Journal on Satisfiability, Boolean Modeling and Computation 4 (2008) 99-116

Parallel SAT Solving using Bit-level Operations[*]

Marijn J.H. Heule[†] marijn@heule.nl
Hans van Maaren h.vanmaaren@ewi.tudelft.nl
Department of Software Technology,
Delft University of Technology,
The Netherlands

Abstract

We show how to exploit the 32/64 bit architecture of modern computers to accelerate some of the algorithms used in satisfiability solving by modifying assignments to variables in parallel on a single processor. Techniques such as random sampling demonstrate that while using bit vectors instead of Boolean values solutions to satisfiable formulae can be obtained faster. Here, we reveal that more complex algorithms, like unit propagation and detection of autarkies, can be parallelized efficiently, as well.

We capitalize on the developed parallel algorithms by modifying the state-of-the-art local search SAT solver UnitWalk accordingly. Experiments show that the parallel version performs much faster than the original implementation.

KEYWORDS: *local search, parallel computing, Boolean Algebra*

Submitted October 2007; revised January 2008; published May 2008

1. Introduction

State-of-the-art satisfiability (SAT) solvers can be divided into complete (solving both satisfiable and unsatisfiable formulae) and incomplete (solving only satisfiables) ones. The former class of solvers uses fast data-structures and reasoning techniques on partial assignments to solve problems. Surprisingly, they also dominate performance of incomplete solvers on most satisfiable structured instances[1]. Incomplete SAT solvers, mostly based on local search, mainly perform modifications on a (full) assignment using "randomized" flipping decisions. In general, these solvers are less complex. Incomplete solvers are very strong on satisfiable random benchmarks.

Today's 32/64 bit architecture enables computers to perform 32 or 64 of the familiar Boolean operations within a single clock cycle. Since assignment modifications can be considered Boolean operations, multiple of those modifications can be parallelized. Incomplete SAT solvers seem the most likely candidates to apply this technique, because they do not use reasoning techniques and because assignment modifications are an important aspect of the used algorithms.

∗ This paper is an extended version of: *From Idempotent Generalized Boolean Assignments to Multi-bit Search.* SAT 2007 Springer LNCS **4501** (2007), pp 134-147.

† Supported by the Dutch Organization for Scientific Research (NWO) under grant 617.023.611.

1. Based on the results on the SAT competitions. See www.satcompetition.org for details.

Current Satisfiability (Sat) solvers do not make use of the opportunity of a p-bit processor to simulate parallel 1-bit (Boolean) search on p 1-bit processors. Conventional parallel Sat solving [3, 4, 12] differs from the proposed method in Section 3: The former gains performance by dividing the workload over multiple processors and by some minor changes to the solving algorithm, while the latter uses a single processor and requires significant modifications to the algorithm. The most closely related work [7] also parallelizes a Sat solver (GSAT), on a single processor. However, they use a vector processor (used in most supercomputers), instead of scalar processor (used in most desktop computers).

Sat solvers that use integer type of heuristics frequently (counters for instance), are not very suitable for modification in this respect. However, Sat solvers whose computational "center of gravity" consists of propagating truth values (or other 1-bit operations) may profit from this opportunity. One of such solvers is the state-of-the-art local search Sat solver UnitWalk [6]. We show that UnitWalk can be upgraded using a single p-bit processor. This results in a considerable speed-up.

This paper, which is an extension [5], describes our Sat solver UnitMarch. The most important addition presented here are the communication enhancements (see Section 5). Most results originate from [5], but we also added some new experiments to show the usefulness of communication on some specific formulae.

2. Multi-Bit Assignments

The *satisfiability* (Sat) problem deals with the question whether there exits an assignment to the *Boolean variables* that satisfies a given *Boolean formula*. Such a formula is represented in *Conjunction Normal Form* (CNF): The formula consists of a conjunction of clauses (e.g. $\mathcal{F} = C_1 \wedge C_2 \wedge C_3$) and each clause consists of a disjunction of literals (e.g. $C_i = l_1 \vee l_2 \vee l_3$). Literals refer either to a Boolean variable x_i or to their complement $\neg x_i$. An assignment satisfies a formula if it satisfies all clauses. A clause is satisfied if at least on of its literals is satisfied. A literal x_i is satisfied if the corresponding variable is assigned to 1, while a literal $\neg x_i$ is satisfied if x_i is assigned to 0.

This paper explores the usefulness of assigning *bit vectors* $\{0,1\}^p$ instead of Boolean values to the variables. We refer to these bit vectors as *multi-bit values*. A non-zero multi-bit value refers to a bit vector containing at least one 1. An assignment which assigns multi-bit values to the variables is called a *multi-bit assignment* (in short MBA).

Example 1.

Consider the 3-bit values. We abbreviate multi-bit values: $(0, 1, 0)$ will be represented by 010. Let $\mathcal{F}$ be the formula

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

and assigning $x := 101$, $y := 001$ and $z := 111$, we calculate

$$\mathtt{101} \wedge \neg\mathtt{001} \wedge \big(\neg(\mathtt{101}) \vee \neg(\mathtt{111})\big) = \mathtt{000} \tag{2}$$

By assigning $x := 101$, $y := 001$ and $z := 011$ however, $\mathcal{F}$ evaluates to the value 100, as the reader may verify. All non-zero multi-bit values verify that the given formula is satisfiable.

Notice that if a certain clause gets a `0` in some bit position (by some partial multi-bit assignment) there is no possibility to extract a satisfying assignment from this bit position, because the `AND`'s of the CNF in this bit position can never undo this "being zero"!

Example 2.

Consider the 2-bit assignments on the formula $x \wedge y$. The reader may check that there are 16 possible 2-bit assignments of which 7 evaluate to a non-zero value. Drawing multi-bit assignments randomly, the probability of hitting a non-zero multi-bit value is $\frac{7}{16}$, while in the conventional Boolean situation this probability is $\frac{1}{4}$. In general, the probability is $1 - (\frac{3}{4})^p$ using the p-bit assignments.

The above example shows that probability to hit a solution using random sampling MBA's increases using more bits. In case multi-bit assignments can be used in approximately the same computational time as Booleans, solutions can be found faster. This is done in [8], where Boolean "patterns" (rather than Booleans) are propagated through a circuit to increase the probability of hitting a solution - indicating an error in their application.

Although this random sampling can be considered a rather straight forward parallelism, we claim that efficient multi-bit propagation for Sat solving is not straight forward at all: In [8], 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.

3. Multi-Bit Unit Propagation

This section describes the use of multi-bit assignments (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 left 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* [10].

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 $\varphi_{\text{master}}$ satisfies $F$ then
3:         break
4:     end if
5:     $\pi$ := random ordering of the variables
6:     $\varphi_{\text{active}} := \emptyset$
7:     for $j$ in 1 to $n$ do
8:         \\ Perform unit propagation
9:         while unit clause $u \in \varphi_{\text{active}} \circ F$ do
10:            $\varphi_{\text{active}}[\ VAR(u)\ ] := TRUTH(u)$
11:        end while
12:        \\ Assign the next free variable according to $\pi_j$
13:        if $\pi(j)$ not assigned in $\varphi_{\text{active}}$ then
14:            $\varphi_{\text{active}}[\ \pi(j)\ ] := \varphi_{\text{master}}[\ \pi(j)\ ]$
15:        end if
16:    end for
17:    if $\varphi_{\text{active}} = \varphi_{\text{master}}$ then
18:        random flip variable in $\varphi_{\text{active}}$
19:    end if
20:    $\varphi_{\text{master}} := \varphi_{\text{active}}$
21: end for
22: return $\varphi_{\text{master}}$
```

Example 3.

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

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

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[2.] 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 $\neg x_2 \vee x_3 \vee x_4$.

Now, consider the same example, this time assigning 4-bit values to all the variables. The reader must keep in mind that by parallelizing the former, we aim to satisfy clauses in each bit position! Recall that once a certain clause gets a `0` at some bit position, no satisfying assignment is possible at that 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*. In the multi-bit case, a clause is called unit with respect to a certain bit position if at that bit position one literal is unassigned and all others are falsified. So, a clause can be(come) unit on multiple bit positions and on different literals at the same time. Further, we keep using the term "truth value" for its multi-bit analogue. Notice that in the initial settings below, the first (most left) bit in φ_{master} equals the 1-bit example and that the ordering is the same. For clarity, in the example recent changes are shown in bold and an underlined bit position means refers to a bit-conflict.

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

2. 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.

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

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

Notice that both variables are assigned immediately, although alternative implementations are possible - see Section 4.1. One of them is selected, say x_1, and assigned to its value:

$$(\neg x_1 = \texttt{**00} \vee x_2 = \texttt{1100} \vee x_3 = \texttt{00**}) \quad \Rightarrow \quad x_3 := \texttt{00}\mathbf{\texttt{11}}$$

Now we assign x_3 which triggers three clauses:

$$\begin{array}{rcl} (\neg x_2 = \texttt{0011} \vee x_3 = \texttt{0011} \vee \neg x_4 = \texttt{****}) & \Rightarrow & x_4 := \mathbf{\texttt{00}}\texttt{**} \\ (\neg x_2 = \texttt{0011} \vee x_3 = \texttt{0011} \vee x_4 = \texttt{00**}) & \Rightarrow & x_4 := \underline{\texttt{00}}\texttt{**} \text{ (bit}-\text{conflict)} \\ (\neg x_3 = \texttt{1100} \vee \neg x_4 = \texttt{11**}) & \Rightarrow & x_4 := \texttt{00}\mathbf{\texttt{00}} \end{array}$$

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. Only in case conflicts occur, the order influences φ_{active}. 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 formulae with a small number of variables. To counter this unwanted effect, we added a technique removing duplicates - see Section 5.1.

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 need 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 u from the multi-set is selected. If the complement of the selected unit clause also occurs in the multi-set - indicating a conflict - all occurrences of u and $\neg u$ 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: The truth value for u in φ_{active} tends to be copied from φ_{master}.

Algorithm 2 UnitPropagation_MultiSet ( )

```
while UnitMultiSet is not empty do
    $u$ := random element from UnitMultiSet
    remove all occurrences of $u$ in UnitMultiSet
    if unit clause $\neg u$ also occurs in UnitMultiSet then
        remove all occurrences of $\neg u$ in UnitMultiSet
    else
        $\varphi_{\text{active}}[\ VAR(u)\ ] := TRUTH(u)$
        for all clauses $C_i$ in which $\neg u$ occurs do
            if $C_i$ becomes a unit clause then
                add $C_i$ to UnitMultiSet
            end if
        end for
    end if
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 in which they are added to the queue. In general, "early" generated unit clauses will have more bits assigned (at the 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 6), both implementations yield comparable results (the average number of periods).

Algorithm 3 UnitPropagation_Queue ( )

```
while UnitQueue is not empty do
    $u$ := removed front element from UnitQueue
    for all clauses $C_i$ in which $\neg u$ occurs do
        if $C_i$ becomes a unit clause then
            $v$ := remaining literal in $C_i$
            $\varphi_{\text{active}}[\ VAR(v)\ ] := TRUTH(v)$
            if $v$ not in UnitQueue then append $v$ to UnitQueue
        end if
    end for
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. Recall that in the 1-bit situation, a potential unit clause can only be unit on a single literal, while in a multi-bit implementation it can become unit on multiple literals (each on a different bit position).

Example 4.

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

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

In general, a clause can become unit on all literals - apart from the propagation literal.

4.2.1 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**[3]. 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 $\neg x_i$. Back to the example.

$$\varphi_{\text{active}} \text{ is stored as } \begin{cases} \varphi^+_-[\ x_1] = \texttt{0100}, \varphi^+_-[\ x_2] = \texttt{1001}, \varphi^+_-[\ x_3] = \texttt{1010}, \varphi^+_-[\ x_4] = \texttt{0001} \\ \varphi^+_-[\neg x_1] = \texttt{1010}, \varphi^+_-[\neg x_2] = \texttt{0100}, \varphi^+_-[\neg x_3] = \texttt{0100}, \varphi^+_-[\neg x_4] = \texttt{0110} \end{cases}$$

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.

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

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(s) 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 into two stages:

- Compute the *unit mask* of a clause - a multi-bit value which is true on all positions with exactly one not falsified literal (denoted by $M_{\text{NF}=1}$) and false elsewhere;
- 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.

3. The bit-conflict value is not possible within or implementation

To compute $M_{\mathrm{NF}=1}$, we use two auxiliary masks, $M_{\mathrm{NF}<1}$ and $M_{\mathrm{NF}<2}$. The masks denote multi-bit values which are `1` on all positions with less than one (and two, respectively) not falsified literals and `0` elsewhere. Notice that $M_{\mathrm{NF}=1} := M_{\mathrm{NF}<1}$ XOR $M_{\mathrm{NF}<2}$. For each literal l_i in a clause we update $M_{\mathrm{NF}<1}$ and $M_{\mathrm{NF}<2}$ by the following two rules:

$$\begin{aligned} M_{\mathrm{NF}<2} &:= (M_{\mathrm{NF}<2} \texttt{ AND } \varphi^{+}_{-}[\,\neg l_{y,i}\,]) \texttt{ OR } M_{\mathrm{NF}<1} \\ M_{\mathrm{NF}<1} &:= M_{\mathrm{NF}<1} \texttt{ AND } \varphi^{+}_{-}[\,\neg l_{y,i}\,] \end{aligned}$$

The implementation of the above is shown in Algorithm 4.

Algorithm 4 ComputeUnitMask (clause C_y)

```
1: $M_{\mathrm{NF}<1}$ := ALL_BITS_TRUE, $M_{\mathrm{NF}<2}$ := ALL_BITS_TRUE
2: for $i$ in 1 to $|C_y|$ do
3:     $M_{\mathrm{NF}<2} := (M_{\mathrm{NF}<2}$ AND $\varphi^{+}_{-}[\,\neg l_{y,i}\,])$ OR $M_{\mathrm{NF}<1}$
4:     $M_{\mathrm{NF}<1} := M_{\mathrm{NF}<1}$ AND $\varphi^{+}_{-}[\,\neg l_{y,i}\,]$
5: end for
6: return $M_{\mathrm{NF}<1}$ XOR $M_{\mathrm{NF}<2}$
```

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

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

5. Communication

The above description of a multi-bit version of the UnitWalk algorithm can be seen as performing the algorithm in parallel without communication. However, communication can be added to the algorithm to possibly further extend performance gain. This section offers two kinds of communication. The first is a parallel detection algorithm for duplicate assignments. This feature repairs an unwanted effect of the UnitWalkalgorithm. Therefore this communication is not really an enhancement but more an essential addition. The second is a parallel algorithm to compute the largest autarky in a given (full) assignment.

5.1 Duplicate assignments

During our experiments we frequently observed convergence of the different bit positions in an assignment. For a given assignment φ, the j-th bit position is called a *duplicate* if there exists a $i < j$ such that all variables are assigned to the same truth value at bit position i and j. On most benchmarks, duplicates were observed. In some cases even (all) $n-1$ bit positions became duplicate. Due to the construction of the UnitWalk algorithm, once a bit position is a duplicate, it will remain a duplicate if no intervention is made. Because duplicates reduce the parallel behavior of the algorithm, we decided to detect duplicates and replace them with a new random assignment.

To detect the duplicates, we used *assignment matrices*: The assignment matrix $M_\varphi(x_i)$ of a variable x_i for a p-bit assignment φ is a symmetric $n \times n$ 0,1-matrix of which each j-th row and column is $\varphi[x_i]$ if x_i is assigned to true on the j-th bit-position and $\varphi[\neg x_i]$ otherwise. The assignment matrix $M_\varphi(\mathcal{F})$ is the entrywise product (so called Hadamard product, denoted by $\circ$) of the the assignment matrices of all the variables in $\mathcal{F}$.

Example 5.

Given $\varphi = \{x_1 = \texttt{010010}, x_2 = \texttt{101101}, x_3 = \texttt{110111}, x_4 = \texttt{000000}\}$. Now we compute the assignment matrices:

$$M_\varphi(x_1) = \begin{bmatrix} 1\,0\,1\,1\,0\,1 \\ 0\,1\,0\,0\,1\,0 \\ 1\,0\,1\,1\,0\,1 \\ 1\,0\,1\,1\,0\,1 \\ 0\,1\,0\,0\,1\,0 \\ 1\,0\,1\,1\,0\,1 \end{bmatrix} \quad M_\varphi(x_2) = \begin{bmatrix} 1\,0\,1\,1\,0\,1 \\ 0\,1\,0\,0\,1\,0 \\ 1\,0\,1\,1\,0\,1 \\ 1\,0\,1\,1\,0\,1 \\ 0\,1\,0\,0\,1\,0 \\ 1\,0\,1\,1\,0\,1 \end{bmatrix}$$

$$M_\varphi(x_3) = \begin{bmatrix} 1\,1\,0\,1\,1\,1 \\ 1\,1\,0\,1\,1\,1 \\ 0\,0\,1\,0\,0\,0 \\ 1\,1\,0\,1\,1\,1 \\ 1\,1\,0\,1\,1\,1 \\ 1\,1\,0\,1\,1\,1 \end{bmatrix} \quad M_\varphi(x_4) = \begin{bmatrix} 1\,1\,1\,1\,1\,1 \\ 1\,1\,1\,1\,1\,1 \\ 1\,1\,1\,1\,1\,1 \\ 1\,1\,1\,1\,1\,1 \\ 1\,1\,1\,1\,1\,1 \\ 1\,1\,1\,1\,1\,1 \end{bmatrix} \quad \Rightarrow \quad M_\varphi(\mathcal{F}) = \begin{bmatrix} 1\,0\,0\,1\,0\,1 \\ 0\,1\,0\,0\,1\,0 \\ 0\,0\,1\,0\,0\,0 \\ 1\,0\,0\,1\,0\,1 \\ 0\,1\,0\,0\,1\,0 \\ 1\,0\,0\,1\,0\,1 \end{bmatrix}$$

Notice that all assignment matrices $M_\varphi(x_i)$ have at least as many 1's as 0's. If a row contains 1's in the lower triangle of $M_\varphi(\mathcal{F})$, the corresponding bit position is a duplicate. In the example above, the 4-th, 5-th and 6-th bit positions are duplicates. Using $M_\varphi(\mathcal{F})$ we can obtain $m_{\text{duplicates}}$: Compute the Hadamard product of the strictly lower triangular matrix and $M_\varphi(\mathcal{F})$. Multiply the result with the all one vector. The resulting mask $m_{\text{duplicates}}$ is a p-bit Boolean which has 1's on all bit positions that are duplicates and 0's otherwise. In this example the computation is:

$$m_{\text{duplicates}} = \left(\begin{bmatrix} 0\,0\,0\,0\,0\,0 \\ 1\,0\,0\,0\,0\,0 \\ 1\,1\,0\,0\,0\,0 \\ 1\,1\,1\,0\,0\,0 \\ 1\,1\,1\,1\,0\,0 \\ 1\,1\,1\,1\,1\,0 \end{bmatrix} \circ \begin{bmatrix} 1\,0\,0\,1\,0\,1 \\ 0\,1\,0\,0\,1\,0 \\ 0\,0\,1\,0\,0\,0 \\ 1\,0\,0\,1\,0\,1 \\ 0\,1\,0\,0\,1\,0 \\ 1\,0\,0\,1\,0\,1 \end{bmatrix} \right) \begin{bmatrix} 1 \\ 1 \\ 1 \\ 1 \\ 1 \\ 1 \end{bmatrix} = \begin{bmatrix} 0\,0\,0\,1\,1\,1 \end{bmatrix} \qquad (3)$$

In UnitMarch $m_{\text{duplicates}}$ is computed as in Algorithm 5 - for p-bit assignments. The algorithm is similar to the above example. Let n denote the number of variables. Although the algorithm has worst case complexity $\mathcal{O}(pn)$, in practice it is quite fast due to the break command at line 11.

Algorithm 5 ComputeDuplicateMask (assignment φ)

```
1: $m_{\text{duplicates}} := [0]^p$
2: for $j$ in 1 to $p-1$ do
3:     $m_{\text{column}} := [0]^j[1]^{p-j}$
4:     for $x_i \in \mathcal{F}$ do
5:         if $x_i$ is assigned to true on the $j$-th bit-position in $\varphi$ then
6:             $m_{\text{column}} := m_{\text{column}}$ AND $\varphi[x_i]$
7:         else
8:             $m_{\text{column}} := m_{\text{column}}$ AND $\varphi[\neg x_i]$
9:         end if
10:        if $m_{\text{column}} = [0]^p$ then
11:            break
12:        end if
13:    end for
14:    $m_{\text{duplicates}} := m_{\text{duplicates}}$ OR $m_{\text{column}}$
15: end for
16: return $m_{\text{duplicates}}$
```

5.2 Autarkies

An *autarky* (or autark assignment) is a partial assignment φ that satisfies all clauses that are "touched" (have at least one literal assigned) by φ. So, all satisfying assignments are autark assignments. Autarkies that do not satisfy all clauses can be used to reduce the size of the formula: Let $\mathcal{F}_{\text{touched}}$ be the clauses in $\mathcal{F}$ that are satisfied by an autarky. The remaining formula $\mathcal{F}^* := \mathcal{F} \setminus \mathcal{F}_{\text{touched}}$ is satisfiability equivalent to $\mathcal{F}$. If we detect an autark assignment we can reduce $\mathcal{F}$ by removing all clauses in $\mathcal{F}_{\text{touched}}$.

Given an assignment, one can compute the largest autarky being a reduction of that assignment using the following algorithm [9]:

- Loop through all the clauses
- If a clause is touched but not satisfied, unassign all variables in that clause
- Repeat the above until no assignment changes have been made

The outcome of the algorithm is either an empty assignment, showing that there exists no autarky which is a reduction of the input assignment, or some variables are still assigned which form an autarky. Notice that the algorithm is confluent: All variables that occur in any autarky being a reduction of the input assignment will be in the output. The larger the number of assigned variables of the input assignment, the higher the probability that algorithm will return an autarky. Especially local search Sat solvers - such as UnitWalk - are likely to profit from the algorithm, since at each period they work with a full assignment.

The above algorithm can easily be parallelized using MBA's: Check whether at one or more bit positions the clause is touched but not satisfied. Then unassign all variables on those bit positions. Again, repeat the above until no assignment changes have been made. The resulting assignment is either empty or contains an autarky at one or more bit positions.

Parallelizing the algorithm has two main advantages: First, since it is easy to perform the detection in parallel, the costs are relatively small. Second, if an autarky is found on a single bit position, clauses can be removed from the formula which will reduce the the propagation costs of the entire solving procedure. Therefore, detecting autarkies and removing clauses in parallel, could (at least in theory) result in a significant speed-up.

Example 6.

To explain the multi-bit autarky detection, we start by using a slightly modified example formula from the multi-bit unit propagation example and the same initial φ_{master}. In this example $\circledast$ denotes a bit position that has recently been unassigned.

$$\begin{aligned}
\mathcal{F}^*_{\text{example}} &:= (x_1 \vee x_2) \wedge (\neg x_1 \vee \neg x_2 \vee x_3) \wedge (\neg x_2 \vee \neg x_3)\\
&\quad\ (\neg x_2 \vee x_3 \vee \neg x_4) \wedge (\neg x_2 \vee x_3 \vee x_4) \wedge (\neg x_3 \vee \neg x_4)\\
\varphi_{\text{master}} &:= \{x_1 = \texttt{0110}, x_2 = \texttt{1100}, x_3 = \texttt{1010}, x_4 = \texttt{0110}\}
\end{aligned}$$

First, we loop once through all the clauses. If a clause is not satisfied on a certain bit position all variables in that clause are unassigned at that bit position:

$$\begin{aligned}
(x_1 = \texttt{0110} \vee x_2 = \texttt{1100}) &\quad\Rightarrow\quad x_1 := \texttt{011}\circledast, x_2 := \texttt{110}\circledast\\
(\neg x_1 = \texttt{100}* \vee \neg x_2 = \texttt{001}* \vee x_3 = \texttt{1010}) &\quad\Rightarrow\quad x_1 := \texttt{0}\circledast\texttt{1}*, x_2 := \texttt{1}\circledast\texttt{0}*, x_3 := \texttt{1}\circledast\texttt{1}\circledast\\
(\neg x_2 = \texttt{0}*\texttt{1}* \vee \neg x_3 = \texttt{0}*\texttt{0}*) &\quad\Rightarrow\quad x_2 := \circledast*\texttt{0}*, x_3 := \circledast*\texttt{1}*\\
(\neg x_2 = **\texttt{1}* \vee x_3 = **\texttt{1}* \vee \neg x_4 = \texttt{1001}) &\quad\Rightarrow\quad x_4 := \texttt{0}\circledast\texttt{10}\\
(\neg x_2 = **\texttt{1}* \vee x_3 = **\texttt{1}* \vee x_4 = \texttt{0}*\texttt{10}) &\quad\Rightarrow\quad x_4 := \circledast*\texttt{1}\circledast\\
(\neg x_3 = **\texttt{0}* \vee \neg x_4 = **\texttt{0}*) &\quad\Rightarrow\quad x_3 := **\circledast*, x_4 := **\circledast*
\end{aligned}$$

Second, we loop again through the clauses. This will unassign one more bit position:

$$(x_1 = \texttt{0}*\texttt{1}* \vee x_2 = **\texttt{0}*) \quad\Rightarrow\quad x_1 := \circledast*\texttt{1}*$$

Since no assignments are unassigned by the other clauses, the algorithm stops. The presence of assigned variables x_1 and x_2 indicate that we found an autarky on bit position 3. This autarky satisfies all clauses except $\neg x_3 \vee \neg x_4$. Since the remaining clause is satisfiability equivalent to $\mathcal{F}^*_{\text{example}}$, the satisfied clauses can be removed from the formula and we can continue solving only the reduced formula. The example shows that detection of autarkies can reduce the formula considerably and speed-up the solving time.

Detection of autarkies can be implemented more efficiently compared to the description above: Only in the first iteration, one needs to loop through all the clauses. In succeeding iterations, only those clauses that contain a variable that was unassigned (at some bit position) in the prior iteration need to be examined. Another technique to reduce the computational costs of the detection algorithm is to call it once every k periods. In case an autarky exists on some bit position(s), the UnitWalk algorithm will not alter the truth values on those bit positions of the variables contributing to the autarky. Therefore, calling the detection algorithm every once in a while will reveal the same autarkies - although slightly later.

6. 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 the method which detects and replaces duplicates with new random assignments (see Section 5.1). Because the autarky detection feature (see Section 5.2) only slightly influences the performance on the selected benchmarks, we decided to present the results from [5]. The performance of UnitMarch is compared with the latest version of UnitWalk[4].

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 [11]. In turn, if that algorithm does not find a solution after a multitude of flips it switches back, etc. We wanted to compare the influence of multi-bit search on the pure UNITWALK algorithm, so switching was disabled.

Table 1 shows a comparison between UnitWalk, UnitMarch 1-bit and UnitMarch 32-bit on various benchmarks. Apart from the *dlx2-bugXX* family[5], all benchmarks can be found on SATlib[6] 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 the same 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, the results are "too close to call": There is no clear winner. Hence, the UNITPROPAGATION_QUEUE procedure shows comparable to the UNITPROPAGATION_MULTISET procedure in terms of performance.

Comparing both 1-bit solvers to 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 simple 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. On instances such as the parity benchmarks, it could be expected that computers with a p-bit architecture with $p > 32$ will boost performance even further.

Although the detection of autarkies sporadically influenced the results on the selected benchmarks, we present the usefulness of this technique using a separate experiment: We concatenated multiple satisfiable random 3-SAT formulae[7] such that each formula uses different variables. Each concatenated formula consists of multiple components and for each component there exists an autarky satisfying only the component. Experiments on similar formulae is discussed in [2].

4. version 1.003 available from http://logic.pdmi.ras.ru/~arist/UnitWalk/
5. available from http://www.miroslav-velev.com/sat_benchmarks.html
6. http://www.satlib.org
7. with 200 variables and 860 clauses, also from 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)

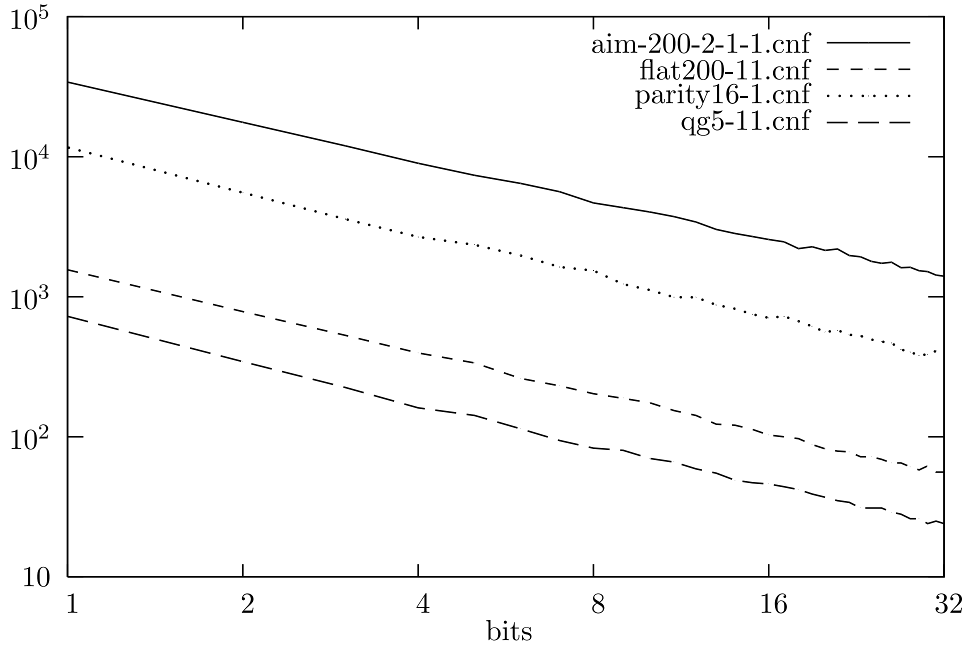

Figure 1. Average number of periods by UnitMarch using different number of bits. Averages are computed using 1000 random seeds. Two logarithmic axes are used.

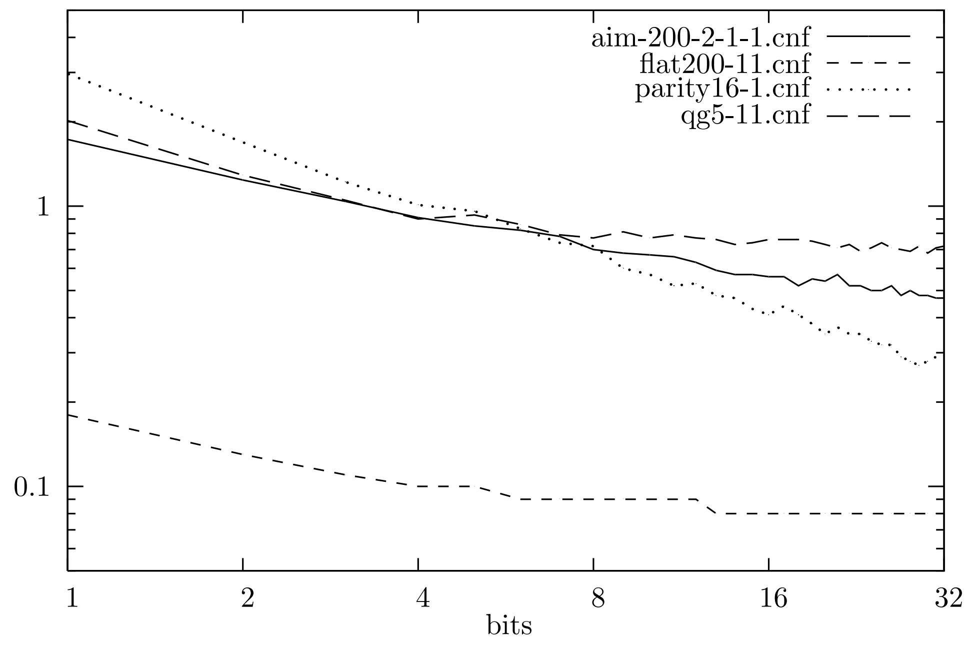

Figure 2. Average time (in seconds) by UnitMarch using different number of bits. Averages are computed using 1000 random seeds. Two logarithmic axes are used.

The performance of UnitMarch on these formulae - with and without the autarky feature - is shown in Figure 3. The version with autarky detection is orders of magnitude faster. Also, the larger the number of components, the larger the speed-up factor realized by the technique. So, if formulae consist of independent components, they can be solved much faster using detection of autarkies. Practical applications for this technique are under current research.

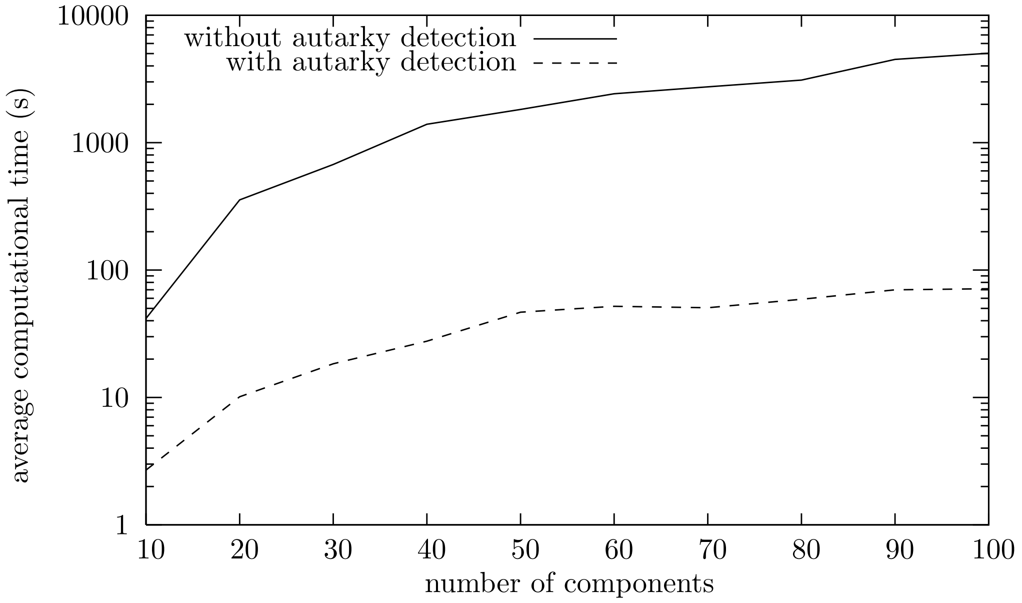

Figure 3. Performance of UnitMarch 32-bit with and without autarky detection on concatenated formulae of random 3-Sat instances.

7. Conclusions and future work

Our first observation is that the probability of hitting a solution of propositional Boolean formulae is increased by assigning multi-bit values instead of Boolean values. 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 suitable first candidate, as well as a state-of-the-art Sat solver [10].

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 shows comparable performance to 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 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

R$^+$AdaptNovelty$^+$ [1] - because they will increase the number of unit propagations. Finally, further experiments (not presented in this paper) showed that 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 detection of autarkies) 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, and Abdul Sattar. Old resolution meets modern SLS. In Manuela M. Veloso and Subbarao Kambhampati, editors, *AAAI*, pages 354–359. AAAI Press / The MIT Press, 2005.

[2] Armin Biere and Carsten Sinz. Decomposing SAT problems into connected components. *Journal on Satisfiability, Boolean Modeling and Computation*, **2**:191–198, 2006.

[3] Wolfgang Blochinger, Carsten Sinz, and Wolfgang Küchlin. Parallel propositional satisfiability checking with distributed dynamic learning. *Parallel Comput.*, **29**(7):969–994, 2003.

[4] Max Böhm and Ewald Speckenmeyer. A fast parallel SAT-solver - efficient workload balancing. *Annals of Mathematics and Artificial Intelligence*, **17**(159):381–400, 1996.

[5] Marijn J.H. Heule and Hans van Maaren. From idempotent generalized boolean assignments to multi-bit search. In João Marques-Silva and Karem A. Sakallah, editors, *SAT*, **4501** of *Lecture Notes in Computer Science*, pages 134–147. Springer, 2007.

[6] Edward A. Hirsch and Arist Kojevnikov. UnitWalk: A new SAT solver that uses local search guided by unit clause elimination. *Annals of Mathematics and Artificial Intelligence*, **43**(1-4):91–111, 2005.

[7] Kazuo Iwama, Daisuke Kawai, Shuichi Miyazaki, Yasuo Okabe, and Jun Umemoto. Parallelizing local search for CNF satisfiability using vectorization and PVM. *ACM Journal of Experimental Algorithms*, **7**:2, 2002.

[8] Florian Krohm, Andreas Kuehlmann, and Arjen Mets. The use of random simulation in formal verification. In *ICCD*, pages 371–376. IEEE Computer Society, 1996.

[9] Oliver Kullmann, Victor W. Marek, and Miroslaw Truszczyński. Computing autarkies and properties of the autarky monoid, 2007. In preparation.

[10] Daniel LeBerre and Laurent Simon. The essentials of the SAT 2003 competition. In Enrico Giunchiglia and Armando Tacchella, editors, *SAT*, **2919** of *Lecture Notes in Computer Science*, pages 452–467. Springer, 2003.

[11] Bart Selman, Henry A. Kautz, and Bram Cohen. Noise strategies for improving local search. In *AAAI '94: Proceedings of the twelfth national conference on Artificial intelligence (vol. 1)*, pages 337–343, Menlo Park, CA, USA, 1994. American Association for Artificial Intelligence.

[12] Hantao Zhang, Maria Paola Bonacina, and Jieh Hsiang. PSATO: a distributed propositional prover and its application to quasigroup problems. *Journal of Symbolic Computation*, **21**(4):543–560, 1996.