Improving XOR-Dominated Circuits by Exploiting Dependencies between Operands

Ajay K. Verma
AjayKumar.Verma@epfl.ch

Paolo Ienne
Paolo.Ienne@epfl.ch

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

ABSTRACT

Logic synthesis has made impressive progress in the last decade and has pervaded digital design replacing almost universally manual techniques. A remarkable exception is computer arithmetic and datapath design, where designers still rely mostly on well studied architectures; on datapaths, in most cases, logic synthesis plays at most a minor role in the optimisation of netlists. A case in point is multiple additions performed in carry-save form, such as those fundamentally constituting parallel multipliers: column compressors are usually built exploiting the regularity of the circuit and, due to the very large number of XOR operations, are hardly optimised further by logic synthesisers. In fact, due to the shortcomings of algebraic factoring, XOR operations are usually left untouched by logic synthesisers. In this paper we show a general technique to optimise XOR dominated circuits and we demonstrate its effectiveness on multiplier-like circuits. We show that it optimises significantly the best parallel multipliers by exploiting complex dependencies between the addenda which escape known manual optimisations. Netlists corresponding to top arithmetic architectures can either be synthesised directly or preprocessed through our technique before standard logic synthesis: our preprocessing stage makes it possible to achieve some 20% speed improvement. To our best knowledge, optimisations of the type we show for multiplier-like structures have never been reported—neither manually derived, in computer arithmetic literature, nor automatically derived, in design automation literature.

1. MOTIVATION

The efficient addition of multiple addenda in hardware is an extremely important topic in datapath design: not only is it a common problem in application specific datapaths for signal processing, but it is also one of the key design issues in parallel multipliers. The general approach in multipliers has been known for decades [18] and consists in using a carry-save representation for the multiple additions and then in employing a fast final adder to produce the result. A compressor tree is used for the multiple additions which takes n input integers $x_1, x_2, \ldots, x_n$ and outputs two words C and S, such that the sum of the two words is same as the sum of input integers. The advantage of using this is that it uses carry-save adders in its implementation, which reduce 3 integers to 2 integers in the time taken by a full adder.

Some recent work has addressed the problem of inferring the use of the carry-save representation and compressors beyond multipliers [15, 6, 16, 17] while a large body of literature has been written on the best ways to implement column compressors for multipliers (e.g., [13, 14]). While we will discuss in more detail some of these contributions in a later section, we notice that all approaches to design good or optimal compressors assume that the inputs of the compressor tree are independent of each other. However, often this is not the case and failure to exploit that leads to inferior results.

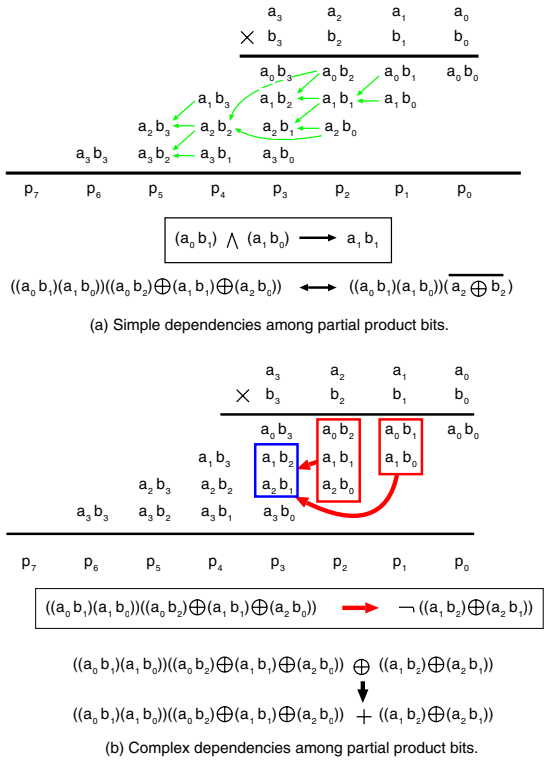

Figure 1: An illustration of various kind of dependencies among the partial product bits of a multiplier.

Consider for instance the multiplication of two integers $\mathcal{A}$ and $\mathcal{B}$: the partial products are generated as the product of all individual bits of $\mathcal{A}$ and $\mathcal{B}$ and accumulated using a compressor tree. As shown in Fig. 1(a) the partial product bits are naturally dependent on each other. For example, if the bits a_0b_1 and a_1b_0 are both 1, then the bit a_1b_1 must also be 1. Using this information, the expression for the 3^{rd} carry bit of the final adder can be simplified: The expression for this carry bit is $a_0b_1a_1b_0(a_0b_2 \oplus a_1b_1 \oplus a_2b_0)$. Since $XY = X(Y \mid X)$, where $(Y \mid X)$ denotes the expression (Y given X), the carry expression can be written as $a_0b_1a_1b_0((a_0b_2 \oplus a_1b_1 \oplus a_2b_0) \mid a_0b_1a_1b_0)$. Using the implication above, the ex-

pression can be reduced to $a_0b_1a_1b_0\ (\overline{a_2 \oplus b_2})$. A much less evident example of dependencies among partial product bits is shown in Fig. 1(b), where, exploiting the dependency among the bits, the expression for the 4^{th} sum bit can be simplified as shown in Fig. 7.

Arguably, exploiting such dependencies is not so much a problem of improving the design of the compressor trees but an issue for the logic synthesiser which should implement them. In fact, logic synthesisers are extremely good at exploiting the dependency among the input bits of AND and OR gates; yet, they never exploit any nontrivial dependency among the input bits of XOR gates. The reason behind this is that most heuristics used for two-level and multi-level optimisation assume that the input is given in the sum of product form. Very few heuristics consider expressions containing XORs (e.g., [1]); however, these heuristics are also very restrictive in nature. One way to optimise XOR-dominated expressions would be to express all XOR expressions in terms of AND and OR gates, and then use normal heuristics. However, this approach both increases the size of the input expression exponentially and the usual heuristics produce much worse results on such expanded expressions due to the limitations of algebraic factoring. For this reason, logic synthesisers rewrite XOR gates in terms of ANDs and ORs only when the operands of XOR gates are very small expressions. Hence, usual synthesisers are unable to exploit nontrivial dependencies among the input bits of XOR gates and, ultimately, do a pretty lousy optimisation job on many arithmetic-relevant circuits as shown in Section 7. Broadly speaking, this is the problem we aim to fix.

In the next section, we introduce the central problem of selectively rewriting XORs, whose solution is at the heart of our optimisation goal. This makes it possible to give a more precise formulation to the problem we tackle. In Section 3 we discuss previous work in the area and outline relevant differences. Section 4 introduces the estimator function which we use to estimate the delay and area of the circuit corresponding to an expression. We then proceed into the main material by explaining in Section 5 the basic strategy to decide which XORs to rewrite. Finally, Section 6 summarises our optimisation algorithm and Section 7 discusses some experimental results using our optimisation technique. Section 8 contains some concluding remarks on this work.

2. REWRITING EVERY XOR OR NOT?

Informally, we already indicated that one of the key shortcomings of logic synthesis when dealing with XOR-dominated arithmetic circuits is not managing to exploit the dependency between operands of XOR gates. We mentioned that such a dependency would be easily exploited by traditional synthesis heuristics if the XORs were rewritten in terms of ANDs and ORs, but also observed that rewriting XORs indiscriminately is a poor strategy. Hence, we need to understand under what conditions rewriting XORs helps logic synthesis. Two examples help us see clearer in this matter:

$$\begin{aligned}(1)\ & P = a_0b_3a_3b_0 + a_0b_3a_1b_2 + a_3b_0a_1b_2,\\ & Q = a_2b_1(a_0b_3 \oplus a_3b_0 \oplus a_1b_2), \text{and}\\ & R = P \oplus Q.\end{aligned}$$

$$\begin{aligned}(2)\ & P = a_0a_1 + a_1a_2 + a_2a_3 + a_3a_4,\\ & Q = b_0b_1 + b_1b_2 + b_2b_3 + b_3b_4, \text{and}\\ & R = P \oplus Q.\end{aligned}$$

The first is an intermediate expression in the implementation of an 8×8 multiplier One can see that expression P corresponds to the majority of the three bits a_0b_3, a_3b_0, a_1b_2, while Q is true only when the parity of the three bits is odd (i.e., the expressions are sufficiently correlated). If we use a logic synthesiser (such as the Synopsys Design Compiler) to get the fastest implementation of R, we get one which has a critical path delay of $0.37ns$ and a standard-cell area of $138.2\mu m^2$. However, if we expand the XOR in $(P \oplus Q)$ in terms of AND and OR gates, the synthesiser produces a circuit with a critical path delay of $0.26ns$ and a standard-cell area of $146.9\mu m^2$ which is 30% faster at a small area cost.

Now consider the second example where P and Q do not share any variables. Here the direct synthesis gives a circuit with a critical path delay of $0.22ns$ and a standard-cell area of $58.8\mu m^2$, while, after expanding the final XOR in terms of AND and OR gates, the logic synthesis results in a circuit with a critical path delay of $0.27ns$ and a standard-cell area of $221.2\mu m^2$. Not only does the critical path now increase, but also the synthesiser is unable to recover the area and the final result is almost four times larger.

The two examples show that sometimes rewriting XOR gates in terms of AND and OR gates is useful and sometimes it is not. Our goal is therefore to recognise the cases where expanding an XOR is useful and separate them clearly from those where it is not. Our problem is therefore the following:

PROBLEM 1. *Given a circuit consisting of AND, OR, NOT, and XOR gates, find the list of XOR gates which, if rewritten in terms of AND and OR gates, result in a circuit with the smallest critical path delay.*

3. STATE OF THE ART

The most general approach to optimise XOR-dominated circuits by exploiting inter operand dependencies is to implement optimisation heuristics better than those currently employed [5]. Not much literature exists in this respect; a recent example [1] has already been mentioned and consists a technique similar to *espresso* [3] but which applies to 2-SPP expressions. These are expressions where both inputs of an XOR must be literals, which is clearly not the case in many circuits of interest, including compressors. Some older work [11] has attempted to exploit *Binary Decision Diagrams (BDD)*, but BDDs for multiplication have been shown to have an exponential number of nodes [4] and make such techniques hardly applicable. This is true also of more recent attempts along similar lines [19, 8].

Focusing on column compressors, a considerable body of work has been accumulated while we tried to improve their speed [10]. Broadly, we can classify these works into two categories: The first consists of works where the counter size inside the compressor tree is increased (a full-adder is a 3-2 counter, that is, a device which counts the ones on 3 inputs and produces a 2-bit binary count). For some time the computer arithmetic community was animated by the problem of identifying the *perfect* counter (e.g., see [13]). The second consists of work where the counter size remains the same but the scheduling of the counters is changed. The *three-greedy* approach [14, 9] falls in this category. It has been proved experimentally that the second approach is way more important than the first one for large input counts. As mentioned previously, the major problem of such work is in ignoring completely what we are most interested in: the correlation across input bits of the compressors.

4. DELAY AND AREA MODEL

Before proceeding, let us introduce the delay and area model which we use to estimate the critical path delay and hardware area of the circuit corresponding to a Boolean expression.

Given a Boolean expression, first we replace each of the XOR subexpressions by a new variable and the new expression can be realized in sum of product form. Next we use *espresso*-like [3] heuristics to simplify the expression and finally factorize it using a heuristic due to Brayton [2]. In the heuristic, the factorisation problem is formulated as a Rectangle-covering problem and a greedy algorithm known as *Ping-pong* is used to solve the same. Although slower than other factoring techniques such as *Quick Factoring (QFACTOR)* [12], it gives much better results. Finally, we replace the new variables by their original XOR expressions and map the resulting expression using NOT gates and two-input AND, OR, XOR gates. To estimate the critical path delay and hardware area, any appropriate values of delay and area can be used for each gate.

5. SELECTIVE EXPANSION OF XORS

As we have seen in Section 2, while expanding XORs sometimes makes the circuit worse in terms of both critical path delay and hardware area, the penalty on hardware area is more severe. The reason for this is that expansion of XOR gate introduces many NOT gates and algebraic factoring does not recognise the complement of an expression as it ignores many Boolean properties. As an example consider the following expression:

$$x = abc + a\overline{b}\overline{c} + \overline{a}b\overline{c} + \overline{a}\overline{b}c,$$
$$factorize(x) \rightarrow x = a(bc + \overline{b}\overline{c}) + \overline{a}(b\overline{c} + \overline{b}c),$$
$$optimise(x) \rightarrow y = b\overline{c} + \overline{b}c,\ x = a\overline{y} + \overline{a}y.$$

Since algebraic factorisation fails to recognise that the expressions $b\overline{c}+\overline{b}c$ and $bc+\overline{b}\overline{c}$ are complements of each other, the new variable y is never introduced as shown above. This is the main reason why expanding XOR increases area severely. As far as critical path delay is concerned, only if the used heuristic is good enough, it might be restored to the original value.

Our problem is to decide whether expanding an XOR gate is useful. As we have seen in the previous section, if the two operands of an XOR gate are significantly correlated, then expanding the XOR gate improves the circuit delay significantly. In the next section we will see that sometimes it is beneficial to expand an XOR gate even with non correlated operands due to extreme correlation between the rest of the expression and the operands of the XOR gate. We call the first kind of correlation *local correlation* and the second *global correlation*. In the next two subsections we discuss how to measure the local and global correlation corresponding to an XOR gate.

5.1 Local Correlation

To measure the local correlation between the operands of an XOR gate we consider the following expansion of XOR:

$$A \oplus B = (\overline{AB})(A + B).$$

This expansion of XOR gate is interesting as it reveals when expanding an XOR gate can be useful. For example, if A and B are such that $AB = 0$, then $A \oplus B$ is equivalent to $A + B$. Similarly if A and B are such that $A + B = 1$, then $A \oplus B$ is equivalent to $\overline{AB}$. More generally, we can say that if A and B are such that either AB or $A + B$ can be computed very quickly compared to A and B, then expanding $A \oplus B$ in terms of AND and OR gates is useful.

Let us see the results of this test on the examples of the previous section. In the first example $PQ = a_0b_0a_1b_1a_2b_2a_3b_3$, which can be computed very quickly compared to P and hence expanding $P \oplus Q$ in terms of AND and OR gates should be useful, which is actually true, as we have seen before. On the other hand, in the second example both of the expressions PQ and $P+Q$ have longer critical path compared to P and Q. Hence, according to our test, we should not expand $P \oplus Q$ in terms of AND and XOR gates. This is reflected by the penalty of expanding XOR, shown in the previous section.

```
isExpansionUseful (Operand A, Operand B) {
  X = gcd(A, B);
  P = A / X;
  Q = B / X;
  (D_P, AR_P) = estimateDelayArea(P);
  (D_Q, AR_Q) = estimateDelayArea(Q);
  (D_PQ, AR_PQ) = estimateDelayArea(PQ);
  (D_P+Q, AR_P+Q) = estimateDelayArea(P + Q);
  ε = 1 - (min(D_PQ, D_P+Q) / max(D_P, D_Q));
  δ = ((AR_PQ + AR_P+Q) / (AR_P + AR_Q)) - 1;
  if (δ ≤ δ_threshold and ε ≤ ε_threshold and (δ / ε) ≤ κ)
    return true;
  return false;
}
```

Figure 2: Algorithm to decide when an XOR gate should be expanded due to local correlation.

The algorithm to decide whether expanding an XOR gate is beneficial or not is shown in Fig. 2. Suppose the two operands of the XOR gate are A and B: First we factor both the expressions and take the common factor X out, and then the correlation between the two quotients P and Q is checked. We estimate the delay and area of the expressions P, Q, PQ, and $P + Q$ using the estimator mentioned in section 4. Based on these values we compute ϵ and δ; the first of the two values denotes the reduction in critical path delay of PQ or $P + Q$ (whichever is smaller) with respect to critical path delay of P or Q (whichever is larger). The second value represents the area overhead of the expressions PQ and $P + Q$ with respect to P and Q. Based on these values an expansion is allowed only if the two operands have a significant correlation ($\epsilon \geq \epsilon_{\text{threshold}}$), the area overhead of expansion is not huge ($\delta \leq \delta_{\text{threshold}}$), and the ratio of the two values δ and ϵ is less than some constant κ (i.e., area penalty per unit gain in critical path delay is small). In Section 7, we show the effects of the constants $\epsilon_{\text{threshold}}$, $\delta_{\text{threshold}}$, and κ on the performance of the resulting circuit.

5.2 Global Correlation

As we have discussed earlier, if the input operands of an XOR gate are correlated enough, then expanding that XOR is advantageous for performance improvement. However, we will see in the next example that the correlation is not a necessary condition to gain by expanding XORs. Consider the two different implementations of a section of a 16-bit adder shown in Fig. 3: It is easy to verify that the two circuit descriptions correspond to equivalent modules of a 16-bit adder. However Design Compiler is unable to recognise this isomorphism and produces significantly different circuits.

The basic difference between the two representations is the definition of p_i and c_i. These two are expressed using XOR gates in the first representation while, in the second, OR gates are used. Now let us check whether our approach mentioned in the earlier section is able to transform the first representation into a new one equivalent to the second representation in terms of performance: First consider the expression for c_i. In the first representation c_i is written as $g_i \oplus p_ic_{i-1}$. The two operands of the XOR gate are g_i and p_ic_{i-1}. Since g_i and p_i correspond to a_ib_i and $a_i \oplus b_i$, they can never be true simultaneously, i.e., $g_ip_ic_{i-1} = 0$. According to the approach of previous section, this implies a strong correlation

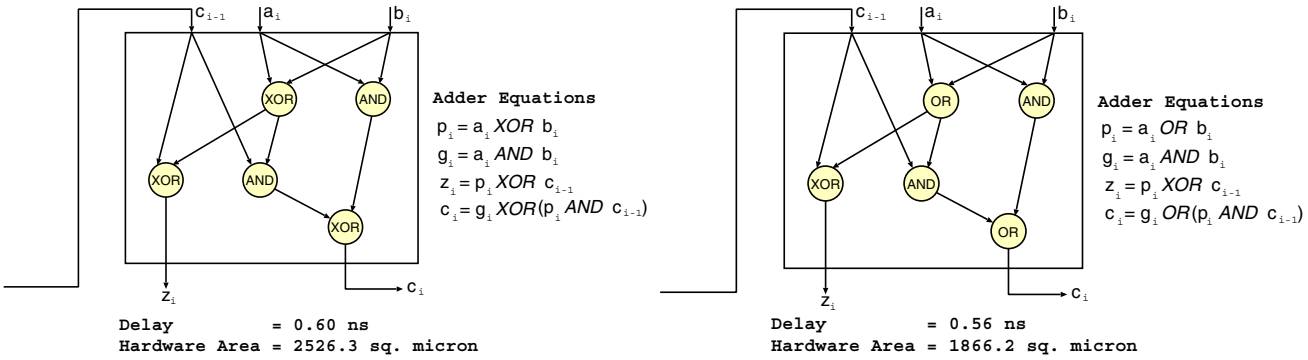

Figure 3: Implementation of a section of a 16-bit ripple carry adder without and with XOR gates.

betwen the operands. Hence we will expand the XOR gate in terms of AND and OR gates resulting in $c_i = g_i + p_i c_{i-1}$, which is the same as the definition of c_i in the second representation:

$$c_i = g_i \oplus p_i c_{i-1},$$
$$c_i = (\overline{g_i p_i c_{i-1}})(g_i + p_i c_{i-1}),$$
$$c_i = g_i + p_i c_{i-1} \qquad \text{(because } g_i p_i c_{i-1} = 0\text{)}.$$

Now let us consider the definition of p_i in the first example, i.e., $a_i \oplus b_i$. Since the inputs of the XOR gate are not correlated at all, according to our approach we should not expand this XOR. However, it can be noted that if we expand the XOR gate of p_i using AND and OR gates, the corresponding definition of c_i will look like $a_i b_i + \overline{a_i b_i}(a_i + b_i)c_{i-1}$. Any logic synthesiser will be able to understand that this expression is the same as $a_i b_i + (a_i + b_i)c_{i-1}$, which is exactly the definition of c_i in the second representation:

$$c_i = a_i b_i + (a_i \oplus b_i)c_{i-1},$$
$$c_i = a_i b_i + (\overline{a_i b_i})(a_i + b_i)c_{i-1},$$
$$c_i = a_i b_i + (a_i + b_i)c_{i-1}.$$

This example shows that sometimes expanding an XOR gate is useful because there is a strong correlation between the operands of the XOR gate and the rest of the expression. This is what we have called global correlation. In order to measure this kind of correlation, we consider the following expansions of an XOR gate inside expression:

$$(A \oplus B) + C = (AB \rightarrow C)(A + B + C),$$
$$(A \oplus B) + C = (\overline{A}\,\overline{B} \rightarrow C)(\overline{AB} + C).$$

Using these rules of rewriting XOR, one can transform the second representation of adder into the first one as shown below:

$$c_i = a_i b_i + (a_i \oplus b_i)c_{i-1},$$
$$c_i = (a_i b_i c_{i-1} \rightarrow a_i b_i)(a_i b_i + (a_i + b_i)c_{i-1}),$$
$$c_i = a_i b_i + (a_i + b_i)c_{i-1}.$$

Note that the expression $(X \rightarrow Y)$ is the same as $(\overline{X} + Y)$. This expansion is advantageous when one of the two expressions $(AB \rightarrow C)$ and $(\overline{A}\,\overline{B} \rightarrow C)$ is almost a tautology (i.e., is true for all but a very sparse set of assignments of Boolean variables). This is because in those cases $(AB \rightarrow C)$ or $(\overline{A}\,\overline{B} \rightarrow C)$ can be computed very quickly and the critical path delay of the whole expressions will be almost same as that of the expressions $(A+B+C)$ or $(\overline{AB} + C)$. Using this kind of expansion in the above case immediately transforms the second representation into the original ripple carry adder as shown above.

A similar illustration of the application of global correlation can be seen in a more complex circuit known as comparator function. A comparator takes two integers $\mathcal{A}$ and $\mathcal{B}$ as inputs, and outputs 1 if $\mathcal{A} > \mathcal{B}$ and 0 otherwise. One way to implement the comparator function is to compare the most significant bits of two integers and if they are different then output 1 or 0 depending on which bit is 1. However if the two bits are equal then one should compare the second most significant bit and so on. In other words the comparator function $c(\mathcal{A}, \mathcal{B})$ can be written like this:

$$c(\mathcal{A}, \mathcal{B}) = (a_n \oplus b_n)a_n + (\overline{a_n \oplus b_n})c(\mathcal{A}', \mathcal{B}'),$$
$$c(\mathcal{A}, \mathcal{B}) = a_n\overline{b_n} + (\overline{a_n \oplus b_n})a_{n-1}\overline{b_{n-1}} + \cdots,$$
$$c(\mathcal{A}, \mathcal{B}) = a_n\overline{b_n} + (a_n \oplus \overline{b_n})a_{n-1}\overline{b_{n-1}} + \cdots.$$

A more efficient way to implement comparator function is to subtract $\mathcal{B}$ from $\mathcal{A}$ and output the sign bit (i.e., the most significant bit) of the result. In other words, the comparator function corresponds to the carry output of $\mathcal{A} - \mathcal{B}$ (the subtraction being $\mathcal{A} + NOT(\mathcal{B}) + 1$). Since the expression for the carry output can be written using propagate (p) and generate (g) functions, the expression for comparator will look like this:

$$c(\mathcal{A}, \mathcal{B}) = g_n + p_n g_{n-1} + \cdots + p_n p_{n-1} \cdots p_1,$$

where

$$(g_n, p_n) = (a_n\overline{b_n}, a_n + \overline{b_n}).$$

Therefore

$$c(\mathcal{A}, \mathcal{B}) = a_n\overline{b_n} + (a_n + \overline{b_n})a_{n-1}\overline{b_{n-1}} + \cdots.$$

Once again, the difference between the two implementation is that the first implementation uses XORs, while the second one uses ORs. It is easy to see that using global correlation the first implementation can be converted into the second one.

$$c(\mathcal{A}, \mathcal{B}) = a_n\overline{b_n} + (a_n \oplus \overline{b_n})a_{n-1}\overline{b_{n-1}} + \cdots,$$
$$c(\mathcal{A}, \mathcal{B}) = (a_n\overline{b_n}a_{n-1}\overline{b_{n-1}} \rightarrow a_n\overline{b_n})$$
$$(a_n\overline{b_n} + (a_n\overline{b_n})a_{n-1}\overline{b_{n-1}}) + \cdots,$$
$$c(\mathcal{A}, \mathcal{B}) = a_n\overline{b_n} + (a_n + \overline{b_n})a_{n-1}\overline{b_{n-1}} + \cdots.$$

In order to ensure that the resulting circuit does not have too much area overhead, we use the parameters $\epsilon_{\text{threshold}}$, $\delta_{\text{threshold}}$, and κ here also. In other words, an expansion is allowed only if it improves the delay at least by $\epsilon_{\text{threshold}}$, does not increase area more than $\delta_{\text{threshold}}$, and the ratio of area overhead and delay gain is less than κ.

```
getScheduling (List L, function f()) {
  if (size(L) = 1) {
    return top(L);
  }
  (x1, x2) = findMinPair(L, f);
  // find x1, x2 ∈ L such that
  // f(x1, x2) is minimum.
  z = x1 ⊕ x2;
  L = (L ∪ {z}) - {x1, x2};
  return getScheduling(L);
}
```

Figure 4: A greedy heuristic to compute the XOR of n expressions where the latency of an XOR gate depends on its inputs.

5.3 Merging Local Correlation and Scheduling Algorithm

It is easy to note that the order in which XOR gates are expanded is also important and affects the final result. This is because the reduction system corresponding to expanding XORs is not persistent and the expansion of an XOR gate might make the expansion of another one disdvantageous. In some cases even locally useful expansions may worsen the overall performance of the circuit. As an example consider the Boolean expression $(A \oplus B \oplus C)$, where the input arrival times of A, B and C are 1, 1, and 2. Assume that the latency of an XOR gate is 0.13 ns and among all XOR gates only $(B \oplus C)$ is beneficial to expand and results in a reduction of 0.08 ns. In other words the direct implementation of $(B \oplus C)$ will have a delay of 2.13 ns; however, after expanding the XOR the corresponding expression will have a delay of 2.05 ns. Now consider two different ways of computing $(A \oplus B \oplus C)$:

$$(A \oplus B) \oplus C$$
$$delay = 0.13 + max(0.13 + max(1, 1), 2) = 2.13,$$
$$A \oplus (B \oplus C)$$
$$delay = 0.13 + max(1, \mathbf{2.05}) = 2.18.$$

Note that the second implementation, where XOR gate in $(B \oplus C)$ is expanded, has a longer critical path compared to the first implementation where no XOR gate was expanded. This is because the second circuit is more lopsided than the first one. This indicates that we also need to consider the input arrival times of the operands of an XOR while deciding whether its expansion is useful. In other words, the problem is to find a scheduling and then expanding those XORs which have correlated inputs, the resulting expression has the least critical path delay. More formally we can formulate the problem as follows.

PROBLEM 2. *Given a list of n expressions $X_1, X_2, \cdots, X_n$, and a function $f(.)$ which takes two expressions and returns the time when their XOR can be computed, find an optimal scheduling to compute the XORs of the n expressions.*

Note that if no expansion of XOR gates is allowed then the function $f(.)$ will be a trivial function and can be written like $f(x, y) = D_{XOR} + max(t_x, t_y)$, where t_x, t_y are the arrival times of expressions x and y. In this case the problem can be solved optimally using a two-greedy strategy: The algorithm takes a list of expressions and chooses the two expressions from the list which have the least arrival time; it then removes them from the list and inserts a new variable corresponding to their XOR. This process is repeated till the size of the list is reduced to one.

However, when XOR-expansion is allowed, the function $f(.)$ denotes the time by which XOR of two expressions can be computed (either by using an XOR gate or by rewriting the XOR using AND and OR gates—whichever is better in terms of delay). This means that the behaviour of $f(.)$ is largely unknown. The only restriction is that $f(x, y) \leq D_{XOR} + max(t_x, t_y)$. In order to solve the problem with general $f(.)$, we use a slight variation of the two-greedy strategy as shown in Fig. 4. The only difference is that in each iteration, instead of choosing the two variables with shortest arrival times, we choose the pair whose XOR can be computed earliest among all pairs.

Although this heuristic will not always produce optimal results, practically it improves the performance of a circuit significantly. Also, it can be proved that this heuristic will never worsen a circuit, i.e., the circuit produced by the heuristic will have a critical path no longer than the critical path in optimal scheduling of the circuit without expanding any XOR gate. More formally:

THEOREM 1. *If the XOR of n expressions $X_1, X_2, \cdots, X_n$ can be computed in time T without expanding any XOR gate, then the above mentioned greedy heuristic of Fig. 4 will result in a circuit with critical path delay $T' \leq T$.*

PROOF. Instead of proving the above theorem we will prove the following more general statement. Assume two lists of n expressions $X_1, X_2, \cdots, X_n$ and $Y_1, Y_2, \cdots, Y_n$, with arrival times t_{x1}, $t_{x2}, \cdots, t_{xn}$ and $t_{y1}, t_{y2}, \cdots, t_{yn}$ respectively, and $t_{yi} \leq t_{xi}$ for all i. If the XOR of $X_1, X_2, \cdots, X_n$ is computed without expanding any XOR in T_x time and the XOR of $Y_1, Y_2, \cdots, Y_n$ is computed using the above mentioned greedy heuristic in T_y time, then $T_y \leq T_x$. It is easy to note that proving this statment suffices to prove the theorem.

The proof is by induction on n. The base case corresponds to $n = 1$, which is trivial. Assume that the statement is true for $(n-1)$ expressions and that the latency of XOR gate is t. Without loss of generality, we can also assume that the arrival times of expressions $X_1, X_2, \cdots, X_n$ as well as the arrival times of expressions $Y_1, Y_2, \cdots, Y_n$ are in non-decreasing order.

We have mentioned before that if the expansion of XOR is not allowed, then the two greedy strategy is optimal; hence, in the first step to compute the XORs of $X_1, X_2, \cdots, X_n$, the optimal strategy computes the XOR of X_1 and X_2. After this step, the list of operands will look like $X_{new}, X_3, \cdots, X_n$, with arrival times $t + t_{X2}, t_{X3}, \cdots, t_{Xn}$. On the other hand, in the first step to compute XORs of $Y_1, Y_2, \cdots, Y_n$, the greedy strategy will compute the XOR of Y_i and Y_j, where (Y_i, Y_j) is the pair whose XOR can be computed earliest among all possible pairs. After this step, the list of operands will be $Y_{new}, Y_1, \cdots, Y_{i-1}, Y_{i+1}, \cdots, Y_{j-1}, Y_{j+1}, \cdots Y_n$, with arrival times $t_{new}, t_{Y1}, \cdots, t_{Yn}$. According to the definition of greedy heuristic, we can see that

$$\begin{aligned}
&\forall \alpha, \beta (t_{new} \leq f(Y_\alpha, Y_\beta)) \\
&\Rightarrow \forall \alpha, \beta (t_{new} \leq t + max(t_{Y\alpha}, t_{Y\beta})) \\
&\Rightarrow t_{new} \leq t + max(t_{Y1}, t_{Y2}) \\
&\Rightarrow t_{new} \leq t + max(t_{X1}, t_{X2}) \\
&\Rightarrow t_{new} \leq t + t_{X2}.
\end{aligned}$$

Also note that $t_{Y1} \leq t_{X1} \leq t_{Xi}$ and $t_{Y2} \leq t_{X2} \leq t_{Xj}$. This means that after one step also the arrival times of expressions satisfy the property $t_{Yk} \leq t_{Xk}$. But after one step we have $(n-1)$ expressions, and now we can say by the induction hypothesis that the XOR of the X_i's can not be computed any sooner than the XOR of Y_i's. This completes the proof. □

```
get_XOR_Delay (X, Y) {
  P = gcd(X, Y);
  (D_X, AR_X) = estimateDelayArea(X);
  (D_Y, AR_Y) = estimateDelayArea(Y);
  (D_new, AR_new) =
      estimateDelayArea(P((X/P)(Y/P))'(X/P + Y/P));

  if ((isExpansionUseful(X, Y))
    return D_new;
  else
    return (t_XOR + max(D_X, D_Y));
}

optimizeUsingLocalCorrelation (tree T) {
  forall children T_i of T do
    optimizebyExpanding(T_i);

  if (op(root(T)) = XOR)
    return (tree)(getScheduling(
    {(expr)T_1, ..., (expr)T_n}, get_XOR_Delay()));
    // return the tree corresponding to the
    // scheduling of XORs.
}
optimizeUsingGlobalCorrelation (tree T) {
  forall nodes n of T in reverse topological order do
  // i.e., traverse the tree from leaves to root
    if (op(n) = OR and (op(n.leftChild) = XOR or
      op(n.rightChild) = XOR))
        rewriteUsingGlobalCorrelation(subtree(n));
        // Check for the subtree rooted at n if the
        // expansion based on global correlation has
        // lower critical path delay, if so replace
        // the subtree by the expansion based on
        // global correlation.
      }
  optimizeTree (tree T) {
    while (improvement in performance) {
      collapseTree(T);
      // collapse the tree using the associativity
      // of the operators, so that the successor of a
      // node must be of different type than this node.

      optimizeUsingLocalCorrelation(T);
      optimizeUsingGlobalCorrelation(T);
    }
  }
```

Figure 5: Algorithm for computing the optimised expression by selective expansion of XOR gates.

6. ALGORITHM AND COMPUTATIONAL COMPLEXITY

The complete algorithm is shown in Fig. 5. The algorithm takes the input expression in the form of a tree where each node corresponds to a Boolean operator and the edges correspond to data dependencies. First, this input tree is collapsed by merging adjacent nodes of same type into a single multi-input node (e.g., multi-input AND etc.). After that, the tree is optimised using local and global correlations. The optimisation using local correlation consists of traversing the tree in topological order from leaves to root and optimise the subtrees rooted at XOR nodes using the greedy heuristic mentioned in the earlier section. Then, we check whether any XOR gate can be expanded using the global correlation criteria to improve the performance of the circuit and, if so, rewrite the corresponding subexpression using the expansion based on global correlation. The above two steps are repeated till we have an improvement in the delay of the circuit.

In order to analyse the time complexity of the algorithm we need to find out how many times we are calling the function estimateDelayArea() because that is the most expensive function. It is easy to see that in the function optimizeUsingGlobalCorrelation() we call the estimator function $O(n)$ times, where n is the number of nodes in the tree. However, in the function optimizeUsingLocalCorrelation() there are $O(n^2)$ calls to the estimator function. To see this, first let us find the time complexity of the getScheduling() function. In the getScheduling() function, the size of the input list decreases by one in each iteration (i.e., if the initial size is m, then there will be $(m-1)$ iterations). Also, in each iteration, we need to find the pair whose XOR can be computed earliest among all pairs. Note that in the first iteration we need to compute the effective XOR latency of all pairs. However, in subsequent iterations, only $O(k)$ new pairs are generated, where k is the list size at the begining of that iteration, i.e., the estimator function is called only $O(m^2)$ times. Since we call the getScheduling() function at each XOR node, the total number of calls to the estimator function will be $O(n^2)$. Also, we observed empirically that we need to iterate the functions optimizeUsingGlobalCorrelation() and optimizeUsingLocalCorrelation() only a constant number of times. Hence the overall complexity of the algorithm is $O(n^2 T_{\text{estimator}})$, where $T_{\text{estimator}}$ is the time complexity to estimate the delay and area of an expression.

ADPCM Decoder		
Unoptimised	$6556.0\mu m^2$	$1.06ns$
Selective Expansion	$6933.8\mu m^2$	$0.90ns$
8×8**-bit Multiplier**		
DesignWare	$4487.6\mu m^2$	$1.60ns$
Three Greedy Approach	$5995.9\mu m^2$	$1.28ns$
Selective Expansion (TGA as input)	$7261.8\mu m^2$	$1.02ns$
Constant Multiplication $(A \times 7)$		
$\mathcal{A} \times 7$	$2586.8\mu m^2$	$0.85ns$
$\mathcal{A} + 2\mathcal{A} + 4\mathcal{A}$	$3155.3\mu m^2$	$0.72ns$
$8\mathcal{A} - \mathcal{A}$	$1940.5\mu m^2$	$0.56ns$
Selective Expansion ($\mathcal{A} + 2\mathcal{A} + 4\mathcal{A}$ as input)	$3018.4\mu m^2$	$0.50ns$
Selective Expansion ($8\mathcal{A} - \mathcal{A}$ as input)	$2822.1\mu m^2$	$0.52ns$
15**-bit Comparator**		
Unoptimised	$514.9\mu m^2$	$0.40ns$
Selective Expansion	$466.2\mu m^2$	$0.33ns$
15**-bit Adder**		
Unoptimised	$1318.5\mu m^2$	$0.56ns$
Selective Expansion	$1318.5\mu m^2$	$0.56ns$

Table 1: Optimisation results for all our benchmarks.

$\epsilon_{\text{threshold}}$	$\delta_{\text{threshold}}$	κ	Delay (ns)	Hardware Area (μm^2)
0.1	0.2	2	1.02	7261.8
0.05	0.1	1	1.07	7337.6
0.05	0.1	3	1.04	7123.4
0.05	0.4	1	1.03	8127.3
0.05	0.4	3	1.00	10123.4
0.2	0.1	1	1.10	6668.5
0.2	0.1	3	1.08	7487.4
0.2	0.4	1	1.07	7786.0
0.2	0.4	3	1.04	9123.8

Table 2: Illustration of the moderate effects of various parameters on the final result output by Selective Expansion for 8×8-bit multiplier.

7. EXPERIMENTS

We implemented our algorithm using Maple 10, which is used as a front-end to Synopsys Design Compiler. We synthesise the input circuit twice: once directly and a second time after optimising using our algorithm, which we call as *Selective Expansion*. All the circuits are synthesised using a common standard-cell library for UMC $0.13\mu m$ CMOS technology. Table 1 shows the results of our implementations. There are qualitatively five different arithmetic circuits on which we ran our algorithm: The first circuit is the kernel of `adpcmdecode` [7]. Here the input is the already optimised circuit using the techniques mentioned in [17]. As we can see, our algorithm improves the delay by 15% with a marginal penalty of

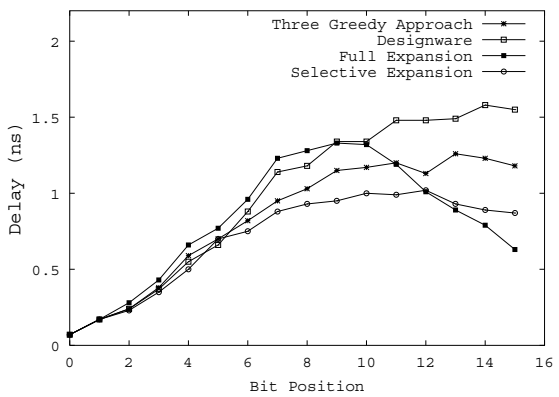

Figure 6: Comparison of bitwise delays of the multiplier generated by DesignWare, Three Greedy Approach, Selective Expansion and Full Expansion.

5% in terms of hardware area.

The second benchmark consists of an 8×8-bit parallel multiplier. Here we compare the results of Selective Expansion with the DesignWare implementation and the Three Greedy Approach (TGA) [14]. The results show that the Three Greedy approach, which is already much better than the DesignWare implementation, is outperformed by Selective Expansion and an improvement of 20% in delay is achieved. In Fig 6, we compare the delay values of individual product bit expressions. Note that Selective Expansion always produces better results compared to DesignWare and TGA. Compared with Full Expansion (where all XOR gates are expanded), Selective Expansion performs better for all but the three most significant bit expressions; however, the area penalty in Full Expansion is huge (almost 20 times the area incurred by Selective Expansion). Table 2 shows the effect of various parameters on the output results for the 8×8-bit multiplier generated by Selective Expansion algorithm. Notice that the delay values for all parameters are pretty close to each other. For other benchmarks we use the combination $\epsilon_{\text{threshold}} = 0.1$, $\delta_{\text{threshold}} = 0.2$, and $\kappa = 2$.

The third benchmark is a constant multiplier where a 16-bit integer $\mathcal{A}$ is multiplied by the constant 7. Note that $(\mathcal{A} \times 7)$ can be implemented in various ways: One possibility can be to use the multiplication provided by Design Compiler, while another is to rewrite $(\mathcal{A} \times 7)$ as $(\mathcal{A} + 2\mathcal{A} + 4\mathcal{A})$ and then use a carry save adder to add the three values. An even better way is to rewrite $(\mathcal{A} \times 7)$ as $(8\mathcal{A} - \mathcal{A})$ and use a single adder to add them. As we can see the, results of the three implementations are very different from each other. We ran Selective Expansion algorithm on both forms of inputs $(\mathcal{A} + 2\mathcal{A} + 4\mathcal{A})$ and $(8\mathcal{A} - \mathcal{A})$. Note that the resulting circuits for the two inputs are almost same in terms of critical path delay.

The fourth benchmark is the comparator function mentioned in section 5.2. One can see that in this benchmark we not only reduce critical path delay but also optimise the circuit in terms of hardware area. The last arithmetic circuit corresponds to the 15-bit adder. Although it is an XOR-dominated circuit, the XOR operands are independent of each other and hence Selective Expansion does not expand any of them. This means that the adder circuit will remain unchanged by the Selective Expansion, which is in fact reflected by the results.

A partial execution of our algorithm to optimise the expression for the 4^{th} bit of the output of an 8×8-bit multiplier is shown in Fig. 7. Note that the expression optimised by our algorithm is very similar to the manually optimised expression.

$z_3 = a_1b_2 \oplus a_2b_1 \oplus a_0b_0a_1b_1(a_0b_2 \oplus a_1b_1 \oplus a_2b_0) \oplus$
$(a_0b_2a_2b_0 + (a_0b_2 + a_2b_0)\ a_1b_1);$

// After simple optimisations based on simple
// dependencies among partial product bits.
$z_3 = a_1b_2 \oplus a_2b_1 \oplus a_0b_0a_1b_1(a_2 \oplus \overline{b_2}) \oplus$
$(a_0b_2a_2b_0 + (a_0b_2 + a_2b_0)\ a_1b_1);$

// Using the dependency between XOR operands
// $(a_1b_2 \oplus a_2b_1)\ (a_0b_0a_1b_1(a_2 \oplus \overline{b_2})) = 0.$
$z_3 = ((a_1b_2 \oplus a_2b_1) + a_0b_0a_1b_1(a_2 \oplus \overline{b_2})) \oplus$
$(a_0b_2a_2b_0 + (a_0b_2 + a_2b_0)\ a_1b_1);$

// (PQR $\rightarrow$ S) $\Rightarrow$ P (Q $\oplus$ R) + S = P (Q + R) + S
// Here P = $a_0b_0a_1b_1$, Q = a_2, R = $\overline{b_2}$, S = $a_1b_2 \oplus a_2b_1$
$z_3 = ((a_1b_2 \oplus a_2b_1) + a_0b_0a_1b_1(a_2 + \overline{b_2})) \oplus$
$(a_0b_2a_2b_0 + (a_0b_2 + a_2b_0)\ a_1b_1);$

// Manually optimised expression
$z_3 = ((a_1b_2 \oplus a_2b_1) + a_0b_0a_1b_1) \oplus$
$(a_0b_2a_2b_0 + (a_0b_2 + a_2b_0)\ a_1b_1);$

Figure 7: A partial execution of our algorithm to optimise the 4^{th} output bit of 8×8-bit multiplier.

8. CONCLUSIONS

Although logic synthesis has made enormous progress and has supplanted manual design in the great majority of digital design tasks, computer arithmetics is possibly the last area where people still manually or semi manually optimise circuits. Many practical arithmetic circuits, both standard and application specific, are dominated by additions; equivalently, at the Boolean level, they contain a large number of nested XORs. We have shown how classic limitations of logic synthesis heuristics can be circumvented by selectively rewriting only some XOR operations in terms of ANDs and ORs. While synthesisers seldom rewrite XOR operations, our heuristic tries to do that in all cases where one can observe some specific form of dependency between the two XOR operands: when this is the case, such rewriting will likely let classic synthesis heuristics achieve better results. In our experiments we have used state-of-the-art logic descriptions of the benchmarks, where all classic architectural optimisations have been applied already, and we have measured up to more than a fifth reduction in critical path for a small area cost. This improvement on already optimised components would hardly be believable, were it not for the qualitative realization that the solutions found by our algorithm on a parallel multiplier, for instance, correspond to a type of optimisation that we have never found described in computer arithmetic literature—probably due to their irregularity and the complexity of implementing them manually.

Our results illustrate that it is high time for design automation to conquer arithmetic territories, so far largely left to manual and architectural optimisations. With the increasing number of software-oriented designers implementing complex signal and media processing algorithms on FPGAs or ASICs, automatically achieving implementations of arithmetic components which are at least equal to manual implementations becomes a must. And, as we show, better-than-manual designs are a concrete possibility.

9. REFERENCES

[1] 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.

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

[3] 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.

[4] 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.

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

[6] T. Kim, W. Jao, and S. Tjiang. Circuit optimization using carry-save-adder cells. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, CAD-17(10):974–84, Oct. 1998.

[7] C. Lee, M. Potkonjak, and W. H. Mangione-Smith. MediaBench: A tool for evaluating and synthesizing multimedia and communicatons systems. In *Proceedings of the 30th Annual International Symposium on Microarchitecture*, pages 330–35, Research Triangle Park, N.C., Dec. 1997.

[8] A. Mishchenko and M. Perkowski. Fast heuristic minimization of exclusive-sums-of-product. In *Proceedings of the 5th International Reed-Muller Workshop*, Los Angeles, Calif., Aug. 2001.

[9] V. G. Oklobdzija, D. Villeger, and S. S. Liu. A method for speed optimized partial product reduction and generation of fast parallel multipliers using an algorithmic approach. *IEEE Transactions on Computers*, C-45(3):294–306, Mar. 1996.

[10] A. R. Omondi. *Computer Arithmetic Systems*. Prentice Hall, New York, 1994.

[11] 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.

[12] E. Sentovich, K. Singh, L. Lavagno, M. C., R. Murgai, A. Saldanha, S. H., S. P., R. Brayton, and A. Sangiovanni Vincentelli. SIS: A system for sequential circuit synthesis. Technical report, University of California, Berkeley, Calif., 1992.

[13] P. Song and G. De Micheli. Circuit and architecture trade-offs for high speed multiplication. *IEEE Journal of Solid-State Circuits*, 26(9), Sept. 1991.

[14] P. F. Stelling, C. U. Martel, V. G. Oklobdzija, and R. Ravi. Optimal circuits for parallel multipliers. *IEEE Transactions on Computers*, C-47(3):273–85, Mar. 1998.

[15] Synopsys. *Creating High-Speed Data-Path Components—Application Note*, Aug. 2001. Version 2001.08.

[16] J. Um and T. Kim. An optimal allocation of carry-save-adders in arithmetic circuits. *IEEE Transactions on Computers*, C-50(3):215–33, Mar. 2001.

[17] A. K. Verma and P. Ienne. Improved use of the carry-save representation for the synthesis of complex arithmetic circuits. In *Proceedings of the International Conference on Computer Aided Design*, pages 791–98, San Jose, Calif., Nov. 2004.

[18] C. S. Wallace. A suggestion for a fast multiplier. *IEEE Transactions on Electronic Computers*, C-13(2):14–17, Feb. 1964.

[19] C. Yang, M. J. Ciesielski, and V. Singhal. A BDD-based logic optimization system. In *Proceedings of the 37th Design Automation Conference*, pages 92–97, Los Angeles, Calif., June 2000.