Automated High-Level Synthesis of Low Power/Area Approximate Computing Circuits

Kumud Nepal Yueting Li R. Iris Bahar Sherief Reda
Brown University
{kumud_nepal, yueting_li,iris_bahar,sherief_reda}@brown.edu

Abstract

Many classes of applications, especially in the domains of signal and image processing, computer vision, and machine learning, are inherently tolerant to inaccuracies in their underlying computations. This tolerance can be exploited to design approximate circuits that perform within acceptable accuracies but have much lower power consumption and smaller area footprints than their exact counterparts. In this paper, we propose a new class of automated synthesis methods for generating approximate circuits directly from behavioral-level descriptions. In contrast to previous methods that operate at the Boolean level or use custom modifications, our automated behavioral synthesis method enables a wider range of possible approximations and can operate on arbitrary designs. Our method first creates an abstract synthesis tree (AST) from the input behavioral description, and then applies variant operators to the AST using an iterative stochastic greedy approach to identify the optimal inexact designs in an efficient way. Our method is able to identify the optimal designs that represent the Pareto frontier trade-off between accuracy and power consumption. Our methodology is developed into a tool we call ABACUS, which we integrate with a standard ASIC experimental flow based on industrial tools. We validate our methods on three realistic Verilog-based benchmarks from three different domains. Our tool automatically discovers optimal designs, providing area and power savings of up to 50% while maintaining good accuracy.

1. Introduction

A wide domain of applications, in general and embedded computation, show resilience to inaccuracies in their underlying computations. This type of error immunity in various applications can be exploited by creating approximate versions of their implementations that result in lower power consumption and smaller area footprints. Using approximate or inexact circuits in general, is attractive for classes of applications that are inherently tolerant to errors, which include signal and image processing, computer graphics, computer vision, and machine learning classifiers [1]. The computations of these applications do not necessarily need to be very precise in order to achieve the desired task. These applications show resiliency to certain kinds of inaccuracies for various reasons ranging from redundancies in large input data-sets, non-existence of a unique golden result, or the aggregating nature of the algorithms leading to errors averaging out [2].

There have been various research works in approximate circuits focusing on issues such as optimizing design metrics like area and power. From a conceptual point of view, these existing circuit approximation techniques aim to generate approximate circuits by either low-level modifications or through application-specific modifications. For example, numerous techniques have been proposed to generate approximate variants for standard building-block circuit components (e.g., adders or multipliers) [3–5]. Techniques that operate on an entire circuit either change the underlying synthesis algorithm used by the compiler[6, 7], or modify the circuit directly using user-defined application-specific changes as in the case of digital signal processing applications [8, 9]. Yet other techniques purposely use fault-prone components or circuits at unreliable operating conditions, not necessarily changing the logic level implementation, to give advantages in speed and/or power [10].

Our approach, to the best of our knowledge, is the first to operate directly at the behavioral descriptions of circuits to automatically generate approximate variants. We name our technique *ABACUS* after **A**utomated **B**ehavioral **A**pproximate **C**irc**u**it **S**ynthesis.[1] *ABACUS* creates an abstract synthesis tree (AST) from the input behavioral description, and then applies transformation operators to the AST within an iterative stochastic greedy approach to identify optimal designs that represent the Pareto frontier trade-off between accuracy and power consumption. Additionally, the tool fits within standard ASIC and FPGA flows, and we use industrial-strength simulation and compilation techniques to evaluate the design metrics of our designs. To evaluate our tool, we created three behavioral Verilog benchmarks from three different domains: a perceptron classifier implementation representing a common algorithm used in machine learning, a Sum of Absolute Difference (SAD) based block matching algorithm commonly used in computer vision, and lastly, a 25-tap FIR filter implementation popular in the signal processing domain.

Compared to previous methods, *ABACUS* has the following advantages.

- *ABACUS* automatically generates approximate designs from input behavioral descriptions. Behavioral descriptions capture the algorithmic structure of the circuits, and thus, ABACUS transformations are of global nature rather than limited to a particular sub-circuit. Furthermore, ABACUS can be applied to generic circuits with no need or limited need for application domain knowledge.

[1] A full version of this paper is appearing in DATE 2014 with title "ABACUS: A Technique for Automated Behavioral Synthesis of Approximate Computing Circuits" in March after the WACAS workshop.

CONF 'yy, Month d–d, 20yy, City, ST, Country.

http://dx.doi.org/10.1145/nnnnnnn.nnnnnnn

- The process of generating the variants is transparent to the design flow, and thus, different design flows (e.g., ASICs or FPGAs) can be used and subsequently all standard synthesis optimization techniques are applicable on the approximate variants.
- The outcome approximate variant designs from *ABACUS* are still expressed in behavioral or RTL code, which makes them easier to understand by the designers if needed.
- The accuracy of the approximate variants are evaluated using standard test benches that are most relevant for application operation.
- Complementary methods for approximate circuit generation (e.g., using an approximate adder, or voltage scaling or even using fault-prone circuitry in implementation) may be still used creating further approximation possibilities.
- *ABACUS* can be applied on an entire design or in modular parts of it giving it good scalability with various designs.

Our novel approach opens a new direction of research for approximate circuit synthesis and intelligent design space exploration. The rest of the paper is organized as follows: Section 2 discusses previous work done in approximate circuit design. Section 3 focuses on our methodology used in the design and development of *ABACUS*. We present the results our technique in Section 4 and conclude in Section 5 with a summary of our findings and a brief discussion on future work.

2. Previous Work

Given that power has emerged as a pressing issue for computing systems, new techniques have been proposed to devise approximate computing circuits in digital signal processing and emerging domains such as machine learning and data mining. Circuits in these areas, because of their error resilience, are explored on the principle of trading accuracy in exchange for better performance or power efficiency. In this section, we describe some of these works closely related to our own.

Two main methodologies have been used to achieve higher power efficiency. Voltage over-scaling is widely used in early works to manage dynamic power consumption for CMOS circuits, due to the quadratic decrease of switching power with supply voltage [10]. In more recent years, logical approximation has been studied as an alternative approach, where the Boolean functions of the underlying cores in computing circuits, such as adders and multipliers, are approximated for less complex design implementations [11–13]. As signal processing, machine learning or data mining applications went through rapid development and grew to be much more complex and power consuming, the benefits that these conventional methodologies can offer have become somewhat limited. Hence the effort of circuit approximation has been shifted to exploring the possibility of multi-level circuits or architecture-level approximation, for instance, as proposed in [14, 15]. Nevertheless, these efforts are either confined to logic approximation or demand high application-specific knowledge.

Whereas previous works in approximate circuit synthesis mostly focused on Boolean level or custom modifications, our work aims to synthesize approximate design variants from generic input behavioral descriptions in an entirely automated fashion. Behavioral descriptions capture the algorithmic intent of the circuit; and thus raising the level of abstraction enables a larger range of approximations that are not possible to apply at low-level design specifications. We draw inspiration for our work from the recent advances in software engineering targeted for automatic bug identification as proposed in [16], but investigate methods that are suitable for hardware designs.

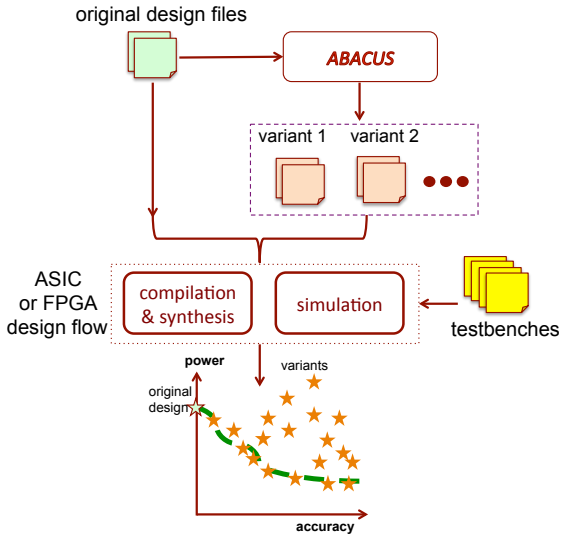

Figure 1. Incorporation of ABACUS within standard design flows.

3. Methodology

As discussed in Section 1 and Section 2, existing circuit approximation techniques aim at generating approximate variants for standard sub-circuits (e.g., adders), given their Boolean descriptions, or through manual modifications for specific applications (e.g., DSPs). In contrast, we aim to generate approximate circuit variants for any system from its high-level behavioral description.

In a basic design flow, the behavioral or register transfer level (RTL) code is first generated by the designers from the design specifications. The code is then simulated functionally using a number of representative testbenches and the simulation results are evaluated to verify correctness of operation. The code is then compiled and synthesized to a netlist using a design compiler, which also takes as input a standard cell library for ASICs or the look-up table and cluster architecture for FPGAs. The netlist is afterwards placed and routed to get the final area, timing, and power metrics. We integrate *ABACUS* with traditional ASIC/FPGA flows but instead of synthesizing a single exact code, multiple approximate code variants are also synthesized at the RTL/behavioral level. As illustrated in Figure 1, these variants are pushed through the standard design flow, and the approximate outcomes are compared with the original exact design in terms of functional accuracy and hardware design metrics such as power, area and timing. The evaluated outcomes are then plotted and a *Pareto frontier* is computed to identify the approximate designs that give the optimal accuracy-power trade-off.

To achieve our goal of generating approximate behavioral variants, we propose (i) to capture the exact RTL or behavioral hardware description language (HDL) design in an Abstract Syntax Tree (AST) structure; (ii) create the approximate design variants through transformations to the AST; and (iii) finally write back the modified AST into readable RTL or behavioral HDL design. The three steps are illustrated in Figure 2. The new approximate design is then pushed through the standard ASIC/FPGA flow for evaluation in terms of accuracy and other design metrics. In an AST, each node represents an action to be taken by the behavioral code, or an object to be acted upon [17]. Building an AST for a HDL syntax automatically captures all the concurrency that is in the design. Compared to a regular parsing tree, the AST captures the logical structures of the statements and shows less of the grammar structure, which makes it a better candidate to analyze and transform the code. Compared to control flow graphs, ASTs are easier to use

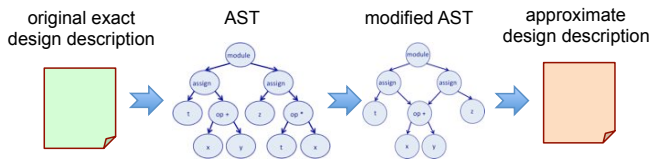

Figure 2. Overall methodology of ABACUS.

to produce readable code. The novelty in our *ABACUS* approach allows us to make automated transformations to any generic HDL design without the need to have any *a priori* knowledge of the functionality or the semantics of the design. Hence, a much broader range of transformations can be explored at the high level, leading to a superior design that what could not be achieved with prior approaches.

There are two key questions we address in our approach:

1. *What kind of transformations can be applied to HDL-based ASTs?* Generating approximate AST variants from the original AST requires the application of HDL-aware transformations that lead to syntactically correct, yet approximate HDL designs. In Section 3.1, we propose a set of transformations that can be applied to the AST to generate meaningful approximate HDL designs.

2. *How do we avoid the explosive increase in design search space resulting from these transformations?* The application of a set of operators combinatorially to the original design can lead to an exponential number of variant designs that need to be explored. Given the size of modern designs and runtimes of typical design flows, this approach is clearly infeasible in terms of total runtime. Thus, we propose in Section 3.2 an approach to effectively explore the design of possible approximate designs and identify the ones that provide the optimal trade-offs between accuracy and hardware design metrics.

3.1 Generating HDL-based Approximate Transformations

We present a set of *transformation operators* that can be applied to the original HDL-based AST to yield meaningful approximate designs for error-resilient applications. Whenever any of these transformations is invoked, *ABACUS* automatically traverses through the AST and searches for places in the AST where the change could be applied. We propose and implement the following five transformation operators in *ABACUS*:

1. Data Type Simplifications: For applications dealing with massive data, truncating the size of intermediate signals may be a good way to achieve savings on power, since it reduces the requirements for the underlying hardware, especially for fixed-point arithmetic operations. *ABACUS* is capable of performing truncation in two ways: first, by resetting a number of the least significant bits to zero, and second by truncating a certain number of significant bits for operands during binary arithmetic operations and then shifting the result of the operation to get the approximation. The latter transformation yields more significant power and area savings.
2. Operation Transformations: Another proposed operator is to substitute an arithmetic operation with one or more arithmetic approximate operations that use less power and hardware area. For example, arithmetic additions could be replaced by bitwise ORs or a multiplication could be replaced by shifts and an addition. Also, a standard adder or multiplier could be replaced by an approximate unit from the ones proposed in the literature [11–13]. Thus, our behavioral-based approach can easily leverage approximate Boolean arithmetic circuits.
3. Arithmetic Expression Transformations: There are cases where near similar arithmetic structures appear in the same statement description. Through a transformation, these near-similar structures could be transformed to approximate similar structures in which case they could be shared and simplified. For instance, we can approximate the expression $(w_i \times x_i + w_j \times x_j)$ with substitutions to the variables or the constants, such as substituting x_j by x_i leading to $(w_i + w_j) \times x_i$ or substituting w_j by w_i leading to $w_i \times (x_i + x_j)$, thus saving one multiplier. We can simplify computations by sharing common or similar operands and get good approximations.
4. Variable-to-Constant Substitution Transformations: Simulation results of the original design contain useful information about the numerical characteristics of the design variables. This information can guide the transformation operations. For instance, if an intermediate variable derived from a certain arithmetic operation appears to be a constant or has a small standard deviation in the simulation results, then we can substitute it with a constant based on its average value, thus saving its computational circuit. *ABACUS* implements this feature by reading simulation results from the original exact design to identify design variables that are constant or are within a 10% standard deviation. These design variables are candidates for substitution by a constant based on their simulation results.
5. Loop Transformations: ABACUS automatically unrolls loops in behavioral descriptions. Loop unrolling is typically used as a compiler transformation technique; however, in our case we use automatic unrolling in the pre-compiler phase of the behavioral description of an algorithm. Loop unrolling opens the door for the application of other simplification operators. In addition, the unrolling can be done in an approximate way by skipping certain iterations and substituting the outcomes of these iterations from the results of prior iterations.

3.2 Effective Design Space Exploration

Application of the proposed operator transformations can lead to a combinatorial explosion in possible approximate design variants, as there are multiple operators, where each operator can be applied at several locations, and the AST resultant from one operator can be used as input for another operator in a chain of transformation. To effectively explore the search space of possible design outcomes and identify the Pareto frontier of optimal trade-off designs, we propose the following *iterative stochastic greedy algorithm* that continuously evolves the approximate designs by doing multiple iterations of transformations to identify the Pareto frontier that gives the optimal tradeoff between accuracy and power.

Algorithm *Approximate Design Space Exploration*
Input: original exact design
Output: approximate design variants
1. Let $O_1 =$ original design
2. **while** $i \leq N$
3. **while** $j \leq M$
4. **do** pick a transformation operator at random;
5. apply the operator to O_i to yield A_j;
6. evaluate accuracy of A_j over input data sets
7. **if** accuracy of A_j is within threshold
8. **then**
9. synthesize A_j;
10. $V = V \cup A_j$;
11. Use results from Steps 6 and 9 to evaluate the fitness, F_j, of A_j;
12. **else** goto step 4;
13. identify $A_k, k \in \{1, \ldots, M\}$, with best F_j;
14. let $O_{i+1} = A_k$;
15. **return** V

ABACUS couples a simulation tool and a synthesis tool respectively to evaluate accuracy and design metrics. The algorithm goes through N iterations, where each iteration attempts M transformation operators. In steps 4-6 above, a transformation operator is picked at random with some probability and applied to the current design and the results are evaluated for accuracy. Accuracy of an approximate design is averaged over a number of input training data sets. If the average accuracy measure meets an accuracy threshold then the design is considered a valid variant and passed on to the synthesis tool in Step 9. The accuracy threshold is pre-set to filter out bad design variants from the good approximate ones. A design with accuracy less than this threshold will not be synthesized and is not further considered for use with the tool. Using the accuracy results from Step 6 and the synthesis results from step 9, the design variant is evaluated for fitness, which is defined as

$$fitness = \alpha_1 \times accuracy + \alpha_2 \times power + \alpha_3 \times area \quad (1)$$

$\alpha_1 > \alpha_2 > \alpha_3$. In step 13, all M variants are ranked and the design with the highest rank for fitness is used in the subsequent iterations as the parent design for transformations as given in Step 14. *ABACUS* repeats these generations of transformations greedily to get the best area and power saving results under the allowed inaccuracy budget until it reaches the defined limit for number of generations.

Whether an operator transformation is applied or not is dependent on a probability function to ensure no bias towards a particular operator. Furthermore, for a given operator, the location where an operator is applied is also randomized with a probability to ensure no bias towards a particular location. Thus the sequence of applied transformations is stochastic in nature. During the iterative greedy procedure, *ABACUS* keeps a log of accuracy, area and power values of each design variant used along the way. A wide range of optimal designs at various accuracy and power savings can be obtained in this manner and help in creating the Pareto frontier for trade-off between accuracy and other metrics. The designs that pass the final generation of mutations are also ranked for fitness; the highest ranked design represents the behavioral description that comes closest to the allowed arithmetic inaccuracy and is better in terms of power and area utilization in that order.

Our methodology avoids explosive design space exploration by constricting the design choices using a greedy heuristic. If every variant generated per generation were to be used as an originating design for further design transformations, the run-time needed for a full evaluation would increase with the number of generations as a geometric series. Lets say, there are N iterations and M designs per generation. This would mean that without the greedy approach, the number of designs to be used as transformation seeds would increase as $M^0 + M^1 + M^2 + M^3 + ... + M^N$, or mathematically, $\frac{(1-M^{N+1})}{(1-M)}$. With *ABACUS*, the number of designs increases linearly with N so the final number of designs to be used for exploration would be MN. The speedup in runtime would hence be $\frac{(1-M^{N+1})}{MN(1-M)}$. So for 20 iterations with 5 variants per generation, the speedup would be about 1.2×10^{12}.

4. Experimental Results

We have implemented our *ABACUS* tool and integrated it with a standard industrial-strength flow comprised of Synopsys Design Compiler and Mentor Graphics ModelSim. Our initial AST parsing front end was based on the ODIN-II tool [18]. We used 45 nm technology libraries. Our three circuits and their test benches are coded in behavioral Verilog. These test cases are briefly described as follows.

1. FIR filter: A 25-tap FIR filter takes in a 308×242 grayscale image and convolves it with a 5×5 low-pass filter coefficient matrix, essentially creating a blur-effect. The quality of an approximate version of this design is assessed using Mean-Squared Error (MSE). The MSE for the original FIR filter circuit was computed with an 8-bit fixed point coefficient and image input compared against a floating point implementation done in software.

2. Perceptron Classifier: Perceptron classifier is a commonly used application in machine learning. A perceptron takes an input data, denoted by vector x, and predicts the class (e.g., -1 or +1) of x by computing $sign(w^T x)$, where w is the weight vector and $sign(\cdot)$ is a function that outputs 1 if its argument is positive and -1 otherwise. The perceptron essentially defines a hyperplane to separate the training data into two classes. Perceptrons are also capable of classifying non-linearly separable points by mapping the input points to another space where they are linearly separable, i.e., $sign(w^T \phi(x))$, where $\phi(\cdot)$ is the mapping function. Our perceptron test case uses a quadratic function to map the input space. The input data set consisted of 1000 randomly generated two-dimensional points from two classes. Classification results were compared against the ground-truth and hence, the total percentage change in classification outputs was considered as the accuracy metric.

3. Block Matcher: Block matching is a technique commonly used in motion estimation and video compression applications. Block matching partitions a given frame into non-overlapping rectangular blocks and tries to find the block from the reference frame in a given search range that best matches the current block. The measure of similarity between the blocks is computed by sum of the differences. For our design, we perform full search block matching over a search window in a reference frame to determine the best match for a block in a current frame. Our particular test case works on 16×16 block sizes from a 352×288 frame sequence. The quality of a design is assessed using the PSNR.

The main hardware design characteristics and quality of these test benches are summarized in Table 1. For each design, a total of six data sets were used to train and evaluate *ABACUS*. Three data sets were to used to generate the approximate designs as described in Section 3, and another three were used to assess the accuracy of ABACUS for the experiments of this section. Using different data sets in the experimental results eliminates possibility of generating approximate variants overfitted for one particular set of input data. In all experiments, we report the average accuracy of the three data sets. We used weights of 0.8, 0.12 and 0.08 as α_1, α_2 and α_3 respectively in Equation 1 for fitness evaluation. *ABACUS* was applied to the computational data-path parts of the designs, but the control signals in the designs were not modified. Using *ABACUS*, we were able to automatically apply multiple iterations of transformations on each test bench to identify the approximate designs with optimal trade-offs between accuracy and power. Figure 3 plots the accuracy vs. power saving results for all approximate designs generated by *ABACUS* for the perceptron, block matching, and FIR designs, respectively. The x-axis gives accuracy and the y-axis gives power savings. A subset of all these points create a Pareto Frontier (solid red line), where the frontier points do not dominate each other in both power and accuracy. The runtimes of ABACUS are mostly dominated by the runtime of the ASIC design flow. On a Intel Core 2 Quad machine at 2.40 GHz with 6 GB RAM, it took 140 seconds, 130 seconds and 529 seconds for generating one instance of the FIR, perceptron and the block matching benchmarks respectively. A total of 10 generations with 5 iterations were run for the FIR design and the perceptron design and 15 generations with 6 iterations were run for the block matching design.

In Table 2, we highlight results from the best approximate designs that allowed for a maximum of 8% degradation in accuracy

Design	Class of Application	#Lines	Area (um^2)	Power (mW)	Quality Measure	Quality
FIR filter	Signal Processing	265	16711.18	0.94	MSE	98.63%
perceptron	Machine Learning	188	19183.12	1.28	classification error	82.88%
block matching	Computer Vision	1277	42532.87	5.51	PSNR	30.44 dB

Table 1. Characteristics of test cases.

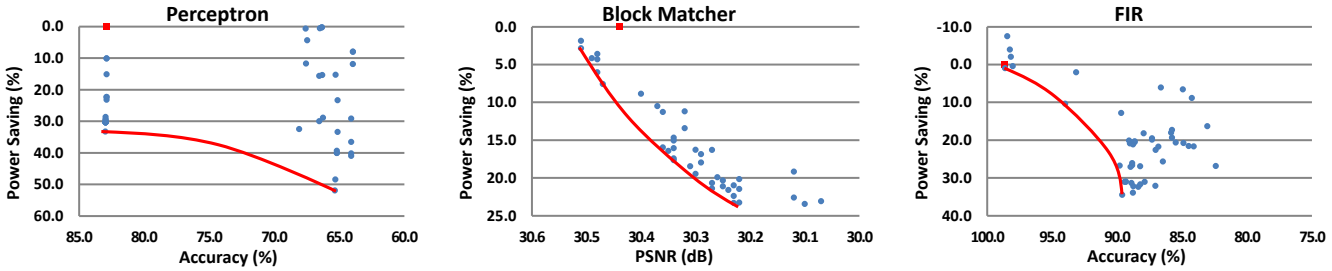

Figure 3. Results from various approximate designs and the Pareto Frontier for the three test benches.

Design	#Iter	Accuracy Threshold	Accuracy Achieved	Power Saving	Area Saving
FIR	10	90.9%	93.9%	10.4%	15.8%
perceptron	10	76.2%	82.9%	33.2%	38.3%
blockmatching	15	28.0 dB	30.0 dB	23.0%	19.4 %

Table 2. Results from ABACUS for the three test benches for an allowed 8% degradation to accuracy.

compared to the original designs. The results show that we are able to attain significant savings in power consumption (ranging from 10% to 33%) with these approximate designs with very modest degradation in accuracy. Figure 4 illustrates the results from the perceptron approximate design of Table 2. Figure 4.a gives the true classification of the data points into the two classes (class A and class B). Figure 4.b gives the classification of both the original and approximate hardware (HW) designs on the same data points. The true-true case is when both HW designs correctly predicted the classes of the data points, while the false-false case is when HW designs incorrectly predicted the classes of the data points. The false-true case is when the original design predicted incorrectly, but the approximate design predicted correctly. Finally, the true-false case is when the original design predicted correctly, but the approximate design incorrectly. The figure shows very few points

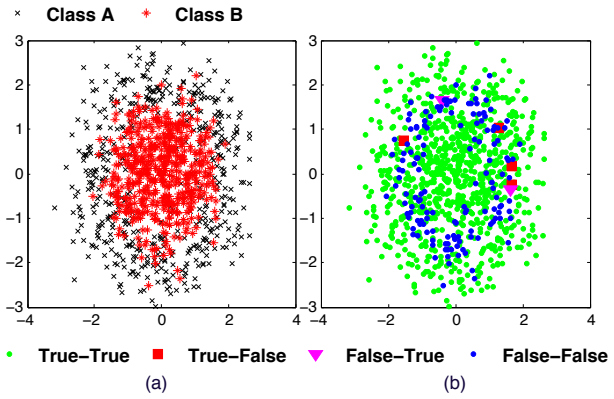

Figure 4. (a) Classification of input data into two classes, (b) comparison between original and approximate designs.

in the true-false case, and almost balanced in number by points in the false-true case. As a result the approximate design attains comparable accuracy to the original, while achieving 33.2% power savings.

The result from the block matching variant in Table 2 also shows great promise. The approximate design is able to achieve 23% reduction in power dissipation for a very small reduction in accuracy from an original PSNR of 30.4 dB to 30.0 dB. Figure 5 illustrates the motion estimation vectors (yellow arrows) from (a) the original, and (b) approximate implementations, where the red arrows in (b) show the few vectors in the approximate design that are not in agreement with the original designs. That is, our results show that it is possible to reduce the power dissipation by a quarter with very limited impact on solution quality.

We also compared the results generated by *ABACUS* to a commonly used technique in approximation as described in Section 2, where approximate versions of standard components are used in place of the accurate ones. We truncated 3 bits off a set of multipliers for the perceptron and block matching implementations and 3 bits off a set of adders for the FIR implementation. Results are shown in Figure 6 for designs that maintain same accuracy. Our method obtains superior results for all three benchmarks. This validates our claim that *ABACUS* is capable of making global arbitrary approximations that may not be obvious to the designer but produce better results. Finally, Table 3 breaks down the amount of data types, operators and arithmetic expressions transformed by *ABACUS* for the three designs in Figure 6.

5. Conclusions and Future Work

With *ABACUS* we have successfully developed a new approach for approximation of circuits that is capable of generating and synthesizing circuits with reasonable error tolerance and significantly less area consumption and power dissipation. While these metrics can be highly application dependent, we were able to show with our three test cases that we can get over 30% savings on both power and area for some designs. *ABACUS* requires no application domain knowledge and applies global transformations that are not obvious to a designer. These features make it unique to some of the previous work in the field of approximate circuits.

Future work: We plan on expanding ABACUS capabilities on multiple fronts:

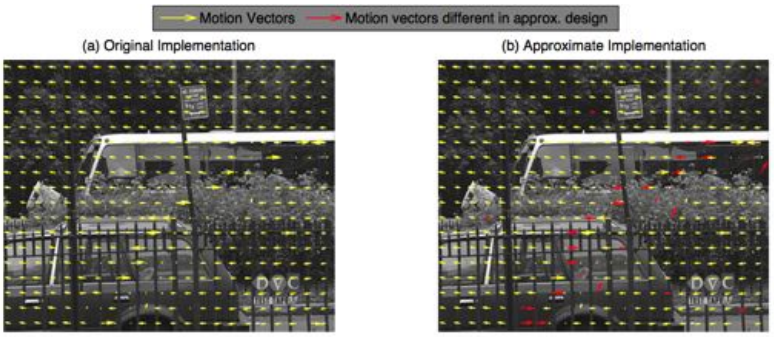

Figure 5. Motion vectors from the original and approximate designs.

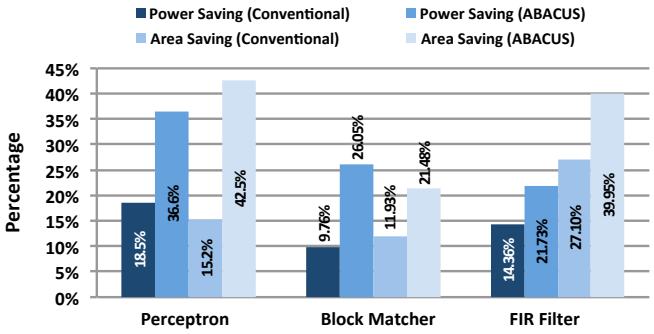

Figure 6. Results for use of conventional technique by use of approximate multipliers compared to results from ABACUS.

Design	Data Types	Operators	Arithmetic Expressions
FIR	16	13	2
perceptron	4	2	1
blockmatching	95	91	33

Table 3. Number of some of the major transformations made in the three benchmarks.

- we plan to incorporate genetic mutation algorithms in *ABACUS* alongside or as an enhancement to our iterative greedy heuristic for effective design mutation and selection;
- we are considering improved optimization techniques for design space exploration and multi-objective optimization techniques [19], [20] into our methodology; and finally
- we also plan on complementing *ABACUS* with techniques to use approximate circuits for error resiliency.

References

[1] Y.-K. Chen, J. Chhugani, P. Dubey, C. Hughes, D. Kim, S. Kumar, V. Lee, A. Nguyen, and M. Smelyanskiy, "Convergence of Recognition, Mining, and Synthesis Workloads and its Implications," *Proceedings of the IEEE*, vol. 96, no. 5, pp. 790–807, 2008.

[2] S. Chakradhar and A. Raghunathan, "Best-effort computing: Rethinking Parallel Software and Hardware," in *DAC*, 2010, pp. 865–870.

[3] L. N. Chakrapani, K. K. Muntimadugu, A. Lingamneni, J. George, and K. V. Palem, "Highly Energy and Performance Efficient Embedded Computing through Approximately Correct Arithmetic: A mathematical Foundation and Preliminary Experimental Validation," in *CASES*, 2008, pp. 187–196.

[4] V. Gupta, D. Mohapatra, S. P. Park, A. Raghunathan, and K. Roy, "Impact: Imprecise Adders for Low-power Approximate Computing," in *ISLPED*, 2011, pp. 409–414.

[5] J. Huang and J. Lach, "Exploring the Fidelity-Efficiency Design Space using Imprecise Arithmetic," in *DAC*, 2011, pp. 579–584.

[6] M. Choudhury and K. Mohanram, "Approximate Logic Circuits for Low Overhead, Non-intrusive Concurrent Error Detection," in *DATE*, 2008, pp. 903–908.

[7] D. Shin and S. Gupta, "Approximate Logic Synthesis for Error Tolerant Applications," in *DATE*, 2010, pp. 957–960.

[8] J. George, B. Marr, B. E. S. Akgul, and K. V. Palem, "Probabilistic Arithmetic and Energy Efficient Embedded Signal Processing," in *CASES*, 2006, pp. 158–168.

[9] L. Leem, H. Cho, J. Bau, Q. Jacobson, and S. Mitra, "ERSA: Error Resilient System Architecture for probabilistic applications," in *DATE*, 2010, pp. 1560–1565.

[10] A. Chandrakasan and R. Brodersen, "Minimizing Power Consumption in Digital CMOS Circuits," *Proceedings of the IEEE*, vol. 83, no. 4, pp. 498–523, 1995.

[11] J. Huang, J. Lach, and G. Robins, "A Methodology for Energy-Quality Tradeoff using Imprecise Hardware," in *DAC*, 2012, pp. 504–509.

[12] P. Albicocco, G. Cardarilli, A. Nannarelli, M. Petricca, and M. Re, "Imprecise Arithmetic for Low Power Image Processing," in *Asilomar Conference on Signals, Systems and Computers*, 2012, pp. 983–987.

[13] V. Gupta, D. Mohapatra, A. Raghunathan, and K. Roy, "Low-power Digital Signal Processing using Approximate Adders," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 32, no. 1, pp. 124–137, 2013.

[14] S. Venkataramani, A. Sabne, V. Kozhikkottu, K. Roy, and A. Raghunathan, "SALSA: Systematic Logic Synthesis of Approximate Circuits," *DAC*, 2012, pp. 796–801.

[15] V. K. Chippa, D. Mohapatra, A. Raghunathan, K. Roy, and S. T. Chakradhar, "Scalable Effort Hardware Design: Exploiting Algorithmic Resilience for Energy Efficiency," in *DAC*, 2010, pp. 555–560.

[16] W. Weimer, S. Forrest, C. Le Goues, and T. Nguyen, "Automatic Program Repair with Evolutionary Computation," *ACM Communications*, vol. 53, no. 5, pp. 109–116, May 2010.

[17] S. S. Muchnick, *Advanced Compiler Design & Implementation*. Morgan Kaufmann, 2003.

[18] P. Jamieson, K. B. Kent, F. Gharibian, and L. Shannon, "ODIN II - An Open-source Verilog HDL Synthesis Tool for CAD Research," in *FCCM*, 2010, pp. 149–156.

[19] K. Nepal, O. Ulusel, R. I. Bahar, and S. Reda, "Fast Multi-objective Algorithmic Design Co-exploration for FPGA-based Accelerators," in *FCCM*, 2012, pp. 65–68.

[20] K. Deb, S. Agrawal, A. Pratap, and T. Meyarivan, "A Fast Elitist Non-dominated Sorting Genetic Algorithm for Multi-objective Optimization: NSGA-II." Springer, 2000, pp. 849–858.