Axilog: Abstractions for Approximate Hardware Design and Reuse

Divya Mahajan, Kartik Ramkrishnan*, Rudra Jariwala*, Amir Yazdanbakhsh, Jongse Park, Bradley Thwaites, Anandhavel Nagendrakumar, Abbas Rahimi§, Hadi Esmaeilzadeh, Kia Bazargan*

Georgia Institute of Technology *University of Minnesota §UC San Diego

axilog@act-lab.org http://www.act-lab.org/artifacts/axilog

***Abstract*—Relaxing the traditional abstraction of "near-perfect" accuracy in hardware design can yield significant gains in efficiency, area, and performance. To exploit this opportunity, there is a need for design abstractions and synthesis tools that can systematically incorporate approximation in hardware design. We define Axilog, a set of language extensions for Verilog, that provides the necessary syntax and semantics for approximate hardware design and reuse. Axilog enables designers to safely relax the accuracy requirements in the design, while keeping the critical parts strictly precise. Axilog is coupled with a Safety Inference Analysis that automatically infers the safe-to-approximate gates and connections from the annotations. The analysis provides formal guarantees that the safe-to-approximate parts of the design are in strict accordance to the designer's intentions. We devise two synthesis flows that leverage Axilog's framework for safe approximation; one by relaxing the timing requirements and the other through gate resizing. We evaluate Axilog using a diverse set of benchmarks that gain 1.54× average energy savings and 1.82× average area reduction with 10% output quality loss. The results show that the intuitive nature of the language extensions coupled with the automated analysis enables safe approximation of designs even with thousands of lines of code.**

I. Introduction

Several works have shown significant benefits with approximation at the circuit level [1–15]. While these techniques allow approximation at circuit-level, there is a lack of design abstractions that enable designers to methodically control which parts of a circuit can be approximated while keeping the critical parts precise. Thus, there is a need for *approximate hardware description languages* enabling systematic synthesis of approximate hardware. We introduce Axilog–a set of concise, intuitive, and high-level annotations–that provides the necessary syntax and semantics for approximate hardware design and reuse in Verilog.

A key factor in our language formalism is to abstract away the details of approximation while maintaining the designer's complete oversight in deciding which circuit elements can be synthesized approximately and which circuit elements are critical and cannot be approximated. Axilog also supports reusability across modules by providing a set of specific reuse annotations. In general, hardware system implementation relies on modular design practices where the engineers build libraries of modules and reuse them across complex hardware systems. Section II elaborates on the Axilog annotations for approximate hardware design and reuse. These annotations are coupled with a Safety Inference Analysis that automatically infers which circuit elements are safe-to-approximate (Section III) with respect to the designer's annotations. Axilog and safety analysis support approximate synthesis, however, they are completely independent of the synthesis process. To evaluate Axilog, we devised two synthesis processes (Section IV). The first synthesis flow focuses on current technology nodes and leverages commercial tools. This synthesis process applies approximation by relaxing the timing constraints of the safe-to-approximate subcircuits. The results shows that this synthesis flow provides, on average, 1.54× energy savings and 1.82× area reduction by allowing a 10% quality loss. The second synthesis flow aims to study the potential of approximate synthesis by using probabilistic gate model for future technology nodes. The results shows that this synthesis flow provides, on average, 2.5× energy and 2.2× PCMOS area reduction (defined in Section IV-B). Axilog yields these significant benefits while only requiring between 2 to 12 annotations even with complex designs containing up to 22,407 lines of code. These results confirm the effectiveness of Axilog in incorporating approximation in the hardware design cycle.

II. Approximate Hardware Design with Axilog

Our principle objectives for approximate hardware design with Axilog are (1) to craft a small number of Verilog annotations that provide designers with complete oversight over the approximation process; (2) to minimize the number of manual annotations while relying on Safety Inference Analysis (Section III) to automatically infer the designer's intent for approximation. This relieves the designer from the details of the approximate synthesis process. (3) to support the reuse of Axilog modules across different designs without the need for reimplementation. Furthermore, Axilog is a backward-compatible extension of Verilog. That is, an Axilog code with no annotations is a normal Verilog code. To this end, Axilog provides two sets of language extensions, one set for the design (Section II-A) and the other for the reuse of hardware modules (Section II-B). Table I summarizes the syntax for the design and reuse annotations. The annotations for design dictate which operations and connections are safe-to-approximate in the module. Henceforth, for brevity, we refer to operations and connections as design elements. The annotations for reuse enable designers to use the annotated approximate modules across various designs without any reimplementation. We provide detailed examples to illustrate how designers are able to appropriately relax or restrict the approximation in hardware modules. In the examples, we use **background shading** to highlight the safe-to-approximate elements inferred by the analysis.

A. Design Annotations

Relaxing accuracy requirements. By default, all design elements are precise. The designer can use the relax(arg)

TABLE I: Summary of Axilog's language syntax.

Phase	Annotation	Arg	Description
Design	relax	wire, reg, output, inout	Declare an argument as safe-to-approximate. Design element that affect the argument are safe to approximate.
	relax_local		Similar to relax but the approximation does not cross module boundaries.
	restrict		Any design element that affects the argument is made precise unless explicitly relaxed.
	restrict_global		All the design elements affecting the argument are precise.
Reuse	approximate	output, inout	Indicates the output carries relaxed sematics.
	critical	input	Indicates the input is critical and approximate elements cannot drive it.
	bridge	wire, reg	Allow connecting an approximate element to a critical input.

statement to *implicitly* approximate a subset of these elements. The variable arg is either a wire, reg, output, or inout. Design elements that *exclusively* affect signals designated by the relax annotation are safe to approximate. The use of relax is illustrated using the following example.

```
module full_adder(a, b, c_in, c_out, s);
    input a, b, c_in; output c_out;
    approximate output s;
    assign s = a ^ b ^ c_in;
    assign c_out = a & b + b & c_in + a & c_in;
    relax(s);
endmodule
```

In this full_adder example, the relax(s) statement implies that the analysis can automatically approximate the XOR operations. The unannotated c_out signal and the logic generating it is not approximated. Furthermore, since s will carry relaxed semantics, its corresponding output is marked with the approximate annotation that is necessary for reusing modules (discussed in Section II-B). With these annotations and the automated analysis, the designer does not need to *individually* declare the inputs (a, b, c_in) or any of the XOR (^) operations as approximate. Thus, while designing approximate hardware modules, this abstraction significantly reduces the burden on the designer to understand and analyze complex data flows within the circuit.

Scope of approximation. The scope of the relax annotation crosses the boundaries of instantiated modules as shown by the code on the left side. The relax(x) annotation in the nand_gate module implies that the AND (&) operation in the and_gate module is safe-to-approximate. In some cases, the designer might not prefer the approximation to cross the scope of the instantiated modules. Axilog provides the relax_local annotation that does not cross module boundaries.

```
module and_gate(n,a,b);
    input a, b; output n;
    assign n = a & b;
endmodule
module nand_gate(x, a, b);
    input a, b;
    approximate output x;
    wire w0;
    and_gate a1(w0, a, b);
    assign x = ~ w0;
    relax(x);
endmodule
```

```
module and_gate(n,a,b);
    input a,b; output n;
    assign n = a & b;
endmodule
module nand_gate(x, a, b);
    input a, b;
    approximate output x;
    wire w0;
    and_gate a1(w0, a, b);
    assign x = ~ w0;
    relax_local(x);
endmodule
```

The code on the right side shows that the relax_local annotation does not affect the semantics of the instantiated and_gate module, a1. However the NOT (~) operation which shares the scope of the relax_local annotation is safe-to-approximate. The scope of approximation for both relax and relax_local is the module in which they are declared.

Restricting approximation. In some cases, the designer might want to *explicitly* restrict approximation in certain parts of the design. Axilog provides the restrict(arg) annotation that ensures that any design element affecting the annotated argument (arg) is precise, *unless* a preceding relax or relax_local annotation has made the driving elements safe-to-approximate. The restrict annotation crosses the boundary of instantiated modules.

Restricting approximation globally. There might be cases where the designer intends to override preceding relax annotations. For instance, the designer might intend to keep certain design elements that are used to drive critical signals such as the control signals for a state machine, write enable of registers, address lines of a memory module, or even clock and reset. To ensure the precision of these signals Axilog provides the restrict_global annotation that has precedence over relax and relax_local. The restrict_global(arg) penetrates through module boundaries and ensures that any design element that affects arg is not approximated.

B. Reuse Annotations

Our principle idea for these language abstractions is to maximize the reusability of the approximate modules across designs that may have different accuracy requirements. This section describes the abstractions that are necessary for reusing approximate modules.

Outputs carrying approximate semantics. As mentioned before, designers can use annotations to selectively approximate design elements in a module. The reusing designer needs to be aware of the accuracy semantics of the input/output ports without delving into the details of the module. To enable the reusing designer to view the port semantics, Axilog requires that all output ports that might be influenced by approximation to be marked as approximate. Below, the code snippets illustrate the necessity of the approximate annotation.

```
module and_gate(n,a,b);
    input a,b;
    approximate output n;
    assign n = a & b;
    relaxn;
endmodule
module nand_gate(x, a, b);
    input a, b;
    approximate output x;
    wire w0;
    and_gate a1(w0, a, b);
    assign x = ~ w0;
endmodule
```

```
module and_gate(n,a,b);
    input a, b;
    output n;
    assign n = a & b;
endmodule
module nand_gate(x, a, b);
    input a, b;
    approximate output x;
    wire w0;
    and_gate a1(w0, a, b);
    assign x = ~ w0;
    relax(x);
endmodule
```

On the left side, output n carries relaxed semantics due to the relax annotation and is therefore declared as an approximate output. Consequently, the a1 instance in the nand_gate module

will cause its x output to be relaxed. Therefore, x is marked as an approximate output. On the right side, the x output is explicitly relaxed and x is marked as an approximate output. The and_gate module here does not carry approximate semantics by default. Therefore, the output of the and_gate is not marked as approximate as the approximation is only limited to the a1 instance.

Critical inputs. A designer may want to prevent approximation from affecting certain inputs, which are critical to the functionality of the circuit. To mark these input ports, Axilog provides critical annotation. Wires that carry approximate semantics cannot drive the critical inputs without the designer's explicit permission at the time of reuse.

Bridging approximate wires to critical inputs. We recognize that there may be cases when the reusing designer entrusts a critical input with an approximate driver. For such situations, Axilog provides an annotation called bridge that shows designer's explicit intent to drive a critical input by an approximate signal.

In summary, the semantics of the relax and restrict annotations provide abstractions for designing approximate hardware modules while enabling Axilog to provide *formal guarantees* of safety that approximation will only be restricted to design elements that are specifically selected by the designer. Moreover, the approximate output, critical input, and bridge annotations enable reusability of modules across different designs. In addition to the modularity, the design and reuse annotations altogether enable *approximation polymorphism* implying that the modules with approximate semantics can be used in a precise manner and vice-versa without any reimplementation. These abstractions provide a natural extension to the current practices of hardware design and enable designers to apply approximation with full control without adding substantial overhead to the conventional hardware design and verification cycle.

III. Safety Inference Analysis

After the designer provides annotations, the compiler needs to perform a static analysis to find the approximate and precise design elements in accordance with these annotations. This section presents the *Safety Inference Analysis*, a static analysis that identifies these safe-to-approximate design elements. The design elements are primarily organized according to the structure of the circuit and not necessarily on the order of the statements in the HDL source code. This property is a fundamental property of Verilog that is inherited by Axilog. Thus, we first translate the RTL design to primitive gates, while maintaining the module boundaries. Then, we apply the Safety Inference Analysis after the code is translated to primitive gates and the structure of the circuit is identified. Consequently, the Safety Inference Analysis can apply all the annotations while considering the structure of the circuit. We apply the *Safety Inference Analysis* that is a backward slicing algorithm that starts from the annotated wires and iteratively traverses the circuit to identify which wires must carry precise semantics. Subtracting the set of precise wires from all the wires in the circuit yields the safe-to-approximate set of wires. The gates that immediately drive these safe-to-approximate wires are the ones that the synthesis engine can approximate. Figure 1(a) illustrates the procedure that identifies the precise wires.

This procedure is a *backward-flow* analysis that has three phases: (1) The **first phase** identifies the *sink* wires, which are either unannotated outputs or wires *explicitly* annotated with restrict. The procedure then identifies the gates that are driving these sink wires and adds their input wires to the precise set. The algorithm repeats this step for the newly added wires until it reaches an input or an explicitly relaxed wire. However, this phase is only limited to the scope of the module-under-analysis; (2) **The second phase** identifies the relaxed outputs of the instantiated submodules. Due to the semantic differences between relax and relax_local, the output of a submodule will be considered relaxed if the following two conditions are satisfied. (a) The output drives another explicitly relaxed wire, which is not inferred due to a relax_local annotation; and (b) the output is not driving a wire already identified as precise. The algorithm automatically annotates these qualifying outputs as relaxed. The analysis repeats these two phases for all the instantiated submodules. For correct functionality of this analysis, all the module instantiations are distinct entities in the set $\mathbb{M}$ and are ordered hierarchically; (3) **In the final phase**, the algorithm marks any wire that affects a globally restricted wire as precise. Finally, the Safety Inference Analysis identifies the safe-to-approximate subset of the gates and wires with regards to the designer annotations. An approximation-aware synthesis tool can then generate an optimized netlist.

IV. Approximate Synthesis

In our framework, approximate synthesis involves two stages. (1) In the **first stage**, annotated Verilog source code is converted to a precise gate-level netlist while preserving the approximate annotations. The Safety Inference Analysis then identifies the safe-to-approximate subset of the design based on designer annotations. (2) In the **second stage**, the synthesis tool applies approximate synthesis and optimization techniques *only* to the safe-to-approximate subset of the circuit elements. The tool has the liberty to apply any approximate optimization technique including gate substitution, gate elimination, logic restructuring, voltage over-scaling, and timing speculation as it deems prudent. The objective is to minimize a combination of error, delay, energy, and area considering final quality requirements. As Figure 2 shows, we developed two approximate synthesis flows to evaluate Axilog. In the next subsections we describe these flows in detail.

A. AST: Approximate Synthesis through Relaxing Timing Constraints

This synthesis flow is applicable to current technology nodes and leverages commercial synthesis tools. As shown in Figure 2a, we first use Synopsys Design Compiler to synthesize the design with no approximation. We perform a multi-objective optimization targeting the highest frequency while minimizing power and area. We will refer to the resulting netlist as the baseline netlist and its frequency as the baseline frequency. We account for variability using Synopsys PrimeTimeVX which, given timing constraints, provides the probability of timing violations due to variations. In case of violation, the synthesis process is repeated by adjusting timing constraints until PrimeTimeVX confirms no violations. Second, as shown in Figure 2b, we only relax the timing constraints for the safe-to-approximate paths. We then extract the post-synthesis gate delay information in Standard Delay Format and perform gate-level timing simulations with a set

Inputs: $\mathbb{M}$: Set of all the ordered modules within the circuit
$\mathbb{R}$: Queue of all the globally restricted wires
Output: $\mathbb{P}$: Set of precise wires

Initialize $\mathbb{P} \leftarrow \emptyset$
for each $m_i \in \mathbb{M}$ **do**
I: Set of all inputs ports in m_i
A: Set of all wires annotated as relaxed wires in m_i
LA: Set of all wires annotated as locally relaxed wires m_i
$Sink$: Queue of all explicitly restricted wires in $m_i \cup$ Set of unannotated output ports
UW: Set of wires driven by modules that are instantiated within m_i

//Phase1: This loop identifies the m_i module's local precise wires (w_i)
Initialize $N \leftarrow \emptyset$ A set of relaxed wires in each module m_i
while ($Sink \neq \emptyset$) **do**
$w_i \leftarrow Sink$.dequeue()
if ($w_i \notin I$ **and** $w_i \notin (A \cup LA)$) **then**
if ($w_i \in UW$) **then**
N.append(w_i)
else
$\mathbb{P}$.append(w_i)
end if
$Sink$.enqueue(for all input wires of gate w_i in m_i)
end if
end while

//Phase 2: Identifying the relaxed wires (w_j) that are driven by the m_j submodules; the m_j submodules are the instantiated modules in m_i
for ($w_j \in UW$) **do**
if ($w_j \notin N$ **and** w_j drives wire $\in A$) **then**
$m_j \leftarrow$ module driving the wire w_j
$m_j.A$.append(w_j)
end if
end for
end for

//Phase 3: Identifying the precise wires (w_k) that are globally restricted
while ($\mathbb{R} \neq \emptyset$) **do**
$w_k \leftarrow \mathbb{R}$.dequeue()
$\mathbb{P}$.append(w_k)
$\mathbb{R}$.append(input wires of the gate that drive w_k)
end while

(a) Part of Safety Inference Analysis that identifies precise wires according to the designer's annotations

Require: K: Netlist for the entire circuit
Θ: Set of safe-to-approximate gates
Σ: Error bound on the approximate output
Ensure: $\Re$: Different gate sizes for safe-to-approximate gates

Initialize $\Re \leftarrow$ Minimum gate size
Initialize $\Psi \leftarrow \emptyset$ {Monte Carlo simulation map}
Initialize $\gamma \leftarrow \emptyset$ {Error propagation map}
Initialize $\Pi \leftarrow \emptyset$ {Primary inputs of the safe-to-approximate circuit}
Initialize $\delta \leftarrow \emptyset$ {Queue for primary inputs of the safe-to-approximate circuit}
Initialize $\Phi \leftarrow \emptyset$ {Primary outputs of the safe-to-approximate circuit}
Initialize $\beta \leftarrow \emptyset$ {Fan-in hash-map}

//Phase 1: Identifying inputs (Π) and outputs (Φ) of the safe-to-approximate subset of the circuit. //
for each $m_i \in \Theta$ **do**
if fanin_of $m_i \not\subset \Theta$ **then**
$\Pi \leftarrow (\Pi \cup \{m_i\})$
enqueue(δ,m_i)
else if m_i fanout $\not\subset \Theta$ **then**
$\Phi \leftarrow (\Phi \cup \{m_i\})$
end if
end for

//Phase 2: Performing Monte Carlo Simulations to calculate probability of 1 or 0 (Ψ) at every node
$\Psi \leftarrow$ monte_carlo_simulation (δ,K,Θ,Ψ)

//Calculating the initial error map (γ) for every output node using boolean error propagation
$\gamma \leftarrow$ boolean_error_propagation (δ,K,Θ,Ψ,γ)

while ($\exists\ w_i \in \Phi$ s.t. $\Sigma(w_i) < \gamma(w_i)$) **do**
//Phase 3: Iteratively calculating the fan-in of every output node using back-propagation and adding the gates to (β)
while ($\exists\ w_i \in \Phi$ s.t $\Sigma(w_i) < \gamma(w_i)$) **do**

$\beta \leftarrow$ Gates $\in \Theta$ that have a path to w_i
$\delta \leftarrow$ Primary inputs $\in \Theta$ that have a path to w_i
define m -999 **//Max sensitivity initialized**

//Phase 4: Calculates the sensitivity of each gate to the output error and permanently resizes the gate with highest sensitivity
$G \leftarrow \emptyset$
for each $y_i \in \beta$ **do**
if (sensitivity of $y_i > m$) **then**
m = sensitivity of y_i
$G \leftarrow y_i$
end if
end for
$\Re(G) \leftarrow \Re(G) * 2$ **//up-size gate permanently**
$\gamma \leftarrow$ boolean_error_propagation (δ,K,Θ,Ψ,γ)
end while
end while

(b) Algorithm up-sizes the least number of gates in a circuit to reduce cost

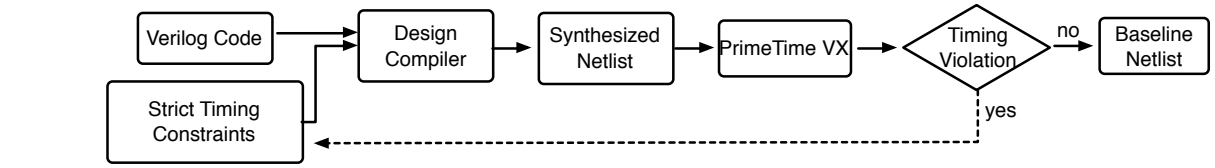
Fig. 1: (a) Part of the Safety Inference Analysis that finds precise wires. (b) Gate sizing algorithm for ASG approximate synthesis flow.

of input datasets. We use the baseline frequency for the timing simulations even though some of the safe-to-approximate paths are synthesized with more timing slack. Timing simulations yield output values that may incur quality loss at the baseline frequency. We then measure the quality loss and if the quality loss is more than designer's requirements, we tighten the timing constraints on the safe-to-approximate paths. We repeat this step until the quality requirements are satisfied. This methodology has a potential to reduce energy and area by utilizing slower and smaller gates for the paths which use relaxed timing constraints.

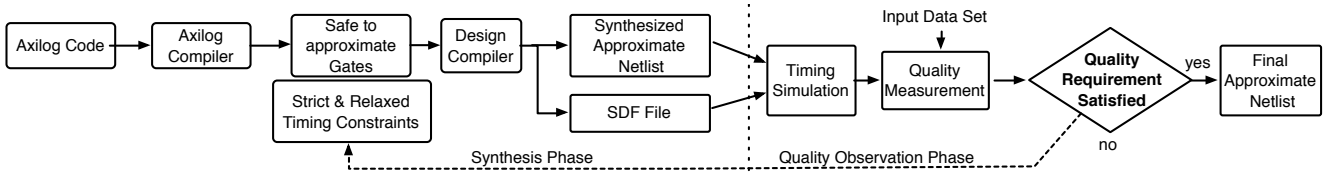
B. ASG: Approximate Synthesis through Gate Resizing

The ASG synthesis flow studies the potential of approximate synthesis for future technology nodes. As the characteristics of transistors and gates for future technologies are unknown, we assume that the probability of error for a gate is an inverse function of its size. As a result, gate size, referred to as the PCMOS [16] area, should be treated as a proxy for the cost we would pay in a future technology node to get more robust gates. That cost could be, thicker gate oxides, higher threshold voltage and higher V_{dd} to make the transistors more robust. The ASG synthesis flow applies approximation by selectively down-sizing the gates as shown in Figure 2c. In this framework, smaller gates dissipate less energy and have smaller PCMOS area, however, may generate incorrect output with some probability. We now describe in detail the probabilistic error model for the gates.

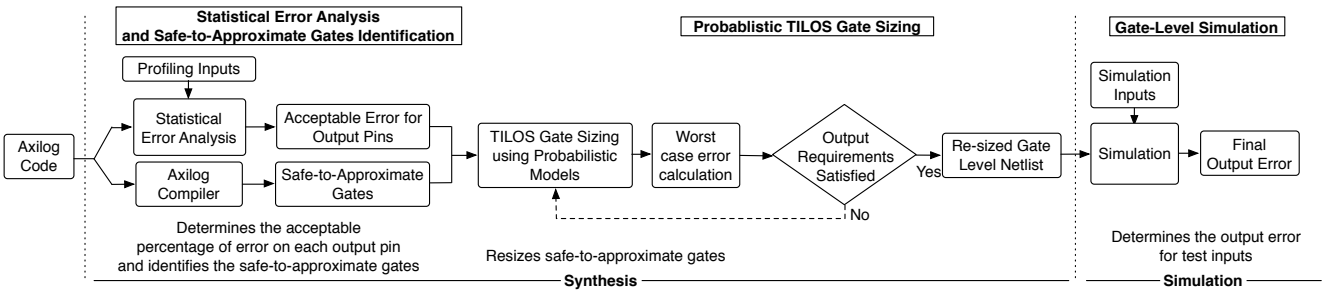
Probabilistic error models for gates. Due to the unavailability of future nodes, we augment a currently available library–NanGate FreePDK 45 nm–with a probabilistic error model for all the gates the library. The error model provides the probability of a bit flip in the gate output. We use transistor-level SPICE simulations to find the probability of an error at



(a) Synthesis flow for baseline error-free gate netlist



(b) Synthesis flow for approximation by relaxing the timing (AST) for safe-to-approximate gates



(c) Synthesis flow for approximation using gate resizing (ASG) that uses probabilistic models as a proxy for future nodes

Fig. 2: Synthesis flow for (a) baseline, (b) approximation using AST (c) approximation using ASG.

the gate output using the Cadence Virtuoso toolset. We take inspiration from the PCMOS models described in [16]. We simulated each gate at different sizes and the gate inputs were injected with Gaussian noise through a minimum-sized buffer. Gate error is also dependent on threshold voltage, however, we focused on gate sizing and its effects on gate error for a fixed threshold voltage. For each input combination, the noise was injected on gate inputs in the form of a Piece Wise Linear voltage source and the output was sampled for 10,000 inputs. Finally, the probability of correct output was computed as follows.

$$P_{\text{correct output}} = 1 - \frac{\text{Number of Incorrect Samples}}{\text{Total Number of Samples}} \quad (1)$$

We repeat this measurement for all the input combinations of the gate and assign the gate with the worst observed error. Next, we use this error model to optimize the power and area of the circuit by up-sizing the least number of gates in a circuit while satisfying the error requirements specified by the designer.

Algorithm 2: Gate sizing optimization. The ASG optimization algorithm shown in Figure 1(b) trades off accuracy for reduction in PCMOS area and energy. We extended the TILOS algorithm [17] to incorporate probabilistic models and changed the objective from minimizing delay to minimizing error and cost. The ASG optimization algorithm comprises of four phases.

In the first phase we extract the adjacency list (a space efficient way of representing a circuit) of the safe-to-approximate subcircuit and determine its inputs and outputs.

In the second phase a Monte-Carlo simulation is used to determine the error-free probability of obtaining a 1 or a 0 at each node of the subcircuit. For the Monte-Carlo simulation, random input vectors are applied to the inputs of the subcircuit and a topological traversal propagates the values through the circuit for each input vector. This process gives us the probability of getting a 1 or 0 at the output of each gate. We then initialize all gates in the safe-to-approximate subcircuit to their minimum size i.e. having maximum error. We calculate the initial error map γ at the output of each gate by propagating the error through the circuit using the Boolean Error Propagation (BEP) algorithm [18]. The boolean error propagation algorithm then estimates the worst-case error probability for the outputs of the design based on each gates error porbability model. If the calculated output error was not within the error requirements we entered phase 3.

In the **third phase**, for each safe-to-approximate output, we identify the gates that are driving that output, called the fan-in-cone, and add it to the fan-in hashmap β.

In the **fourth phase**, for each gate in the fan-in-cone of safe-to-approximate output, we calculate the sensitivity of the output error to that gate. The sensitivity is measured by temporarily increasing the size of the gate to the next possible size and calculating the ratio of decrease in error to increase in gate size. Finally, after calculating the sensitivity for each fan-in gate we permanently up-size only the gate that shows the *largest impact* towards the output error. We perform the BEP using the changed gate size and update the error map γ. We repeat the fourth phase for each safe-to-approximate output, until user specified error bounds are satisfied for each safe-to-approximate output. The most compute intensive part of the algorithm is the Phase 3's boolean_error_propgation function

TABLE II: Benchmarks, input datasets, and error metrics.

Benchmark Name	Domain	Input Data Set	Quality Metric	# of Lines	# of Annotations	
					Design	Reuse
Brent-Kung (32-bit adder)	Arithmetic Computation	1,000,000 32-bit integers	Avg Relative Error	352	1	1
FIR (8-bit FIR filter)	Signal Processing	1,000,000 8-bit integers	Avg Relative Error	113	6	5
ForwardK (forward kinematics for 2-joint arm)	Robotics	1,000,000 32-bit fixed-point values	Avg Relative Error	18,282	5	4
InverseK (inverse kinematics for 2-joint arm)	Robotics	1,000,000 32-bit fixed-point values	Avg Relative Error	22,407	8	4
K-means (K-means clustering)	Machine Learning	1024x1024-pixel color image	Image Diff	10,985	7	3
Kogge-Stone (32-bit adder)	Arithmetic Computation	1,000,000 32-bit integers	Avg Relative Error	353	1	1
Wallace Tree (32-bit Multiplier)	Arithmetic Computation	1,000,000 32-bit integers	Avg Relative Error	13,928	5	3
Neural Network (feedforward neural network)	Machine Learning	1024x1024-pixel color image	Image Diff	21,053	4	3
Sobel (sobel edge dectector)	Image Processing	1024x1024-pixel color image	Image Diff	143	6	3

with a complexity of $O(n^3)$. We optimized this function and reduced its complexity to $O(n^2)$ by decreasing the its iteration count by grouping gates together. These groups are resized together.

In the next section, we evaluate Axilog and the approximate synthesis processes with a set of benchmark designs.

V. Evaluation

Benchmarks and Code Annotation. Table II lists the Verilog benchmarks. We use Axilog annotations to judiciously relax some of the circuit elements. The benchmarks span a wide range of domains including arithmetic units, signal processing, robotics, machine learning, and image processing. Table II also includes the input datasets, application-specific quality metrics, the number of lines, and the number of Axilog annotations for design and reuse.

Axilog annotations. We annotated the benchmarks with the Axilog extensions. The designs were either downloaded from open-source IP providers or developed without any initial annotations. After development, we analyzed the source Verilog codes to identify safe-to-approximate parts. The last two columns of Table II show the number of design and reuse annotations for each benchmark. The number of annotations range from 2 for Brent-Kung with 352 lines to 12 for InverseK with 22,407 lines. The Axilog framework enabled us to only use a handful of annotations to effectively approximate designs that are implemented with thousands of lines of Verilog.

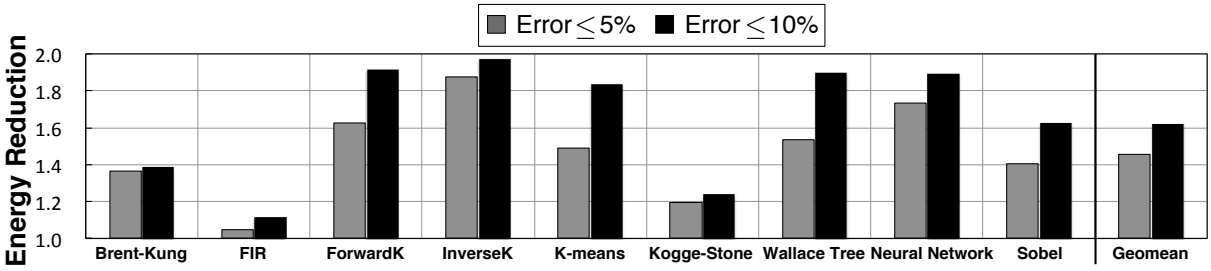
The safe-to-approximate parts are more common in datapaths of the benchmarks rather than their control logic. For example, K-means involves a large number of multiplications and additions. We used the relax annotations to declare these arithmetic operations approximiable; however, we used restrict to ensure the precision of all the control signals. For smaller benchmarks, such as Brent-Kung, Kogge-Stone and Wallace Tree, only a subset of the least significant output bits were annotated to limit the quality loss. We also annotated the benchmarks with reuse annotations. The number of this type of annotation are listed in the last column of Table II. Overall, one graduate student was able to annotate all the benchmarks within two days without being involved in their design. The intuitive nature of Axilog extensions makes annotating straightforward.

Application-specific quality metrics. Table II shows the application-specific error metrics to evaluate the quality loss due to approximation. Using application-specific quality metrics is commensurate with prior work on approximate computing and language design [19,20]. In all cases, we compare the output of the original baseline application to the output of the approximated design.

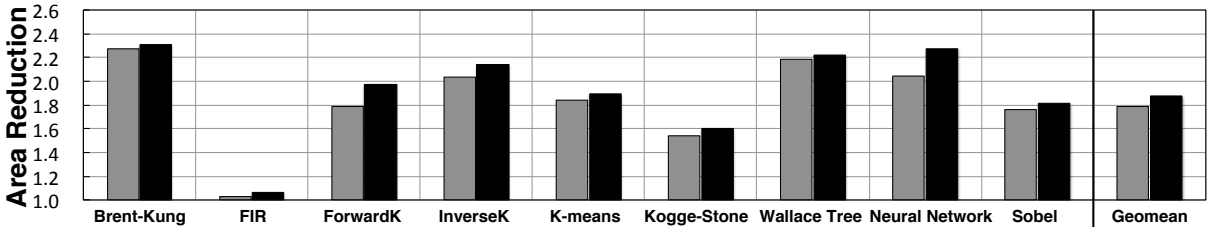
Experimental results. Both the synthesis techniques use Synopsys Design Compiler (G-2012.06-SP5) and Synopsys PrimeTime (F-2011.06-SP3-2) for synthesis flows and energy analysis, respectively.

AST Evaluation. We used Cadence NC-Verilog (11.10-s062) for timing simulation with SDF back annotations extracted from various operating corners. We use the TSMC 45-nm multi-V_t standard cells libraries and the primary results are reported for the slowest PVT corner (SS, 0.81V, 0°C). The AST approach generates approximate netlists for the current technology node and provides, on average, 1.45× energy and 1.8× area reduction for the 5% limit. With the 10% limit, the average energy and area gains grow to 1.54× and 1.82× as shown in Figure 3a and 3b.

Benchmarks such as InverseK, Wallace Tree, Neural Network, and Sobel–that have a larger datapath–provide a larger scope for approximation and are usually the ones that see larger benefits. The structure of the circuit also affects the potential benefits. For instance, Brent-Kung and Kogge-Stone adders benefit differently from approximation due to the structural differences in their logic trees. The FIR benchmark shows the smallest energy savings since it is a relatively small design which does not provide many opportunities for approximation. Nevertheless, FIR still achieves 11% energy savings and 7% area reduction with 10% quality loss, suggesting that even designs with limited opportunities for approximation can benefit significantly from Axilog.



(a) Energy Reduction = (Precise circuit energy)/(Approximate circuit energy)

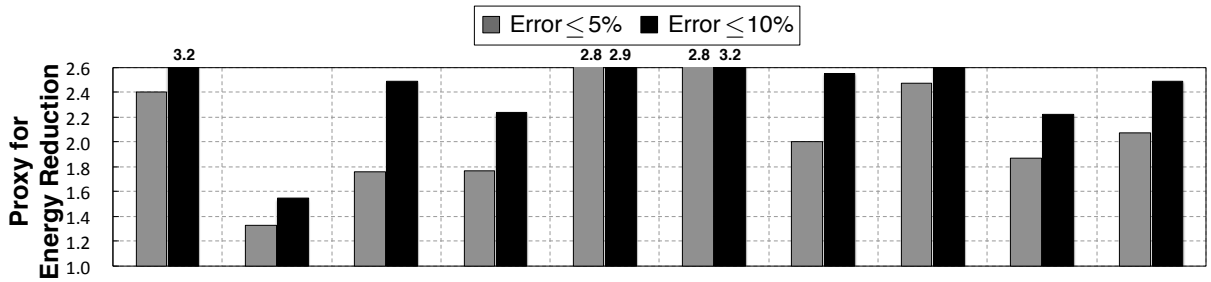


(b) Area Reduction = (Precise circuit area)/(Approximate circuit area)

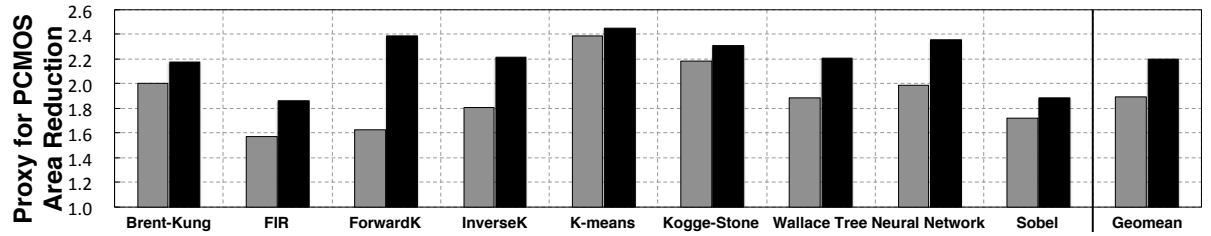
PVT Corners	Brent-Kung	FIR	ForwardK	InverseK	K-means	Kogge-Stone	Wallace Tree	Neural Network	Sobel	Geomean
(SS, 0.81V, 0°C)	34%	11%	78%	87%	69%	24%	65%	83%	57%	54%
(SS, 0.81V, 125°C)	32%	7%	72%	79%	65%	21%	63%	72%	41%	48%

(c) Energy reduction when the quality degradation limit is set to 10% for two different PVT corners. Here, we consider temperature variations.

AST Synthesis Flow: reductions in (a) energy and (b) area when the quality degradation limit is set to 5% and 10%. (c) Energy reduction for two different PVT corners.



(d) Proxy for Energy Reduction = (Precise circuit energy)/(Approximate circuit energy)



(e) Proxy for PCMOS area Reduction = (Precise circuit PCMOS area)/(Approximate circuit PCMOS area)

ASG Synthesis Flow: reductions in (d) energy and (e) area when the quality degradation limit is set to 5% and 10% for the ASG synthesis flow.

Fig. 3: (a, b, c) Energy and Area reduction for AST flow. (d,e) Energy and PCMOS area reduction for ASG flow.

We also evaluated the effectiveness of our AST technique in the presence of temperature variations for a full industrial range of 0°C to 125°C. We measured the impact of temperature fluctuations on the energy benefits for the same relaxed designs. Table 3c compares the energy benefits at the lower and higher temperatures (the quality loss limit is set to 10%). In this range of temperature variations, the average energy benefits ranges from 1.54× (at 0°C) to 1.48× (at 125°C). These results confirm the robustness of our framework; it yields significant benefits even when temperature varies.

ASG Evaluation. We used the NanGate FreePDK 45 nm multi-speed standard cells library. The AST and ASG techniques use different libraries because FreePDK 45 nm library allowed SPICE simulations required for the ASG flow. As mentioned before, the ASG flow aims to study the rends in future technology nodes when gates might show probabilistic behavior. We develop PCMOS models with the available libraries at 45 nm. The area numbers reported here are the ones set by the PCMOS model to satisfy the fixed gate robustness. These numbers do not necessarily correspond to actual area numbers in any future technology. The PCMOS area shows the relative cost savings across benchmarks and delineate the anticipated trends. As shown in Figures 3d and 3e, the ASG flow, provides, on average, 2× energy and 1.9× PCMOS area reduction for the 5% error limit. With the 10% limit, the average energy and area gains grow to 2.5× and 2.2×.

Summary. Both the synthesis frameworks show the effectiveness of Axilog and achieve significant savings while preserving the application functionality. This tradeoff is attainable because of the high-level language annotations and design abstractions allow the designer to target approximation where it is most effective without compromising the critical parts of the computation.

VI. Related Work

A growing body of research shows the applicability and significant benefits of approximation [1–15]. However, prior research has not explored extending hardware description languages for systematic and reusable approximate hardware design. Below, we discuss the most related works.

Approximate programming languages. EnerJ [19] provides a set of type qualifier to manually annotate all the approximate variables in the program. If we had extended EnerJ's model to Verilog, the designer would have had to manually annotate all approximate wires/regs. Rely [20] asks for manually marking both approximate variables and operations, which requires more annotations. With our annotations, the designer marks a few wires/regs and then the analysis automatically infers which other connections and gates are safe to approximate.

Approximate circuit design and synthesis. Prior work proposes imprecise implementations of custom instructions and specific hardware blocks [3,4,6–9]. The work in [5,10–15] propose algorithms for approximate synthesis that leverages gate pruning, timing speculation, or voltage over-scaling. While all these synthesis techniques provide significant improvements, they do not focus on approximate hardware design and reuse. In fact, our framework can benefit and leverage all these synthesis techniques.

VII. Conclusion

Axilog's automated analysis enables approximate hardware design and reuse without exposing the intricacies of synthesis and optimization. Furthermore, all the abstractions presented in this paper are concrete extensions to the mainstream Verilog HDL providing designers with backward compatibility. We evaluated Axilog, its automated Safety Inference Analysis, and presented two approximate synthesis techniques. Both flows demonstrate significant cost savings with merely 2 to 12 annotations per benchmark. These results confirm that Axilog is a methodical step toward practical approximate hardware design and reuse.

VIII. Acknowledgments

The authors thank the anonymous reviewers for their insightful comments. This work was supported in part by Semiconductor Research Corporation (SRC) contract #2014-EP-2577, the Qualcomm Innovation Fellowship, and a gift from Google.

References

[1] L. N. Chakrapani, P. Korkmaz, B. E. Akgul, and K. V. Palem, "Probabilistic system-on-a-chip architectures," *in TODAES*, 2007.
[2] H. Cho, L. Leem, and S. Mitra, "Ersa: Error resilient system architecture for probabilistic applications," *in TCAD*, 2012.
[3] V. Gupta *et al.*, "IMPACT: imprecise adders for low-power approximate computing," in *ISLPED*, 2011.
[4] R. Ye, T. Wang, F. Yuan, R. Kumar, and Q. Xu, "On reconfiguration-oriented approximate adder design and its application," in *ICCAD*, 2013.
[5] D. Shin and S. K. Gupta, "Approximate logic synthesis for error tolerant applications," in *DATE*, 2010.
[6] P. Kulkarni, P. Gupta, and M. Ercegovac, "Trading accuracy for power with an underdesigned multiplier architecture," in *VLSI*, 2011.
[7] A. Kahng and S. Kang, "Accuracy-configurable adder for approximate arithmetic designs," in *DAC*, 2012.
[8] D. Mohapatra *et al.*, "Design of voltage-scalable meta-functions for approximate computing," in *DATE*, 2011.
[9] S.-L. Lu, "Speeding up processing with approximation circuits," *Computer*, 2004.
[10] S. Venkataramani *et al.*, "Salsa: systematic logic synthesis of approximate circuits," in *DAC*, 2012.
[11] K. Nepal, Y. Li, R. Bahar, and S. Reda, "Abacus: a technique for automated behavioral synthesis of approximate computing circuits," in *DATE*, 2014.
[12] Y. Liu *et al.*, "On logic synthesis for timing speculation," in *ICCAD*, 2012.
[13] A. Lingamneni *et al.*, "Algorithmic methodologies for ultra-efficient inexact architectures for sustaining technology scaling," in *CF*, 2012.
[14] J. Miao, A. Gerstlauer, and M. Orshansky, "Approximate logic synthesis under general error magnitude and frequency constraints," in *ICCAD*, 2013.
[15] S. Ramasubramanian *et al.*, "Relax-and-retime: A methodology for energy-efficient recovery based design," in *DAC*, 2013.
[16] S. Cheemalavagu, P. Korkmaz, K. V. Palem, B. E. S. Akgul, and L. N. Chakrapani, "A probabilistic cmos switch and its realization by exploiting noise," in *the Proceedings of the IFIP international*, 2005.
[17] S. S. Sapatnekar, V. B. Rao, P. M. Vaidya, and S. mo Kang, "An exact solution to the transistor sizing problem for cmos circuits using convex optimization," *IEEE Transactions on Computer-Aided Design*, vol. 12, pp. 1621–1634, 1993.
[18] N. Mohyuddin, P. Ehsan, and P. Massoud, *Probabilistic error propagation in a logic circuit using the boolean difference calculus*. Advanced Techniques in Logic Synthesis, Optimizations and Applications, 2011.
[19] A. Sampson *et al.*, "EnerJ: Approximate data types for safe and general low-power computation," *PLDI*, 2011.
[20] M. Carbin, S. Misailovic, and M. Rinard, "Verifying quantitative reliability of programs that execute on unreliable hardware," 2013.