AUTOMATIC IN-FLIGHT REPAIR OF FPGA COSMIC RAY DAMAGE

Sarah THOMPSON, **Alan MYCROFT**

Computer Laboratory, University of Cambridge, William Gates Building, JJ Thomson Avenue, Cambridge, CB3 0FD, UK

Guillaume BRAT

Research Institute for Advanced Computer Science, NASA Ames Research Center, Mail Stop 269-2, Moffett Field, CA 94035-1000

Arnaud VENET

Kestrel Technology LLC, 3260 Hillview Avenue, Palo Alto, CA 94304

ABSTRACT:

FPGAs are finding an increasing number of applications within NASA in deep space probes, planetary rovers and manned vehicles. Like other silicon devices, FPGAs can be damaged by high energy cosmic ray impacts, resulting in permanent latch-up conditions that manifest as 'stuck-at' faults. Traditionally, multiple redundancy and voting logic have been employed as a work-around, particularly for high reliability, extreme environment applications. However, reconfigurable FPGAs are becoming increasingly common in flight systems, offering a potentially valuable possibility for improved levels of fault recovery – after a fault is detected and localised within an FPGA, it is feasible to reprogram the device, in flight, with an alternative, equivalent, circuit that does not depend upon the damaged portion of the chip.

Designing such alternative chip layouts by hand is a valid option, though costly in terms of the man-hours of effort required; a fully automated alternative would be far preferable. In this paper, a technique is presented that allows the automatic generation of FPGA configurations for fault recovery purposes by means of non-clausal SAT solver technology.

1 INTRODUCTION

Designing hardware capable of reliable operation in deep space is far from trivial. The familiar, tried and trusted design techniques employed by engineers working on conventional, ground-based electronics are not sufficient to ensure reliability in the extreme environment of deep space. Radiation, extreme temperatures, hard vacuum and many other challenges must be addressed whilst accommodating a requirement for extremely high reliability – deep space probes typically must operate for decades, with no possibility of servicing by astronauts if anything goes badly awry.

Inherently radiation hard semiconductor devices do exist, though they carry a very significant cost penalty, as well as generally requiring more power in return for less performance in comparison with commercially available off-the-shelf (COTS) devices. A common radiation hardening design approach involves taking an existing COTS standard cell design, then synthesising a new version where some or all of the original gates and flip flops are replaced with more complex, internally redundant equivalents.

The widely used RAD6000 processor was created by replacing the standard cells of the original IBM RS/6000 design with hardened versions, resulting in a processor with greatly improved radiation hardness with respect to the original. Such chips are more radiation resistant than the COTS equivalent, but are slower, require more power and are typically extremely costly ($100k per device is not unusual) due to the need to amortise foundry set-up costs over a relatively small number of saleable devices. Designers operating within contemporary budgetary constraints often therefore prefer to use COTS devices where possible, reserving extremely expensive radiation hard components for critical subsystems only. For example, a mission critical guidance system might be implemented with radiation hardened chips, but a less critical instrument package might use COTS components instead, achieving a significant cost, mass and power saving as well as allowing higher clock rates.

1.1 FPGAs in Space

The Apollo programme at it height consumed more than half the world's entire chip manufacturing capacity, comprising many custom-built ASICs. Modern spacecraft, however, are designed within budgetary constrains that mean that full custom ASICs are far too expensive to be considered. Nevertheless, mass limitations[1] still mean that custom chips are necessary. Field programmable gate arrays (FPGAs) offer a good compromise; though less efficient than full-custom ASICs in terms of density and power consumption, they nevertheless offer a means by which custom chips can be incorporated into designs without incurring the huge (approximately US$2 million per iteration) fabrication costs of full-custom devices. FPGAs typically contain a large array of general purpose logic that only 'becomes' the target circuit after an appropriate configuration bit stream is uploaded. In some FPGA families, particularly those manufactured by Actel, programming is carried out once only, after manufacture but typically before the chip is incorporated into a board-level system. Other families, particularly those manufactured by Xilinx and Altera, hold their bit stream in static RAM, thereby making it possible to reconfigure such FPGAs dynamically.

As with any other semiconductor device, FPGAs are susceptible to radiation effects including single-event upsets (SEUs) and permanent latch-up faults. Radiation hard FPGAs are commercially available[2], though they tend to have lower density, lower performance and significantly higher cost than commercial grade devices. At the time of writing, both approaches are in use in ongoing missions – the Galileo/Huygens spacecraft incorporates a number of Actel radiation hardened FPGAs, whereas the Mars Exploration Rover mission's twin rovers, Spirit and Opportunity, depend on COTS devices sourced from Xilinx.

1.2 Radiation Damage

Radiation levels in space vary widely; in low earth orbit, levels can be sufficiently low that conventional electronics can be used unmodified[3]. As spacecraft venture outside the protective effects of the Earth's magnetic field, radiation levels increase both in terms of the frequency and energy of particle impacts.

Fig. 1 shows the effect of a heavy ion moving at a relativistic velocity (cosmic ray) passing through the gate of a field effect transistor in a typical gate. The ion leaves a trail of charge that transiently affects the operation of the transistor, which may manifest as an unwanted voltage spike in the circuit. In many cases, such spikes are benign and do not cause circuit behaviour to deviate from specification. Often, however, such a spike, often referred to as a Single Event Upset (SEU), may cause a circuit to enter an invalid state. Normally, such conditions are detected by watchdog circuits and are cleared by simply resetting the malfunctioning subsystem.

Sufficiently high energy particle impacts can cause permanent damage. Often referred to as permanent latch-up, such damage manifests as signals ceasing to function correctly and appearing to be stuck permanently at logic *true* or *false.* Such damage can not be cleared by a reset, so some form of redundancy is required in order for the subsystem to continue to function.

[1] Largely due to launch costs of the order of approximately $30,000 per kg to low earth orbit.
[2] See also `http://www.actel.com/`
[3] On the International Space Station (ISS), many computing tasks are carried out by COTS laptop PCs.

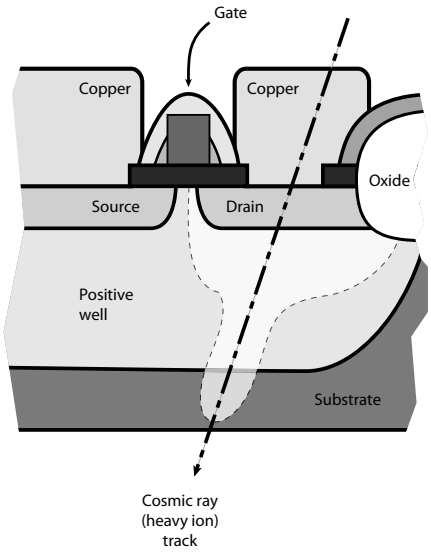

Figure 1: SEU triggered by a cosmic ray impact

1.3 MODULAR REDUNDANCY

Traditionally, modular redundancy has been the standard approach toward mitigating the effects of permanent latch-up. In this approach, majority voting logic [19] allows the incorrect output of one or more faulty subsystems to be ignored. In 3-way modular redundancy (see Figs. 2 and 3), any two subsystems can override the output of the third, allowing one subsystem to fail completely without affecting system level behaviour. 5-way modular redundancy, as employed by the Shuttle main computers, allows up to two subsystems to fail without affecting functionality.

Modular redundancy is certainly effective, but its requirement for duplication of subsystems carries a significant mass and power consumption penalty. Whilst it is likely to remain a requirement for critical subsystems, its cost precludes its universal applicability.

1.4 EXPLOITING REDUNDANCY WITHIN FPGAS

For practical reasons, most FPGA layouts are typically restricted to using no more than approximately 60 – 80% of the chip's theoretical optimal capacity. FPGA layout is thought to be an NP-complete problem, though good heuristics exist that can do a reasonable job of automatically mapping designs to configuration bit streams. These algorithms tend to reach a solution much faster when the design can be mapped to a relatively small proportion of the chip's resources, and can fail to generate a layout

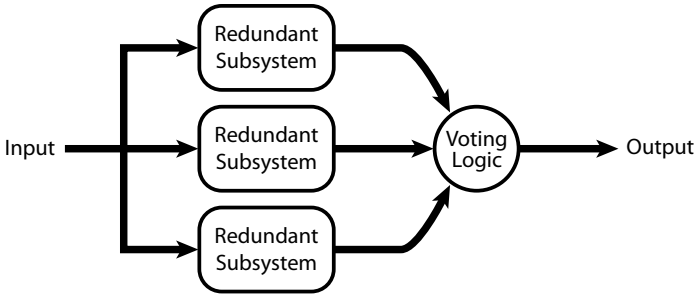

Figure 2: Modular Redundancy

completely in cases where the proportion is close to 100%. As a consequence of this, almost all practical FPGA layouts contain a significant amount of unused resources – though FPGA circuits are not usually in and of themselves redundant, spare FPGA logic capacity unused by the circuit can nevertheless be exploited in order to improve reliability.

A tempting possibility would be resynthesising logic locally within small areas of the chip, adding redundancy to the circuit until the chip is completely full. This approach, however, would incur a power and performance penalty, whilst adding redundancy to circuits in an unpredictable way, without any guarantee that the resulting layout would in practice survive any particular fault.

A more practical approach is to lay out the FPGA conventionally, then *locally resynthesise* logic around faults as and when they are detected (see Section 2.4). Having spare capacity in terms of unused logic blocks and wiring resources spread across the chip layout makes it feasible to consider only a small area near the fault, avoiding the need to generate a complete new layout from scratch. In outline, this approach may be summarised as follows (see also Fig. 4):

1. FPGA running normally (Fig. 4.i)
2. Fault detected (Fig. 4.ii)
3. Take FPGA off line and put it through a test procedure in order to localise the fault or faults
4. Locally resynthesise logic around each fault, resulting in a working, work-around layout
5. Upload new configuration bit stream to FPGA
6. Put chip back on line (Fig. 4.iii)

Several alternatives are possible as regards the implementation of local resynthesis. Most obvious is perhaps re-running the software responsible for the original FPGA layout again with appropriate constraints preventing it from using damaged parts of the chip – whilst technically feasible, this approach

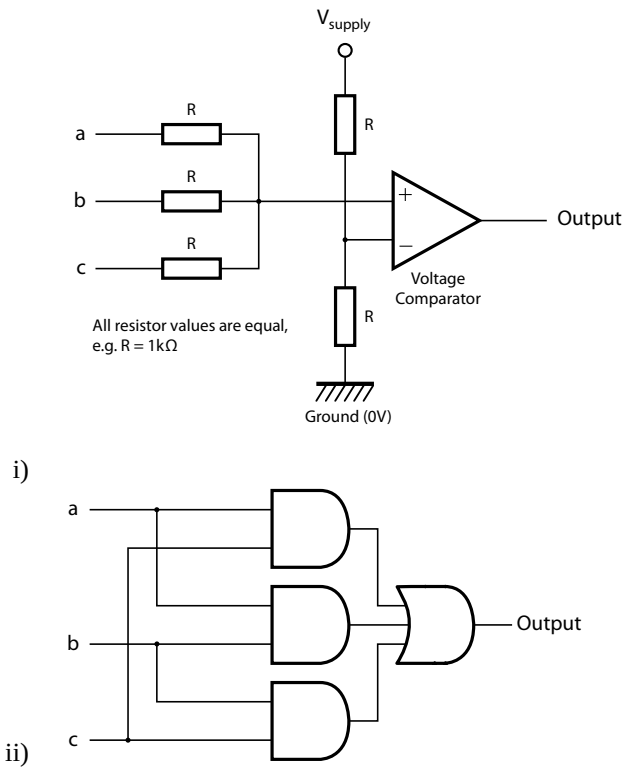

Figure 3: Typical Majority Voting Logic Implementations: i. Analogue, ii. Digital

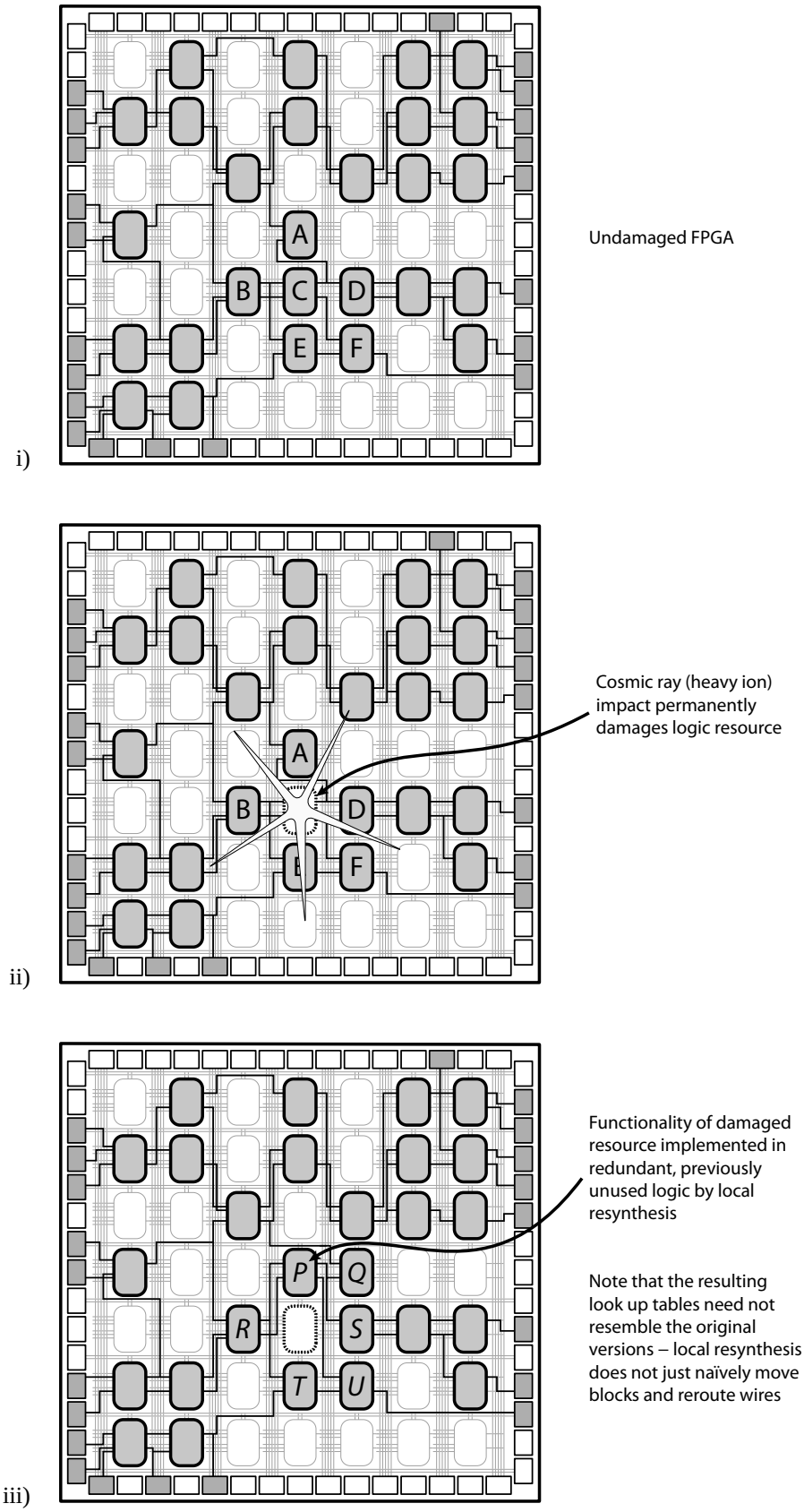

Figure 4: Using available FPGA resources to work around permanent latch-up damage

is not well suited to automated in-flight use, since the software required typically assumes a powerful workstation class computer, often with some human intervention.

Jason Lohn's group at the NASA Ames Research Centre [8, 7] have experimented with automatically generating FPGA layouts with genetic algorithms. A population of random FPGA bit streams are tested, with their behaviour compared with ideal test traces derived from the original circuit. Over many generations, functionality tends to converge on the desired circuit, even though no formal link other than observed behavior exists between the original design and the generated design. Good results have been achieved on a number of test circuits, but the difficulty of proving that a generated circuit that includes flip flops really does implement the intended behaviour (as opposed to just happening to respond correctly to a non-exhaustive set of tests) is likely to limit the technique's applicability.

1.5 AVAILABILITY

An FPGA undergoing repair will, of necessity, not be able to continue performing its intended function during the repair process. As a consequence, our technique will not be suitable for applications requiring high availability unless the FPGA is itself part of a modular redundant subsystem. In such situations, the ability to repair faulty subsystems is still a significant advantage, because it allows redundancy to be maintained over far longer periods.

1.6 LOCAL RESYNTHESIS AS A SAT PROBLEM

In this paper, we describe a technique that can automatically perform local resynthesis whilst retaining functionality that is formally identical to that of the original circuit. In essence, formally correct local resynthesis requires an alternative, work-around bit stream to be determined such that for all possible inputs and/or internal states, the outputs and next internal state of the work-around circuit matches exactly that of the original circuit. Finding such configurations is computationally hard, perhaps prompting the adoption by Lohn's group of heuristic search algorithms that do not attempt to ensure formal correctness.

In the remainder of this paper, we demonstrate how local resynthesis can be transformed into a equivalent SAT problem [3, 2, 1], thereby demonstrating that local resynthesis is no harder than NP-complete[4]. The resulting SAT problems are suitable for attack by SAT solvers, with solutions guaranteed to preserve correctness with respect to the original circuit.

2 DEFINING THE SAT PROBLEM

Given an original, correct, bit stream b along with a model of a correct FPGA f, a work-around bit stream b' for a faulty FPGA f' must possess the following property:

$$\forall i \;.\; f(b, i) \Leftrightarrow f'(b', i)$$

Informally, this states that for all possible inputs i, the bit stream b' causes the damaged FPGA to behave exactly identically to the original FPGA and bit stream (see also Fig. 5). Letting b' represent any potential work-around bit stream, this expression will evaluate to *true* if and only if correct functionality is preserved – in effect, the expression embodies formal verification of a work-around bit stream with respect to an original bit stream[5]. Alternatively, the expression may be thought of as defining a Boolean satisfiability problem whose solutions represent all possible work-around bit streams – solving such a SAT problem is therefore equivalent to the local resynthesis problem.

After constant propagation, quantifier elimination and (if necessary) transformation to CNF or NNF form, feeding the resulting expression to a SAT solver allows b' to be calculated.

It is noteworthy that no inherently complex conventional chip layout, placement or routing algorithms are required, suggesting that this functionality might be implemented within embedded systems carried

[4]We conjecture (assuming $P \neq NP$) that no complete P space/time algorithm exists, though such speculation is beyond the scope of this paper.

[5]Note that this approach may be used to verify the correctness of *any* work around bit stream, including those generated by genetic algorithms or by other means.

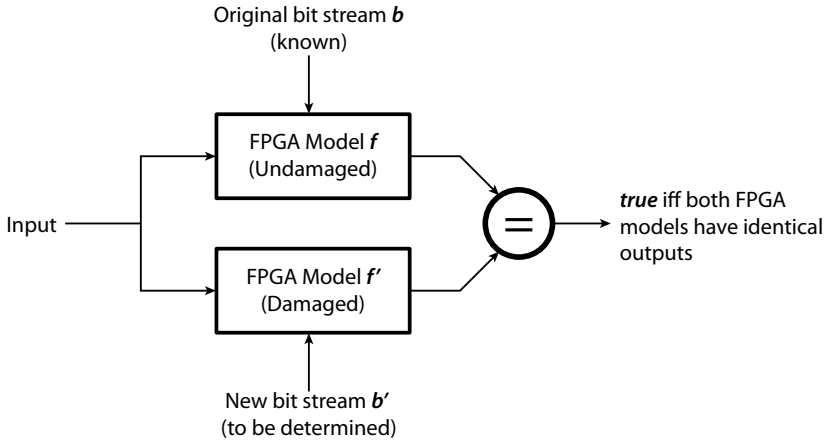

Figure 5: FPGA Repair as a SAT Problem

on the spacecraft itself. Typical SAT solver memory requirements are generally not particularly severe for the kinds of problem we consider, requiring approximately 3MB for the circuit shown in Fig. 6. Run times are of the order of tens of seconds on contemporary CPUs (see Section 3, Fig. 7).

The SAT problems that result from this process are typically quite *hard*, in the sense that standard SAT solvers do not typically find solutions very quickly. Empirically, *non-clausal* SAT solvers (i.e. those that do not require their formulas to be converted to CNF form) appear to be most effective, possibly because they allow circuits to be modeled in a form that is closer to their original structure.

2.1 QUANTIFIER ELIMINATION

SAT solvers typically do not directly support quantifiers, so the first step involves eliminating them from the expression. Removing the universal quantifier $\forall i$ is therefore essential. Since i may consist of several Boolean variables, it is helpful to (equivalently) express the problem as

$$\forall i_1 \; \forall i_2 \; \ldots \forall i_n \; . \; f(b, i) \Leftrightarrow f'(b', i)$$

We can now eliminate these quantifiers one by one by applying the rewrite rule

$$\forall a \; . \; F(a) \quad \longrightarrow \quad F(1) \wedge F(0)$$

repeatedly until none remain. Since this operation has an expression size and space upper bound of 2^N, this restricts our technique's applicability to fairly small sub-circuits, though this is less significant when slicing techniques are adopted (see Section 2.2).

After constant folding and common subexpression elimination, the resulting expression is a directed acyclic graph, with exactly one 'output' node representing the result of the expression, and one 'input' node for each bit in b'. The variables i and b are no longer externally exposed, with the resulting expression depending only upon b'. At this point, the expression may be passed to a suitable SAT solver, e.g. NNF-WALKSAT, as described in Appendix A.

2.2 SLICING

Attempting to resynthesise a complete FPGA is infeasible with our method due to the tendency for the size of the SAT problem to be proportional to 2^N, where N is the total number of inputs and flip flop outputs (see Section 2.3). It is therefore necessary to work on a small *slice* of the chip. The rationale behind this approach is that, whilst a cosmic ray impact might render the original circuit useless, many possible work-around bit streams with low Hamming distance from the original bit stream typically exist, differing only near the damage site. Several variant approaches are feasible:

1. *Slicing by Coordinate.* In this case, a slice is chosen such that inclusion is based on physical distance (in terms of the 2D chip layout) from the damage site.

2. *Slicing by Connectivity.* Such a slice might be generated by beginning at the damage site and including all bit stream bits that are electrically reachable through a predetermined number of logic blocks.

3. *Slicing by Heuristic.* In this case, a slice might be generated by some device-specific algorithm capable of exploiting aspects of its design in order to create a more effective slice than either of the above simpler approaches.

It is possible that, in some cases, no local solution may exist, but solutions that differ more significantly may still be possible. The probability that this might occur can be reduced by arranging the original design such that used resources are spread evenly across the chip rather than clustered together, but in extreme cases the fall back option still exists of creating an alternative layout manually (e.g. remotely on Earth). Our experimental results suggest that local solutions are possible in most cases, however.

2.3 HANDLING FLIP FLOPS

The technique presented here essentially considers combinational circuits; clocked synchronous circuits may be accommodated by a small modification:

1. If a working flip-flop necessary to implement the original circuit falls within the slice under repair, treat its output as if it was an external *input* of the subcircuit. Similarly, treat its input as an *output* of the subcircuit.

2. If a damaged flip-flop necessary to implement the original circuit falls within the slice, exclude its connections from the slice and substitute an alternative, working flip flop. Local resynthesis will take advantage of the alternative flip-flop and avoid the damaged original.

2.4 DETECTION AND LOCALISATION OF FAULTS

It is envisaged that faults will initially be detected as a consequence of observably incorrect behaviour of a subsystem implemented on an FPGA. Well known techniques already exist, such as watchdog circuits, suicide/fratricide logic, etc. In a practical implementation, when incorrect behaviour is detected, an embedded processor[6] will be triggered to begin a repair cycle.

Initially, the fault will only be known to exist somewhere within a particular chip, but gate-level fault information is required in order to allow a work around bit stream to be generated. Most FPGAs support in-circuit testing via the industry standard JTAG interface – this typically allows all flip flops to be temporarily reconnected as a single shift register, allowing the internal state of the chip to be uploaded or downloaded. Assuming that the chip is not so badly damaged that its JTAG interface no longer functions, uploading a series of test vectors and examining their results potentially allows faults to be localised with considerable accuracy. Such testing procedures are ubiquitously employed by automated test equipment during chip manufacture, so this requirement is unlikely to be prohibitive.

3 EXPERIMENTAL RESULTS

As a proof-of-concept, a small, FPGA-like circuit was modeled (see Fig. 6). Eight inputs, split into two groups of four, feed the inputs of four 16-bit look-up tables, whose outputs feed a fifth 16-bit look up table. The model was configured by randomly generated 'bit streams', each 80 bits long, mapping to the configuring bits of the look up tables. Stuck-at faults were simulated by fixing the values of one or more bits at 0 or 1. For simplicity, fixed wiring was assumed. A non-clausal variant of the WALKSAT algorithm [11] (see Appendix A) was used to solve the resulting SAT problems.

[6] This could either be an on-chip CPU or an external, possibly radiation hardened, general purpose processor.

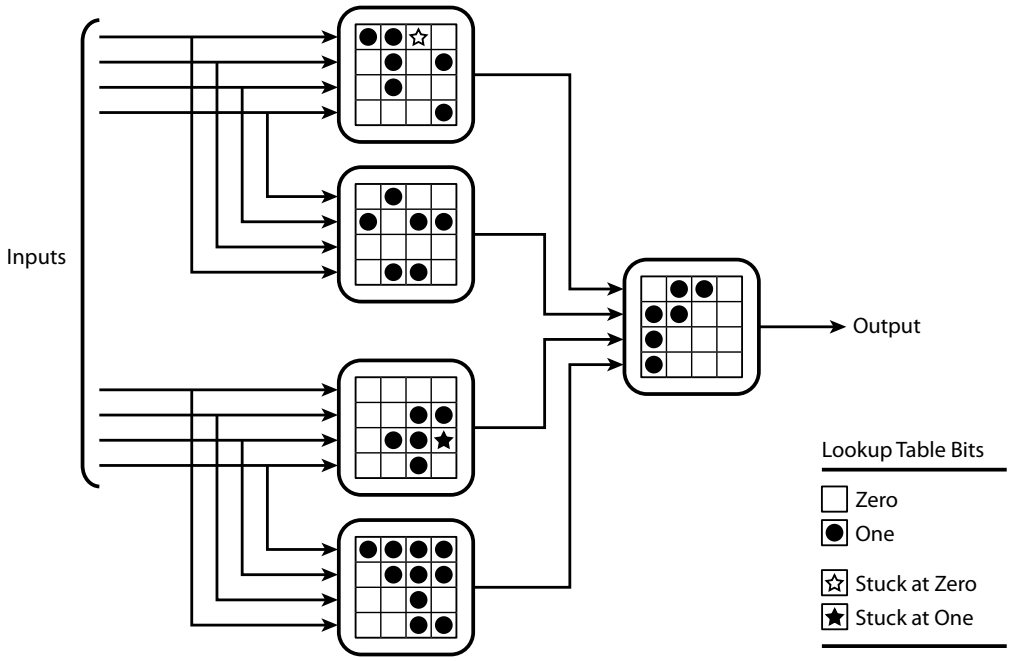

Figure 6: Example Test Circuit Model

In our experiments, the SAT problems were generated by a modified version of the HarPE hardware partial evaluator [16]. HarPE is a C++ template library [18] that allows circuits to be described in a high-level language, then manipulated by partial evaluation [5, 10, 9]. The library was extended slightly to allow its output to be represented in the format necessary for typical SAT solvers.

Test runs were repeated with between 1 and 6 simulated faults. Run times (C++, `gcc -O3`, running on a 1.6GHz Pentium III) and success rate are shown in Fig. 7, where 'success' was defined empirically as the SAT solver finding a solution within 20 minutes[7].

4 RELATED WORK

The original concept of generating FPGA bit streams with SAT solvers is due to David Greaves at the Computer Laboratory, University of Cambridge [4].

[7]Note that no attempt was made to verify whether the generated problems were actually soluble – this corresponds well to reality, in that some damage sites in critical positions may not allow any possible work around configuration to be determined.

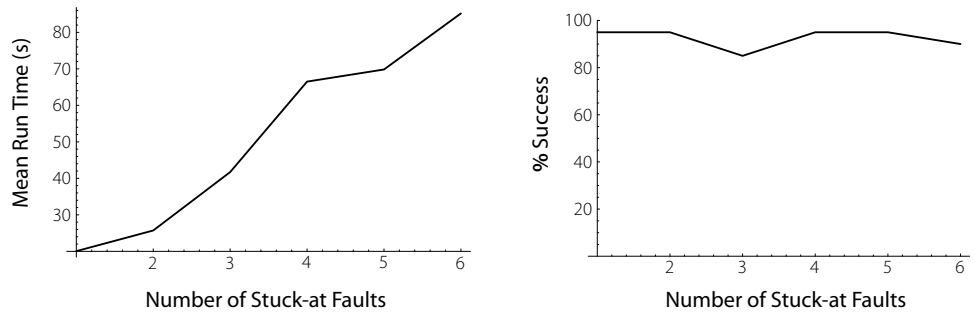

Figure 7: Test Results

The Dynamic Evolution for Fault Tolerance (ITSR/ES) project headed by Jason Lohn at the NASA Ames Research Centre is applying genetic algorithms to FPGA repair [8, 7, 6]. This approach has been shown to work, but suffers from the problem that its generated circuits are not guaranteed to be formally equivalent to the original.

Adrian Stoica's group at JPL is working on the synthesis and repair of analogue field programmable transistor array (FPTA) devices with genetic algorithms [13].

Toby Walsh's group at the School of Computer Science and Engineering, University of New South Wales, Australia are working on non-clausal SAT solvers, one of which, NOCLAUSE, is due to be released into the public domain shortly [14].

There is a huge amount of literature on the subject of SAT, particularly with regard to resolution of Boolean expressions in CNF form. The web site `http://www.satlive.org/` is a widely-used and very useful resource for information about SAT/QBF solvers.

5 CONCLUSIONS

In this paper, we have described a method for restating the problem of finding an alternative work-around FPGA configuration as a Boolean SAT problem, along with modifications to the WALKSAT algorithm that allows solutions to the resulting problems to be found efficiently. The feasibility of using SAT solvers for FPGA repair has been empirically demonstrated.

6 FUTURE WORK

The approach described in this paper assumes an underlying clocked synchronous model. We hope to apply similar techniques to the synthesis and manipulation of a wider class of circuits whose dynamic characteristics are critical, e.g. self-timed circuits and globally asynchronous locally synchronous (GALS) circuits.

In recent papers [17, 15], we describe a multi-value logic that is capable of reasoning about asynchronous circuits, and also about such circuits' behaviour in response to SEUs and permanent latch up faults. An approach, similar to that described in this paper, but using our more accurate logics may make it feasible to automatically repair FPGA-based circuits whose asynchronous behaviour is more critical than those relying upon the synchronous model assumed here.

Our finding that non-clausal SAT solvers appear to work better for FPGA synthesis has also been noted by Greaves [4]. Finding out exactly why this is the case may be useful both within our own problem domain and also in the wider SAT solver community.

The Boolean SAT expression necessary for local resynthesis can also be used to check the validity of solutions that have been arrived at by other means, including those generated by genetic algorithms, so it is possible that a combined approach may offer further benefits.

7 ACKNOWLEDGEMENTS

This work was supported by the NASA Ames Summer Internship programme, managed by MCT/QSS for NASA during the summer of 2004. Discussions with members of the Automated Software Engineering group at Ames, particularly Arnaud Venet, Guillaume Brat, Corina Pasarenau and Mike Lowry, and with the Jason Lohn and Gregory Larchev of the Genetic Algorithms group were gratefully appreciated. The first author wishes to thank NASA, MCT, QSS, Intel Research Cambridge, EPSRC, Big Hand Ltd. and St Edmund's College for financially supporting this work.

A NON-CLAUSAL SAT SOLVER

An NNF-compliant SAT solver was implemented as a C++ library for the purposes of supporting experimentation on this project.

NNF-GSAT Initially, the relatively simple GSAT algorithm [12] was adopted. Though originally intended for use with CNF problems, it was relatively straightforward to adapt the algorithm for use with NNF. In outline, the algorithm works as follows:

1. Initialise the variables to random initial values
2. Check to see whether the current variable values satisfy the expression completely. If so, a solution has been found, so the loop terminates.
3. For each variable, flip its state (i.e. change 0 to 1 and vice-versa), then note the number of subexpressions that are *satisfied* (i.e. evaluate to 1) as a consequence. Return each variable to its initial state after each count.
4. Choose the variable that most increases the number of satisfied subexpressions, then flip it permanently.
5. If a predetermined number of attempts has been exceeded, go back to step 1, otherwise go to step 2.

In practice, this algorithm is a little too simplistic, and requires some extra heuristics in order to prevent it from becoming trivially stuck in local minima. Initial results indicated that, though slow, GSAT was actually surprisingly effective, given its extreme simplicity.

NNF-WALKSAT In order to improve upon the performance of NNF-GSAT, the WALKSAT algorithm [11] was similarly adapted for use with NNF. The basic WALKSAT algorithm may be summarised as follows:

1. Choose a clause at random that is currently unsatisfied
2. Depending on whether a random number exceeds the current $heat$ parameter, either:
   (a) Randomly choose a variable that appears within the clause and flip it, or
   (b) Attempt flipping each variable that appears within the clause in turn, noting the number of unsatisfied clauses that result in each case, then choose the one flip that results in the lowest number of unsatisfied clauses. This is referred to hereinafter as a *greedy flip.*

WALKSAT is superficially similar to GSAT, but due to the need on each iteration only to enumerate the variables within a single clause rather than all unbound variables in the entire expression, it is generally much faster whilst retaining roughly equivalent power. As with GSAT, the basic WALKSAT algorithm is intended for use with expressions in CNF, so it was necessary to extend and modify it to deal with the more general NNF case.

The resulting SAT solver, is able to solve the majority of our test cases rapidly, even where multiple stuck-at faults were simulated. The basic WALKSAT algorithm required some modifications and extra heuristics, due to a bad tendency to get stuck in local minima. The extensions we used are summarised as follows:

Supporting terms as well as clauses In an expression in CNF, one single outer *term* encapsulates possibly many clauses, and clauses may only contain variables or their negations, not terms. NNF relaxes this somewhat, in that terms may contain clauses and vice-versa, with the only significant restriction in comparison with general Boolean expressions being the requirement that negation may only appear adjacent to a variable.

In NNF-WALKSAT, we perform a preprocessing stage, whereupon for each term and each clause, the list of variables contained within them is cached. Variables that appear directly within a term are regarded as equivalent to singleton clauses containing only that variable.

Pre-optimisation of the NNF expression A simple pre-optimisation pass is performed first, such that clauses that are of the form $a \vee \neg a \vee b \vee \ldots$ are replaced with $true$, terms of the form $a \wedge \neg a \wedge b \wedge \ldots$ are replaced with $false$, then any remaining constants are evaluated out and folded into the expression.

Giving clauses close to the root preference When randomly selecting a clause, preference is given to clauses that appear close to the root of the expression tree, on the basis that such variables are more likely to have a wide impact, so it is appropriate to try to make an estimate of their value early.

Super-flips We add a third kind of flip, in addition to random flips and greedy flips. A *super-flip* requires trying all possible combinations of variables, then selecting the combination resulting in the best score. Since this algorithm has a complexity of $O(2^N)$, it makes sense to set a fairly low upper limit on the number of variables to which it can be applied – in our current implementation, super-flips are only attempted for clauses with 8 variables or less.

Super-flips do not appear to make a big difference to many problems, but in some cases they appear to make it possible to find a solution quickly when the standard algorithm gets stuck for a long time, even when the probability of performing a super-flip is very small. A useful heuristic appears to be to have the probability $\frac{p}{2^N}$ of performing a super-flip, where p is an empirically-derived constant[8].

Dynamic control of the heat parameter The original WALKSAT algorithm suggests choosing between random and greedy flips with a probability of approximately 0.5. Our finding was that this does not work for expressions in NNF – though random flips are essential for avoiding local minima, they often significantly increase the number of unsatisfied clauses in the expression as a whole. We found that a random flip probability in the range $0.01..0.1$ normally works, but found that the ideal value was highly dependent on the expression being solved. If the probability is too low, the solver gets stuck in local minima, but if it is too high, the algorithm does not converge on a solution at all.

Our implementation dynamically varies the heat in accordance with the following heuristics:

1. If the most recent flip reduced the number of unsatisfied clauses, reduce the heat exponentially.
2. If the same variable is flipped twice in succession, suggesting that a local minimum has been encountered, increase the heat by a (fairly large) constant.
3. Otherwise, very gradually move the heat toward a default (small) value (0.001 in our implementation).

This approach works well for most of the SAT problems we have examined – early in the run, the heat is kept very low by rule 1, which makes it possible to converge quickly on a possible result. In many cases, a solution will fall out of this initial attempt immediately. However, if the SAT solver gets stuck in a local minimum, this frequently results in the same variable being toggled repeatedly – rule 2 picks up on this, increasing the heat, thereby pushing the variable bindings away from the minimum.

Retries Whenever a set of variable bindings is found that results in an improvement to the number of unsatisfied clauses, a snapshot of these bindings is taken for later use. If no improvement beyond this snapshot is seen for a predetermined number of attempts (1000 in our implementation), the last snapshot reverted to, giving the search procedure another attempt at finding an improved result. In a significant proportion of cases, this leads to a solution being found after a small number of retries.

[8] Our implementation uses $p = 1$.

Restarts If retrying does not succeed after a large number of attempts (5 in our implementation), this generally means that the solver is stuck in a local minimum that it can not climb out of by normal means. In this case, we reset the variable bindings to new, unrelated values then start again. By experimentation, it was found that determining these values according to the following algorithm is beneficial:

1. Initially, set all variables to 0
2. On the first restart, set all variables by counting the number of times that each appears negated and non-negated, choosing a value likely to satisfy the greatest number of clauses in each case
3. On the second restart, set all variables to 1
4. On all subsequent restarts, set all variables randomly

This can be visualised as initially trying one extreme of the problem space, then a case roughly in the middle of the problem space, then the other extreme, and then finally trying cases at random until a solution is found.

This approach works well as a general purpose SAT solver, although in our application we find it beneficial to first attempt an initial variable set initialised to the existing FPGA bit stream – in many cases, this proves to be a considerable speedup, whilst also increasing the percentage of successful runs.

BIBLIOGRAPHY

[1] COOK, S. The complexity of theorem proving procedures. In *Proc. 3rd Annual ACM Symposium on Theory of Computing* (1971), pp. 151–158.

[2] DAVIS, M., LOGEMANN, G., AND LOVELAND, D. A machine program for theorem proving. *Communications of the ACM 5*, 7 (July 1962), 394–397.

[3] DAVIS, M., AND PUTNAM, H. A computing procedure for quantification theory. *Journal of the ACM 7*, 3 (July 1960), 201–215.

[4] GREAVES, D. J. Direct synthesis of logic using a SAT solver. Unpublished research note, available at http://www.cl.cam.ac.uk/users/djg/wwwhpr/dslogic.html, 2004.

[5] JONES, N., GOMARD, C., AND SESTOFT, P. *Partial Evaluation and Automatic Program Generation*. Englewood Cliffs, NJ: Prentice Hall, 1993.

[6] LARCHEV, G., AND LOHN, J. D. Hardware-in-the-loop evolution of a 3-bit multiplier. In *Proc. 12th Annual IEEE Symposium on Field Programmable Custom Computing Machines, FCCM-2004* (2004), pp. 277–278.

[7] LOHN, J. D., LARCHEV, G., AND DEMARA, R. F. Evolutionary fault recovery in a Virtex FPGA using a representation that incorporates routing. In *Proc. IPDPS 2003* (2003).

[8] LOHN, J. D., LARCHEV, G., AND DEMARA, R. F. A genetic representation for evolutionary fault recovery in Virtex FPGAs. In *Proc. ICES 2003* (2003), pp. 47–56.

[9] LOMBARDI, L. Incremental computation. In *Advances in Computers, vol. 8*, F. Alt and M. Rubinoff, Eds. New York: Academic Press, 1967, pp. 247–333.

[10] LOMBARDI, L., AND RAPHAEL, B. Lisp as the language for an incremental computer. In *The Programming Language Lisp: Its Operation and Applications* (1964), E. Berkeley and D. Bobrow, Eds., Cambridge, MA: MIT Press, pp. 204–219.

[11] SELMAN, B., KAUTZ, H., AND COHEN, B. Noise strategies for improving local search. In *Proc. 12th National Conference on Artificial Intelligence, AAAI'94* (1994), vol. 1, MIT Press, pp. 337–343.

[12] SELMAN, B., LEVESQUE, H. J., AND MITCHELL, D. A new method for solving hard satisfiability problems. In *Proc. 10th National Conference on Artifical Intelligence, AAAI'92* (1992), pp. 440–446.

[13] STOICA, A., ARSLAN, T., KEYMEULEN, D., DUONG, V., GUO, X., ZEBULUM, R., FERGUSON, I., AND DAUD, T. Evolutionary recovery of electronic circuits from radiation induced faults. In *Proc. IEEE Conference on Evolutionary Computation* (2004), CEC.

[14] THIFFAULT, C., BACCHUS, F., AND WALSH, T. Solving non-clausal formulas with DPLL search. In *10th International Conference on Principles and Practice of Constraint Programming (CP-2004)* (2004), vol. 3258 of *Lecture Notes in Computer Science*, Springer-Verlag.

[15] THOMPSON, S., AND MYCROFT, A. Abstract interpretation of combinational asynchronous circuits. In *11th International Static Analysis Symposium (SAS'04)* (2004), R. Giacobazzi, Ed., vol. 3148 of *Lecture Notes in Computer Science*, Springer-Verlag, pp. 181–196.

[16] THOMPSON, S., AND MYCROFT, A. Bit-level partial evaluation of synchronous circuits. In preparation, draft available at http://findatlantis.com/mypapers/, 2004.

[17] THOMPSON, S., AND MYCROFT, A. Sliding window logic simulation. In *15th UK Asynchronous Forum* (2004), Cambridge.

[18] VELDHUIZEN, T. Using C++ template metaprograms. *C++ Report 7*, 4 (May 1995), 36–43. Reprinted in C++ Gems, ed. Stanley Lippman.

[19] VON NEUMANN, J. Probabilistic logics and the synthesis of reliable organisms from unreliable components. *Automata Studies* (1956), 43–98.