Optimizing SAT Encodings for Arithmetic Constraints

Neng-Fa Zhou[1] and Håkan Kjellerstrand[2]

[1] CUNY Brooklyn College & Graduate Center
[2] hakank.org

Abstract. The log encoding has been perceived to be unsuited to arithmetic constraints due to its hindrance to propagation. The surprising performance of PicatSAT, which is a pure eager SAT compiler based on the log encoding, in the MiniZinc Challenge 2016 has revived interest in the log encoding. This paper details the optimizations used in PicatSAT for encoding arithmetic constraints. PicatSAT adopts some well-known optimizations from CP systems, language compilers, and hardware design systems for encoding constraints into compact and efficient SAT code. PicatSAT is also empowered by a novel optimization, called equivalence reasoning, for arithmetic constraints, which leads to reduction of code size and execution time. In a nutshell, this paper demonstrates that the optimized log encoding is competitive for encoding arithmetic constraints.

1 Introduction

The drastic enhancement of SAT solvers' performance has made SAT a viable backbone for general CSP (Constraint Satisfaction Problem) solvers [8, 21, 30, 33, 34]. Many real-world combinatorial problems involve arithmetic constraints, and it remains a challenge to efficiently encode arithmetic constraints into SAT. The *sparse* encoding [20, 36] and *order* encoding [13, 25, 34] can easily blow up the code size, and the *log* encoding [22, 18] is perceived to be a poor choice, despite its compactness, due to its failure to maintain arc consistency, even for binary constraints. This dilemma of the *eager* approach has led to the emergence of the *lazy* approach, as represented by SMT solvers that use integer arithmetic as a theory [6, 14, 26] and the *lazy clause generation* (LCG) solver that combines SAT and constraint propagation [17, 29]. Both the eager and lazy approaches have strengths and weaknesses [27]. For problems that require frequent checking of arithmetic constraints the lazy approach may not be competitive due to the overhead, even when checking is done incrementally and in a priori manner. From an engineering perspective, the eager approach also has its merit, just like the separation of computer hardware and language compilers is beneficial.

The surprising performance of the PicatSAT compiler in the MiniZinc Challenge 2016 is thought-provoking.[3] PicatSAT is a pure SAT compiler that trans-

[3] PicatSAT with the Lingeling SAT solver won two silver medals and one bronze medal in the competition (http://www.minizinc.org/challenge2016/results2016.html)

lates CSPs into log-encoded SAT code. PicatSAT adopts the sign-and-magnitude log encoding for domain variables. For a domain with the maximum absolute value n, it uses $log_2(n)$ Boolean variables to encode the domain. If the domain contains both negative and positive values, then another Boolean variable is employed to encode the sign. Each combination of values of the Boolean variables represents a valuation for the domain variable. The addition constraint is encoded as logic adders, and the multiplication constraint is encoded as logic adders using the *shift-and-add* algorithm.

PicatSAT adopts some well-known optimizations from CP systems, language compilers, and hardware design systems for encoding constraints into compact and efficient SAT code: it preprocesses constraints before compilation in order to remove no-good values from the domains of variables whenever possible; it eliminates common subexpressions so that no primitive constraint is duplicated; it uses a logic optimizer to generate optimized code for adders. These optimizations significantly improve the quality of the generated code.

This paper proposes a new optimization, named *equivalence reasoning*, for log-encoded arithmetic constraints. Equivalence reasoning identifies information about if a Boolean variable is 0 or 1, if a Boolean variable is equivalent to another Boolean variable, or if a Boolean variable is the negation of another Boolean variable. This optimization can reduce both the number of Boolean variables and the number of clauses in the CNF code.

The experimental results show that equivalence reasoning reduces code sizes, and for some benchmarks, significantly reduces the solving time. The MiniZinc Challenge 2016 results show that PicatSAT outperformed some of the fastest CP solvers in the competition. Our new comparisons of PicatSAT with fzn2smt, an SMT-based CSP solver, and Chuffed, a cutting-edge LCG solver, also reveal the competitiveness of PicatSAT.

2 The PicatSAT Compiler

PicatSAT is offered in Picat as a module named `sat`. In addition to the `sat` module, Picat offers two other solver modules, named `cp` and `mip`, respectively. All these three modules implement the same set of basic linear constraints over integer domains and Boolean constraints. The `cp` and `sat` modules also implement non-linear and global constraints, and the `mip` module also supports real-domain variables. The common interface that Picat provides for the solver modules allows seamless switching from one solver to another.

In order to give the reader a complete picture of the PicatSAT compiler, we give in this section an overview of the compiler, including the adopted optimizations. A description of a preliminary version of PicatSAT with no optimizations is given in [38].

2.1 Preprocessing and Decomposition

In general, a constraint model consists of a set of *decision variables*, each of which has a specified domain, and a set of *constraints*, each of which restricts the

possible combinations of values of the involved decision variables. A constraint program normally poses a problem in three steps: (1) generate variables; (2) generate constraints over the variables; and (3) call `solve` to invoke the solver in order to find a valuation for the variables that satisfies the constraints and possibly optimizes an objective function.

PicatSAT preprocesses the accumulated constraints when the `solve` predicate is called. For binary equality constraints, PicatSAT excludes no-good values from the domains to achieve arc consistency, unless the domains are too big.[4] For instance, for the constraint $X = 9 * B + 1$, where B is a Boolean variable that has the domain 0..1, PicatSAT narrows X's domain to $\{1, 10\}$. For other types of constraints, including binary equality constraints that involve large-domain variables, PicatSAT narrows the bounds of domains to archieve interval consistency.

PicatSAT decomposes arithmetic constraints into basic constraints, including *primitive*, *reified*, and *implication* constraints. A primitive constraint is one of the following: $\Sigma_i^n B_i\ r\ c$ (r is $=$, $\geq$, or $\leq$, and c is 1 or 2),[5] $X\ r\ Y$ (r is $=$, $\neq$, $>$, $\geq$, $<$, or $\leq$), $X + Y = Z$, and $X \times Y = Z$, where B_i is a Boolean variable, and X, Y, and Z are integers or integer domain variables. A reified constraint, after decomposition, takes the form $B \Leftrightarrow C$, and an implication constraint has the form $B \Rightarrow C$, where B is a Boolean variable, and C is a primitive constraint.

For a linear constraint, PicatSAT sorts the terms by the variables. This ordering facilitates merging terms of the same variables, but is hardly optimal for generating efficient SAT code. PicatSAT breaks down Pseudo-Boolean (PB) constraints, including cardinality constraints, in the same way as other linear constraints, unless they are cardinality constraints of the form $\Sigma_i^n B_i\ r\ c$ (c =1 or 2). For example, the cardinality constraint $U + V + W + X \leq 3$, where all the variables are Boolean, is split into the following primitive constraints:

$$
\begin{aligned}
&U + V = T_1\\
&W + X = T_2\\
&T_1 + T_2 = T_3\\
&T_3 \leq 3
\end{aligned}
$$

This method of decomposing PB constraints is simple, and generates compact code. For a cardinality constraint that has n variables, this method introduces $O(n)$ auxiliary integer-domain variables, which require a total number of $O(n \times log_2(n))$ Boolean variables to encode.[6]

During decomposition, PicatSAT introduces auxiliary variables to combine terms in the same way as language compilers break expressions into triplets.

[4] The default bounds of small domains are `-3200` and `3200`, which can be reset by using the built-in `fd_vector_min_max(LB,UB)`.

[5] The cardinality constraint is treated as a normal linear constraint when $c > 2$.

[6] This information is disclosed here in order to give the reader a complete picture of PicatSAT. A study is underway to investigate how this adder-based encoding compares with other encodings, such as sorting networks [16], totalizers [5], BDDs [7], and the decomposition method for adders by [37].

PicatSAT makes efforts not to create variables with both positive and negative values in their domains, if possible, because such a domain requires a sign bit and cannot be encoded compactly. For example, for the constraint $U-V+W-X \leq 3$, PicatSAT merges U with W, and V with X so that no auxiliary variables have negative values in their domains if the domains of the original variables do not contain negative values.

PicatSAT eliminates common subexpressions in constraints. Whenever PicatSAT introduces an auxiliary variable for a primitive constraint, it tables the constraint. When the same primitive constraint is encountered again, PicatSAT reuses the auxiliary variable, rather than introducing a new variable for the constraint. For example, when a reification constraint $B \Leftrightarrow C$ is generated, PicatSAT tables it and reuses the variable B, rather than introducing a new variable for the primitive constraint C, when C is encountered again in another constraint. The algorithm used in PicatSAT for identifying common subexpressions is not as sophisticated as those used in [4, 28]. It incurs little overhead on compilation, but fails to eliminate common subexpressions in many cases.

2.2 The Sign-and-Magnitude Log Encoding

PicatSAT employs the *log-encoding* for domain variables. For a domain variable, $\lceil log_2(n) \rceil$ Boolean variables are used, where n is the maximum absolute value in the domain. If the domain contains both negative and positive values, then another Boolean variable is employed to represent the sign. In this paper, for a log-encoded domain variable X, X`.s` denotes the sign, X`.m` denotes the magnitude, which is a vector of Boolean variables $<X_{n-1}X_{n-2}\ldots X_1X_0>$.

This *sign-and-magnitude* encoding requires a clause to disallow negative zero if the domain contains values of both signs. Each combination of values of the Boolean variables represents a valuation for the domain variable: $X_{n-1} \times 2^{n-1} + X_{n-2} \times 2^{n-2} + \ldots + X_1 \times 2 + X_0$. If there are holes in the domain, then not-equal constraints are generated to disallow assigning those hole values to the variable. Also, inequality constraints ($\geq$ and $\leq$) are generated to prohibit assigning out-of-bounds values to the variable if either bound is not $2^k - 1$ for some k.

For small-domain variables, PicatSAT calls the logic optimizer, Espresso [9], to generate an optimal or near-optimal CNF formula. For example, for the domain constraint $X :: [-2, -1, 2, 1]$, one Boolean variable, S, is utilized to encode the sign, and two variables, X_1 and X_0, are employed to encode the magnitude. A naive encoding with *conflict clauses* [18] for the domain requires four clauses:

$\neg S \vee \neg X_1 \vee \neg X_0$
$\neg S \vee X_1 \vee X_0$
$S \vee X_1 \vee X_0$
$S \vee \neg X_1 \vee \neg X_0$

These clauses correspond to four no-good values: -3, -0, 0, and 3, where -0 denotes the negative 0. Espresso only returns two clauses for the domain:

$\texttt{X}_0 \vee \texttt{X}_1$
$\neg\texttt{X}_0 \vee \neg\texttt{X}_1$

Note that the sign variable is optimized away.

2.3 Encoding Basic Constraints

The encodings for the addition and multiplication constraints will be described in later sections. This subsection briefly describes the Booleanization of other basic constraints.

The *at-least-one* constraint $\Sigma_i^n B_i \geq 1$ is encoded into one CNF clause:

$$B_1 \vee B_2 \vee \ldots \vee B_n$$

The *at-least-two* constraint $\Sigma_i^n B_i \geq 2$ is converted into n at-least-one constraints: for each $n-1$ variables, the sum of the variables is at least one. The *at-most-one* constraint $\Sigma_i B_i \leq 1$ is encoded into CNF by using the two-product algorithm [12]. The *at-most-two* constraint is converted into n at-most-one constraints. The *exactly-one* constraint $\Sigma_i^n B_i = 1$ is converted into a conjunction of an at-least-one constraint and an at-most-one constraint. The *exactly-two* constraint is compiled similarly.

A recursive algorithm is utilized to compile binary primitive constraints. For example, consider $X \geq Y$. This constraint is translated to the following:

$$\begin{array}{l} X.\mathrm{s} = 0 \wedge Y.\mathrm{s} = 1 \vee \\ X.\mathrm{s} = 1 \wedge Y.\mathrm{s} = 1 \Rightarrow X.\mathrm{m} \leq Y.\mathrm{m} \vee \\ X.\mathrm{s} = 0 \wedge Y.\mathrm{s} = 0 \Rightarrow X.\mathrm{m} \geq Y.\mathrm{m} \end{array}$$

PicatSAT simplifies the formula if the variables' signs are known at compile time. Let $X.\mathrm{m} = <X_{n-1}X_{n-2}\ldots X_1X_0>$, $Y.\mathrm{m} = <Y_{n-1}Y_{n-2}\ldots Y_1Y_0>$.[7] PicatSAT introduces auxiliary variables T_0, T_1, ..., T_{n-1} for comparing the bits:

$$\begin{array}{l} T_0 \Leftrightarrow (X_0 \geq Y_0) \\ T_1 \Leftrightarrow (X_1 > Y_1) \vee (X_1 = Y_1 \wedge T_0) \\ \qquad \vdots \\ T_{n-1} \Leftrightarrow (X_{n-1} > Y_{n-1}) \vee (X_{n-1} = Y_{n-1} \wedge T_{n-2}) \end{array}$$

PicatSAT then encodes the constraint $X.\mathrm{m} \geq Y.\mathrm{m}$ as T_{n-1}. When either X or Y is a constant, PicatSAT compiles the constraint without introducing any auxiliary variables.[8]

The reified constraint $B \Leftrightarrow C$ is equivalent to $B \Rightarrow C$ and $\neg B \Rightarrow \neg C$, where $\neg C$ is the negation of C. Let $C_1 \wedge \ldots \wedge C_n$ be the CNF formula of C after Booleanization. Then $B \Rightarrow C$ is encoded into $C'_1 \wedge \ldots \wedge C'_n$, where $C'_i = (C_i \vee \neg B)$ for $i = 1, ..., n$.

[7] The two bit strings are made to have the same length after *padding with zeros*.

[8] This is done by using a recursive algorithm. When X or Y is a constant, the number of clauses in the generated code is still $O(n)$ even though no auxiliary variables are used.

3 Equivalence Reasoning

Equivalence reasoning is an optimization that reasons about a possible value for a Boolean variable or the relationship between two Boolean variables. This reasoning exploits the properties of constraints. For example, consider the domain constraint X :: [2,6]. The magnitude of X is encoded with three Boolean variables X`.m = <`X_2`,`X_1`,`X_0`>`. PicatSAT infers $X_1 = 1$ and $X_0 = 0$ from the fact that the binary representations of both 2 and 6 end with 10. With this reasoning, PicatSAT does not generate a single clause for this domain.

The following gives several constraints on which PicatSAT performs equivalence reasoning:

$$\begin{array}{ll} X = \text{abs}(Y) & \Rightarrow X.\text{m} = Y.\text{m},\ X.\text{s} = 0 \\ X = \text{-}Y & \Rightarrow X.\text{m} = Y.\text{m},\ X.\text{s} = Y.\text{s} = 0 \rightarrow X.\text{m} = 0 \\ X = Y \bmod 2^K & \Rightarrow X_0 = Y_0,\ X_1 = Y_1,\ \ldots,\ X_{k-1} = Y_{k-1} \\ X = Y \text{ div } 2^K & \Rightarrow X_0 = Y_K,\ X_1 = Y_{K+1},\ \ldots \end{array}$$

Note that the constraint $X.\text{m} = Y.\text{m}$ is enforced by unifying the corresponding Boolean variables of X and Y at compile time. Equivalence reasoning considerably eases encoding for some of the constraints. For example, the following clause encodes the constraint $X = \text{-}Y$, regardless of the sizes of the domains:[9]

$$\neg X.\text{s} \vee \neg Y.\text{s}$$

and no clause is needed to encode the constraint $X = \text{abs}(Y)$.

Equivalence reasoning can be applied to those addition and multiplication constraints that involve constants. We call this kind of equivalence reasoning *constant propagation*. The remaining of this section gives the propagation rules. In order to make the description self-contained, we include the encoding algorithms for addition and multiplication constraints described in [38].

3.1 Constant Propagation for $X + Y = Z$

For the constraint $X + Y = Z$, if either X or Y has values of mixed signs in its domain, then PicatSAT generates implication constraints to handle different sign combinations [38]. In the following we assume that both operands X and Y are non-negative (i.e., X`.s = 0` and Y`.s = 0`), so the constraint $X + Y = Z$ can be rewritten into the unsigned addition X`.m+`Y`.m =` Z`.m`.

Let X`.m` $= X_{n-1} \ldots X_1 X_0$, Y`.m` $= Y_{n-1} \ldots Y_1 Y_0$, and Z`.m` $= Z_n \ldots Z_1 Z_0$. The unsigned addition can be Booleanized by using logic adders as follows:

$$\begin{array}{rcccc} & X_{n-1} & \ldots & X_1 & X_0 \\ + & Y_{n-1} & \ldots & Y_1 & Y_0 \\ \hline Z_n & Z_{n-1} & \ldots & Z_1 & Z_0 \end{array}$$

[9] Recall that since no negative zeros are allowed, the domain constraints already guarantee that $X.\text{m} = 0 \Rightarrow \neg X.\text{s}$ and $X.\text{m} = 0 \Rightarrow \neg Y.\text{s}$.

A half-adder is employed for $X_0+Y_0 = C_1Z_0$, where C_1 is the carry-out. For each other position i ($0 < i \leq n-1$), a full adder is employed for $X_i+Y_i+C_i = C_{i+1}Z_i$. The top-most bit of Z, Z_n, is equal to C_n. This encoding corresponds to the *ripple-carry adder* used in computer architectures.

The full adder $X_i + Y_i + C_{in} = C_{out}Z_i$ is encoded with the following 10 CNF clauses when all the operands are variables:

$$
\begin{array}{l}
X_i \vee \neg Y_i \vee C_{in} \vee Z_i \\
X_i \vee Y_i \vee \neg C_{in} \vee Z_i \\
\neg X_i \vee \neg Y_i \vee C_{in} \vee \neg Z_i \\
\neg X_i \vee Y_i \vee \neg C_{in} \vee \neg Z_i \\
\neg X_i \vee C_{out} \vee Z_i \\
X_i \vee \neg C_{out} \vee \neg Z_i \\
\neg Y_i \vee \neg C_{in} \vee C_{out} \\
Y_i \vee C_{in} \vee \neg C_{out} \\
\neg X_i \vee \neg Y_i \vee \neg C_{in} \vee Z_i \\
X_i \vee Y_i \vee C_{in} \vee \neg Z_i
\end{array}
$$

If any of the operands is a constant, then the code can be simplified. For example, if C_{in} is 0, then the full adder becomes a half adder, which can be encoded with 7 CNF clauses.

For the half adder $X_i + Y_i = C_{out}Z_i$, if any of the operands is a constant, PicatSAT infers that the other two variables are equal or one variable is the negation of the other. PicatSAT performs the following inferences when X or Z is a constant:

Rule-1 : $X_i = 0 \Rightarrow C_{out} = 0 \wedge Z_i = Y_i$.
Rule-2 : $X_i = 1 \Rightarrow C_{out} = Y_i \wedge Z_i = \neg Y_i$.
Rule-3 : $Z_i = 0 \Rightarrow C_{out} = X_i \wedge X_i = Y_i$
Rule-4 : $Z_i = 1 \Rightarrow C_{out} = 0 \wedge X_i = \neg Y_i$.

Similar inference rules apply when Y_i is a constant.

For example, consider the addition:

$$
\begin{array}{cccc}
 & X_2 & X_1 & X_0 \\
+ & 1 & 0 & 0 \\
\hline
Z_3 & Z_2 & Z_1 & Z_0
\end{array}
$$

PicatSAT infers the following equivalences:

$$
\begin{array}{r}
X_0 = Z_0 \\
X_1 = Z_1 \\
\neg X_2 = Z_2 \\
X_2 = Z_3
\end{array}
$$

Consider, as another example, the addition:

$$\begin{array}{cccc} & X_2 & X_1 & X_0 \\ + & Y_2 & Y_1 & Y_0 \\ \hline 1 & 0 & 1 & 1 \end{array}$$

PicatSAT infers the following equivalences:

$$\begin{aligned} \neg X_0 &= Y_0 \\ \neg X_1 &= Y_1 \\ X_2 &= Y_2 \\ X_2 &= 1 \\ Y_2 &= 1 \end{aligned}$$

The last two equivalences, $X_2 = 1$ and $Y_2 = 1$, are obtained by Rule-3 and the fact that $Z_3 = 1$.

When PicatSAT infers an equivalence between two variables or between one variable and another variable's negation, PicatSAT only uses one variable in the CNF code for the two variables, and eliminates the two CNF clauses for the equivalence.

4 Constant Propagation for $X \times Y = Z$

PicatSAT adopts the *shift-and-add* algorithm for multiplication. Let `X.m` be $X_{n-1} \ldots X_1 X_0$. The *shift-and-add* algorithm generates the following conditional constraints for $X \times Y = Z$.

$$\begin{aligned} &X_0 = 0 \Rightarrow S_0 = 0 \\ &X_0 = 1 \Rightarrow S_0 = Y \\ &X_1 = 0 \Rightarrow S_1 = S_0 \\ &X_1 = 1 \Rightarrow S_1 = (Y << 1) + S_0 \\ &\vdots \\ &X_i = 0 \Rightarrow S_i = S_{i-1} \\ &X_i = 1 \Rightarrow S_i = (Y << i) + S_{i-1} \\ &\vdots \\ &X_{n-1} = 0 \Rightarrow S_{n-1} = S_{n-2} \\ &X_{n-1} = 1 \Rightarrow S_{n-1} = (Y << (n-1)) + S_{n-2} \\ &Z = S_{n-1} \end{aligned}$$

The operation $(Y << i)$ shifts the binary string of Y to left by i positions. Let the length of the binary string of Y be u. The length of S_0 is u, that of S_1 is $u + 1$, and so on. So the total number of auxiliary Boolean variables that are created to hold the sums is $\Sigma_{i=u}^{(n+u-2)} i$ plus the number of auxiliary variables used for carries in the additions. Note that because S_{n-1} is the same as Z, S_{n-1} is never created.

If either X or Y is a constant, the basic algorithm can be improved to reduce the number of auxiliary variables. In the following of this subsection, we assume that X is a constant.

In the conditional constraints:

$X_i = 0 \Rightarrow S_i = S_{i-1}$
$X_i = 1 \Rightarrow S_i = (Y << i) + S_{i-1}$

If X_i is 0, then S_i is the same as S_{i-1}, and the variables in S_{i-1} can be reused for S_i. If X_i is 1, then the lowest i bits of S_i are the same as the lowest i bits of S_{i-1}, and these i variables can be copied from S_{i-1} to S_i. The following two rules perform these propagation:

Rule 5 : $X_i = 0 \Rightarrow$ copy all of the bits of S_{i-1} into S_i.
Rule 6 : $X_i = 1 \Rightarrow$ copy the lowest i bits of S_{i-1} into S_i.

Let X_i be the lowest bit of X that is 1, meaning that $X_j = 0$ for $j \in 0..i-1$. Then Z_i is the same as Y_0, the lowest bit of Y, and $Z_k = 0$ for $k \in 0..i-1$. Here is the rule that performs this propagation:

Rule 7 : $X.m = <X_{n-1} \ldots X_i 0 \ldots 0> \wedge X_i = 1 \quad \Rightarrow$
$Z_i = Y_0 \wedge Z_k = 0$ for $k \in 0..(i-1)$.

In particular, if X is a power of 2, then Z is the result of shifting Y to left by $n-1$ positions, and no auxiliary variables are needed.

The number of additions performed by the shift-and-add algorithm is the number of 1s in the binary string of X. If X is a constant that is not a power of 2 but is close to a power of 2, then PicatSAT converts the multiplication into an addition. The following rules perform this optimization:

Rule-8 : $X = 2^K - 1 \quad \Rightarrow$
rewrite $X \times Y = Z$ into $2^K \times Y = Z + Y$.
Rule-9 : $X = 2^K - 2 \quad \Rightarrow$
rewrite $X \times Y = Z$ into $2^K \times Y = Z + 2 \times Y$.

For example, PicatSAT converts the constraint $7 \times X = Z$ to $8 \times X = Z + X$, which requires one addition while the original constraint requires three additions.

5 Experimental Results

PicatSAT, which is implemented in Picat, has about 8,000 lines of code, excluding comments. PicatSAT connects to Lingeling SAT solver (version 587f) through a C interface.

We have done two experiments in order to evaluate the compiler using Picat version 2.1.[10] In the first experiment, we ran PicatSAT on several benchmark problems that involve arithmetic constraints, and measured the code size and the execution time of each of the benchmarks. These results show how effective equivalence reasoning is on reducing the code size and the execution time.

[10] http://picat-lang.org

In the second experiment, which was conducted on Linux Ubuntu 14.04LTS with an Intel i7-5820K 3.30GHz CPU and 32GHz RAM, we compared PicatSAT with Chuffed[11] and fzn2smt version 2.0.02.[12] Chuffed is a cutting-edge LCG solver. The `fzn2smt` solver, which took the second place in MiniZinc Challenge 2012, translates FlatZinc[13] into SMT that is solved by Yices[14].

This section does not include comparisons of PicatSAT with many other state-of-the-art CSP solvers. In the free search category of the MiniZinc Challenge 2016, an old version of PicatSAT outperformed some of the fastest CSP solvers, and was only second to HaifaCSP [15] by a small margin.

Table 1 evaluates how effective equivalence reasoning is on reducing the code size (#vars, the number of variables, and #cls, the number of clauses). The first three benchmarks are well known puzzles in the CP community. The next three are instances of the magic square problem for the grid sizes of 7×7, 8×8, and 9×9. The remaining 5 benchmarks are integer programming benchmarks taken from [30].[16] The column PicatSAT^{nor} shows the results obtained with the equivalence reasoning optimization disabled, and the column PicatSAT shows the results obtained when the optimization was enabled.

Table 1. Evaluation of effectiveness of equivalence reasoning (code size)

Benchmark	PicatSAT^{nor}		PicatSAT	
	#vars	#cls	#vars(%)	#cls(%)
crypta	3445	15374	1893 (**54**)	11537 (**75**)
eq10	10212	46087	6043 (59)	36696 (79)
eq20	19292	86469	11397 (59)	68869 (79)
magic_square_7	3543	81588	3463 (97)	81428 (99)
magic_square_8	6882	56324	6864 (99)	56288 (99)
magic_square_9	10306	86268	10226 (99)	85908 (99)
maxclosed_10_100_10	3778	23219	3091 (83)	21150 (91)
maxclosed_10_100_100	25420	162110	21128 (83)	148580 (91)
maxclosed_10_200_10	3066	18864	2559 (83)	17418 (92)
maxclosed_20_100_1000	221454	1577283	198137 (89)	1464492 (92)
maxclosed_30_200_1000	417816	3078098	379477 (90)	2881243 (93)

[11] https://github.com/chuffed/chuffed, released in December 2016.

[12] The fzn2smt solver (http://ima.udg.edu/recerca/lap/fzn2smt) has not been updated since 2012. There have been no significant speedups in the past five years in SAT solvers, on which both PicatSAT and SMT are based. PicatSAT uses Lingeling version 587f, which was released in February 2011 but is still faster than recent versions on most MiniZinc Challenge benchmarks. Therefore, this comparison is still relevant, if not completely up to date.

[13] http://www.minizinc.org

[14] http://yices.csl.sri.com

[15] http://strichman.net.technion.ac.il/haifacsp/

[16] Instances that can be solved by the preprocessor are not included.

As can be seen, the optimization led to certain amounts of reduction in the code size for each of the benchmarks. For *crypta*, the reduction in code size is most significant; the number of variables is reduced to 54%, and the number of clauses is reduced to 75%. For *magic_square*, whose code size is dominated by an *all-different* constraint, the reduction in code size is only 1%.

Table 2 evaluates how effective equivalence reasoning is on reducing the compile time (comp) and the solving time (solve). The solving time of the CNF code was measured using Lingeling version 587f on a Cygwin notebook computer with 2.60GHz Intel i7 and 64GB RAM. Interestingly, while the code reduction for *magic_square* is the least significant, the speedups are the most significant. Overall, the optimization reduces both the compile time and the solving time, and the results show that the equivalence reasoning optimization is worthwhile to incorporate.

Table 2. Evaluation of effectiveness of equivalence reasoning (time, seconds)

Benchmark	PicatSAT^{nor}		PicatSAT	
	comp	solve	comp(%)	solve(%)
crypta	0.18	0.337	0.17 (94)	0.310 (91)
eq10	0.32	1.177	0.26 (81)	1.000 (84)
eq20	0.61	1.239	0.48 (80)	1.134 (91)
magic_square_7	0.50	2.414	0.47 (94)	1.020 (**42**)
magic_square_8	0.57	14.908	0.56 (98)	6.835 (45)
magic_square_9	0.66	37.501	0.63 (95)	25.326 (67)
maxclosed_10_100_10	0.43	0.420	0.40 (93)	0.393 (93)
maxclosed_10_100_100	2.50	2.041	2.36 (94)	1.877 (91)
maxclosed_10_200_10	0.23	0.353	0.25 (108)	0.306 (86)
maxclosed_20_100_1000	29.00	14.809	28.47 (98)	14.352 (96)
maxclosed_30_200_1000	42.713	31.030	39.58 (92)	28.584 (92)

Table 3 compares PicatSAT with fzn2smt and Chuffed on the instances used in the MiniZinc Challenge 2012.[17] All of the instances were translated from MiniZinc into FlatZinc using each individual solver's global constraints.[18] The time limit was set to 15 minutes per instance, which limits the total of the conversion time and the solving time. For each benchmark, the number in the parentheses is the total number of instances, and the number in each column indicates the number of completely solved instances by the solver. For optimization problems, an instance is considered solved if an optimal solution was given and its optimality was proven. PicatSAT solved 77 instances, while fzn2smt solved 52, and Chuffed solved 67 instances. This experiment demonstrates the com-

[17] The benchmarks of MiniZinc 2012 were used because fzn2smt took the second place in that competition, and didn't compete ever since.

[18] fzn2smt, which does not have any solver-specific globals, uses MiniZinc's default decomposer.

petitiveness of PicatSAT in comparison with two lazy-approach-based solvers. When equivalence reasoning was disabled, PicatSAT solved 75 instances. This result also shows the worthiness of the optimization.

Table 3. A comparison of three CSP solvers (solved instances)

Benchmark	PicatSAT	PicatSAT^{nor}	fzn2smt	Chuffed
amaze (6)	6	6	5	5
amaze2 (6)	6	6	6	6
carpet-cutting (5)	1	0	0	2
fast-food (5)	5	5	5	4
filters (5)	4	3	2	3
league (6)	3	3	2	2
mspsp (6)	6	6	6	6
nonogram (5)	5	5	5	5
parity-learning (7)	7	7	0	2
pattern-set-mining-k1 (2)	1	1	0	0
pattern-set-mining-k2 (3)	2	2	1	1
project-planning (6)	5	5	0	6
radiation (5)	3	3	0	2
ship-schedule (5)	5	5	5	5
solbat (5)	5	5	5	5
still-life-wastage (5)	5	5	3	5
tpp (7)	7	7	7	7
train (6)	1	1	0	1
vrp (5)	0	0	0	0
total (100)	77	75	52	67

6 Related Work

A wide variety of problems have been encoded into SAT and solved by SAT solvers. SAT is also the backbone of many logic language systems, such as formal methods [23], answer set programming [10, 19], and NP-SPEC [11]. PicatSAT is a compiler that translates high-level constraints into SAT. Other SAT compilers include BEE [25], FznTini [21], meSAT [33], and Sugar [34]. PicatSAT, like FznTini, adopts the log encoding, while the other compilers are based on the order encoding.

Despite the compactness of the log encoding, it has received little attention for CSP solving, probably because of its weak propagation strength and its requirement of engineering efforts. FznTini was the only known log-encoding based CSP solver before PicatSAT. FznTini employs the 2's complement encoding for domain variables, while PicatSAT uses the sign-and-magnitude encoding. Fzn-

Tini demonstrated the promise of the log encoding, but it lacks optimizations, and is not considered competitive with recent CP solvers [30].

The perception that eager SAT-encoding approaches are not suited to arithmetic constraints has motivated the development of lazy approaches such as SMT [6, 27] and lazy clause generation [17, 29] that integrates CP and SAT solving techniques. Recent MiniZinc competitions have been dominated by LCG solvers; the HaifaCSP solver [35] also performs learning during search.

Encodings, such as BDDs [1, 7] and sorting networks [16], have been proposed for special form of arithmetic constraints, such as Boolean cardinality constraints and Pseudo-Boolean (PB) constraints. Linear arithmetic constraints can be compiled into SAT through PB constraints [2]. This encoding through PB constraints is less compact than the adder-based log encoding, but has stronger propagation power. The hybrid encoding that integrates order and log encodings [32] compiles linear constraints that involve large-domain variables into SAT through PB constraints.

The idea of identifying equivalences and exploiting them to reduce code sizes has been explored in both SAT compilation and SAT solving. The BEE compiler [25] performs *equi-propagation*, which takes advantage of the properties of the order-encoding to infer equivalences. Although equivalences could be detected to some extent by SAT solvers, it is always beneficial to do the reasoning at compile time, because SAT solvers are unaware of the meaning of the original constraints, and it is expensive to detect equivalences at preprocessing time [15] or reason about them at solving time [24].

PicatSAT embodies optimizations used in CP systems for processing constraints [31], in language compilers for eliminating sub-expressions [3], and in hardware design systems for optimizing logic circuits [9]. The equivalence reasoning optimization, which is specific to the log encoding, has not been implemented in any other SAT compilers.

7 Conclusion

In this paper we have presented the PicatSAT compiler and its optimizations. PicatSAT employs the log encoding, which has received little attention by SAT-based CSP solvers for its lack of propagation strength. PicatSAT adopts optimizations from CP systems (preprocessing constraints to narrow the domains of variables), language compilers (decomposing constraints into basic ones and eliminating common subexpressions), and hardware design systems (using a logic optimizer to optimize codes for adders and other basic constraints). Furthermore, PicatSAT reasons about equivalences in arithmetic constraints, and exploit them to eliminate variables and clauses. With these optimizations, PicatSAT is able to generate more compact and faster code for arithmetic constraints.

This work has shed a light on the debate between the eager and lazy approaches to constraint solving with SAT. The failed attempts to find efficient encodings for arithmetic constraints have motivated the development of the lazy approaches. This paper has shown that the eager approach based on the opti-

mized log encoding is not as bad for arithmetic constraints as it was perceived before. A comparison with an SMT-based CSP solver and a LCG solver shows the competitiveness of PicatSAT.

In addition to the optimizations, including the novel equivalence reasoning optimization, and the engineering effort, the success of PicatSAT is also attributed to Picat, the implementation language. The log encoding is arguably more difficult to implement than the sparse and order encodings. Picat's features, such as attributed variables, unification, pattern-matching rules, and loops, are all put into good use in the implementation. There are hundreds of optimization rules, and they can be described easily as pattern-matching rules in Picat. Logic programming has been proven to be suitable for language processing in general, and for compiler writing in particular; PicatSAT has provided another testament.

The success of PicatSAT does not in any way undermine the lazy approaches: there are certainly many problems for which the lazy approaches prevail, and many theories incorporated in SMT solvers do not yet have efficient SAT encodings. A comprehensive comparison of eager encoding and lazy approaches is on the stack for future work.

PicatSAT still has plenty of room for improvement, especially concerning global constraints and special constraints. One direction for future work is to carry out these improvements.

The optimizations reported in this paper could be only the tip of the iceberg. Numerous algorithms and optimizations have been proposed for hardware design systems, such as multi-bit and multi-operand adders and multipliers. When multiple bits are considered at once, there will be more reasoning opportunities opening up. Another direction for future work is to investigate these algorithms for SAT encodings.

Acknowledgements

The authors would like to thank the anonymous reviewers for helpful comments. This project was supported in part by the NSF under grant number CCF1618046.

References

1. Ignasi Abío, Robert Nieuwenhuis, Albert Oliveras, Enric Rodríguez-Carbonell, and Valentin Mayer-Eichberger. A new look at BDDs for Pseudo-Boolean constraints. *J. Artif. Intell. Res. (JAIR)*, 45:443–480, 2012.
2. Ignasi Abío and Peter J. Stuckey. Encoding linear constraints into SAT. In *CP*, pages 75–91, 2014.
3. A. V. Aho, Monica S. Lam, R. Sethi, and J. D. Ullman. *Compilers: Principles, Techniques, Tools*. Addison-Wesley, 2007.
4. Ignacio Araya, Bertrand Neveu, and Gilles Trombettoni. Exploiting common subexpressions in numerical csps. In *CP*, pages 342–357, 2008.

5. Olivier Bailleux, Yacine Boufkhad, and Olivier Roussel. New encodings of Pseudo-Boolean constraints into CNF. In *Theory and Applications of Satisfiability Testing - SAT 2009*, pages 181–194, 2009.
6. Clark W. Barrett, Roberto Sebastiani, Sanjit A. Seshia, and Cesare Tinelli. Satisfiability modulo theories. In *Handbook of Satisfiability*, pages 825–885. 2009.
7. Constantinos Bartzis and Tevfik Bultan. Efficient BDDs for bounded arithmetic constraints. *International Journal on Software Tools for Technology Transfer (STTT)*, 8(1):26–36, 2006.
8. Lucas Bordeaux, Youssef Hamadi, and Lintao Zhang. Propositional satisfiability and constraint programming: A comparative survey. *ACM Comput. Surv.*, 38(4):1–54, 2006.
9. Robert King Brayton, Gary D. Hachtel, Curtis T. McMullen, and Alberto L. Sangiovanni-Vincentelli. *Logic Minimization Algorithms for VLSI Synthesis*. Kluwer Academic Publishers, 1984.
10. Gerhard Brewka, Thomas Eiter, and Miroslaw Truszczyński. Answer set programming at a glance. *Commun. ACM*, 54(12):92–103, 2011.
11. Marco Cadoli and Andrea Schaerf. Compiling problem specifications into SAT. *Artif. Intell.*, 162(1-2):89–120, 2005.
12. Jingchao Chen. A new SAT encoding of the at-most-one constraint. In *Proc. of the 9th Int. Workshop of Constraint Modeling and Reformulation*, 2010.
13. James M. Crawford and Andrew B. Baker. Experimental results on the application of satisfiability algorithms to scheduling problems. In *AAAI*, pages 1092–1097, 1994.
14. Bruno Dutertre and Leonardo Mendonça de Moura. A fast linear-arithmetic solver for DPLL(T). In *Computer Aided Verification, CAV*, pages 81–94, 2006.
15. Niklas Eén and Armin Biere. Effective preprocessing in SAT through variable and clause elimination. In *SAT*, pages 61–75, 2005.
16. Niklas Eén and Niklas Sörensson. Translating Pseudo-Boolean constraints into SAT. *JSAT*, 2(1-4):1–26, 2006.
17. Thibaut Feydy and Peter J. Stuckey. Lazy clause generation reengineered. In *CP*, pages 352–366, 2009.
18. Marco Gavanelli. The log-support encoding of CSP into SAT. In *CP*, pages 815–822, 2007.
19. Martin Gebser, Benjamin Kaufmann, André Neumann, and Torsten Schaub. Conflict-driven answer set solving. In *IJCAI*, pages 386–, 2007.
20. Ian P. Gent. Arc consistency in SAT. In *ECAI*, pages 121–125, 2002.
21. Jinbo Huang. Universal Booleanization of constraint models. In *CP*, pages 144–158, 2008.
22. Kazuo Iwama and Shuichi Miyazaki. SAT-varible complexity of hard combinatorial problems. In *IFIP Congress (1)*, pages 253–258, 1994.
23. Daniel Jackson. *Software Abstractions: Logic, Language, and Analysis*. MIT Press, 2012.
24. Chu Min Li. Integrating equivalency reasoning into davis-putnam procedure. In *AAAI*, pages 291–296, 2000.
25. Amit Metodi and Michael Codish. Compiling finite domain constraints to SAT with BEE. *Theory and Practice of Logic Programming*, 12(4-5):465–483, 2012.
26. Robert Nieuwenhuis. The IntSat method for integer linear programming. In *CP*, pages 574–589, 2014.
27. Robert Nieuwenhuis, Albert Oliveras, and Cesare Tinelli. Solving SAT and SAT modulo theories: From an abstract davis–putnam–logemann–loveland procedure to DPLL(T). *J. ACM*, 53(6):937–977, 2006.

28. Peter Nightingale, Patrick Spracklen, and Ian Miguel. Automatically improving SAT encoding of constraint problems through common subexpression elimination in savile row. In *CP*, pages 330–340, 2015.
29. Olga Ohrimenko, Peter J. Stuckey, and Michael Codish. Propagation via lazy clause generation. *Constraints*, 14(3):357–391, 2009.
30. Justyna Petke. *Bridging Constraint Satisfaction and Boolean Satisfiability*. Artificial Intelligence: Foundations, Theory, and Algorithms. Springer, 2015.
31. Francesca Rossi, Peter van Beek, and Toby Walsh. *Handbook of Constraint Programming*. Elsevier, 2006.
32. Takehide Soh, Mutsunori Banbara, and Naoyuki Tamura. Proposal and evaluation of hybrid encoding of CSP to SAT integratin order and log encodings. *International Journal on Artificial Intelligence Tools*, 26(1):1–29, 2017.
33. Mirko Stojadinovic and Filip Maric. meSAT: multiple encodings of CSP to SAT. *Constraints*, 19(4):380–403, 2014.
34. Naoyuki Tamura, Akiko Taga, Satoshi Kitagawa, and Mutsunori Banbara. Compiling finite linear CSP into SAT. *Constraints*, 14(2):254–272, 2009.
35. Michael Veksler and Ofer Strichman. Learning general constraints in CSP. *Artif. Intell.*, 238:135–153, 2016.
36. Toby Walsh. SAT v CSP. In *CP*, pages 441–456, 2000.
37. Joost P. Warners. A linear-time transformation of linear inequalities into conjunctive normal form. *Inf. Process. Lett.*, 68(2):63–69, 1998.
38. Neng-Fa Zhou and Håkan Kjellerstrand. The Picat-SAT compiler. In *PADL*, pages 48–62, 2016.