From Idempotent Generalized Boolean Assignments to Multi-bit Search

Marijn Heule* and Hans van Maaren

Department of Software Technology
Faculty of Electrical Engineering, Mathematics and Computer Sciences
Delft University of Technology
marijn@heule.nl, h.vanmaaren@tudelft.nl

Abstract. This paper shows that idempotents in finite rings of integers can act as Generalized Boolean Assignments (GBA's) by providing a completeness theorem. We introduce the notion of a generic Generalized Boolean Assignment. The mere propagation of such an assignment reveals feasibility (existence of a solution) of a formula in propositional logic. Then, we demystify this general concept by formulating the process on the bit-level: It turns out that propagation of a GBA only simulates bitwise (non-communicating) parallel computing. We capitalize on this by modifying the state-of-the-art local search SAT solver UnitWalk accordingly. This modification involves a more complicated parallelism.

1 Introduction

Propositional Logic and Elementary Arithmetic are - in some sense - similar systems. We provide additional evidence of this by introducing Generalized Boolean Models (GBM's) as certain sets of idempotents in finite residue class rings of integers, and a completeness theorem. We also offer a construction of so-called generic Generalized Boolean Assignments (generic GBA's). We show that formula feasibility can be checked by evaluating its "truth" value under one single generic assignment. These modeling possibilities feature an attractive mathematical simplicity. However, analysis of the proof of the completeness theorem and the process of constructing generic GBA's shows that the above modeling possibility only simulates (non-communicating) parallel computing.

Current Satisfiability (SAT) solvers do not use the opportunity of a k-bit processor to simulate parallel 1-bit (Boolean) search on k 1-bit processors. Conventional parallel SAT solving [3,4,9] differs from the proposed method in section 3: The former realizes performance gain by dividing the workload over multiple processors and some minor changes to the solving algorithm, while the latter uses a single processor and requires significant modifications to the algorithm.

SAT solvers that use multi-bit heuristics frequently (counters for instance), are not very suitable for modification in this respect. However, SAT solvers whose

* Supported by the Dutch Organization for Scientific Research (NWO) under grant 617.023.306

computational "center of gravity" consists of propagating truth values (or other 1-bit operations) may profit from this opportunity. One of such is the state-of-the-art local search SAT solver UnitWalk [6]. We show that UnitWalk can be upgraded using a single k-bit processor. This results in a considerable speed-up.

2 Idempotents and Generalized Boolean Assignments

The concepts in this section could have been cast into the format of Boolean Algebras [5]. As such, we do not claim that the ideas and results are completely new. However, using a little bit of elementary number theory it is possible to directly relate the concepts needed to familiar arithmetical operations. We preferred to do the latter. On the other hand, to understand the essentials of the next sections, it does not hurt the reader much to continue reading from **Back to Booleans** at the end of this section.

An *idempotent* x in the ring of integers modulo m is an element satisfying $x^2 \equiv x$ (modulo m). For given m, a *Generalized Boolean Model* (GBM) $\mathcal{I}$ is a set of idempotents modulo m obeying the three closure rules:

- $0, 1 \in \mathcal{I}$;
- If $x \in \mathcal{I}$ then $1 - x \in \mathcal{I}$. Notice that $(1 - x)^2 \equiv 1 - x$ (modulo m).
- If $x, y \in \mathcal{I}$ then $xy \in \mathcal{I}$. Notice that $(xy)^2 \equiv xy$ (modulo m).

Given a formula $\mathcal{F}$ in Propositional Logic a *Generalized Boolean Assignment* (GBA) is a mapping from its set of variables to a GBM $\mathcal{I}$. Evaluating the "truth" value of $\mathcal{F}$ under a GBA simply follows the rule of translating $\neg x$ by the arithmetic operation 1 - *value*(x) and conjunction $x \wedge y$ by the operation *value*(x)·*value*(y) (both modulo m), recursively.

Example 1. Consider Z_6, the ring of integers modulo 6. Idempotents modulo 6 are 0, 1, 3 and 4. Let $\mathcal{F}$ be the formula

$$\neg(x \rightarrow (y \vee (x \wedge z))) \tag{1}$$

which is equivalent to

$$x \wedge (\neg y \wedge \neg(x \wedge z)) \tag{2}$$

and assigning $x := 3$, $y := 4$ and $z := 1$, we calculate

$$3 \times ((1 - 4) \times (1 - (3 \times 1))) \equiv 0 \text{ (modulo 6)} \tag{3}$$

By assigning $x := 3$, $y := 4$ and $z := 0$ however, $\mathcal{F}$ evaluates to the value 3, as the reader may verify. Following from the above, each evaluation of a formula under a GBA results in an idempotent in $\mathcal{I}$, due to the closure rules posed on $\mathcal{I}$.

Example 2. Again consider Z_6 and the formula $x \wedge y$. The reader may check that there are 16 possible GBA's of which 7 evaluate to a non-zero idempotent. Drawing GBA's randomly, the probability of hitting a non-zero idempotent outcome is $\frac{7}{16}$, while in the standard Boolean situation this probability is $\frac{1}{4}$.

The above example shows that random sampling GBA's (or, equivalently, multi-bit Boolean patterns) may hit solutions earlier, in about the same time. As such this is done in [7], where Boolean "patterns" (rather than Booleans) are propagated through a circuit in the order to increase the probability of hitting a solution - indicating an error in their application.

Although this random sampling can be viewed as a rather straight forward parallelism, we claim that to perform efficient multi-bit propagation for SAT solving is not straight forward at all: In [7] at each step, variables are either unassigned or assigned a *full* Boolean pattern, while in the proposed propagation variables can also be assigned a *partial* Boolean assignment.

Theorem 1 (Completeness Theorem): If $\mathcal{F}$ is a formula and $\mathcal{I}$ a Generalized Boolean Model, $\mathcal{F}$ is Satisfiable if and only if there exists a GBA under which $\mathcal{F}$ evaluates to a non-zero idempotent.

Proof: *If $\mathcal{F}$ is Satisfiable, then a $\{0,1\}$-assignment exists under which $\mathcal{F}$ evaluates to 1. This evaluation remains valid in each $\mathcal{I}$. If $\mathcal{F}$ evaluates to a non-zero idempotent w modulo m, there must be a prime factor of m, say p, such that w is non-zero modulo p. Modulo a prime however, the only existing idempotents are 0 and 1, since $x^2 \equiv x$ (modulo p) reduces to $x \equiv 0$ or $x \equiv 1$ (modulo p). Under these circumstances there must be a prime number p which reduces, when calculating modulo p, the GBA to a simple Boolean assignment that satisfies $\mathcal{F}$.*

Construction of generic GBM's. GBM's are constructed as follows: Let m be the product of the first k primes. Let A be the product of a subset of these primes and B the product of the complementary subset. Since A and B are relatively prime, integers r and s exists such that

$$rA + sB = 1 \tag{4}$$

Set $x \equiv rA$ (modulo m). Then $1 - x \equiv sB$ (modulo m) and thus $x(1-x) \equiv 0$ (modulo m). The above observation shows that precisely 2^k different idempotents modulo m exist.

Generic GBA's. Let $\mathcal{F}$ be a formula on n variables and m be the product of the first 2^n primes. As we have seen above, there are 2^{2^n} different idempotents modulo m. This is the same amount as the number of logically independent Boolean functions on n variables. In fact, it is not hard to demonstrate that in the above situation a GBA to the variables exists such that each formula on n variables evaluates to its associated idempotent modulo m, each idempotent representing an equivalence class of Boolean functions. For example:

Example 3. Consider the Boolean functions with $n = 2$, $m = 2 \cdot 3 \cdot 5 \cdot 7 = 210$. The full GBM is $\{0, 1, 15, 21, 36, 70, 85, 91, 105, 106, 120, 126, 141, 175, 190, 196\}$. A generic GBA is for instance $x = 15$, $y = 21$. In this case $x \wedge \neg y$ evaluates to 120, $\neg(\neg x \wedge y)$ to 85, $x \Leftrightarrow y$ to 175 and $\neg(x \Leftrightarrow y)$ to 36. In this case, every formula on 2 variables can be checked on feasibility by propagating the values $x = 15$ and $y = 21$, and only the outcome 0 (modulo 210) reflects a contradiction. That $(15, 21)$ is generic follows from the fact that $(15, 21)$ is $(1, 1)$ modulo 2, $(0, 0)$ modulo 3, $(0, 1)$ modulo 5 and $(1, 0)$ modulo 7. Notice that - in some sense - we are just doing ordinary SAT in the exponents of the prime factors involved.

Working with GBA's could be beneficial in situations when the arithmetic operations involved can be performed in a small number of clock cycles. More specifically: If we have a 32-bit processor available, formulas with up to 5 variables can be resolved in one propagation run using generic GBA's in about the same time an ordinary Boolean assignment is propagated.

Back to Booleans. Despite the arithmetic elegance of generic GBM's in their capability of representing Boolean functions, it is clear that on the level of implementation integers are not a very welcome ingredient. In fact, the processes explained above are even easier to understand if we return to the Boolean level. To see this, consider the case of functions on 3 variables and the following table:

$$\begin{array}{lll} x & := & \texttt{0 1 0 0 1 1 0 1} \\ y & := & \texttt{0 0 1 0 1 0 1 1} \\ z & := & \texttt{0 0 0 1 0 1 1 1} \end{array}$$

Consider the rows as 8 parallel Boolean assignments to the individual variables (using 8 1-bit processors). We refer to such an assignment as a *multi-bit assignment* (MBA). Notice that the 8 different columns represent the 8 different Boolean assignments in total. Therefore, the above MBA is a *generic MBA* - analogue to generic GBA's. Having an 8-bit processor at our disposal the evaluation of the and-gate $x \wedge y$ results in $\texttt{01001101} \wedge \texttt{00101011} = \texttt{00001001}$, an operation performed in one clock cycle. Bits 5 and 8 certify feasibility. In general, any formula on n variables - based on the primitive operations `AND` ($\wedge$) and `NOT` ($\neg$) - can be resolved, using a generic MBA of 2^n bits (and a k-bit processor with $2^n \leq k$), in as many clock cycles as there are logical operators to perform. The formula is Satisfiable if and only if in the end there is at least one bit equal to 1. In terms of generic GBM's the above 8-bit example would involve calculating modulo $2 \cdot 3 \cdot 5 \cdot 7 \cdot 11 \cdot 13 \cdot 17 \cdot 19 = 9699690$.

3 Multi-Bit Unit Propagation

This section describes the use of MBA's to parallelize a SAT solving algorithm. However, this differs from conventional parallelism: Modifications of MBA's can be processed in parallel, while, for instance, operations on counters cannot. In general, only 1-bit operations can be parallelized. Therefore, algorithms that potentially benefit from MBA's should have their computational "center of gravity" on assignment modifications.

A widely used procedure for assignment modifications is *unit propagation*: Given a formula $\mathcal{F}$ and an assignment φ. If φ applied to $\mathcal{F}$ (denoted by $\varphi \circ \mathcal{F}$) contains *unit clauses* (clauses of size 1) then the remaining literal in each unit clause is forced to be true - thereby expanding φ. This procedure continues until there are no unit clauses in $\varphi \circ \mathcal{F}$. This section describes a SAT solving algorithm that uses unit propagation at its computational "center of gravity".

The UnitWalk algorithm. For a possible application we focused on local search (incomplete) SAT solvers. In contrast to complete SAT solvers, they are less complicated and work with full assignments. A generic structure of local search SAT solvers is as follows: An assignment φ is generated, earmarking a random Boolean value to all variables. By flipping the truth values of variables, φ can be modified to satisfy as many clauses as possible of the formula at hand. If after a multitude of flips φ still does not satisfy the formula, a new random assignment is generated.

Most local search SAT solvers use counting heuristics to flip the truth value of the variables in a turn-based manner. These heuristics appear hard to parallelize on a single processor. However, the UNITWALK algorithm [6] is an exception. Instead of counting heuristics, it uses unit propagation to flip variables. The UnitWalk SAT solver - based on this algorithm - is the fastest local search SAT solver on many structured instances and won the SAT 2003 competition in the category *All random SAT* [2].

The UNITWALK algorithm (see algorithm 1) flips variables in so-called *periods*: Each period starts with an initial assignment (referred to as master assignment φ_{master}), an empty assignment φ_{active} and an ordering of the variables π. First, unit propagation is executed on the empty assignment. Second, the first unassigned variable in π is assigned to its value in φ_{master}, followed by unit propagation of this value. A period ends when all variables are assigned a value in φ_{active}. Notice that *conflicts* - clauses with all literals assigned to false - are more or less neglected, depending on the implementation. A new period starts with the resulting φ_{active} as initial φ_{master} and a new ordering of the variables.

Algorithm 1 FLIP_UNITWALK(φ_{master})

```
1: for i in 1 to MAX_PERIODS do
2:     if φ_master satisfies F then
3:         break
4:     end if
5:     π := random ordering of the variables
6:     φ_active := ∅
7:     for j in 1 to n do
8:         while unit clause u ∈ φ_active ∘ F do
9:             φ_active[ VAR(u) ] := TRUTH(u)
10:        end while
11:        if π(j) not assigned in φ_active then
12:            φ_active[ π(j) ] := φ_master[ π(j) ]
13:        end if
14:    end for
15:    if φ_active = φ_master then
16:        random flip variable in φ_active
17:    end if
18:    φ_master := φ_active
19: end for
20: return φ_master
```

Example 4. Consider the example formula and initial settings below. Unassigned values in φ_{active} are denoted by $*$.

$$\begin{aligned}
\mathcal{F}_{\text{example}} &:= (x_1 \vee x_2) \wedge (\neg x_1 \vee x_2 \vee x_3) \wedge (\neg x_2 \vee \neg x_3) \\
&\quad\ (\neg x_2 \vee x_3 \vee \neg x_4) \wedge (\neg x_2 \vee x_3 \vee x_4) \wedge (\neg x_3 \vee \neg x_4) \\
\varphi_{\text{master}} &:= \{x_1 = \mathtt{0}, x_2 = \mathtt{1}, x_3 = \mathtt{1}, x_4 = \mathtt{0}\} \\
\varphi_{\text{active}} &:= \{x_1 = *, x_2 = *, x_3 = *, x_4 = *\} \\
\pi &:= (x_2, x_1, x_4, x_3)
\end{aligned}$$

Since the formula contains no unit clauses, the algorithm starts by selecting the first variable from the ordering - x_2. We assign this variable to true (as in φ_{master}) and perform unit propagation. Due to $\neg x_2 \vee \neg x_3$ this results in one unit clause $\neg x_3$. Propagation of this unit clause - assigning x_3 to false - results in unit clauses x_4, and $\neg x_4$. Because two complementary unit clauses have been generated we found a conflict. However, the UNITWALK algorithm does not resolve this conflict.

Instead, it continues by selecting[1] one of them, say $\neg x_4$, and assign x_4 to false. After this assignment $\varphi_{\text{active}} \circ \mathcal{F}$ does not contain unit clauses anymore. We conclude this period by assigning x_1 to its value in φ_{master}. This results in the full assignment $\varphi_{\text{active}} = \{x_1 = \mathtt{0}, x_2 = \mathtt{1}, x_3 = \mathtt{0}, x_4 = \mathtt{0}\}$. Notice that the new assignment does not satisfy clause $\neg x_2 \vee x_3 \vee x_4$.

[1] In [6] the authors suggest to select the truth value used in φ_{master}. However, this is not implemented in the latest version of the solver and we consider it as a choice.

Now, consider the same example, this time using a 4-bit assignment to all the variables. The reader must keep in mind that by parallelizing the former, we try to satisfy clauses in each bit position! Hence, variables may be flipped in multiple bits, and "conflict" means a conflict in some bit position. For the latter we shall use the term *bit-conflict*. Further, we keep using the term "truth value" for its multi-valued analogue. Notice that in the initial settings below, the first bit in φ_{master} equals the 1-bit example and that the ordering is the same.

$$\begin{aligned}\varphi_{\text{master}} &:= \{x_1 = \texttt{0110}, x_2 = \texttt{1100}, x_3 = \texttt{1010}, x_4 = \texttt{0110}\}\\ \varphi_{\text{active}} &:= \{x_1 = ****,\ x_2 = ****,\ x_3 = ****,\ x_4 = ****\}\\ \pi &:= (x_2, x_1, x_4, x_3)\end{aligned}$$

Again, we start by assigning x_2 to its value in φ_{master} followed by unit propagation. This will result in two unit clauses :

$$\begin{aligned}(x_1 = **** \vee x_2 = \texttt{1100}) &\Rightarrow x_1 := **\texttt{11}\\ (\neg x_2 = \texttt{0011} \vee \neg x_3 = ****) &\Rightarrow x_3 := \texttt{00}**\end{aligned}$$

One of them is selected, say x_1 and assigned to its value, resulting in:

$$(\neg x_1 = **\texttt{00} \vee x_2 = \texttt{1100} \vee x_3 = \texttt{00}**) \Rightarrow x_3 := \texttt{0011}$$

Now we assign x_3 which triggers three clauses:

$$\begin{aligned}(\neg x_2 = \texttt{0011} \vee x_3 = \texttt{0011} \vee \neg x_4 = ****) &\Rightarrow x_4 := \texttt{00}**\\ (\neg x_2 = \texttt{0011} \vee x_3 = \texttt{0011} \vee x_4 = \texttt{00}**) &\Rightarrow \textbf{bit}-\textbf{conflict}\\ (\neg x_3 = \texttt{1100} \vee \neg x_4 = \texttt{11}**) &\Rightarrow x_4 := \texttt{0000}\end{aligned}$$

When unit propagation stops, only the first two bits of x_1 are still undefined. These bits are set to their value in φ_{master} assigning all variables. The period ends with $\varphi_{\text{active}} = \{x_1 = \texttt{0111}, x_2 = \texttt{1100}, x_3 = \texttt{0011}, x_4 = \texttt{0000}\}$ - which satisfies the formula in the third and fourth bit.

The reader may check that: (1) The order in which unit clauses are propagated, as well as the order in which clauses are evaluated is not fixed. The order influences φ_{active} in case of conflicts. For example, evaluating $\neg x_2 \vee x_3 \vee x_4$ before $\neg x_2 \vee x_3 \vee \neg x_4$ results in a different final φ_{active}. (2) In the 4-bit example the third and fourth bit are the same for all variables. This effect could reduce the parallelism, because the algorithm as such does not intervene here and in fact maintains this collapse. This effect is not restricted to formulas with few variables. During our experiments we frequently detected a convergence to identical assignments over a considerable number of bit positions (sometimes even over all 32 positions, when using a 32-bit processor). We implemented a fast detection algorithm which replaces a duplicate with a new random assignment. Due to page limitations we cannot go into detail at this stage. Notice however that by doing so the first "communication" aspect is introduced.

4 Implementation UnitMarch

4.1 Unit propagation

The UnitPropagation procedure within the UnitWalk algorithm is not confluent: Different implementations yield different results. In short, two design decisions have to be made:

- In case of multiple unit clauses: which one to select for propagation;
- In case of a conflict: whether or how to act.

The most recent UnitWalk (version 1.003) implements the following UnitPropagation procedure: Unit clauses are stored in a multi-set (a set that can contain duplicate elements) data-structure. For each iteration a random element from the multi-set is selected. If the complement of the selected unit clause also occurs in the multi-set - meaning a conflict - all occurrences of x and $\neg x$ are removed from the multi-set. The algorithm continues with the next random element - see algorithm 2. Notice that this is a defensive flip strategy: Because of the removal, the truth value for x in φ_{active} tends to be the one copied from φ_{master}.

Algorithm 2 UnitPropagation_MultiSet ( )

```
1: while UnitMultiSet is not empty do
2:     x := random element from UnitMultiSet
3:     remove all occurrences of x in UnitMultiSet
4:     if unit clause ¬x also occurs in UnitMultiSet then
5:         remove all occurrences of ¬x in UnitMultiSet
6:     else
7:         φ_active[ VAR(x) ] := TRUTH(x)
8:         for all clauses C_i in which ¬x occurs do
9:             if C_i becomes a unit clause then
10:                add C_i to UnitMultiSet
11:            end if
12:        end for
13:    end if
14: end while
```

In our implementation we took a slightly different approach, since the above algorithm was hard to implement efficiently in a multi-bit version. Instead of the multi-set we used a queue (first in, first out) data-structure - see algorithm 3: Unit clauses are selected in the order they are added to the queue. In general, "early" generated unit clauses will have more bits assigned (at time of propagation) compared to "recent" unit clauses. Therefore the queue seems a useful data-structure since it always propagates the "earliest" unit clause left.

In addition, conflicts are handled differently: The queue is not allowed to contain complementary or duplicate unit clauses. The truth value of the first generated unit clause will be used during the further propagation. Notice that this flip strategy is more offensive: Given a bit-conflict, the truth value of the variable is flipped in approximately half of the cases. As we will see in the results (section 5), both implementations yield comparable results.

Algorithm 3 UnitPropagation_Queue ( )

```
1: while UnitQueue is not empty do
2:     x := removed front element from UnitQueue
3:     for all clauses C_i in which ¬x occurs do
4:         if C_i becomes a unit clause then
5:             y := remaining literal in C_i
6:             φ_active[ VAR(y) ] := TRUTH(y)
7:             if y not in UnitQueue then append y to UnitQueue
8:         end if
9:     end for
10: end while
```

4.2 Detection of Unit Clauses

The UnitWalk algorithm spends most computational time in detecting which clauses became unit clauses given an expansion of φ_{active}. If a variable is assigned a Boolean value, all clauses in which it occurs with complementary polarity are potential unit clauses. In a 1-bit implementation, only one unit clause could be detected in such a potential clause, while in a multi-bit implementation multiple unit clauses could be detected:

Example 6. Given $\varphi_{\text{active}} = \{x_1 = \texttt{010*}, x_2 = \texttt{10*1}, x_3 = \texttt{101*}, x_4 = \texttt{*001}\}$ with x_3 as unit clause to be propagated and potential clause $x_1 \vee \neg x_2 \vee \neg x_3 \vee x_4$.

$$(x_1 = \texttt{010*} \vee \neg x_2 = \texttt{01*0} \vee \neg x_3 = \texttt{010*} \vee x_4 = \texttt{*001}) \;\Rightarrow\; x_2 := \texttt{1001},\; x_4 := \texttt{1001}$$

In general, all literals besides the propagation literal are potential unit clauses.

Encoding. Since each bit in φ_{active} consists of three possible values (`*`,`0`,`1`), we used two bits to encode each value: `00 = *`, `01 = 0`, `10 = 1`, and `11 = bit-conflict`[2]. We used an array φ^{+}_{-} in which both x_i and $\neg x_i$ have a separate assignment: The first bit of each value is stored in x_i while the second bit is stored in $\neg x_i$. For example:

$$\varphi_{\text{active}}[x] = \texttt{101**0*1} \text{ is stored as } \begin{cases} \varphi^{+}_{-}[\;\; x] = \texttt{10100001} \\ \varphi^{+}_{-}[\neg x] = \texttt{01000100} \end{cases}$$

Using φ^{+}_{-} we can compute the unit clauses as below. Conflicts are ignored by only allowing unassigned bits - computed by $\texttt{NOT}(\varphi^{+}_{-}[x_i]\ \texttt{OR}\ \varphi^{+}_{-}[\neg x_i])$ - to be assigned. Back to the example:

$$\begin{aligned} x_1 &:= \varphi^{+}_{-}[x_3]\ \texttt{AND}\ \texttt{NOT}(\varphi^{+}_{-}[x_1]\ \texttt{OR}\ \varphi^{+}_{-}[\neg x_1])\ \texttt{AND}\ \varphi^{+}_{-}[x_2]\ \texttt{AND}\ \varphi^{+}_{-}[\neg x_4] \\ \neg x_2 &:= \varphi^{+}_{-}[x_3]\ \texttt{AND}\ \varphi^{+}_{-}[\neg x_1]\ \texttt{AND}\ \texttt{NOT}(\varphi^{+}_{-}[x_2]\ \texttt{OR}\ \varphi^{+}_{-}[\neg x_2])\ \texttt{AND}\ \varphi^{+}_{-}[\neg x_4] \\ x_4 &:= \varphi^{+}_{-}[x_3]\ \texttt{AND}\ \varphi^{+}_{-}[\neg x_1]\ \texttt{AND}\ \varphi^{+}_{-}[x_2]\ \texttt{AND}\ \texttt{NOT}(\varphi^{+}_{-}[x_4]\ \texttt{OR}\ \varphi^{+}_{-}[\neg x_4]) \end{aligned}$$

[2] The bit-conflict value is not possible within our implementation

The above shows a potential disadvantage of the multi-bit propagation: To check whether a clause of size k becomes a unit clause and to determine the remaining literal is not trivially computed in $\mathcal{O}(k)$ steps - as is the case with 1-bit propagation. However, a $\mathcal{O}(k)$ implementation can be realized by splitting the computation:

- Compute the *unit mask* - a multi-bit Boolean which is true on all positions with exactly one not falsified literal (denoted by $M_{\mathrm{NF}=1}$);
- Use the unit mask to quickly determine the newly created unit clauses: All literals that are unassigned at a true position in the unit mask became unit.

To compute $M_{\mathrm{NF}=1}$, we use two auxiliary masks, $M_{\mathrm{NF}\geq 1}$ and $M_{\mathrm{NF}\geq 2}$. The masks denote multi-bit Booleans which are true on all positions with at least one (and two, respectively) falsified literals and false elsewhere. Notice that $M_{\mathrm{NF}=1} := M_{\mathrm{NF}\geq 1}$ `XOR` $M_{\mathrm{NF}\geq 2}$. For each literal l_i in a clause we update $M_{\mathrm{NF}\geq 1}$ and $M_{\mathrm{NF}\geq 2}$ by the following two rules:

$$M_{\mathrm{NF}\geq 2} := (\ M_{\mathrm{NF}\geq 2}\ \texttt{OR}\ \texttt{NOT}(\varphi^+_-[\ \neg l_i\ ])\ )\ \texttt{AND}\ M_{\mathrm{NF}\geq 1}$$
$$M_{\mathrm{NF}\geq 1} := M_{\mathrm{NF}\geq 1}\ \texttt{OR}\ \texttt{NOT}(\varphi^+_-[\ \neg l_i\ ])$$

By negating the operations above, the computation becomes more efficient. Algorithm 4 shows the proposed implementation.

Algorithm 4 ComputeUnitMask (clause C_y)

```
1: M_I := ALL_BITS_TRUE, M_II := ALL_BITS_TRUE
2: for i in 1 to |C_y| do
3:     M_II := (M_II AND φ⁺₋[ ¬l_{y,i} ]) OR M_I
4:     M_I := M_I AND φ⁺₋[ ¬l_{y,i} ]
5: end for
6: return M_I XOR M_II
```

Once $M_{\mathrm{NF}=1}$ is computed ($M_{\mathrm{NF}=1}$ = `1010` in the example) we can determine the newly create unit clauses. For the example we only need the computations:

$$x_1 := M_{\mathrm{NF}=1}\ \texttt{AND}\ \texttt{NOT}(\varphi^+_-[x_1]\ \texttt{OR}\ \varphi^+_-[\neg x_1])$$
$$\neg x_2 := M_{\mathrm{NF}=1}\ \texttt{AND}\ \texttt{NOT}(\varphi^+_-[x_2]\ \texttt{OR}\ \varphi^+_-[\neg x_2])$$
$$x_4 := M_{\mathrm{NF}=1}\ \texttt{AND}\ \texttt{NOT}(\varphi^+_-[x_4]\ \texttt{OR}\ \varphi^+_-[\neg x_4])$$

5 Results

We implemented the UnitWalk algorithm as a multi-bit local search solver using UnitPropagation_Queue. The resulting solver, called UnitMarch, can be used for any number of bits. We added a method which replaces double

assignments with new random assignments (see section 3). The performance of UnitMarch is compared with the latest version of UnitWalk[3].

The latter is a hybrid solver: If after a number of periods the number of unsatisfied clauses is not reduced the solver switches to WalkSat [8]. If that algorithm does not find a solution after a multitude of flips it switches back, etc. Because we wanted to compare the influence of multi-bit search on the pure UnitWalk algorithm, the switching was disabled.

Table 1 shows a comparison between UnitWalk, UnitMarch 1-bit and UnitMarch 32-bit on various benchmarks. Besides the *dlx2-bugXX* family[4], all benchmarks can be found on SATlib[5] along with a description. For each solver, we set `MAX_PERIODS` $:= \infty$. We used 100 random seeds for all benchmarks.

The solvers UnitWalk and UnitMarch 1-bit show comparable performance. First, the number of periods executed per second is almost equal for all checked benchmarks. This shows that our implementation, with some overhead for parallelization, is fast enough on the benchmarks at hand. Second, the average number of periods between the two versions is comparable. Although they differ slightly between instances, no clear winner shows itself. Hence, the UnitPropagation_Queue procedure shows comparable to the UnitPropagation_MultiSet procedure in terms of performance.

Comparing the 1-bit solvers with UnitMarch 32-bit shows that the latter is the clear winner on almost all experimented instances. We found few exceptions (see *logistics-d*); all having less than 100 periods on the three solvers. Apparently, multi-bit search as implemented is not effective on these easy instances. Figures 1 and 2 present the effect of using different numbers of bits in more detail. Both figures use logarithmic axes - thus $f(x) = \frac{c}{x}$ is represented as a straight line. Four benchmarks are tested for all bits sizes 1 to 32. Using double logarithmic scaling, these instances show a linear dependency between the average number of periods and the number of used bits. The average time is also diminished on all these instances, although this reduction varies per instance. Notice that on all these instances the trend is strictly decreasing. It could be expected that computers with a k-bit architecture with $k > 32$ will boost performance even further.

6 Conclusions and future work

Our first observation is that propositional Boolean formulas with n variables can be mathematically elegantly checked on feasibility with a single assignment using the idempotents modulo the product of the first 2^n primes. Compared to conventional checking algorithms, the above just exchanges time for space. However, the architecture of today's computers is 32- or 64-bit - which enables execution of 32 (64) 1-bit operations simultaneously. Although many algorithms do not seem suitable for this kind of parallelism, the UnitWalk algorithm appears to be a good first candidate, as well as a state-of-the-art Sat solver [2].

[3] version 1.003 available from http://logic.pdmi.ras.ru/~arist/UnitWalk/
[4] available from http://www.miroslav-velev.com/sat_benchmarks.html
[5] http://www.satlib.org

Table 1. Comparison between the performance - in average number of periods and average time and standard deviation - of UnitWalk, UnitMarch 1-bit, and UnitMarch 32-bit on various benchmarks. The presented data averages runs using 100 different random seeds.

	UnitWalk 1.003		UnitMarch 1-bit		UnitMarch 32-bit	
	periods	**time**	**periods**	**time**	**periods**	**time**
aim-2-1-1	119336	6.13 (6.36)	37520	1.62 (1.65)	1339	**0.32** (0.33)
aim-2-1-2	1395975	73.56 (71.97)	1001609	44.67 (43.37)	45934	**11.35** (10.68)
aim-2-1-3	26487	1.40 (1.39)	12147	0.53 (0.60)	646	**0.16** (0.15)
aim-2-1-4	57794	3.13 (3.01)	30708	1.38 (1.58)	945	**0.23** (0.22)
aim-3-4-1	89923	7.57 (7.05)	62191	3.19 (3.07)	2134	**1.40** (1.42)
aim-3-4-2	99744	8.43 (7.98)	181623	9.33 (8.51)	5838	**3.81** (3.33)
aim-3-4-3	51898	4.33 (4.07)	20870	1.7 (0.90)	738	**0.48** (0.45)
aim-3-4-4	264125	21.96 (17.79)	240856	21.21 (13.43)	6234	**4.29** (3.15)
bw-large.b	441	0.32 (0.33)	311	0.18 (0.13)	13	**0.05** (0.03)
bw-large.c	13870	47.61 (40.90)	9342	19.85 (22.05)	498	**7.63** (7.44)
dlx2-bug17	1102	6.40 (9.53)	432	2.31 (2.80)	7	**0.43** (0.41)
dlx2-bug39	2830	6.78 (6.13)	1899	4.38 (3.72)	69	**1.33** (1.76)
dlx2-bug40	1632	3.96 (4.02)	988	2.34 (2.20)	26	**0.55** (0.55)
flat200-05	19384	3.46 (3.40)	19880	2.19 (2.35)	704	**0.81** (0.75)
flat200-24	5247	0.98 (1.02)	5145	0.56 (0.56)	130	**0.16** (0.18)
flat200-39	12142	2.16 (2.29)	12048	1.31 (1.21)	391	**0.44** (0.45)
flat200-48	2941	0.52 (0.54)	2346	0.26 (0.25)	84	**0.10** (0.10)
flat200-64	6406	1.14 (1.03)	6799	0.75 (0.75)	268	**0.34** (0.35)
logistics-a	1970338	636.47 (563.21)	863165	369.09 (383.97)	25100	**55.97** (43.53)
logistics-b	6313	1.91 (2.24)	11878	5.43 (5.76)	354	**0.73** (0.63)
logistics-c	133572	72.16 (69.36)	310450	228.49 (224.92)	9803	**34.19** (31.75)
logistics-d	23	0.11 (0.07)	24	**0.08** (0.04)	5	0.11 (0.03)
par16-1	14245	4.97 (4.73)	11267	2.65 (2.85)	365	**0.21** (0.20)
par16-2	21417	7.43 (8.08)	20601	5.05 (5.18)	702	**0.42** (0.34)
par16-3	17913	6.31 (7.04)	16872	3.98 (3.93)	551	**0.33** (0.42)
par16-4	16955	5.94 (5.77)	14087	3.33 (3.47)	523	**0.34** (0.32)
par16-5	18889	6.60 (6.70)	23028	5.41 (5.00)	640	**0.36** (0.36)
qg1-08	101390	424.17 (399.59)	121127	362.74 (377.55)	4229	**127.57** (120.87)
qg2-08	803258	3404.49 (3501.46)	1005351	4360.92 (4518.23)	26223	**991.23** (967.20)
qg3-08	165	0.08 (0.06)	166	0.10 (0.10)	5	**0.03** (0.03)
qg4-09	1344	1.10 (0.96)	2098	1.82 (1.66)	66	**0.53** (0.53)
qg5-11	591	1.92 (1.82)	670	2.13 (2.00)	23	**0.82** (0.68)
qg7-13	92600	492.66 (465.71)	98172	408.35 (419.56)	2937	**171.63** (146.69)
uf250-054	307317	33.69 (35.84)	472970	30.03 (27.82)	14851	**10.74** (11.57)
uf250-062	42137	4.60 (4.85)	88670	5.61 (5.44)	2427	**1.74** (1.84)
uf250-071	135296	14.49 (12.79)	218375	13.92 (13.70)	6404	**4.59** (4.66)
uf250-072	126387	13.91 (13.33)	172789	10.95 (9.81)	5624	**4.10** (4.28)
uf250-093	92110	9.78 (9.71)	146132	9.23 (8.37)	4521	**3.25** (2.94)

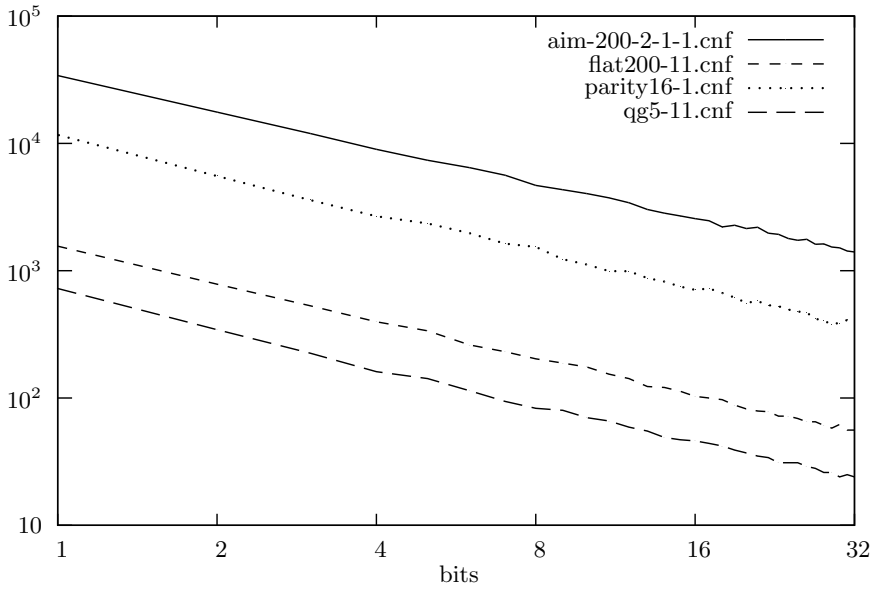

Fig. 1. Average number of periods by UnitMarch using different number of bits. Averages are computed using 1000 random seeds. Two logarithmic axes are used.

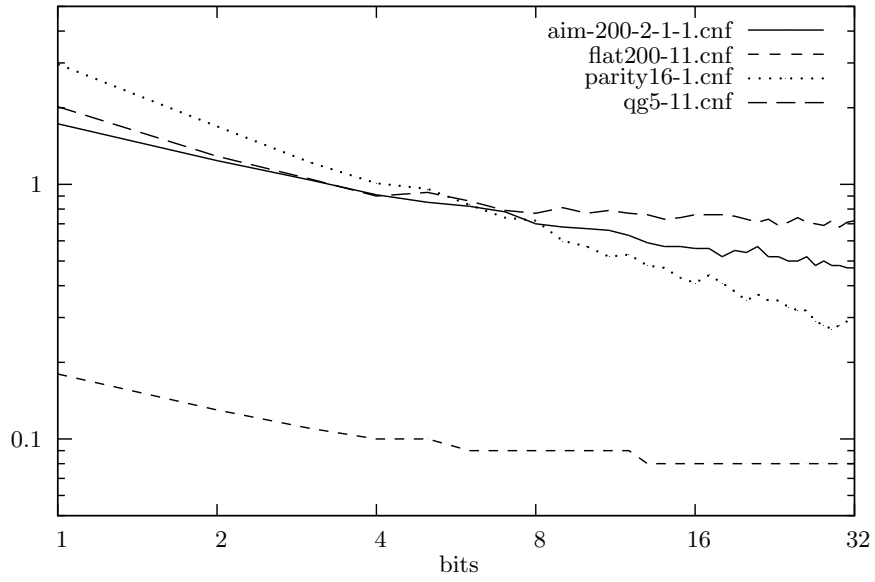

Fig. 2. Average time (in seconds) by UnitMarch using different number of bits. Averages are computed using 1000 random seeds. Two logarithmic axes are used.

Our multi-bit implementation of this algorithm, called UnitMarch, shows that this algorithm can be parallelized in such a way that the 1-bit version has comparable performance with the UnitWalk solver. Using double logarithmic scaling, these instances show a linear dependency between the average number of periods and the number of used bits. Most importantly, the average time to solve instances is largely reduced by using the 32-bit version.

The implementations of UnitWalk and UnitMarch are currently comparable (regardless the multi-bit feature) but are far from optimal: For instance, in both solvers unit clauses in the original CNF are propagated in each period. Another performance boost is expected by adding (redundant) clauses - for instance as implemented in the local search solver $\mathsf{R}^{+}\mathsf{AdaptNovelty}^{+}$ [1] - because they will increase the number of unit propagations. Finally, further experiments (not presented in this paper) showed that by ordering the variables less randomly and more based on multi-bit heuristics results in improved performance on many benchmarks. Developing enhancements (like replacement of duplicate assignments) and effective multi-bit heuristics is under current research.

Acknowledgments

The authors would like to thank Denis de Leeuw Duarte for his contributions in the development of UnitMarch and Sean Weaver for his comments.

References

1. Anbulagan, Duc Nghia Pham, John K. Slaney, Abdul Sattar, *Old Resolution Meets Modern SLS*. AAAI-05 (2005), 354–359.
2. D. Le Berre and L. Simon, *The essentials of the SAT03 Competition*. Springer Verlag, Lecture Notes in Comput. Sci. **2919** (2004), 452–467.
3. W. Blochinger, C. Sinz, and W. Kuchlin, *Parallel propositional satisfiability checking with distributed dynamic learning*. Parallel Computing, **29**(7) (2003), 969-994.
4. M. Boehm and E. Speckenmeyer, *A fast parallel SAT-solver - efficient workload balancing*. Ann. Math. Artif. Intell. **17**(3-4) (2006), 381-400.
5. Frank M. Brown. *Boolean Reasoning: The Logic of Boolean Equations*. Kluwer Academic Publishers, Dordrecht (1990)
6. E. A. Hirsch and A. Kojevnikov, *UnitWalk: A new SAT solver that uses local search guided by unit clause elimination*. Ann. Math. Artif. Intell. **43**(1) (2005), 91–111.
7. F. Krohm, A. Kuehlmann, and A. Mets. *The Use of Random Simulation in Formal Verification*. Proc. of Int'l Conf. on Computer Design (1996), 371–376.
8. Bart Selman, Henry Kautz, and Bram Cohen. *Local search strategies for satisfiability testing*. In David S. Johnson and Michael A. Trick, editors, Cliques, Coloring, and Satisfiability: the Second DIMACS Implementation Challenge (1996), 521-532.
9. H. Zhang, M. P. Bonacina, and J. Hsiang. *PSATO: A distributed propositional prover and its application to quasigroup problems*. Journal of Symbolic Computation **21** (1996), 543-560.