Single Constant Multiplication for SAT

Hendrik Bierlee[1,2(✉)], Jip J. Dekker[1,2], Vitaly Lagoon[3], Peter J. Stuckey[1,2], and Guido Tack[1,2]

[1] Monash University, Wellington Road, Clayton, VIC 3800, Australia
{hendrik.bierlee,jip.dekker,peter.stuckey,guido.tack}@monash.edu
[2] OPTIMA ARC Industrial Training and Transformation Centre, Carlton, Australia
[3] Cadence Design Systems, San Jose, USA
lagoon@cadence.com

Abstract. This paper presents new methods of encoding the multiplication of a binary encoded integer variable with a constant value for Boolean Satisfiability (SAT) solvers. This problem, known as the Single Constant Multiplication (SCM) problem, is widely studied for its application in hardware design, but its techniques are currently not employed for SAT encodings. Considering the smaller and variable bit sizes and the different cost of operations in SAT, we propose further improvements to existing methods by minimizing the number of full/half adders, rather than the number of ripple carry adders. We compare our methods to simple encodings into adders, currently employed by SAT encoders, and direct Boolean encoding using logic minimization tools. We show that our methods result in improved solve-time for problems involving integer linear constraints. A library of optimal recipes for each method to encode SCM for SAT is made available as part of this publication.

Keywords: Single Constant Multiplier · Boolean Satisfiability · Encoding constraints

1 Introduction

Boolean Satisfiability (SAT) is a powerful approach to solving combinatorial problems, but in order use a SAT solver, we need to encode the problem into clauses. One fundamental constraint is the constant multiplication $y = c \times x$ of input and output integer variables x and y, and constant c. If x and y are encoded using a binary representation, any such multiplication can be achieved using a combination of additions, subtractions, and multiplications by a power of two. For example, the multiplication $5 \times x$ can be decomposed into $4 \times x + 1 \times x$. Since multiplication by a power of two in a binary encoding is a simple left-shift operation, which does not incur any cost, this decomposition effectively turns the multiplication into a sequence of additions.

The most basic decomposition for a multiplication $c \times x$ therefore simply considers the binary representation of c and introduces one addend for every 1 in this binary representation – such as in the example of $5x = 1x + 4x$ above,

B. Dilkina (Ed.): CPAIOR 2024, LNCS 14742, pp. 84–98, 2024.
https://doi.org/10.1007/978-3-031-60597-0_6

or $117x = 64x + 32x + 16x + 4x + 1x$. However, decompositions that reuse intermediate results or use a combination of additions and subtractions can result in fewer additions. For example, we could define $63x = 64x - x$, $59x = 63x - 4x$, $117x = 2 \times 59x - x$. This only requires 3 additions/subtractions, compared to the 4 in the original example.

In hardware applications such as ASIC and FPGA, it is important for the performance and cost of the application to use small circuits, and therefore there is considerable previous research on finding a circuit for a given constant that minimizes the number k of additions and subtractions. This is known as the *Single Constant Multiplication* (SCM) problem. SCM is NP-hard [7], and has an upper bound of $k \leq \lceil \log_2(c)/2 \rceil$ [3]. For many practical instances of the problem, however, k is known to be much smaller.

When encoding constant multiplication for SAT, an approach similar to SCM can be used since integer variables – like in hardware – are often encoded in a binary representation [5]. Unlike hardware, the number of bits w to represent x is not fixed, but depends on the domain of x (e.g., only $w = 2$ bits are required to represent $x \in [0, 2]$ with Boolean variables x_0, x_1).

Furthermore, the complexity of the SAT encoding of a single addition or subtraction (using a ripple carry adder [17]) varies. The required number of full (and half) adders is determined by the constant as well as the number of bits of the inputs. In fact, the addition of $x+4x = 5x$ requires *no* adders if x has two bits x_0, x_1, since $5x$ would be represented using the existing bits x_0, x_1, x_0, x_1. If x has three bits, multiple adders are required. Consequently, instead of minimizing the number of additions/subtractions k, a better circuit for SAT minimizes the total number of full/half adders a.

Example 1. Consider two circuits for $c = 117$: $17x = 16x + x; 19x = 2x + 17x; 117x = 8 \times 17x(= 136x) - 19x$ and $63x = 64x - x; 59x = 63x - 4x; 117x = 2 \times 59x(= 118x) - x$. Both are minimal with respect to $k = 3$, but assuming $w = 4$ bits for $x \in [0, 7]$, the first requires $a = 18$ full/half adders, while the second requires $a = 31$. Yet another circuit, $3x = 2x + x; 49x = 16 \times 3x(= 48x) + x; 17x = 16x + x; 117x = 4 \times 17x(= 68x) + 49x$, requires only $a = 12$ full/half adders, even though it uses $k = 4$ additions/subtractions. For $w = 4$, $a = 12$ is optimal for $c = 117$, but for different w the best circuit changes.

Note that if we constrain all intermediates to be represented in a fixed bit width (common in hardware solutions), then this may change the answer. For example, given fixed 12 bit width for intermediates the first recipe is not applicable when x is a 5 bit number, since intermediate $136x$ is not guaranteed to fit in 12 bits, while $117x$ does. □

The main contribution of this paper is the application of the SCM methodology in the context of SAT, the extension to the basic approach that result in better SAT encodings by minimizing full/half adders, and a database with pre-computed *recipes* of the SCM problem for a range of constants and bit widths. Our database exhaustively covers the parameter space over $1 \leq w \leq 16$ and $1 \leq c \leq 2047$. We assume no fixed bit width restriction on intermediate results.

We present a set of recipes to compute $c \times x$ optimal in terms of number of additions/subtractions for the general SCM problem, independent of the size of x, assuming we can use any number of bits for intermediate results. Then, we produce a set of recipes that minimize the number of full/half adders to compute $c \times x$. These recipes differ depending on the bit size of x. The SCM SAT encoding database is available as part of the released implementation and benchmarks [8].

We evaluate the different SCM recipes in Sect. 4, where we use the pre-computed circuits to encode problems which focus on constant multiplication. We observe significant improvements in terms of the size of the final encoding and the SAT solver performance. In Sect. 5, we discuss future work and conclude our findings.

2 Preliminaries and Related Work

In this section, we give preliminaries on the *Constraint Satisfaction Problem* (CSP) and SAT problem. We then formally define the SCM problem and discuss existing approaches for solving it.

2.1 Constraint Satisfaction Problem

A CSP [15] instance, $P = (\mathcal{X}, D, \mathcal{C})$, consists of a set of variables $\mathcal{X}$, with each $x \in \mathcal{X}$ restricted to take values from some domain $D(x)$. For this paper we assume domains are ordered sets of integers, and denote by $\text{lb}(x)$ and $\text{ub}(x)$ the least and greatest values in $D(x)$. We will use interval notation $[l, u]$ to represent the set of integers $\{l, l+1, \ldots, u\}$. A set of constraints $\mathcal{C}$ expresses relationships between the variables. An *assignment* of a CSP instance is a mapping of variables to values. An assignment that is consistent with the domains and constraints is a *solution* to the instance. A *Constraint Optimization Problem* (COP) is a CSP with an additional objective function f that assigns a value to each assignment. An optimal solution to a minimization (maximization) COP is one that minimizes (maximizes) f.

2.2 Boolean Satisfiability

A SAT problem can be seen as a special case of a CSP, where the domain for all variables x is $D(x) \in \{0, 1\}$, representing the values *false* and *true*. A *literal* is either a Boolean variable x or its negation $\neg x$. We extend the negation operation to operate on literals, i.e., $\neg l = \neg x$ if $l = x$ and $\neg l = x$ if $l = \neg x$. We use the notation $l = v$ where l is a literal and $v \in \{0, 1\}$ to encode the appropriate form of the literal, i.e., if $v = 1$ it is equivalent to l and if $v = 0$ it is equivalent to $\neg l$. A *clause* is a disjunction of literals. In a SAT problem P, the constraints are in *Conjunctive Normal Form* (CNF), which means that $\mathcal{C}$ is a set of clauses.

A general CSP can be mapped, or encoded, into SAT, by encoding each integer variable into a set of Boolean *encoding* variables, and each constraint into a set of clauses and additional auxiliary Boolean variables. There are a

number of different ways of encoding integer variables [5], such as the direct and order encodings. In this paper, we focus on the *binary* encoding.

Given an integer variable x with non-negative, possibly non-contiguous domain $D(x)$, the unsigned binary encoding method maps x to $m_x = \lfloor \log_2 \mathrm{ub}(x) \rfloor + 1$ encoding variables $[\![\mathrm{bit}(x,i)]\!], 0 \leq i < m_x$. The semantics of the encoding can be expressed by the equation $x = \sum_{i=0}^{m_x-1} 2^i \times [\![\mathrm{bit}(x,i)]\!]$. For example, for integer variables d with domain $D(d) = [0, 127]$, an encoding with $m_d = 7$ bits can be used. In an assignment with $d = 117 = 0b1110101$, we would have $[\![\mathrm{bit}(d,0)]\!] = 1$, $[\![\mathrm{bit}(d,1)]\!] = 0$, $[\![\mathrm{bit}(d,2)]\!] = 1$ and so on. If the initial domain $D(x)$ is non-contiguous, or has bounds that are not powers of 2, these have to be enforced using additional constraints (clauses). The representation can be extended in a straightforward way to support negative domain values, by assuming a two's complement encoding instead of a simple unsigned binary encoding. For the rest of the paper, we will assume non-negative domains to simplify presentation, and only discuss the general approach where required.

2.3 The Single Constant Multiplication Problem

The SCM problem can be formalized as follows. Given the multiplier target c, we are to decide a sequence of up to n equations of the form $c' \times x = 2^{s_l} \times (c_l \times x) \pm 2^{s_r} \times (c_r \times x)$. The left argument $c_l \times x$ is either a previously computed multiple or $c_l = 1$, and $s_l \geq 0$ is the left-shift of the left argument. Similarly, $c_r \times x$ is the right argument, and $s_r \geq 0$ is its left-shift.[1] The final equation should define $c \times x$.

We can thus encode a recipe for $c = c^k$ as a sequence of k equations represented by tuples $\langle c_l^i, s_l^i, c_r^i, s_r^i, \#^i, c^i \rangle$, where $c^i = (c_l^i \ll s_l^i)\ \#^i\ (c_r^i \ll s_r^i)$ such that $k \leq n$, $\#^i \in \{+, -\}$, $\{c_l^i, c_r^i\} \subseteq \{c^j \mid j < i\} \cup \{1\}$ and $s_l^i, s_r^i \geq 0$.

Example 2. The first recipe for 117 of Example 1 is defined by the sequence $[\langle 1,4,1,0,+,17\rangle,\ \langle 1,1,17,0,+,19\rangle,\ \langle 17,3,19,0,-,117\rangle]$. The second recipe is defined by $[\langle 1,6,1,0,-,63\rangle,\ \langle 63,0,1,2,-,59\rangle,\ \langle 59,1,1,0,-,117\rangle]$.

Usually, the goal of the SCM problem is to find the minimal k recipe for a given c (knowing that it is bounded by $k \leq \lceil \log_2(c)/2 \rceil$ [3]).

Related Work. The SCM problem has been studied extensively in the context of hardware. Many approaches for finding optimal circuits have been proposed, using graph based techniques [9], or using *Integer Linear Programming* (ILP) solvers [11], SAT solvers [2,12], as well as heuristic approaches [2]. A generalization of SCM is the MCM, which builds a circuit for multiple target constants. Compared to SCM, this enables further sharing of intermediate results between the circuits for different constants. Apart from minimizing the number of nodes

[1] Some versions of SCM also allows right shifts, and while these are necessary for optimal *Multiple Constant Multiplication* (MCM) solutions, we did not come across any instances of SCM where they improved the solution.

(additions and subtractions), some of these related works consider other objectives such as minimizing the required surface area, delay, or power consumption of the generated circuits. These metrics are obviously useful when generating hardware circuits, but they are not directly relevant for SAT encodings. We are not aware of approaches that minimize the number of full/half adders in the resulting circuit. This metric is useful in the context of SAT solvers, since it results in a smaller CNF in terms of the number of variables, clauses and literals, as we shall see in Sect. 4. We briefly discuss MCM for SAT as future work in Sect. 5.

Another difference with existing approaches is that the domains of intermediate results is left unbounded when solving SCM for SAT. Approaches that model hardware circuits have a fixed bit width h for all results $c' \times x$. We could modify our approach easily by fixing the number of bits for the result (and intermediates) as well. Either we would allow overflow (computing $(c \times x) \mod 2^h$), or we would enforce the (intermediate) results to always fit in h bits. However, for SAT we can consider arbitrary bit widths (up to $m_{c\times x}$ bits) for the intermediate results.

3 Applying Single Constant Multiplication to SAT

In this section, we present different approaches for solving the SCM problem. We start with well-known, simple approaches, and then introduce an encoding of SCM into COP. We discuss these approach with a particular focus on their use in SAT encodings.

3.1 A Baseline Algorithm for Encoding $y = c \times x$

In order to establish a baseline, we will first discuss a well-known, simple solution to the SCM problem that does not optimize the resulting circuit at all.

Multiplication by a constant can always be represented using a combination of shifts and additions. Suppose x is defined as a m_x width binary encoded integer. Then we know that $c \times x$ can be at most $c \times (2^{m_x} - 1)$, thus requiring $m_{c\times x} = \lceil \log(c \times (2^{m_x} - 1)) \rceil$ bits to encode.

A basic shift-and-add algorithm will construct a solution to the above SCM problem by decomposing $c \times x$ by a shifted term for every 1 in c's binary representation:

$$c \times x = \sum_{\substack{0 \leq i < m_c \\ [\![\text{bit}(c,i)]\!]=1}} 2^i x$$

The number of additions required is equal to the number of 1s in c's binary representation minus one, or at most $\lceil \log(c) \rceil - 1$.

3.2 Using Boolean Circuit Minimization to Tackle SCM

Another option for encoding SCM into SAT is to use algorithms such as the Espresso [6] logic minimizer that directly produce small logic circuits based on truth tables or similar representations of Boolean formulae. These algorithms are remarkably powerful, and for small enough problems can produce a guaranteed minimal size circuit for a formula *without introducing auxiliary Boolean variables for intermediate results.*

Example 3. For example, we can construct a CNF encoding for $y = 117 \times x$ with $x \in [0, 15]$ by feeding the following truth table between the x and y bits into Espresso:

	x	y		x	y
0	0 0 0 0	0 0 0 0 0 0 0 0 0 0 0 0	8	1 0 0 0	0 1 1 1 0 1 0 1 0 0 0
1	0 0 0 1	0 0 0 0 1 1 1 0 1 0 1	9	1 0 0 1	1 0 0 0 0 0 1 1 1 0 1
2	0 0 1 0	0 0 0 1 1 1 0 1 0 1 0	10	1 0 1 0	1 0 0 1 0 0 1 0 0 1 0
3	0 0 1 1	0 0 1 0 1 0 1 1 1 1 1	11	1 0 1 1	1 0 1 0 0 0 0 0 1 1 1
4	0 1 0 0	0 0 1 1 1 0 1 0 1 0 0	12	1 1 0 0	1 0 1 0 1 1 1 1 1 0 0
5	0 1 0 1	0 1 0 0 1 0 0 1 0 0 1	13	1 1 0 1	1 0 1 1 1 1 1 0 0 0 1
6	0 1 1 0	0 1 0 1 0 1 1 1 1 1 0	14	1 1 1 0	1 1 0 0 1 1 0 0 1 1 0
7	0 1 1 1	0 1 1 0 0 1 1 0 0 1 1	15	1 1 1 1	1 1 0 1 1 0 1 1 0 1 1

The circuit minimization of this truth table results in a CNF encoding with 15 variables (simply representing the bits of x and y) and 84 clauses.

Unfortunately as the width w of the input variable x and the size of the constant c grows, the number of Boolean variables in the formula $y = c \times x$ quickly becomes too large for logic minimization. However, for small widths w and constants c logic minimization can generate the best SAT encoding for $y = c \times x$, as demonstrated by our experiments in Sect. 4.

3.3 Formulating SCM as a COP

We will now introduce a general method for solving the SCM problem, by formulating it as a COP.

The following proposition shows that we only need concern ourselves with generating SCM encodings for odd numbers c (since we can derive even multiples by left shifts).

Proposition 1. *Suppose* $c = 2^i c'$ *where* $c' \mod 2 = 1$ *and* $i \geq 1$ *then we can compute* $c \times x$ *from* $c' \times x$ *without any operations.*

Given the above result we can reduce the possible forms of the equation to just three:

Proposition 2. *Given a* k *equation recipe to compute* $c \times x, c \mod 2 = 1$ *using the general form above, then there is a* k *equation recipe using equations of the form* $c' =$ `SPLUS`(c_l, s, c_r)*:* $c' = 2^s c_l + c_r$*,* $c' =$ `SMINUS`(c_l, s, c_r)*:* $c' = 2^s c_l - c_r$*, and* $c' =$ `MINUSS`(c_l, s, c_r)*:* $c' = c_l - 2^s c_r$*.*

Proof. Given a k length list of tuples computing c, we show by induction we can replace this by an equal length list of the new equation forms:

Suppose the first tuple not of these forms is $\langle c_l, s_l, c_r, s_r, \#, c' \rangle$. If $s_r = 0$ then we can rewrite this as either $c' = \texttt{SPLUS}(c_l, s_l, c_r)$ if $\# = +$ or $c' = \texttt{SMINUS}(c_l, s_l, c_r)$ if $\# = -$. If $s_l = 0$ then we can rewrite this as either $c' = \texttt{SPLUS}(c_r, s_r, c_l)$ if $\# = +$ or $c' = \texttt{MINUSS}(c_l, s_r, c_r)$ if $\# = -$.

Otherwise, $s_l \geq 1 \wedge s_r \geq 1$. Suppose $s_l = s_r$, then we can rewrite the tuple to either $c'' = \texttt{SPLUS}(c_r, 0, c_l)$ if $\# = +$ or $c'' = \texttt{SMINUS}(c_l, 0, c_r)$ if $\# = -$, and replace all later usages of c' by c'' by adding s_l to the matching shift argument. Suppose $s_l > s_r$ then we can replace the tuple by recipe $c'' = \texttt{SPLUS}(c_l, s_l - s_r, c_r)$ if $\# = +$ and $c'' = \texttt{SMINUS}(c_l, s_l - s_r, c_r)$ if $\# = -$, and replace later uses of c' by c'' adding s_r to the shift argument similarly. Suppose $s_l < s_r$ then we can replace the tuple by recipe $c'' = \texttt{SPLUS}(c_r, s_r - s_l, c_l)$ if $\# = +$ and $c'' = \texttt{MINUSS}(c_l, s_r - s_l, c_r)$ if $\# = -$, and replace later uses of c' by c'' adding s_r to the shift argument similarly.

The requirement that $c \mod 2 = 1$ means that one of the first two cases is applicable to the last equation. □

Example 4. Both recipes for 117 of Example 2 can be represented equivalently as $[17 = \texttt{SPLUS}(1, 4, 1),\ 19 = \texttt{SPLUS}(1, 1, 17),\ 117 = \texttt{SMINUS}(17, 3, 19)]$. The second recipe is defined by $[63 = \texttt{SMINUS}(1, 6, 1),\ 59 = \texttt{MINUSS}(63, 2, 1),\ 117 = \texttt{SMINUS}(59, 1, 1)]$.

A MiniZinc [13] model for solving SCM as a combinatorial optimization problem is shown in Listing 1. Note that it only makes use of the three recipes from Proposition 2. Each equation is represented by its type `ty`, with `NOP` added for an unused equation; the equation numbers defining its left (`left`) and right (`right`) arguments, and the shift amount `shift`.[2] We use equation number 0 to represent $1 \times x$. The multiplier resulting from an equation is defined by `mult`. Line 14 sets the dummy equation multiplier to 1, and enforces the final result to be the target. In lines 16–18 we enforce that all equations after `used` are NOPs, and set their other components to dummy values. Line 19 enforces that equations use only earlier defined multiples. The computation of the multiplier for each equation (lines 20–27) follows the definition, where `NOP` just returns the previous multiplier. We minimize the number of used equations. Note that the model "pre-computes" the powers of 2 in `p2` (line 13), since solvers typically propagate poorly for exponential expressions.

3.4 Encoding an SCM Recipe

Given a solution (e.g., Example 2) to the SCM problem, we can encode the shifts and adds of each equation $(c' \times x) = 2^{s_l} \times (c_l \times x) \pm 2^{s_r} \times (c_r \times x)$. We obtain the

[2] The careful reader will note that we limit the maximum shift using the target. Relaxing this restriction and never found better recipes (in terms of equations or adders), but we have no formal proof for it.

```
int: xbits; % number of bits to represent x
int: n;     % max number of shift +- equations
set of int: EQ = 1..n;
set of int: EQ0 = 0..n; % equation number 0 = 1x
int: maxsh = ceil(log2(target)); % maximum left shift
int: target;  % target multiplier
enum TYPE = { SPLUS, SMINUS, MINUSS, NOP };
array[EQ] of var TYPE: ty;   % type of equation
array[EQ] of var EQ0: left;  % left input equation numnber
array[EQ] of var EQ0: right; % right input equation number
array[EQ] of var 0..maxsh: shift; % shift left applied
array[EQ0] of var 0..infinity: mult; % multiplier value
array[1..maxsh+xbits] of int: p2 = [2^i | i in 1..maxsh+xbits];
constraint mult[0] = 1 /\ mult[n] = target;
var EQ0: used; % number of equations used
constraint forall(e in EQ)(e > used <-> ty[e] = NOP);
constraint forall(e in EQ)(e > used ->
                 left[e] = 0 /\ right[e] = 0 /\ shift[e] = 1);
constraint forall(e in EQ)(left[e] < e /\ right[e] < e);
constraint forall(e in EQ)(mult[e] =
  if ty[e] = SPLUS then
     p2[shift[e]] * mult[left[e]] + mult[right[e]]
  elseif ty[e] = SMINUS then
     p2[shift[e]] * mult[left[e]] - mult[right[e]]
  elseif ty[e] = MINUSS then
     mult[left[e]] - p2[shift[e]] * mult[right[e]]
  else mult[e-1] endif);
solve minimize used; % minimize equations
```

Listing 1. A MiniZinc model to find a recipe of at most size `n` for computing `target` $\times$ `x` where `x` is represented by `xbits` bits.

binary encoding of the output $z = (c' \times x)$ from the already computed binary encodings of the inputs $z_l = (c_l \times x)$ and $z_r = (c_r \times x)$.

To apply a shift $y_l = z_l \ll s$ (or equivalently, $y_l = 2^s z_l$) on the binary encoding of z_l (or to apply $y_r = z_r \ll s$), we simply extend its encoding by $[\![\text{bit}(y_l, i)]\!] = 0, 0 \leq i < s$ and $[\![\text{bit}(y_l, i)]\!] = [\![\text{bit}(z_l, i - s)]\!], s \leq i \leq m_{z_l}$ using no additional variables or clauses.

After shifting both inputs, it remains to encode an addition of the form $z = y_l + y_r$. Here, we can encode a ripple carry adder [17] which produces the output (sum) bits $[\![\text{bit}(z, i)]\!]$, and auxiliary carry bits $[\![\text{bit}(c, i)]\!]$. Inputs variables y_l, y_r might contain fixed literals due to the applied shifts, which in turn leads to output variables that are fixed, or that are equivalent to input bits, or their negations. The initial carry bit is also fixed as $[\![\text{bit}(r, 0)]\!] = 0$ (which is essentially why a half-adder is used to add $[\![\text{bit}(y_l, 0)]\!]$ and $[\![\text{bit}(y_r, 0)]\!]$). We optimize the ripple carry adder encoding to account for this.

The current sum bit $[\![\text{bit}(z, i)]\!]$ is the result of the `xor` operation on up to three input variables, $[\![\text{bit}(y_l, i)]\!] \oplus [\![\text{bit}(y_r, i)]\!] \oplus [\![\text{bit}(r, i)]\!]$, which we reformulate more generically as $\oplus([[\![\text{bit}(y_l, i)]\!], [\![\text{bit}(y_r, i)]\!], [\![\text{bit}(r, i)]\!]])$.

We remove any fixed variables from the `xor`'s input, and compute the sum of their values. Then, we apply the `xor` on the remaining non-fixed variables, negating the output if the fixed sum is odd. Given a list l of bits, some of which are fixed, define $\mathsf{fs}(l)$ as the sum of the fixed bits in l and $\mathsf{vb}(l)$ the list of unfixed (variable) bits in l. We can encode the $[\![\text{bit}(z,i)]\!] = \oplus([[\![\text{bit}(y_l,i)]\!], [\![\text{bit}(y_r,i)]\!], [\![\text{bit}(r,i)]\!])$ as follows:

$$\oplus(l) = \begin{cases} \mathsf{fs}(l) \mod 2, & \text{if } \mathsf{vb}(l) = [] \\ b_1, & \text{if } \mathsf{vb}(l) = [b_1] \wedge \mathsf{fs}(l) = 0 \\ \neg b_1, & \text{if } \mathsf{vb}(l) = [b_1] \wedge \mathsf{fs}(l) = 1 \\ b_1 \oplus b_2 & \text{if } \mathsf{vb}(l) = [b_1, b_2] \wedge \mathsf{fs}(l) = 0 \\ b_1 \oplus b_2 \oplus 1 & \text{if } \mathsf{vb}(l) = [b_1, b_2] \wedge \mathsf{fs}(l) = 1 \\ b_1 \oplus b_2 \oplus b_3 & \text{otherwise } \mathsf{vb}(l) = [b_1, b_2, b_3] \end{cases}$$

If $\oplus(l)$ is constant or a literal, we can just equate $[\![\text{bit}(z,i)]\!]$ with the result, requiring no encoding clauses. Otherwise, $[\![\text{bit}(z,i)]\!]$ is equated to a $\oplus$ expression, which can be encoded using a single XOR clause if supported by the SAT solver [16], or encoded with (standard) clauses in a well understood manner.

The next carry bit is set to true if at least 2 of its input variables are. In other words, $[\![\text{bit}(r,i+1)]\!]$ is true if and only if the expression $[\![\text{bit}(y_l,i)]\!] + [\![\text{bit}(y_r,i)]\!] + [\![\text{bit}(r,i)]\!] \geq 2$ is true. Again, we split off and sum the fixed variables, and compute the carry by evaluating the generic version of the at-least-2 constraint as follows:

$$\left(\sum(l) \geq 2\right) = \begin{cases} 1, & \text{if } \mathsf{fs}(l) \geq 2 \\ 0 & \text{if } \mathsf{vb}(l) = [] \wedge \mathsf{fs}(l) < 2 \\ 0, & \text{if } \mathsf{vb}(l) = [b_1] \wedge \mathsf{fs}(l) = 0 \\ b_1, & \text{if } \mathsf{vb}(l) = [b_1] \wedge \mathsf{fs}(l) = 1 \\ b_1 \wedge b_2, & \text{if } \mathsf{vb}(l) = [b_1, b_2] \wedge \mathsf{fs}(l) = 0 \\ b_1 \vee b_2, & \text{if } \mathsf{vb}(l) = [b_1, b_2] \wedge \mathsf{fs}(l) = 1 \\ (b_1 \wedge b_2) \vee (b_1 \wedge b_3) \vee (b_2 \wedge b_3) & \text{otherwise } \mathsf{vb}(l) = [b_1, b_2, b_3] \end{cases}$$

If $\sum(l) \geq 2$ is constant or a literal, we can just equate $[\![\text{bit}(r,i+1)]\!]$ with the result, requiring no encoding clauses. Otherwise, $[\![\text{bit}(r,i+1)]\!]$ is equated to the CNF expression as normal.

To handle subtraction, we use the fact that ripple-carry adders directly handle 2's complement. So, $y_l + (-y_r) = y_l + (\bar{y_r} + 1)$ where the encoding of $\bar{y_r}$ is the complement of y_r, i.e., $[\![\text{bit}(\bar{y_r},i)]\!] = \neg[\![\text{bit}(y_r,i)]\!]$. To offset the constant 1, we can add an initial carry of 1.

3.5 Minimizing the Number of Adders

The COP formulation of Sect. 3.3 minimizes the number of equations, k. However, we show in Sect. 3.4 how the true encoding complexity for each equation is more closely related to the number of (full/half) adders. In this section, we formulate an alternative objective which minimizes the number of adders a instead.

We are now required to consider the bit width w of the input x being multiplied as an additional parameter. Consider the addition $c' \times x = 2^{s_l} \times (c_l \times x) + 2^{s_r} \times (c_r \times x)$. Assuming the left input $c_l \times x$ is represented in $m_{c_l \times x}$ bits and the right input $c_r \times x$ is represented in $m_{c_t \times x}$ bits, then the number of full/half adders naïvely required for the addition is the width of the result $m_{c' \times x}$. However, we can omit full/half adders for low bits where one argument is guaranteed 0, and if the non-zero bits never overlap we need no adders. So the number of full/half adders required is 0 if $s_l > m_{c_r \times x} + s_r$ or if $s_r > m_{c_l \times x} + s_l$, otherwise the number of adders required is $\max(m_{c_l \times x} + s_l, m_{c_r \times x} + s_r) - \max(s_l, s_r)$. For subtractions, the only bits not requiring adders are the low bits where both operands are known to be 0, so we require $m_{c' \times x} - \min(s_l, s_r)$ full/half adders.

We can similarly show that we do not need the more general recipes, we can restrict to the recipes of Proposition 2, since these also do not change the number of full/half adders required.

Proposition 3. *Given an a adder recipe to compute* $c \times x, c \mod 2 = 1$ *using the general form of* k *equations, then there is an a adder recipe using equations of the form* $c' = \texttt{SPLUS}(c_l, s, c_r)$*:* $c' = 2^s c_l + c_r$*,* $c' = \texttt{SMINUS}(c_l, s, c_r)$*:* $c' = 2^s c_l - c_r$*, and* $c' = \texttt{MINUSS}(c_l, s, c_r)$*:* $c' = c_l - 2^s c_r$*.*

Proof. (Sketch) The proof follows that of Proposition 2. Note that if we shift both addends right by the same amount, then we don't change the number of adders required since the low bits do not require adders. When we replace the addition of c' *by* c'' *shifted left by some amount* s*, we are simply computing the same addition since* $c' = 2^s c''$ *which does not change the number of adders required.* □

We can modify our MiniZinc model (see Listing 1) to keep track of the number of bits required for the result of each equation, and count the number of full/half adders used for each equation, and make that the new objective. The additions to the model (and replacement objective) are shown in Listing 2. The number of bits required for the result of each equation are defined by `bits` while the number of full/half adders for each equation are given by `adders`. In lines 4–5 we compute the bit width directly from the multiple (the least power big enough to hold the maximum result). We compute the number of adders required for each recipe in lines 6–12. Finally, we minimize the sum of adders required.

Note that for minimizing the SAT model we only minimize a proxy for the size of the resulting SAT model, the number of full/half adders. We experimented with directly minimizing the number of clauses or literals (sum of size of clauses) in the SAT encoding. The resulting models were much harder to solve, we usually couldn't prove optimality, and for the small examples where we could the result

```
1 array[EQ] of var 0..infinity: bits; % number of bits
2 array[EQ] of var 0..infinity: adders; % number of adders
3 include "arg_max.mzn";
4 constraint forall(e in EQ)(bits[e] =
5   arg_max([mult[e]*(p2[xbits]-1)<=p2[i]|i in 1..maxsh+xbits ]));
6 constraint forall(e in EQ)(adders[e] =
7   if ty[e] = SPLUS then
8      if shift[e] >= bits[right[e]] then 0
9      else max(bits[left[e]],bits[right[e]]-shift[e]) endif
10  elseif ty[e] = SMINUS then bits[e]
11  elseif ty[e] = MINUSS then bits[e]
12  else 0 endif);
13 solve minimize sum(adders);  % minimize adders
```

Listing 2. Additions to the MiniZinc model of Listing 1 to optimize for the number of full/half adders.

was rarely better than the SAT encoding resulting from minimizing full/half adders. It remains interesting future work to see if this more direct approach to encoding minimization could be improved.

4 Experimental Evaluation

To evaluate the effectiveness of the proposed approach, we compare the size of the encoding of $c \times x$ for a range of constants c and bit widths for x in Sect. 4.1. In Sect. 4.2 we apply our approach to a realistic benchmark, comparing both the encoding size and solve times.

4.1 Constructing and Analyzing the SCM Databases

In this section, we compare the different SCM approaches in terms of the size of their respective encodings. For each bit width $2 \leq w \leq 16$, we average the number of variables, clauses, and literals that are generated when encoding $c \times x$ over $1 \leq c \leq 2047$. The four methods are `base` (the baseline from Sect. 3.1), `espresso` (using logic minimization with Espresso 2.3 from Sect. 3.2), `min-k` (minimizing the number of ripple carry adders) and `min-a` (minimizing the total number of adders). It becomes increasingly hard for Espresso to compute CNF for all constants as the number of input bits grows, so for `espresso` we are limited to $w \leq 12$. The results are shown in Fig. 1.

It is clear that the naive decomposition of `base` produces by far the most variables. In contrast, `min-k` achieves the same multiplication using fewer ripple carry adders. Importantly, `min-a` has even fewer variables than `min-k`, indicating that minimizing the number of adders serves as a proxy for avoiding additional variables. The effect is stronger for smaller bit widths, presumably because then there is more opportunity for completely free additions where one input is shifted by the other input's bit width. The number of variables for `espresso` is always minimal, since no additional variables are ever constructed. In terms of clauses and literals, we see a linear relationship between the other methods other than `espresso`, which generates an exponential number of clauses and literals, `min-a` better than `min-k` better than `base`.

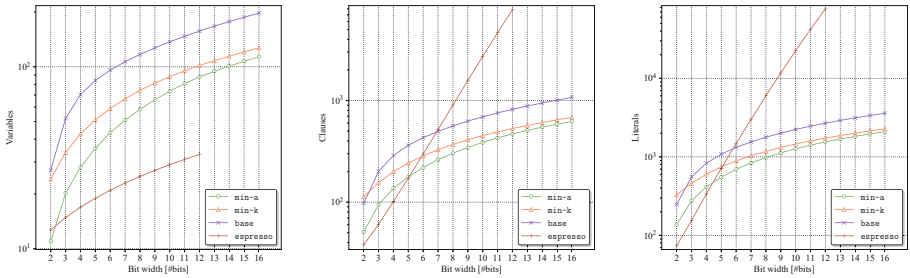

Fig. 1. Encoding size of $c \times x$ using different SCM approaches

4.2 Solving Multidimensional Bounded Knapsack Problem

To evaluate the different approaches, we adapt the approach from [10] to generate hard *Multidimensional Bounded Knapsack Problem* (MBKP) instances. In MBKP, we decide for N item types how many x_i to pack (up to B) such that a minimum profit $\sum_{i=1}^{N} x_i p_i \geq P$ is reached. Each item is restricted by M dimensions of weight with $\bigwedge_{i=1}^{M} \sum_{j=1}^{N} x_j w_{i,j} \leq W_i$. Instances can be generated by first generating C coefficient sets for $w_{i,j}$ and p_i, sampling uniformly from $[1, Q]$. Then, for each coefficient set, we choose S capacity sets for $1 \leq s \leq S$ with a capacity factor $f = \frac{s}{S}$: generating $W_i = f \sum_{j=1}^{N} B w_{i,j}, 1 \leq i \leq M$, and $P = (1 - f) \sum_{j=1}^{N} B p_j$. As f increases, the constraints become less strict, and instances turn from unsatisfiable to satisfiable. To avoid an excess of trivial instances on the lower and higher ends of f, we normalize f to be within $0.2 \leq f \leq 0.8$. One instance set is generated for each $B = 2^w - 1, 4 \leq w \leq 16$, with parameters $N = 15$, $M = 100$, $S = 100$, $C = 3$.

In the SAT encoding of MBKP each integer variable x_i is binary encoded in the least number of bits required to represent its full domain. Each linear constraint of the form $\sum_{j=1}^{N} w_{i,j} x_j \leq W_i$ is encoded by first constructing $y_j = w_{i,j} \times x_j$ using the SCM method, and then computing $y_0 = \sum_{j=1}^{N} y_j$ by repeated use of ripple-carry adders, and finally constraining $y_0 \leq W_i$ using a lexicographic constraint.

To provide another control method, we encode the benchmarks using PicatSAT (version 3.5) [18], a state-of-the-art SAT encoder that uses the binary encoding. The `picat` encodes SCM similar to `base`, but with additional ad-hoc optimizations [19]. Note that for these problems with large coefficients and domain sizes alternate encoding approaches, e.g., domain or order encodings are non-competitive.

All encodings produced in our experiments are solved using the same binary of the SAT solver CaDiCal (version 1.9.1) [4] with default parameters, 8 GB memory limit and a time limit of 180 s. A PAR2 penalty applies to timeouts.

In Fig. 2, we compare the encoding sizes of the MBKP, similar to the results from Sect. 4.1. `picat` matches `min-a` in the number of variables. However, for clauses and especially for literals, larger bit widths result in larger encodings,

closer to `base`. For unknown reasons, `picat` shows a temporary spike in clauses and literals at $w = 5$ (for some constraints, `picat` also relies on Espresso; which it perhaps uses in this case).

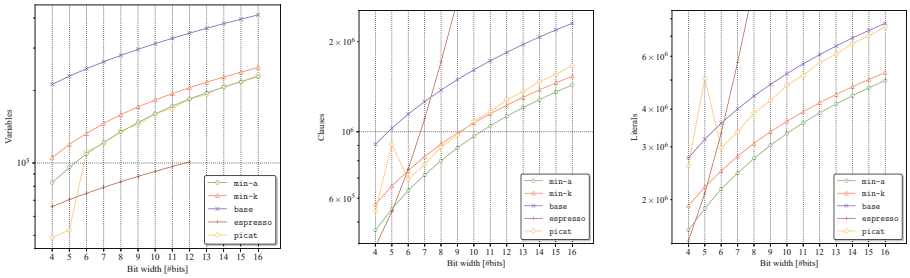

Fig. 2. Encoding size of MBKP using different SCM approaches

Comparing solve times in Fig. 3, we observe that `espresso` performs best up to a bit width of 10, after which its performance drops off drastically. Even if pre-computing larger input bit widths would be tractable for `espresso`, we can extrapolate that an encoding without auxiliary variables loses its effectiveness. In contrast, `min-k` and `min-a` remain relatively constant, even though the instances grow exponentially in size. Furthermore, `min-a` performs better than `min-k`, which shows the benefit of minimizing adders over ripple carry adders, even if some recipes have sub optimal length (such as in Example 1). Finally, `picat` performs much better than `base` for smaller bit widths but similar for larger bit widths, suggesting that `picat` is able to apply certain encoding optimizations in the former case.

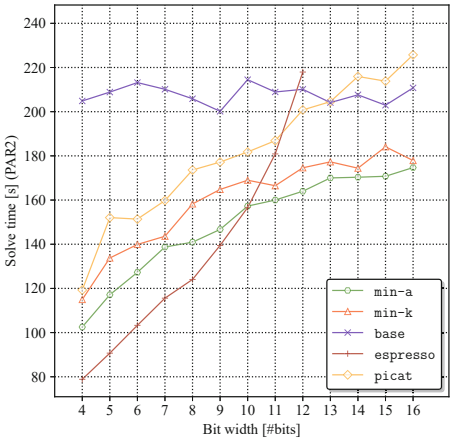

Fig. 3. Solve time comparison for MBKP

5 Conclusion and Future Work

In conclusion, we have shown how to tackle and apply SCM specifically for SAT in order to encode linear constraints. Encoding linear constraints is a key challenge for SAT-based solvers [1]. To our knowledge, this is the first work that adapts SCM circuits to encode linear constraints for SAT. Since ripple carry adders are not uniformly encoded in the presence of partially fixed inputs, this gives us the opportunity to develop different optimal circuits compared to the traditional SCM problem. In the experimental evaluation, we have seen how this approach significantly improves both the encoding size as well as the solver's performance. Finally, we have made the pre-computed SCM encodings freely available.

In future work, we aim to extend our approach by applying MCM to SAT, which – unlike SCM – cannot be comprehensively pre-computed. Instead, we would first have to collect the target constants (e.g., through common sub-expression elimination [14]) from a given instance and then dynamically solve the MCM problem at encode time. This poses an interesting trade-off between encode and solve time, but has the potential for additional improvements.

Acknowledgements. This research was partially funded by the Australian Government through the Australian Research Council Industrial Transformation Training Centre in Optimisation Technologies, Integrated Methodologies, and Applications (OPTIMA), Project ID IC200100009.

References

1. Abío, I., Mayer-Eichberger, V., Stuckey, P.J.: Encoding Linear Constraints into SAT. CoRR **abs/2005.02073** (2020). https://arxiv.org/abs/2005.02073
2. Aksoy, L., Flores, P.F., Monteiro, J.: Exact and approximate algorithms for the filter design optimization problem. IEEE Trans. Signal Process. **63**(1), 142–154 (2015). https://doi.org/10.1109/TSP.2014.2366713
3. Avizienis, A.: Signed-digit number representations for fast parallel arithmetic. IRE Trans. Electron. Comput. **EC-10**(3), 389–400 (1961). https://doi.org/10.1109/TEC.1961.5219227
4. Biere, A., Fazekas, K., Fleury, M., Heisinger, M.: CaDiCaL, Kissat, Paracooba, Plingeling and Treengeling Entering the SAT Competition 2020. In: Balyo, T., Froleyks, N., Heule, M., Iser, M., Järvisalo, M., Suda, M. (eds.) Proc. of SAT Competition 2020 – Solver and Benchmark Descriptions. Department of Computer Science Report Series B, vol. B-2020-1, pp. 51–53. University of Helsinki (2020)
5. Biere, A., Heule, M., van Maaren, H., Walsh, T. (eds.): Handbook of Satisfiability, 2 edn. IOS Press (2021). google-Books-ID: dUAvEAAAQBAJ
6. Brayton, R.K., Hachtel, G.D., McMullen, C.T., Sangiovanni-Vincentelli, A.L.: Logic minimization algorithms for VLSI synthesis. In: The Kluwer International Series in Engineering and Computer Science, vol. 2. Springer, Heidelberg (1984). https://doi.org/10.1007/978-1-4613-2821-6
7. Cappello, P., Steiglitz, K.: Some complexity issues in digital signal processing. IEEE Trans. Acoust. Speech Signal Process. **32**(5), 1037–1041 (1984)

8. Dekker, J.J., Bierlee, H.: Pindakaas: CPAIOR-24 (2024). https://doi.org/10.5281/zenodo.10851856
9. Gustafsson, O.: Towards optimal multiple constant multiplication: a hypergraph approach. In: 42nd Asilomar Conference on Signals, Systems and Computers, ACSSC 2008, Pacific Grove, CA, USA, 26–29 October 2008, pp. 1805–1809. IEEE (2008). https://doi.org/10.1109/ACSSC.2008.5074738
10. Han, B., Leblet, J., Simon, G.: Hard multidimensional multiple choice knapsack problems, an Empirical Study. Comput. Oper. Res. **37**(1), 172–181 (2010). https://doi.org/10.1016/j.cor.2009.04.006
11. Kumm, M.: Optimal constant multiplication using integer linear programming. IEEE Trans. Circuits Syst. II Express Briefs **65-II**(5), 567–571 (2018).https://doi.org/10.1109/TCSII.2018.2823780
12. Ma, S., Ampadu, P.: Optimal SAT-based minimum adder synthesis of linear transformations. In: Lee, H., Geiger, R.L. (eds.) 62nd IEEE International Midwest Symposium on Circuits and Systems, MWSCAS 2019, Dallas, TX, USA, 4–7 August 2019, pp. 335–338. IEEE (2019). https://doi.org/10.1109/MWSCAS.2019.8885033
13. Nethercote, N., Stuckey, P., Becket, R., Brand, S., Duck, G., Tack, G.: MiniZinc: towards a standard CP modelling language. In: Bessiere, C. (ed.) CP 2007. LNCS, vol. 4741, pp. 529–543. Springer, Heidelberg (2007). https://doi.org/10.1007/978-3-540-74970-7_38
14. Nightingale, P., Spracklen, P., Miguel, I.: Automatically improving SAT encoding of constraint problems through common subexpression elimination in savile row. In: Pesant, G. (ed.) CP 2015. LNCS, vol. 9255, pp. 330–340. Springer, Heidelberg (2015). https://doi.org/10.1007/978-3-319-23219-5_23
15. Rossi, F., van Beek, P., Walsh, T. (eds.): Handbook of Constraint Programming, Foundations of Artificial Intelligence, vol. 2. Elsevier (2006). https://www.sciencedirect.com/science/bookseries/15746526/2
16. Soos, M., Nohl, K., Castelluccia, C.: Extending SAT solvers to cryptographic problems. In: Kullmann, O. (ed.) SAT 2009. LNCS, vol. 5584, pp. 244–257. Springer, Heidelberg (2009). https://doi.org/10.1007/978-3-642-02777-2_24
17. Warners, J.P.: A linear-time transformation of linear inequalities into conjunctive normal form. Inf. Process. Lett. **68**(2), 63–69 (1998). https://doi.org/10.1016/S0020-0190(98)00144-6
18. Zhou, N., Kjellerstrand, H.: The Picat-SAT Compiler. In: Gavanelli, M., Reppy, J.H. (eds.) PADL 2016. LNCS, vol. 9585, pp. 48–62. Springer, Heidelberg (2016). https://doi.org/10.1007/978-3-319-28228-2_4
19. Zhou, N., Kjellerstrand, H.: Optimizing SAT encodings for arithmetic constraints. In: Beck, J.C. (ed.) CP 2017. LNCS, vol. 10416, pp. 671–686. Springer, Heidelberg (2017). https://doi.org/10.1007/978-3-319-66158-2_43