Automated Least-Significant Bit Datapath Optimization for FPGAs

Mark L. Chang and Scott Hauck
Department of Electrical Engineering
University of Washington
Seattle, Washington
Email: {mchang,hauck}@ee.washington.edu

Abstract— In this paper we present a method for FPGA datapath precision optimization subject to user-defined area and error constraints. This work builds upon our previous research [1] which presented a methodology for optimizing for dynamic range—the most significant bit position. In this work, we present an automated optimization technique for the least-significant bit position of circuit datapaths. We present results describing the effectiveness of our methods on typical signal and image processing kernels.

I. INTRODUCTION

With the widespread growth of reconfigurable computing platforms in education, research, and industry, more software developers are being exposed to hardware development. Many are seeking to achieve the enormous gains in performance demonstrated in the research community by implementing their software algorithms in a reconfigurable fabric. For the novice hardware designer, this effort usually begins and ends with futility and frustration as they struggle with unwieldy tools and new programming paradigms.

One of the more difficult paradigm shifts to grasp is the notion of bit-level operations. On a typical FPGA fabric, logical and arithmetic operators can work at the bit level instead of the word level. With careful optimization of the precision of the datapath, the overall size and relative speed of the resulting circuit can be dramatically improved.

In this paper we present a methodology that broadens the work presented in [1]. We begin with background on precision analysis and previous research efforts. We describe the problem of least-significant bit optimization and develop several optimization techniques that provide finer control of area-to-error tradeoffs than more traditional methods. We then present a simulated annealing-based approach to automatically applying these optimizations to a datapath. Finally, we present the results of using our techniques to optimize the datapath of image processing circuits and draw some conclusions.

II. BACKGROUND

General-purpose processors are designed to perform operations at the word level, typically 8, 16, or 32 bits. Supporting this paradigm, programming languages and compilers abstract these word sizes into storage classes, or data-types, such as `char`, `int`, and `float`. In contrast, most mainstream reconfigurable logic devices, such as FPGAs, operate at the bit level. This allows the developer to tune datapaths to any word size desired. Unfortunately, choosing the appropriate size for datapaths is not trivial. Choosing a wide datapath, as in a general-purpose processor, usually results in an implementation that is larger than necessary. This consumes valuable resources and potentially reduces the performance of the design. On the other hand, if the hardware implementation uses too little precision, errors can be introduced at runtime through quantization effects, such as roundoff and truncation.

To alleviate the programmer's burden of doing manual precision analysis, researchers have proposed many different solutions. Techniques range from semi-automatic to fully-automated methods that employ static and dynamic analysis of circuit datapaths. We will touch on some of these efforts in the following section.

A. The Least-Significant Bit Problem

In determining the fixed-point representation of a floating-point datapath, we must consider both the most-significant and least-significant ends. Reducing the relative bit position of the most-significant bit reduces the maximum value that the datapath may represent, sometimes referred to as the dynamic range. On the other end, increasing the relative bit position of the least-significant bit (toward the most-significant end) reduces the maximum precision that the datapath may attain. For example, if the most-significant bit is at the 2^7 position, and the least-significant bit is at the 2^{-3} position, the maximum value attainable by an unsigned number will be $2^8 - 1 = 255$, while the precision will be quantized to multiples of $2^{-3} = 0.125$. Values smaller than 0.125 cannot be represented as the bits necessary to represent, for example, 0.0625, do not exist.

Having a fixed-point datapath means that results or operations may exhibit some quantity of error compared to their floating-point counterparts. This quantization error can be introduced in both the most-significant and least-significant sides of the datapath. If the value of an operation is larger than the maximum value that can be represented by the datapath, the quantization error is typically a result of truncation or saturation, depending on the implementation of the operation. Likewise, error is accumulated at the least-significant end of the datapath if the value requires greater precision than the

datapath can represent, resulting in truncation or round-off error.

Previous research includes [2], [3], which only performs the analysis on the most-significant bit position of the datapath. While this method achieves good results, it ignores the potential optimization of the least-significant bit position. Other research, including [4], [5] begin to touch on fixed-point integer representations of numbers with fractional portions. Finally, more recent research, [6], [7] begin to incorporate error analysis into the overall optimization of the fractional width of the datapath elements.

Most of the techniques introduced deal with either limited scope of problem, such as linear time-invariant (LTI) systems, and/or perform the analysis completely automatically, with minimal input from the developer. While again, these methods achieve good results, it is our belief that the developer should be kept close at hand during all design phases, as there are some things for which an automatic optimization method simply cannot handle.

Simply put, a "goodness" metric must be devised in order to guide an automatic precision optimization tool. This "goodness" function is then evaluated by the automated tool to guide its precision optimization. In some cases, such as image processing, a simple block signal-to-noise ratio (BSNR) may be appropriate. In many cases, though, this metric is difficult or impossible to evaluate programmatically. A human developer, therefore, has the benefit of having a much greater sense of context in evaluating what is an appropriate tradeoff between error in the output and performance of the implementation. We have used this idea as the guiding principle behind the design of our precision analysis tool Précis [1]. In this paper we provide the metrics and methodology for performing least-significant-bit optimization.

III. ERROR MODELS

The observation that the relative bit position of the least-significant bit introduces a quantifiable amount of error over a floating-point datapath is an important one. After performing the optimization for the most-significant bit position as described in [1], we must perform an area/error analysis phase to optimize the position of the least-significant bit. In order to quantify changes to the datapath, we introduce models for area and error estimation of a general island-style FPGA.

Consider an integer value that is M' bits in length. This value has an implicit binary point at the far right—to the right of the least-significant bit position. By truncating bits from the least-significant side of the word, we reduce the area impact of this word on downstream arithmetic and logic operations. It is common practice to simply truncate the bits from the least-significant side to reduce the number of bits required to store and operate on this word. We propose an alternate method—replace the bits that would normally be truncated with constants, in this case zeros. Therefore, for an M'-bit value, we will use the notation $A_m 0_p$. This denotes a word that has m correct bits and p zeros inserted to signify bits that
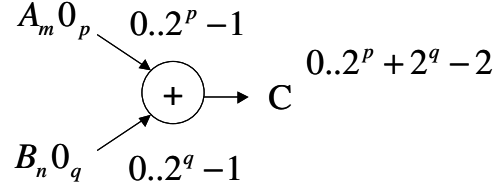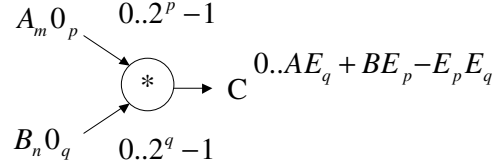

Fig. 1. Error model of an adder.

Fig. 2. Error model of a multiplier.

have been effectively truncated, resulting in an $M' = m+p$-bit word.

Having performed a reduction in the precision that can be obtained by this datapath with a substitution of zeros, we have introduced a quantifiable amount of error into the datapath. For an $A_m 0_p$ value, substituting p zeros for the lower portion of the word, gives us a maximum error of $2^p - 1$. This maximum error occurs when the bits replaced were originally *ones*, making this result too low by the amount $2^p - 1$. If the bits replaced were originally *zeros*, we will have incurred no error. We will use the notation $[0..2^p - 1]$ to describe this resultant error range that our substitution method produces.

This error model can be used to estimate the effective error of combining quantized values in arithmetic operators. To investigate the impact, we will discuss an adder and multiplier in greater detail.

A. Adder Error Model

An adder error model is shown in Fig. 1. The addition of two quantized values, $A_m 0_p + B_n 0_q$, results in an output, C, which has a total of $\max(M', N') + 1$ bits. Of these bits, $\min(p, q)$ of them are substituted zeros at the least-significant end. In an adder structure, the range of error for the output, C, is the sum of the error ranges of the two inputs, A and B. This gives us an output error range of $[0..2^p + 2^q - 2]$.

B. Multiplier Error Model

Just as we can derive an error model for the adder, we do the same for a multiplier. Again we have two quantized input values, $A_m 0_p * B_n 0_q$. These are multiplied together to form the output, C, which has a total of $M' + N'$ bits. Here, $p + q$ of them are substituted zeros at the least-significant end. This structure is shown in Fig. 2.

The output error is more complex in the multiplier structure than the adder structure. The input error ranges are the same, $[0..2^p - 1]$ and $[0..2^q - 1]$ for $A_m 0_p$ and $B_n 0_q$, respectively. Unlike the adder, multiplying these two inputs together requires

us to multiply the error terms as well, as shown in (1).

$$C = A * B$$
$$= (A - (2^p - 1)) * (B - (2^q - 1)) \qquad (1)$$
$$= AB - B(2^p - 1) - A(2^q - 1) + (2^p - 1)(2^q - 1)$$

The first line of (1) indicates the desired multiplication operation between the two input signals. Since we are introducing errors into each signal, line two shows the impact of the error range of A_m0_p by subtracting $2^p - 1$ from the error-free input A. The same occurs for input B.

Performing a substitution of $E_p = 2^p - 1$ and $E_q = 2^q - 1$ into (1) yields the simpler (2):

$$C = AB - BE_p - AE_q + E_pE_q$$
$$= AB - (AE_q + BE_p - E_pE_q) \qquad (2)$$

From (2) we can see that the range of error resulting on the output C will be $[0..AE_q + BE_p - E_pE_q]$. That is to say, the error that the multiplication will incur is governed by the actual correct values of A and B, multiplied by the error attained by each input. In terms of maximum error, this occurs when we consider the maximum attainable value of the inputs multiplied by the maximum possible error of the inputs.

IV. HARDWARE MODELS

In the previous section we derived error models for adder and multiplier structures. Error is only one metric upon which we will base optimization decisions. Another crucial piece of information is hardware cost in terms of area.

By performing substitution rather than immediate truncation, we introduce a critical difference in the way hardware will handle this datapath. Unlike the case of immediate truncation, we do not have to change the implementation of downstream operators to handle different bit-widths on the inputs. Likewise, we do not have to deal with alignment issues, as all inputs to operators will have the same location of the binary point.

For example, in an adder, as we reduce the number of bits on the inputs, the area requirement of the adder decreases. The same relationship holds true when we substitute zeros in place of variable bits on an input. This is true because we can simply use wires to represent static zeros or static ones, so the hardware cost in terms of area is essentially zero.

If the circuit is specified in a behavioral fashion using a hardware description language (HDL), this optimization is likely to fall under the jurisdiction of vendor tools such as the technology mapper and the logic synthesizer. Fortunately, this constant propagation optimization utilizing wires is implemented in most current vendor tools.

In the next sections we outline the area models used to perform area estimation of our datapath. We will assume a simple 2-LUT architecture for our target FPGA and validate this assumption through implementation on target hardware.
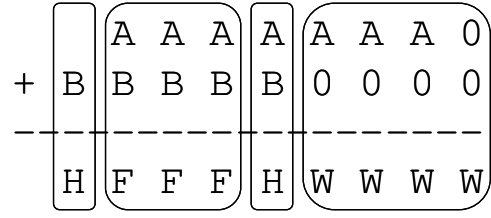

Fig. 3. Adder hardware requirements.

TABLE I
ADDER AREA

Number	Hardware
$\max(|M' - N'|, 0)$	half-adder
$\max(M', N') - \max(p, q) - |M' - N'| - 1$	full-adder
1	half-adder
$\max(p, q)$	wire

A. Adder Hardware Model

In a 2-LUT architecture, a half-adder can be implemented with a pair of 2-LUTs. Combining two half-adders together and an OR gate to complete a full-adder requires five 2-LUTs. To derive the hardware model for the adder structure as described in previous sections, we utilize the example shown in Fig. 3.

Starting at the least-significant side, all bit positions that overlap with zeros need only wires. The next most significant bit will only require a half-adder, as there can be no carry-in from any lower bit positions, as they are all wires. For the rest of the overlapping bit positions, we require a regular full-adder structure, complete with carry propagation. Finally, at the most-significant end, if there are any bits that do not overlap, we require half-adders to add together the non-overlapping bits with the possible carry-out from the highest overlapping full-adder bit.

The relationship described in the preceding paragraph is generalized into Table I, using the notation previously outlined. For the example in Fig. 3, we have the following formula to describe the addition.

$$A_m0_p + B_n0_q$$
$$m = 7, p = 1, n = 5, q = 4$$

This operation requires two half-adders, three full-adders, and four wires. In total, 19 2-LUTs.

B. Multiplier Hardware Model

We use the same approach to characterize the multiplier. A multiply consists of a multiplicand (top value) multiplied by a multiplier (bottom value). The hardware required for an array multiplier consists of AND gates, half-adders, full-adders, and wires. The AND gates form the partial products, which in turn are inputs to an adder array structure as shown in Fig. 5.

Referring to the example in Fig. 4, each bit of the input that has been substituted with a zero manipulates either a row or column in the partial product sum calculation. For each bit of
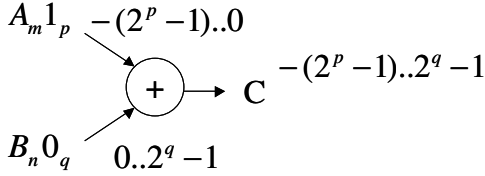
```
          A   A   A   0
      x   B   B   0   0
      ---------------
          A0  A0  A0  00
      A0  A0  A0  00
      AB  AB  AB  0B
  +   AB  AB  AB  0B
  -------------------------
```

Fig. 4.   Multiplication example.

Fig. 5.   Multiplication structure.

the multiplicand that is zero, we effectively remove an inner column. For each bit of the multiplier that is zero, we remove an inner row. Thus:

$$A_m0_p * B_n0_q$$
$$m = 3, p = 1, n = 2, q = 2$$

is effectively a 3x2 multiply, instead of a 4x4 multiply. This requires two half-adders, one full-adder, and six AND gates, for a total of 15 2-LUTs. This behavior has been generalized into formulas shown in Table II.

C. Model Verification

To verify our hardware models against real-world implementations, we implemented both the adder and multiplier structures in Verilog on the Xilinx Virtex FPGA using vendor-supplied place and route tools.

For the adder structure, we observe in Fig. 6 that our model closely follows the actual implementation area, being at worst within two percent of the actual Xilinx Virtex hardware implementation. The number of bits substituted was the same for each input at each data point.

TABLE II

MULTIPLIER AREA

Number	Hardware
$\min(m, n)$	half-adder
$mn - m - n$	full-adder
mn	AND
$p + q$	wire

Fig. 6.   Adder model verification.

Fig. 7.   Multiplier model verification.

The multiplier in Fig. 7 has a similar result to the adder, being at worst within 12 percent of the Xilinx Virtex implementation. These results support the use of our simple 2-LUT approximation of general island-style FPGAs to within a reasonable degree of accuracy.

V. OPTIMIZATION METHODS

Using the models described in the previous sections, we can now quantify the tradeoffs between area and error of various optimization methodologies.

A. The Nature of Error

Looking at the typical error introduced into a data path using the standard method of simple truncation, we see that the error is skewed, or biased, only in the positive direction. As we continue through datapath elements, if we maintain the same truncation policy to reduce the area requirement of our circuits, our lower-bound error will remain zero while our upper bound will continue to skew toward larger and larger positive values. This behavior also holds true for our own zero-substitution policy in Fig. 1 and Fig. 2.

This error profile does not coincide with our natural understanding of error. In most cases we consider the error of a result to be the *net distance from the correct value*,

$A_m 1_p$ $-(2^p-1)..0$

C $-(2^p-1)..2^q-1$

$B_n 0_q$ $0..2^q-1$

Fig. 8. Normalized error model of an adder.

$A_m 0_p$ $0..2^p-1$

C $-\dfrac{E_q}{2}(2A-E_p)..\dfrac{E_p}{2}(2B+E_q)$

$B_n 1_q$ $-(2^q-1)..0$

Fig. 9. Normalized error model of a multiplier.

$A_m 0_p$ $0..2^p-1$

$0..2^p+2^q-2$

C

$-(2^q-1)..2^p-1$

$B_n 0_q$ $0..2^q-1$

C

1_q

Fig. 10. Inserting a constant add performs an "active renormalization".

implying that the error term can be either positive or negative. Unfortunately, neither straight truncation nor our zero-substitution policy, as defined in previous sections, matches this notion of error. Fortunately, substituting constants for the least-significant bits allows us to manipulate their static values and capture this more intuitive behavior of error. We call this process "renormalization".

B. Renormalization

It is possible for us to capture the more natural description of error with our method of zero-substitution because the least-significant bits are still present. We can use these bits to manipulate the resultant error range. An example of renormalization in an adder structure is shown in Fig. 8. We describe this method as "in-line renormalization" as the error range is biased during the calculation. It is accomplished by modifying one of the input operands with one-substitution instead of zero-substitution. This effectively flips the error range of that input around zero. The overall effect is to narrow the resultant error range, bringing the net distance closer to zero. Specifically, if the number of substituted zeros and ones are equal, we achieve an error range whose net distance from zero is half that if we were to use zero substitution only. If instead truncation were performed, no further shaping of the error range would be possible, leaving us with a positively skewed error range not consistent with our natural notion of error.

For example, in Fig. 1, a substitution of p, q zeros results in an error range of $[0..2^p + 2^q - 2]$. By using renormalization, this same net distance from the real value can be achieved with more bit substitutions, $p + 1, q + 1$, on the input. This will yield a smaller area requirement for the adder. Likewise, the substitution of p, q zeros with renormalization now incurs half the error on the output, $[-(2^p - 1)..2^q - 1]$, as shown in Fig. 8.

As with the adder structure, renormalization of the multiplier is possible by using different values for least-significant bit substitution, yielding an error range that can be biased. Fig. 9 depicts a normalization centered on zero by substituting ones instead of zeros for input B. The derivation of the resultant
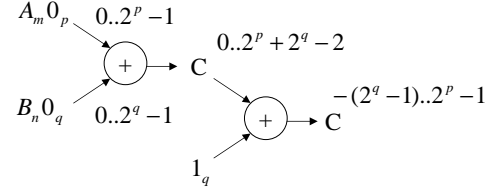
error range is as follows in (3):

$$
\begin{aligned}
C &= (A - E_p)(B + Eq) \\
&= AB + AE_q - BE_p - E_pE_q \\
&= AB + AE_q - (BE_p + E_pE_q) \\
&= AB + AE_q - \frac{E_pE_q}{2} - \left(BE_p + \frac{E_pE_q}{2}\right) \\
&= AB + \frac{E_q}{2}(2A - E_p) - \frac{E_p}{2}(2B + E_q)
\end{aligned}
\tag{3}
$$

Another method of renormalization can be accomplished after an operation, or operations, have been completed. By inserting a constant addition, we can accomplish a very similar biasing of error range, this time referred to as "active renormalization". An example is shown in Fig. 10.

C. Renormalization Area Impact

The benefits of renormalization can come very cheaply in terms of area for the "in-line" method. Our adder structure example in Fig. 3 originally requires 19 2-LUTs and has an error range of $[0..16]$. We can achieve a completely negative bias of $[-16..0]$ without an area penalty by modifying the structure of the least-significant half-adder to have a constant carry-in of 1. At the 2^4 bit position, this effectively adds 16 to the addition without incurring an area penalty. This has the same effect as using the "active renormalization", where an explicit addition is performed to change the error bias of the datapath. Alternatively, if we wanted to balance the error, we could achieve an error range of $[-8..8]$ by doing the same thing but at one bit position lower, 2^3. Unfortunately, since there is no existing half-adder hardware to modify for this bit position, we must create a half-adder structure at the 2^3 bit position to add together the value from input A and a constant "1". We also must change the existing half-adder at the 2^4 position into a full-adder to compensate for the possibility of a carry-out from the newly added half-adder. Together, this increases the area requirement of this adder by 5 2-LUTs.

Finally, we can do a smaller renormalization by substituting a "1" for one of the least-significant bits on one of the inputs. This would yield an output error range of $[-1..15]$. While not particularly biased, it doesn't incur any area penalty as the newly substituted "1" lines up with a zero from the other input, requiring no computational hardware.

Even when substituted ones and zeros on the inputs completely overlap, consideration must be made for downstream operations, as we now have ones in the least-significant bit positions which may need to be operated upon in subsequent
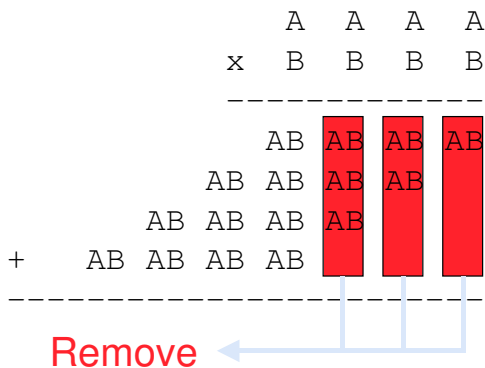
Fig. 11. A truncated multiplier removes least-significant columns from the partial product array.
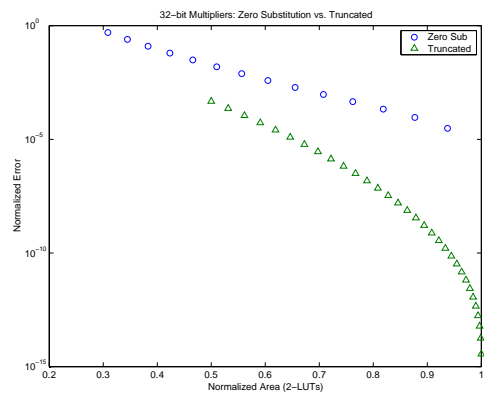

Fig. 12. Error to area profile of zero-substitution 32-bit multiplier and truncated 32-bit multiplier.

Fig. 13. Error to area profile of zero-substitution multipliers and truncated multipliers.

operations. This may adversely impact the overall area of the circuit, at which point "active" renormalization should be considered as an alternative that can be implemented cheaply later in the datapath to "fix up" the error range using a constant bias.

The behavior of renormalization in multiplier structures is equally interesting. As can be seen in Fig. 4, zeros substituted at the least-significant end of either the multiplier or the multiplicand "fall" all the way through to the result. For the multiplication $A_m 0_p * B_n 1_q$, p zeros will be present at the least-significant end of the result. With this behavior, we can obtain a renormalized error result while still providing zero-substituted bit positions that will not have to be operated upon in downstream operations. This is important in providing opportunities for area savings throughout the datapath. As with the adder structure, we pay a penalty for this renormalization. For the multiplier, we must put back an inner row and column for each one-substitution present in the multiplier and multiplicand, respectively.

Finally, active renormalization has an area penalty. As it is simply an addition between an input value and a constant positive bias, the impact is simply the area requirement of the biasing adder.

D. Alternative Arithmetic Structures

As discussed in previous sections, our zero-substitution method for multipliers gives a reduced area footprint at the cost of increased error in the output over an exact arithmetic multiplication. An alternative to this method of area/error tradeoff is one described in [8]. This work, and the work of others ([9], [10]), focuses on removing a number of least-significant columns of the partial-product array.

As described in [9], by removing the n least-significant columns from an array-multiplier multiplication, we save (for $n \geq 2$) $\frac{n(n+1)}{n}$ AND gates, $\frac{(n-1)(n-2)}{2}$ full adders, and $(n-1)$ half adders. The column removal is depicted in Fig. 11.

This method has a different area-to-error tradeoff profile, and is shown in Fig. 12 for a 32-bit multiplier, and in Fig. 13 for a range of differently-sized multipliers.
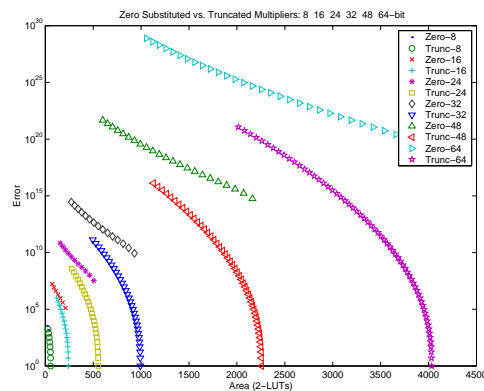
While the truncated multipliers have a more favorable area-to-error profile, one drawback in their use is that they require the full precision of both operands to be present at the inputs of the multiplier. This has the effect of requiring higher precision on upstream computations, possibly negating the area gain at a particular instance of a multiplier by requiring larger operations at upstream nodes. This makes it more valuable in multiplications closer to the inputs than those closer to the outputs.

VI. AUTOMATED OPTIMIZATION

We have presented in the previous section several optimization methods designed to allow more control of the area/error profile of our datapath. Unfortunately, due to the strongly interconnected nature of datapaths and dataflow graphs in general, it is hard to analytically quantify the impact of each method on the overall profile of the system. Making a small change, such as increasing the number of zero-substituted bits at a particular primary input, will impact the breadth of possible optimizations available at every node.

Fortunately, we have provided a model that can accurately determine the area and error of each node within the datapath. With these measurements and optimization "moves", we can utilize simulated annealing [11] to choose how to use our

palette of optimizations to achieve an efficient implementation area under a user-specified error constraint. We have developed an automated approach using simulated annealing principles similar to those found in [12] to area-optimize a dataflow graph. Simulated annealing has shown to produce good results on often intractable problems, and is a good candidate for our design challenge.

The possible moves in our system are the various optimization methods. At each temperature we choose randomly between altering the amount of zero-substitution at the inputs and changing multiplier structures. Our cost function for determining the quality of moves is determined by the area estimate of the entire datapath combined with a user-specified error constraint. This error constraint is identified as an error range at a particular node, dubbed the error node. Our cost function is defined in (4), where $error$ is the absolute value of the difference between the maximum error and the target error at the error node. We have determined through experimentation that $\beta = 0.25$ gives a good balance between an area efficient implementation and meeting the error constraint.

$$cost = \beta * area + (1 - \beta) * error \qquad (4)$$

When modifying an input, we allow the annealer to randomly choose to increase or decrease the number of bits substituted with constants by one bit. Thus, an input $A_5 0_2$ can move to $A_6 0_1$ or $A_4 0_3$.

When modifying the structure of a multiplier, we randomly choose a multiplier and adjust its degree of truncation. As with the inputs, we allow the annealer to increase or decrease by one the number of columns truncated from the partial product array. This allows a smooth transition from the traditional array multiplier to a highly-truncated multiplier.

At each temperature, after the move has been completed, we perform a greedy renormalization. Recalling from previous sections, there are several instances where the effect of renormalization can be achieved without an area impact. For each adder that may be renormalized without area penalty, we perform renormalization and observe the impact on the error node of interest. The adder that exhibits the most reduction in maximum error at the error node through renormalization is renormalized. This process is repeated until either our list of candidate adders is exhausted, or there can be no error improvement through renormalization. After the annealer has finished, we optionally apply active renormalization at the error node if it yields a lower overall implementation cost.

VII. EXPERIMENTAL RESULTS

We have implemented our automated optimization techniques as a subset of our design-time tool presented in [1]. To test the effectiveness of our methodologies, we have used our technique to optimize several benchmark image processing kernels. These include a matrix multiply, wavelet transform, CORDIC, and a one-dimensional discrete cosine transform.

A typical use of our methods would begin with the user performing basic truncation. As mentioned before, while basic truncation does afford an area savings throughout the datapath,
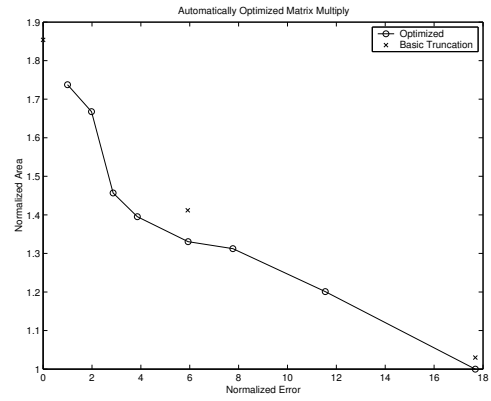

Fig. 14. Optimized results for matrix multiply.

there is very little guidance as to which inputs to manipulate, and how changes might affect the overall performance of the implementation. The starting points we have used in our experiments are truncating zero, one, and two bits from every input. These can be seen in Figs. (14-19) as the "Basic Truncation" points on the plots.

From these initial estimates of area and error, we performed the automated optimization using these points as guidelines for error constraints. The flexibility of our methods allows us to choose any error constraint, giving us far more area/error profiles to consider for implementation. As can be seen in the plots, the automated optimization method is able to obtain better area/error tradeoffs than the basic truncation method, except in a few cases in the wavelet transform and 1-D discrete cosine transform. We attribute this to the need for further tuning of some of the parameters in our simulated annealing algorithm. In particular, tuning the β parameter to adjust the weighting of meeting the error constraint vs. obtaining an area-efficient datapath. In the future, perhaps this parameter could be influenced by the user.

Careful observation will note a difference in performance between Figs. (16,17) and Figs. (18,19). In the experiements for the latter figures, we performed a slightly different experiment to determine whether or not our tool would be able to more aggressively optimize a single "precision critical path" in a circuit. In both the CORDIC and DCT, there were several output nodes to be considered. In our experiments for (18,19), we only constrained the error on one output node. From the plots it can be seen that the tool was able to maintain the desired precision at the output nodes of interest while finding more area efficient implementations. This type of optimization can be very useful when the developer is aware of varying degrees of precision required at the outputs.

VIII. CONCLUSIONS AND FUTURE WORK

We have described and motivated the need to investigate the optimization of the least-significant bit position. In order to do so, we have proposed models of area and error for an alternative area reduction technique to straight truncation— constant substitution. Using this method and models, we have
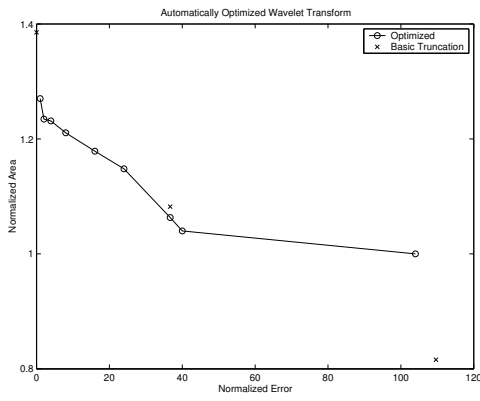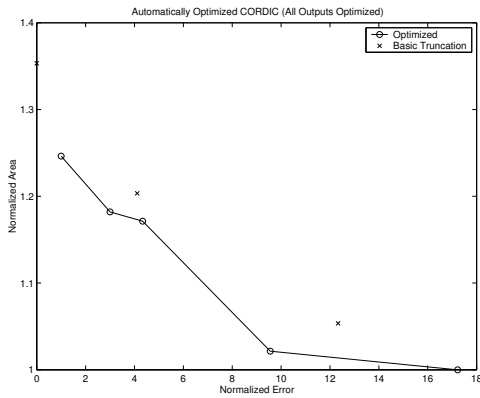
Fig. 15.   Optimized results for wavelet transform.

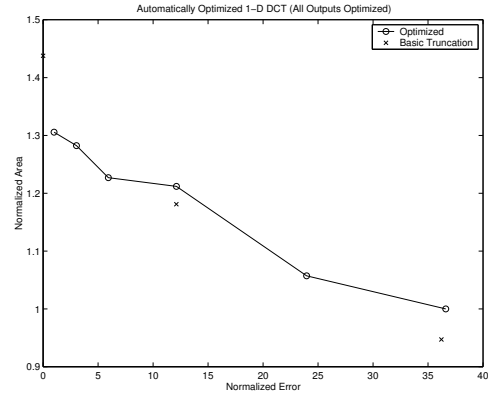Fig. 16.   Optimized results for CORDIC, all outputs optimized.

Fig. 17.   Optimized results for 1-D discrete cosine transform, all outputs optimized.
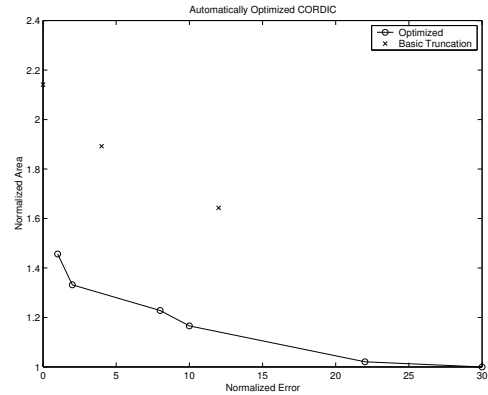

Fig. 18.   Optimized results for CORDIC, single output selected for optimization.
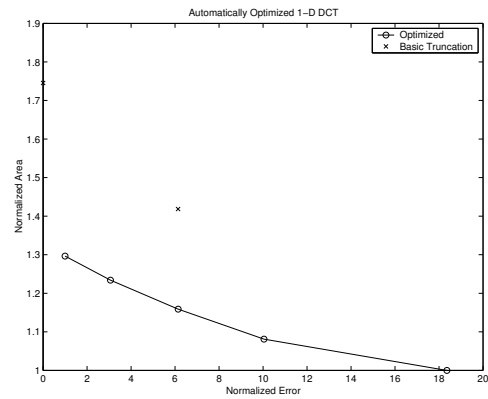

Fig. 19.   Optimized results for 1-D discrete cosine transform, single output selected for optimization.

proposed several optimization techniques aimed at giving the developer more control over the area-to-error tradeoff during datapath precision optimization that would not be available if simple truncation were used. We have proposed techniques for area-efficient renormalization, allowing us to more effectively capture our intuitive notion of error. We have introduced the use of alternative arithmetic structures, such as the truncated multiplier, in datapath optimization. Finally, we have implemented our techniques in an automated tool that is able to optimize a datapath subject to a user-supplied error constraint. More importantly, our techniques and tools give the user a broader range of options to consider, as well as a mechanism to achieve specific area/error targets when performing implementations.

In future work, we will incorporate more optimizations to further expand the design space. We will implement more of the renormalization techniques presented here in our automated tool. This will require a more comprehensive renormalization routine that will attempt the transformations that may increase the cost of a design. We hope to incorporate further alternative structures, such as floating-point and pseudo-floating-point to allow for high-precision (and high-area) portions of the datapath to be realized.

REFERENCES

[1] M. L. Chang and S. Hauck, "Précis: A design-time precision analysis tool," in *IEEE Symposium on Field-Programmable Custom Computing Machines*, 2002, pp. 229–238.

[2] M. Stephenson, J. Babb, and S. Amarasinghe, "Bitwidth analysis with application to silicon compilation," in *Proceedings of the SIGPLAN conference on Programming Language Design and Implementation*, June 2000.

[3] M. W. Stephenson, "Bitwise: Optimizing bitwidths using data-range propagation," Master's thesis, Massachusetts Institute of Technology, May 2000.

[4] W. Sung and K.-I. Kum, "Simulation-based word-length optimization method for fixed-point digital signal processing systems," *IEEE Transactions on Signal Processing*, vol. 43, no. 12, pp. 3087–3090, December 1995.

[5] S. Kim, K.-I. Kum, and W. Sung, "Fixed-point optimization utility for C and C++ based digital signal processing programs," in *Workshop on VLSI and Signal Processing*, Osaka, 1995.

[6] A. Nayak, M. Haldar, *et al.*, "Precision and error analysis of MATLAB applications during automated hardware synthesis for FPGAs," in *Design Automation & Test*, March 2001.

[7] G. A. Constantinides, P. Y. Cheung, and W. Luk, "The multiple wordlength paradigm," in *IEEE Symposium on Field-Programmable Custom Computing Machines*, 2001.

[8] Y. Lim, "Single-precision multiplier with reduced circuit complexity for signal processing applications," *IEEE transactions on Computers*, vol. 41, no. 10, pp. 1333–1336, October 1992.

[9] M. J. Schulte and J. Earl E. Swartzlander, "Truncated multiplication with correction constant," in *VLSI Signal Processing VI, IEEE Workshop on VLSI Signal Processing*, October 1993, pp. 388–396.

[10] K. E. Wires, M. J. Schulte, and D. McCarley, "FPGA resource reduction through truncated multiplication," in *Proceedings of the 11th International Conference on Field Programmable Logic and Applications*, August 2001, pp. 574–583.

[11] S. Kirkpatrick, J. C. D. Gelatt, and M. P. Vecchi, "Optimization by simulated annealing," *Science*, vol. 220, no. 4598, pp. 671–680, May 13 1983.

[12] V. Betz and J. Rose, "VPR: A new packing, placement and routing tool for FPGA research," in *Proceedings of the Seventh International Workshop on Field-Programmable Logic and Applications*, 1997, pp. 213–222.