Solving Huge Instances with Intel® SAT Solver

Alexander Nadel
Intel Corporation, Haifa, Israel
Faculty of Data and Decision Sciences, Technion, Haifa, Israel

Abstract

We introduce a new release of our SAT solver `Intel® SAT Solver`. The new release, called `IS23`, is targeted to solve huge instances beyond the capacity of other solvers. `IS23` can use 64-bit clause-indices and store clauses compressedly using bit-arrays, where each literal is normally allocated fewer than 32 bits. As a preliminary result, we show that only `IS23` can handle a gigantic trivially satisfiable instance with over 8.5 billion clauses. Then, we demonstrate that `IS23` enables a significant improvement in the capacity of our industrial tool for cell placement in physical design, since only `IS23` can solve placement instances with up to 4.3 billion clauses. Finally, we show that `IS23` is substantially more efficient than other solvers for finding many (10^6) placements on instances with up to 170 million clauses. We use the latter application to demonstrate that *variable succession*, that is, the order in which the variables are provided to the solver, might have a significant impact on `IS23`'s performance, thereby providing a new dimension to SAT encoding considerations.

2012 ACM Subject Classification Mathematics of computing → Solvers

Keywords and phrases SAT, CDCL

Digital Object Identifier 10.4230/LIPIcs.SAT.2023.17

Supplementary Material *Software (Source Code)*: `https://github.com/alexander-nadel/intel_sat_solver`, archived at `swh:1:dir:f665615bfa3f6d0af72cf9e1b4ec027c57a9f4c7`

1 Introduction

A SAT solver decides the classical NP-complete problem of whether the given propositional formula F in Conjunctive Normal Form (CNF)[1] is satisfiable. Modern Conflict-Driven-Clause-Learning (CDCL) SAT solvers are widely used [5]. They implement backtrack search, enhanced by *conflict clause* learning and many other techniques.

SAT research is mostly focused on developing algorithms for solving, within the given timeout, empirically difficult, but not necessarily large benchmarks. In this study, we targeted improving the SAT capacity to enable solving huge instances with billions of clauses (cf. the size of the instances in the main track of the latest SAT competition 2022 [2] ranged from 264 to 214,309,011 clauses with 7,117,471 being the average). SAT solvers might fail on huge instances due to limitations related to memory management, uncharacteristic for other use-cases. To better understand these limitations, recall how SAT solvers manage clauses.

Long clauses (that is, clauses having at least 3 literals) are stored in the so-called *clause buffer*. An initial clause C is typically represented by $(1 + |C|)$ 32-bit words, containing C's size, followed by C's literals (where conflict clauses have some extra-fields). Let *variable succession* be the order in which the variables are provided to the solver. The internal indices, which represent variables and literals, depend on the variable succession. In most solvers since `MiniSat` [9], a positive literal v_i (where i reflects its order in the succession) is represented by the 32-bit *literal-index* $li(v_i) = 2 \times i$, while a negative literal $\neg v_i$ is represented by $li(\neg v_i) = 2 \times i + 1$. For example, consider the following formula E:

$$E = (C_1 = v_1 \vee v_2 \vee \neg v_3) \wedge (C_2 = \neg v_1 \vee v_2 \vee v_3)$$

[1] A CNF formula is a conjunction of clauses (aka, *initial clauses*), each *clause* being a disjunction of Boolean literals, where a *literal* is a Boolean variable or its negation.

E would be represented in the clause buffer by the following eight 32-bit words:

$$\left\langle \overbrace{\boxed{3}}^{|C_1|}, \overbrace{2,4,7}^{C_1}, \overbrace{\boxed{3}}^{|C_2|}, \overbrace{3,4,6}^{C_2} \right\rangle$$

In order to uniquely identify and access a clause C, solvers use its *clause-index* ci(C) in the clause buffer. In our example, $ci(C_1)$=0 and $ci(C_2)$=4.

The fundamental capacity limitation of the older solvers (such as `MiniSat` and `Glucose` [1]), but also some of the most modern solvers (such as `MergeSat` [15] and the baseline solver for recent SAT competition winners `Kissat` [4]) is caused by their clause-index width being limited to 32 bits or even fewer due to additional bookkeeping (e.g., 31 bits in `Kissat`).

The first open-source solver to offer a 64-bit-clause-index version was `CryptoMiniSat` [25], which can be compiled with a 64-bit clause-index since May 2017 [24]. A 64-bit clause-index is also used by `CaDiCaL` [4]. Although using 64-bit clause-indices eliminates the major SAT capacity limitation while not affecting the size of the clause buffer, it comes with the price of inflating data structures which point to clauses (notably, including the Watch Lists (WLs) [17], which contain two clause-indices per clause), thus increasing the solver's memory consumption.

`Intel® SAT Solver` (`IntelSAT`) is our CDCL SAT solver, which we released as open-source last year [18]. We optimized it for incremental SAT solving in the presence of many satisfiable queries. The original `IntelSAT` uses a 32-bit clause-index. This paper introduces a new release of `IntelSAT` – `IS23`, aimed at extending the solver's capacity. `IS23` can be compiled into various versions, including the default `IS23` (similar to the original `IntelSAT`), `IS23-64` and `IS23-64C`. `IS23-64` extends the clause-index width from 32 bits to 64 bits. `IS23-64C` uses *bit-arrays* to store clauses *compressedly*, where the goal is to reduce the memory footprint (thus, potentially, also reducing the number of cache misses) at the expense of applying additional bit-wise operations to access clauses.

We demonstrate our core idea on our example formula $E = (C_1 = v_1 \vee v_2 \vee \neg v_3) \wedge (C_2 = \neg v_1 \vee v_2 \vee v_3)$. Given a clause C, let its *literal-width* $lw(C)$ be the minimal number of bits required to store its highest literal index. To store C, we allocate its every literal $lw(C)$ bits. Observe that, in E, we have $lw(C_1) = lw(C_2) = 3$, thus the formula (without the clause sizes) can be represented using 18 bits as $\left\langle \overbrace{\underbrace{010}_{2}, \underbrace{100}_{4}, \underbrace{111}_{7}}^{C_1}; \overbrace{\underbrace{011}_{3}, \underbrace{100}_{4}, \underbrace{110}_{6}}^{C_2} \right\rangle$ (in binary encoding), which requires only one 32-bit word instead of eight. Apparently, to access clause's literals, the literal-width must be known upfront. To support clauses with arbitrary literal-widths, `IS23-64C` stores clauses in multiple bit-arrays, where all the clauses in a single bit-array share the same literal-width (along with two other fields as detailed in Sect. 3). In `IS23-64C`, the 64-bit clause-index of every clause C contains the unique ID of C's bit-array (11 bits) and the bit number where C starts in its bit-array (the remaining 53 bits).

Notably, the `sharpSAT` model counter [26] first applied the idea of storing subsets of clauses (aka components) compressedly by limiting the number of bits in every clause to the maximal literal-width in that component. However, while `sharpSAT` only stashed the components compressedly for future usage, we have implemented a full-fledged CDCL SAT solver with the compressed clause buffer as the underlying data structure.

We carried out several experiments to evaluate the different versions of `IS23` against other solvers, including `Kissat`, `CaDiCaL`, `CryptoMiniSat` and `MergeSat`.

In our first preliminary experiment, we show that only `IS23-64C` can solve a huge trivially satisfiable instance having 2^{33} = 8,589,934,592 clauses and 292,057,776,128 literals overall (in all the input clauses).

Our own interest in extending the capacity of SAT stems from our industrial placement application. Cell placement is one of the most important problems in VLSI automation [23]. Its most basic version concerns placing without overlap a set of rectangles on a grid. In [8], we have presented our SAT-based placement tool, which starts with finding one placement and then optimizes it with incremental SAT queries. We initiated the development of `IS23`, since we had been observing an increasing number of cases where our tool failed to find even the initial placement due to capacity limitations of `IntelSAT`. Furthermore, recently, we encountered the need to solve another flavor of the placement problem, we call *N-placements*: find a user-given number of different placements (from which promising placements are subsequently selected and might be further optimized). In the rest of paper, we consider the problems of finding 1 or $N > 1$ placements, leaving optimization outside of our scope.

In our second experiment, we show that only with `IS23` can we find one placement for huge problems, whose corresponding CNF instances have up to 4.3 billion clauses.

In our third experiment, we show that only `IS23-64C` can find 1,000,000 placements for instances having up to 170 million clauses, where, to achieve the best results, the variable succession scheme must be carefully chosen.

The rest of this paper is organized as follows. Sect. 2 presents preliminaries. Sect. 3 introduces `IS23`. Sect. 4 is about experimental results. Sect. 5 concludes our work.

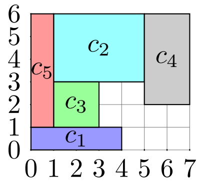
2 Preliminaries

A *literal* l is a Boolean variable v, in which case l is *positive*, or a variable's negation $\neg v$, in which case l is *negative*. A *clause* is a disjunction of literals. Let an *n-clause* and *>n-clause* be a clause of size n and >n, respectively. A *long clause* is a >2-clause; a *binary clause* is a 2-clause.

A formula F is in *Conjunctive Normal Form (CNF)* if it is a conjunction (set) of clauses. A SAT solver receives a CNF formula F and returns a satisfying assignment (aka, model or solution) μ, which assigns a Boolean value $\mu(v) \in \{0, 1\}$ to every variable v, where $\mu(\neg v) = \neg\mu(v)$. For a literal l, let *the projection of l in μ* $_{\mu}l \in \{l, \neg l\}$ be l iff $\mu(l) = 1$ or, otherwise, $\neg l$.

In *incremental SAT solving (under assumptions)* [9, 21], the user may invoke the solver multiple times, each time with a new set of zero or more *assumption literals* (called, simply, the *assumptions*), while adding zero or more clauses in-between the queries. The solver then checks the satisfiability of all the clauses provided so far, while enforcing the values of the current assumptions.

Cell placement (placement) is one of the most important problems in VLSI automation [23]. We consider the following basic (but already NP-complete [13]) version which concerns placing without overlap a set of rectangles on a grid. The input of the placement problem comprises the following two components: a *rectangular grid region* of fixed size and a finite set of *rectangular cells* of user-given widths and heights. We are interested in *feasible* placements, that is, placements in which no cell overlaps other cells. An example of a feasible placement is shown below (placing five cells of sizes 4×1, 4×3, 2×2, 2×4 and 1×5 on a 7×6 grid):



To encode placement into SAT, first, we associate two *bitvectors* c^l and c^b with the left-bottom coordinate (c^l, c^b) of every cell c (where a *Bitvector (BV)* $b = \{b_n, b_{n-1}, \ldots, b_1\}$ is a sequence of $|b|$ Boolean variables, called *bits*). Second, we create two sets of constraints in BV logic [3] over c^l's and c^b's to ensure that all the cells are inside the grid and there is no overlap. Third, we apply an eager BV solver [10], which, after preprocessing, translates the formula to SAT and solves it using a SAT solver. We refer the reader to [8] for all the details.

This paper also considers the *N-placement* problem of finding a user-given number of placements. To solve N-placement, we apply the following algorithm, we call `SimpleBlock` (first proposed in [16] in the context of model checking). `SimpleBlock`, shown below, iteratively finds a solution (placement) μ and immediately blocks it using a single *blocking clause* containing the falsified literal per every *important variable*, where, in our case, the set of the important variables comprises all the bits of the left-bottom coordinates of every cell:

1: Create a CNF formula F representing the given problem.
2: Invoke a SAT solver over F. Let μ be the returned model (if any).
3: **while** F is satisfied with μ and the user-given solution threshold N not reached **do**
4: Block the current solution by adding the following blocking clause to F:

$$(\bigvee_{c \in \mathcal{C}} \neg_\mu c^l_1 \vee \neg_\mu c^l_2 \vee \ldots \vee \neg_\mu c^l_{|c^l|}) \vee (\bigvee_{c \in \mathcal{C}} \neg_\mu c^b_1 \vee \neg_\mu c^b_2 \vee \ldots \vee \neg_\mu c^b_{|c^b|})$$

5: Invoke a SAT solver over F. Let μ be the returned model (if any).

To evaluate different SAT solvers within `SimpleBlock`, we have implemented `SimpleBlock` in both `IS23` and `CaDiCaL`, whereas `CryptoMiniSat` already supports it.

AllSAT is the problem of enumerating all the solutions in a CNF formula. In practice, AllSAT tools can stop after finding N solutions, which makes them applicable for solving N-placement. [27] contains an extensive survey of AllSAT approaches; it also presents three state-of-the-art AllSAT tools, called `Toda` *tools (solvers)* herein. The `Toda` tools include one solver per each of the following three families of AllSAT algorithms. The first family of the so-called *blocking solvers* use `SimpleBlock` enhanced (mainly by generalizing each solution by turning as many variables as possible into don't cares, thus shortening the blocking clauses). The second family of *nonblocking solvers* [11] modifies the SAT solver to enumerate the solutions explicitly without using blocking clauses. The third family is based on BDD caching [12] and can be combined with the other two methods. Our empirical evaluation of *N-placement* approaches in Sect. 4.3 includes the `Toda` tools.

3 `IS23`: the New Release of `IntelSAT`

This section introduces the `IS23` release of `IntelSAT`. Sect. 3.1 describes the new parametrized API. Sect. 3.2 is about clause compression.

We would also like to mention a new feature of *out-of-memory recovery*: when the operating system refuses to allocate memory, `IS23` compacts its data structures and retries, rather than immediately returning a failure.

3.1 The API

The users of the solver's C++ library class, denoted herein by `IS23` $\langle \alpha, \beta, \gamma \rangle$, can now parametrize the solver at compile-time with the following template parameters:

1. *clause-index width* α: the width of the C++ variables, used to represent the clause indices.
2. *literal-index width* β: the width of the C++ variables, used to represent the literal indices.
3. *compression flag* γ: a Boolean flag indicating whether to *compress* clauses using bit-arrays.

For the solver to compile, α and β must be powers of 2 and the following assertion must hold: $8 \leqslant \beta \leqslant \alpha \leqslant K$ for a K-bit operating system.

The default version is `IS23` $\equiv$ `IS23` $\langle 32, 32, 0\rangle$. In this paper, we also experiment with `IS23-64` $\equiv$ `IS23` $\langle 64, 32, 0\rangle$ and `IS23-64C` $\equiv$ `IS23` $\langle 64, 32, 1\rangle$, where `IS23-64` and `IS23-64C` can also be accessed from the command-line of the solver's executable (the executable works with the standard DIMACS file format).

The literal-index width β had been 32 bits for every open-source SAT solver so far, hence they can accommodate *at most* $2^{31} - 1$ variables. In fact, it is $2^{31} - 1$ for `CaDiCaL`, but only $2^{28} - 1$ for `Kissat` and `CryptoMiniSat` due to additional bookkeeping. Specifically, `Kissat` borrows bits from the literal-index to be able to inline binary clauses (that is, store them in the WLs only without maintaining a copy in the clause buffer), while efficiently implementing inprocessing [6] as well as failed literal probing and vivification [14]. In `IntelSAT`, the WLs are organized similarly to `Kissat`, but there is currently no need to borrow bits from the literal-index as inprocessing, failed literal probing and vivification are expected to be too heavy for both solving rapid satisfiable incremental queries (the original `IntelSAT` application) and solving gigantic instances (the current `IntelSAT` application).

Notably, `IS23` is the first solver which can be compiled to allow for a practically unlimited number of variables ($2^{63} - 1 = 9{,}223{,}372{,}036{,}854{,}775{,}807$ variables using $\beta = 64$, if no bits are borrowed from the literal-index), where borrowing several bits, if required, is not expected to limit the number of variables in practice. One could also potentially take advantage of `IS23`'s architecture for *saving the memory* when the number of variables is limited by $2^{15} - 1$ (using $\beta = 16$). We leave experiments with different literal-index widths to future work.

3.2 Clause Compression

In this section, we describe how `IS23-64C` manages clauses. For simplicity, we assume herein that the literal-index width β is 32. Similarly to most SAT solvers, `IS23` represents a positive literal v_i by the literal index $li(v_i) = 2 \times i$ and a negative literal $\neg v_i$ by the literal index $li(\neg v_i) = 2 \times i + 1$. As we have mentioned, `IS23` inlines any binary clauses into the WLs [4,7], hence the discussion below concerns long clauses only.

For our purposes, a *bit-array* is a data structure which supports reading and writing of up to 64 bits starting from a specific bit to a dynamically allocated buffer (using several bitwise operations for every access [22]). We have engineered efficient bit-array support in `IS23`.

Recall from Sect. 1 that the literal-width $lw(C)$ represents the minimal number of bits, required to store C's highest literal-index. Our core idea is compressing memory by storing clauses as bit-arrays, where each literal is represented by $lw(C)$ bits, and the width of the clause-size field is also clause-dependent. Consequently, we have implemented a new data structure for storing and accessing clauses, which serves as an alternative for the clause buffer. The vast majority of the solver's code is agnostic to how clauses are managed underneath.

Clearly, to access literals in a clause C, $lw(C)$ must be known. To avoid the overhead of storing $lw(C)$ with every C, we maintain a hash-table of bit-arrays which store clauses, where the bit-array of a given clause C is determined by its 11-bit *hash ID* $hash(C)$, including:

1. 5 bits: the literal-width *lw*,
2. 5 bits: *clause-size-width sw*, that is, the number of bits allocated per clause size, and
3. 1 bit: *learnt-status ls*, that is, whether the clause is learnt or initial.

The last two fields are useful for compactly storing the clause sizes and simplifying the implementation of clause deletion strategies, respectively.

For a clause C, $hash(C)$ is maintained as part of its clause-index $ci(\mathrm{C})$, which, for α=64, leaves more than enough bits (64-11=53) to store the *bit-index*, where the clause starts in its bit-array.

Given C, let $|C|^*$ be *C's compressed size*, which we store instead of C's actual size to save memory (details will follow).

The layout of a clause $C = l_1 \vee l_2 \vee \ldots \vee l_{|C|}$ in a bit-array looks as follows (the width is shown over-brace; `glue`, `stay` and `act` are commonly used fields [1, 18] present only in learnt clauses):

$$\left\langle \overbrace{|C|^*}^{sw(C)}, \underbrace{\overbrace{\texttt{glue}}^{11}, \overbrace{\texttt{stay}}^{1}, \overbrace{\texttt{act}}^{31}}_{\text{learnt clauses only}}; \overbrace{li(l_1)}^{lw(C)}; \overbrace{li(l_2)}^{lw(C)}; \ldots; \overbrace{li(l_{|C|})}^{lw(C)} \right\rangle$$

Given a clause C, how do we determine its clause-size-width $sw(C)$ and its compressed size $|C|^*$? Our guiding principle is to use as few bits as possible. Specifically, we use $sw(C) = 0$ for storing 3-clauses, that is, $|C|^*$ is not stored for them at all. For every $sw(C) > 0$, we reserve the special value $|C|^* = 0$ for clause-deletion heuristic's machinery. Therefore, the clause-size-width $sw(C) = 1$ can accommodate only clauses of size 4, where $|C|^* = 1$ for every such clause. To determine $sw(C)$ for arbitrary clauses, we pre-compute, for clause-size-widths $0 \leqslant w < 32$, the *minimal clause size* $mcs(w)$ stored using w bits to accommodate the special value 0 and as many clauses sizes as possible for every w. The first 10 values and the recursive function for $mcs(w)$ are shown below:

w	0	1	2	3	4	5	6	7	8	9	an arbitrary $n > 2$
$mcs(w)$	3	4	5	8	15	30	61	124	251	506	$mcs(n-1) + 2^{n-1} - 1$

Given a clause C, let w_C be the highest w, such that $|C| \geqslant mcs(w)$. We set $sw(C) = w_C$ and for $|C| > 3 : |C|^* = |C| - mcs(w_C) + 1$.

Let G be the following example formula, where C_1 and C_2 are initial and C_3 is learnt:

$$G = \underbrace{(C_1 = v_1 \vee v_2 \vee \neg v_6) \wedge (C_2 = \neg v_1 \vee v_2 \vee v_6)}_{\text{initial clauses}} \wedge \underbrace{(C_3 = v_1 \vee v_2 \vee v_3 \vee v_4 \vee v_5)}_{\text{learnt clause}}$$

Note that the clauses C_1 and C_2 share the hash ID $\{lw = 4, sw = 0, ls = 0\}$, while C_3 has the hash ID $\{lw = 4, sw = 2, ls = 1\}$. Thus, G would be stored in two bit-arrays as follows (the widths are shown over-brace, while labels appear under-brace):

Bit-array's Hash ID	Clauses
$\underbrace{\overbrace{4}^{5}}_{lw}, \underbrace{\overbrace{0}^{5}}_{sw}, \underbrace{\overbrace{0}^{1}}_{ls}$	$\underbrace{\overbrace{2}^{4}, \overbrace{4}^{4}, \overbrace{13}^{4}}_{C_1}, \underbrace{\overbrace{3}^{4}, \overbrace{4}^{4}, \overbrace{12}^{4}}_{C_2}$
$\underbrace{\overbrace{4}^{5}}_{lw}, \underbrace{\overbrace{2}^{5}}_{sw}, \underbrace{\overbrace{1}^{1}}_{ls}$	$\underbrace{\overbrace{|C_3|^* = 1}^{2}, \overbrace{\texttt{glue}}^{11}, \overbrace{\texttt{stay}}^{1}, \overbrace{\texttt{act}}^{31}, \overbrace{2}^{4}, \overbrace{4}^{4}, \overbrace{6}^{4}, \overbrace{8}^{4}, \overbrace{10}^{4}}_{C_3}$

The 64-bit clause-indices would be as follows:

$ci(C_1) = 2^{61}$	$ci(C_2) = 2^{61} + 12$	$ci(C_3) = 2^{61} + 2^{55} + 2^{53}$
$\underbrace{\overbrace{4}^{5}}_{lw}, \underbrace{\overbrace{0}^{5}}_{sw}, \underbrace{\overbrace{0}^{1}}_{ls}, \underbrace{\overbrace{0}^{53}}_{bit\text{-}index}$	$\underbrace{\overbrace{4}^{5}}_{lw}, \underbrace{\overbrace{0}^{5}}_{sw}, \underbrace{\overbrace{0}^{1}}_{ls}, \underbrace{\overbrace{12}^{53}}_{bit\text{-}index}$	$\underbrace{\overbrace{4}^{5}}_{lw}, \underbrace{\overbrace{2}^{5}}_{sw}, \underbrace{\overbrace{1}^{1}}_{ls}, \underbrace{\overbrace{0}^{53}}_{bit\text{-}index}$

3.2.1 Clause Compression and Variable Succession

Variables succession has an immediate impact on the memory footprint of `IS23-64C`, since it impacts the literal-widths of clauses.

For example, in our latest toy formula G, swapping v_3 and v_6 would reduce the literal-width of both C_1 and C_2 from 4 to 3 without changing the literal-width of C_3, thus saving 1 bit per every literal in C_1 and C_2 and 6 bits overall.

Our core observation is that, if a variable is likely to appear in many clauses, it is crucial for this variable to appear as early as possible in variable succession. In this work, we suggest relying on the expert user knowledge of the problem to determine a good variable succession. We provide two examples below, one of which is backed up by experimental results later in the paper.

Recall the SAT-based `SimpleBlock` N-placement algorithm, where we add many blocking clauses over the same set of important variables. In Sect. 4, we evaluate two versions of `IS23-64C`: `IS23-64CL` has the important variables *first* in the succession, while `IS23-64CH` has them *last*. Unsurprisingly, `IS23-64CL` turns out to be significantly more efficient.

Furthermore, many applications of incremental SAT solving augment clauses with the so-called *selector variables (selectors)* to be able to enable and disable clauses using assumptions. Normally, selectors appear late in variable succession, since they are created after the rest of the instance, thus they are associated with highest possible indices. We expect that, for certain applications, having the selectors early in the succession would have a substantial positive impact on `IS23-64C`'s performance. We leave testing this hypothesis to future work.

Automating the variable succession, that is, having the solver renumber the variables automatically, while still compressing the clauses efficiently, would not be trivial. In principle, the solver could try to figure out a good variable succession out of existing clauses, when a sufficient number of them is provided by the user, and then renumber the variables and compress the clauses. However, that would require to temporarily store a significant amount of clauses *non-compressedly*, which might ruin the compression's efficiency. To alleviate this problem, one might renumber variables and recompress clauses frequently, but that might have a negative impact on the solver's performance. Hence, automating the variable succession is a non-trivial task, which we leave to future work.

4 Experimental Results

We carried out three sets of experiments. We denote by `CrM-32` and `CrM-64` the versions of `CryptoMiniSat` with a 32- and 64-bit clause-index, respectively. We omit the results of the previous version of `IntelSAT`, since the default `IS23` performs at least equally as well, while our goal is introducing the novel `IS23-64` and `IS23-64C` variants of `IS23`.

We dub solver errors and exceptions as follows: `CIErr` or `VIErr` mean that the clause-index space or the variable-index space, respectively, has been exhausted; `TO` or `MO` stand for a time-out or a memory-out, respectively; `Err` stands for other errors (mostly crashes).

The code and the binaries of all the tools and all the benchmarks are publicly available at [20]. Additionally, `IntelSAT`'s repository [19] has been updated to `IS23`.

Table 1 Solving `S`(n). Rows represent instances. The first column contains n. Each pair of subsequent columns shows the time in seconds and memory in GB for the corresponding SAT solver.

n	Kissat		CaDiCaL		MergeSat		CrM-64		IS23		IS23-64		IS23-64C	
	T	M	T	M	T	M	T	M	T	M	T	M	T	M
27	171	21	172	26	5070	35	247	30	227	21	228	21	261	10
28	374	42	347	52	CIErr		505	60	CIErr		459	44	508	19
29	CIErr		702	105	CIErr		1254	125	CIErr		975	90	1023	43
30	CIErr		1523	226	CIErr		2468	249	CIErr		1914	184	2096	81
31	CIErr		3348	453	CIErr		6117	515	CIErr		4262	379	4509	199
32	CIErr		8186	784	CIErr		Err		CIErr		9064	774	9723	365
33	CIErr		Err		CIErr		Err		CIErr		MO		20311	678

Table 2 Finding one placement. The first three columns provide the number of rectangles (in hundreds), variables in CNF (in millions) and clauses in CNF (in millions). Each subsequent pair or triplet of columns corresponds to one solver. Each shows, for the corresponding solver, either: (1) the run-time (in hours), the memory usage (in GB) and, optionally, the number of conflicts (in thousands), or (2) the reason for a failure.

$\frac{R}{10^2}$	$\frac{V}{10^6}$	$\frac{C}{10^6}$	IS23		IS23-64			IS23-64C			CrM-64		Kissat		CaDiCaL	
			T	M	T	M	$\frac{CO}{10^3}$	T	M	$\frac{CO}{10^3}$	T	M	T	M	T	M
20	152	682	1.4	41	1.6	61	19	2.2	60	19	1.7	114	1.6	64	4.3	151
25	238	1066	CIErr		4.2	95	26	4.3	93	21	7.0	180	CIErr		23.5	217
30	342	1535	CIErr		5.6	138	31	8.9	136	31	VIErr		CIErr		27.1	349
35	466	2089	CIErr		14.4	190	54	13.8	186	39	VIErr		CIErr		TO	
40	608	2728	CIErr		13.8	245	46	24.0	245	44	VIErr		Err		Err	
45	770	3453	CIErr		21.8	308	55	25.7	306	55	VIErr		Err		Err	
50	950	4263	CIErr		33.3	382	59	TO			VIErr		Err		Err	

4.1 Gigantic Trivially Satisfiable Instances

To compare solvers' capacity, we created a family of trivially satisfiable instances as follows.

First, consider the following family `U` of trivially unsatisfiable instances: `U`(n) contains 2^n clauses, where every clause contains a literal for every one of the n variables $v_1, \ldots, v_n$, and all the clauses are different (so, every clause falsifies exactly one potential solution). For example, `U`$(2) = (\neg v_1 \vee \neg v_2) \wedge (\neg v_1 \vee v_2) \wedge (v_1 \vee \neg v_2) \wedge (v_1 \vee v_2)$.

The trivially satisfiable family `S` is generated from `U` by adding a new variable v_{n+1} to every clause. For example, `S`$(2) = (\neg v_1 \vee \neg v_2 \vee v_3) \wedge (\neg v_1 \vee v_2 \vee v_3) \wedge (v_1 \vee \neg v_2 \vee v_3) \wedge (v_1 \vee v_2 \vee v_3)$.

For the experiments in this and the next subsection (Sect 4.2), we used a machine having 790Mb of memory and an Intel® Xeon® processor of 2.70Ghz CPU frequency. Table 1 compares `Kissat`, `CaDiCaL`, `MergeSat`, `CrM-64`, `IS23`, `IS23-64` and `IS23-64C` on `S` instances without any time or memory limits (`CrM-32` failed with `CIErr` already on `S`(25)).

`IS23-64C` is the only solver, which can solve the gigantic instance `S`(33) having 2^{33} = 8,589,934,592 clauses and $2^{33} \times (33 + 1)$ = 292,057,776,128 literals overall.

Note that `IS23-64C` consumes around half the memory of `IS23-64`. Why is the gap so low, given that, for e.g. $n = 32$, each literal takes $32/6 = 5.3$ times fewer bits in the clause buffer (so, seemingly, one could expect `IS23-64C` to use 5 times rather than 2 times less memory than `IS23-64`)? Shortly, because of the Watch Lists. WLs occupy the same amount of memory for both `IS23-64` and `IS23-64C`, but, for `IS23-64C`, they dominate the memory consumption using over 60% of the memory. Thus, compressing the WLs is a promising direction for future work.

4.2 Finding One Placement

We generated publicly available placement instances in CNF as follows. Each instance in the family `P`(R) corresponds to the problem of placing R rectangles of randomly chosen width and height in the range $[1 - 10]$ on a $10^3 \times 10^3$ grid. The results on these instances roughly correspond to results on industrial instances of similar size, which we, unfortunately, cannot share due to IP restrictions.

For finding one placement, we ran `Kissat`, `CaDiCaL`, `MergeSat`, `CrM-32` `CrM-64`, `IS23`, `IS23-64` and `IS23-64C` with the timeout of 48 hours and the memory limit of 512Gb on instances in $[\mathtt{P}(2000), \mathtt{P}(2500), \ldots, \mathtt{P}(5000)]$ (all the solvers failed on $\mathtt{P}(5500)$). The results are shown in Table 2 (`MergeSat` and `CrM-32` are omitted as they solved none of the instances).

Our new release of `IntelSAT`, `IS23`, is clearly the most scalable solver as `IS23-64` solved even the instance $\mathtt{P}(5000)$ having almost 1 billion variables and 4.3 billion clauses, whereas the next best solver `CaDiCaL` managed to solve only the $\mathtt{P}(3000)$ instance, while being $4.8X$ slower and using $2.5X$ more memory than `IS23-64` for $\mathtt{P}(3000)$.

Compare `IS23-64` with `IS23-64C`. Usually, `IS23-64` outperformed `IS23-64C` in terms of run-time. `IS23-64C` never generated more conflicts than `IS23-64`, but was almost always slower, apparently because of the overhead of the bit-wise operations. Surprisingly, `IS23-64C` was only slightly more efficient than `IS23` in terms of memory consumption. Further analysis showed that `IS23-64C` did compress the clause buffer (e.g., by $1.5X$ for $\mathtt{P}(4500)$), but other data structures (WLs and variable/literal-indexed arrays) dominated the memory usage.

4.3 Finding Many Placements

In our last experiment, we evaluated the different solvers for finding $N = 1{,}000{,}000$ placements. Since finding 10^6 placements is substantially more difficult than only one, we used smaller instances in $[\mathtt{P}(200), \mathtt{P}(300), \mathtt{P}(400), \ldots]$. However, we decided to also limit the resources: we used machines with 32Gb of memory only running Intel® Xeon® processors of 3Ghz CPU frequency and set the timeout to 10 hours.

We ran `CaDiCaL`, `CrM-64`, `IS23`, `IS23-64` and the two versions of `IS23-64C`, `IS23-64CL` and `IS23-64CH` (recall Sect. 3.2.1), within the `SimpleBlock` algorithm. We also launched the `Toda` tools (recall Sect. 2): `bc_minisat_all`, `nbc_minisat_all` and `bdd_minisat_all`. The last instance for which at least one solver succeeded to find 10^6 placements was $\mathtt{P}(1000)$.

The results are shown in Table 3. Observe that only `IS23-64CL` was able to find 10^6 placements for all the instances. Unlike for finding one placement, `IS23-64` consumed significantly more memory than `IS23-64CL`, since the long blocking clauses dominated the memory consumption (the size of every blocking clause for $\mathtt{P}(R)$ is the number of important variables $= 20 \times R$). Observe that the various `IS23` versions managed to squeeze the memory usage into 31Gb for several instances of different complexity due to the out-of-memory recovery feature (recall Sect. 3).

`IS23-64CH` failed on two instances providing evidence that variable succession scheme is crucial. In addition to `IS23-64CL` and `IS23-64CH`, we have also tested `IS23-64CD`: the variable succession scheme, generated by default by our in-house eager SMT solver. `IS23-64CD` was able to solve $\mathtt{P}(900)$, but not $\mathtt{P}(1000)$, hence we upgraded our default to `IS23-64CL`.

Notably, `IntelSAT` scaled substantially better than both `CaDiCaL` and `CrM-64` within `SimpleBlock`. The explanation may be related to the *Incremental Lazy Backtracking (ILB)* principle, implemented already in the original `IntelSAT` [18]. Specifically, before every incremental SAT query, `CaDiCaL` and `CrM-64` backtrack to the global decision level after each model is found, while `IntelSAT` backtracks to the highest possible decision level, where the latest blocking clause halts to be falsified. Note that implementing or disabling ILB in any of the solvers would have no impact on the experiments reported in Table 1 and Table 2, since the benchmarks used in these experiments are not incremental.

Finally, our `IS23-64CL`-based *N-placement* tool scaled much better than the state-of-the-art AllSAT solvers (`Toda` tools), despite us using only the basic `SimpleBlock` algorithm, which can be substantially improved by techniques, inspired by blocking AllSAT solvers.

Table 3 Finding 10^6 placements. The first column in both the sub-tables shows the number of rectangles (in hundreds); the upper table also contains two columns with the number of variables and clauses in CNF (in millions). Each subsequent triplet of columns shows, for one solver: the number of solutions (in thousands), the run-time (in hours) and the memory usage (in GB); in case of a failure, the last two columns per solver show its reason instead.

$\frac{R}{10^2}$	$\frac{V}{10^6}$	$\frac{C}{10^6}$	IS23			IS23-64			IS23-64CL			IS23-64CH		
			$\frac{S}{10^3}$	T	M	$\frac{S}{10^3}$	T	M	$\frac{S}{10^3}$	T	M	$\frac{S}{10^3}$	T	M
3	3	15	1000	0.4	15	1000	0.5	16	1000	0.6	10	1000	0.5	14
4	6	27	829	CIErr		1000	0.7	22	1000	0.9	12	1000	0.8	20
5	10	43	644	CIErr		1000	0.9	28	1000	1.3	17	1000	1.2	29
6	14	61	513	CIErr		1000	1.3	31	1000	1.8	23	1000	1.4	31
7	19	84	438	CIErr		1000	1.6	31	1000	2.2	25	1000	1.8	31
8	24	109	372	CIErr		668	MO		1000	2.7	31	1000	2.4	31
9	31	138	338	CIErr		482	MO		1000	3.2	31	769	MO	
10	38	171	265	CIErr		449	MO		1000	3.6	31	682	MO	

$\frac{R}{10^2}$	CrM-64			CaDiCaL			bc_minisat_all			nbc_minisat_all			bdd_minisat_all		
	$\frac{S}{10^3}$	T	M	$\frac{S}{10^3}$	T	M	$\frac{S}{10^3}$	T	M	$\frac{S}{10^3}$	T	M	$\frac{S}{10^3}$	T	M
3	30	TO		15	TO		19	TO		1000	0.1	1	0	Err	
4	15	TO		8	TO		8	TO		1000	3.4	3	0	Err	
5	10	TO		5	TO		0	TO		0	TO		0	TO	
6	6	TO		3	TO		0	TO		0	TO		0	TO	
7	4	TO		2	TO		0	TO		0	TO		0	TO	
8	3	TO		2	TO		0	TO		0	TO		0	TO	
9	0	Err		1	TO		0	TO		0	TO		0	TO	
10	0	Err		1	TO		0	TO		0	TO		0	TO	

5 Conclusion

We introduced the `IS23` release of our SAT solver `IntelSAT`, targeted to solve huge instances beyond the capacity of other solvers. `IS23` can compress the memory by storing clauses in bit-arrays. We showed that only `IS23` can solve a gigantic trivially satisfiable instance with over 8.5 billion clauses. `IS23` also enabled solving huge instances of the industrial placement problem with up to 4.3 billion clauses. Additionally, `IS23` turned out to be substantially more efficient than other solvers for finding 10^6 placements on instances with up to 170 million clauses, where a carefully chosen variable succession scheme enabled the best results.

References

1 Gilles Audemard and Laurent Simon. On the glucose SAT solver. *Int. J. Artif. Intell. Tools*, 27(1):1840001:1–1840001:25, 2018. `doi:10.1142/S0218213018400018`.

2 Tomas Balyo, Marijn J.H. Heule, Markus Iser, Matti Järvisalo, and Martin Suda, editors. *Proceedings of SAT Competition 2022: Solver and Benchmark Descriptions*. Department of Computer Science Series of Publications B. Department of Computer Science, University of Helsinki, Finland, 2022.

3 Clark Barrett, Pascal Fontaine, and Cesare Tinelli. The SMT-LIB Standard: Version 2.6. Technical report, Department of Computer Science, The University of Iowa, 2017. Available at `http://smt-lib.org/`.

4 Armin Biere, Katalin Fazekas, Mathias Fleury, and Maximillian Heisinger. CaDiCaL, Kissat, Paracooba, Plingeling and Treengeling entering the SAT Competition 2020. In Tomas Balyo, Nils Froleyks, Marijn Heule, Markus Iser, Matti Järvisalo, and Martin Suda, editors, *Proc. of SAT Competition 2020 – Solver and Benchmark Descriptions*, volume B-2020-1 of *Department of Computer Science Report Series B*, pages 51–53. University of Helsinki, 2020.

5 Armin Biere, Marijn Heule, Hans van Maaren, and Toby Walsh, editors. *Handbook of Satisfiability*, volume 336 of *Frontiers in Artificial Intelligence and Applications*. IOS Press, 2021.

6 Armin Biere, Matti Järvisalo, and Benjamin Kiesl. Preprocessing in sat solving. In Armin Biere, Marijn Heule, Hans van Maaren, and Toby Walsh, editors, *Handbook of Satisfiability*, volume 336 of *Frontiers in Artificial Intelligence and Applications*, pages 391–435. IOS Press, 2021. `doi:10.3233/FAIA200992`.

7 Geoffrey Chu, Aaron Harwood, and Peter J. Stuckey. Cache conscious data structures for boolean satisfiability solvers. *J. Satisf. Boolean Model. Comput.*, 6(1-3):99–120, 2009. doi:10.3233/sat190064.

8 Aviad Cohen, Alexander Nadel, and Vadim Ryvchin. Local search with a SAT oracle for combinatorial optimization. In Jan Friso Groote and Kim Guldstrand Larsen, editors, *Tools and Algorithms for the Construction and Analysis of Systems – 27th International Conference, TACAS 2021, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2021, Luxembourg City, Luxembourg, March 27 – April 1, 2021, Proceedings, Part II*, volume 12652 of *Lecture Notes in Computer Science*, pages 87–104. Springer, 2021. doi:10.1007/978-3-030-72013-1_5.

9 Niklas Eén and Niklas Sörensson. An extensible sat-solver. In Enrico Giunchiglia and Armando Tacchella, editors, *Theory and Applications of Satisfiability Testing, 6th International Conference, SAT 2003. Santa Margherita Ligure, Italy, May 5-8, 2003 Selected Revised Papers*, volume 2919 of *Lecture Notes in Computer Science*, pages 502–518. Springer, 2003. doi:10.1007/978-3-540-24605-3_37.

10 Vijay Ganesh and David L. Dill. A decision procedure for bit-vectors and arrays. In Werner Damm and Holger Hermanns, editors, *Computer Aided Verification, 19th International Conference, CAV 2007, Berlin, Germany, July 3-7, 2007, Proceedings*, volume 4590 of *Lecture Notes in Computer Science*, pages 519–531. Springer, 2007. doi:10.1007/978-3-540-73368-3_52.

11 Orna Grumberg, Assaf Schuster, and Avi Yadgar. Memory efficient all-solutions SAT solver and its application for reachability analysis. In Alan J. Hu and Andrew K. Martin, editors, *Formal Methods in Computer-Aided Design, 5th International Conference, FMCAD 2004, Austin, Texas, USA, November 15-17, 2004, Proceedings*, volume 3312 of *Lecture Notes in Computer Science*, pages 275–289. Springer, 2004. doi:10.1007/978-3-540-30494-4_20.

12 Jinbo Huang and Adnan Darwiche. The language of search. *J. Artif. Intell. Res.*, 29:191–219, 2007. doi:10.1613/jair.2097.

13 Richard Korf, Michael Moffitt, and Martha Pollack. Optimal rectangle packing. *Annals OR*, 179:261–295, September 2010. doi:10.1007/s10479-008-0463-6.

14 Chu-Min Li, Fan Xiao, Mao Luo, Felip Manyà, Zhipeng Lü, and Yu Li. Clause vivification by unit propagation in CDCL SAT solvers. *Artif. Intell.*, 279, 2020. doi:10.1016/j.artint.2019.103197.

15 Norbert Manthey. The mergesat solver. In Chu-Min Li and Felip Manyà, editors, *Theory and Applications of Satisfiability Testing – SAT 2021 – 24th International Conference, Barcelona, Spain, July 5-9, 2021, Proceedings*, volume 12831 of *Lecture Notes in Computer Science*, pages 387–398. Springer, 2021. doi:10.1007/978-3-030-80223-3_27.

16 Kenneth L. McMillan. Applying SAT methods in unbounded symbolic model checking. In Ed Brinksma and Kim Guldstrand Larsen, editors, *Computer Aided Verification, 14th International Conference, CAV 2002,Copenhagen, Denmark, July 27-31, 2002, Proceedings*, volume 2404 of *Lecture Notes in Computer Science*, pages 250–264. Springer, 2002. doi:10.1007/3-540-45657-0_19.

17 Matthew W. Moskewicz, Conor F. Madigan, Ying Zhao, Lintao Zhang, and Sharad Malik. Chaff: Engineering an efficient SAT solver. In *Proceedings of the 38th Design Automation Conference, DAC 2001, Las Vegas, NV, USA, June 18-22, 2001*, pages 530–535. ACM, 2001. doi:10.1145/378239.379017.

18 Alexander Nadel. Introducing intel(r) SAT solver. In Kuldeep S. Meel and Ofer Strichman, editors, *25th International Conference on Theory and Applications of Satisfiability Testing, SAT 2022, August 2-5, 2022, Haifa, Israel*, volume 236 of *LIPIcs*, pages 8:1–8:23. Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2022. doi:10.4230/LIPIcs.SAT.2022.8.

19 Alexander Nadel. Intel® SAT Solver. https://github.com/alexander-nadel/intel_sat_solver, 2022–2023.

20 Alexander Nadel. Solving huge instances with intel® sat solver: Supplementary material. https://technionmail-my.sharepoint.com/:f:/g/personal/alexandernad_technion_ac_il/EtihJbiS1XBJoi0DRTGbCnIBlfs1__hkPRiZ3uTpIR_x_g, 2023.

21 Alexander Nadel and Vadim Ryvchin. Efficient SAT solving under assumptions. In Alessandro Cimatti and Roberto Sebastiani, editors, *Theory and Applications of Satisfiability Testing – SAT 2012 – 15th International Conference, Trento, Italy, June 17-20, 2012. Proceedings*, volume 7317 of *Lecture Notes in Computer Science*, pages 242–255. Springer, 2012. doi:10.1007/978-3-642-31612-8_19.

22 Collin Peterson. How to define and work with an array of bits in C?, November 2020. URL: https://stackoverflow.com/a/30590727.

23 Naveed A. Sherwani. *Algorithms for VLSI physical design automation.* Kluwer, 3 edition, November 1998.

24 Mate Soos. Allow memory to grow larger than 4gb per thread. https://github.com/msoos/cryptominisat/issues/389, May 2017.

25 Mate Soos, Karsten Nohl, and Claude Castelluccia. Extending SAT solvers to cryptographic problems. In Oliver Kullmann, editor, *Theory and Applications of Satisfiability Testing – SAT 2009, 12th International Conference, SAT 2009, Swansea, UK, June 30 – July 3, 2009. Proceedings*, volume 5584 of *Lecture Notes in Computer Science*, pages 244–257. Springer, 2009. doi:10.1007/978-3-642-02777-2_24.

26 Marc Thurley. sharpSAT – Counting models with advanced component caching and implicit BCP. In Armin Biere and Carla P. Gomes, editors, *Theory and Applications of Satisfiability Testing – SAT 2006, 9th International Conference, Seattle, WA, USA, August 12-15, 2006, Proceedings*, volume 4121 of *Lecture Notes in Computer Science*, pages 424–429. Springer, 2006. doi:10.1007/11814948_38.

27 Takahisa Toda. Implementing efficient all solutions SAT solvers. *J. Exp. Algorithmics*, 21:1, 2016. doi:10.1145/2975585.