A Runtime Reconfigurable Implementation of the GSAT Algorithm

Wong Hiu Yung, Yuen Wing Seung, Kin Hong Lee, and Philip Heng Wai Leong

Department of Computer Science and Engineering
The Chinese University of Hong Kong, Shatin, N.T. Hong Kong
{hywong2,wsyuen,khlee,phwl}@cse.cuhk.edu.hk

Abstract. Boolean satisfiability (SAT) problems are an important subset of constraint satisfaction problems (CSPs) which have application in such areas as computer aided design, computer vision, planning, resource allocation and temporal reasoning. In this paper we describe an implementation of an incomplete heuristic search algorithm called GSAT to solve 3–SAT problems. In contrast to other approaches, our design is runtime configurable. The input to this system is a 3–SAT problem from which a software program directly generates a problem–specific configuration which can be directly downloaded to a Xilinx XC6216, avoiding the need for resynthesis, placement and routing for different constraints. We envisage that such systems could be used in hardware based real time constraint solving systems.

1 Introduction

A constraint satisfaction problem (CSP) is a problem with a finite set of variables. These variables can take values within a certain finite domain subject to a set of constraints which restricts them. The solution of a CSP involves finding an assignment of the variables which violates no constraints. Many real life problems such as scheduling, graph coloring, circuit test pattern generation, circuit synthesis and scene labeling can be formulated as constraint satisfaction problems. These are mostly NP hard problems and algorithms to efficiently solve them have been the field of active research in artificial intelligence (AI).

Algorithms for solving CSPs can be complete or incomplete. Complete algorithms, by definition, can find all of the solutions to a problem and typically involve pruned tree searches. However, since CSPs are NP hard, in practice the long execution times involved make it difficult to find any solution. In many applications, completeness is unnecessary and by removing this restriction, much faster heuristic algorithms can be employed to quickly scan the search space. Such incomplete algorithms are able to solve otherwise intractable CSPs. GSAT [1] is an incomplete algorithm for solving the boolean SAT problem. The boolean SAT problem is a CSP in which the variables are binary and the constraints are represented by a boolean equation in a product of sums form. Each sum term is called a clause and is the sum of single literals, where a literal is a variable or its negation. If there are n literals in each clause, the problem is called an n–SAT problem.

Several hardware designs which solve CSPs have been proposed [2–5] and all shown significant speed improvement over software implementations. An important limitation of all of the complete and incomplete implementations described above is that they generated a high level description of a circuit which was customised for the constraint problem. Thus a complete iteration of the synthesis, place and route cycle was required for each new set of constraints. These steps are time consuming (it can take several hours to synthesize, place and route a large design) and precludes its use in real time systems.

In this paper we describe an FPGA architecture and its implementation which enables the direct generation of a problem specific configuration for solving the GSAT problem. Our design exploits the open architecture of the Xilinx XC6200 series FPGAs which is used to directly configure the circuit from the constraints. The remainder of the circuit is fixed. Using this technique, the need for synthesis, placement and routing of a new circuit for different constraints is eliminated, resulting in a large savings in compilation time. It should be easy to adapt this reconfiguration technique to other constraint solving algorithms.

2 Implementation

The GSAT algorithm [1] is a simple greedy search based method for solving SAT problems. Basically, each iteration of the algorithm is performed by flipping the bit of the variable which would result in the largest improvement in the number of satisfied clauses. This process is repeated Maxflips times. If no solution is found, a new random variable assignment is made and the process repeated (up to Maxtries times). For a boolean constraint equation F, the algorithm can be described by the following pseudocode

```
GSAT(int Maxtries, int Maxflips) {
   for (i=1 to Maxtries) {
       V = a random instantiation of the variables;
       for (i = 1 to Maxflips) {
           p = variable whose negation yields largest
               increase in number of satisfied clauses;
           V = V with p flipped;
       }
       if (F(V) is true)
           return V;
   }
   return the best instantiation found;
}
```

Our implementation of the GSAT algorithm uses reconfigurable hardware to implement the inner loop of the algorithm described in the previous section, i.e. the calculation of p. The rest of the GSAT algorithm is implemented in software which maximises the flexibility of our hardware. The software downloads a variable assignment to the board. The hardware flips each variable in turn and computes the number of satisfied clauses and after each variable has been flipped, the variable whose negation yielded largest increase in number of satisfied clauses is returned to the software program. The software then computes the next variable assignment by flipping that variable.

A block diagram of the hardware architecture is shown in Figure 1. The *variable memory* is a register that is used to hold the current variable configuration is called the variable memory. The host computer writes a new variable configuration every iteration of the algorithm. The *flip–bit vector* is a shift register used to cycle through each variable of the variable memory in sequence. Only one bit of the flip–bit vector is asserted at any time so an exclusive–OR of the flip–bit vector and the variable memory will negate that bit (see Figure 2). The *clause checkers* are used to implement the sum terms of the constraint equation. Inputs to the clause checkers are variable assignments. For a problem with n clauses, n parallel clause checkers are used to implement all of the clauses and since our hardware is restricted to 3–SAT problems, each clause checker has 3 inputs. Thus each clause checker is simply a 3 input OR gate with active low or active high inputs depending on the equation. The *sum register* is used to store the largest number of satisfied clauses. The *M–bit adder* is used to calculate the number of satisfied clauses. For a problem with M clauses, M 1 bit numbers must be summed to find the number of satisfied clauses. Since the adder is in the critical path of the design, a tree adder was used. As a result, there will be $\log_2 M$ levels of delay. Moreover, each level towards the root of the tree has an additional bit of precision. Thus in the kth level, all inputs are k–bits and they are added together in a pairwise fashion to generate a k+1 bit result. The *comparator* is used to compare the current number of the satisfied clauses with the value in the sum register to see if a better result had been found. In the event that the current number of satisfied clauses is equal to the best value stored in sum register, the *random bit generator* is used to decide which solution to keep. This prevents the algorithm from being captured in a local minima. The Random bit generator is implemented as a linear feedback shift register and the equation used is $\text{bit}(0) = \text{bit}(3) \text{ xor } \overline{\text{bit}(4)} \text{ xor } \text{bit}(5) \text{ xor } \text{bit}(7)$.

The target device for our implementation was a Xilinx XC6216 reconfigurable processing unit (RPU) [6]. This device was selected mainly because the open architecture of the RPU documents how the low level configuration bits relate to the hardware of the device. Other FPGA architectures do not provide this information making it extremely difficult to generate configurations without going through the vendor's CAD software. The Virtual Computer Company (VCC) H.O.T. Works development system was used to test the design. This PCI board includes a Xilinx XC6216 reconfigurable processing unit (RPU) together with SRAM memory and a PCI interface.

With the exception of the clause checkers, the circuits were synthesised from VHDL descriptions using the Velab VHDL elaborator version 0.52. They were then placed and routed using the Xilinx XACTstep 6000 tools. The clause checkers are problem dependent and are customised by a C program. The implementation of a clause is illustrated in Figure 3. All of the variables are routed in horizontal lines and the logic to implement a particular clause are distributed in a vertical direction. In order to implement the routing and logic, 5 different logic configurations are required as detailed in the figure. Compared with the general problem of placement, routing or logic synthesis, this task is trivial and is performed in linear time. The software customises the logic equation of each clause checker and writes the new configuration into the address mapped configuration of the XC6200 memory [6]. In the XC6200 devices, this can be done without affecting the nonconfigurable parts of the circuit.

Ultimately, the compactness of the circuit implementation determines the size of the SAT problem that can be solved using the system. For v variables and n clauses, the amount of logic required for our design is variable memory (v); flip–bit ($2v$); clause checkers (nv); sum register ($3\log_2 v$); adder ($3n\sum_{k=1}^{\log_2 n}(k/2)^k+\log_2 n$); comparator ($8\log_2 n$); and random number generator (12). For the XC6200 device, we estimated that the total resources required (i.e. logic+routing) were approximately $4\times$ that of the logic alone. The clause checkers dominate the size of the circuit. A method of overcoming this problem is to use a non–runtime reconfigurable clause checker which utilises exactly the same design except that the clause checker circuitry is synthesised from VHDL. This customisation of the clause checker reduces its logic from nv to $2n$ since a specific routing pattern is much more compact than the general case. With a non–runtime reconfigurable clause checker, the 50 variable 80 clause problem to fit on the XC6216 which contains 4096 cells. A smaller 8 variable 16 clause problem was used to test the runtime reconfigurable clause checker.

3 Results

The design was tested on the aim-50-1_6-yes1-1.cnf (aim) benchmark problem from the Second DIMACS Implementation challenge on NP Hard Problems: Maximum Clique, Graph Coloring, and Satisfiability [7]. This problem has 50 variables and 80 clauses. A smaller problem (small) with 8 variables and 16 clauses was also tested. This problem was generated using the *mwff* program from DIMACS [7, 8].

Table 1 shows the total VHDL compile time, VHDL compile time of just the clause checker and the execution time of our runtime configuration approach using a Pentium II 300MHz 64MB Ram system running Winows 98. It can be seen that a four orders of magnitude improvement in the time required to translate the problem into the clause checking circuit is seen. The same table also compares the execution time of a software implementation of the GSAT program version 41 written by Selman and Kautz [1] running on a Sun Ultra–5 machine with the hardware execution times. The final column shows the maximum clock frequency given by the XACTstep software.

Our design is currently faster than the software version for the small problem but slower for the aim problem. Although a software implementation on a typical workstation is roughly two times faster for the aim problem, it must be

Problem	Total	Clause Checker	Runtime Config	Software	Hardware	Frequency
	(s)	(s)	(ms)	(ms)	(ms)	(MHz)
small	52	16	1.2	0.59	0.032	9.12
aim	900	90	7	5.14	13.5	4.17

Table 1. Table comparing the compilation and execution times of various GSAT implementations.

remembered that it is achieved using very few logic gates. For embedded systems where circuit size, cost, power dissipation and reliability are concerned the RPU implementation offers significant advantages.

4 Conclusion

Using a problem specific architecture and runtime configuration, a four order of magnitude speedup in the reconfiguration time of the GSAT algorithm over the conventional approach involving resynthesis, placement and routing was demonstrated. The design was tested on hardware and achieved approximately the same performance as that of a modern workstation but at greatly reduced hardware cost, power consumption and memory requirements. We envisage that such systems could be used in hardware based real time constraint solving systems and may have applications in signal processing, robotics and control.

References

1. B. Selman et. al., "A new method for solving hard satisfiability problems," in *Proc. of AAAI-92*, pp. 440–446, 1992.
2. M. Yokoo et. al., "Solving satisfiability problems using field programmable gate arrays: First results," in *Proc. of the 2nd Inter. Conf. on Principles and Practice of Constraint programming*, pp. 497–509, 1996.
3. P. Zhong et. al., "Accelerating boolean satisfiability with configurable hardware," in *IEEE Symposium on Field-Programmable Custom Computing Machines*, pp. 186–195, 1998.
4. Y. Hamadi and D. Merceron, "Reconfigurable architectures: A new vision for optimization problems," in *Principles and Practice of Constraint Programming CP97*, (Austria), pp. 209–215, 1997.
5. T. Suyama et. al., "Solving satisfiability problems using logic synthesis and reconfigurable hardware," in *Proc. 31st Annual Hawaii Internation Conf. on System Sciences.*
6. X. Inc., *XC6200 Programmable Gate Arrays (data sheet)*. 1997.
7. *Dimacs challenge benchmarks.* ftp://dimacs.rutgers.edu/pub/challenge.
8. D. Mitchell et. al., "Hard and easy distributions of sat problems," in *Proc. AAAI-92*, pp. 459–465, 1992.

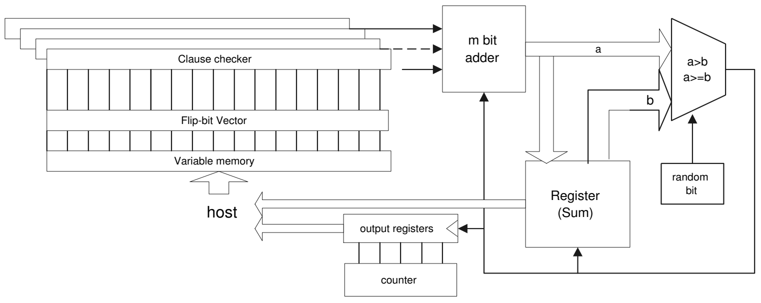

Fig. 1. Block diagram of the GSAT hardware.

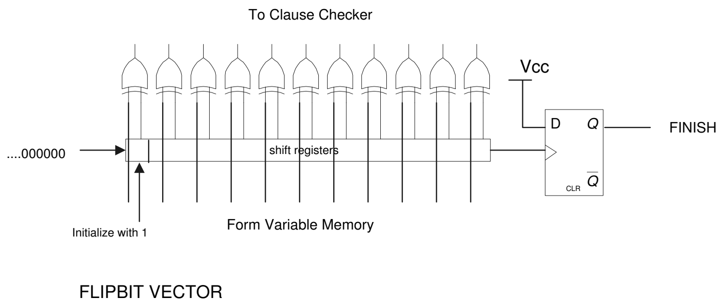

Fig. 2. Flip bit vector detail.

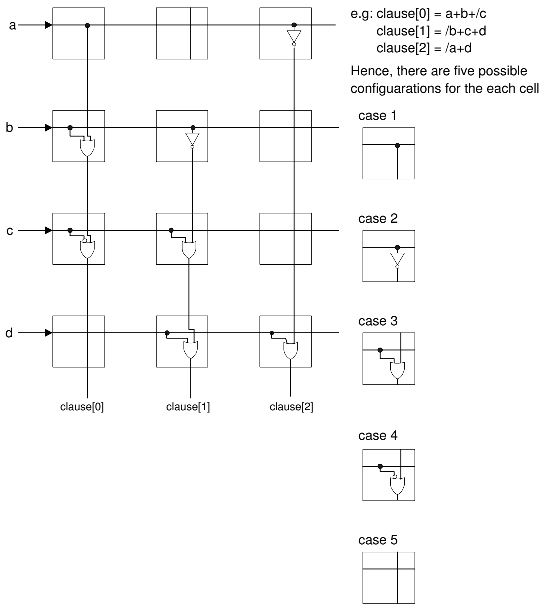

Fig. 3. Implementation of the reconfigurable clause checker.