_i^{\mathcal{C}(v^+)}$ to denote the evaluation circuit for clause i in the clause set $\mathcal{C}(v^+)$, and similarly $\texttt{EVAL}_i^{\mathcal{C}(v^-)}$. The idea in the previous section was that it was sufficient to consider the relative effect on the score on a per variable basis. We use the term $rscore(v)$ to denote the relative score for the clauses defined on v with respect to the current assignment. We know that when $v = 1$, all the clauses in $\mathcal{C}(v^+)$, but not necessarily all clauses in $\mathcal{C}(v^-)$, are satisfied, which results in:

$$
rscore(v) = \begin{cases} \texttt{SUM}(\{\texttt{EVAL}_i^{\mathcal{C}(v^-)} : i \in [1 \ldots |\mathcal{C}(v^-)|]\}) + |\mathcal{C}(v^+)| & \text{if } v = 1 \\ \texttt{SUM}(\{\texttt{EVAL}_j^{\mathcal{C}(v^+)} : j \in [1 \ldots |\mathcal{C}(v^+)|]\}) + |\mathcal{C}(v^-)| & \text{if } v = 0 \end{cases}
$$

To simplify the discussion and program, we define

$$
\begin{aligned}
\mathit{Dyn1}[v] &= \texttt{SUM}(\{\texttt{EVAL}_i^{\mathcal{C}(v^-)} : i \in [1 \ldots |\mathcal{C}(v^-)|]\}) \\
\mathit{Dyn0}[v] &= \texttt{SUM}(\{\texttt{EVAL}_j^{\mathcal{C}(v^+)} : j \in [1 \ldots |\mathcal{C}(v^+)|]\})
\end{aligned}
$$

These refer to the evaluation of the reduced two new clauses where v occurs positively and negatively only. Note that v itself is not used in the circuit. Furthermore, we define the constant values

$$
\mathit{Static}[v] = |\mathcal{C}(v^+)| \; - |\mathcal{C}(v^-)|
$$

The relative change to the score when a variable v is flipped from 0 to 1 is the difference in *rscore* for both values of v, which is:

$$
\mathit{Diff}'[v] = \mathit{Dyn1}[v] - \mathit{Dyn0}[v] + \mathit{Static}[v]
$$

Note that this is not the same as $\mathit{Diff}[v]$ in Program 5 since the sign depends on the direction in which v is flipped.

We illustrate the computation with the following example where $n = 4$ and $m = 8$. A clause $v_1 \vee v_2 \vee \neg v_3$ is written in the form $(1\ 2\ \ -3)$. The current assignment of the variables v_1, v_2, v_3, v_4 is the sequence $[1, 1, 1, 0]$.

All clauses	Clauses with variable	
	1^+	1^-
(1 2 3)	(1 2 3)	(-1 -2 -3)
(1 2 4)	(1 2 4)	(-1 -2 3)
(-1 -2 -3)	(1 3 4)	
(-1 -2 3)	(1 -3 -4)	
(1 3 4)	Simplified clauses	
(-2 3 -4)	$\mathcal{C}(1^+)$	$\mathcal{C}(1^-)$
(1 -3 -4)	(2 3)	(-2 -3)
(2 3 4)	(2 4)	(-2 3)
	(3 4)	
	(-3 -4)	

$\mathit{Static}[1] = 2$
$\mathit{Dyn1}[1] = 1$
$\mathit{Dyn0}[1] = 4$

flip $0 \to 1$ gives:
$\mathit{Diff}'[1] = 1 - 4 + 2 = -1$
flip $1 \to 0$ is $-\mathit{Diff}'[1]$

Program 6 shows the complete design on the FPGA with a five staged multi-try pipeline, labelled *s1* to *s5*. Each stage is executed in one cycle, thus we will assume that the circuit for each stage can execute within the time constraints of one cycle. `RECEIVE_INITIAL_ASSIGNMENT()` and `SEND_ASSIGNMENT()` perform the data transfer from and to the software in that order. The `SATISFIED`(V) macro (discussed later) exits the loop, when a satisfying assignment is found. Both the *Dyn0* and *Dyn1* are computed in parallel at stage *s1*, and are used

Program 6 Final implementation

MAIN():

  $V :=$ RECEIVE_INITIAL_ASSIGNMENT();

  for $i := 1$ **to** *maxflips* **do**

s1   **if** SATISFIED(V) **then BREAK** ;

s1 $\mathit{Dyn0} := \{\texttt{SUM}(\{\texttt{EVAL}_j^{\mathcal{C}(i^+)} : j \in [1 \ldots |\mathcal{C}(i^+)|]\}) : i \in [1 \ldots n]\}$;

s1 $\mathit{Dyn1} := \{\texttt{SUM}(\{\texttt{EVAL}_j^{\mathcal{C}(i^-)} : j \in [1 \ldots |\mathcal{C}(i^-)|]\}) : i \in [1 \ldots n]\}$;

s2 $\mathit{Diff}' := \{\mathit{Dyn1}[i] - \mathit{Dyn0}[i] + \mathit{Static}[i] : i \in [1 \ldots n]\}$;

s3 $\mathit{MaxDiff} :=$ OBTAIN_MAX(Diff');

s4 $\mathit{MaxVars} := \{\mathit{Diff}'[i] = \mathit{MaxDiff} : i \in [1 \ldots n]\}$;

s5 $v :=$ CHOOSE_ONE($\mathit{MaxVars}$);

s5 $V[v] := \neg V[v]$;

  end ;

  SEND_ASSIGNMENT(V);

to compute Diff' at stage *s2*. At stage *s3*, the OBTAIN_MAX macro retrieves the maximum relative score difference for all variables stored in the sequence Diff'. Upon knowing the value of the maximum change in the score, stage *s4* finds and selects all variables that correspond to the highest increase in score. In the last stage *s5*, we integrate both the CHOOSE_ONE and the actual flipping of the variable into a single stage. The CHOOSE_ONE makes a fair selection of one variable from a list of variables in *MaxVars*. After we flip the variable, the flip counter i is incremented and all stages are repeated.

The multi-tryp pipeline that parallelizes five tries corresponding to the five pipeline stages is realized using an additional scheduling queue to switch between multiple tries. Separate queues are added for the results of each stage in the pipeline. Due to the constant overhead for pipelining, the resulting design has an asymptotic performance of one-flip per clock cycle as *maxflips* increases.

Support Macros

The SATISFIED Macro. This macro represents the entire CNF formula. Due to the optimization for the clause evaluation based on the relative scores of variables, the information that all clauses are satisfied is lost, and thus this macro is needed. The macro implements an conjunction of disjunctions, each representing a clause. Thus we get $d(\texttt{SATISFIED}) = \mathcal{O}(\log m \ \log \mathit{MaxVars})$ and $g(\texttt{SATISFIED}) = \mathcal{O}(m \ \mathit{MaxVars})$.

The OBTAIN_MAX Macro. This macro returns the maximum value from a sequence. We use comparators structured in a binary tree, similar to the SUM macro in Section 2. The complexities are $d(\texttt{OBTAIN\_MAX}) = \mathcal{O}(\log n)$ and $g(\texttt{OBTAIN\_MAX}) = \mathcal{O}(n)$.

The CHOOSE_ONE Macro. This macro selects one variable at random from the input set of variables. To make the variable selection fair, we implement a shift-register-based pseudo random number generator where $g(\texttt{RANDOM}) = \mathcal{O}(1)$ and $d(\texttt{RANDOM}) = \mathcal{O}(1)$. While it is possible to use mod, to simplify the circuit, we use instead a binary decision tree where a random bit selects between the left and right branches. This gives $d(\texttt{CHOOSE\_ONE}) = \mathcal{O}(\log n)$ and $g(\texttt{CHOOSE\_ONE}) = \mathcal{O}(n)$.

6 Experimental Evaluation

In the implementation for Program 6, we have used a C-like design language, Handel-C [Pag96,APR$^+$96] which compiles the program to a gate level description. Handel-C was chosen, because it has a simple timing model which fits well the analysis of gates and depth used here. Handel-C does not have the sequences used here but has a parallel construct which can be used to implement the parallel evaluation of sequences. Individual statements execute in one clock cycle and thus sequencing and loops fit the model here. Expressions and variables can be declared on arbitrary bit sizes, which is consistent with the $\mathcal{O}(1)$ assumptions for operations on integers. Handel-C is convenient for rapid prototyping and we observed a shorter development cycle than with traditional hardware design languages such as VHDL or Verilog. While VHDL and Verilog give finer control and possibly better performance, the Handel-C implementation used in the experimental evaluation is sufficient to demonstrate the efficiency and efficacy of our GSAT designs.

The hardware used with Handel-C is their supplied prototyping board, RC-1000PP. The RC-1000PP board includes an XCV1000 FPGA from Xilinx, and allows a maximum clock rate of 33MHz when using the 4 Mbytes of on-board RAM. The XCV1000 itself is capable of running at clock speeds of up to 300MHz and includes 1Mbits of distributed RAM. The XCV1000 chip contains 6144 CLBs (configurable logic blocks), which roughly amounts to 1M system gates. Each CLB in the virtex series is divided into 2 slices, and thus the chip is capable of programming 12,288 slices.

The preliminary experimental results reported in Table 1 compare the flip rate per second between:

- the software implementation of GSAT41 by Selman and Kautz with the hill-climbing option run on a Pentium II-400MHZ machine with 256MB of memory (Software),
- the FPGA implementation of Program 2 with pipelining (Naive GSAT), and
- the FPGA implementation of Program 6 (Multi-Try).

Our implementations for both Naive and Multi-Try use software for the outer loop and the FPGA for the entire inner loop. The measurements are the average times from measuring the time used for the FPGA itself, and is subject to some experimental timing variation. The theoretical flip rate for Multi-Try is approximately equal to the clock rate since it achieves one flip per clock cycle.

The results in Table 1 shows the disadvantage of the naive implementation. Its speed in flips per second (fps) is inversely proportional to the number of variables in the problem. As the number of variables increases, the fps of Multi-Try

Table 1 Speed Comparison of Different GSAT Implementations

SAT Problems			*Software*	*Naive*	*Multi-Try*	*Speed-Up Ratio*	
Name	Var (n)	Clause (m)	(Selman 41) K *fps*	@20MHZ K *fps*	@20MHZ M *fps*	vs. SW	vs. Naive
uf20-01	20	91	47.7	962.9	35	734	36
uf50-01	50	218	74.4	383.6	24	323	63
uf100-01	100	430	72.7	194.9	21.6	297	111
uf200-01	200	860	70.8	98.6	20.7	292	210
aim-50-1_6-yes1-1	50	80	129.8	383.4	23.9	184	62
aim-50-2_0-yes1-1	50	100	111.1	383.4	24	216	63
aim-50-3_4-yes1-1	50	170	75.4	383.7	24	318	63
aim-50-6_0-yes1-1	50	300	40.5	383.5	23.9	590	62
aim-100-1_6-yes1-1	100	160	140.1	194.9	21.7	155	111
aim-100-2_0-yes1-1	100	200	111.0	194.9	21.7	195	111
aim-100-3_4-yes1-1	100	340	71.8	194.9	21.6	301	111
aim-100-6_0-yes1-1	100	600	39.6	194.9	21.6	545	111
aim-200-1_6-yes1-1	200	320	121.4	98.6	20.7	171	210
aim-200-2_0-yes1-1	200	400	98.5	98.6	20.7	210	210
aim-200-3_4-yes1-1	200	680	67.5	98.6	20.7	307	210
aim-200-6_0-yes1-1	200	1200	38.9	98.6	20.7	532	210
flat30-1	90	300	94.4	216.0	21.9	232	101
flat50-1	150	545	92.7	130.9	21	227	160
rti_k3_n100_m429_0	100	429	72.5	195.0	19.8	273	102
bms_k3_n100_m429_0	100	429	117.3	195.0	21.6	184	111

only decreases by a small amount. We see that due to the subsumption of the cost of `SUM` within a clock cycle the flip rate is not affected by the number of clauses. The speed-up for Multi-Try versus Naive is at least one order of magnitude and increases with the problem size. When compared with the optimized software implementation, Multi-Try exhibits a speed-up of two orders of magnitude. Note that the software is running on a machine with a clock rate, which is one order of magnitude faster.

Due to the absense of date dependencies, the parallelism to be extracted from the outer loop is unlimited. Such algorithms are often called "embarassingly parallel". The cost of exploiting this parallelism lies in the hardware needed. A performance measure for computing devices that takes this hardware cost into account is called computational density [DeH96] and measures bit operations per second per micron square. We propose to apply this cost measure to SLS algorithms running on FPGAs. We define flip density to be the number of flips per second per slice of the FPGA. For a given FPGA architecture (here the Xilinx Virtex family), the flip density adequately measures the peroformance of a GSAT implementation.

In Table 2, the size of the circuits for both designs are listed in terms of slices. The minimal gate delay—as reported by the Xilinx synthesis tools—for these examples lies between 13 and 22 nanoseconds, but does not vary significantly between the two implementations. By cross referencing the fps from the first

Table 2 Performance Comparison of FPGA-based Implementations

Problem	*Naive*			*Multi-Try*			
	Delay (ns)	Size (slice)	Flip Density (fps/slice)	Delay (ns)	Size (slice)	Flip Density (fps/slice)	Impv.
uf20-01	11	511	1884	14	1483	23601	13
uf50-01	16	1006	381	14	3149	7621	20
uf100-01	24	1825	107	25	5882	3672	34
uf200-01	21	3481	28	27	11848	1747	62
aim-50-1_6-yes1-1	12	650	590	14	1816	13161	22
aim-50-2_0-yes1-1	14	705	544	16	1772	13544	25
aim-50-3_4-yes1-1	15	889	432	12	2462	9748	23
aim-50-6_0-yes1-1	15	1219	315	15	3530	6771	21
aim-100-1_6-yes1-1	18	1136	172	16	3402	6379	37
aim-100-2_0-yes1-1	17	1242	157	19	3191	6800	43
aim-100-3_4-yes1-1	17	1559	125	16	4714	4582	37
aim-100-6_0-yes1-1	19	2271	86	22	6635	3255	38
aim-200-1_6-yes1-1	22	2100	47	14	6442	3213	68
aim-200-2_0-yes1-1	17	2304	43	20	6304	3284	76
aim-200-3_4-yes1-1	23	3019	33	21	9103	2274	69
aim-200-6_0-yes1-1	26	4328	23	30	12286	1685	73
flat30-1	18	1440	150	15	3546	6176	41
flat50-1	20	2409	54	20	6042	3476	64
rti_k3_n100_m429_0	19	1824	107	21	5874	3371	32
bms_k3_n100_m429_0	16	1463	133	20	4955	4359	33

table, a result in terms of flip density is shown. The last column compares the two algorithms with respect to flip density, and shows an improvement of factors between 13 and 76. The improvement factor increases with the problem size.

We remark that the results are preliminary as we are using a beta version of the Handel-C version 3 software. There are limitations in the beta version for compiling larger designs. We are also limited by the maximum clock speed of the RC-1000-PP board due to the interaction between external RAM and the simple Handel-C timing model, even though the XCV1000 FPGA is itself capable of being clocked at higher speeds. This does not diminish our results as it is possible to implement our design and algorithms in VHDL or Verilog, which would incur a slower development cycle.

7 Conclusion

We have shown that previous work on implementing the GSAT family of algorithms using FPGAs leave considerable room for improvement. From an implementation of the algorithms described by Hamadi and Merceron [HM97] and Yung et al [YSLL99], we proceeded in three stages:

– We achieved a uniform random selection of candidate flips by storing the candidate flips in a vector and employing a binary decision tree (`CHOOSE_ONE`).

– We parallelized the score computation and still avoided excessive use of gates.
– We exploited the absense of data dependencies by using multi-try pipelining.

The resulting algorithm achieves an improvement of the depth by at least a factor of n, where n is the number of variables. It implementation on an FPGA achieves one flip per clock cycle. Preliminary experimental evaluation shows that formula of realistic size can be solved using the presented algorithm with current FPGA technology running at reasonably high clock speed. The improvement over an optimized sequential implementation is more than two orders of magnitude. We analysed the combined effect of increased flip rate and increased space consumption using the cost measure of flip density, which showed an improvement of more than one order of magnitude compared to existing FPGA-based implementations.

Acknowledgements

We thank the company Celoxica for providing technical support and an educational license for Handel-C.

References

[APR+96] M. Aubury, I. Page, G. Randall, J. Saul, and R. Watts. Handel-C language reference guide. Technical report, Oxford University Computing Laboratory, Oxford, UK, 1996.

[BHSZ95] Guy Blelloch, Jonathan Hardwick, Jay Sipelstein, and Marco Zagha. NESL user's manual, version 3.1. Technical Report CMU-CS-95-169, Carnegie Mellon University, Pittsburgh, PA, 1995.

[BM99] Guy Blelloch and Bruce Maggs. Parallel algorithms. In *Algorithms and Theory of Computation Handbook*. CRC Press, Boca Raton, Florida, 1999.

[DeH96] A. DeHon. *Reconfigurable Architectures for General-Purpose Computing*. PhD thesis, The MIT Press, Cambridge, MA, September 1996.

[Gu92] J. Gu. Efficient local search for very large-scale satisifiability problems. *SIGART Bulletin*, (3):8–12, 1992.

[HM97] Youssef Hamadi and David Merceron. Reconfigurable architectures: A new vision for optimization problems. In Gert Smolka, editor, *Principles and Practice of Constraint Programming - CP97, Proceedings of the 3rd International Conference*, Lecture Notes in Computer Science 1330, pages 209–221, Linz, Austria, 1997. Springer-Verlag, Berlin.

[Hoo96] Holger Hoos. Aussagenlogische SAT-Verfahren und ihre Anwendung bei der Lösung des HC-Problems in gerichteten Graphen. Diplomarbeit. Fachbereich Informatik, Technische Hochschule Darmstadt, Germany, March 1996.

[HS00] Holger H. Hoos and Thomas Stützle. Local search algorithms for SAT: An empirical evaluation. *Journal of Automated Reasoning*, 24(4):421–481, 2000.

[MSK97] David McAllester, Bart Selman, and Henry Kautz. Evidence for invariants in local search. In *Proceedings Fourteenth National Conference on Artificial Intelligence (AAAI-97)*, 1997.

[Pag96] I. Page. Constructing hardware-software systems from a single description. *Journal of VLSI Signal Processing*, (12):87–107, 1996.

[SKC94] B. Selman, H. Kautz, and B. Cohen. Noise strategies for improving local search. In *Proceedings of AAAI-94*, pages 337–343, 1994.

[SLM92] B. Selman, Hector Levesque, and David Mitchell. A new method for solving hard satisfiability problems. In *Proceedings of AAAI-92*, pages 440–446, 1992.

[YSLL99] Wong Hiu Yung, Yuen Wing Seung, Kin Hong Lee, and Philip Heng Wai Leong. A runtime reconfigurable implementation of the GSAT algorithm. In Patrick Lysaght, James Irvine, and Reiner W. Hartenstein, editors, *Field-Programmable Logic and Applications*, pages 526–531. Springer-Verlag, Berlin, / 1999.