Approximate Logic Synthesis under General Error Magnitude and Frequency Constraints

Jin Miao, Andreas Gerstlauer, and Michael Orshansky
Department of Electrical & Computer Engineering, The University of Texas at Austin
{jinmiao, gerstl, orshansky}@utexas.edu

Abstract

Recent interest in approximate circuit design is driven by its potential for large energy savings. In this paper, we address the problem of approximate logic synthesis (ALS). ALS is concerned with formally synthesizing a minimum-cost approximate Boolean network whose behavior deviates in a well-defined manner from a specified exact Boolean function, where in this work, we allow the deviation to be constrained by both the magnitude and frequency of the error.

We make two contributions in solving this general ALS problem: The first contribution is to establish that the approximate synthesis problem un-constrained by the frequency of errors is isomorphic with the Boolean relations (BR) minimization problem. That equivalence allows us to exploit recently developed fast algorithms for BR problems to solve the error magnitude-only constrained ALS problem. The second contribution is an efficient heuristic algorithm for iteratively refining the magnitude-constrained solution to arrive at a solution also satisfying the error frequency constraint.

Our combined greedy approximate logic synthesis (GALS) approach is able to operate on any Boolean network for which the deviation measures can be specified and is most immediately applicable to arithmetic blocks. Experiments on adder and multiplier blocks demonstrate literal count reductions of up to 60% under tight error frequency and magnitude constraints.

1. Introduction

Energy minimization has become the major concern in the design of VLSI systems. One way to reduce energy consumption is to exploit the trade-off between reduced computation accuracy and improved energy efficiency for digital systems that naturally tolerate errors, such as signal processing circuits. Many recent approaches have studied the possibility of approximate computation at different levels ranging from algorithms [5, 18] and architectures [8, 9, 11] to the logic [4, 22] and transistor levels [7].

One class of techniques seeks to realize approximate computation by deriving approximate, or inexact, versions of specified combinational Boolean functionality. Essentially, this is accomplished by modifying some outputs of a function's truth table such that the produced error is tolerable. Such modifications typically result in logic implementations of reduced complexity, smaller area, delay, and energy. Such logic-level optimizations have been applied to several arithmetic building blocks, such as adders and multipliers [1, 14, 16, 23]

Most efforts in this area so far have been ad hoc. There is a need to develop effective rigorous techniques for automated approximate logic synthesis. Designing approximate circuits in an ad hoc manner is not a viable option as there exists a large design space with trade-offs between acceptable accuracy and energy, where acceptable errors may vary from application to application. Importantly, depending on the application, error tolerance is primarily a function of either the frequency of errors, the magnitude of errors, or both. For example, in applications that can not directly accept erroneous results, the frequency of triggering correction mechanisms determines ultimate overhead. By contrast, in image and video processing applications, if the produced pixel values have small error magnitudes, a human will not be able to distinguish such subtle changes. In typically employed Peak Signal-to-Noise Ratio (PSNR) metrics, a quadratic relationship between error frequency and magnitude is used to assess perceived quality. For a high-quality PSNR value of 50dB at 8-bit pixel depth, this can, for example, be translated into a frequency constraint of 7% for error magnitudes no more than 3 or up to 65% errors if pixel values have error magnitudes no more than 1.

Thus, overall, there is a need for rigorous automation to perform approximate logic synthesis (ALS) under both types of constraints. Existing ALS approaches thus far have focused on single error metrics only. A two-level approximate logic synthesis algorithm was introduced in [19]. In that work, the objective was to synthesize a minimized circuit under constrained error frequency. The algorithm did not consider constraints on error magnitude. Moreover, it suffers from high runtime complexity, especially, at large error frequencies. By contrast, in [16] and [20], absolute and relative error magnitude constraints were set without limiting error frequency. Both techniques are built upon an unmodified conventional logic synthesis flow, which is not as efficient as integrating support for approximation into the logic synthesis engine directly. In [4, 13], the authors consider both error frequency and relative error metrics. However, distinct solutions are provided and the two constraints are never explored jointly. Furthermore, due to the nature of the proposed pattern-driven approach, the optimization space is restricted to only a small subset of inputs.

In this paper, we address the problem of approximate logic synthesis (ALS) under arbitrary error magnitude and error frequency constraints. We develop a two-level logic minimization algorithm that rigorously synthesizes a minimum-cost cover of a Boolean function that is allowed to deviate from an exact Boolean function in a constrained manner. We adopt a two-phase approach to solve the minimization. The first phase solves the problem that is constrained only by magnitude of error. In the second phase, this frequency unconstrained problem is iteratively refined to arrive at a solution that also satisfies the original error frequency constraint.

We make two major contributions. The first contribution is the realization that the approximate synthesis problem un-constrained by the frequency of errors is isomorphic with the Boolean relations minimization problem. A *Boolean relation* is a one-to-many, multi-output Boolean mapping, $R : B^n \rightarrow B^k$. Thus, Boolean relations are a generalization of Boolean functions. We show how error magnitude constraints can be formulated as constraints on the possible values of Boolean function outputs and thus be equivalent to Boolean relations. That mapping allows us to exploit recently developed fast algorithms for BR problems to solve the error magnitude-only constrained ALS problem.

The second contribution is an efficient heuristic algorithm for iteratively refining the magnitude-constrained solution to arrive at a solution also satisfying the error frequency constraint. The algorithm (a) finds the optimal set of function minterms on which the exact outputs must be enforced, and (b) systematically corrects, in a greedy fashion, the erroneous outputs of the BR solution that lead to the smallest cost increase until the error frequency constraint is met.

In summary, we describe an efficient algorithm, which we call greedy approximate logic synthesis (GALS), that can rigorously

handle both error magnitude and frequency constraints as part of synthesizing an approximate two-level Boolean network. Experiments on adders and multipliers demonstrate literal count reductions of up to 60% under tight magnitude and frequency constraints.

2. ALS Constrained by Error Magnitude Only

In this section, we discuss the approximate logic synthesis problem when constraining the magnitude of allowed error only. Thus, we consider only the patterns of allowed errors that the function may produce, but not how often the errors occur.

We focus on our first contribution, which is the realization that the approximate synthesis problem un-constrained by the frequency of errors is isomorphic with the Boolean relations problem.

2.1 Isomorphism between Frequency-Unconstrained ALS and Boolean Relations

The most immediate domain of application of ALS is in synthesizing approximate arithmetic blocks for error-tolerant computing algorithms and applications. In applications that involve approximate arithmetic functions, such as in the signal processing domain, it is typically important to satisfy constraints on the magnitude of the possible error as well as the frequency of such errors. Here, frequency is defined as the number of minterms on which an error occurs as a fraction of the total number of minterms.

Constraining the magnitude of error is the most natural approach to limiting the outputs of arithmetic circuits, since a clear notion of distance is available for these functions. Consider a multi-output Boolean function $F : B^n \rightarrow B^k$ that defines a combinational network of an arithmetic circuit, e.g., an adder. First, we consider constraining the magnitude of possible errors. The output of F is the result of binary arithmetic computation. We aim to synthesize its magnitude-constrained approximate version F_m, such that the only constraint is that $|F - F_m| \leq M$. Here, $|\cdot|$ is the absolute value operator, and thus we constrained the range of possible output values of the approximate function to be no greater than M. Note an important implicit aspect of our definition. The frequency-unconstrained function F_m will have an *arbitrary* error frequency. Specifically, there is no implication that it has an error on every input.

To explicitly account for the error frequency (rate) of an approximate function, we introduce a modified notation and denote as $F_{m,r}$ an approximate version of F with exactly r minterms in error and with the constraint on the magnitude of error (no greater than M). Let the error frequency constraint be R indicating that no more than R minterms are allowed to be in error. With that, the full approximate logic synthesis problem is:

$$\begin{aligned} \min \quad & L(F_{m,r}) \\ \text{s.t.} \quad & r \leq R, \\ & |F(x) - F_{m,r}(x)| \leq M \quad \forall x \in B^n \end{aligned} \tag{1}$$

where $L(F)$ is the number of literals in a sum-of-products representation of function F.

One possible strategy for solving the above problem is to start with an exact function F and gradually introduce errors while controlling *both* the frequency and magnitude of allowed errors.

However, the strategy we pursue in this paper is based on a *two-phase* solution. In the first phase, the frequency unconstrained problem is solved. In the second phase, the unconstrained solution is iteratively refined to arrive at the solution that satisfies the original error frequency constraint. The frequency un-constrained problem is given by:

$$\begin{aligned} \min \quad & L(F_m) \\ \text{s.t.} \quad & |F(x) - F_m(x)| \leq M \quad \forall x \in B^n \end{aligned} \tag{2}$$

where F and F_m are the exact and approximate functions, respectively.

The key observation is that the above ALS problem constrained only by error magnitude is isomorphic with minimization of Boolean relations, which is a known and extensively-studied problem in traditional synthesis. A Boolean relation can be formally defined as follows [3]:

Definition 2.1. Boolean relation. A *Boolean relation* is a one-to-many, multi-output Boolean mapping, $R : B^n \rightarrow B^k$. A set of multi-output Boolean functions, f_i, each compatible with R, is associated with a relation. A Boolean relation is specified by defining for each input $x \in B^n$ a set of equivalent outputs, $I_x \subseteq B^k$.

Thus, Boolean relations are a generalization of Boolean functions, where each input corresponds to more than one output. An incompletely specified logic function with *don't care* is a special case of a single-output Boolean relation.

To establish the equivalence of ALS with the Boolean relation problem, we observe that the constraint $|F - F_m| \leq M$ can be rewritten minterm-wise: for each minterm x_i of function F, allow the value of $F_m(x_i)$ to take values in the set $F \cup E_i$, where E_i is the specified output error set for x_i. Thus, E_i represents the additional values that the function can take while satisfying the error magnitude constraint. Now, each input corresponds to more than one output. The new formulation is given by:

$$\begin{aligned} \min \quad & L(F_m) \\ \text{s.t.} \quad & F_m(x_i) \in F(x_i) \cup E_i(x_i) \quad \forall x_i \in B^n \end{aligned} \tag{3}$$

Example 2.1. We use the simple example of an adder to illustrate the concepts being introduced. For a 1-bit half adder, the equivalence is illustrated via a tabular representation for $M = 1$:

F	
a, b	c, s
00	{00}
01	{01}
10	{01}
11	{10}

F_m	
a, b	c, s
00	{00, 01}
01	{01, 00, 10}
10	{01, 00, 10}
11	{10, 01, 11}

It is clear that the above tabular form sets up a Boolean relation (BR) representation, according to Def.2.1, where each input corresponds to more than one output.

2.2 Boolean Relation Solvers

We have established the equivalence between the error frequency-unconstrained approximate logic synthesis problem and the Boolean relation minimization problem. This is advantageous as there exist several exact and heuristic approaches for solving the BR problem. Here, we give a brief overview of the available BR minimization techniques. The exact method reported in [3] employs an approach similar to the Quine-McCluskey procedure [17]. The minimization is formulated as a binate covering problem and solved by integer linear programming. Other exact methods are [12] and [10]. As is common, the exact approaches are limited to solving small and medium-size BR instances due to the algorithm complexity. Heuristic solutions trade result optimality for computational tractability. Herb [6] is based on the two-level minimization algorithm of ESPRESSO [15] and test pattern generation techniques. Gyocro [21] also relies on ESPRESSO. While it improves on some of the weakness in Herb, it still remains slow.

We adopt a recently developed heuristic algorithm BREL [2]. BREL is a recursive algorithm that uses a branch-and-bound solution strategy. It first over-approximates (using the maximum flexibility provided by the relation) the BR into a multi-output Boolean

function where each output is minimized independently using standard techniques for function minimization. If the minimized Boolean function is compatible with the original Boolean relation, then it is accepted as the solution. Otherwise, the algorithm splits the original Boolean relation R into two sub-BRs R_1 and R_2. This is done by selecting one conflict minterm such that each sub-BR operates on one output component of this minterm. Sub-BRs are then solved independently following the same procedure recursively. BREL substantially outperforms the earlier tools in terms of runtime and result quality.

3. Frequency-Constrained ALS Algorithm

This section describes our second major contribution: the development of an effective heuristic logic optimizer that accepts the solution of the frequency-unconstrained ALS and carries out further optimizations to guarantee the solution feasibility with respect to the frequency of errors.

Because the result of solving the Boolean relation minimization for F_m does not constrain the number of minterms in error, the solution may not satisfy the constraints on error frequency. Let the result of solving the Boolean relation be the function $F_{M,k}$, where k refers to the resulting actual error frequency. If the error frequency constraint R is smaller than k, then we need to reduce the number of minterms on which the function is different from the exact one.

3.1 Mapping to Min-Cost Increase Problem

We first clarify an important property of the solution of the Boolean relations problem. As a solution to the problem of Equation (3), the function $F_{M,k}$ has the minimal cover (in terms of literals) among all functions that satisfy $F_{M,k} \in F \cup E_i$. Therefore, we know that the following holds:

Theorem 3.1. *For any function* $F_{M,r}$

$$L(F_{M,r}) \geq L(F_{M,k}), \quad \textit{for any } r < k.$$

Proof. If there is an r such that $L(F_{M,r}) < L(F_{M,k})$, then the BR solver reports $F_{M,r}$ as the BR solution since $F_{M,r}$ also satisfies the specified Boolean relation and has fewer literals. □

We now reformulate the problem to be solved in the second phase. To solve the problem in Equation (1), we need to find the function $F_{M,R}$ that minimizes the literal increase $L(F_{M,R}) - L(F_M)$:

$$\begin{aligned} \min \quad & L(F_{M,R}) - L(F_M) \\ \text{s.t.} \quad & |F_{M,R} - F| \leq M \end{aligned} \tag{4}$$

We propose an iterative and greedy algorithm that searches for $F_{M,R}$ by repeatedly identifying the minterms on which the correctness of the function should be enforced. The algorithm proceeds by making localized changes to the function by accepting steps that minimize literal increase while reducing the maximum number of error-minterms and guaranteeing that the magnitude constraint remains satisfied.

3.2 Formalization of the Frequency-Constrained ALS Algorithm

The algorithm works with a set of minterms on which the function F_M, produced by the frequency-unconstrained minimizer, is in error. We first formally describe all such minterms and distinguish the types of error that they exhibit.

Definition 3.1. DIFF minterm and DIFF set. A minterm x on which $F(x) \neq F_M(x)$ is called a *difference* minterm and is referred to as the *DIFF* minterm. The difference set for a function, which we call the *DIFF* set, contains all *DIFF* minterms regardless of the type of error.

Definition 3.2. Error types. We designate the error type by ET. If for a given minterm and for a single output bit, $F(x) = 0$ and $F_M(x) = 1$, the error is of the $0 \rightarrow 1$ type. It is encoded as a two-bit value $ET = 01$. If for a given minterm and for a single output bit, $F(x) = 1$ and $F_M(x) = 0$, the error is of the $1 \rightarrow 0$ type. It is encoded as as two-bit value $ET = 10$. If there is no error, $ET = 00$. Let $ET_{i,j}$ be the two-bit encoding of the error type for an output bit i on the minterm j. Let CET_j be the concatenation of $ET_{i,j}$ for $i = 1$ to k, where k is the number of outputs. CET_j encodes the entire error pattern for function F on the minterm j.

Example 3.1. We illustrate the definitions with an example below. The shaded minterms $x_1'x_0'$, x_1x_0' in Table 1 form the $DIFF$ set for the 2-input, 2-output Boolean function. Minterm $x_1'x_0'$ has an error on the output bit y_0, which is an $ET = 01$; while there is no error for y_1 on this minterm. Minterm x_1x_0' has errors on both y_1 and y_0 output bits, where y_1 has the $ET = 01$ error and y_0 has the $ET = 10$ error.

Table 1: Example of DIFF minterms (shaded).

F		F_M		CET
x_1x_0	y_1y_0	x_1x_0	y_1y_0	y_1y_0
00	{00}	00	{01}	$\{ET = 00, ET = 01\}$
01	{01}	01	{01}	$\{ET = 00, ET = 00\}$
10	{01}	10	{10}	$\{ET = 01, ET = 10\}$
11	{10}	11	{10}	$\{ET = 00, ET = 00\}$

The algorithm we construct seeks to find $F_{M,R}$ by enforcing correctness on some of the minterms of F that have been modified by the solution to the Boolean relations problem. The key part of the algorithm is therefore the notion of correcting the function on a given minterm. To correct an $ET = 01$, the minterm needs to be moved from the ON-set of the function for this output bit back to the OFF-set. We call this a *correct-to-0* change. To correct an $ET = 10$, the minterm needs to be moved from the OFF-set of the function for this output bit to the ON-set, which we call a *correct-to-1* change. The result of minterm correction is a change in the literal count in the cover of the function $F_{M,r}$. It should be noted that both types of corrections may result in a literal count increase. Also, note that at an equal literal count increase, the algorithm will accept both types of corrections equally as long as the magnitude of error is not increased.

The algorithm we develop is greedy and gradually identifies the best minterms to correct. One possible approach is to correct one DIFF minterm at a time by selecting a minterm that causes the least literal cost increase. However, this is sub-optimal. Instead, our algorithm is based on the principle that *at each step the largest number of DIFF minterms should be corrected for the minimum available literal increase.*

The central challenge of the algorithm is in identifying the optimal *changes* to the ON/OFF-sets of the function such that the cover is minimized. (Note the difference between the conventional logic minimization (LM) and the above problem. Conventional LM is to find the minimum cover for given ON/OFF-sets. Our problem is its dual and seeks to find the optimal *change* to the ON/OFF-sets for a minimum cover increase.)

First, consider a single-output function F. The set of possible correction decisions, which is represented by the DIFF set, can be represented separately by a pair of correction functions: one for *correct-to-1* and one for *correct-to-0*, where a correction function is 1 iff the given minterm is a member of the corresponding DIFF set and thus a candidate for correction. We define the *correct-to-1* function $CT1$ by the set of its minterms, which are the DIFF set minterms with $ET = 10$. Correspondingly, we define the *correct-to-0* function $CT0$ by the set of its minterms, which are the DIFF set minterms with $ET = 10$.

The key aspect of the algorithm is the idea that the identification of minterms to correct should proceed by first constructing a minimial cover for the two correction functions ($CT0$, $CT1$) and by using the prime implicants (PIs) of the covers to seek optimal changes to the ON/OFF-sets of the function. We call the prime implicants of the minimum cover of a correction function the **DIFF primes**.

The following notion of *cost* is used to compare the effectiveness of correcting a specific DIFF prime j of a function:

$$cost_j = \frac{\text{literal increase due to correction of DIFF prime } j}{\text{number of minterms covered by DIFF prime } j} \quad (5)$$

The greedy decision-making is driven by selecting at every iteration the best decision understood as the decision with the least cost, as defined above. We can formalize this principle in the following theorem, which is proven later in the derivation after the function update strategy is fully explained:

Theorem 3.2. *For a single-output function F, the optimal set of minterms to add to the ON/OFF-set at the minimum literal increase in the cover of function $F_{M,r}$ lies among the prime implicants of the minimum cover of correction functions $CT0$ and $CT1$.*

The above results can be extended to multi-output functions and their corresponding correction functions. An important aspect of the allowed corrections for multi-output functions is that the magnitude of error cannot be increased. This can be guaranteed only if either the entire function is corrected on a given minterm or the entire output is not modified at all. This constraint, combined with the result of Theorem 3.2 that directs us to seek optimal decisions among the minimum covers, leads to us to define the correction function for a multi-output case *not by the individual DIFF minterms but by subsets of DIFF minterms.* The sub-set, referred to as the DIFF group, is defined as:

Definition 3.3. DIFF group A DIFF group is a set of all DIFF minterms with identical CET.

Example 3.2. Table 2 shows an example of grouping the DIFF minterms, where four DIFF minterms are grouped into three DIFF groups.

Table 2: Example of DIFF groups.

DIFF min.	F	F_M	CET	Group
x_1x_0	y_1y_0	y_1y_0	y_1y_0	#
00	{01}	{10}	$\{ET = 01, ET = 10\}$	1
01	{00}	{11}	$\{ET = 01, ET = 01\}$	2
10	{10}	{00}	$\{ET = 10, ET = 00\}$	3
11	{11}	{01}	$\{ET = 10, ET = 00\}$	3

The correction function for a multi-output case is defined in the same way as before, i.e., by its constituents DIFF minterms. In this case, the minterms of a correction function belong to the same DIFF group. Each group contains minterms with identical error behavior on all outputs and thus logic minimization of each correction function individually allows us to find the least cost ways of carrying out the same change to function $F_{M,r}$. The result of the above definition is that for a multi-output function with k outputs, we may have up to 3^n distinct correction functions.

Each correction function is minimized using 2-level Boolean minimization. We use the standard Boolean minimization tool ESPRESSO to generate a minimum cover of all DIFF minterms within each correction function. Algorithm 1 summarizes the procedure of getting the DIFF groups and DIFF primes.

Example 3.3. See Table 3, where the shaded DIFF prime covers the two shaded DIFF minterms in Table 2.

Table 3: Example of DIFF primes.

DIFF prime	CET	Group
x_1x_0	y_1y_0	#
00	$\{ET = 01, ET = 10\}$	1
01	$\{ET = 01, ET = 01\}$	2
1-	$\{ET = 10, ET = 00\}$	3

Algorithm 1: Correction Function Minimization.

Input: Frequency unconstrained approximate function F_M
Output: DIFF primes for every correction function

```
// identify DIFF minterms and their error
   structure
foreach minterm in F do
    foreach output bit j do
        compare F and F_M;
        record error type ET at bit j: 00, 01, 10;
    end
end
// determine DIFF groups and correction
   functions they define
group all DIFF minterms with identical error behavior;
// determine the DIFF primes
foreach DIFF group do
    call ESPRESSO to minimize the correction function for
    this DIFF group and return DIFF primes;
end
```

3.3 Function Updates and Cost Calculation

The algorithm repeatedly eliminates the best candidate DIFF primes in the current DIFF set and modifies the function $F_{M,r}$. The following sequence is thus executed repeatedly: (1) the best correction is identified, (2) the ON/OFF-sets of function $F_{M,r}$ are updated, and (3) all correction functions impacted by the current change are updated. The multi-output $F_{M,r}$ is algorithmically treated as a union of single-output functions whose ON/OFF-sets are defined individually. However, the procedure outlined in the previous section means that when the best DIFF prime is selected for a correction, the function $F_{M,r}$ needs to be updated *on all of its outputs*.

In the following, we discuss along with the update strategy a related issue of efficient cost computation. The restriction of the search space to the primes of correction functions reduces the number of possible solutions. Despite that, we still need to evaluate candidates based on the specific increase in the literal count they produce.

The denominator of Equation (5) refers to how many DIFF minterms are simultaneously corrected by correcting the single DIFF prime j. It is easily computed as 2^{n-s}, where n is the number of input variables in function F and s is the number of literals in the DIFF prime j.

Unfortunately, evaluating the numerator is difficult. Only after we complete Boolean minimization on the updated $F_{M,r}$ can we know the literal changes exactly, where a function update is the update of the ON/OFF-sets of $F_{M,r}$ for all outputs that are prescribed by the correction function currently being evaluated. However, since the cost computation needs to be done often, running a 2-level minimizer for each evaluation is too expensive in terms of computation time. To address this issue, we propose a proxy metric for estimating literal changes. One approximation that our proxy metric adopts is that the literal cost increase is the sum of literal cost increases for each output individually. In other words, an n-output function is treated as a collection of n single-output functions. This is a conservative

assumption that ignores the sharing of terms in the covers of multi-output functions.

Before we describe the details of our function update strategy, we need to introduce a basic encoding scheme for performing operations on prime implicants.

Definition 3.4. Positional-Cube Notation. The *positional-cube* notation is a binary encoding of implicants. The symbols used in the input part are {0,1,-}. The positional-cube notation encodes each symbol by 2-bit *fields* as follows:

Ø	00
0	10
1	01
-	11

where the symbol Ø means none of the allowed symbols, i.e. the presence of Ø means this implicant is void and should be removed.

The proxy computation to estimate the corresponding literal changes depends on whether the candidate update is a *correct-to-0* or a *correct-to-1* update. We first discuss the *correct-to-0* update strategy. Consider estimating the literal changes for a candidate DIFF prime p_i^{dif}. First, we identify the subset of primes in the current cover of $F_{M,r}$ that has a non-zero intersection (where the **intersect** operation is defined in Def. 3.5) with p_i^{dif}. Let this subset be P^f and denote each specific prime in this subset as p_j^f, where the uppercase P indicates a set, and lowercase p indicates a single prime. Let $p_j^{intr} = p_i^{dif} \cap p_j^f$ be the result of each intersection.

Definition 3.5. Intersection of two implicants. The *intersection* of two implicants is the largest cube contained in both. It is computed by the bitwise product using a positional-cube encoding. If the result contains Ø, i.e. a void implicant, the two implicants do not intersect.

To perform the update, we modify all the primes in P^f, since they are modified after we remove the candidate DIFF prime. For a *correct-to-0* update, we need to keep all and only those minterms covered by the p_j^f and not p_j^{intr}. This can be done by performing a **sharp** operation (defined in Def. 3.6) on p_j^f and p_j^{intr}. The resulting prime(s) replace p_j^f as the new prime(s). If the resulting prime is void, i.e. if $p_j^f = p_j^{intr}$, then p_j^f is removed entirely.

Definition 3.6. Sharp Operation. The *sharp* operation, when applied to two implicants, returns a set of implicants covering all and only those minterms covered by the first one and not by the second one. The sharp operator is denoted by #. Let $\alpha = a_1 a_2 \dots a_n$, and $\beta = b_1 b_2 \dots b_n$, where $a_i, b_i, i = 1, 2, \dots, n$, represents their fields. The sharp operation can be defined as:

$$\alpha \# \beta = \begin{cases} a_1 b_1' & a_2 & \dots & a_n \\ a_1 & a_2 b_2' & \dots & a_n \\ \dots & \dots & \dots & \dots \\ a_1 & a_2 & \dots & a_n b_n' \end{cases} \quad (6)$$

Given the replacement of primes p_j^f with the results of the sharp operation, let N be the number of inputs in function F, d_j be the cardinality of p_j^f *XOR* p_j^{intr}, and M_j be the number of variables for p_j^f. Then, the literal change δL_j^{01} to correct an error of type $ET = 01$ on a single output is:

$$\delta L_j^{01} = (d_j - 1) \times M_j + d_j \quad (7)$$

Proof. Let the set of literals corresponding to p_j^f be X. Then, p_j^{intr} must have the form $X a_1 a_2 \dots a_d$. (Obviously, $p_j^{intr} \subset p_j^f$, i.e. there are more variables in p_j^{intr} than in p_j^f). Then the Boolean subtraction of $X - X a_1 a_2 \dots a_d$ reduces to $X a_1' + X a_2' + \dots + X a_d'$. By counting the literal changes before and after the Boolean subtraction, we get Equation (7). □

To estimate the total change in the literal count due to elimination of the DIFF prime p_i^{dif}, we sum up individual costs for every output that has an error of type $ET = 01$. Let h be the number of outputs with error of type $ET = 01$ and let the cardinality of P^f be l:

$$\Delta L_i^{01} = \sum^{h} \sum^{l} \delta L_j^{01} \quad (8)$$

We now discuss the update strategy for a *correct-to-1* update and describe a way to efficiently estimate the literal changes after adding a new prime to $F_{M,r}$. We start by finding a subset of primes P^f of $F_{M,r}$ that are **adjacent** to the DIFF prime p_i^{dif}. Adjacency is an important criterion since it indicates that the selected primes can be merged to a larger prime (i.e., a prime with fewer literals). The adjacency information is acquired by using an **intersect** operator on p_i^{dif} and p_j^f. If the result contains only one empty field (Ø), then the two implicants are adjacent. If P^f is empty, i.e. there are no primes that are adjacent to the DIFF prime p_i^{dif}, then the literal change δL_j^{10} is equal to the number of literals in p_i^{dif} itself. However, the when P^f is not empty two possibilities exist:

- p_i^{dif} and p_j^f together form a new single prime, which reduces the current literal counts;
- p_i^{dif} becomes larger (has fewer literals) due to the adjacency with p_j^f;

Therefore, we need to count the literal changes by selecting *one* p_j^f that causes the minimum literal increase out of all primes in this P^f. To compute the literal increase for each pair of p_j^f and P^f, we evaluate the literals of the **consensus** of the two primes, which we denote by p_j^{consen}. Because the two primes intersect, there is only a single implicant for the consensus operation, which is defined in Def. 3.7. Then, the literal increase is computed as:

$$\delta L_j^{10} = L(p_j^{consen}) \quad (9)$$

Definition 3.7. Consensus Operation. The *consensus* operation is defined as follows. The consensus returns void when the two implicants have a distance larger than or equal to 2. The consensus returns a single implicant when the two implicants are adjacent. The consensus returns more than or equal to 2 implicants, when the two implicants are intersecting. The consensus operator is denoted by ∇.

$$\alpha \nabla \beta = \begin{cases} a_1 + b_1 & a_2 b_2 & \dots & a_n b_n \\ a_1 b_1 & a_2 + b_2 & \dots & a_n b_n \\ \dots & \dots & \dots & \dots \\ a_1 b_1 & a_2 b_2 & \dots & a_n + b_n \end{cases} \quad (10)$$

The overall literal change due to a *correct-to-1* update is the sum of costs for all output bits with error type $ET = 10$. Notice that for each of the output bits, we pick the minimum δL_j^{10} out of all primes in P^f. For h output bits with error type $ET = 10$, the overall literal increase as a result of an update due to p_i^{dif} is:

$$\Delta L_i^{10} = \sum^{h} \min \{\delta L_j^{10}\} \quad (11)$$

Once we identify the p_j^{consen} with the minimum literal increase, we add the resulting consensus prime to the erroneous output bit function of $F_{M,r}$. Importantly, if p_j^f is covered by p_j^{consen}, it is removed. This happens under the first scenario considered above for the case of P^f not being empty.

The overall literal change for a candidate DIFF prime p_i^{dif} is the sum of ΔL_i^{01} and ΔL_i^{10}:

$$\Delta L_i = \Delta L_i^{01} + \Delta L_i^{10} \tag{12}$$

So far we discussed the proposed proxy metrics to estimate the literal changes due to the DIFF prime corrections. After eliminating a candidate DIFF prime, we simultaneously modify the corresponding primes in $F_{M,r}$. This change affects the *cost* values of the remaining primes. Thus, we need to update the *cost* values of all remaining DIFF primes that are *impacted* by the just-modified prime. To achieve this, we store dependency information for each prime of $F_{M,r}$ that records all DIFF primes that use this prime to compute their cost values. Once a prime of $F_{M,r}$ is modified, we immediately get a list of the associated DIFF primes for which cost updates are needed.

We are now in a position to prove Theorem 3.2, which we repeat here for the ease of reading.

Theorem. *For a single-output function F, the optimal set of minterms to add to ON/OFF-set at the minimum literal increase in the cover of function $F_{M,r}$ lies among the prime implicants of the minimum cover of correction functions $CT0$ and $CT1$.*

Proof. Consider a correction function $CT1$ and its minimum cover. Let p_j be a prime in that cover. Let $p_j \ni M_j = \{m_1, m_2, \ldots, m_h\}$ be the set of minterms covered by p_j. The theorem is true if the literal increment of correcting any subset of M_j is larger than correcting the entire M_j, i.e. the p_j. Letting $lit(\cdot)$ be the literal number in a cube, it is clear that $lit(p_j) < lit(M_{j,i})$ for any i. For p_j that has error type $ET = 10$, the literal increment is smaller when adding p_j to ON-set rather than any subset of M_j.

Now consider a correction function $CT0$ and its minimum cover. Let p_j be a prime in that cover. To correct an error of type $ET = 01$, the minterms covered by p_j are to be removed from the cover of $F_{M,r}$. For the prime implicants of $F_{M,r}$ that have non-zero intersection with p_j we have the following: the larger is the intersection between p_j and primes of $F_{M,r}$, the fewer non-overlapping variables there are between p_j and a prime of $F_{M,r}$. This results in smaller d_j in Equation (7) and hence a smaller literal increment. □

A complete description of GALS algorithm is in Algorithm 2.

Algorithm 2: Approximate logic synthesis algorithm.

Input: frequency-unconstrained approximate Boolean function
Output: minimized boolean function with constrained error magnitude and error frequency

```
// get the DIFF primes
call "Correction Function Minimization" subroutine;
// initialize the current solution, k is the
   initial error frequency by BR solver
F_{M,r} = F_{M,k};
// get the initial error count for F_{M,r}
ErrorCount = k;
if ErrorCount ≤ Error Frequency Constraint then
    return F_{M,r};
end
// initialize the Cost-List
foreach DIFF prime p_i do
    compute the cost and push p_i to Cost-List;
    foreach prime in F_{M,r} that are associated with p_i do
        push p_i to association list of this prime;
    end
end
sort Cost-List by cost values with ascending order;
// main loop
while ErrorCount > Error Frequency Constraint do
    pop the DIFF prime with least cost value in Cost-List;
    modify the F_{M,r} after eliminating this DIFF prime;
    update all associated DIFF primes due to modifying the
    F_{M,r};
    Sort Cost-List by cost values with ascending order;
    update ErrorCount;
end
return F_{M,r};
```

4. Experimental Results

We have implemented GALS in a C++ environment using BREL [2] as the embedded BR solver engine for the first phase of the algorithm. To evaluate the capability of GALS for significant literal reductions under general magnitude and frequency constraints, we have used GALS to generate a range of approximate solutions of adders and multipliers. All experiments were performed on an Intel 3.4GHz Core i7 workstation.

We first demonstrate the basic operation of the algorithm on a simple 2-bit adder example with 4 inputs (a_1, a_0 and b_1, b_0) and 3 outputs (sum bits S_0 and S_1 and the carry C). Figure 1 shows the resulting logic equations for the exact adder (F), the frequency unconstrained solution (F_M), and both frequency and magnitude-constrained ($F_{M,R}$) approximate adder variants. In all cases, we applied a magnitude constraint of $M = 1$. We applied frequency constraints of one or two erroneous outputs out of the $2^4 = 16$ total minterms, i.e., $R = 1/16 = 6.25\%$ or $R = 2/16 = 12.5\%$. As expected, the frequency unconstrained solution (F_1) has the smallest literal count. The expression complexity increases with a decreasing frequency constraint. It is interesting to point out that the evolution of the logic does not follow an obvious pattern.

To further evaluate the effectiveness of the second-phase of GALS based on Theorem 3.2, which operates with the primes of the correction functions, we compare its performance against an alternative implementation that greedily corrects only the single best minterm in each iteration. We refer to this alternative implementation as Single-GALS. Figure 2 plots the literal reductions achieved

$$F_1 = \begin{cases} C &= a_1 b_1; \\ S_1 &= a_1 b_1' + a_1' b_1; \\ S_0 &= 1; \end{cases}$$

$$F_{1,\frac{2}{16}} = \begin{cases} C &= a_1 b_1; \\ S_1 &= a_1 b_1' + a_1' b_1 + a_0 b_0; \\ S_0 &= a_1 b_1' b_0 + a_1' b_1 b_0 + a_0 b_0' + a_0' b_0; \end{cases}$$

$$F_{1,\frac{1}{16}} = \begin{cases} C &= a_0 b_1 b_0 + a_1 b_1; \\ S_1 &= a_1' b_1 b_0' + a_1' a_0' b_1 + a_0 b_1' b_0 \\ &\quad + a_1 a_0 b_0 + a_1 b_1'; \\ S_0 &= a_1 b_1' b_0 + a_0 b_0' + a_0' b_0; \end{cases}$$

$$F = \begin{cases} C &= a_0 b_1 b_0 + a_1 a_0 b_0 + a_1 b_1; \\ S_1 &= a_1' a_0 b_1' b_0 + a_1 a_0 b_1 b_0 + a_1 b_1' b_0' \\ &\quad + a_1' b_1 b_0' + a_1 a_0' b_1' + a_1' a_0' b_1; \\ S_0 &= a_0 b_0' + a_0' b_0; \end{cases}$$

Figure 1: Synthesized 2-bit adder variants.

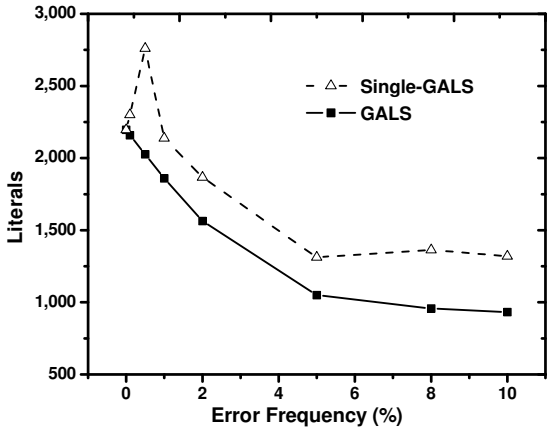


Figure 2: Effectiveness of GALS for 6-bit adders with a magnitude constraint of $M = 1$.

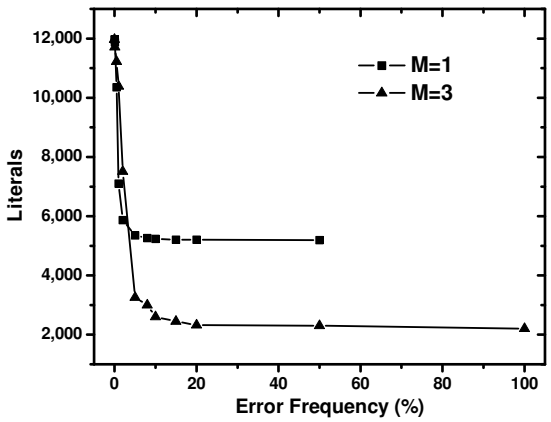


Figure 3: Synthesis results for 8-bit adders by GALS.

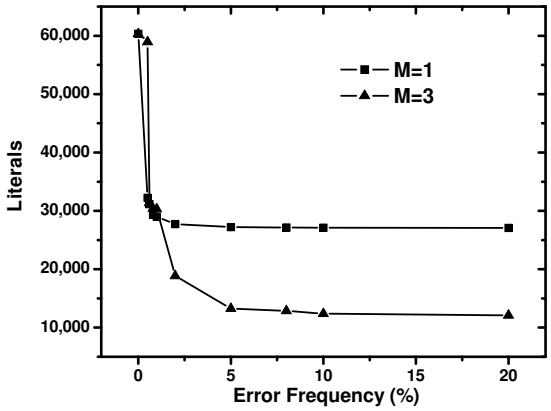
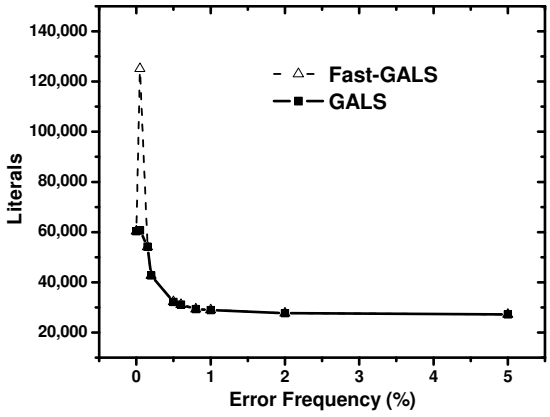


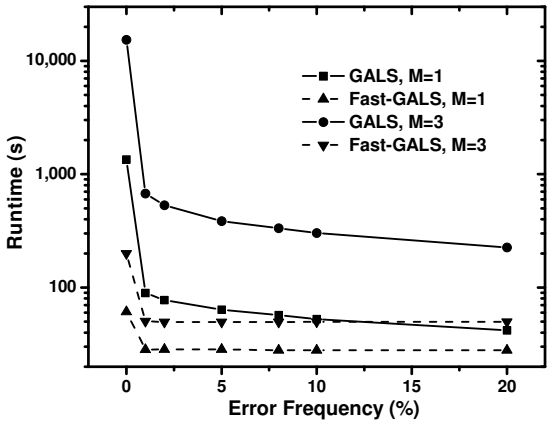
Figure 4: Synthesis results for 10-bit adders by GALS.

by both algorithms when applied to a 6-bit adder with a magnitude constraint of $M = 1$ and varying frequency constraints. Results validate the effectiveness of the proposed strategy: our algorithm substantially outperforms the naive approach. On average, it produces 20% fewer literals while also being 37x faster.

Next, we use GALS to synthesize 8-bit and 10-bit approximate adders under magnitude constraints of $M = 1$ and $M = 3$ (Figures 3 and 4). Independent of the adder size, the frequency unconstrained solution at the output of the first phase BR solver results in an error frequency of 50% and 100% at literal reductions of around



(a) Literals for 10-bit adders with $M = 1$.



(b) Runtime for 10-bit adders.

Figure 5: Comparison of GALS and Fast-GALS algorithms.

55% and 80% for $M = 1$ and $M = 3$, respectively. Results after further constraining error frequencies using GALS show that similar literal reductions can be maintained all the way down to error rates as low as 1-2%. Note that at extremely tight frequency constraints, literal counts of synthesized solutions for $M = 3$ grow faster than those for $M = 1$. We conjecture that this is caused by the use of proxy cost metric in the second phase of GALS and the resulting sub-optimality of the greedy decision-making. We note that this effect is limited to only very small frequencies (below 0.6% for the 10-bit adder).

Runtimes for the first-phase BR solver range between 1s and 5s for 8-bits and between 50s and 2.5m for the 10-bit adder. Runtimes of the second-phase of GALS range between 2s and less than 5m for 8-bit adders, and between 30s and more than 3h for 10-bit designs. To further reduce runtime, we investigated the use of a speed-up technique. One of the computationally expensive steps in GALS is the cost-updating routine that is repeatedly executed in the main loop of the algorithm (lines 17 - 18 in Algorithm 2). We find that using the Cost-List that is initialized once but is not updated on every iteration leads, in most cases, to a relatively small loss of optimality in the choice of a DIFF prime to be removed. Yet the runtime of the second-phase of the algorithm can be reduced significantly. We developed such a Fast-GALS algorithm. Results of the comparison between GALS and Fast-GALS for the 10-bit adder are shown in Figure 5. We observe that in most cases, resulting literal counts are very close while the runtime of Fast-GALS is one order of magnitude lower than GALS (Figure 5b). At tight frequency constraints, however, the cost updating mechanism plays a vital role. As a result, at very low frequencies, Fast-GALS produces solutions substantially worse

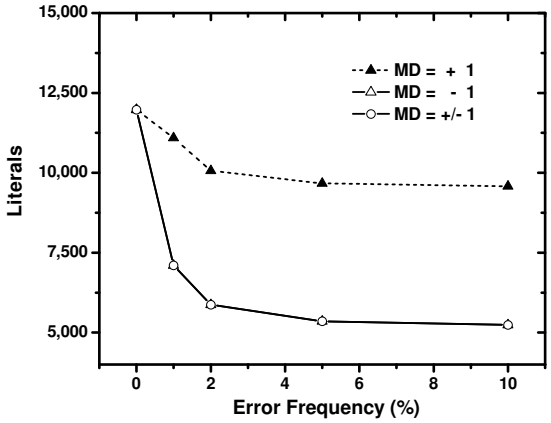


Figure 6: 8-bit adders under different error directions by GALS.

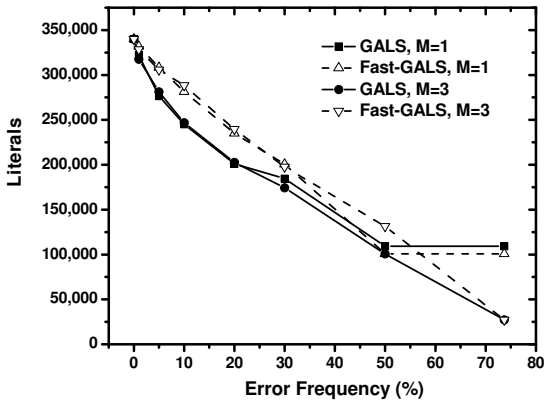


Figure 7: Synthesis results for 8-bit truncated multipliers.

than those of GALS, and in some cases even worse than the exact solution.

Depending on the application, not only error magnitude but also the error direction can be of importance. GALS supports setting different output relations during the first BR solving phase. We constructed experiments on 8-bit adders in which the direction of error is further constrained to be only positive or negative. Results are shown in Figure 6, where *MD* is the value of the allowed adder error. It can be observed that for addition logic, allowing negative errors (exclusively or combined in both directions) results in circuits that are synthesized to have smaller literals.

Finally, we applied GALS and Fast-GALS algorithms to the larger test-case: an 8-bit truncated multiplier (Figure 7). Runtimes for the first-phase BR solver range between 4m and 5m. Runtimes for the two algorithms range between 20m and 3.3h for GALS and between 5m and 13m for Fast-GALS. Fast-GALS produces solutions that can be up to 20% worse in terms of literal count to the ones obtained with the slower GALS algorithm. Note that in the case of the multiplier, there is a significant dependence of literal count on error frequency over nearly entire range of frequencies. This further motivates the need for an application-specific synthesis solution.

5. Conclusions

In this paper, we presented a heuristic approach for solving a general approximate logic synthesis problem. We first address the error magnitude-only constrained problem by casting it to a Boolean relation minimization, which is solved using recently proposed fast algorithms. The frequency-constrained problem is further solved by a novel greedy algorithm that finds the optimal set of function minterms on which the exact outputs must be enforced, and systematically corrects erroneous outputs until a given error frequency constraint is met. The proposed algorithm is capable of synthesizing approximate circuits for arbitrarily specified error deviations, and is most immediately applicable to arithmetic blocks, for which experiments demonstrate the effectiveness in achieving significantly reduced literal counts across a wide range of flexible error frequency and magnitude constraints.

Acknowledgements

This work was supported by NSF grant CCF-1018075.

References

[1] P. Albicocco, G. C. Cardarilli, A. Nannarelli, M. Petricca, and M. Re. Imprecise arithmetic for low power image processing. In *Signals, Systems and Computers (ASILOMAR), 2012*.

[2] D. Baneres, J. Cortadella, and M. Kishinevsky. A recursive paradigm to solve boolean relations. In *DAC*, 2004.

[3] R. Brayton and F. Somenzi. An exact minimizer for boolean relations. In *ICCAD*, 1989.

[4] L. Chakrapani and K. Palem. A probabilistic boolean logic for energy efficient circuit and system design. In *ASP-DAC*, 2010.

[5] V. Chippa, A. Raghunathan, K. Roy, and S. Chakradhar. Dynamic effort scaling: Managing the quality-efficiency tradeoff. *DAC*, 2011.

[6] A. Ghosh, S. Devadas, and A. Newton. Heuristic minimization of boolean relations using testing techniques. In *ICCD*, 1990.

[7] V. Gupta, D. Mohapatra, S. P. Park, A. Raghunathan, and K. Roy. IMPACT: imprecise adders for low-power approximate computing. In *ISLPED*, 2011.

[8] K. He, A. Gerstlauer, and M. Orshansky. Controlled timing-error acceptance for low energy idct design. In *DATE*, 2011.

[9] R. Hegde and N. Shanbhag. Soft digital signal processing. *TVLSI01*.

[10] S.-W. Jeong and F. Somenzi. A new algorithm for the binate covering problem and its application to the minimization of boolean relations. In *ICCAD*, 1992.

[11] F. Kurdahi, A. Eltawil, K. Yi, S. Cheng, and A. Khajeh. Low-power multimedia system design by aggressive voltage scaling. *TVLSI10*.

[12] B. Lin and F. Somenzi. Minimization of symbolic relations. In *ICCAD90*.

[13] A. Lingamneni, C. Enz, J. L. Nagel, K. Palem, and C. Piguet. Energy parsimonious circuit design through probabilistic pruning. In *DATE2011*.

[14] S.-L. Lu. Speeding up processing with approximation circuits. *Computer*, 2004.

[15] P. McGeer, J. Sanghavi, R. Brayton, and A. Vincentelli. Espresso-signature: A new exact minimizer for logic functions. In *DAC*, 1993.

[16] J. Miao, K. He, A. Gerstlauer, and M. Orshansky. Modeling and synthesis of quality-energy optimal approximate adders. In *ICCAD12*.

[17] G. D. Micheli. *Synthesis and Optimization of Digital Circuits*. McGraw-Hill Higher Education, 1994.

[18] S. H. Nawab, A. V. Oppenheim, A. P. Chandrakasan, J. M. Winograd, and J. T. Ludwig. Approximate signal processing. *VLSI Signal Processing*, 15, 1997.

[19] D. Shin and S. K. Gupta. Approximate logic synthesis for error tolerant applications. In *DATE*, 2010.

[20] S. Venkataramani, A. Sabne, V. Kozhikkottu, K. Roy, and A. Raghunathan. Salsa: systematic logic synthesis of approximate circuits. In *DAC*, 2012.

[21] Y. Watanabe and R. Brayton. Heuristic minimization of multiple-valued relations. *TCAD*, 1993.

[22] T. Xanthopoulos and A. Chandrakasan. A low-power dct core using adaptive bitwidth and arithmetic activity exploiting signal correlations and quantization. *J. Solid-State Circuits*, 2000.

[23] N. Zhu, W. L. Goh, W. Zhang, K. S. Yeo, and Z. H. Kong. Design of low-power high-speed truncation-error-tolerant adder and its application in digital signal processing. *TVLSI*, 2010.