Approximate Computing
with
Approximate Circuits:
Methodologies and Applications

ESWEEK 2017 Tutorial

Lukáš Sekanina
Faculty of Information Technology
Brno University of Technology
Brno, Czech Republic
sekanina@fit.vutbr.cz

Jie Han
Department of Electrical and Computer Engineering, University of Alberta
Edmonton, AB, Canada
jhan8@ualberta.ca
Approximate Computing with Approximate Circuits: Methodologies and Applications

Part II: Design automation methods

Lukáš Sekanina
Faculty of Information Technology
Brno University of Technology
Brno, Czech Republic
sekanina@fit.vutbr.cz
Tutorial Outline – Part II.

• **Introduction**
  • Design automation methods for approximate circuits
    – Classification and overview
    – Circuit parameter estimation
    – Error computation
    – Relaxed equivalence checking
    – Evaluation methodology
  • Examples of design automation methods for approximate circuits
    – Minterm complements, SASIMI, AIG rewriting, ABACUS, GRATER
  • Evolutionary algorithms, CGP and circuit optimization
  • Applications of CGP-based approximation methods
    – Open-source library of approximate adders and multipliers
    – Approximate TMR
    – Approximate multipliers in neural networks
    – Symbolic error analysis using BDDs/SAT solving in CGP-based tools
    – Approximate image filters
  • **Conclusions**
Sensitivity analysis

- The goal is to identify subsystems suitable for undergoing the approximation.
- Method: Random/guided modification of the original implementation and statistical evaluation of the impact on the quality of result.

In software
- precision of number representation
- data storage strategies
- code simplification
- relaxed synchronization
- unfinished loops
- skipped function calls

In hardware
- bit width reduction
- intentional disconnecting of components
- timing changes
- power supply voltage changes
- fault injection

Chippa et al., ACSSC 2013 & DAC 2013
Error metrics: Notation

- **Arithmetic error metrics**
  - The worst-case error (error magnitude, error significance)
  - Relative worst-case error
  - The average-case error (average error magnitude, mean error distance)

- **Generic error metrics**
  - Error probability (error rate)
  - Maximum Hamming distance (bit-flip error)
  - Average Hamming distance

- **Application-specific error metrics**
  - Distance error
  - Accumulated worst-case error and accumulated error rate

\[
e_{\text{wst}}(f, \hat{f}) = \max_{\forall x \in B^n} | \text{int}(f(x)) - \text{int}(\hat{f}(x)) |
\]

\[
e_{\text{rel}}(f, \hat{f}) = \max_{\forall x \in B^n} \frac{| \text{int}(f(x)) - \text{int}(\hat{f}(x)) |}{\text{int}(f(x))}
\]

\[
e_{\text{avg}}(f, \hat{f}) = \frac{1}{2^n} \sum_{\forall x \in B^n} | \text{int}(f(x)) - \text{int}(\hat{f}(x)) |
\]

\[
e_{\text{prob}}(f, \hat{f}) = \frac{1}{2^n} \sum_{\forall x \in B^n} \left[ f(x) \neq \hat{f}(x) \right]
\]

\[
e_{\text{bf}}(f, \hat{f}) = \max_{\forall x \in B^n} \left( \sum_{i=0}^{m-1} f_i(x) \oplus \hat{f}_i(x) \right)
\]

\[
e_{\text{hd}}(f, \hat{f}) = \frac{1}{2^n} \sum_{\forall x \in B^n} \sum_{i=0}^{m-1} f_i(x) \oplus \hat{f}_i(x)
\]

\( \hat{f}, f \) – original and approximate solution
\( n, m \) – the number of inputs and outputs
\( \text{int} \) – returns a decimal value from \( m \) bits
Approximation techniques - examples

- precision scaling
- loop perforation
- load value approximation
- memorization
- task dropping/skipping
- memory access skipping
- data sampling
- using different program (circuit) versions
- etc.

- using inexact or faulty hardware
- voltage scaling
- refresh rate reducing
- inexact read/write
- reducing divergence in GPUs
- lossy compression
- use of neural networks.
Functional approximation

- **Principle**: Given $F(x)$, implement a different function $F'(x)$ that minimizes power, area and other circuit parameters, but satisfies the requirements on the quality of output.

**Example:**

- **Traditional view**
  - $F(x)$: Power: 193 uW, Delay: 10 ns, Area: 35 um²
  - $F'(x)$: Power: 100 uW, Delay: 5 ns, Area: 20 um²

- **Approximate computing**
  - Functional equivalence is requested between the specification and implementation at all levels.
  - Relaxed functional equivalence
    - Error: 5%

**A complex multi-objective design/optimization problem!**
• Introduction
• Design automation methods for approximate circuits
  – Classification and overview
  – Circuit parameter estimation
  – Error computation
  – Relaxed equivalence checking
  – Evaluation methodology
• Examples of design automation methods for approximate circuits
  – Minterm complements, SASIMI, AIG rewriting, ABACUS, GRATER
• Evolutionary algorithms, CGP and circuit optimization
• Applications of CGP-based approximation methods
  – Open-source library of approximate adders and multipliers
  – Approximate TMR
  – Approximate multipliers in neural networks
  – Symbolic error analysis using BDDs/SAT solving in CGP-based tools
  – Approximate image filters
• Conclusions
Languages supporting approximate computing

- **EnerJ** [Sampson et al., PLDI 2011]
  - An extension to Java that adds approximate data types. Approximate operations introduced by generating code with cheaper approximate instructions. The system can statically guarantee isolation of the precise program component from the approximate component.

- **Rely** [Carbin et al., OOPSLA 2013]
  - Programmer can mark both variables and operations as approximate. Rely works at the granularity of instructions and symbolically verifies whether the quality-of-result requirements are satisfied for each function. Rely requires programmer to provide preconditions on the reliability and range of the data.

- **Axilog** [Yazdanbakhsh et al., DATE 2015]
  - A set of language annotations that provide the necessary syntax and semantics for approximate hardware design and reuse in Verilog. Axilog’s language semantics and the Relaxability Inference Analysis are independent of the approximate synthesis, i.e. Axilog can be used with virtually any approximate synthesis tool.

- **ExpAX** [Tech. Report GT-CS-14-05, Georgia Tech., 2014]
  - A static safety analysis is performed that uses the high-level (error) expectations to automatically infer a safe-to-approximate set of program operations

- Others: Chisel, ...

- They require a hardware (CPU) supporting approximate computing.
Functional approximation of digital circuits

Original design:
- gate level / RTL / behavioral

Function approximation

Quality metrics, constraints, data

Design methodology

- **Manual** [Kulkarni et al.: J. Low Power Electronic 2011 and others]
- **Design automation methods** (= some heuristics used)
  - SALSA (DAC 2012), SASIMI (DATE 2013), ABACUS (DATE 2014), ASLAN (DATE 2014), AIG-Rewriting (ICCAD 2016) ...

- **Voltage over-scaling not covered in this tutorial.**
Functional circuit approximation: Classification

• Where is the approximation conducted?
  – Component (e.g. adder) / module (e.g. DCT) / application (e.g. video compression)

• What is the level of abstraction?
  – transistor, gate, RTL, behavioral, abstract representation (e.g. SoP, BDD, AIG ...)

• How is the circuit approximated?
  – truncation
  – pruning
  – component replacement (using a library of approximate components)
  – re-synthesis
  – others

• How are candidate approximate circuits evaluated?
  – quality (at different levels of the application)
    • simulation/probabilistic/formal-based methods
  – electrical parameters
    • power, delay, area, ...

• How is the approximation method evaluated?
  – The approximation methods are often heuristics! A proper statistical evaluation is requested (the best vs median value out of several independent runs).
<table>
<thead>
<tr>
<th>First Auth., Conf/Journal, Tool</th>
<th>Method, description</th>
<th>Error comp.</th>
<th>Benchmarks</th>
</tr>
</thead>
<tbody>
<tr>
<td>Shin, DATE10</td>
<td>Elimination in SoP</td>
<td>Exhaustive sim.</td>
<td>&lt;16 inputs: rd73, sym10, rd73, clip, sao2, 5xp1, t481</td>
</tr>
<tr>
<td>Shin, DATE11</td>
<td>Greedy, fault injection</td>
<td>Simulation</td>
<td>c880, c1908, c3540, c5315, c7552</td>
</tr>
<tr>
<td>Venkataramani, DAC12, SALSA</td>
<td>Don’t care simplification</td>
<td>SAT</td>
<td>32-bit+, 8-bit *, 8-bit MAC, SAD, BUT, FIR, IIR, DCT</td>
</tr>
<tr>
<td>Venkataramani, DATE13, SASIMI</td>
<td>Similar signal detection</td>
<td>Simulation</td>
<td>ISCAS85, 32-bit +, 8-bit *, MAC, SAD, …</td>
</tr>
<tr>
<td>Ranjan, DATE14, ASLAN</td>
<td>Sequential/heuristics</td>
<td>SAT</td>
<td>FIR, IIR, MAC, DCT, Sobel, SAD, BUT …</td>
</tr>
<tr>
<td>Nepal, DATE14, ABACUS</td>
<td>Greedy over AST</td>
<td>Simulation</td>
<td>FIR, FFT, perceptron, block matcher, …</td>
</tr>
<tr>
<td>Venkataraman, DATE15</td>
<td>Probabilistic pruning</td>
<td>Simulation</td>
<td>Filters, QRS in ECG</td>
</tr>
<tr>
<td>Li, DAC15</td>
<td>Replacement in HLS</td>
<td>Probabilistic</td>
<td>MediaBench, IIR, FIR, …</td>
</tr>
<tr>
<td>Soeken, ASPDAC16, ABM</td>
<td>Heuristics over BDD</td>
<td>BDD</td>
<td>6 ISCAS-85</td>
</tr>
<tr>
<td>Chandrasekharan, ICCAD16</td>
<td>Greedy, rewriting, AIG</td>
<td>BDD, SAT</td>
<td>LGSynth91, 8/16-bit +, 8 bit *, MAC, parity</td>
</tr>
<tr>
<td>Jain, DATE16</td>
<td>Logic isolation</td>
<td>Probabilistic</td>
<td>32-bit +, 12-bit *, 8-bit DCT, FFT, FIR, …</td>
</tr>
<tr>
<td>Lofti, DATE16, GRATER</td>
<td>Truncation, OpenCL</td>
<td>Simulation</td>
<td>Sobel, DCT, recurs. Gaussian, n-body, convolution</td>
</tr>
<tr>
<td>Sekanina, SSCI-ICES13</td>
<td>CGP</td>
<td>Exhaustive sim.</td>
<td>4 ISCAS85 circuits, adders</td>
</tr>
<tr>
<td>Vašíček, IEEE Tr. on EC, 2015</td>
<td>CGP</td>
<td>Simulation</td>
<td>Multipliers, 9/25-input median</td>
</tr>
<tr>
<td>Vašíček, GPEM, 2016</td>
<td>CGP</td>
<td>BDD</td>
<td>Selected circuits from LGSynth, ITC and ISCAS</td>
</tr>
<tr>
<td>Češka, ICCAD17</td>
<td>CGP</td>
<td>SAT</td>
<td>32-bit *, 128-bit +</td>
</tr>
</tbody>
</table>
**Search-based synthesis of approximate circuits**

- The optimization engine applies various transformation rules on a given circuit and gradually modifies the circuit with the aim to obtain its approximate version which satisfies a given condition (e.g. maximal error).
- See the CGP-based approximation in this Tutorial.

Circuit parameter estimation

- Basic circuit parameters: delay, area, power, ...
- Professional CAD tools
  - Good quality
  - Slow if thousands of candidate approximate circuits have to be evaluated
- Simple methods
  - Fast, but could be inaccurate
    - Area = sum of the areas of the gates involved
    - Delay = delay along the longest path; the capacitive output load not ignored
    - Power = static (leakage) + dynamic (switching activity simulation)
    - Calibration is needed!
  - They are used during the approximation process.
  - The resulting approximate circuits have to be validated using professional tools.

How to determine the error?
Are $F_1$ and $F_2$ functionally equivalent?

$F_1 = (x_1 \land \neg x_2) \lor \neg x_3$

$F_2 = \neg (\neg (x_1 \land \neg x_2) \land x_3)$

<table>
<thead>
<tr>
<th>x3</th>
<th>x2</th>
<th>x1</th>
<th>F1</th>
<th>F2</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

• Functional equivalence checking methods have been developed for decades.
  – They exploit the model canonicity, SAT solving, algebraic approaches, ...

• Relaxed functional equivalence checking is a new topic!
  – How to prove the equivalence up to some bound?

• Scalability problem of (relaxed) equivalence checking!
How to determine the error?

Error “estimation”
• (Functional) circuit simulation
• Probabilistic models, e.g. Li at al., DAC 2015

Exact error calculation
• Exhaustive simulation – small problem instances only
• Analysis of Binary decision diagrams
  • Average error, worst case, error rate ...
    • M. Soeken et al., ASP-DAC 2016
  • Average Hamming distance:
  • Not scalable for some circuits such as multipliers
• Transforming to SAT problem
  • Worst case error
    • S. Venkataramani et al. : DAC 2012 (SALSA), A. Chandrasekharan et al. DAC 2016, M. Ceska et al., ICCAD 2017
  • Not suitable if counting the number of solutions is requested.
Error computation: Probabilistic methods

• For a given approximate circuit and the input data distribution, a probabilistic model is constructed and the error statistics are derived.

• Examples
  – The error statistics can be expressed as functions of the number of input bits, carry-chain length, number of overlapping prediction bits and number of sub-adders in the case of approximate adders [Mazahir et al. IEEE TC 66(3), 2017]
  – In the context of approximate HLS, an error of approximate adders and multipliers was characterized by its mean and variance. The mean is systematic and can be compensated. The overall computation precision is then determined by the variance which, after the constant compensation corresponds, to the Mean Squared Error [Li et al. DAC 2015].

• Advantages
  – Fast error computation

• Disadvantages
  – An error model has to be derived for all components which is time consuming and impractical for circuits different to adders and multipliers.
  – It is hard to provide formal guarantees in terms of the error bound.
Assumptions
- Error can be modeled as a random variable described by its mean and variance: $Mean(\varepsilon)$, $Var(\varepsilon)$
- Mean value of the error can be canceled out by a constant bias
- First order model is sufficient

Basic operations error model
- $y = a + b \rightarrow y + \varepsilon_y = (a + \varepsilon_a) + (b + \varepsilon_b) + \varepsilon_+$
- $y = a \times b \rightarrow y + \varepsilon_y = ab + a\varepsilon_b + b\varepsilon_a + \varepsilon_+ + \varepsilon_a\varepsilon_b$

Pre-processing
- Compute the error sensitivity ($ES_{O_i,Y}$) of output $O_i$ to an error introduced by node $Y$.
- Searching all paths from $O_i$ to $Y$, using modified DFS traversal.

Error evaluation
- $Var(\varepsilon_{O_i}) = \sum_{y \in \text{Nodes}} ES_{O_i,Y} \times Var(\varepsilon_y)$

In this case
- $Var(\varepsilon_{O_0}) = 1 \times Var(\varepsilon_D) + 1 \times Var(\varepsilon_B) + 1 \times Var(\varepsilon_C)$
  - (the impact of component A is eliminated)
Binary Decision Diagrams

\[ f = ac + bc \]

<table>
<thead>
<tr>
<th>a</th>
<th>b</th>
<th>c</th>
<th>f</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>

Truth table

Decision tree

Reduced Ordered BDD (ROBDD) (canonical form)

Operations over (RO)BDDs implemented by many libraries, e.g. Buddy.
Pitfalls of Binary Decision Diagrams

- Variable ordering is important, may result in a more complex (or simple) BDD.

\[ x_1 x_2 + x_3 x_4 \]

\[
\begin{align*}
&x_1 < x_2 < x_3 < x_4 \\
&(\text{optimal})
\end{align*}
\]

\[
\begin{align*}
&x_1 < x_3 < x_2 < x_4
\end{align*}
\]
The decision procedure is trivial and reduces to pointer comparison.

Equivalence checking using ROBDDs

Are circuits C1 and C2 functionally equivalent?

ROBDD construction:
Apply \((op, a, b)\) – creates ROBDD representing logic function \(op\) over two ROBDDs \(a\) and \(b\)
Other operations on ROBDDs

• Many logic operations can be performed efficiently on BDDs
  – usually in linear time
  – tautology and complement are constant time

<table>
<thead>
<tr>
<th>Procedure</th>
<th>Result</th>
<th>Time Complexity</th>
</tr>
</thead>
<tbody>
<tr>
<td>Reduce</td>
<td>$G$ reduced to canonical form</td>
<td>$O(</td>
</tr>
<tr>
<td>Apply</td>
<td>$f_1 \langle \text{op} \rangle f_2$</td>
<td>$O(</td>
</tr>
<tr>
<td>Restrict</td>
<td>$f</td>
<td>_{x_i=b}$</td>
</tr>
<tr>
<td>Compose</td>
<td>$f_1</td>
<td>_{x_i=f_2}$</td>
</tr>
<tr>
<td>Satisfy-one</td>
<td>some element of $S_f$</td>
<td>$O(n)$</td>
</tr>
<tr>
<td>Satisfy-all</td>
<td>$S_f$</td>
<td>$O(n \cdot</td>
</tr>
<tr>
<td>Satisfy-count</td>
<td>$</td>
<td>S_f</td>
</tr>
</tbody>
</table>

Bryant R.: Graph-Based Algorithms for Boolean Function Manipulation. IEEE Trans. on Comp. 1986
Average Hamming distance using BDDs

- Create ROBDD for \( C_A \), \( C_B \) and the XOR gates.
- Average Hamming distance:

\[
e_{HD} = \frac{1}{2^{\text{inputs}}} \sum_{i=1}^{\text{outputs}} \text{SatCount}(z_i)
\]

SatCount \((f)\) – gives the number of input assignments for which \( f \) is ‘1’.

\[
\begin{align*}
\text{SatCount}(z_1) &= 2 \\
\text{SatCount}(z_2) &= 0
\end{align*}
\]

<table>
<thead>
<tr>
<th>( x_1 )</th>
<th>( x_2 )</th>
<th>( x_3 )</th>
<th>( x_4 )</th>
<th># combinations</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
</tbody>
</table>
Average-case and worst-case error analysis

• Let $f : \mathcal{B}^n \rightarrow \mathcal{B}^m$ be a Boolean function that describes correct functionality and $\hat{f} : \mathcal{B}^n \rightarrow \mathcal{B}^m$ an approximation of it. The average-case error is defined as the sum of absolute differences in magnitude between the original and approximate circuit, averaged over all inputs:

$$e_{avg}(f, \hat{f}) = \frac{1}{2^n} \sum_{\forall x \in \mathcal{B}^n} | \text{int}(f(x)) - \text{int}(\hat{f}(x)) |$$

where $\text{int}(x)$ represents a function returning a decimal value of the m-bit binary vector $x$.

• The worst-case error is defined:

$$e_{wst}(f, \hat{f}) = \max_{\forall x \in \mathcal{B}^n} | \text{int}(f(x)) - \text{int}(\hat{f}(x)) |$$
Algorithm 2: average-case error analysis

Input: BDD representation of the virtual circuit (d)
Output: The average arithmetic error ($\varepsilon_{avg}$)

1. $\varepsilon_{avg} \leftarrow 0$;
2. for $i \in \{m - 1, m - 2, \ldots, 0\}$ do
3. \hspace{1em} $\varepsilon_{avg} \leftarrow \varepsilon_{avg} + 2^{i-2n} \cdot \text{satcount}(d_i)$;
4. return $\varepsilon_{avg}$;

Algorithm 1: BDD worst-case error analysis

Input: BDD representation of the virtual circuit (d)
Output: The maximum arithmetic error ($\varepsilon_{max}$)

1. $\varepsilon_{max} \leftarrow 0$, $\mu \leftarrow true$;
2. for $i \in \{m - 1, m - 2, \ldots, 0\}$ do
3. \hspace{1em} if satisfiable($\mu \land d_i$) then
4. \hspace{2em} $\mu \leftarrow \mu \land d_i$; $\varepsilon_{max} \leftarrow \varepsilon_{max} + 2^i$;
5. return $\varepsilon_{max}$;

Example for $n = 4$: Because the result of SUB is -32 … +31, the max absolute value is 32 and 6 bits are needed for $m$. 

$m = n + 2$

Error analysis using BDD (adders)

VASICEK Z., MRAZEK V., SEKANINA L.: Towards Low Power Approximate DCT Architecture for HEVC Standard. DATE 2017

Soeken et al. BDD Minimization for Approximate Computing ASPDAC 2016
BDD vs exhaustive simulation: Adders

The average time needed to perform the worst-case and the average-case error analysis for $w$-bit adders:

<table>
<thead>
<tr>
<th>bit-width</th>
<th>inputs</th>
<th>parallel simulation $\varepsilon_{\text{max}} + \varepsilon_{\text{avg}}$</th>
<th>BDD-based method $\varepsilon_{\text{max}}$</th>
<th>$\varepsilon_{\text{avg}}$</th>
<th>speedup $\varepsilon_{\text{max}}$</th>
<th>$\varepsilon_{\text{avg}}$</th>
</tr>
</thead>
<tbody>
<tr>
<td>4-bit</td>
<td>8</td>
<td>4.5 us</td>
<td>10.3 us</td>
<td>14.0 us</td>
<td>0.43 $\times$</td>
<td>0.32 $\times$</td>
</tr>
<tr>
<td>8-bit</td>
<td>16</td>
<td>1.9 ms</td>
<td>3.5 ms</td>
<td>4.6 ms</td>
<td>0.54 $\times$</td>
<td>0.42 $\times$</td>
</tr>
<tr>
<td>12-bit</td>
<td>24</td>
<td>682.4 ms</td>
<td>127.9 ms</td>
<td>312.7 ms</td>
<td>5.33 $\times$</td>
<td>2.18 $\times$</td>
</tr>
<tr>
<td>16-bit</td>
<td>32</td>
<td>140.9 s</td>
<td>1.38 s</td>
<td>2.93 s</td>
<td>102.3 $\times$</td>
<td>48.09 $\times$</td>
</tr>
</tbody>
</table>

Notes
1) 100 randomly generated approximate adders were evaluated for each bit-width.
2) The time required to construct a BDD for the virtual circuit is included.

Practical experience: BDD-based analysis of multipliers is >10 times slower than simulation.

VASICEK Z., MRAZEK V., SEKANINA L.: Towards Low Power Approximate DCT Architecture for HEVC Standard. DATE 2017
If $C_1$ and $C_2$ are not functionally equivalent then there is at least one assignment to the inputs for which the output of $G$ is 1.
Example: \( y = \text{not } (x) \)

CNF formula \( g(x, y) = 1 \) if the predicate \( y = \text{OP}(x) \) holds true

<table>
<thead>
<tr>
<th>( x )</th>
<th>( y )</th>
<th>( g )</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
</tbody>
</table>

\[
g = (\lnot x \lor \lnot y)(x \lor y)
\]

\[
(x_2 + x_8)(\overline{x_2} + \overline{x_8})
\]

\[
(x_8 + x_9)(x_3 + x_9)(\overline{x_8} + \overline{x_3} + \overline{x_9})
\]

\[
(x_1 + x_{10})(x_9 + x_{10})(\overline{x_1} + \overline{x_9} + \overline{x_{10}})
\]

\[
(\overline{x_7} + x_{10} + x_{11})(x_7 + \overline{x_{10}} + x_{11})(\overline{x_7} + \overline{x_{10}} + \overline{x_{11}})(x_7 + x_{10} + \overline{x_{11}})
\]

\[
(\overline{x_6} + x_9 + x_{12})(x_6 + \overline{x_9} + x_{12})(\overline{x_6} + x_9 + \overline{x_{12}})(x_6 + x_9 + \overline{x_{12}})
\]

\[
(x_{11} + x_{12} + \overline{x_{13}})(x_{13} + \overline{x_{11}})(x_{13} + \overline{x_{12}})
\]

\[
(\overline{x_{13}})
\]
SAT solver in action

SAT solver: MiniSAT
variables: 13, clauses: 30, time elapsed: 0.03ms
result: SATISFIABLE / NONEQUIVALENT
model / counter example: 0011111101011
Worst-case error analysis using SAT solver

- The common approach is to use SAT-solver and binary search to find WCE (= \( X \)).
- Example: WCE for approximate n-bit adders

\[
\begin{align*}
(x_6 + x_9 + x_{12})(x_6 + \overline{x_9} + x_{12}) \\
(x_6 + \overline{x_9} + x_{12})(x_6 + x_9 + \overline{x_{12}}) \\
(x_{11} + x_{12} + \overline{x_{13}})(x_{13} + \overline{x_{11}}) \\
(x_{13} + \overline{x_{12}})(x_{13})
\end{align*}
\]

Algorithm 2: SAT worst-case error analysis

\begin{verbatim}
Input: SAT representation of the accurate circuit \( f \) and the approximate circuits \( f' \) with \( m \) outputs

Output: The maximum arithmetic error \( (\varepsilon_{\text{max}}) \)

1. \( lbound \leftarrow 0; \)
2. \( ubound \leftarrow 2^m - 1; \)
3. \textbf{while} \( lbound < ubound \) \textbf{do}
4. \hspace{1em} \( X \leftarrow \left\lfloor \frac{ubound + lbound}{2} \right\rfloor; \)
5. \hspace{1em} \textbf{if} satisfiable(\texttt{ApproxMiter}([\( f - f' \), \( X \)])) \textbf{then}
6. \hspace{2em} \( lbound \leftarrow X; \)
7. \hspace{1em} \textbf{else}
8. \hspace{2em} \( ubound \leftarrow X - 1; \)
9. \hspace{1em} \( \varepsilon_{\text{max}} \leftarrow lbound; \)
10. \textbf{return} \( \varepsilon_{\text{max}}; \)
\end{verbatim}
On a fair comparison of automated approx. methods

• **Common practice**: The original circuit and approximate circuits created using a given method are compared -> **not sufficient**!

• A comparisons with other approximation methods is needed!

• Important assumptions for a fair comparison:
  – the original circuits are the same
  – the error is calculated using the same method (simulation vs. exact)
  – electrical parameters are calculated using the same tool and for the same technology library
  – the time/resources for the approximation methods under investigation are the same
  – the same statistically relevant values are reported (best, median, mean etc.)
Benchmarks for approximate computing

• Adders and multipliers
  – IpACLib Library
    • https://sourceforge.net/projects/lpaclib/
  – GeaR Library:
    • https://sourceforge.net/projects/approxadderlib/
  – Evoapprox8b Library
    • http://www.fit.vutbr.cz/research/groups/ehw/approxlib/

• Other
  – AxBench (GPU, CPU, Verilog)
    • http://axbench.org/
  – ApproxBench
    • http://approxbench.org/
  – AcHEe
Tutorial Outline – Part II.

• Introduction
• Design automation methods for approximate circuits
  – Classification and overview
  – Circuit parameter estimation
  – Error computation
  – Relaxed equivalence checking
  – Evaluation methodology
• Examples of design automation methods for approximate circuits
  – Minterm complements, SASIMI, AIG rewriting, ABACUS, GRATER
• Evolutionary algorithms, CGP and circuit optimization
• Applications of CGP-based approximation methods
  – Open-source library of approximate adders and multipliers
  – Approximate TMR
  – Approximate multipliers in neural networks
  – Symbolic error analysis using BDDs/SAT solving in CGP-based tools
  – Approximate image filters
• Conclusions
Finding minterm complements to reduce # literals

- The objective is to obtain designs that have a minimum number of literals for a given error rate threshold.
- Method: Identify minterm complements that produce an approximate circuit version that has the smallest number of literals for a given error rate threshold.
- Exhaustive search for simple functions, a heuristics approach for more complex functions.

Original solution:
\[ \bar{x}_1x_2x_4 + x_2x_3x_4 + x_1\bar{x}_2\bar{x}_3x_4 \]

Approximation 1: \[ x_2x_4 + x_1\bar{x}_3x_4 \]

Approximation 2: \[ \bar{x}_1x_2x_4 + x_2x_3x_4 \]
SASIMI: Substitute and Simplify

**Key Idea:** Identify signal pairs (TS and SS) that are similar in functionality i.e. produce the same value for most of the inputs among signal pairs.

- **Substitute** one in place of the other
  - Circuit becomes approximate
- **Simplify** the circuit: Logic Deletion & Downsizing

The signal probability calculation engine in Synopsys Power Compiler was used to obtain difference probabilities.

S. Venkataramani, K. Roy, and A. Raghunathan: Substitute-and simplify: a unified design paradigm for approximate and quality configurable circuits, DATE’13, pp. 1367–1372
SASIMI: Substitute and Simplify

Approximation-aware Rewriting of AIGs

- **Principle**: allow AIG rewriting to change the functionality of the circuit without violating a predefined error bound.
- Rewriting (at the level of cuts on selected paths) takes a greedy approach.
- Worst-case error, bit-flip error and error rate determined exactly (formally).
- Evaluated: 8/16-bit adders, LGSYnth91, 8-bit multipliers, 32-bit parity, ...

Heuristics:
- replace the cut by constant 0

2-bit adder
ABACUS: Approximations at Behavioral RT-level

- Original file: Verilog
- Abstract Syntax Tree (AST) transformations (mutations)
  - Data type simplification
  - Operation transformations (e.g. + -> or)
  - Arithmetic expression transformation
  - Variable to Constant transformations
  - Loop transformations
- Search algorithm: Greedy / NSGA-II
- Fitness is obtained by circuit simulation and combines the error & power

ABACUS: Results

Benchmark problems:

<table>
<thead>
<tr>
<th>Design</th>
<th>Class of Application</th>
<th>#Lines</th>
<th>Area (um²)</th>
<th>Power (mW)</th>
<th>Quality Measure</th>
<th>Quality</th>
</tr>
</thead>
<tbody>
<tr>
<td>perceptron</td>
<td>Machine Learning</td>
<td>188</td>
<td>37775.16</td>
<td>2.74</td>
<td>classification error</td>
<td>82.9%</td>
</tr>
<tr>
<td>FIR filter</td>
<td>Signal Processing</td>
<td>265</td>
<td>40390.20</td>
<td>6.89</td>
<td>MSE</td>
<td>99.45%</td>
</tr>
<tr>
<td>FFT</td>
<td>Signal Processing</td>
<td>255</td>
<td>18480.96</td>
<td>2.07</td>
<td>MSE</td>
<td>100%</td>
</tr>
<tr>
<td>block matching</td>
<td>Computer Vision</td>
<td>1277</td>
<td>80272.44</td>
<td>30.42</td>
<td>PSNR</td>
<td>30.66 dB</td>
</tr>
</tbody>
</table>

Results of evolutionary approximation:

(a) Perceptron  
(b) FIR Filter  
(c) FFT  
(d) Block Matching
GRATER: GA-based optimization of data types

- Sensitivity analysis performed to find safe-to-approximate variables (AV) in OpenCL kernel.
- Encoding: $n$ integers specifying precision (i.e. data type) of $n$ variables from AV.
- Objective: to find an approximate kernel that minimizes the resource utilization on FPGA while meeting the target quality.
Tutorial Outline – Part II.

• Introduction
• Design automation methods for approximate circuits
  – Classification and overview
  – Circuit parameter estimation
  – Error computation
  – Relaxed equivalence checking
  – Evaluation methodology
• Examples of design automation methods for approximate circuits
  – Minterm complements, SASIMI, AIG rewriting, ABACUS, GRATER
• Evolutionary algorithms, CGP and circuit optimization
• Applications of CGP-based approximation methods
  – Open-source library of approximate adders and multipliers
  – Approximate TMR
  – Approximate multipliers in neural networks
  – Symbolic error analysis using BDDs/SAT solving in CGP-based tools
  – Approximate image filters
• Conclusions
Evolutionary algorithms: GA, ES, EP, GP, LGP, CGP, ...

• The term **Evolutionary Algorithm** covers various search algorithms that have the following common features:
  – There is a population of candidate solutions (inherent parallelism).
  – New candidate solutions are created using operators inspired in genetics (crossover, mutation).
  – Nothing is expected about the objective (fitness) function.

• Main branches:
  – Genetic Algorithms – GA (Holland ~1973)
  – Evolution Strategies – ES (Rechenberg and Schwefel ~1964)
  – Evolutionary Programming - EP (Fogel ~1962)
  – and others such as differential evolution, grammatical evolution, Cartesian genetic programming etc.
Evolutionary algorithms: GA, GP, LGP, CGP, GE ...

- Create the initial population
- Evaluation
- Selection of parents
- Recombination (crossover)
- Mutation
- Replacement

GA chromosome: binary string
Cartesian Genetic Programming (CGP) [Miller, 1999]

- $n_i$ primary inputs
- $n_o$ primary outputs
- $n_c$ columns
- $n_r$ rows

- $n_a$ inputs of each node
- $\Gamma$ function set
- L-back parameter

Nodes in the same column are not allowed to be connected to each other. No feedback!
CGP: Representation for logic networks

Genotype (netlist):
$n_a+1$ integers per node; $n_o$ integers for outputs;
Constant size: $n_c n_r (n_a + 1) + n_o$ integers

Phenotype (directed acyclic graph ⇒ circuit):
Variable size; unused nodes are ignored.

CGP parameters
- $n_r = 3$ (#rows)
- $n_c = 3$ (#columns)
- $n_i = 3$ (#inputs)
- $n_o = 2$ (#outputs)
- $n_a = 2$ (max. arity)
- $L = 3$ (level-back parameter)
- $\Gamma = \{\text{NAND}^{(0)}, \text{NOR}^{(1)}, \text{XOR}^{(2)}, \text{AND}^{(3)}, \text{OR}^{(4)}, \text{NOT}^{(5)}\}$
CGP: Fitness function for circuit design

Typical fitness function (circuit functionality):

$$f = \sum_{i=1}^{K} HD(y_i, w_i)$$

- Hamming distance (between circuit and desired response)
- The number of test vectors $K = 2^{\text{inputs}}$ for combinational circuits. Not scalable!!!

Additional objectives:
- area (the number of gates)
- delay
- power consumption etc.

Target table:

<table>
<thead>
<tr>
<th>$a$</th>
<th>$b$</th>
<th>$c$</th>
<th>$d$</th>
<th>$s$</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>

$\Rightarrow$ fitness = 16

$\Rightarrow$ fitness = 10

Specification (1-bit adder):

1, 2, 1; 1, 2, 2; 0, 1, 2; 4, 2, 5; 5, 4, 3; 3, 0, 2; 7, 1, 2; 1, 6, 5; 1, 1, 3; 8, 9
**CGP: Mutation-based search**

- Mutation: Randomly select *h* integers and replace them by randomly generated (but legal) values:

```
1 3
1 3
2 5
2 5
2 5
2 5
3 7
3 7
3 7
3 7
2 8
2 8
2 8
2 8
2 8
2 8
2 8
2 8
2 8
2 8
2 8
2 8
2 8
2 8
2 8
2 8
2 8
2 8
2 8
2 8
1 9
1 9
1 9
1 9
1 9
1 9
1 9
1 9
1 9
1 9
1 9
1 9
1 9
1 9
1 9
1 9
```

```
1 2,1; 1,2,2; 0,1,2; 4,2,5; 5,4,3; 3 0,2; 7,1,2; 1,6,5; 1,1,3; 8,9
1 2,1; 1,2,2; 0,1,2; 4,2,5; 5,4,3; 3 0,2; 7,1,2; 1,6,5; 1,1,3; 8,9
1 2,1; 1,2,2; 0,1,2; 4,2,5; 5,4,3; 3 0,2; 7,1,2; 1,6,5; 1,1,3; 8,9
```

### Mutation Example

For a full adder:

<table>
<thead>
<tr>
<th>a</th>
<th>b</th>
<th>c</th>
<th>d</th>
<th>s</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>

*=> fitness = 10*

<table>
<thead>
<tr>
<th>a</th>
<th>b</th>
<th>c</th>
<th>d</th>
<th>s</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>

*=> fitness = 16*  
(for a full adder)
Algorithm 1: CGP

**Input:** CGP parameters, fitness function

**Output:** The highest scored individual \( p \) and its fitness

1. \( P \leftarrow \text{randomly generate population}; // \text{or use conventional designs} \)
2. EvaluatePopulation\((P)\); \( p \leftarrow \text{highest-scored-individual}(P) \);
3. \textbf{while} \langle \text{terminating condition not satisfied} \rangle \textbf{do}
4. \hspace{1em} \( \alpha \leftarrow \text{highest-scored-individual}(P) \);
5. \hspace{1em} \textbf{if} fitness\((\alpha) \geq \text{fitness}(p)\) \textbf{then}
6. \hspace{2em} \( p \leftarrow \alpha \);
7. \hspace{1em} \( P \leftarrow \text{create } \lambda \text{ offspring of } p \text{ using mutation} \);
8. \hspace{1em} EvaluatePopulation\((P)\);
9. \textbf{return} \( p, \text{fitness}(p) \);
CGP for optimization of complex circuits

- SAT solver is used to decide whether candidate circuit $C_i$ and reference circuit $C_1$ are functionally equivalent.
  - If so, then $\text{fitness}(C_i) = \text{the number of gates in } C_i$;
  - Otherwise: discard $C_i$.

Original circuit $C$ (BLIF) \rightarrow Conventional synthesis (ABC, SIS…) \rightarrow Optimized circuit $C_1$ (≡ a seed for the initial population; reference circuit) \rightarrow CGP \rightarrow Even more optimized $C_1$

Vasicek, Sekanina: Genetic Programming and Evolvable Machines 12(3), 2011
CGP with SAT solver (no approximation)

SAT solver is called only if the circuit simulation performed for a small subset of vectors has indicated no error in the candidate circuit.

100 combinational circuits (≥15 inputs) - IWLS2005, MCNC, QUIP benchmarks
Heavily optimized by ABC
1: alcom ($N_G = 106$ gates; $N_{PI} = 15$ inputs; $N_{PO} = 38$ outputs)
100: ac97ctrl ($N_G = 16,158$; $N_{PI} = 2,176$; $N_{PO} = 2,136$)
CGP with SAT solver (no approximation)

CGP + SAT solver + circuit simulation
Y-axis: Gate reduction w.r.t. ABC after 15 minutes, 34% on average
▲ Gate reduction w.r.t. ABC after 24 hours

Properly optimize before doing approximations!

Vasicek Z.: EuroGP 2015
Tutorial Outline – Part II.

• Introduction
• Design automation methods for approximate circuits
  – Classification and overview
  – Circuit parameter estimation
  – Error computation
  – Relaxed equivalence checking
  – Evaluation methodology
• Examples of design automation methods for approximate circuits
  – Minterm complements, SASIMI, AIG rewriting, ABACUS, GRATER
• Evolutionary algorithms, CGP and circuit optimization
• Applications of CGP-based approximation methods
  – Open-source library of approximate adders and multipliers
  – Approximate TMR
  – Approximate multipliers in neural networks
  – Symbolic error analysis using BDDs/SAT solving in CGP-based tools
  – Approximate image filters
• Conclusions
Why EA in approximate computing?

• In approximate computing, *partially working* solutions are sought.
• In EA, *partially working* solutions are improved.
• EAs are excellent in *multi-objective* design and optimization.
• *Constraints* can easily be handled.
• EA can be seeded with the original code (circuit).
• EA is easy to implement and parallelize.
CGP for circuit (functional) approximation

• **Error-oriented (single-objective) method**
  • CGP gradually degrades a fully functional circuit until a circuit with a *required error* is obtained. Then, the area (and so power consumption) is minimized for this error.

• **Resources-oriented (single-objective) method**
  • CGP is used to minimize the error, but only limited resources (components) are provided, insufficient for constructing a fully functional circuit.

• **Multi-objective optimization**
  • All target parameters are optimized together.
Library of approximate 8 bit adders and multipliers

- **Parallel multi-objective CGP:**
  - CGP + Non-dominated Sorting Genetic Algorithm II (NSGA-II) [Hrbáček, GECCO 2015]
  - Parallel implementation: vectorized, multi-threaded, multiple islands (computer cluster employed)
- **Constraints:** worst case error, worst case relative error
- **Initial population:** a set of fully working conventional circuits
- **Fitness:** mean relative error, power consumption, delay

\[
f_{mre} := \sum_{\forall i} \frac{|O_{\text{orig}}^{(i)} - O_{\text{approx}}^{(i)}|}{\max(1,O_{\text{orig}}^{(i)})} \frac{1}{2^{N_i}}
\]

\(O^{(i)}\) is the \(i\)-th circuit output

\(i = 1 \ldots 2^{N_i}\)

Target circuits - Inputs: \(N_i = 16\); Outputs: \(N_o = 9\) (adders), 16 (multipliers)
CGP parameters

- Population size: 500 candidate circuits
- Generations: 100k
- Mutation: 5%
- Parallel CGP: 10 islands exchanging circuits every 1000 generations (120 cores)
- CGP array: 1 x 200 nodes (adders), 1 x 1000 nodes (mult.)
- CGP function set (180/45 nm technology library):
  - BUF, INV, AND2, OR2, XOR2, NAND2, NOR2, XNOR2, NAND3, NOR3, MUX2, AOI21, OAI21, Full Adder, Half Adder
  - 3-input/2-output nodes used
### CGP: Initial population

<table>
<thead>
<tr>
<th>Architecture</th>
<th>Power</th>
<th>Area</th>
<th>Delay</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Ripple-Carry Adder</strong></td>
<td>100.00%</td>
<td>100.00%</td>
<td>100.00%</td>
</tr>
<tr>
<td>Carry-Select Adder</td>
<td>201.18%</td>
<td>174.78%</td>
<td>61.15%</td>
</tr>
<tr>
<td>Carry-Lookahead Adder</td>
<td>414.74%</td>
<td>334.78%</td>
<td>61.99%</td>
</tr>
<tr>
<td>HVTA (Brent-Kung)</td>
<td>286.00%</td>
<td>201.74%</td>
<td>68.52%</td>
</tr>
<tr>
<td>HVTA (Han-Carlson)</td>
<td>286.00%</td>
<td>201.74%</td>
<td>68.52%</td>
</tr>
<tr>
<td>HVTA (Kogge-Stone)</td>
<td>371.48%</td>
<td>257.39%</td>
<td>59.77%</td>
</tr>
<tr>
<td>HVTA (Sklansky)</td>
<td>305.07%</td>
<td>215.65%</td>
<td>60.45%</td>
</tr>
<tr>
<td>TA (Brent-Kung)</td>
<td>282.99%</td>
<td>201.74%</td>
<td>67.25%</td>
</tr>
<tr>
<td>TA (Han-Carlson)</td>
<td>295.74%</td>
<td>212.17%</td>
<td>61.87%</td>
</tr>
<tr>
<td>TA (Knowles)</td>
<td>362.25%</td>
<td>257.39%</td>
<td>59.94%</td>
</tr>
<tr>
<td>TA (Kogge-Stone)</td>
<td>342.20%</td>
<td>243.48%</td>
<td>57.68%</td>
</tr>
<tr>
<td>TA (Ladner-Fischer)</td>
<td>282.99%</td>
<td>201.74%</td>
<td>67.25%</td>
</tr>
<tr>
<td>TA (Sklansky)</td>
<td>298.34%</td>
<td>212.17%</td>
<td>57.84%</td>
</tr>
</tbody>
</table>

**13 conventional 8-bit adders**

- RA = Tree Adder
- HVTA = Higher Valency Tree Adder

<table>
<thead>
<tr>
<th>Architecture</th>
<th>Power</th>
<th>Area</th>
<th>Delay</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Ripple-Carry Array</strong></td>
<td>100.00%</td>
<td>100.00%</td>
<td>100.00%</td>
</tr>
<tr>
<td>Carry-Save Array using RCA</td>
<td>102.30%</td>
<td>100.00%</td>
<td>71.16%</td>
</tr>
<tr>
<td>Carry-Save Array using CSA</td>
<td>108.42%</td>
<td>106.16%</td>
<td>62.03%</td>
</tr>
<tr>
<td>Wallace Tree using RCA</td>
<td>104.29%</td>
<td>107.39%</td>
<td>68.91%</td>
</tr>
<tr>
<td>Wallace Tree using CLA</td>
<td>116.10%</td>
<td>148.48%</td>
<td>51.26%</td>
</tr>
<tr>
<td>Wallace Tree using CSA</td>
<td>120.12%</td>
<td>122.35%</td>
<td>53.28%</td>
</tr>
</tbody>
</table>

**6 conventional 8-bit multipliers**

- RCA = Ripple-Carry Adder
- CSA = Carry-Save Adder
- CLA = Carry-Lookahead Adder
Library of 8-bit approx. adders and multipliers

- Comprehensive library of approximate arithmetic circuits
  - 430 non-dominated adders (evolved from 13 accurate adders)
  - 471 non-dominated multipliers (evolved from 6 accurate multipliers)

Approximate adders
(100% is Ripple-Carry Adder)

Approximate multipliers
(100% is Ripple-Carry Array Multiplier)

Library of 8-bit approx. adders and multipliers

Approximate adders (430), exact adders (43)

<table>
<thead>
<tr>
<th>Circuit</th>
<th>Est. area</th>
<th>Est. delay</th>
<th>Est. power</th>
<th>Nodes</th>
<th>HD</th>
<th>MAE</th>
<th>MSE</th>
<th>MRE</th>
<th>WCE</th>
<th>WCRE</th>
<th>EP</th>
<th>OPS</th>
</tr>
</thead>
<tbody>
<tr>
<td>add8_000</td>
<td>820 µm²</td>
<td>1.314 ns</td>
<td>194.31 µW</td>
<td>10</td>
<td>138496</td>
<td>1.71875</td>
<td>6.00000</td>
<td>0.88 %</td>
<td>7</td>
<td>100 %</td>
<td>71.875 %</td>
<td>Verilog, Matlab</td>
</tr>
<tr>
<td>add8_001</td>
<td>2040 µm²</td>
<td>0.718 ns</td>
<td>681.20 µW</td>
<td>42</td>
<td>0</td>
<td>0.00000</td>
<td>0.00000</td>
<td>0.00 %</td>
<td>0</td>
<td>0 %</td>
<td>0.000 %</td>
<td>Verilog, Matlab</td>
</tr>
<tr>
<td>add8_002</td>
<td>836 µm²</td>
<td>1.282 ns</td>
<td>194.75 µW</td>
<td>13</td>
<td>140448</td>
<td>1.69531</td>
<td>5.85038</td>
<td>0.88 %</td>
<td>7</td>
<td>100 %</td>
<td>71.484 %</td>
<td>Verilog, Matlab</td>
</tr>
<tr>
<td>add8_003</td>
<td>912 µm²</td>
<td>0.379 ns</td>
<td>286.66 µW</td>
<td>20</td>
<td>192640</td>
<td>9.64844</td>
<td>138.25000</td>
<td>5.21 %</td>
<td>24</td>
<td>100 %</td>
<td>96.875 %</td>
<td>Verilog, Matlab</td>
</tr>
<tr>
<td>add8_004</td>
<td>708 µm²</td>
<td>1.213 ns</td>
<td>205.54 µW</td>
<td>9</td>
<td>134628</td>
<td>1.37500</td>
<td>3.25000</td>
<td>0.75 %</td>
<td>5</td>
<td>200 %</td>
<td>76.562 %</td>
<td>Verilog, Matlab</td>
</tr>
</tbody>
</table>

Approximate multipliers (471), exact multipliers (28)

<table>
<thead>
<tr>
<th>Circuit</th>
<th>Est. area</th>
<th>Est. delay</th>
<th>Est. power</th>
<th>Nodes</th>
<th>HD</th>
<th>MAE</th>
<th>MSE</th>
<th>MRE</th>
<th>WCE</th>
<th>WCRE</th>
<th>EP</th>
<th>OPS</th>
</tr>
</thead>
<tbody>
<tr>
<td>mul8_000</td>
<td>9224 µm²</td>
<td>3.015 ns</td>
<td>4933.22 µW</td>
<td>137</td>
<td>176134</td>
<td>98.52710</td>
<td>27520.00000</td>
<td>1.99 %</td>
<td>820</td>
<td>500 %</td>
<td>86.490 %</td>
<td>Verilog, Matlab</td>
</tr>
<tr>
<td>mul8_001</td>
<td>5200 µm²</td>
<td>3.366 ns</td>
<td>2524.84 µW</td>
<td>91</td>
<td>310762</td>
<td>239.95550</td>
<td>108908.84375</td>
<td>5.36 %</td>
<td>1671</td>
<td>100 %</td>
<td>98.169 %</td>
<td>Verilog, Matlab</td>
</tr>
<tr>
<td>mul8_002</td>
<td>6715 µm²</td>
<td>2.086 ns</td>
<td>2789.47 µW</td>
<td>132</td>
<td>338806</td>
<td>329.88147</td>
<td>207883.35278</td>
<td>6.70 %</td>
<td>2193</td>
<td>700 %</td>
<td>98.482 %</td>
<td>Verilog, Matlab</td>
</tr>
<tr>
<td>mul8_003</td>
<td>4172 µm²</td>
<td>1.963 ns</td>
<td>1816.06 µW</td>
<td>79</td>
<td>376002</td>
<td>624.46875</td>
<td>67898.57422</td>
<td>10.00 %</td>
<td>2911</td>
<td>700 %</td>
<td>98.984 %</td>
<td>Verilog, Matlab</td>
</tr>
<tr>
<td>mul8_004</td>
<td>5034 µm²</td>
<td>1.944 ns</td>
<td>1893.73 µW</td>
<td>104</td>
<td>382402</td>
<td>639.22653</td>
<td>709554.15625</td>
<td>9.76 %</td>
<td>3143</td>
<td>253 %</td>
<td>99.071 %</td>
<td>Verilog, Matlab</td>
</tr>
</tbody>
</table>

Synthesis results for 45 nm and 180 nm technology (Synopsys Design Compiler)
7 error metrics
New: 12-bit multipliers online, 16 – 32-bit multipliers completed

http://www.fit.vutbr.cz/research/groups/ehw/approxlib/
Incorrect subspace: The subset of input vectors for which the correct circuit and approximate circuit produce different outputs.

F (under-approximation):
Incorrect subspace is a subset of the on-set. 1 → 0 errors are produced

H (over-approximation)
Incorrect subspace is a subset of the off-set. 0 → 1 errors are produced

At most one of the circuits is allowed to produce an incorrect output for any input vector.

Energy-efficient implementation of ANNs

Approximations proposed:

- Pruning – weights and neurons
- Data compression (weights)
- Memory – approximate cells and Load/Store

- Datapath
  - Reducing data bit-width

- Multiplication
  (~45% of total power)
  - Multiplierless multiplication
  - Weights: {-1, 1}
- Activation function
- Sum function

[Judd et.al. WAPCO’16]
Energy-efficient implementation of ANNs

MNIST dataset classification: 32x32 – 100 – 10 MLP network (classification accuracy 94.16% with accurate implementation). We introduced an approximate multiplier by adding a jitter function $\Delta(a, b)$, resulting in a 5.2% error for multiplication.

**Scenario A:**
- Multiplication $m(a, b) = a \cdot b + \Delta(a, b)$
- Classification accuracy: 10.77%

**Scenario B:**
- 80% of multiplications are by 0
- Multiplication $m'(a, b) = \begin{cases} 0 & \text{if } a = 0 \lor b = 0 \\ a \cdot b + \Delta(a, b) & \text{otherwise} \end{cases}$
- Classification accuracy: 94.20%

CGP in approx. multiplier design for ANNs

Accurate multiplier – initial circuit (6)

- CSAM RCA, CSAM RCA, RCAM, WTM CLA, WTM CSA, WTM RCA

Allowed errors: \( \varepsilon \in \{0.5\%, 1\%, 2\%, 5\%, 10\%, 15\%, 20\%\} \)

CGP parameters

- \( n_i \in \{14,22\}; n_o \in \{14,22\}; n_r = 1; 250 < n_c < 780 \)
- Functions: \{NOT, AND, NAND, OR, NOR, XOR, XNOR\}
- Error constraints:
  1. \( \forall a, b: |m(a, b) - a \times b| \leq \varepsilon \cdot 2^{n_o} \)
  2. \( \forall a: m(a, 0) = m(0, a) = 0 \)
- Fitness function:
  \[ C(m) = \begin{cases} -\text{GatesCount}(m) & \text{if constraints (1) and (2) met,} \\ -\infty & \text{otherwise} \end{cases} \]
CGP in approx. multiplier design for ANNs

- In total, 852 approximate 7-bit and 11-bit multipliers were evolved by CGP.
- Multipliers were sign-extended using one’s complement.
- The 8-bit and 12-bit multipliers were applied in NNs.
- The NNs were retrained with approximate multiplication operation using the backpropagation algorithm.
- Approximate multipliers showing the best trade off between power and accuracy in NN were selected (for different error targets).
Evolved approximate multipliers for ANNs

Results of synthesis of sign-extended multipliers with Synopsys DC 45 nm technology

Timing:
- 8-bit multipliers: 2.5 GHz
- 12-bit multipliers: 2 GHz

Accurate multiplier was implemented in Verilog using standard * arithmetic operator

Energy-efficient implementation of ANNs: MLP

- Handwritten number dataset (dataset used for benchmarking)
- Fully connected MLP network
- 28x28 inputs, 300 hidden neurons, 10 outputs
- 60k training images
- 10k testing images
- More than 238k multiplications for approximation
- Initial classification accuracy:
  - 8b: 97.67%
  - 12b: 97.70%
Energy-efficient implementation of ANNs: LeNet

- Complex real-world problem
- Convolutional LeNet NN
- 278,104 multiplications in 6 layers
- 73k training images
- 26k testing images
- Approximation introduced in L1, L3, L5 and L6 layers
- Initial classification accuracy:
  - 8b: 86.85%
  - 12b: 86.90%

![Input image](32x32) 6@28x28 6@14x14 16@10x10 16@5x5 120@1x1 10 values
Energy-efficient implementation of ANNs: Summary

Classification Accuracy and power reduction (in multiplication)

Power (8 bit) (12 bit) | -20% | -30% | -57% | -77% | -82% | -91% | -91% | -36% | -25% | -9%

Approximation error $\varepsilon$ of multipliers

- 0%: 0%
- 0.50%: 5%
- 1%: 10%
- 2%: 15%
- 5%: 20%
- 10%: 25%
- 15%: 30%
- 20%: 35%
- 25%: 40%
- 30%: 45%
- 35%: 50%
- 40%: 55%
- 45%: 60%
- 50%: 65%
- 55%: 70%
- 60%: 75%
- 65%: 80%
- 70%: 85%
- 75%: 90%
- 80%: 95%
- 85%: 100%

Multiplierless multiplication by Sarwar et al. DATE’2016

Circuit approximation with CGP and BDD

• Three criteria
  • relative area, delay and error
  • Error is the average Hamming distance (10 target error values $E_i = 0.1 \ldots 0.9 \%$)

• CGP parameters
  • Rows = 1; Columns = # of gates in the original circuit
  • 5 mut./chromosome, $\lambda = 5$, 30 min/run, 10 independent runs
  • Function set (relative area): and (1.333), or (1.333), xor (2.0), nand (1.0), nor (1.0), xnor (2.0), buf (1.333), inv (0.667)

• Two stages:
  • Find a circuit showing $E_i$, but a small (< 5%) imperfection tolerated
  • weight fitness (error / area / delay): $(w_e; w_a; w_d) = (0.12; 0.5; 0.38)$
    (but the error still kept under 5% of $E_i$)

• 16 benchmark circuits

Hamming distance using BDDs

- Create ROBDD for the parent circuit $C_A$, the offspring circuit $C_B$ and the XOR gates.
- Average Hamming distance:

$$e_{HD} = \frac{1}{2^{\text{inputs}}} \sum_{i=1}^{\text{outputs}} \text{SatCount}(z_i)$$

**SatCount** ($f$) – gives the number of input assignments for which $f$ is ‘1’.

\[
\text{SatCount}(z_1) = 2 \\
\text{SatCount}(z_2) = 0
\]

<table>
<thead>
<tr>
<th>$x_1$</th>
<th>$x_2$</th>
<th>$x_3$</th>
<th>$x_4$</th>
<th># combinations</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
</tbody>
</table>

CGP with BDD in the fitness function: Example 1

- Clmb (bus interface): 46 inputs, 33 outputs
- Original clmb: 641 gates, 19 logic levels, $|\text{BDD}| = 6966$, $|\text{BDD}_{\text{opt}}| = 627$ (SIFT in 2.3 s)
- Optimized by CGP (no error allowed):
  - Best: 410 gates, 12 logic levels -- in 29 minutes ($2.9 \times 10^6$ generations)
  - Median: 442 gates, 13 logic levels

Properly optimize before doing approximations!
Approximate circuits: CGP with SAT solver

- Worst case absolute error (WCAE) computation based on SAT solving (for adders and multipliers)
- Improved miter construction
- SAT solver terminated if no decision after spending a predefined time.
- Integrated to ABC

\[
\text{fitness}(C) = \begin{cases} 
\text{size}(C) & \text{if } \text{WCAE}(C) < \tau \\
\infty & \text{else}
\end{cases}
\]

- 16-bit multipliers for 9 target WCAE
- 2 hours/1 run
- 30 circuits analyzed for each WCAE
- Synopsys Design Compiler, 45 nm
- \(L\) is the max. number of conflicts for an AIG node, \(L = 160\,K\) (~120 seconds) and \(L = 20\,K\) (~3 seconds).
Approximate 16-bit multipliers: Comparison

- M1: lpACLib, DAC’15 (2x2 multipliers composition)
- M2: BSDLC (DATE 2016)
- M3: Bit-width truncation
- M4: Kulkarni, 2011 (2x2 multipliers composition)
- M5: EvoApproxLib (8x8 multipliers composition)

Truncated multi. 15x15, 14x14 etc.

Evolved approx. multipliers
Approximate adders and multipliers (exact error)
Non-linear image filters

Corrupted image
(10% pixels, impulse noise)

Filtered image
(9-input median filter)
Non-linear image filters: Approximation strategies

- Approximation of the comparator element

- Approximation of the network (pruning)
  - CGP used to find a network of $N$ comparators minimizing the error w.r.t. the original median (consisting of $K$ comparators), but resources are limited, i.e. $N < K$.

- Evolutionary image filter design from scratch
  - CGP used to evolve an image filter showing a minimal error and cost. Filters are composed of elementary 2-input functions (min, max, +, logic functions over 8 bits).
Approximate median using CGP

- Median network (consisting of up to $N$ operations) is represented by means of a one-dimensional array of $N$ nodes.
- Each node can act as: identity (0), minimum (1), maximum (2) over 8 bits.
- Each candidate solution is encoded using $3N + 1$ integers.
- Fitness function (single objective)
  \[
  \text{error} = \sum_{i \in S} |O_{\text{candidate}}(i) - O_{\text{reference}}(i)|
  \]
- Example for a 3-input median:

Chromosome: 0, 2, 3; 3, 2, 0; 0, 2, 2; 5, 3, 1; 6, 1, 2; 7, 0, 0; 6, 8, 2; 8
Approximate median using CGP

Experimental setup

- (1+4)-ES, no crossover, 5% of the chromosome mutated

<table>
<thead>
<tr>
<th></th>
<th>Median-9</th>
<th>Median-25</th>
</tr>
</thead>
<tbody>
<tr>
<td>Inputs</td>
<td>9</td>
<td>25</td>
</tr>
<tr>
<td>Outputs</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>Generations</td>
<td>$3 \times 10^6$ (3 hours)</td>
<td>$3 \times 10^5$ (3 hours)</td>
</tr>
<tr>
<td>Training vectors</td>
<td>$1 \times 10^4$</td>
<td>$1 \times 10^5$</td>
</tr>
<tr>
<td>Exact solution (K)</td>
<td>38 operations</td>
<td>220 operations</td>
</tr>
<tr>
<td>Available nodes (N)</td>
<td>6 – 34 operations</td>
<td>10 – 200 operations</td>
</tr>
</tbody>
</table>

Approximate median: Distance error analysis

9-input median
fully-working: 38 operations

21% reduction

52% reduction

84% reduction

25-input median
fully-working: 220 operations

27% reduction

54% reduction

81% reduction

Evolutionary design of image filters from scratch

\[ \text{fitness} = \sum_{i=1}^{N} \sum_{j=1}^{M} |v(i, j) - w(i, j)| \]
Comparison of approximate median filters and evolved filters for salt and pepper noise

PSNR – mean PSNR on 30 images
Synopsys Design compiler; 45 nm PDK
All filters are pipelined with $f_{\text{min}} = 1 \text{ GHz}$
Conclusions – Part II

• Design automation methods implementing functional circuit approximation
  – work at various levels (abstract, source code, RTL, gate),
  – use different strategies and heuristics to introduce the approximation (truncation, pruning, component replacement, local re-synthesis, ...),
  – evaluate the quality of approximate circuits by means of simulation, probabilistic or formal methods,
  – have not been systematically compared in terms of quality.

• CGP-based methods can provide quite competitive approximate circuits
  – at different levels of abstraction (very flexible representation),
  – with formally proven quality of result (when needed),
  – because the problem can be formulated as a multi-objective one with various constraints and solved by means of a multi-objective approach,
  – but it is a computationally demanding approach.

• Properly optimize before doing approximations!
See references on particular slides

Selected tutorial and survey papers on Approximate Computing

- J. Han and M. Orshansky, “Approximate computing: An emerging paradigm for energy-efficient design,” in Proc. of the 18th IEEE European Test Symposium. IEEE, 2013, pp. 1–6
Acknowledgement

• EHW group at Brno University of Technology
  – Zdeněk Vašíček, Michal Bidlo, Roland Dobai
  – Michaela Šikulová, Radek Hrbáček, Vojtěch Mrázek, David Grochol, Miloš Minařík, Jakub Husa, Marek Kidoň, Michal Wiglasz and other students

• Research funding
  – IT4Innovations Centre of Excellence – National supercomputing center
  – IT4Innovations excellence in science - LQ1602
  – Advanced Methods for Evolutionary Design of Complex Digital Circuits, 2014 – 2016 (Czech Science Foundation)
  – Relaxed equivalence checking for approximate computing, 2016 – 2018 (Czech Science Foundation)
  – Brno University of Technology