XP^2: A New Compact Representation for Manipulating Arithmetic Circuits

Ajay K. Verma
AjayKumar.Verma@epfl.ch

Philip Brisk
Philip.Brisk@epfl.ch

Paolo Ienne
Paolo.Ienne@epfl.ch

Ecole Polytechnique Federale de Lausanne (EPFL)
School of Computer and Communication Sciences
CH-1015 Lausanne, Switzerland

ABSTRACT

The complexity and performance of most algorithms used in logic synthesis depend greatly on the representation of input Boolean expression. In other words, finding an appropriate representation for Boolean expressions is a key problem in logic synthesis. This is the reason why plethora of representations such as *Sum-of-Product* (SOP), *Product-of-Sum* (POS), *Binary Decision Diagram* (BDD), *Factored Form* [6], etc. have been suggested in the past. These representations offer a trade-off between the compactness and the complexity of manipulation algorithms. However, most of these representations are targeted towards common logic functions, and hence they perform badly on arithmetic circuits. In this paper we propose a new representation named XP^2 for XOR-dominated circuits, which offers great reduction in expression size only at the cost of slight increased complexity of manipulation algorithms. We also show that in some cases our representation is as compact as the manually crafted one.

1. INTRODUCTION

In logic synthesis multi-level optimisations play an important role in optimising delay and area of a circuit. However, the most important and difficult step is to convert a Boolean expression into a multiple-level logic network. Since most algorithms for optimising multi-level logic networks depend on the representation of input expressions, the choice of an input representation of Boolean expression plays a vital role in multi-level optimisation.

There are various ways to represent a Boolean function. The most commonly used representation for multi-level optimisation is the SOP form due to availability of efficient algorithms for various procedures used in multi-level optimisations. However, some expressions are exponentially large when represented in SOP form, which makes even the efficient algorithms impractical for these expressions. In order to overcome these limitations, other representations of Boolean expressions have been presented in the past. The notable examples of such representations are factored form introduced by Brayton *et al.* [6], BDD representations etc. Although these representations are significantly compact compared to SOP form, the logic optimisations on these forms lead to additional complications.

Apart from these limitations one common weakness in all these representations is that they are targeted towards ordinary logic functions, and for XOR-dominated expressions they show exponential blow up in size. This makes these representations not so useful for arithmetic circuits. Verma *et al.* [15] suggested the use of *Reed-Muller* form (XOR-of-Product) form to represent arithmetic circuits. Although Reed-Muller form has certain advantages over SOP form such as canonicity, better factorisation algorithms etc., it still can be extremely large for certain arithmetic circuits.

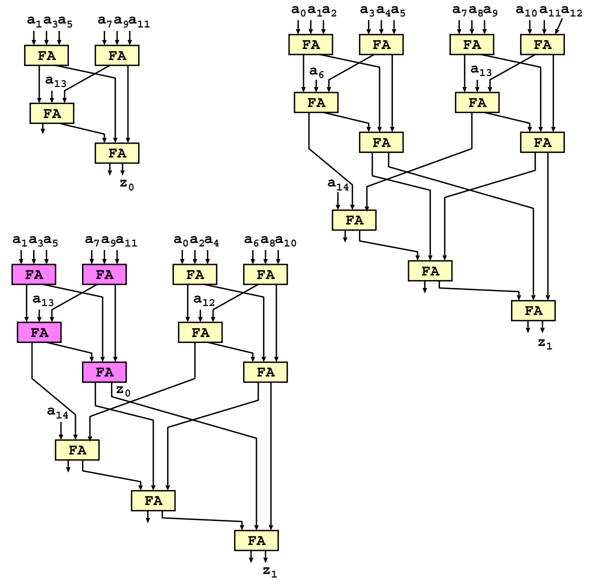

Figure 1: Although a hierarchical representation is very compact, it is not suitable for circuit manipulation algorithms such as common subexpression elimination.

The most compact representation for a Boolean expression is the circuit representation which uses all sort of gates. For the sake of brevity we will call this representation *Hierarchical Representation* now onwards. The only problem with the hierarchical representation is the high complexity of various procedures of multi-level optimisations (e.g., common subexpression elimination) on input expressions of this form. Let us consider the example shown in Fig. 1. The first circuit corresponds to the majority of bits a_1, a_3, ..., a_{13}, and the second circuit corresponds to the majority of bits a_0, a_1, a_2, ..., a_{14}. The two circuits are implemented using full adders (FA), which in fact is the most compact representation of the two expressions. When we synthesise the two circuits separately with emphasis on optimising hardware area, we get the circuits having delay and hardware area 0.35 ns (705 μm^2) and 0.74 ns (2200 μm^2) correspondingly. However, when we synthesise both circuits together, we get an implementation with delay 0.75 ns and cell area 2815 μm^2. On the other hand, if we have a circuit similar to third circuit in Fig. 1 where there is a significant sharing between the computation of two output bits, we get an implementation with delay 0.75 ns and cell area 2275 μm^2.

Instead of the above representation of majority bit, if we give the flat SOP form representation for logic synthesis, we get circuits with delays and cell area much higher than the aforementioned values (delay = 0.79 ns, cell area = 16379.2 μm^2). This is because the hierarchical representation, which uses FA, is very compact but does not have efficient algorithms to find common subexpressions. On the contrary, the flat SOP form has efficient algorithms to compute the common subexpressions, but the input expressions are so large that these algorithms fail to produce meaningful results.

Hence, one needs to find a representation of Boolean expressions corresponding to arithmetic circuits which is compact and has efficient algorithms for various procedures used in multi-level optimisation. In this paper we propose such a representation named XP^2, and show that it is more compact than the other representations mentioned above. We also give efficient algorithms to minimise an expression in XP^2 form, and to compute the common subexpression (which is the most important procedure in multi-level optimisation) of two expressions given in XP^2 form.

The rest of the paper is organised as follows: In the next section we discuss the related work on Boolean expression representation. Section 3 discusses the XP^2 form as well shows its effectiveness over other representations. The following section discusses some preliminary results from abstract algebra which are used in Section 5 and Section 6 to give algorithms for minimising an expression in XP^2 form, and to find the common subexpression of two expressions. Finally we discuss results of our experiments in Section 7 followed by conclusions and future work.

2. STATE OF THE ART

The most common representations for Boolean expressions are SOP/POS form [10], for which there are efficient algorithms to minimise the expression [7] as well as for various procedures of multi-level optimisations [5]. Brayton suggested a new representation named factored form [6] and presented various heuristics for minimisation of this form. Although this form is more compact than SOP and POS form, we have shown that even this form is not appropriate for XOR-dominated expressions.

It was suggested by several researchers in their work [14, 15] that Reed-Muller and Generalised Reed-Muller form (GRM) are more appropriate for XOR-dominated circuits and heuristics [13, 4] are presented to minimise the Boolean expression in Reed-Muller form. Although Reed-Muller form is more compact than SOP and POS form for many arithmetic circuits, there are some circuits such as parallel counter etc. which are exponentially large even in Reed-Muller form. Another problem with Reed-Muller is that if there is some control oriented logic involved at the periphery of arithmetic circuit (which is most often the case), then also the size of the whole expression in Reed-Muller form is exponentially large.

Lee presented a new representation for Boolean expression in his work [12] which is known as *Binary Decision Diagram* (BDD). This representation was used by several researchers [1, 8] as a new data structure for Boolean functions and the minimisation algorithm for BDD's have been developed; however, Bryant showed in [9] that even BDD's are not appropriate for arithmetic circuits such as multipliers. In order to overcome the problem of huge size in BDD representation Yang *et al.* [16] suggested a new method to partition the BDD into several BDD's. Although this method reduces the size of larger BDD's, the efficiency of multi-level optimisation algorithms such as common subexpression elimination depends on the partition.

Very recently Bernasconi *et al.* [3] suggested a new representation called 2-SPP for Boolean expressions and proposed heuristic methods to minimise the expression in this form. 2-SPP form is similar to SOP form with the exception that the product terms correspond to AND of literals and XOR of two literals. Since in general arithmetic circuits are k-SPP, this representation is useful only if the given expression has XOR's only at the peripherals.

A similar representation to XP^2 has been proposed by Ishikawa *et al.* [11], and also some minimisation algorithms have been presented based on pseudo-cubes. However, they do not present any algorithm for common procedures in multi-level optimisation such as common subexpression elimination based on their representation.

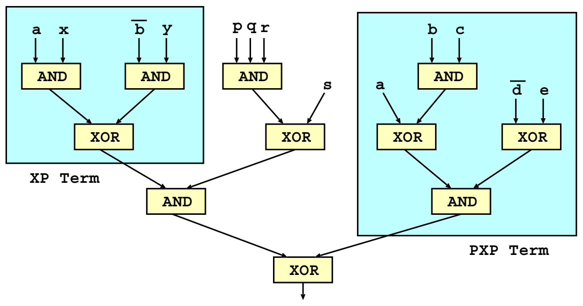

Figure 2: An Example illustrating XP^2 representation of an expression, and the corresponding PXP and XP terms.

3. NEW REPRESENTATION AND ITS SUPERIORITY

In this section, we formally prove the weakness of some of the previous representations, and then propose a new representation which we name XP^2. We also present some results which show the superiority of the new representation over some other representations such as factored form. However, before that let us formally define the factored form which we also name as SP^k.

DEFINITION 1. *For $k = 1$, SP^k corresponds to SOP representation, and for any other k, SP^k corresponds to sum of products of $\mathsf{SP}^{\mathsf{k}-1}$ expressions.*

For example an expression in SP^2 looks like $A_1A_2\cdots A_p + B_1B_2\cdots B_q + C_1C_2\cdots C_r + \cdots$, where each of the A_i's, B_j's, C_k's is a Boolean expressions represented in SOP form. The next theorem shows that the SP^2 representation is not appropriate for XOR-dominated expression.

THEOREM 1. *The SP^k representation of an expression corresponding to the XOR of n variables is exponentially large, for any constant k.*

As mentioned earlier that Reed-Muller form is more appropriate for arithmetic circuits, however, according to second theorem Reed-Muller form explodes if there is some control oriented logic involved in the underlying arithmetic circuit.

THEOREM 2. *The Reed-Muller expansion of the Boolean expression corresponding to the OR of n variables is exponentially large.*

Next we propose a new representation XP^2 for arithmetic circuits, which is similar to SP^2, except that here we use XOR instead of OR. More formally we can define XP^2 like this:

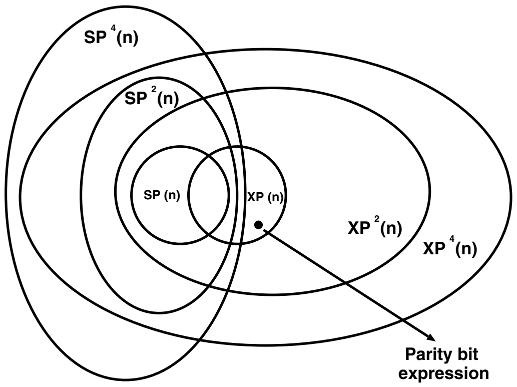

Figure 3: Relative compactness of various XP^k and SP^k representation. Note that the expression for parity bit does not have a compact representation in any SP^k.

DEFINITION 2. *An expression in* XP^2 *form is* XOR *of product terms, where each product term is a product of Generalised Reed-Muller expressions.*

Fig. 2 shows how an example of XP^2 expression. Each XP^2 expression can be represented as a tree of depth 4, where in any path from leaves to root AND and XOR occur alternatively, and the leaf node corresponds to a variable or its complement. The operands of the root XOR node in the tree are called PXP terms, and each operand of the AND node in PXP a term is called as an XP term.

Clearly XP^2 form is more compact than Reed-Muller form, i.e., the representation size of any Boolean expression in Reed-Muller form is at least as large as in XP^2 form. we also claim that XP^2 form is at least as compact as SOP and POS form.

THEOREM 3. *If the SOP (POS) representation of a Boolean expression has size k, then the* XP^2 *representation of the same expression cannot be larger than* $O(k)$.

PROOF. Let us assume that SOP form of a Boolean expression A is given by $p_1 + p_2 + \cdots + p_m$, where p_i's are the product terms. Now we can write A like this:

$$\begin{aligned} A &= p_1 + p_2 + \cdots + p_m \\ &= \overline{\bar{p_1}\bar{p_2}\cdots\bar{p_m}} \\ &= 1 \oplus (\bar{p_1}\bar{p_2}\cdots\bar{p_m}) \\ &= 1 \oplus ((1 \oplus p_1)(1 \oplus p_2)\cdots(1 \oplus p_m)). \end{aligned}$$

Note that each of the $(1 \oplus p_i)$ is in Reed-Muller form, hence the final expression is in XP^2 form, and has the same literal count as the original expression in SOP form.

Similarly assume that the expression A is represented in POS form like $s_1 s_2 \cdots s_n$, where each of the s_i is a sum term. Note that each s_i can be represented in Generalised Reed-muller form efficiently in the following way:

$$\begin{aligned} s_i &= x_1 + x_2 + \cdots + x_r \\ &= 1 \oplus (\bar{x_1}\bar{x_2}\cdots\bar{x_r}). \end{aligned}$$

This means the product of s_i will correspond to product of Generalised Reed-Muller expressions. In other words, the literal count in XP^2 representation of A is same as that in POS representation. $\square$

In a similar way we can also define XP^k for any constant k. It can be noticed easily that as we increase k, the representation will become increasingly compact. The relative compactness of these representations in shown in Fig. 3, where region $\mathsf{XP}^k(n)$ ($\mathsf{SP}^k(n)$) denotes the set of Boolean expression whose representation size in XP^k (SP^k) is bounded by n. We can see that even $\mathsf{XP}(n)$ is not fully contained in any of the $\mathsf{SP}^k(n)$. This is because the expression corresponding to parity of n-bits cannot be expressed in polynomial size in SP^k for any constant k. However, $\mathsf{SP}^k(n)$ is always contained in $\mathsf{XP}^{2k}(n)$.

Although increasing the number of levels offers compactness, it will be significantly difficult to use the various procedures of multi-level optimisation on XP^k representation. In this paper we consider only the XP^2 representation. In our experiments, we observe that there is a huge reduction in the representation size of a Boolean function when we go from XP to XP^2. In fact the representation size in XP^2 is very close to representation size in XP^k (with unbounded k) as depicted by Fig. 6. In general, increasing k beyond two yields diminishing returns in terms of the compactness of the resulting circuit.

Apart from the above justifications in order to accept XP^2 as a new representation for Boolean expressions we must introduce the algorithms to find the minimal representation in XP^2 form of an expression, and to find the common subexpression of two expressions represented in XP^2 form. We discuss these algorithms in Section 5 and Section 6 However, before discussing these algorithms we introduce certain concepts from abstract algebra which will be used in the two algorithms.

4. RESULTS FROM ABSTRACT ALGEBRA

In this section we present few results from ring algebra, which can be used effectively in Boolean factorisation of an expression. We use the notation of null space introduced by Verma *et al.* [15]. The null space $N(P)$ of an expression P is defined as the set of all Boolean expressions X, which cannot be true simultaneously with P (e.g., $ab \in N(a \oplus b)$ because $ab(a \oplus b) = 0$). More formally:

$$N(P) = X \mid PX = 0.$$

The null space of an expression is closed under the operations XOR and AND, and forms a ring. We can also define the operations on null spaces like this:

$$\begin{aligned} N(P) \oplus N(Q) &= x \oplus y \mid x \in N(P), y \in N(Q), \\ N(P) \cdot N(Q) &= xy \mid x \in N(P), y \in N(Q), \\ \overline{N(P)} &= x \mid \overline{x} \in N(P) \end{aligned}$$

4.1 Efficient Factorisation via Null Spaces

The advantage of null space comes from the following two theorems, which help in factoring a Boolean expression, which is irreducible in normal algebra. In other words, these theorems make our factorisation algorithm closer to Boolean factorisation without increasing the complexity significantly. The first of the two theorems has already been mentioned in [15].

THEOREM 4. *An expression in the form* $PQ \oplus RS$ *can be factored like* $(P \oplus R)T$ *for some* T, *if* $Q \oplus S \in N(P) \oplus N(R)$.

As a corollary, if we put $Q = R = 1$, we get that expression P is divisible by S if $\overline{S} \in N(P)$. This method of division is advantageous over the Boolean division, because $N(P) \oplus N(R)$ is a ring, and hence the membership of $Q \oplus S$ in it can be checked easily using *Ideal Membership Problem* [2].

THEOREM 5. *An expression in the form $PQ \oplus RS$ can be factored like $(P \oplus R)T$ for some T, if there exist expressions $x \in N(P) \cap \overline{N(R)}$ and $y \in \overline{N(P)} \cap N(R)$.*

Let us consider a few examples which illustrate the use of the two theorems.

$$\begin{aligned}
&z = (a \oplus b)(cd \oplus e) \oplus (c \oplus d)(ab \oplus e),\\
&P = a \oplus b, \quad Q = cd \oplus e, \quad R = c \oplus d, \quad S = ab \oplus e,\\
&Q \oplus S = cd \oplus ab,\\
&cd \in N(c \oplus d), \quad ab \in N(a \oplus b),\\
&\text{i.e., } Q \oplus S \in N(P) \oplus N(R).
\end{aligned}$$

Hence, $z = (a \oplus b \oplus c \oplus d)(ab \oplus cd \oplus e)$.

In the above example we use the Theorem 4 to factor the given expression. In order to see the use of the second Theorem consider the following expression.

$$z = (k_2 \oplus g_2)k_1k_0 \oplus p_2g_1k_0 \oplus p_2p_1g_0,$$

where p_i, g_i and k_i denote the *Propagate*, *Generate* and *Kill* signals at the i^{th} bit position in the addition of two integers. This expression can be factored according to the second theorem like this.

$$\begin{aligned}
g_1 \oplus p_1g_0 &\in N(k_1k_0) \cap \overline{N(g_1k_0)} \cap \overline{N(p_1g_0)},\\
k_1 \oplus p_1k_0 &\in N(g_1k_0) \cap N(p_1g_0) \cap \overline{N(k_1k_0)}.\\
\text{Hence, } z &= (k_1k_0 \oplus g_1k_0 \oplus p_1g_0)((k_2 \oplus g_2)K \oplus p_2G),\\
&\text{where } K = k_1 \oplus p_1k_0, \quad G = g_1 \oplus p_1g_0.
\end{aligned}$$

4.2 Computation of Null Spaces

From the previous subsection it is clear that the efficacy of division depends on how efficiently we compute the null spaces of Boolean expressions. In the earlier work null spaces of Boolean expressions are computed like this:

$$\begin{aligned}
N(x) &= \{\bar{x}\} \text{ if } x \text{ is a variable},\\
N(P \oplus Q) &= rC(N(P) \cdot N(Q))\\
N(PQ) &= rC(N(P) \cup N(Q)).
\end{aligned}$$

Here $rC()$ corresponds to ring closure, and returns the smallest ring containing the given elements. We have observed that the above computation of null spaces is an underestimation of null space, i.e., null space of expression may contain more than the above specified values. Here we compute the null space more accurately. Along with the above values we also insert some new values in $N(P \oplus Q)$ and $N(PQ)$, which are based on the following theorem.

THEOREM 6. *Assume that the* Least Common Multiple *(lcm) of P and Q is R.*
(1) *If the following is true:*

$$\begin{aligned}
&(\exists x, y)Px = Qy = R, \textit{ and}\\
&\quad (x \oplus y) \in N(P) \oplus N(Q)\\
&\quad \textit{i.e., } x \oplus y = p_1 \oplus q_1, \quad p_1 \in N(P), \quad q_1 \in N(Q),
\end{aligned}$$

then $(p_1 + x) \in N(P \oplus Q)$.
(2) *If* r_1, r_2, ..., r_k *are divisors of R, then the* XOR *of any even number of* r_i*'s will be in* $N(PQ)$ *as well as in* $N(R)$.

The computational complexity of algebraic factorisation is less than that of factorisation based on null spaces; therefore, we use the former to compute the lcm R and its divisors. Let us consider an example to illustrate the efficacy of above theorem. According to the first part of the theorem

$$\begin{aligned}
abc(d) &= abd(c),\\
c \oplus d = (c \oplus 1) \oplus (d \oplus 1) &\in N(abc) \oplus N(abd), \text{ i.e.,}\\
(c \oplus d \oplus 1) &\in N(abc \oplus abd).
\end{aligned}$$

Since $(a \oplus b) \in N(ab) \subseteq N(ab(c \oplus d))$, we can infer that

$$(a \oplus b \oplus c \oplus d \oplus 1) \in N(abc \oplus abd).$$

Similarly, we can deduce that $(a \oplus b \oplus c \oplus d \oplus 1) \in N(acd \oplus bcd)$. Combining this with the earlier result we get that

$$(a \oplus b \oplus c \oplus d \oplus 1) \in N(abc \oplus abd \oplus acd \oplus bcd).$$

Now according to the corollary of Theorem 4 we can say that $(a \oplus b \oplus c \oplus d)$ is a factor of $(abc \oplus abd \oplus acd \oplus bcd)$.

5. MINIMISATION OF XP^2 REPRESENTATION

In this section we discuss the algorithm to minimise the representation of an expression in XP^2 form. We reduce the size of the expression by iteratively applying two procedures called split and merge on the input expression till they improve the size of the expression. We discuss the two procedures in detail in the following subsections.

5.1 Minimising the Representation by Splitting a Product Term

In the split procedure we choose an XP term and factorise it. The factorisation is done in two steps: The first step which is fast, but less effective is a modified version of kernel extraction algorithm [10]. On other other hand, the second step is slow and more effective. It does the factorisation using the null spaces as described in Section 4.

The modified kernel extraction is same as the kernel extraction with the exception that we might add a product term twice in the expression, if it reduces the size of the XP term by increasing the coefficient of a kernel. To understand it in a better way consider the following example

$$A = abcpq \oplus abcrs \oplus dpq \oplus drs \oplus abcx.$$

After kernel extraction

$$A = (abc \oplus d)(pq \oplus rs) \oplus abcx.$$

Now if we add dx twice in this expression (which will be equivalent to add 0), the coefficient of the kernel $abc \oplus d$ will increase, and the overall literal count in the expression will decrease. The modified expression will look like:

$$A = (abc \oplus d)(pq \oplus rs \oplus x) \oplus dx.$$

Once we factorise the expression, it no longer remains in Generalised Reed-Muller form, i.e., the overall expression will no longer be in XP^2 form. Hence, we need to split the successor PXP expression of this expression into multiple PXP's. Here we show an example of splitting caused by factoring the above mentioned expression.

$$\begin{aligned}
B &= (m \oplus n)(abcpq \oplus abcrs \oplus dpq \oplus drs \oplus abcx),\\
B &= (m \oplus n)((abc \oplus d)(pq \oplus rs \oplus x) \oplus dx),\\
B &= (m \oplus n)(abc \oplus d)(pq \oplus rs \oplus x) \oplus (m \oplus n)(dx).
\end{aligned}$$

Since both $(m \oplus n)(abc \oplus d)(pq \oplus rs \oplus x)$ and $(m \oplus n)(dx)$ are PXP terms, the overall expression is still in XP^2 form. However, the literal count of the expression reduced from 22 to 15. In some cases, this splitting might increase the literal count. Hence, we factor an XP term only if the splitting caused by its factoring reduces the literal count of the whole expression. Also we choose the XP term to factor first, which reduces the literal count by most.

5.2 Minimising the Representation by Merging Product Terms

This procedure is exactly opposite to the previous procedure and merges two PXP terms into a single PXP term. The basic idea is that if there are two PXP terms AB and AC, where A, B and C are products of XP terms, then they can be merged into a single PXP term $A(B \oplus C)$, where $(B \oplus C)$ is expanded to be in XP form. This means the first job is to find out the common product terms A between the two PXP's. We find these common terms in the three steps which are following:

The first step finds all algebraically common product terms between the two PXP's, e.g., in $(a \oplus b)(c \oplus d)$ and $(a \oplus b)(e \oplus f)$ the algebraically common product term is $(a \oplus b)$.

In the second step the common product terms are found by using properties of Boolean algebra. For this purpose we use the divisibility test using null spaces as described in the corollary of Section 4. The common term between $(a \oplus b \oplus c \oplus d)$ and $(abc \oplus abd \oplus acd \oplus bcd)$ can be found by this method, as we can deduce from the null space information that latter is a multiple of the first.

In the third step we check whether in the merged PXP some new product terms can be found. For this purpose we use Theorem 4 and Theorem 5 to factor an expression of the form $PQ \oplus RS$. To see it more clearly let us consider this example, where the two PXP's are A_1 and A_2.

$$A_1 = (p \oplus q)(a \oplus b \oplus rs),$$
$$A_2 = (r \oplus s)(a \oplus b \oplus pq).$$

Since

$$(a \oplus b \oplus rs) \oplus (a \oplus b \oplus pq) = pq \oplus rs \text{ and}$$
$$(pq \oplus rs) \in N(p \oplus q) \oplus N(r \oplus s),$$
$$A_1 \oplus A_2 = (p \oplus q \oplus r \oplus s)(a \oplus b \oplus pq \oplus rs).$$

After finding all the common product terms we just expand the remaining product terms of the two PXP's into XP form and add them together. Sometimes this expansion might cause increment in literal count of the whole expression. Hence, we should merge two PXP's only if they reduce the literal count in the overall expression

Also note that it is possible that the new merged XP term can be factored according to split procedure which might cause this newly merged PXP term to split again in multiple PXP's causing a reduction in literal count. As an example suppose the two PXP's to be merged are

$$A_1 = (p \oplus q)(acy \oplus ady \oplus x),$$
$$A_2 = (p \oplus q)(c \oplus d)bef.$$

Merge

$$A_{12} = (p \oplus q)(acy \oplus ady \oplus bcef \oplus bdef \oplus x).$$

Split

$$A_{12} = (p \oplus q)((ay \oplus bef)(c \oplus d) \oplus x).$$
$$A_1' = (p \oplus q)(ay \oplus bef)(c \oplus d),$$
$$A_2' = (p \oplus q)x.$$

Since the literal count of the merged representation A_{12} is 17

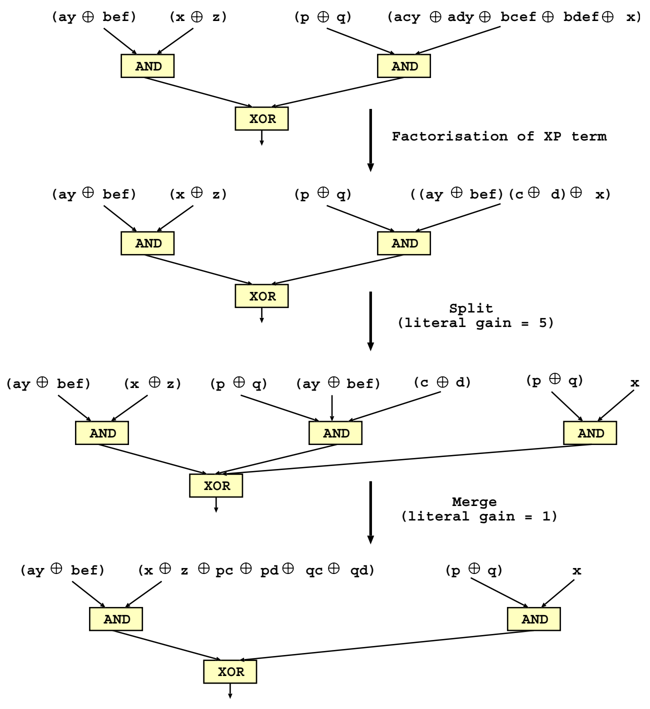

Figure 4: Illustration of the XP^2 minimisation algorithm via an example.

compared to the original literal count 16, this merging should not be useful. However, the merged XP can be factored again which causes the splitting of merged $PXPX$ resulting in a final literal count 12, which is better than the original. This suggests that in the merging step we should also check whether the merged XP can be factored to result in a new beneficial splitting although stand alone merging was not beneficial. Fig. 4 shows an example of minimising the expression by applying both procedures split and merge.

Note that both the procedures split and merge always reduce the representation size. This might lead us to a local optimal solution. Hence, with some small probability one must also accept literal count increasing representations. However, in the current work we allow only literal count decreasing transformations and consider the suggested approach as a future work.

6. NODE SHARING AMONG XP^2 EXPRESSIONS

In this section we discuss how to compute the common subexpression of two expressions represented in XP^2 form. In order to find out the common subexpressions of two expressions A and B, we define a new expression $C = \lambda A \oplus \mu B$, where λ and μ are two new variables. Since computing the common subexpression of two expressions is equivalent to represent both the expressions using minimum number of nodes in XP^2 representation tree (i.e., to share the largest portion of the XP^2 trees of two expressions). This informally suggests that computing the common subexpression of A and B is equivalent to minimise the XP^2 representation of $\lambda A \oplus \mu B$. Note that Since both A and B are in XP^2 form, $\lambda A \oplus \mu B$ can also be represented in XP^2 form by adding a product term λ in each PXP of A, and a product term μ in each PXP of B.

This means that to find the common subexpression of the two expressions we can use the same minimisation algorithm mentioned in the previous section. Note that after minimising $\lambda A \oplus \mu B$, each of its PXP must have a product term λ, μ or $\lambda X + \mu Y$, The PXP's which have a product term of the third form have the common subexpressions between the two expressions. In fact in these PXP's except the product term $\lambda X + \mu Y$ all other product terms are common between the two expressions.

Also note that since our task was not to compute $\lambda A \oplus \mu B$, but to represent the two expressions A and B together, the XP term $\lambda X + \mu Y$ is a pseudo XP term. The reason is that λ and μ are inserted by us and if they were not there, it would be just X or Y. In other words X and Y do not have to be in XP form, they can be PXP form. However, we cannot allow some product terms of a PXP in PXP form. For this reason we keep $\lambda X + \mu Y$ in XP form, but in the literal count of this XP, we convert both X and Y into PXP form and then count the number of literals. also we do not count λ and μ in literal count.

After the minimisation we get a tree, and to minimise the number of internal nodes, we convert it into a DAG by eliminating duplicate nodes. This is done first at the PXP level, where common product terms between two PXP's are shared, next we share the nodes in XP level, where overlapping XP's are handled to remove the shared nodes. Finally the leaf level product terms are considered and common subexpressions among them are eliminated.

7. EXPERIMENTAL RESULTS

We have implemented the algorithm for minimising a Boolean expression in XP^2 form as well as to compute the common subexpression of various expressions in Maple. Our tool takes a set of input expression in the XP^2 format, and first minimises the expressions according to literal count, and then reduces the node count in the XP^2 representation by eliminating common subexpressions among them.

In our experiments we supply the Reed-Muller form as inputs. However, some benchmarks such as comparator, CSA, *Leading Zero Anticipator* (LZA) have extremely large Reed-Muller form. For Comparator we take the Generalised Reed-Muller form as input, and for CSA and LZA, we first find the XP^2 representation of the addition of first two inputs and then using it compute the representation of final output.

Table 1 shows the results of our experiments. The first column corresponds to the literal count in the SOP form of the input expressions optimised by Boolean minimisation tool *espresso* [7]. Note that for 12-bit CSA as well as for $16 - bit$ LZA, the expressions in SOP form were so large that *espresso* failed to optimise them. The second and third column correspond to the literal count of input expressions in Reed-Muller and Generalised Reed-Muller form. The next column represents the output of our tool, i.e., the minimised expression in XP^2 form. However, the expressions here do not share any nodes among them. The following column shows the representation of input expressions in XP^2 form by sharing the common nodes in XP^2 tree. Finally, the last column shows the number of nodes in the manual representation of input expression. Note that the manual implementation of each circuit uses all type of logic gates, not just AND/XOR.

We can see that for the first three benchmarks, which are common logic functions, the SOP representation is better than Reed-Muller representation. However, for the latter circuits, which are purely arithmetic circuits, Reed-Muller form is better than SOP. Another aspect to notice is that for some expressions which consists of negated literals, the size of Reed-Muller expression is enormously huge (e.g., comparator, LZD etc.). This is because Reed-Muller form has no NOT gate, which can be overcome in Generalised Reed-Muller form due to the existence of NOT gate.

In all the benchmarks XP^2 representation outperforms SOP and

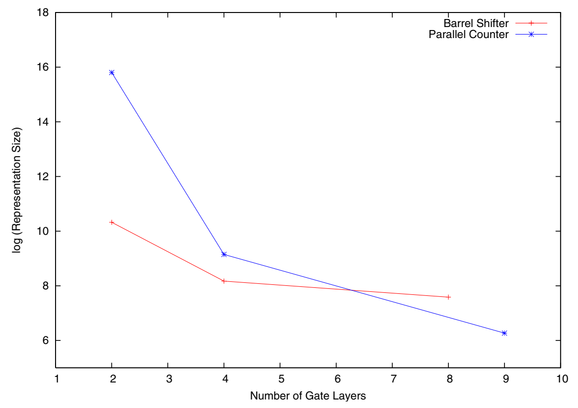

Figure 6: The compactness of representation increases very slowly beyond XP^2 by increasing the gate layers in the circuit, i.e., when we increase the number of layers from 2 to 4, the representation size reduces significantly, however further increase in layers reduces the representation size marginally.

Reed-Muller form representations by a huge factor. This implies that having an extra layer of XOR and AND gates reduce the expression size to great extent. The XP^2 representation where the nodes are shared, is almost one third of the representation when no sharing was allowed. This means that our algorithm for common subexpression elimination is very effective.

In Fig. 5, we compare the representation size of adder and barrel shifter of various bitwidths. We can see that for adder the representation size in SOP/GRM increases exponentially when we increase the bitwidth. However, in XP^2 representation, the representation size grows polynomially. This implies the scalability of our representation. For barrel shifter, the representation size is similar in all three representations. This is because of the non-arithmetic nature of Barrel shifter, i.e., Barrel shifter is not XOR-dominated circuit.

Compared to a manual representation the circuit XP^2 is worse in general. This is because in the manual representation all kind of Boolean gates are used, i.e., the number of different layers of gates from the leaf to root is too many. This makes these representations very difficult to manipulate as two different kind of gates do not commute or associate. However, in the XP^2 representation there are only four different kind of gates from any leaf to root. Also these nodes are XOR and AND, which still satisfy the distributive property, using which they can be manipulated.

The graph in Fig. 6 shows the number of different gates in paths from root to leaf verses the log of representation size. As we can see that the curve has similar structure as that of e^{-x}. This means that increasing the gate layers from 2 (Reed-Muller form) to 4 (XP^2 form) has a great advantage, but beyond that increasing the gate layers do not reduce the size of representation so effectively.

8. CONCLUSIONS

We have shown the need for a new representation for arithmetic circuits, which is not only compact but also has efficient algorithms to manipulate the expressions. The most commonly used representations are shown to be inappropriate and a new representation named XP^2 has been proposed. We also give efficient algorithms to manipulate Boolean expressions represented in XP^2 form. Our experiments show that XP^2 reduces the representation size enormously without complicating the manipulation algorithms. We argue through this new representation that all other representations have fundamental shortcomings, and it only takes the judicious ad-

Benchmark	SOP form	Reed-Muller Form	Generalised Reed-Muller	XP^2 Form (No CSE)	XP^2 Form (CSE)	Manually Designed
16-bit LZD	220	1.81×10^6	332	154	66	64
16-bit LOD	220	602	332	154	66	64
16-bit Barrel Shifter	1280	4752	1280	896	288	192
16-bit Adder	5.24×10^6	9.18×10^5	9.18×10^5	1194	237	89
16-bit Comparator	1.05×10^6	4.81×10^8	1.05×10^6	246	85	63
15 : 4-Parallel Counter	5.19×10^5	5.72×10^4	5.72×10^4	1854	567	77
15-bit Majority	5.15×10^4	5.15×10^4	5.15×10^4	1479	543	71
12-bit CSA	Large	1.24×10^{12}	1.24×10^{12}	3336	250	149
16-bit LZA	Large	Large	Large	42602	3136	152

Table 1: Results of minimisation of XP^2 form and common subexpression elimination.

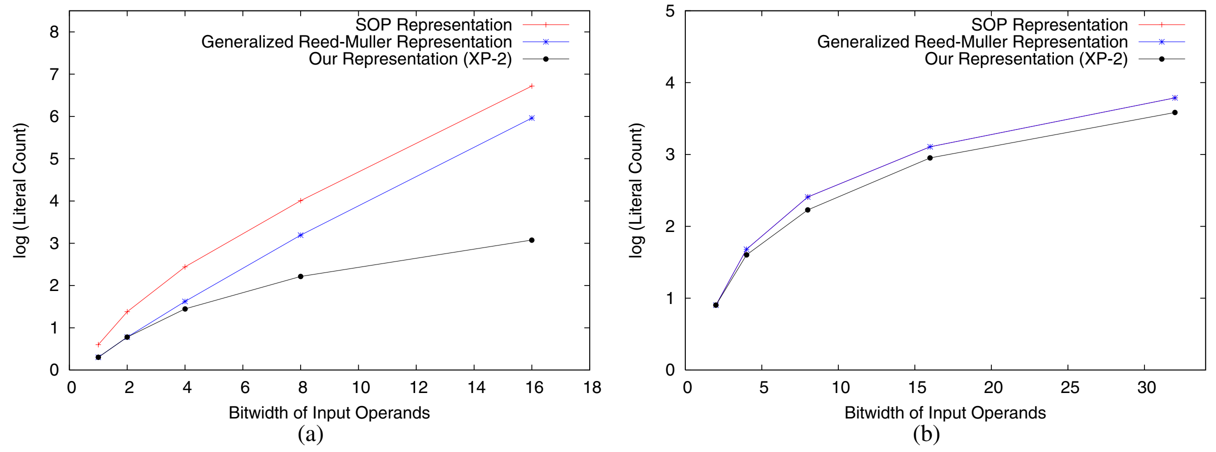

Figure 5: Comparison of representation size for various bitwidth for (a) an adder, (b) a barrel shifter. In case of adder the representation size in SOP and Generalized Reed Muller increases exponenially with the bitwidth of operands, however, in XP^2 the representation size grows polynomially.

dition of two logic layers to the representation to capture efficiently even complex logic functions.

9. REFERENCES

[1] S. B. Akers. Functional testing with binary decision diagrams. In *Eighth Annual Conference Fault-Tolerant Computing*, pages 75–82, 1978.

[2] T. Becker and V. Weispfenning. *Gröbner Bases: A Computational Approach to Commutative Algebra*. Springer, New York, 1993.

[3] A. Bernasconi, V. Ciriani, R. Drechsler, and T. Villa. Efficient minimization of fully testable 2-SPP networks. In *Proceedings of the Design, Automation and Test in Europe Conference and Exhibition*, Munich, Mar. 2006.

[4] D. Brand and T. Sasao. "minimization of and-exor expressions using rewrite rules". *IEEE Transactions on Computers*, 42, May 1993.

[5] R. Brayton and C. McMullen. The decomposition and factorization of boolean expressions. In *Proceedings of the International Symposium on Circuits and Systems*, pages 49–54, Rome, May 1982.

[6] R. K. Brayton. Factoring logic functions. *IBM Journal of Research and Development*, 31(2):187–98, Mar. 1987.

[7] R. K. Brayton, G. D. Hachtel, C. T. McMullen, and A. L. Sangiovanni Vincentelli. *Logic Minimization Algorithms for VLSI Synthesis*. Kluwer Academic, Boston, Mass., 1984.

[8] R. E. Bryant. Graph-based algorithms for boolean function manipulation. *IEEE Transactions on Computers*, 35:677–91, Aug. 1986.

[9] R. E. Bryant. The complexity of VLSI implementations and graph representations of Boolean functions with application to integer multiplication. *IEEE Transactions on Computers*, 40(2):205–13, Feb. 1991.

[10] G. De Micheli. *Synthesis and Optimization of Digital Circuits*. McGraw-Hill, New York, 1994.

[11] R. Ishikawa, T. Hirayama, G. Koda, and K. Shimizu. Three-level boolean expressions based on EXOR gates. In *IEICE Transactions on Information and Systems 5*, 2004.

[12] C. Y. Lee. Representation of switching circuits by binary decision programs. *Bell System Techn. J.*, 38(4):985–99, June 1959.

[13] T. Sasao. An exact minimization of AND-EXOR expressions using BDD's. In *Proccedings of IFIP WG 10.5 Workshop on Applications of the Reed-Muller Expansion in Circuit Design*, 1993.

[14] T. Sasao and P. Besslich. "on the complexity of mod-2 sum PLA's". *IEEE Transactions on Computers*, 39, Feb. 1990.

[15] A. K. Verma, P. Brisk, and P. Ienne. Progressive decomposition: A heuristic to structure arithmetic circuits. In *Proceedings of the 44th Design Automation Conference*, pages 404–09, San Diego, Calif., June 2007.

[16] C. Yang and M. Ciesielski. BDS: A bdd-based logic optimisation system. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 21(7):866–76, July 2002.