Axilog: Abstractions for Approximate Hardware Design and Reuse

Divya Mahajan
Georgia Institute of Technology

Kartik Ramkrishnan
Rudra Jariwala
University of Minnesota

Amir Yazdanbakhsh
Jongse Park
Bradley Thwaites
Anandhavel Nagendrakumar
Georgia Institute of Technology

Abbas Rahimi
University of California, San Diego

Hadi Esmaeilzadeh
Georgia Institute of Technology

Kia Bazargan
University of Minnesota

Relaxing the traditional abstraction of "near-perfect" accuracy in hardware design can yield significant gains in efficiency, area, and performance. Axilog, a set of language extensions for Verilog, provides the necessary syntax and semantics for approximate hardware design and reuse, letting designers safely relax accuracy requirements in the design while keeping the critical parts strictly precise.

Several techniques have shown significant benefits with approximation at the circuit level (see the "Related Work in Approximation" sidebar), but they lack design abstractions that enable designers to methodically control which parts of a circuit can be approximated while keeping the critical parts precise. Thus, a need persists for approximate hardware description languages enabling systematic synthesis of approximate hardware. To meet this need, we introduce Axilog, a set of concise, intuitive, and high-level annotations that provide 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 ones are critical and therefore 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 in which engineers build libraries of modules and reuse them across complex hardware systems. In this article, we elaborate on the Axilog annotations for approximate hardware design and reuse. These annotations are coupled with a safety inference analysis (SIA) that automatically infers which circuit elements are safe to approximate with respect to the designer's annotations. Axilog and safety analysis support approximate synthesis and are completely independent of the synthesis process.

To evaluate Axilog, we devised two synthesis processes. 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. Results show that this synthesis flow provides, on average, 1.54× energy savings and 1.82× area reduction by allowing a 10 percent quality loss. The second synthesis flow studies the potential of approximate synthesis

Published by the IEEE Computer Society

Related Work in Approximation

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.

Approximate programming languages

EnerJ provides a set of type qualifiers to manually annotate all the approximate variables in the program.[16] If we had extended EnerJ's model to Verilog, the designer would have had to manually annotate all approximate wires and registers. Rely asks for manually marking both approximate variables and operations, which requires more annotations.[17] With our annotations, the designer marks a few wires and registers, 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] Other recent work proposes algorithms for approximate synthesis that leverages gate pruning, timing speculation, or voltage overscaling.[5,10–15] Although 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.

References

1. L.N. Chakrapani et al., "Probabilistic System-on-a-Chip Architectures," *ACM Trans. Design Automation of Electronic Systems*, vol. 12, no. 3, 2007, article 29.
2. H. Cho, L. Leem, and S. Mitra, "ERSA: Error Resilient System Architecture for Probabilistic Applications," *IEEE Trans. Computer-Aided Design of Integrated Circuits and Systems*, vol. 31, no. 4, 2012, pp. 546–558.
3. V. Gupta et al., "IMPACT: Imprecise Adders for Low-Power Approximate Computing," *Proc. 17th IEEE/ACM Int'l Symp. Low-Power Electronics and Design*, 2011, pp. 409–414.
4. R. Ye et al., "On Reconfiguration-Oriented Approximate Adder Design and its Application," *Proc. Int'l Conf. Computer-Aided Design*, 2013, pp. 48–54.
5. D. Shin and S.K. Gupta, "Approximate Logic Synthesis for Error Tolerant Applications," *Proc. Conf. Design, Automation, and Test in Europe*, 2010, pp. 957–960.
6. P. Kulkarni, P. Gupta, and M. Ercegovac, "Trading Accuracy for Power with an Underdesigned Multiplier Architecture," *Proc. 24th Int'l Conf. VLSI Design*, 2011, pp. 346–351.
7. A. Kahng and S. Kang, "Accuracy-Configurable Adder for Approximate Arithmetic Designs," *Proc. 49th Ann. Design Automation Conf.*, 2012, pp. 820–825.
8. D. Mohapatra et al., "Design of Voltage-Scalable Meta-Functions for Approximate Computing," *Proc. Int'l Conf. Computer-Aided Design*, 2011, pp. 667–673.
9. S.-L. Lu, "Speeding Up Processing with Approximation Circuits," *Computer*, vol. 37, no. 3, 2004, pp. 67–73.
10. S. Venkataramani et al., "SALSA: Systematic Logic Synthesis of Approximate Circuits," *Proc. 49th Ann. Design Automation Conf.*, 2012, pp. 796–801.
11. K. Nepal et al., "ABACUS: A Technique for Automated Behavioral Synthesis of Approximate Computing Circuits," *Proc. Conf. Design, Automation, and Test in Europe*, 2014, article 361.
12. Y. Liu et al., "On Logic Synthesis for Timing Speculation," *Proc. Int'l Conf. Computer-Aided Design*, 2012, pp. 591–596.
13. A. Lingamneni et al., "Algorithmic Methodologies for Ultra-Efficient Inexact Architectures for Sustaining Technology Scaling," *Proc. 9th Conf. Computing Frontiers*, 2012, pp. 3–12.
14. J. Miao, A. Gerstlauer, and M. Orshansky, "Approximate Logic Synthesis under General Error Magnitude and Frequency Constraints," *Proc. Int'l Conf. Computer-Aided Design*, 2013, pp. 779–786.
15. S. Ramasubramanian et al., "Relax-and-Retime: A Methodology for Energy-Efficient Recovery Based Design," *Proc. Int'l Conf. Computer-Aided Design*, 2013, pp.779–786.
16. A. Sampson et al., "EnerJ: Approximate Data Types for Safe and General Low-Power Computation," *Proc. 32nd ACM Sigplan Conf. Programming Language Design and Implementation*, 2011, pp. 164–174.
17. M. Carbin, S. Misailovic, and M. Rinard, "Verifying Quantitative Reliability of Programs that Execute on Unreliable Hardware," *Proc. ACM Sigplan Int'l Conf. Object Oriented Programming Systems Languages & Applications*, 2013, pp. 33–52.

by using a probabilistic gate model for future technology nodes. This synthesis flow provides, on average, 2.5× energy and 2.2× probabilistic CMOS (PCMOS) area reduction. Axilog yields these significant benefits while only requiring between two to 12 annotations, even with complex designs containing up to 22,407 lines of code. These results confirm Axilog's effectiveness in incorporating approximation in the hardware design cycle.

Approximate hardware design with Axilog

Our principal objectives for approximate hardware design with Axilog are to

Table 1. Summary of Axilog's language syntax.

Phase	Annotation	Argument	Description
Design	`relax`	Wire, reg, output, inout	Declare an argument as safe to approximate. Design elements 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 that the output carries relaxed semantics.
	`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.

- craft a small number of Verilog annotations that provide designers with complete oversight over the approximation process;
- minimize the number of manual annotations while relying on SIA to automatically infer the designer's intent for approximation, thereby relieving the designer of the details of the approximate synthesis process; and
- 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 for the design and one for the reuse of hardware modules. Table 1 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 let designers use the annotated approximate modules across various designs without any reimplementation. We provide detailed examples to illustrate how designers can 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.

Design annotations relaxing accuracy requirements

By default, all design elements are precise. The designer can use the `relax(arg)` 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 following example illustrates the use of `relax`:

```
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 are not approximated. Furthermore, because `s` will carry relaxed semantics, its corresponding output is marked with the `approximate` annotation that is necessary for reusing modules. 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 dataflows within the circuit.

Scope of approximation. The scope of the `relax` annotation crosses the boundaries of instantiated modules, as shown in Figure 1.

The `relax(x)` annotation in the `nand_gate` module in Figure 1a 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, which does not cross module boundaries.

On the other hand, the code in Figure 1b 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 of `relax` and `relax_local` is the module in which they are declared.

```
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
```

(a)

```
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
```

(b)

Figure 1. Code segments showing the difference in the scope of `relax` and `relax_local` annotation. (a) Scope of `relax` annotation. (b) Scope of `relax_local` annotation.

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, which 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. In some cases, the designer might intend 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, which 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.

Reuse annotations

Our principal idea for these language abstractions is to maximize the reusability of the approximate modules across designs that might have different accuracy requirements.

Outputs carrying approximate semantics. As we mentioned earlier, designers can use annotations to selectively approximate design elements in a module. The reusing designer must be aware of the accuracy semantics of the I/O ports without delving into the details of the module. To enable the reusing designer to view the port semantics, Axilog requires that

```
module and_gate(n,a,b);
        input a,b;
        approximate output n;
        assign n = a & b;
        relax(n);
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
(a)

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);
        assignx = ~w0;
        relax(x);
endmodule
(b)
```

Figure 2. Code segments showing the necessity of the `approximate` annotation. (a) Approximate output when `relax` annotation applied within the submodule (`and_gate`). (b) Approximate output when relax annotation is applied within the main module (`nand_gate`).

all output ports that might be influenced by approximation be marked as `approximate`. The two code snippets in Figure 2 illustrate the necessity of the `approximate` annotation. In Figure 2a, 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.

In Figure 2b, 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`, because the approximation is limited to the `a1` instance.

Critical inputs. A designer might want to prevent approximation from affecting certain inputs, which are critical to the circuit's functionality. 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.

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 be restricted to only those design elements that the designer specifically selected. 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 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 naturally extend current hardware-design practices and let designers apply approximation with full control without adding substantial overhead to the conventional hardware design and verification cycle.

Safety inference analysis

After the designer provides annotations, the compiler must perform a static analysis to find the approximate and precise design elements in accordance with these annotations. We present the SIA, a static analysis that identifies these safe-to-approximate design elements. The design elements are organized primarily according to the circuit's structure and not necessarily on the order of the statements in the HDL source code. This property is a fundamental property of Verilog that Axilog inherited. Thus, we first translate the RTL design to primitive gates, while maintaining the module boundaries. Then, we apply the SIA after the code is translated to primitive gates and the structure of the circuit is identified. Consequently, the SIA can apply all the annotations while considering the circuit's structure. The SIA 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 3 illustrates the procedure that identifies the precise wires.

This procedure is a backward-flow analysis that has three phases. 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 limited to the scope of the module under analysis.

The second phase identifies the relaxed outputs of the instantiated submodules. Because of the semantic differences between `relax` and `relax_local`, a submodule's output will be considered relaxed if two conditions are satisfied:

- the output drives another explicitly relaxed wire, which is not inferred due to a `relax_local` annotation; and
- 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 M and are ordered hierarchically.

In the final phase, the algorithm marks any wire that affects a globally restricted wire as precise. Finally, the SIA 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.

Approximate synthesis

In our framework, approximate synthesis involves two stages. In the first stage, annotated Verilog source code is converted to a precise gate-level netlist while preserving the approximate annotations. The SIA then identifies the safe-to-approximate subset of the design based on designer annotations. 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 may apply any approximate optimization technique—including gate substitution, gate elimination, logic restructuring, voltage overscaling, 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 4 shows, we developed two approximate synthesis flows to evaluate Axilog.

AST: Approximate synthesis through relaxing timing constraints

The AST synthesis flow is applicable to current technology nodes and leverages commercial synthesis tools. As Figure 4a shows, we first use the Synopsys Design Compiler to synthesize the design with no approximation. We perform a multiobjective 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 Figure 4b shows, we relax the timing constraints only for the safe-to-approximate paths. We then extract the

```
Inputs:  𝕄 : Set of all the ordered modules within the circuit
       ℝ: Queue of all the globally restricted wires
Output:  ℙ: Set of precise wires
       Initialize ℙ → ∅
       for each m_i ∈ 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 ∪ Set of
unannotated output ports
          UW: Set of wires driven by modules which are instantiated within m_i

          //Phase1: This loop identifies the m_i module's local precise wires (w_i)
          Initialize N ←∅ A set of relaxed wires in each module m_i
          while (Sink ≠∅) do
             w_i ← Sink.dequeue()
             if (w_i ∉ I and w_i ∉ (A ∪ LA)) then
                if (w_i ∈ UW) then
                   N.append(w_i)
                else
                   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 ∈ UW) do
             if (w_j ∈ N and w_j drives wire ∈ A) then
                m_j ←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 (ℝ ≠ ∅) do
       w_k ←ℝ.dequeue()
       ℙ.append(w_k)
          ℝ.append(input wires of the gate that drive w_k)
end while
```

Figure 3. The part of the safety inference analysis (SIA) that identifies precise wires according to the designer's annotations.

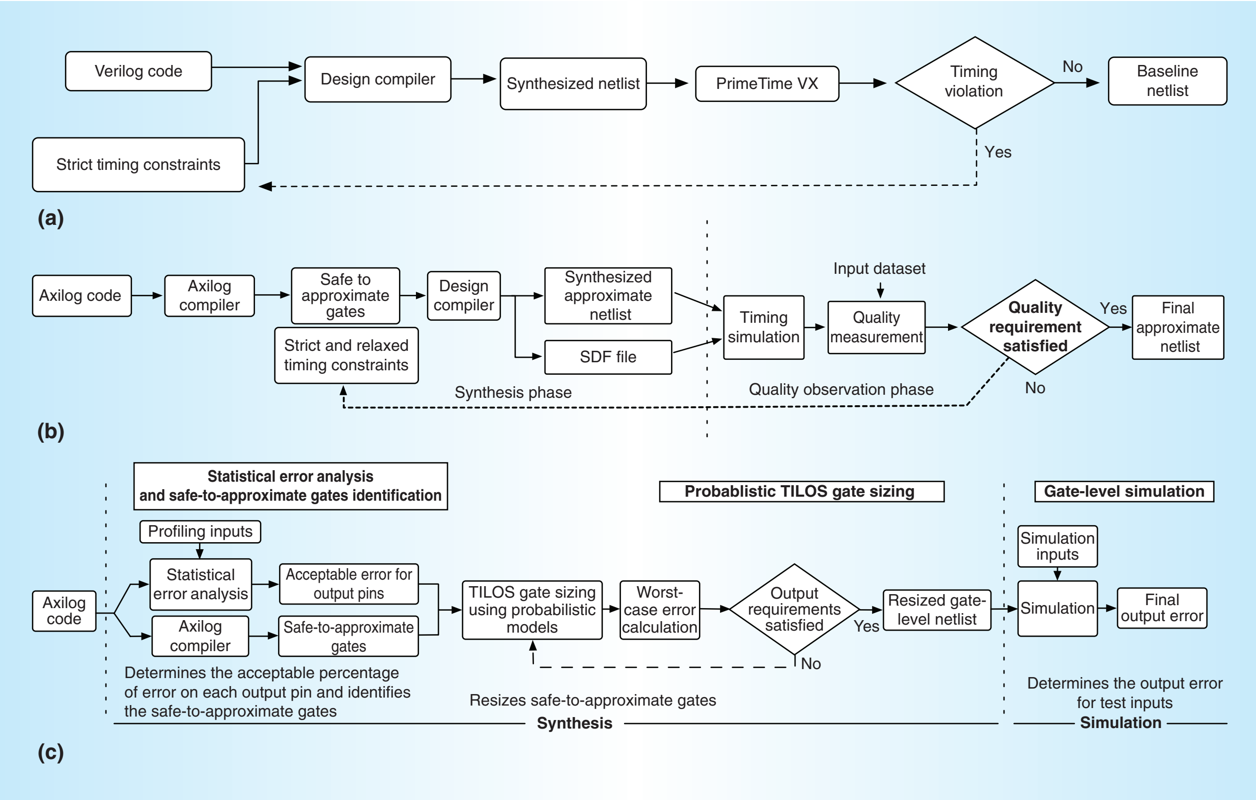


Figure 4. Synthesis flows for the baseline, for approximate synthesis through relaxing timing constraints (AST), and for approximate synthesis through gate resizing (ASG). (a) Synthesis flow for baseline error-free gate netlist. (b) Synthesis flow AST for safe-to-approximate gates. (c) Synthesis flow for ASG, which uses probabilistic models as a proxy for future nodes.

post-synthesis gate delay information in Standard Delay Format and perform gate-level timing simulations with a set 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 higher than the 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 could reduce energy and area by using slower and smaller gates for the paths that use relaxed timing constraints.

ASG: Approximate synthesis through gate resizing

The ASG synthesis flow studies the potential of approximate synthesis for future technology nodes. Because 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 area,*[1] 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 flow V_{DD} to make the transistors more robust. The ASG and synthesis applies approximation by selectively downsizing the gates as shown in Figure 4c. In this framework, smaller gates dissipate less energy and have smaller PCMOS area, but they may generate incorrect output with some probability.

Probabilistic error models for gates. Owing 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 in 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 the gate output using the Cadence Virtuoso toolset. We take inspiration from the PCMOS models described by Cheemalavagu et al.[1]

We simulated each gate at different sizes and injected the gate inputs with Gaussian noise through a minimum-sized buffer. Gate error also depends 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 piecewise linear voltage source, and the output was sampled for 10,000 inputs. Finally, we computed the probability of correct output as follows:

$$P_{\text{correct output}} = 1 - \frac{\text{Number of incorrect samples}}{\text{Total number of samples}}.$$

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 circuit's power and area by upsizing the fewest gates in a circuit while satisfying the designer-specified error requirements.

Gate sizing optimization. The ASG optimization algorithm shown in Figure 5 trades off accuracy for reductions in PCMOS area and energy. We extended the Tilos algorithm[2] to incorporate probabilistic models and changed the objective from minimizing delay to minimizing error and cost.

The ASG optimization algorithm comprises 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, the algorithm uses a Monte Carlo simulation 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 subcircuit's inputs, 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 each gate's output. We then initialize all gates in the safe-to-approximate subcircuit to their minimum size (that is, having maximum error). We calculate the initial error map at each gate's output by propagating the error through the circuit using the Boolean Error Propagation (BEP) algorithm.[3] The BEP algorithm then estimates the worst-case error probability for the design's outputs using each gate's error probability model. If the calculated output error is not within the error requirements, we enter phase 3.

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

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 by temporarily increasing the gate's size 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 upsize only the gate that shows the largest impact toward the output error. We perform the BEP using the changed gate size and update the error map. 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 computationally intensive part of the entire algorithm is phase 3's BEP function, with a complexity of $O(n^3)$. We optimized this function and reduced its complexity to $O(n^2)$ by decreasing its iteration count by grouping gates together. These groups are resized together.

Evaluation

We evaluated Axilog and the approximate synthesis processes using a set of benchmark designs.

Benchmarks and code annotation

Table 2 lists the Verilog benchmarks. We used Axilog annotations to judiciously relax some of the circuit elements. The benchmarks span many domains, including arithmetic units, signal processing, robotics, machine learning, and image processing. Table 2 also includes the input datasets, application-specific quality metrics, number

```
Require: K: Netlist for the entire circuit
       Θ: Set of safe-to-approximate gates
       Σ: Error bound on the approximate output
Ensure:  ℜ: Different gate sizes for safe-to-approximate gates
       Initialize ℜ ← ∅ Minimum gate size
       Initialize ψ ← ∅ {Monte Carlo simulation map}
       Initialize γ ← ∅ {Error propagation map}
       Initialize Π ← ∅ {Primary inputs of the safe-to-approximate circuit}
       Initialize δ ← ∅ {Queue  for  primary  inputs  of  the  safe-to-approximate circuit}
       Initialize Φ ← ∅{Primary outputs of the safe-to-approximate circuit}
       Initialize β ← ∅{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 ∈ Θ do
         if fanin_of m_i ⊄ Θ then
           Π ← (Π ∪ {m_i})
           enqueue(δ, m_i)
         else if m_i fanout ⊄ Θ then
           Φ ← (Φ ∪ {m_i})
         end if
       end for

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

       //Calculating the initial error map (γ) for every output node using Boolean Error Propagation
       γ← boolean_error_propagation (δ, K, Θ, Ψ, γ)
       while (∃ w_i ∈ Φ s.t. Σ(w_i) < γ(w_i)) do

       //Phase 3: Iteratively calculating the fan-in of every output node using
       back-propagation and adding the gates to (β)
         while (∃ w_i ∈ Φ s.t. Σ(w_i) < γ(w_i)) do
           β ← Gates ∈ Φ that have a path to w_i
           δ ← Primary inputs ∈ Φ 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 ←∅
           for each y_i ∈ β do
             if (sensitivity of y_i > m) then
               m = sensitivity of y_i
               G ← y_i
             end if
           end for
           ℜ(G) ← ℜ(G)*2 //upsize gate permanently
           γ← boolean_error_propagation (δ, K, Θ, Ψ, γ)
         end while
       end while
```

Figure 5. Gate-sizing algorithm for approximate synthesis through gate resizing (ASG) approximate synthesis flow. The algorithm upsizes the fewest gates in a circuit to reduce cost.

Table 2. Benchmarks, input datasets, and error metrics.

Benchmark name	Domain	Input dataset	Quality metric	No. of lines	No. of annotations	
					Design	Reuse
Brent-Kung (32-bit adder)	Arithmetic computation	1,000,000 32-bit integers	Average relative error	352	1	1
FIR (8-bit finite impulse response filter)	Signal processing	1,000,000 8-bit integers	Average relative error	113	6	5
ForwardK (forward kinematics for two-joint arm)	Robotics	1,000,000 32-bit fixed-point values	Average relative error	18,282	5	4
InverseK (inverse kinematics for two-joint arm)	Robotics	1,000,000 32-bit fixed-point values	Average relative error	22,407	8	4
k-means (K-means clustering)	Machine learning	1,024-×-1,024-pixel color image	Image difference	10,985	7	3
Kogge-Stone (32-bit adder)	Arithmetic computation	1,000,000 32-bit integers	Average relative error	353	1	1
Wallace tree (32-bit multiplier)	Arithmetic computation	1,000,000 32-bit integers	Average relative error	13,928	5	3
Neural Network (feedforward neural network)	Machine learning	1,024-×-1,024-pixel color image	Image difference	21,053	4	3
Sobel (Sobel edge detector)	Image processing	1,024-×-1,024-pixel color image	Image difference	143	6	3

of lines, and number of Axilog annotations for design and reuse.

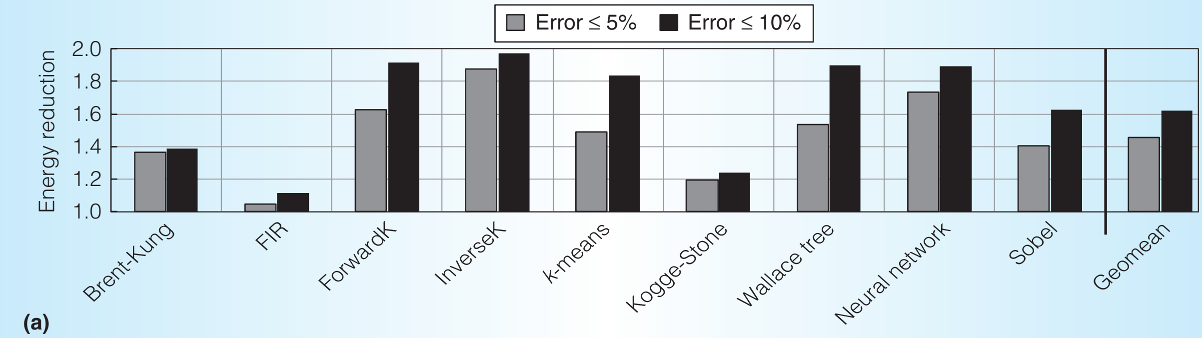
Axilog annotations

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 2 show the number of design and reuse annotations for each benchmark. The number of annotations ranges from two for Brent-Kung with 352 lines to 12 for InverseK with 22,407 lines. The Axilog framework let us use only a handful of annotations to effectively approximate designs that are implemented with thousands of lines of Verilog.

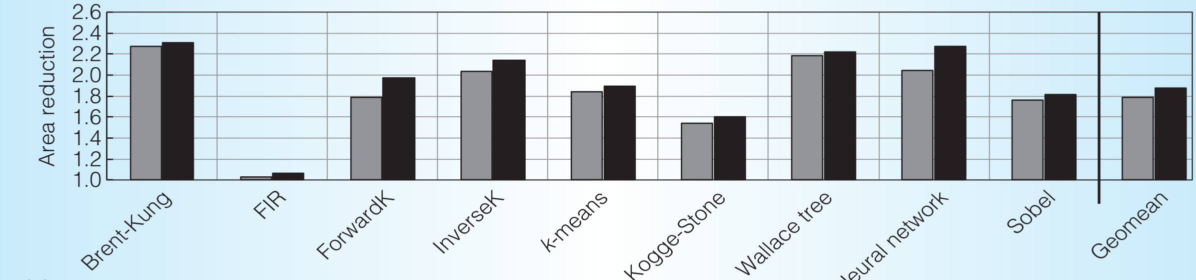
The safe-to-approximate parts are more common in the benchmarks' datapaths rather than their control logic. For example, *k*-means involves a large number of multiplications and additions. We used the `relax` annotations to declare these arithmetic operations approximable; however, we used `restrict` to ensure the precision of all the control signals. For smaller benchmarks, such as Brent-Kung, Kogge-Stone, and Wallace Tree, we annotated only a subset of the least-significant output bits in order to limit the quality loss. We also annotated the benchmarks with reuse annotations. The last column in Table 2 lists the number of reuse annotations. 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 2 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.[4,5] In all cases, we compared the output of the original baseline application to the output of the approximated design.



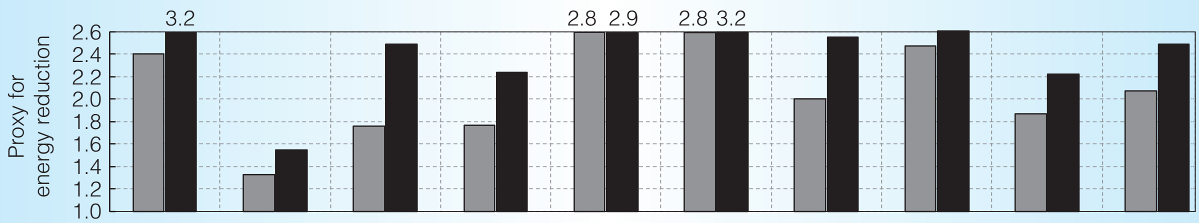
(a)



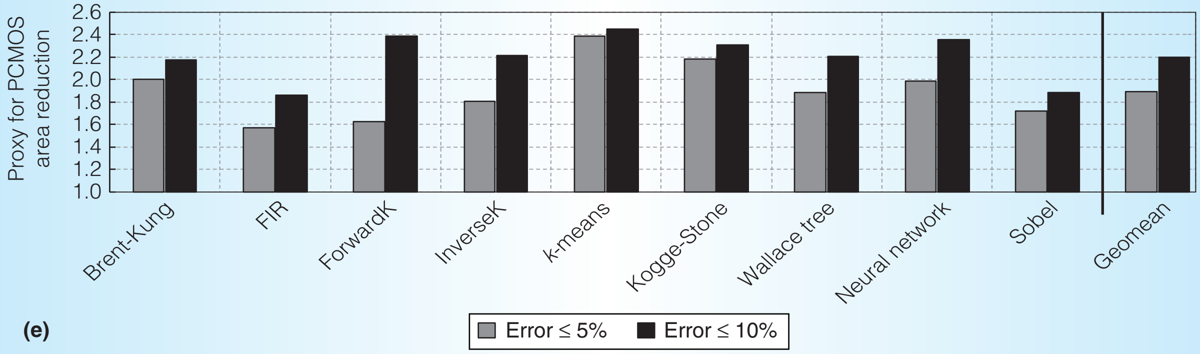
(b)

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

(c)



(d)



(e)

Figure 6. Energy and area reduction for AST flow and energy and PCMOS area reduction for ASG flow. (a) Energy reduction for AST flow = (Precise circuit energy)/(Approximate circuit energy). (b) Area reduction for AST flow = (Precise circuit area)/(Approximate circuit area). (c) Energy reduction for ASG flow when the quality degradation limit is set to 10 percent 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).

Experimental results

Both synthesis techniques used 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 Standard Delay Format back annotations extracted from various operating corners. We used the Taiwan Semiconductor Manufacturing Company 45-nm multi-V_t standard cells libraries and reported the primary results for the slowest process, voltage, and temperature corner (Slow Slow, 0.81 V, 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 percent limit. With the 10 percent limit, the average energy and area gains grow to 1.54× and 1.82×, as shown in Figures 6a and 6b.

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

We also evaluated our AST technique's effectiveness in the presence of temperature variations for a full industrial range of 0 to 125°C. We measured the impact of temperature fluctuations on the energy benefits for the same relaxed designs. Figure 6c compares the energy benefits at the lower and higher temperatures (the quality loss limit is set to 10 percent). In this range of temperature variations, the average energy benefits range from 1.54× (at 0°C) to 1.48× (at 125°C). These results confirm our framework's robustness; it yields significant benefits even when temperature varies.

ASG evaluation. For ASG, we used the NanGate FreePDK 45-nm multispeed standard cells library. The AST and ASG techniques use different libraries because the FreePDK 45-nm library allows Spice simulations required for the ASG flow. As we mentioned earlier, the ASG flow aims to study the trends in future technology nodes when gates might show probabilistic behavior. We developed 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 delineates the anticipated trends. As Figures 6d and 6e show, the ASG flow provides, on average, 2× energy and 1.9× PCMOS area reduction for the 5 percent error limit. With the 10 percent limit, the average energy and area gains grow to 2.5× and 2.2×.

Axilog's automated analysis enables approximate hardware design and reuse without exposing the intricacies of synthesis and optimization. We aim to extend Axilog's annotations to enable designers to specify their desired quality requirements. We also aim to refine the capabilities of the synthesis techniques to better control the approximation versus performance-energy tradeoff such that the designer's quality requirements are met while maximizing the benefits from approximation. Furthermore, the ASG technique now has a complexity on the order of $O(n^2)$, and we aim to devise techniques that would reduce ASG's computational complexity. Finally, we will make Axilog tools and benchmarks open source and available at www.act-lab.org/artifacts/axilog to further facilitate research and development in approximate hardware design. MICRO

Acknowledgments

We thank the anonymous reviewers for their insightful comments. This work was supported in part by Semiconductor Research

Corporation (SRC) contract #2014-EP-2577, Qualcomm Innovation Fellowship, and a gift from Google.

References

1. S. Cheemalavagu et al., "A Probabilistic CMOS Switch and Its Realization by Exploiting Noise," *Proc. IFIP Int'l*, 2005; www.ece.rice.edu/~al4/visen/2005vlsisoc.pdf.
2. S.S. Sapatnekar et al., "An Exact Solution to the Transistor Sizing Problem for CMOS Circuits Using Convex Optimization," *IEEE Trans. Computer-Aided Design*, vol. 12, no. 11, 1993, pp. 1621–1634.
3. 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, pp. 359–381.
4. A. Sampson et al., "EnerJ: Approximate Data Types for Safe and General Low-Power Computation," *Proc. 32nd ACM Sigplan Conf. Programming Language Design and Implementation*, 2011, pp. 164–174.
5. M. Carbin, S. Misailovic, and M. Rinard, "Verifying Quantitative Reliability of Programs that Execute on Unreliable Hardware," *Proc. ACM Sigplan Int'l Conf. Object Oriented Programming Systems Languages & Applications*, 2013, pp. 33–52.

Divya Mahajan is a PhD student in the Computer Science Department at the Georgia Institute of Technology, where she works in the Alternate Computing Technologies Lab. Her research interests include computer architecture, microarchitecture design, and hardware acceleration. Mahajan has an MS in electrical and computer engineering from the University of Texas at Austin. Contact her at divya_mahajan@gatech.edu.

Kartik Ramkrishnan is a PhD candidate in the Department of Computer Science and Engineering at the University of Minnesota. His research focuses on the development of novel techniques for improving memory system performance and approximate circuit design. Ramkrishnan has an MS in electrical engineering from the University of Minnesota, Minneapolis. Contact him at ramkr004@umn.edu.

Rudra Jariwala is a CAD Engineer at Apple. His research interests include approximate hardware design, low-power CAD algorithms, and integrated circuit design. Jariwala has an MS in electrical engineering from the University of Minnesota, Twin Cities, where he completed the work for this article. Contact him at jariw004@umn.edu.

Amir Yazdanbakhsh is a PhD student in the School of Computer Science at the Georgia Institute of Technology, where he works as a research assistant in the Alternative Computing Technologies Lab. His research interests include computer architecture, approximate general-purpose computing, mixed-signal accelerator design, machine learning, and neuromorphic computing. Yazdanbakhsh has an MS in computer engineering from the University of Wisconsin–Madison and an MS in electrical and computer engineering from the University of Tehran. He is a student member of IEEE and the ACM. Contact him at a.yazdanbakhsh@gatech.edu.

Jongse Park is a PhD student in the College of Computing at the Georgia Institute of Technology. His research interests include computer architecture, programming languages, and alternative computing technology. Park has an MS in computer science from the Korea Advanced Institute of Science and Technology. Contact him at jspark@gatech.edu.

Bradley Thwaites is a PhD student in the Department of Electrical and Computer Engineering at the Georgia Institute of Technology. His interests are in computer architecture, memory systems, operating systems, acceleration, and machine learning. Thwaites has an MS in electrical and computer engineering from the Georgia Institute of Technology. Contact him at bthwaites@gatech.edu.

Anandhavel Nagendrakumar is a product engineer at IM Flash Technologies. His research interests include VLSI design, computer architecture, and approximate computing. Nagendrakumar has an MS in electrical and computer engineering from the Georgia Institute of Technology, where he completed the work for this article. Contact him at anandhavel@gatech.edu.

Abbas Rahimi is a PhD candidate in the Department of Computer Science and Engineering at the University of California, San Diego. His research interests include resilient system design, approximate computing, neuro-inspired computing, and on-chip interconnections. Rahimi has a BS in computer engineering from the School of Electrical and Computer Engineering at the University of Tehran. Contact him at abbas@cs.ucsd.edu.

Hadi Esmaeilzadeh is the inaugural holder of the Catherine M. and James E. Allchin Early Career Professorship in the School of Computer Science at the Georgia Institute of Technology. He is also the founder and director of the Alternative Computing Technologies Lab. His research focuses on developing new technologies and cross-stack solutions to improve the performance and energy efficiency of computer systems for emerging applications. Esmaeilzadeh has a PhD in computer science and engineering from the University of Washington. Contact him at hadi@cc.gatech.edu.

Kia Bazargan is an associate professor in the Electrical and Computer Engineering Department at the University of Minnesota. His research interests include FPGA architectures, physical design, and applications and stochastic computing. Bazargan has a PhD in electrical and computer engineering from Northwestern University. Contact him at kia@umn.edu.