# A Hardware Relaxation Paradigm for Solving NP-Hard Problems

Paul Cockshott, Andreas Koltes, John O'Donnell, Patrick Prosser, Wim Vanderbauwhede
Department of Computing Science, University of Glasgow

**Abstract**

**Digital circuits with feedback loops can solve some instances of NP-hard problems by relaxation: the circuit will either oscillate or settle down to a stable state that represents a solution to the problem instance. This approach differs from using hardware accelerators to speed up the execution of deterministic algorithms, as it exploits stabilisation properties of circuits with feedback, and it allows a variety of hardware techniques that do not have counterparts in software. A feedback circuit that solves many instances of Boolean satisfiability problems is described, with experimental results from a preliminary simulation using a hardware accelerator.**

*Keywords: NP-hard problem, Boolean satisfiability, digital circuit with feedback, relaxation, simulated annealing*

## 1. INTRODUCTION

NP-complete problems lie on the boundary of what is economically computable. They are effectively computable on a Turing Machine, but their worst-case run times are believed to grow exponentially with problem size. This can make large instances of NP-complete problems too expensive for us to obtain solutions. It is suspected, but not proven, that no polynomial time algorithm exists for NP-complete problems, and that if a deterministic algorithm rather than an oracle is used to obtain the solution, then in the worst case the algorithm must perform an exhaustive search through a solution space whose size is an exponential function of the input size.

There have been numerous recent proposals to overcome the barrier of effective computability in computation, and proposals [5, 7, 12, 16] have been put forward for *hypercomputers* that could compute functions which are uncomputable on a Turing machine. The feasibility of building such devices remains in dispute [8, 26, 34, 9].

A related question concerns the time complexity of computable functions. Many models of computation are mathematical state machines that are provably equivalent to a Turing Machine, but some physical systems that can perform computation have not been proven to be Turing equivalent, either in terms of computability or time complexity. Do there exist physical systems that can solve computable problems with a lower time order than a Turing Machine?

We would be pleasantly surprised if such physical systems turn out to exist, but we are not holding our breath. Instead, we are investigating a particular class of physical system, not reducible to an algorithmic state machine, to determine its applicability to NP hard problem.

The particular type of physical computation system in question is a circuit comprising Boolean logic gates and (possibly) flip flops. Such circuits are normally designed according to a strongly disciplined synchronous style in order to keep their behaviour simple, digital, and predictable. Synchronous circuits behave like mathematical state machines. However, unconstrained Boolean networks with feedback can exhibit a variety of complex behaviours, including non-digital behaviour such as metastability [37]. Given constant inputs, a circuit may stabilise, it may settle down into an oscillation among a set of states, or it may fluctuate chaotically.

Kauffman has shown [21] that random Boolean networks of size $n$ have expected median state cycle lengths of $O(\sqrt{n})$. Thus a system with a very large state space (e.g. $2^{10000} \approx 10^{3000}$) may settle down and cycle among a quite small number of states (e.g. $100$).

In this paper we investigate the computational complexity of Boolean networks with feedback for solving instances of Boolean Satisfiability (SAT), a standard NP-complete problem. We show how to compile (in polynomial time) an instance of SAT into a circuit whose fixed point (where the signals remain stable) represents a solution to that problem instance. The circuit may not reach a fixed point; oscillation among a set of states constitutes a failure to solve the problem instance. Kauffman's result suggests that there

is a reasonable probability that such a circuit will indeed solve the instance. We have experimented with a prototype of the system, using FPGA technology to simulate the general class of circuit we define. Preliminary experimental results show that the approach does indeed solve many SAT problem instances quickly.

In Section 2 we consider the problem of Boolean satisfiability, and Section 3 reviews existing solvers. Section 4 outlines an ASIC (application-specific integrated circuit) design that can solve problem instances by relaxation, and we show how to compile an arbitrary instance of SAT in order to run on the circuit. Section 5 discusses initial results obtained by a hardware simulator, and Section 6 concludes.

## 2. THE PROBLEM DOMAIN: BOOLEAN SATISFIABILITY (SAT)

The problem domain we consider is Boolean satisfiability. Given an arbitrary Boolean expression over a set of variables, the problem is to determine whether there exists a set of variable settings (to true or false) that makes the entire expression true. A specific Boolean expression is called an *instance* of the general problem. We restrict the Boolean expressions to a canonical form: the logical conjunction of clauses, where each clause is the logical disjunction of one or more literals, and a literal is either a variable or the negation of a variable. This restricted version of Boolean satisfiability is called SAT, and it is also NP-complete. For example, the following expression is an instance of SAT:

$$(a \vee \neg d \vee e) \wedge (d \vee e \vee f) \wedge (b \vee \neg c \vee \neg d)$$

Although SAT is an NP complete problem, not all instances of it are hard to solve. Previous research has shown that the set of SAT problems has an interesting structure, with a phase change from a subset of problems with few solutions to a subset of problems with many solutions [20, 19, 38]. The instances of SAT that are hard lie mostly near the phase change. This previous research is experimental: large sets of problem instances are generated randomly and their solution times measured.

Cheeseman, Kanefsky and Taylor observed an abrupt phase transition from solubility to insolubility in graph colouring problems as average degree was increased [4]. A complexity peak was observed at this transition, and it was conjectured that this would be algorithm independent and common to all NP-complete problems. Graph colouring problems were mapped to SAT and the same phenomenon was observed, i.e. an abrupt phase transition with a corresponding complexity peak. Later studies showed that incomplete algorithms also experience the complexity peak when applied to satisfiable instances: easy solvable instances are easy, hard solvable instances are hard, and rare solvable instances found within the easy insolvable region are also easy. Much research has been done to pin down the location of the SAT phase transition and to develop theories about the location of this phase transition for problems that are NP-complete [14] or in higher complexity classes (such as quantifies SAT (QSAT)). Research to date appears to confirm that the complexity peak is indeed independent of the algorithm, and it is an open question whether physical systems that do not implement mathematical state machines have the same properties.

## 3. RELATED WORK ON SAT SOLVERS

Because of its theoretical interest and its practical importance, there has been extensive work on solvers for SAT.

### 3.1. Software solvers

There are two broad classes of SAT solvers: complete and incomplete. Complete solvers are guaranteed to find a solution if one exists and to terminate on unsatisfiable instances. They typically use a backtracking search based on the DPLL (Davis, Putnam, Logemann, and Loveland) algorithm. State of the art solvers, such as Zchaff2004 [25], MiniSAT [11, 13, 35] and BerkMin [15] employ relevance bounded learning, intelligent backjumping, and dynamic variable ordering heuristics along with smart data structures such as watched literals.

Incomplete solvers typically use a neighbourhood search algorithm, and often operate as hill climbers (or descenders). Given complete or partial setting of the variables, the settings are improved by making local changes. Solvers such as WalkSat [22] (and its predecessor GSAT) have features that are similar to Tabu search. Heuristics for optimisation strategies are discussed in [10], and runtime distributions of SAT solvers are reviewed in [17]. The algorithms for WalkSat and GSAT are shown below:

```
procedure WalkSat
   input f: array[1..c] of clauses {in CNF}
```

```
   output v:array[1..n] of boolean {a variable assignment that satisfies f}
begin
   for a := 1 to MaxTries do
      v := random truth assignment;
      for b := 1 to MaxFlips do
         if all f are true given v then return Success;
         choose a random clause cl in f such that cl=false;
         if random(0..1)<p then
            j := a random variable that appears in cl
         else
            j:= the variable in cl that will produce the biggest
                   increase in satisfied clauses when flipped
         v[j] := not v[j];
   return Fail;
end;

procedure GSAT
   input f: array[1..c] of clauses {in CNF}
   output v:array[1..n] of boolean {a  variable assignment that satisfies f}
begin
   for a := 1 to MaxTries do
      v := random truth assignment;
      for b := 1 to MaxTries do
         if all f are true given v then return Success;
         else
            PossFlips := set of vars which increase SAT most
            j := a random element of PossFlips
            v[j] := not v[j]
   return Fail;
end;
```

State of the art SAT solvers are highly optimised pieces of code. Practical applications of SAT solvers include scheduling problems, planning (for example, in interplanetary space within Deep Space 1), configuration problems, hardware design and verification, and cryptanalysis of hash functions. SAT instances solved to date contain some hundreds of thousands of variables and millions of clauses, typically taking a handful of hours to solve.

### 3.2. Hardware acceleration of SAT solvers with FPGAs

There has also been extensive work on using FPGAs to accelerate satisfiability algorithms. Many of these projects use FPGAs to accelerate components of the Davis-Putnam algorithm. Skliarova and Ferrari give a survey [33]; specific projects include [3] [39] [28] [1] [40] [2] [27] [32] [29] [36] [30] [41] [31].

Our approach differs from previous work in several key respects. It has an efficient polynomial time compilation of a problem instance onto the circuit; it uses relaxation rather than an algorithmic state machine to attempt to solve the instance; it uses a parallel randomised approach rather than the Davis-Putnam algorithm; it uses hardware techniques that have no counterpart in software, including pulse logic, asynchronous timing, and the use of noise to generate random numbers.

## 4. A HARDWARE RELAXATION PARADIGM FOR A FAST INCOMPLETE SOLVER

We now describe a new form of circuit that is capable of implementing an incomplete solver for an arbitrary instance of SAT, provided that the instance is not too large to fit on the chip. The circuit is structured as a programmable regular array of logic elements, related to but distinct from PLA, PAL, and FPGA logic, and it is suitable for implementation on an ASIC (application specific integrated circuit). In addition to the generic circuit, we describe a simple polynomial time method for compiling an arbitrary SAT instance to run on the circuit.

The approach is similar to simulated annealing [23] with a local potential energy function for each variable (Figure 1). The energy for the 0 or 1 states of a variable will be a function of the number of unsatisfied Boolean clauses in which the variable participates. Since the number of unsatisfied clauses depends on the states of other variables, the flipping of one variable will shift the energies of other variables.

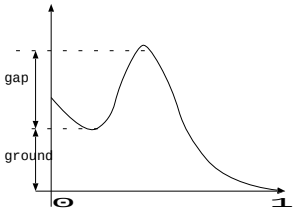


**FIGURE 1:** We can consider each variable to be subject to a local potential which varies according to whether the variable is true or false. To transition between Boolean values the variable has to use thermal noise to overcome a potential barrier separating the two states.

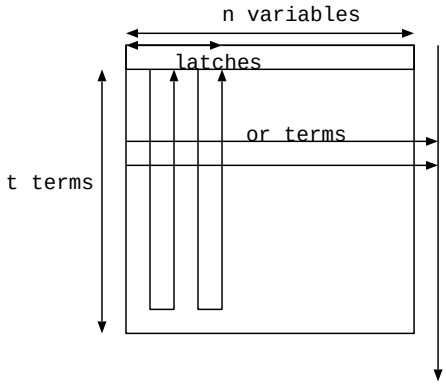


**FIGURE 2:** The layout is a regular two dimensional array with $t$ clauses (corresponding to the rows) and $n$ variables (corresponding to the columns)

A potential barrier separates the energies associated with the 0 and 1 states of a variable. At indeterminate moments, thermal noise will cause variables to flip state, and the probability that a flip will occur is an inverse function of the potential barrier. We can arrange the potentials so that the probability of a flip occurring to a variable will be zero if all the clauses which contain that variable are satisfied. Once all clauses have been satisfied, the system will be in a global energy minimum.

Our aim is to design an electronic circuit that can, in polynomial time, be configured to exhibit these dynamical properties for any SAT instance (up to some given size). Since chips are two dimensional and since SAT problems have two characteristic dimensions : $t$ Boolean clauses and $n$ variables, there is in principle a good match between the two. An obvious approach is to arrange the chip as an array with each of the $t$ clauses constituting a row and each of the $n$ variables a column (Figure 2). Each row must be able to represent an arbitrary Boolean clause that has to be satisfied.

In order to configure a row as a particular Boolean clause we select which variables participate in the clause, and also whether the variable is complemented. Therefore an arbitrary clause in $n$ variables could be encoded in $2n$ bits. Each of the $t$ clauses is represented by a shift register of length $2n$ bits. A simple option would be to concatenate these configuration shift registers into one long shift register of length $2nt$. Given a SAT problem instance in the form of a product of sums, then generating a two dimensional array of configuration bits can be computed in polynomial time on a standard computer. The computer can then shift the configuration array onto the chip, also in polynomial time.

The flip columns can be implemented as wired ORs, and ensure that the flip probability is an increasing function of the number of unsatisfied clauses into which a variable enters. This models our original requirement that the flip probability should be an inverse function of a potential barrier, which is itself an inverse function of the number of unsatisfied clauses using a variable. A $t$ input AND gate along one side of the chip can detect when all clauses are satisfied. Judicious design of the thermal noise source can mimic the effect of cooling as required by simulated annealing.

### 4.1. Structure of the programmable array circuit

Figure 3 gives an overview of the circuit. The current value of each of the Boolean variables is carried on two vertical lines (one giving the variable's value, the other its complement). There is a horizontal line that calculates the value of each clause (these are the horizontal lines that have $\times$ at some of the intersections, and which terminate at an or gate symbol). This line calculates the logical disjunction of the values carried by vertical lines that have a $\times$ at the intersection; if its value is true, then the clause is satisfied by the current

variable settings. That signal is inverted, producing a signal whose meaning is “this clause is *not* satisfied”, which is then transmitted to the left on a second horizontal line. If the “not satisfied” signal is true, one or more of the variables appearing in the clause must be wrong. The circuit labelled “?” controls the probability that one of these variables will be changed, and the result is carried up to the top of the circuit on another vertical signal. For example, the variable $a$ is carried downwards on two signals (one for $a$, one for $\neg a$, and the “$a$ may be wrong” signal $a^W$ is carried back up. The circuit labelled *fix* takes the current variable value, and flips it if the variable is “wrong”. The circuit is programmed to solve a specific instance of SAT by determining where the $\times$ and “?” connections are made. Those connections are controlled by flip flops, enabling the circuit to be reconfigured rapidly.

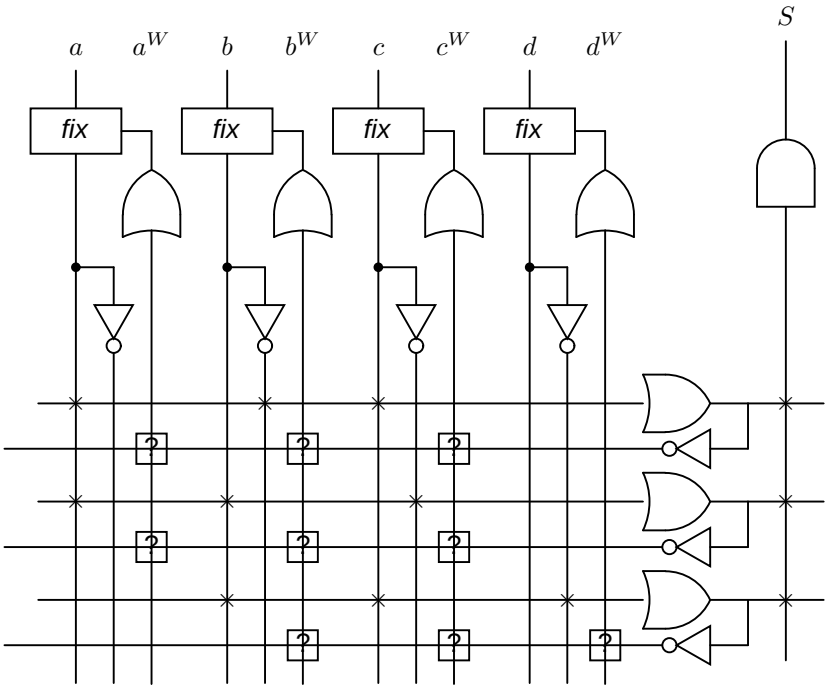


**FIGURE 3:** A circuit with feedback that attempts to satisfy $(a \vee \neg b \vee c) \wedge (a \vee b \vee \neg c) \wedge (b \vee c \vee \neg d)$. The circuit’s behaviour is parameterised by three black box circuits, labeled *fix*, $\times$, and ?. The output $S$ is 1 if the current values of the variables satisfy the expression. If the expression is not satisfied, the feedback loop changes values and continues trying. The behaviour of the circuit depends on the black box *fix* circuit.

Figure 4 shows in more detail a portion of a possible clause-line design, including the intersection between the horizontal clause-line and two variables A and B on vertical lines. A 4 bit register (shown as 1) selects whether each variable or its complement enters into the clause. This is organised as a shift register, so that the settings can be loaded efficiently into the chip. Each variable is represented by true (2) and complement (3) columns. Associated with each variable is a ‘flip’ column (5), which when activated will cause a set/reset latch at the top of the column to flip, changing the current state of the variable. Three input *and* gates (4) act to pull up the flip column if all of the following hold:

1. the clause (7) is currently unsatisfied;
2. the variable is selected as part of the clause by the shift register;
3. a thermal noise output (6) is true.

### 4.2. A tunable digital noise-based random generator

Rather than using traditional pseudo-random numbers, several hardware techniques are available to improve the efficiency of the randomisation; true random nomber generation via hardware was used as early as the 1940s (see [6] pp. 173–174). A train of random pulses can be used instead of random integers to control the toggling. The pulse train can be generated using noise, and probabilities can be combined using a logical and-gate. The generator must produce a spike train with a random delay between subsequent spikes, and the spikes must be wide enough to toggle a latch. The average delay between subsequent spikes (called the ”period”) must be controllable. Ideally, the period will grow exponentially longer over the duration of a 3SAT search. We propose to vary the supply voltage of the circuit over time along a negative exponential:

$$V_{DD} = V_{DD_{min}} + \left(V_{DD_{max}} - V_{DD_{min}}\right) \times e^{-\frac{t}{\tau}}$$

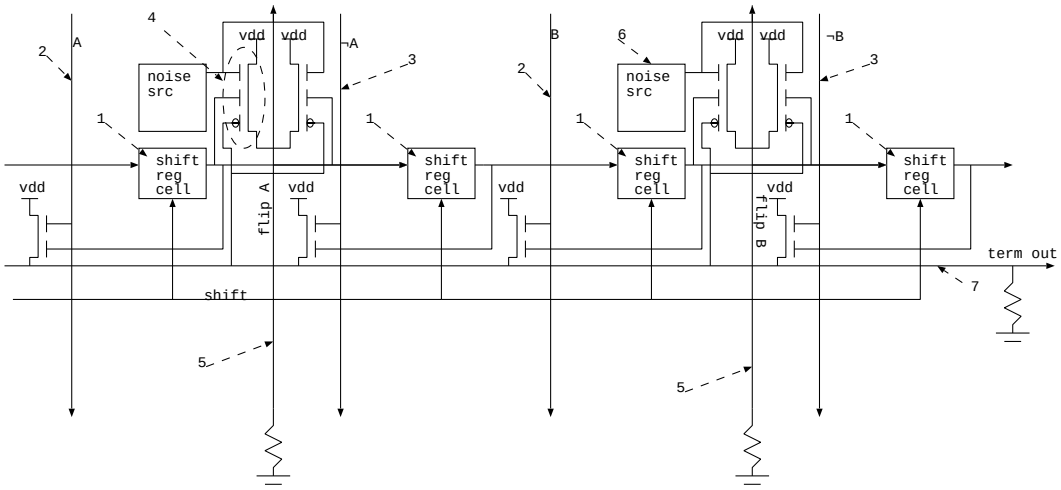


**FIGURE 4:** This diagram shows the intersection between two variables (labeled A,B) and a Boolean clause. Corresponding to each variable are two vertical lines for the true and complement values of the variable. Configuration information in shift register cells records whether the true or complement of the variables or neither are to be included in the clause. The clause is implemented by a wired OR. Noise anded with the value of the clause anded with the output of the configuration bit determines whether a vertical wired-OR causes the value of each variable to flip.

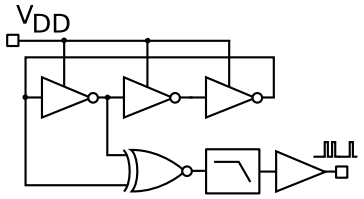


**FIGURE 5:** Generating random numbers using noise

where $\tau$ is the time constant of the system. This behaviour is easily obtained as the response of a step function to an RC filter. The actual random generator is based on a ring oscillator circuit. We exploit the well-known high-jitter behaviour of this type of oscillator to create random pulses, simply by XOR-ing two subsequent nets (Figure 5).

Because of the jitter, the XOR output will be a pulse of varying width, including zero-width. By low pass-filtering this signal and then recovering it, we obtain a random pulse train. The frequency of a ring oscillator is proportional to the VDD. An accurate model for the jitter of a ring oscillator is presented in [18]. Using this model it is possible to design a circuit that will generate a pulse with a probability p at a frequency governed by the VDD of the oscillator.

### 4.3. Algorithmic description of the hardware solver

The hardware solver can be executed in several modes: fully synchronous, asynchronous, or partially synchronous.

A fully synchronous version of the circuit would use a flip flop to hold the value of each variable, the flip flops would be clocked so that they change states simultaneously, and the clock would run slowly enough to allow the long paths through the logic array to settle down completely. This would cause the circuit to act as a large state machine, and its behaviour would correspond to an algorithm. However, it is costly to propagate a clock through a large circuit, and this approach would use a lot of chip area (reducing the size of problem instance that could be handled) and time (reducing the speed of the search).

A fully asynchronous version of the circuit would allow a variable to change any time a ‘wrong’ signal is received. Different variables would change their values at different times, and the horizontal lines could be calculating results based on variable values that are about to be toggled. The behaviour of the circuit may depend on infinitesimal variations in timing; indeed the results of running the circuit may not be repeatable.

There are also intermediate approaches, where the variables are clocked but the circuit is not completely synchronous.

If the hardware solver were to run in synchronous mode, its behaviour would correspond to a highly parallel randomised algorithm ProbSat:

```
procedure ProbSat;
   input f: array[1..c] of clauses {in CNF}
   output v: array[1..n] of boolean {a  variable assignment that satisfies f}
begin
   v:= random truth assignment;
   while true do
      if all f are true given v then return Success;
      parfor i in [1..n] do
         toggle v[i] with a probability proportional to the number of
                       unsatisfied clauses that contain the variable
end;
```

The hardware solver is an unbounded loop that performs a generate and test strategy. In practice, the circuit is stopped after a fixed number of clock cycles if it has failed to find a solution.

The hardware algorithm differs from WalkSat and GSAT in several respects:

- WalkSat and GSAT toggle a single variable at a time, while the circuit toggles many.
- WalkSat chooses the variable to toggle randomly from a set of clauses where that variable appears, while the circuit bases the decision to toggle a variable on the number of unsatisfied clauses it occurs in. This has something in common with GSAT.
- The hardware solver is highly parallel, speeding up the evaluation of the formula and the selection of variables to toggle.
- The hardware solver can be implemented asynchronously (the pseudo-algorithm shown above is synchronous). An asynchronous circuit may be faster, and it may find a solution more quickly.

## 5. EXPERIMENTAL RESULTS

We have completed a successful simulation of the hardware SAT solver using FPGA technology, using an Altera Cyclone chip [24]. An FPGA is a two-dimensional array of programmable components, which are connected by a programmable interconnection network. The circuits used were fully synchronous, but made essential use of randomisation (with a pseudo-random number generator, rather than truly random circuit noise). Thus the FPGA simulation constitutes an intermediate point between conventional synchronous circuits and fully asynchronous circuits. The FPGA simulation is highly parallel. As the circuit runs, it may stabilise with a solution for the problem instance, or it may oscillate indefinitely. Thus the circuit corresponds to an incomplete software solver, which may terminate or loop forever.

We developed a prototype compiler that reads an arbitrary instance of 3SAT, and then (in polynomial time) outputs a VHDL specification that describes a circuit specialised to solve that 3SAT instance. The circuit has the structure described above. Altera software tools compile the VHDL specification into the specific machine language programming needed for the Cyclone FPGA. The compilation and control of the FPGA are performed on a host PC.

After the circuit for a problem instance is loaded onto the FPGA, the chip is given a fixed interval of time to run. If it attains a fixed point within this time, the problem instance has been solved, and the exact solution time is measured by an accurate clocked counter on the FPGA. If the circuit does not reach a fixed point, the attempt is abandoned as a failure.

We performed a number of experiments using the FPGA simulation, to assess the performance of the system on solvable SAT instances. The experiments were carried out on a set of random satisfiable SAT instances. Each 3SAT instance contains 100 variables, which keeps the resulting circuits small enough to fit onto the Altera FPGA. The ratio of clauses to variables was chosen from 3.7 to 4.3 in steps of 0.1, with 100 instances for each ratio. These were generated as follows: (1) a set of random 3SAT instances was generated with variables appearing according to a uniform probability distribution; (2) unsuitable instances were removed from the set (i.e. instances where a variable appears several times in the same clause, or where some variables do not appear in any clause, or where the instance is easily partitionable); (3) the remaining instances were then checked using MiniSat, a complete SAT solver, and only solvable instances were retained.

The performance of the hardware solver was measured on 50 random solvable problem instances. Each instance was run 256 times, with different random seeds, in order to determine the distribution of run times

**TABLE 1:** Count of clock cycles for solution of an instance run 256 times

| | min | max | mean | std |
|---|---|---|---|---|
| *instance 019* | | | | |
| circuit | 27 | 3,452 | 795 | 665 |
| WalkSat | 53,184 | 991,656 | 237,057 | 142,231 |
| MiniSat | 286,816 | | | |
| *instance 007* | | | | |
| circuit | 1,542 | 5,553,701 | 1,567,975 | 1,596,822 |
| WalkSat | 77,392 | 9,926,096 | 1,452,258 | 1,444,226 |
| MiniSat | 4,623,352 | | | |
| *mean over 50 instances* | | | | |
| circuit | 3,979 | 963,500 | 277,825 | |
| WalkSat | 89,772 | 5,139,622 | 891,809 | |
| MiniSat | 2,335,202 | | | |

for a particular instance. This entire set of measurements was repeated for a variety of tuning factors. The performance was also measured on WalkSat (an incomplete software solver) and MiniSat (a complete software solver). Table 1 is an extract of the measurements (see Tables D.18 and D.23 in [24]).

The timing results are stated as the number of clock cycles executed. The circuit on the FPGA contains a hardware cycle counter, and the software solvers used CPU cycle counters. However, the clock speed is not the same on the FPGA chip and the CPU chip: the fastest FPGAs have slower clocks than CPUs, and the FPGA we used is an older and slower model, while a recent fast CPU was used. Therefore these measurements are only a guide to performance, and do not constitute real “wall clock time”. However, the ASIC circuit discussed in this paper would be significantly faster than even the fastest FPGAs.

Table 1 shows the performance results for a problem instance that happened to be easy, one that happened to be hard, and the mean over 50 instances. Each instance was run 256 times, with different initial random seeds. For each of these cases, the results are shown for the circuit running on the FPGA, for WalkSat, and for MiniSat. The most relevant comparisons are between the circuit and WalkSat; the MiniSat data are provided to indicate the effect on performance of a complete solver.

- *Easy problem: instance 019.* The fastest solution was obtained by the circuit in 27 clock cycles, but WalkSat needed 53,184 cycles for its fastest solution. The best case time for the circuit is 197 times faster than the best case time for WalkSat; for the worst case the circuit is 287 times faster, and for the mean over all 256 runs the circuit is 298 times faster.
- *Hard problem: instance 007.* For the best case the circuit was 50 times faster than WalkSat; for the worst case it was 91 times faster, but for the average over 256 runs the circuit was 1.08 times *slower*.
- *Averages over the complete set of 50 random solvable instances.* The minimum number of clock cycles required by the circuits, averaged over 50 instances, was 3,979; this is 22 times better than the corresponding figure for WalkSat. Considering the worst case times, the advantage of the circuit drops to a factor of 5.3, and for means over all 256 runs per instance, the circuit is 3.2 times faster.

These measurements indicate that the circuit is, in general, faster than WalkSat. It is much faster on problem instances that turn out to be easy, and such cases are common.

Large variances were observed in the runtime required by the hardware solver: for most intances, some of the runs finished after only a few hundred clock cycles, but some runs required a long time. The results suggest that, if the circuit does not settle down with a solution in a reasonable time, it is better to restart it with a new random seed rather than leaving it to continue.

The performance of the hardware solver depends strongly on the probability that a variable in an unsatisfied clause is toggled. When the probability is too high, the solver is unable to stabilise on a solution. As conjectured, the hardware solver demonstrates a phase change between easy and hard instances, as has already been observed for software solvers.

## 6. CONCLUSION

We have presented a design paradigm that exploits some of the capabilities of physical systems comprising networks of Boolean logic gates in order to solve instances of an NP-complete problem.

The feasibility of this approach has been demonstrated by parallel hardware simulation using FPGA technology. According to our preliminary results, the hardware solver gives good results on problem instances that are not too close to the phase change boundary. To achieve this, the solution requires randomisation to determine when variables are changed, and the performance is sensitive to the toggling probability. The best toggling probability depends slightly on the ratio of clauses to variables; there is not one fixed probability that is always best.

According to our preliminary results, the hardware solver produces more very short runs than WalkSat, so it is often faster, but for hard instances the hardware solver is often slower than WalkSat. This is likely caused by the ability of the hardware solver to toggle many variables in parallel during a single clock cycle, while WalkSat only flips one variable in each iteration. For problem instances that are relatively easy (i.e. which have a solution that is not near the phase boundary) the hardware solver is on average significantly faster, and there are many such cases.

FPGAs cannot achieve the full performance inherent in our technique. Their general interconnection networks and logic boxes require more area than the building block circuits outlined in this paper, yet they have higher latency. The commercial software for compiling arbitary circuits onto FPGAs is not guaranteed to be polynomial time, and in practice we have found it to be unacceptably slow. FPGAs do not offer the control over clocking, and the flexibility with feedback, that our circuits require. FPGAs do not allow some useful VLSI hardware techniques, such as true random number generation using noise. FPGAs are intended for general circuits, but are not well suited for the class of circuit described in this paper. For these reasons, future research will require the design of suitable programmable array circuits using ASIC (application-specific integrated circuit) technology.

**REFERENCES**


[1] Miron Abramovici and José T. de Sousa. A virtual logic algorithm for solving satisfiability problems using reconfigurable hardware. In *FCCM*, pages 306–307, 1999.

[2] Miron Abramovici and José T. de Sousa. A sat solver using reconfigurable hardware and virtual logic. *J. Autom. Reasoning*, 24(1/2):5–36, 2000.

[3] Miron Abramovici and Daniel G. Saab. Satisfiability on reconfigurable hardware. In *FPL*, pages 448–456, 1997.

[4] P. Cheeseman, B. Kanefsky, and W.M. Taylor. Where the REALLY Hard Problems Are. In *Twelfth International Joint Conference on Artificial Intelligence (IJCAI-91)*, 1991.

[5] B. J. Copeland. Hypercomputation. *Minds and Machines*, 12:461–502, 2002.

[6] B. Jack Copeland. *Colossus: The Secrets of Bletchley Park's Codebreaking Computers*. Oxford University Press, 2006. ISBN 0-19-284055-X.

[7] J. Copeland and R. Sylvan. Beyond the universal Turing machine. *Australasian Journal of Philosophy*, 77(1):46..66, 1999.

[8] Paolo Cotogno. Hypercomputation and the physical church-turing thesis. *Brit. J. Phil. Sci.*, 54:181–223, 2003.

[9] Martin Davis. The Church-Turing Thesis: Consensus and Opposition. In A. Beckmann, U. Berger, B. Lowe, and J. V. Tucker, editors, *Logical Approaches to Computational Barriers: Second Conference on Computability in Europe, CiE 2006, Swansea, UK*, number 3988 in LNCS, pages 125–132. Springer, June/July 2006.

[10] Nachum Dershowitz, Ziyad Hanna, and Alexander Nadel. A clause-based heuristic for sat solvers. Technical report, School of Computer Science, Tel Aviv University, 2005.

[11] N. Een and N. Sorensson. An extensible SAT-solver. In *SAT 2003 (LNCS 2919)*, pages 502–518, 2003.

[12] G. Etesi and I. Németi. Non-Turing Computations Via Malament–Hogarth Space-Times. *International Journal of Theoretical Physics*, 41(2):341–370, 2002.

[13] Niklas En and Niklas Srensson. Translating pseudo-boolean constraints into sat. *Journal on Satisfiability, Boolean Modelling and Computation*, 2:1–25, 2006.

[14] I.P. Gent, E. MacIntyre, P. Prosser, and T. Walsh. The constrainedness of search. In *AAAI-96*, pages 246–252, 1996.

[15] E. Goldberg and Y. Novikov. BerkMin: a Fast and Robust SAT-Solver. In *Design Automation and Test in Europe*, pages 142–149, 2002.

[16] D.Q. Goldin, S.A. Smolka, P.C. Attie, and E.L. Sonderegger. Turing machines, transition systems, and interaction. *Information and Computation*, 194(2):101–128, 2004.

[17] Carla P. Gomes, Bart Selman, Nuno Crato, and Henry Kautz. Heavy-tailed phenomena in satisfiability and constraint satisfaction problems. *Journal of Automated Reasoning*, 24:67–100, 2000.

[18] Ali Hajimiri, Sotirios Limotyrakis, and Thomas H. Lee. Jitter and phase noise in ring oscillators. *IEEE Journal of Solid-state Circuits*, 34(6):790–804, June 1999.

[19] Brian Hayes. Can't get no satisfaction. *American Scientist*, 85(2):108–112, March–April 1997.

[20] Brian Hayes. On the threshold. *American Scientist*, 91(1):12–17, January–February 2003.
[21] Stuart A. Kauffman. *The Origins of Order*. Oxford University Press, 1993.
[22] H. Kautz and B. Selman. Pushing the envelope: planning, propositional logic, and stochastic search. In *Thirteenth National Conference on Artificial Intelligence (AAAI'96)*, pages 1194–1201, 1996.
[23] S. Kirkpatrick and CD Gelatt Jr. MP Vecchi Optimization by Simulated Annealing. *Science*, 220(4598):671680, 1983.
[24] Andreas Koltes. Solving NP-complete problems in hardware. Technical report, University of Glasgow, 2007. Available on www.dcs.gla.ac.uk/∼jtod/satcircuit/.
[25] Y.S. Mahajan, Z. Fu, and S. Malik. Zchaff2004: An Efficient SAT solver. In *SAT 2004: Theory and Applications of Satisfiability Testing (LNCS 3542)*, 2004.
[26] G. Michaelson and P. Cockshott. Constraints on Hypercomputation. In A. Beckmann, U. Berger, B. Lowe, and J. V. Tucker, editors, *Logical Approaches to Computational Barriers: Second Conference on Computability in Europe, CiE 2006, Swansea, UK*, number 3988 in LNCS, pages 378–387. Springer, June/July 2006.
[27] Marco Platzner. Reconfigurable accelerators for combinatorial problems. *IEEE Computer*, 33(4):58–60, 2000.
[28] Marco Platzner and Giovanni De Micheli. Acceleration of satisfiability algorithms by reconfigurable hardware. In *FPL*, pages 69–78, 1998.
[29] N. A. Reis and José T. de Sousa. On implementing a configware/software sat solver. In *FCCM*, pages 282–283, 2002.
[30] Mona Safar, M. Watheq El-Kharashi, and Ashraf Salem. Fpga based accelerator for 3-sat conflict analysis in sat solvers. In *CHARME*, pages 384–387, 2005.
[31] Mona Safar, Mohamed Shalan, M. Watheq El-Kharashi, and Ashraf Salem. Interactive presentation: A shift register based clause evaluator for reconfigurable sat solver. In *DATE*, pages 153–158, 2007.
[32] Iouliia Skliarova and António de Brito Ferrari. Design and implementation of reconfigurable processor for problems of combinatorial computations. In *DSD*, pages 112–119, 2001.
[33] Iouliia Skliarova and António de Brito Ferrari. Reconfigurable hardware sat solvers: A survey of systems. *IEEE Trans. Computers*, 53(11):1449–1461, 2004.
[34] Warren D. Smith. Three counterexamples refuting Kieu's plan for "quantum adiabatic hypercomputation" and some uncomputable quantum mechanical tasks. *J.Applied Mathematics and Computation*, 187(1):184–193, 2006.
[35] Niklas Srensson and Niklas En. Minisat v1.13 - a sat solver with conflict-clause minimization. Technical report, Chalmers University of Technology, Sweden, 2005.
[36] C. J. Tavares, C. Bungardean, G. M. Matos, and José T. de Sousa. Solving sat with a context-switching virtual clause pipeline and an fpga embedded processor. In *FPL*, pages 344–353, 2004.
[37] John F. Wakerly. *Digital Design: Principles and Practices*. Prentice Hall, 1990.
[38] Toby Walsh. 2+p-col. Technical report, Cork Constraint Computation Center, University College Cork, 2002.
[39] Peixin Zhong, Pranav Ashar, Sharad Malik, and Margaret Martonosi. Using reconfigurable computing techniques to accelerate problems in the cad domain: A case study with boolean satisfiability. In *DAC*, pages 194–199, 1998.
[40] Peixin Zhong, Margaret Martonosi, Pranav Ashar, and Sharad Malik. Using configurable computing to accelerate boolean satisfiability. *IEEE Trans. on CAD of Integrated Circuits and Systems*, 18(6):861–868, 1999.
[41] Romanelli Lodron Zuim, José T. de Sousa, and Claudionor José Nunes Coelho Jr. A fast sat solver strategy based on negated clauses. In *VLSI-SoC*, pages 110–115, 2006.