Bit Inference

Nachum Dershowitz

16 David Avidan St., Tel Aviv, Israel
nachumd@gmail.com

Abstract. Bit vectors and bit operations are proposed for efficient propositional inference. Bit arithmetic has efficient software and hardware implementations, which can be put to advantage in Boolean satisfiability procedures. Sets of variables are represented as bit vectors and formulæ as matrices. Symbolic operations are performed by bit arithmetic. As examples of inference done in this fashion, we describe ground resolution and ground completion.

"It does take a little bit of inference."
– Tony Fratto, Deputy Press Secretary, USA

1 Introduction

Boolean satisfiability, though NP-complete, is a problem that is solved on a daily basis with real-life industrial instances comprising millions of variables and clauses. See, for example, [18].

1.1 The Problem

Suppose B is a Boolean formula and $p_1, \ldots, p_v$ are its propositional variables. The *Boolean satisfiability (SAT) problem* is to find an assignment of truth values (0 and 1) to a subset of the variables, such that the formula becomes a tautology, or else to determine that no such assignment exists, in which case the formula is *unsatisfiable*.

Formulæ are often framed in clausal form. A *literal* is any variable p_j or its negation $\overline{p_j}$. A *clause* c is a (multi-) set of positive and negative literals, intending their disjunction. A *(clausal) formula* C is a set or list of clauses, intending their conjunction.

1.2 An Idea

Bit arithmetic enjoys efficient software and hardware implementations. These can be put to great advantage in satisfiability procedures. Sets of variables can be represented as bit vectors, rather than as (linked) lists, or tries. Formulæ would be represented as matrices, rather than as linked lists or binary decision diagrams [5]. Symbolic operations are, accordingly, replaced by bit arithmetic.

1.3 Related Work

There has been considerable work on the use of reconfigurable hardware for SAT solving in general or for individual instances (e.g. [24,22]). In contrast, here we are interested in leveraging the native operations of binary hardware for the problem.

1.4 This Paper

The use of bit operations on large bit arrays for the purpose of large-scale propositional inference, as elaborated here, appears to be novel.

The next section shows how formulæ are encoded as vectors of bits. As examples of the use of bit operations, the following two sections consider two important families of propositional inference, namely, ground resolution and ground completion. *Ground resolution* is the resolution rule for variable-free clauses, as used for SAT in [8]. *Ground completion* is an inference rule for variable-free equations, using equations from left-to-right to replace "equals-by-equals". The final two sections discuss aspects of the practicality of the suggestion.

2 Representation

A clause c can be represented by two bit vectors $c^0[1\colon v]$ and $c^1[1\colon v]$, where v is the number of bits in the vector, $c^0[j] = 1$ iff the negative literal $\overline{p_j}$ occurs in c, and $c^1[j] = 1$ iff the positive literal p_j occurs therein. Thus, a variable p_k (or literal $\overline{p_k}$) is identified with the vector containing a single 1 in position k (or $k + v$, respectively). Let c also denote the $2v$-bit-long concatenation of c^0 and c^1, symbolized $c^0 \frown c^1$, and c^* the reverse concatenation $c^1 \frown c^0$. To encode a tautological clause "true", one can add a bit in the 0th position, $c[0]$, to clauses c, and use $\top$ to abbreviate the corresponding vector p_0.

The standard set operations will denote the corresponding bit-vector functions. For example, $\cap$ represents logical-and and $\varnothing$ is the zero-vector, which corresponds to the value *false*. So, if $c^0 \cap c^1 \neq \varnothing$, then c is tautological, as it includes both a literal and its negation. Symmetric-difference (exclusive-or) is $\oplus$. Set difference can be obtained in two steps when it is not directly available: $x \setminus y = x \cap \overline{y}$. We will let $\|c\|$ count the number of ones (the "population count") in vector c. Inequalities of bit vectors treat the low-index bits as most significant. It is customary to also use 0 and 1 for *false* and *true*, respectively.

A *binomial* equation $e^L = e^R$ between Boolean monomials (products of propositional variables) can likewise be represented as two bit vectors $e^L[1\colon v]$ and $e^R[1\colon v]$, where $e^L[j] = 1$ iff the variable p_j occurs in the left side e^L and $e^R[j] = 1$ iff it occurs in the right side e^R. In this case, the most significant bits $e^L[0]$ and $e^R[0]$ can conveniently be set to indicate the monomial 0 (*false*), regardless of the values of other bits (thinking of the zero-bit as indicating a 0-factor). Thus, p_0 represents the truth constant *false*, but so does any vector with its most significant bit on. Accordingly, the truth constant *true* is denoted by the zero-vector (empty monomial) $\varnothing$.

A list C of n clauses $c_1, \ldots, c_n$ may be represented as a pair of $n \times (v+1)$ matrices, C^0 and C^1, where $C^r[i,j] = 1$ iff $c_i^r[j] = 1$ (for $r = 0, 1$). To refer to the whole jth column, one can write $C^r[*, j]$ ($r = 0$, 1, or blank). All or half of the ith row, $C^r[i, *]$, is just c_i^r ($r = 0$, 1, or blank, for left half, right half, or both halves, respectively). Similarly, a list of n Boolean equations may be represented as a pair of matrices, C^L and C^R, for left and right sides of equations.

3 Resolution

A clause is *empty*, and hence unsatisfiable, if $c = \varnothing$ (that is, $\|c\| = 0$). A clause is a *unit* if $\|c\| = 1$, which coerces the truth value of its one literal. A clause is *trivial* (tautological), and may be deleted, if $c^0 \cap c^1 \neq \varnothing$, since it disjoins a literal and its complement. To delete a clause, we will set its (high-order) 0-bit to 1. Two clauses c and d *resolve on* p_k if $c^* \cap d = p_k$, for some (positive or negative) literal p_k, producing a new clause $(c \cup d) \setminus (p_k \cup p_k^*)$. The resultant clause may be empty or a unit, but resolving non-units yields a non-empty clause.

Resolution provers invariably include simplification stages, such as unit propagation and subsumption, which we discuss next.

3.1 Unit Propagation

A unit clause c propagates and simplifies clause d if $c^* \subseteq d$, in which case the result is $d' = d \setminus c^*$. If the result d' is empty ($d = c^*$), the problem is unsatisfiable. If the result is a unit, then d' can be used in the same fashion. *Binary constraint propagation* (*BCP*) is the repeated application of subsumption by units and unit propagation – until no further simplifications are possible. BCP is a central component of the Davis-Putnam-Logemann-Loveland backtracking SAT procedure [7], and its modern incarnations. It is expensive (typically consuming 80–90% of the running time), but is not necessary for completeness (and can significantly degrade proof search; see [11]).

Let n be the number of clauses, and let $u[0{:}\,2v]$ be a bit-vector of length $2v+1$. At the conclusion of the algorithm in Fig. 1, all the units obtained by propagating the clauses of C will be marked in u. The n-step main loop repeats at most v times. An empty clause c_i means the problem is unsatisfiable. To delete a row, we set $c_i := \top$; it would be enough to let $c_i[0] := 1$. The matrix can be compacted by removing the deleted rows (at any juncture) and/or the columns marked in u (after any complete pass).

3.2 Subsumption

Clause c *subsumes* clause d if $c \subseteq d$, in which case d is superfluous. For this to be the case, we must have $c \leq d$, as binary numbers, but this is an insufficient condition. Using standard operations, $c \subseteq d$ iff $c \cup d = d$.

Subsumption is more expensive than unit propagation and should normally be preceded by BCP. It can be implemented like sorting, with the addition of

$u := \varnothing$
$b := true$
while b **do**
    $b := false$
    for $i := 1, \ldots, n$ **do**
        if $u \cap c_i = \varnothing$
          then $c_i := c_i \setminus u^*$
              if $c_i = \varnothing$ **then fail**
              if $\|c_i\| = 1$
                then $u := u \cup c_i$
                    $b := true$
          else $c_i := \top$

Fig. 1. Binary constraint propagation

for $i := 1, \ldots, n-1$ **do**
    if $c_i \neq \top$ **then**
      for $j := i+1, \ldots, n$ **do**
        if $c_i > c_j$
          then if $c_j \subseteq c_i$
                then $c_i := c_j$
                    $c_j := \top$
                else $c_j :=: c_i$
          else if $c_i \subseteq c_j$
                then $c_j := \top$

Fig. 2. Subsumption

checking whether the smaller of any pair subsumes the larger, in which case, the larger is deleted – for a cost of $O(n \lg n)$ vector-operations to check all clauses. Deleted rows should be removed. For an n^2 version, à la selection sort, see Fig. 2. Subsumption is often not cost-effective in standard implementations, but might be in this context. (Satellite [3,12], interestingly, does use bit vectors to estimate the applicability of subsumption.)

Other implementations of these algorithms, taking advantage of matrix operations, are conceivable.

3.3 The Davis-Putnam Resolution Procedure

The original Davis-Putnam (DP) procedure resolves clauses, variable by variable [8]. See Fig. 3. There are various heuristics for ordering the variables, such as choosing the one that appears in the most clauses. Columns can be presorted to reflect such policies. BCP can be incorporated, and perhaps subsumption, taking into account that the literals p_k and $\overline{p_k}$ are removed with each iteration on k.

$m := n$
for $k := 1, \ldots, v$ **do**
  $n := m$
  for $i := 1, \ldots, n-1$ **do**
    for $j := i+1, \ldots, n$ **do**
      if $c_i \cap c_j^* \subset (p_k \cup \overline{p_k})$
        then $m := m + 1$
          $c_m := (c_i \cup c_j) \setminus (p_k \cup \overline{p_k})$
          if $c_m = \varnothing$ **then fail**

Fig. 3. Davis-Putnam resolution

$m := 0$
$k := n$
while $k > m$ **do**
  $n := m$
  $m := k$
  for $i := 1, \ldots, m$ **do**
    for $j := n+1, \ldots, m$ **do**
      if $e_i^L \subseteq e_j^R$
        then $e_j^R := e_j^R \setminus e_i^L \cup e_i^R$
      if $e_i^L \subseteq e_j^L$
        then $e_j^L := e_j^L \setminus e_i^L \cup e_i^R$
          if $e_j^R > e_j^L$ **then** $e_j := e_j^*$
      else if $e_i^L \cap e_j^L \neq \varnothing$
        then $k := k + 1$
          $e_k^L := e_j^L \setminus e_i^L \cup e_i^R$
          $e_k^R := e_i^L \setminus e_j^L \cup e_j^R$
          if $e_k^R > e_k^L$ **then** $e_k := e_k^*$

Fig. 4. Knuth-Bendix completion

4 Completion

Knuth-Bendix *completion* [14], and its extensions, repeatedly finds overlaps between equations (using only the larger side of any equation), to infer new equations. (In contrast, paramodulation [23] looks at both sides of equations.) Equational reasoning provides an alternative inference paradigm to propositional reasoning, with equations in completion playing an analogous rôle to clauses in resolution.

We are interested in the ground (variable-free) case of completion, where the operations are associative and commutative [1,16,17]. As examples of completion in the realm of Boolean formulæ, we will consider ground Horn-clause theories and Gaussian elimination over $\mathbb{Z}_2$.

```
b := true
while b do
    b := false
    for i := 1, ..., n − 1 do
        for j := i + 1, ..., n do
            if e_i^L ⊆ e_j^L
              then e_j^L := e_j^L \ e_i^L ∪ e_i^R
                   if e_j^R > e_j^L then e_j^L :=: e_j^R
                   b := true
            if e_i^L ⊆ e_j^R
              then e_j^R := e_j^R \ e_i^L ∪ e_i^R
                   b := true
```

Fig. 5. Inter-reduction

4.1 Horn-Clause Completion

A clause is *Horn* if it has at most one positive literal. A Horn clause $p_0 \vee \neg p_1 \vee \cdots \vee \neg p_n$ is equivalent to the binomial equation $p_0 p_1 \cdots p_n = p_1 \cdots p_n$; a negative Horn clause $\neg p_1 \vee \cdots \vee \neg p_n$ is equivalent to the monomial equation $p_1 \cdots p_n = 0$. See [4] for details regarding such representations.

Two equations e_i and e_j are *critical* iff $e_i^L \cap e_j^L \neq \varnothing$. The *critical equation* (or *critical pair*) is $e^L = e^R$, where $e^L := e_j^L \setminus e_i^L \cup e_i^R$ and $e^R := e_i^L \setminus e_j^L \cup e_j^R$. Critical equations may need to be oriented. Knuth-Bendix (KB) completion (or the analogous Gröbner basis construction [6]) is the repeated generation of critical pairs, interleaved with inter-reduction.

In this manner, completion serves as the inference engine, generating critical pairs from the equational representation of Horn clauses, as shown in Fig. 4.

4.2 Reduction

A major component of completion is *simplification*, akin to *demodulation* [23], by which we mean using equations in one direction to "simplify" other equations (with respect to some measure).

An oriented equation $e^L = e^R$ is *unitary* and can be used to simplify in any of the following three cases:

- *Positive Unit.* If $e^R = \varnothing$, then the equation signifies $e^L = 1$ (since we agreed in Sect. 2 to interpret $\varnothing$ as truth). It follows that $p_i = 1$ for every $p_i \in e^L$. Apply $e^R = \varnothing$ to a monomial m by removing the (superfluous) positive bits: $m := m \setminus e^R$.
- *Negative Unit.* If $\|e^L\| = 1$ and $e^R[0] = 1$, then $p_k = 0$ for the $p_k \in e^L$. Apply $p_k = 0$ by zeroing any monomial in which it appears: **if** $m[k]$ **then** $m[0] := 1$.
- *Unit Equivalence.* If $\|e^L\| = \|e^R\| = 1$ and $e^L[0] = e^R[0] = 0$, then $p_k = p_j$ for the $p_k \in e^L$ and $p_j \in e^R$. Apply $p_k = p_j$ by replacing occurrences of p_k with p_j: **if** $m[k]$ **then** $m[j] := 1$.

```
i := 1
k := 1
while k ≤ v ∧ i ≤ n do
      m := i
      while m ≤ n ∧ ¬c_m[k] do m := m + 1
      if m ≤ n then
         c_m :=: c_i
         for j := 1, ..., i − 1, m + 1, ..., n do
               if c_j[k] then c_j := c_j ⊕ c_i
         i := i + 1
      k := k + 1
```

Fig. 6. Gaussian elimination

The results of such unit simplifications can propagate as in resolution.

More generally, an equation $e^L = e^R$ can be used to simplify a monomial m provided all the variables in e^L appear in m, that is, when $e^L \subseteq m$. The *rewrite step* is the assignment $m := m \setminus e^L \cup e^R$. If we use the 0-bit to signify the term 0, as explained above, then reducing products to 0 works as expected.

The lexicographic ordering of monomials is ordinary bit-string inequality. An equation c needs to *reoriented* if $e^R > e^L$, which may transpire after reducing a left side. Other orderings are possible.

To inter-reduce a system C of equations, applying all equations to all equations, as much as possible, first sort C in ascending order according to $\langle \|e^R\| - \|e^L\|, e^L, e_1 \rangle$ and then apply the algorithm in Fig. 5. The idea is that reducing with a "rewrite rule" $\ell \to r$ decreases the binary value of the string it is applied to by $\|\ell\| - \|r\|$, and, long range, one wants to maximize the decreases obtained with each reduction, so as to converge as quickly as possible. This naïve program can presumably still require exponentially many vector operations, but hopefully much better algorithms for inter-reduction can be devised (compare the non-commutative case [13,21]). One may prefer to limit reduction to equations with few variables on the left.

4.3 Gaussian Elimination

A linear equation over $\mathbb{Z}_2$ takes the form $P = 0$, where P is an exclusive disjunction of some of the propositional variables $p_1, \ldots, p_v$. (Since we are using $\oplus$, coefficients are 0 or 1.)

We represent an equation $P = 0$ as a bit vector c of length $v+1$, where $c[k] = 1$ iff p_k is a summand in P and p_0 is the constant 1. Adding (or subtracting) a linear equation c to d is just $d := d \oplus c$. A standard quadratic (vn vector operations) Gaussian elimination procedure is given in Fig. 6.

When (after elimination, say) $\|c\| \leq 2$, the equation c is unitary and is of one of the following three forms: $p_k = 0$, $p_k = 1$, or $p_k = p_j$, for some $k \geq 1$ and $1 \leq j \neq k$.

```
LOAD 0, ci0      |   LOAD 0, ci1
OR 0, cj0        |   OR 0, cj1
LOAD 2, ci0      |   LOAD 2, ci1
AND 2, cj1       |   AND 2, cj0
DIFF 0, 2        |   DIFF 0, 2
STORE 0, c0      |   STORE 0, c0
```

Fig. 7. A resolution step in an assembly language

4.4 Combining the Two

For non-Horn clauses, one needs also to incorporate negation in some form. The BinLin representation of propositional formulæ, proposed in [9,10], uses a combination of equations between monomials and linear equations over $\mathbb{Z}_2$ to represent propositional formulæ in exclusive-or (Boolean ring) normal-form. It provides an alternative to other propositional satisfiability procedures, whether search-based, saturation-based, or hybrid intersection-based methods. In this formalism, variables and equations are added in a satisfiability-preserving fashion, to obtain a set of binomial equations and a set of linear Boolean equations. The binomials undergo inter-reduction and the linear equations undergo Gaussian elimination. Unitary equations are propagated among both sets. This method, too, can be implemented naturally within the framework proposed here.

5 Implementation

Most of the bit-vector operations used in the above sections are readily available on digital computers. Some processors, even way back to the IBM Stretch, provide a hardware instruction for the number of ones in a machine word; in any case, computing $\|c\|$ requires only a few machine instructions [2, No. 169]. Most operations are also available in many software languages (e.g. C). They are all easy to implement in general-purpose or special-purpose hardware.

For example, resolving two single-word (or double-word – for machines with double-word operations) clauses requires approximately 12 machine instructions. Thus v variables require $12\lceil v/w \rceil$ instructions on a w-bit machine. For example, if $w = 64$ and $v = 1000$, fewer than 200 machine instructions are needed. See Fig. 7. This should be contrasted with the large number of machine operations used in a pointer-based implementation.

For large (but presumably sparse) vectors, (iterated) summary bits should prove helpful. (The *summary bit* for a subvector x is 0 iff $x = 0$.) Column operations, such as erasing all occurrences of a true propositional variable, may be sped up by also maintaining transpose matrices [20].

Industrial-strength problems can easily involve hundreds of thousands of variables and millions of clauses. The storage requirements for a bit matrix of that size is in the hundreds-of-gigabyte range. Given enough storage, full-fledged $n \lg n$ subsumption would take a few minutes on a 5000 MIPS 64-bit machine.

6 Discussion

Davis-Putnam resolution is a saturation-based methods for checking propositional satisfiability. The original set of clauses is satisfiable if and only if resolution terminates without having derived the empty clause. Similarly, Knuth-Bendix completion derives the contradiction $1 = 0$ if and only if the input clauses are unsatisfiable. Thus, both methods (Figs. 3 and 4) repeatedly add rows to the matrices of formulæ.

Saturation is often considered too costly in practice. Instead, a backtrack search [7] – based on the clausal representation with unit propagation and subsumption – can easily be built around the above procedures. One simple way to keep track would be to mark rows of the matrix that are added or deleted with the search level. (Instead of changing a row, one would delete and add.) After a significant number of assignments, it may pay to compact the matrix.

Similarly, a recursive-learning intersection-based method [15,19], combining limited saturation, generous simplification, and judicious search can be designed.

The algorithms given here are readily adaptable to highly parallel vector or array architectures. Experiments with simulations are needed to evaluate their practical feasibility.

Acknowledgement

I thank Guan-Shieng Huang for many ideas, discussions and vegetarian meals.

References

1. A. M. Ballantyne and D. S. Lankford. "New decision algorithms for finitely presented commutative semigroups". *J. Computational Mathematics with Applications*, vol. 7, 1981, pp. 159–165
2. M. Beeler, R. W. Gosper, and R. Schroeppel. "HAKMEM", Artificial Intelligence Memo No. 239, Massachusetts Institute of Technology, A. I. Laboratory, Feb. 1972. Available at `http://www.inwap.com/pdp10/hbaker/hakmem/hakmem.html`
3. A. Biere, "Resolve and expand, *Proc. of the 7th International Conference on Theory and Applications of Satisfiability Testing (SAT'04)*, Lecture Notes in Computer Science, vol. 3542, Springer-Verlag, Berlin, 2005, pp. 59–60
4. M.-P. Bonacina and N. Dershowitz. "Canonical Inference for Implicational Systems", Proc. of the 4th Intl. Joint Conference on Automated Reasoning, A. Armando, P. Baumgartner, and G. Dowek, eds., Lecture Notes in Computer Science, Springer-Verlag, Berlin, Aug. 2008, pp. 380–395
5. R. E. Bryant. "Symbolic Boolean manipulation with ordered binary-decision diagrams", *ACM Computing Surveys*, vol. 24, 1992, pp. 293–318
6. B. Buchberger. "Gröbner bases: An algorithmic method in polynomial ideal theory". Bose, N. K., ed., *Multidimensional Systems Theory*, Reidel, 1985, pp. 184–232
7. M. Davis, G. Logemann, and D. Loveland. "A machine program for theorem proving", *Communications of the ACM*, vol. 5, 1962, pp. 394–397
8. M. Davis and H. Putnam. "A computing procedure for quantification theory". *J. of the ACM*, vol. 7, no. 3, July 1960, pp. 201–215.

9. N. Dershowitz, J. Hsiang, G.-S. Huang and D. Kaiss. "Boolean ring satisfiability", *Proc. 7th Intl. Conf. on Theory and Applications of Satisfiability Testing (SAT 2004)*, May 2004, pp. 281–286
10. N. Dershowitz, J. Hsiang, G.-S. Huang and D. Kaiss. "Boolean rings for intersection-based satisfiability", *Proc. of the 13th Intl. Conf. on Logic for Programming and Artificial Intelligence and Reasoning*, M. Hermann and A. Voronkov, eds., Lecture Notes in Computer Science, vol. 4246, Springer-Verlag, Berlin, Nov. 2006, pp. 482–496
11. N. Dershowitz and A. Nadel, "From total assignment enumeration to a modern SAT solver", submitted
12. N. Eén and A. Biere, "Effective preprocessing in SAT through variable and clause elimination", *Proc. of the 8th Intl. Conf. on Theory and Applications of Satisfiability Testing (SAT'05)*, Lecture Notes in Computer Science, vol. 3569, Springer-Verlag, Berlin, 2005, pp. 61–75
13. J. Gallier, P. Narendran, D. Plaisted, S. Raatz, and W. Snyder. "An algorithm for finding canonical sets of ground rewrite rules in polynomial time", *Journal of Association for Computing Machinery*, vol. 40, no. 1, 1993, pp. 1–16.
14. D. E. Knuth and P. B. Bendix. "Simple word problems in universal algebras". J. Leech, ed., *Computational Problems in Abstract Algebra*. Oxford: Pergamon Press, 1970, pp. 263–297
15. W. Kunz. "Recursive learning: A new implication technique for efficient solutions to CAD problems — test, verification, and optimization", *IEEE Trans. on Computer-Aided Design of Integrated Circuits and Systems*, vol. 13, 1994, pp. 1143–1158.
16. C. Marché. "On ground AC-completion". *Proc. of the 4th Intl. Conf. on Rewriting Techniques and Applications*, R. V. Book, ed., Lecture Notes in Computer Science, vol. 488, Springer-Verlag, Berlin, Apr. 1991, 411–422
17. P. Narendran and M. Rusinowitch. "Any ground associative commutative theory has a finite canonical system". *J. Automated Reasoning*, vol. 17, no. 1, Aug. 1996, pp. 131–143
18. M. R. Prasad, A. Biere, and A. Gupta. "A survey of recent advances in SAT-based formal verification". *Software Tools for Technology Transfer*, vol. 7, no. 2, 2005, pp. 156–173
19. M. Sheeran and G. Stålmarck. "A tutorial on Stålmarck's proof procedure for propositional logic". *Proc. of the 2nd Intl. Conference on Formal Methods in Computer-Aided Design*, Lecture Notes in Computer Science, Springer-Verlag, Nov. 1998, pp. 82–99
20. I. Skliarova and A. B. Ferrari. "The design and implementation of a reconfigurable processor for problems of combinatorial computation". *J. Syst. Archit.*, vol. 49, nos. 4–6, Sep. 2003, pp. 211–226
21. W. Snyder. "A fast algorithm for generating reduced ground rewriting systems from a set of ground equations," *J. of Symbolic Computation*, vol. 15, no. 4, 1993, pp. 415–450
22. T. Suyama, M. Yokoo, H. Sawada, and A. Nagoya. "Solving satisfiability problems using reconfigurable computing". *IEEE Trans. VLSI Systems*, vol. 9, no. 1, Feb. 2001, pp. 109–116
23. L. Wos, G. A. Robinson, D. F. Carson, and L. Shalla. "The concept of demodulation in theorem proving". *J. of the ACM*, vol. 14, no. 4, Oct. 1967, pp. 698–709
24. P. Zhong, P. Ashar, S. Malik, and M. Martonosi. "Using reconfigurable computing techniques to accelerate problems in the CAD domain: A case study with Boolean satisfiability". *Proc. of the Design Automation Conf. (DAC)*, 1998, pp. 194–199