Approximate Computing:

An Evolutionary Computing Perspective

(Tutorial at IEEE CEC 2017)

Lukáš Sekanina

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

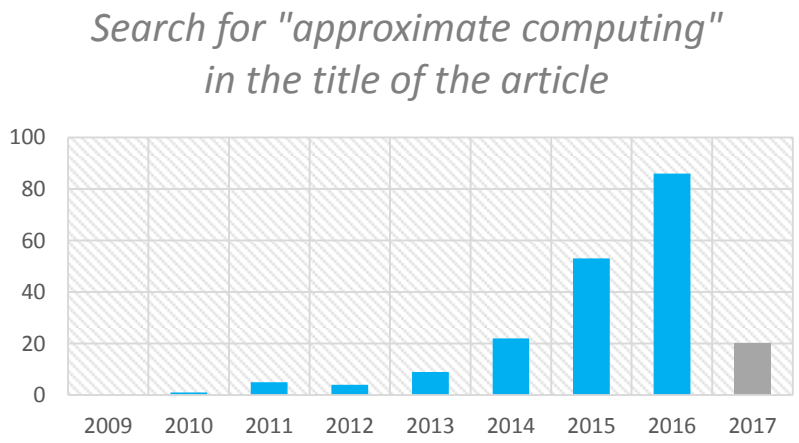
The notion of approximation is well known in …

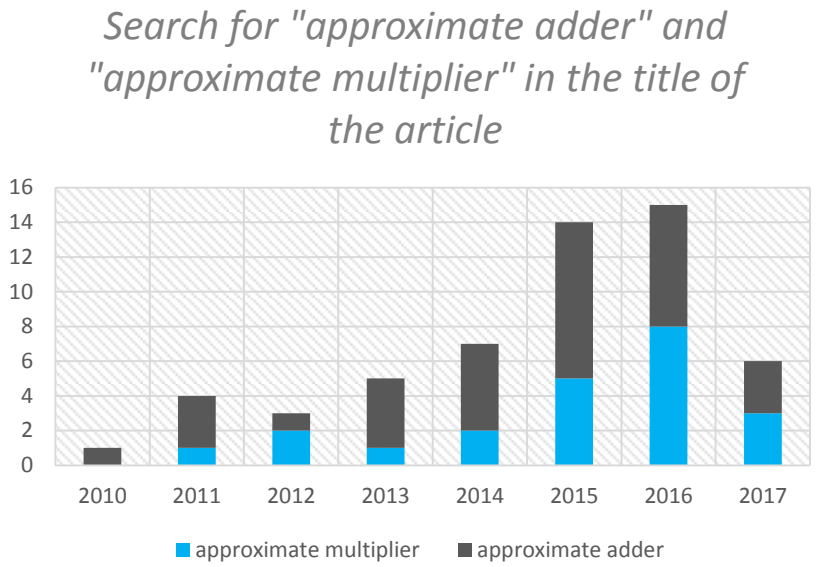
- Theoretical computer science
  - polynomial time approximation algorithms to find approximate solutions to NP-hard optimization problems
- Stringology, bioinformatics
  - approximate string matching
- Bio-inspired models in AI
  - approximation of functions using artificial neural networks
- Mathematics
  - approximation of functions; numerical mathematics
- Computer engineering
  - FP numbers, computer arithmetic, …
  - *S. H. Nawab et al.: Approximate Signal Processing, Journal of VLSI Signal Processing , vol. 15, pp. 177-200, Jan. 1997.*

Why Approximate Computing again?

Approximate computing (AC) in literature

- *Search for "approximate computing" in articles by Google Scholar (April, 2017)*

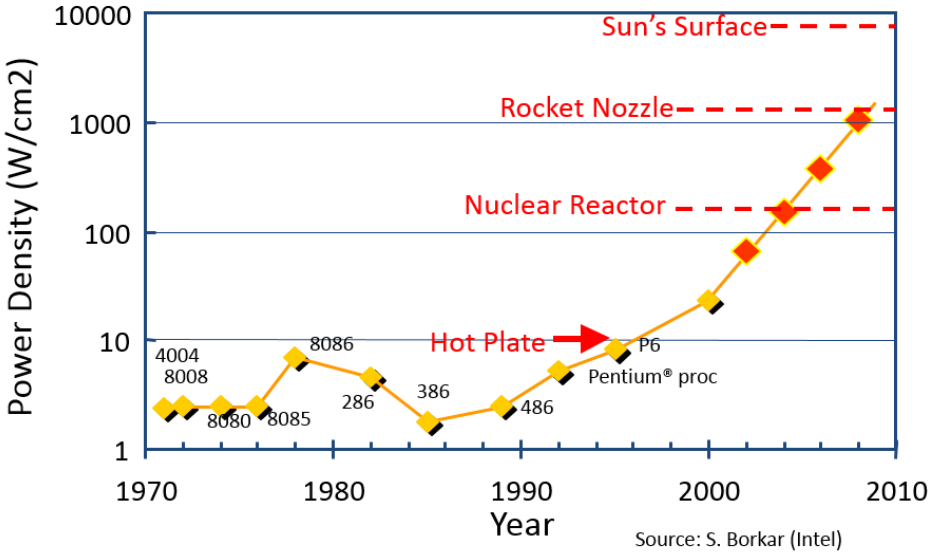


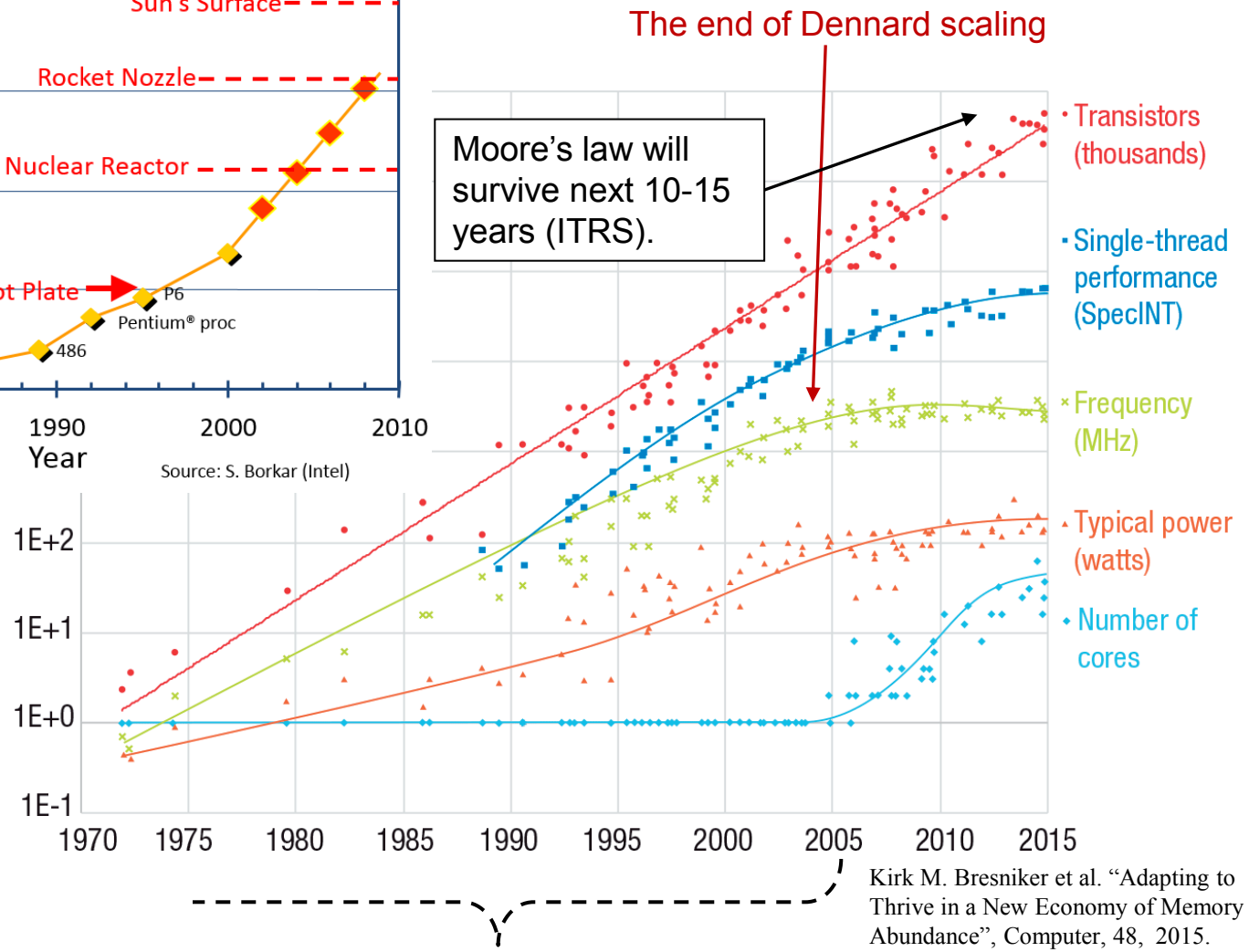


Tutorial Outline

- Approximate Computing
  - Motivation
  - Error resilience
  - Sensitivity analysis and error metrics
  - Overview of approximation techniques
- Evolutionary algorithms and genetic improvement
- EA in SW approximations
  - Extension of Java - ExpAX
  - Median
- EA in HW approximations
  - Approximations at the hardware description language level
  - Approximate multipliers in ANN
  - Library of approximate components
- Formal relaxed equivalence checking in approximate computing
  - Binary decision diagrams
  - Approximate circuits with formal error guarantees
- Conclusions

The end of Dennard scaling





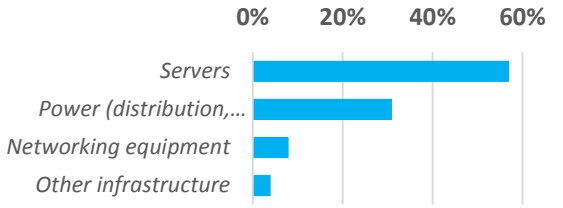
Kirk M. Bresniker et al. "Adapting to Thrive in a New Economy of Memory Abundance", Computer, 48, 2015.

Power density (power consumption per chip area) remained acceptable from one technology node to another even with increasing frequency (based on data for common processors).

Roots of the new interest in AC

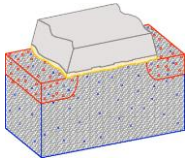
- Energy efficiency and dark silicon
  - High performance & low power computing is requested (Big data processing in data centres; Mobile electronics with limited power budget)

Data centers - monthly costs



Gupta, Accelerating Datacenter Workloads Intel, FPL 2016

- Variability issues
  - Many "unreliable" components on a chip fabricated with modern process technologies (Limited use of fault tolerant mechanisms because they are expensive; Reliable computing with unreliable components)

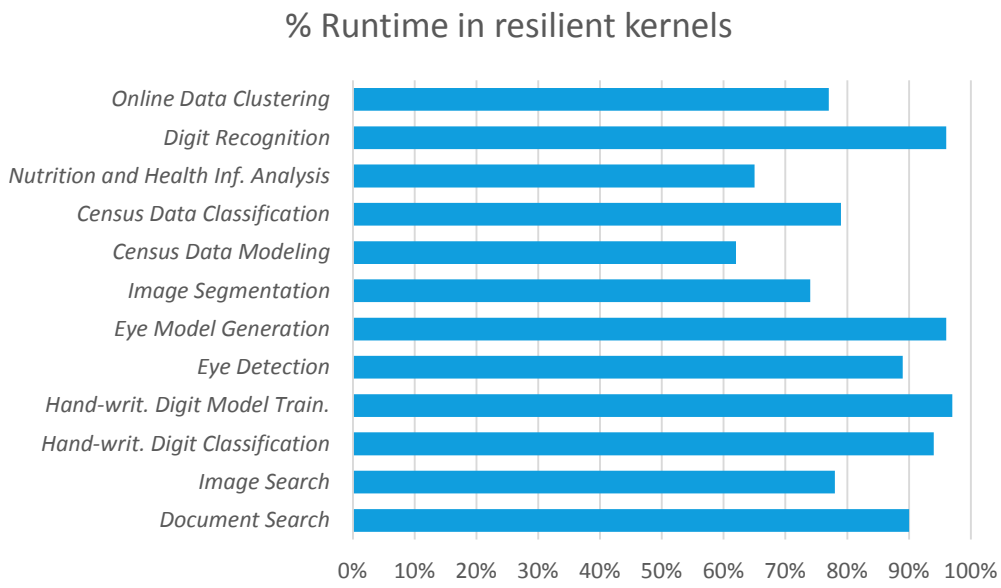


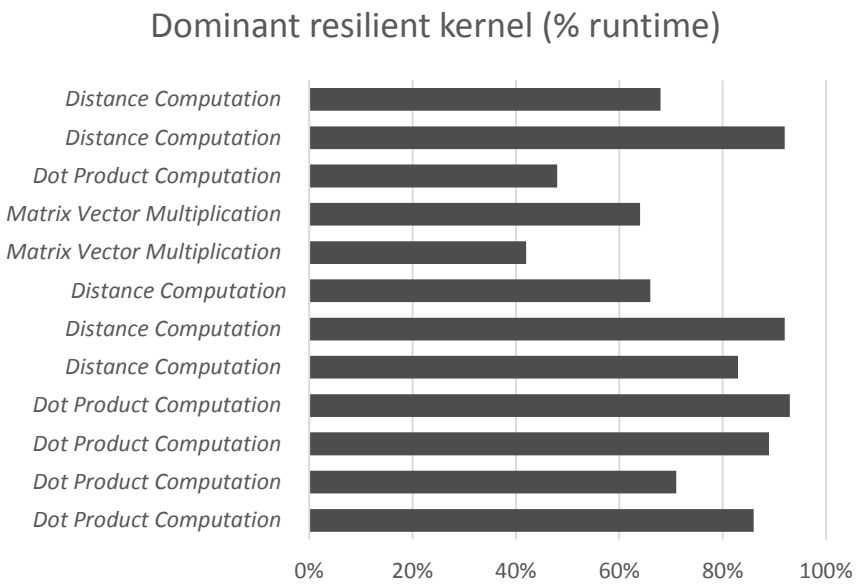
22nm MOSFET (line-edge and surface roughness, random dopant fluctuations => threshold voltage variation)

- Error resilience
  - Many applications are error-resilient. (We are willing to tolerate errors.)

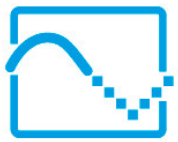
Error resilience

- Typical machine learning, signal processing and document processing applications have a mix of resilient and sensitive computations.
- Chippa, 2013: 83% runtime spent in computations (such as matrix and vector operations) that can be approximated.





The origin of resilience



Applications with analog inputs

– image processing, sensor data processing, voice recognition, etc., that operate on noisy real-world data. They are inherently resilient to some noise.

Applications with analog output

– multimedia, image rendering, sound synthesis, etc. The output is intended for human perception and can inherently tolerate errors imperceptible to users.

Applications with no unique answer

– web search, machine learning, autonomous agents, etc., which do not offer a unique answer and multiple possible answers are acceptable.

Iterative and convergent applications

– data analytics and numerical computations that iteratively process large amounts of data. They often sample data, stop the convergence procedure early, or apply approximate heuristics.

Approximate computing

- The concept of approximate computing has been developed in different ways and at various levels of computing stack (circuit, component, memory, processor, compilers, applications, …)
- Software-level approximations
  - Extensions of general purpose languages (Java, Verilog) to support approximations in data types, operators, … e.g. EnerJ, Axilog, ExpAx …
  - Neural network replaces general purpose code [Esmaeilzadeh el al., 2013]
- Specialized processors supporting approximate computing
  - Improving Efficiency of Extensible Processors by Using Approximate Custom Instructions [Kamal et al., 2014]
- Circuit approximation
  - over-scaling based approximations
  - functional approximations
- Memory approximation
  - approximations in memory cells, organization, access, hierarchy …

What is Approximate computing?

"Approximate computing exploits the gap between the level of accuracy required by the applications/users and that provided by the computing system, for achieving diverse optimizations."

[Mittal S., ACM Computing Surveys 2016]

"The requirement of exact numerical or Boolean equivalence between the specification and implementation of a circuit is relaxed in order to achieve improvements in performance or energy efficiency."

[Venkatesan et al., 2011]

"Computing efficiently by producing results that are good enough or of sufficient quality."

[Venkataramani et al., DAC 2015]

Approximate computing as a new paradigm

- Approximations are conducted across the whole computer stack:
  - Circuit
  - Component
  - Memory
  - (Parallel) Processor architecture
  - Algorithm
  - Compiler
  - Operating system
  - Application
- A holistic approach is needed to find the best trade-off between error, power and performance at the global (system) level.
- AC reduced energy requirements of many applications: image processing, video processing, deep neural networks, …

A great opportunity for EAs!

Sensitivity analysis

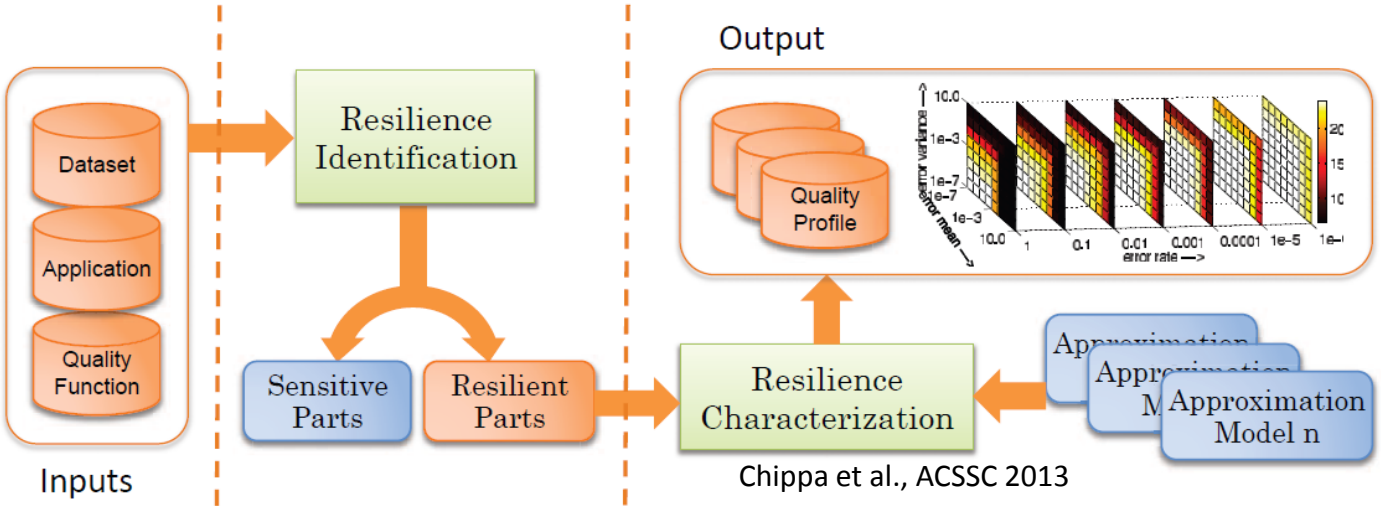
- The goal is to identify subsystems suitable for undergoing the approximation.
- Method: Random/guided modification of the original implementation and statistical evaluation of the impact on the quality of result.

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

Error metrics used in approximate computing

- 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

$$e_{wst}(f,\hat{f}) = \max_{\forall x \in \mathcal{B}^n} | \operatorname{int}(f(x)) - \operatorname{int}(\hat{f}(x)) |$$

$$e_{rel}(f,\hat{f}) = \max_{\forall x \in \mathcal{B}^n} \frac{| \operatorname{int}(f(x)) - \operatorname{int}(\hat{f}(x)) |}{\operatorname{int}(f(x))}$$

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

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

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

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

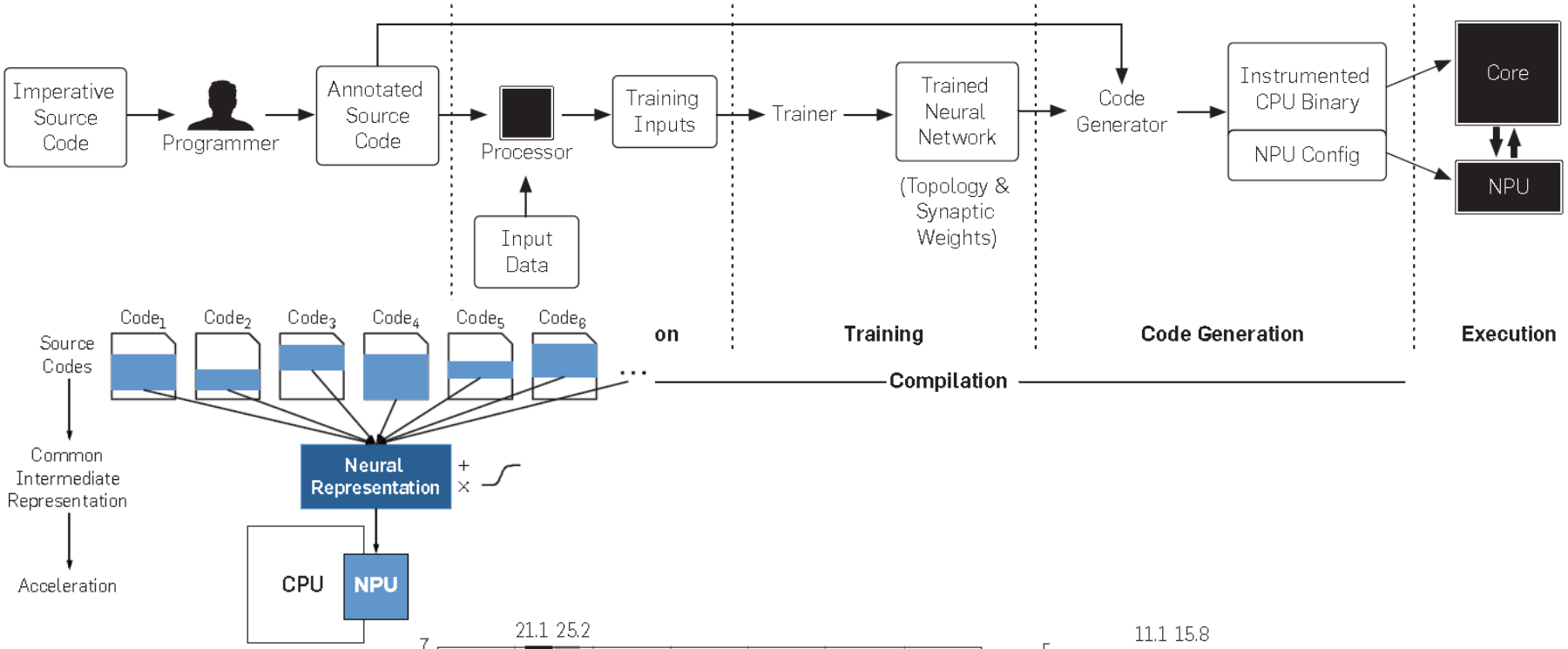
$f, \hat{f}$ – original and approximate solution
n, m – the number of inputs and outputs
int – returns a decimal value from m bits

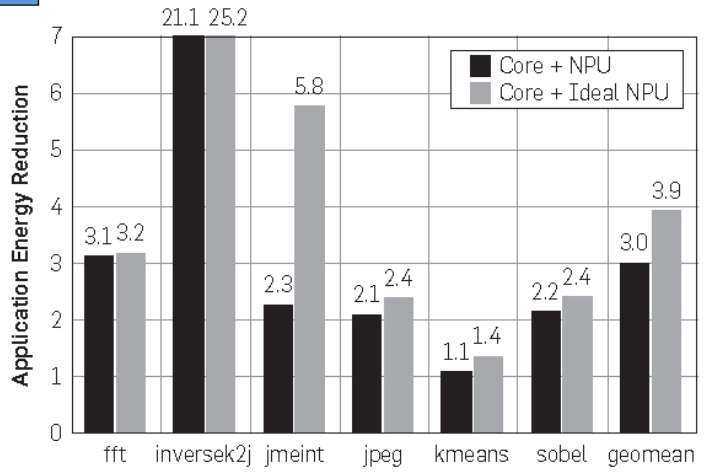
Approximation techniques - examples

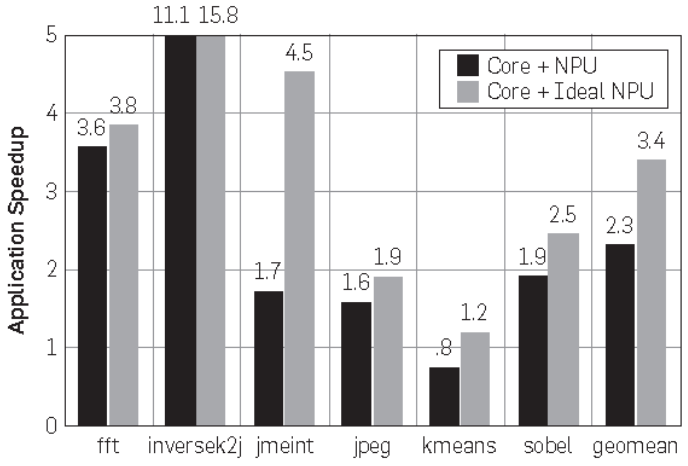
- precision scaling
- loop perforation
- load value approximation
- memorization
- task dropping/skipping
- memory access skipping
- data sampling
- using different program (circuit) versions
- using inexact or faulty hardware
- voltage scaling
- refresh rate reducing
- inexact read/write
- reducing divergence in GPUs
- lossy compression
- use of neural networks.

Mittal S., ACM Computing Surveys, 2016

SW approximation: Code replaced by ANN

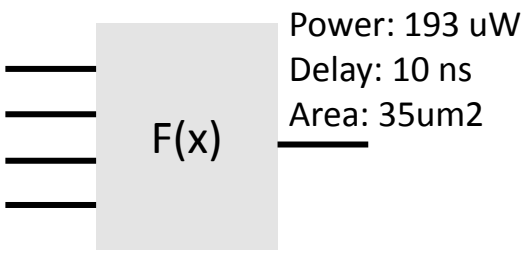


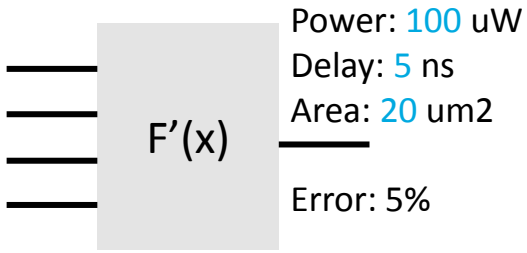


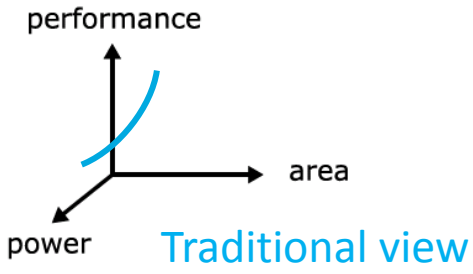


Functional approximation

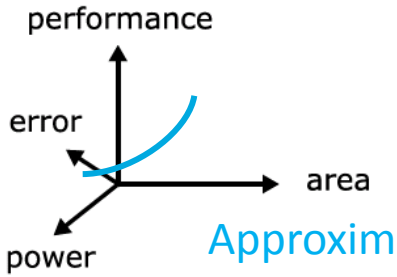
- Principle: To implement a slightly different function that leads to energy/delay/area reduction but a non-zero error.







Traditional view



Approximate computing

Functional equivalence
is requested between the specification and implementation at all levels.

Relaxed functional equivalence

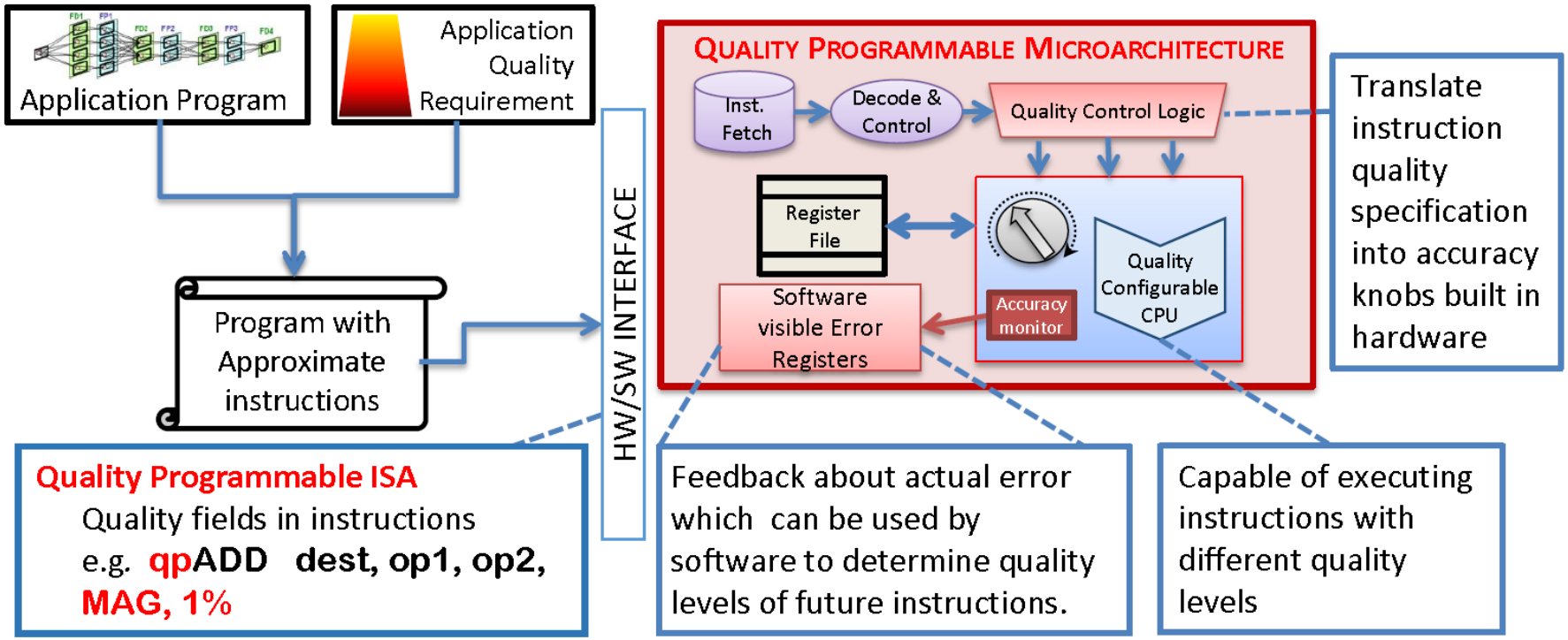
Error as a design metric!

A complex multi-objective design/optimization problem!

Languages supporting approximate computing

- EnerJ [Sampson et al., PLDI 2011]
  - An extension to Java that adds approximate data types.
  - Approximate data can be processed more cheaply but less reliably.
  - Approximate operations 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 enables the designer to relax the accuracy requirements in certain parts of the design, while keeping the critical parts strictly precise.
- Others: ExpAX, Chisel, …
- They require a hardware (CPU) supporting approximate computing

Quality programmable processors: Concept



Example: Quora is an experimental quality configurable vector processor with 289 processing elements in 45 nm technology [Venkataramani et al. Micro 46, 2013]

Timing induced approximations

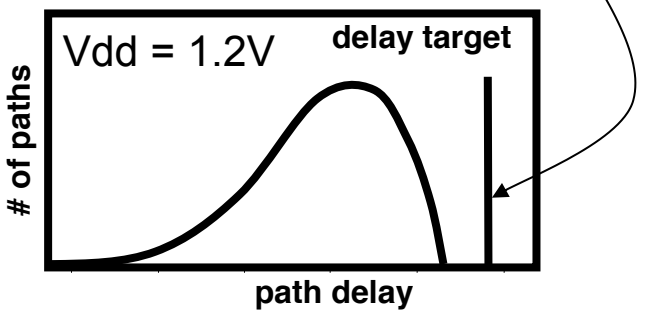
- Power reduction tricks
  - Assume: Accurate circuit D_1 at frequency f_1
  - D_1 is approximated to D_2 which can work at higher freq. f_2 ($f_2 > f_1$)
  - But, D_2 is operated at f_1 with lower Vdd => power saving

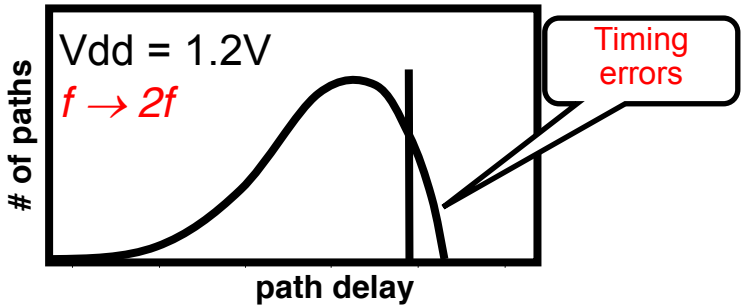
- Design techniques
  - over-clocking
  - voltage over-scaling

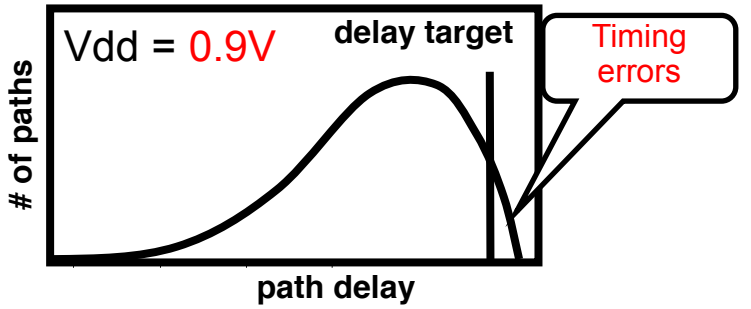
↑speed

↓power

$$P_{dyn} = CV_{dd}^2 f$$







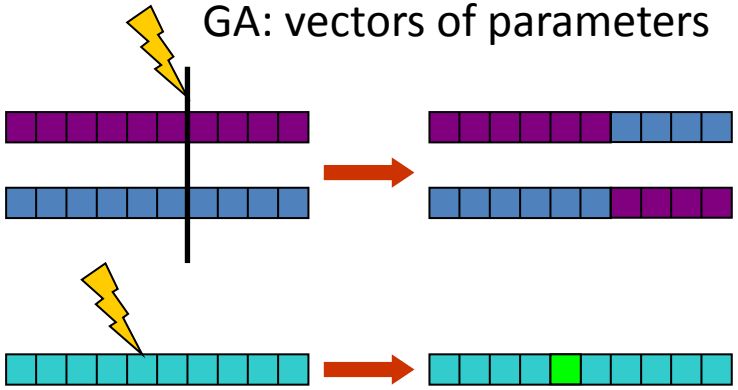
Courtesy of K. Roy

Tutorial Outline

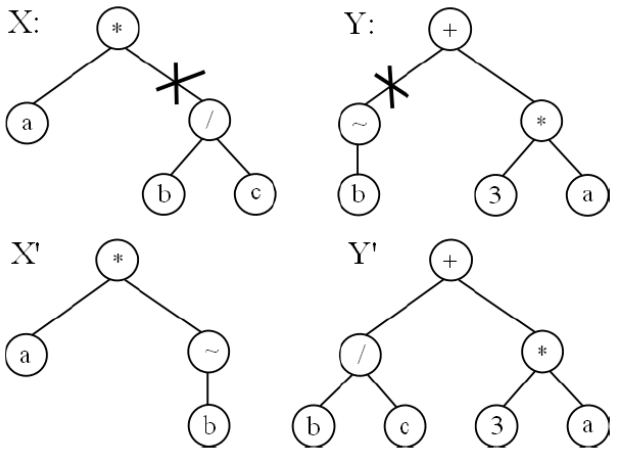
- Approximate Computing
  - Motivation
  - Error resilience
  - Sensitivity analysis and error metrics
  - Overview of approximation techniques
- Evolutionary algorithms and genetic improvement
- EA in SW approximations
  - Extension of Java - ExpAX
  - Median
- EA in HW approximations
  - Approximations at the hardware description language level
  - Approximate multipliers in ANN
  - Library of approximate components
- Formal relaxed equivalence checking in approximate computing
  - Binary decision diagrams
  - Approximate circuits with formal error guarantees
- Conclusions

Evolutionary algorithms: GA, GP, LGP, CGP, GE …

GA: vectors of parameters



GP: syntax trees



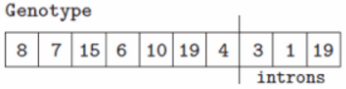
LGP: machine level code

```
double LGP (double x ){
  r[0] = x

  r[2] = r[0] * r[0]
  r[1] = r[2] + r[0]
  r[3] = r[1] + r[0]
  r[3] = r[3] + r[2]
  r[0] = r[2] * r[1]
  r[1] = r[1] + r[4]
  r[0] = r[0] + r[3]
  r[0] = r[1] * r[0]
  return r0
}
```

GE: grammatical evolution

Genotype



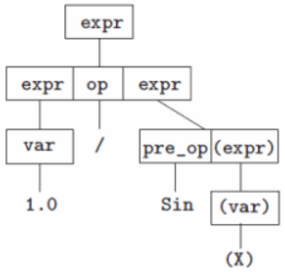
Mapping process

```
<expr>
   8 % 4 = 0
<expr> <op> <expr>
   7 % 4 = 3
<var> <op> <expr>
  15 % 2 = 1
1.0 <op> <expr>
   6 % 4 = 2
1.0 / <expr>
  10 % 4 = 2
1.0 / <pre_op> (<expr>)
   (No calculation, because <pre_op>
    has only one possible choice.)
1.0 / Sin (<expr>)
  19 % 4 = 3
1.0 / Sin (<var>)
   4 % 2 = 0
1.0 / Sin (X)
```

BNF grammar

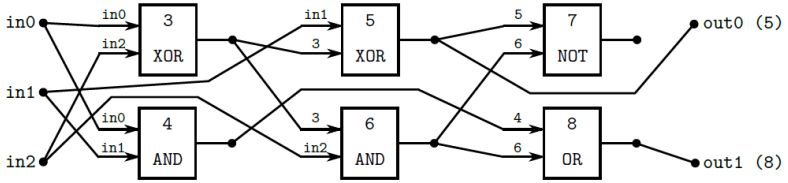
```
N = {expr, op, pre_op, var}
T = {Sin, +, -, /, *, X, 1.0, (, )}
S = {expr}
P =
(1) <expr>    := <expr> <op> <expr>    (0)
               |(<expr> <op> <expr>)   (1)
               |<pre_op> (<expr>)      (2)
               |<var>                  (3)
(2) <op>      := +   (0)
               | -   (1)
               | /   (2)
               | *   (3)
(3) <pre_op>  := Sin
(4) <var>     := X     (0)
               | 1.0   (1)
```

Phenotype



cf. HeuristicLab

Cartesian GP: Directed acyclic graphs



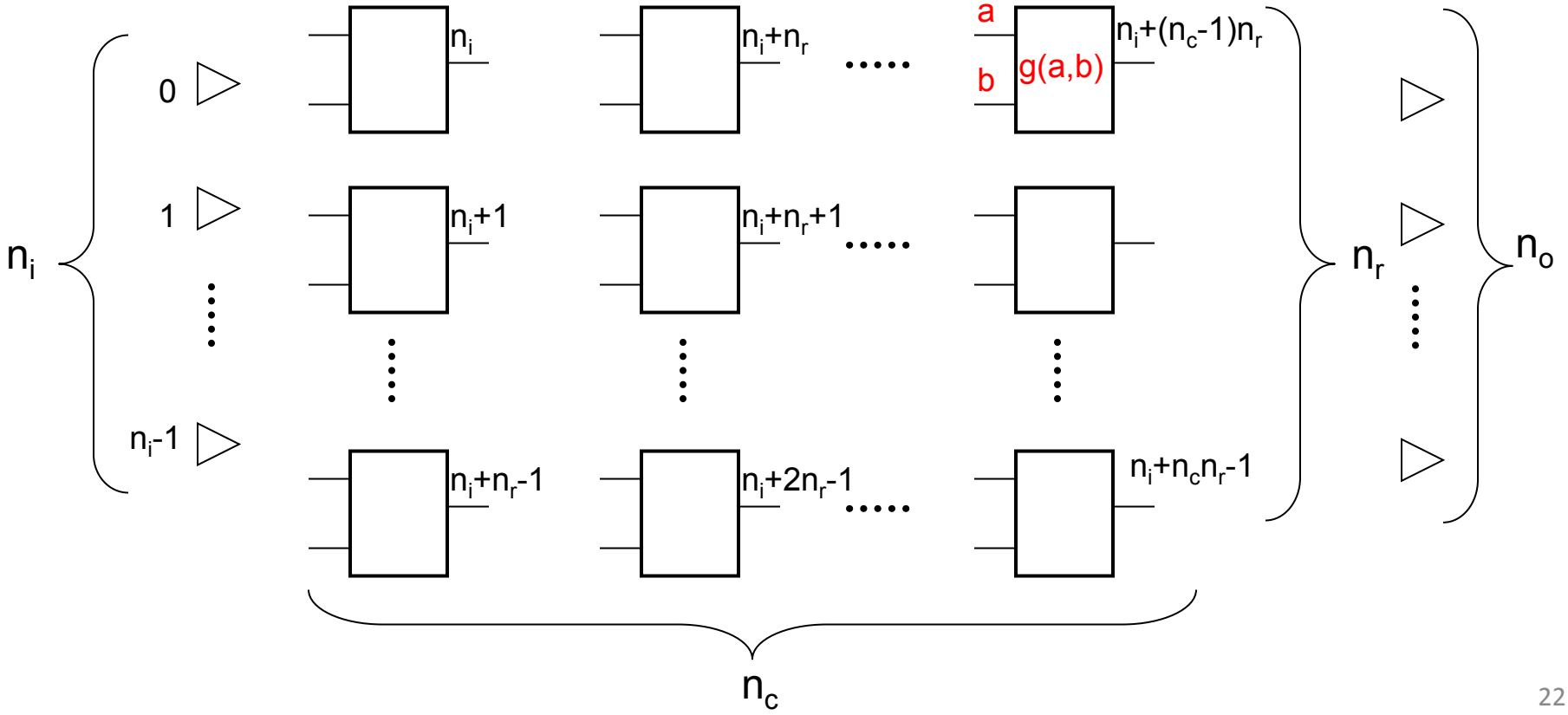
(0, 2, 2) (0, 1, 0) (1, 3, 2)(3, 2, 0) (5, 6, 3) (4, 6, 1) (5, 8)

Cartesian Genetic Programming (CGP) [Miller, 1999]

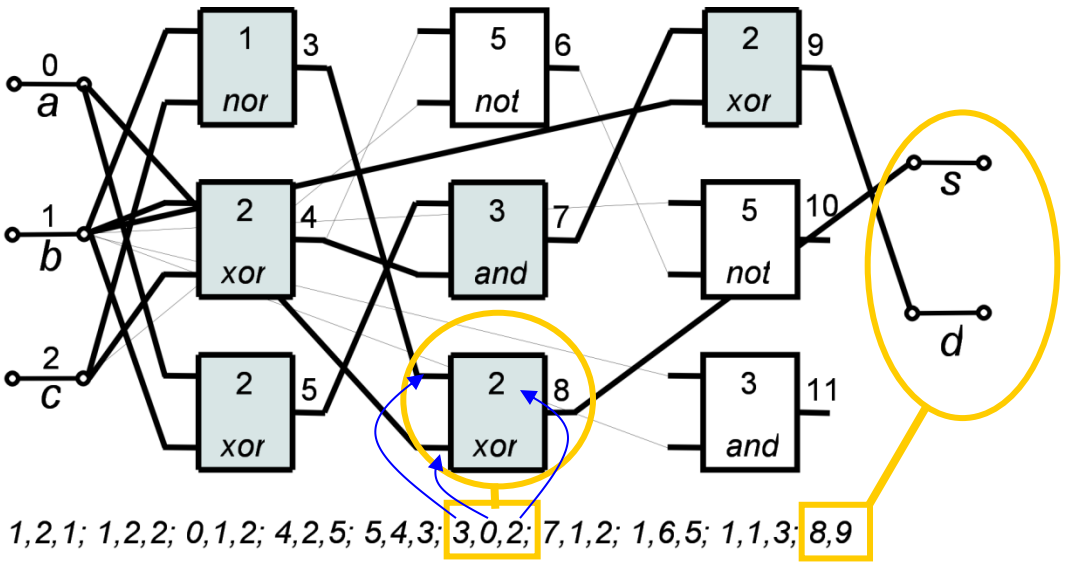
- n_i primary inputs
- n_o primary outputs
- n_c columns
- n_r rows

- n_a inputs of each node
- Γ 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



- 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)
  - Γ= {NAND$^{(0)}$, NOR$^{(1)}$, XOR$^{(2)}$, AND$^{(3)}$, OR$^{(4)}$, NOT$^{(5)}$}

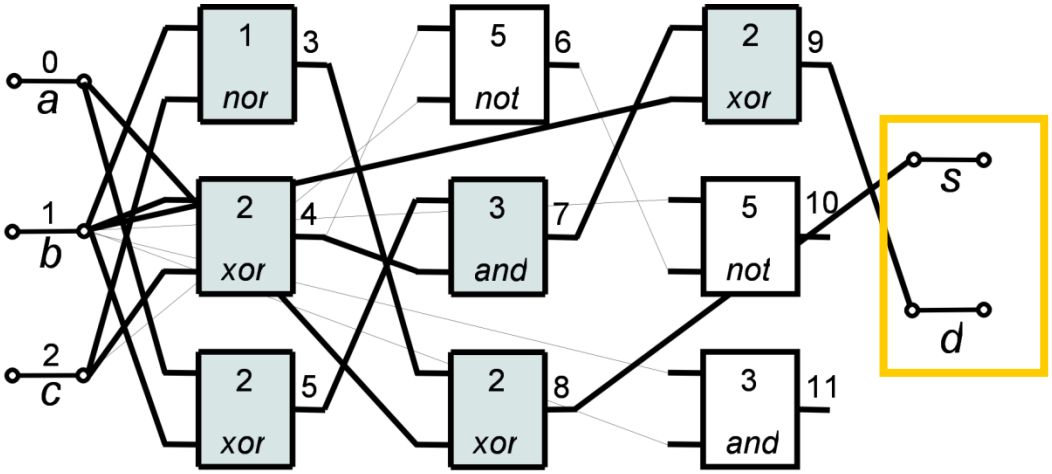
Genotype (netlist):

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: Fitness function for circuit design



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

a	b	c	d	s
0	0	0	0	0
0	0	1	0	1
0	1	0	0	1
0	1	1	1	0
1	0	0	0	1
1	0	1	1	0
1	1	0	1	0
1	1	1	1	1

=> fitness = 16

Specification (1-bit adder), target table:

a	b	c	d	s
0	0	0	0	1
0	0	1	0	0
0	1	0	0	0
0	1	1	1	0
1	0	0	0	0
1	0	1	1	1
1	1	0	1	1
1	1	1	1	1

=> fitness = 10

Typical fitness function (circuit functionality):

$$f = \sum_{i=1}^{K} |y_i - w_i|$$

K ← The number of test vectors

w_i ↑ Desired response

y_i ↑ Circuit response

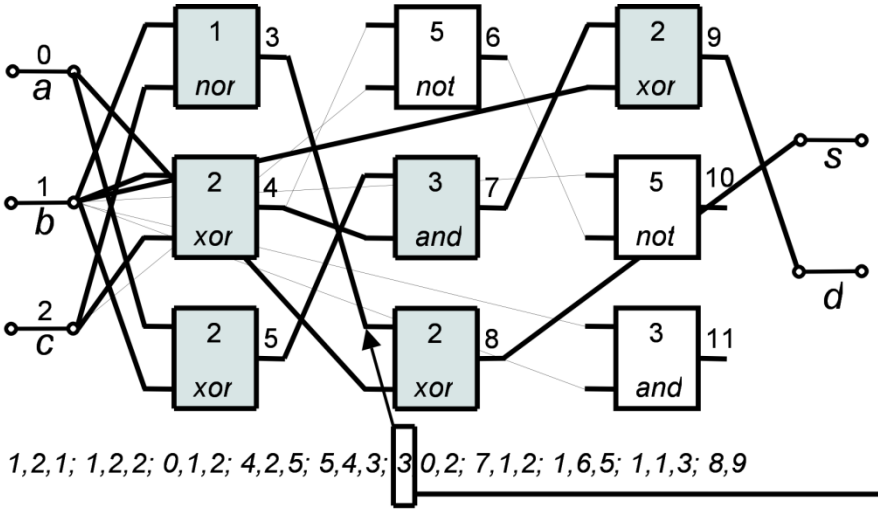
Additional objectives:

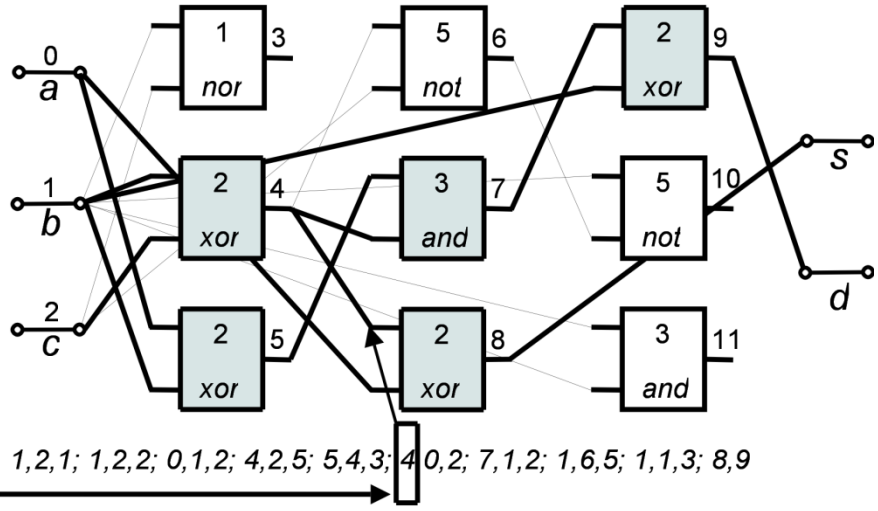
- area (the number of gates)
- delay
- power consumption etc.

$K = 2^{inputs}$ for combinational circuits. Not scalable!!!

CGP: Mutation-based search

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





mutation

(a)

a	b	c	d	s
0	0	0	0	1
0	0	1	0	0
0	1	0	0	0
0	1	1	1	0
1	0	0	0	0
1	0	1	1	1
1	1	0	1	1
1	1	1	1	1

=> fitness = 10

(b)

a	b	c	d	s
0	0	0	0	0
0	0	1	0	1
0	1	0	0	1
0	1	1	1	0
1	0	0	0	1
1	0	1	1	0
1	1	0	1	0
1	1	1	1	1

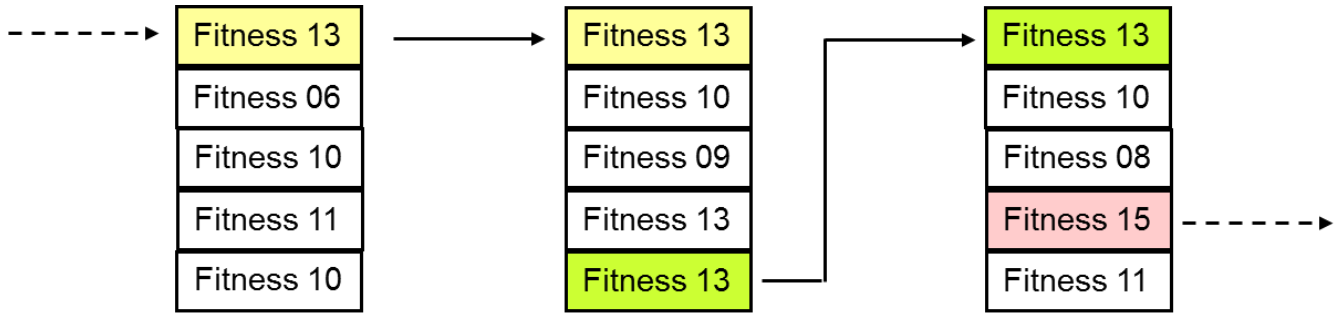
=> fitness = 16
(for full adder)

CGP: Search algorithm (1 + λ)

Algorithm 1: CGP

Input: CGP parameters, fitness function
Output: The highest scored individual p and its fitness

```
1 P ← randomly generate population
2 EvaluatePopulation(P);
3 while ⟨terminating condition not satisfied⟩ do
4     α ← highest-scored-individual(P);
5     if fitness(α) ≥ fitness(p) then
6         p ← α;
7     P ← create λ offspring of p using mutation;
8     EvaluatePopulation(P);
9 return p, fitness(p);
```

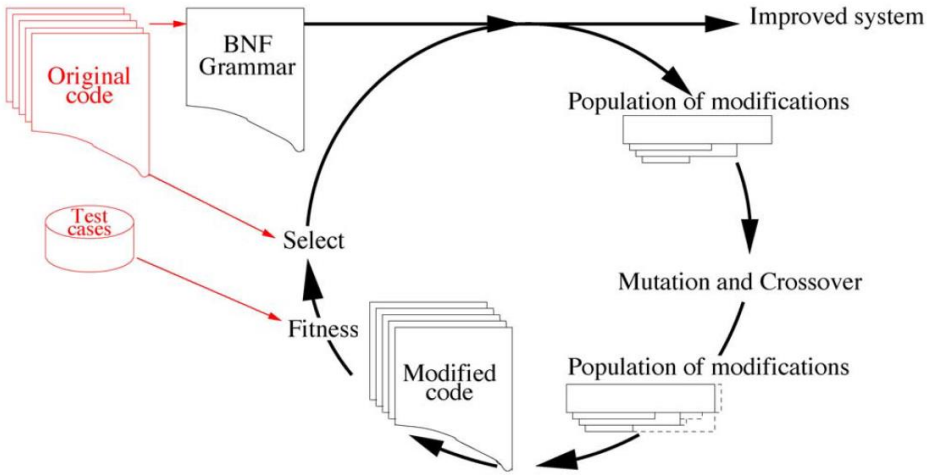


Why EA in Approximate computing?

- Why EA in AC?
  - In AC, 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).
- Is AC similar to Genetic improvement?
  - Genetic improvement (of existing SW/HW) is the application of evolutionary and search-based optimization methods with the aim of improving functional and/or non-functional properties of existing software/hardware

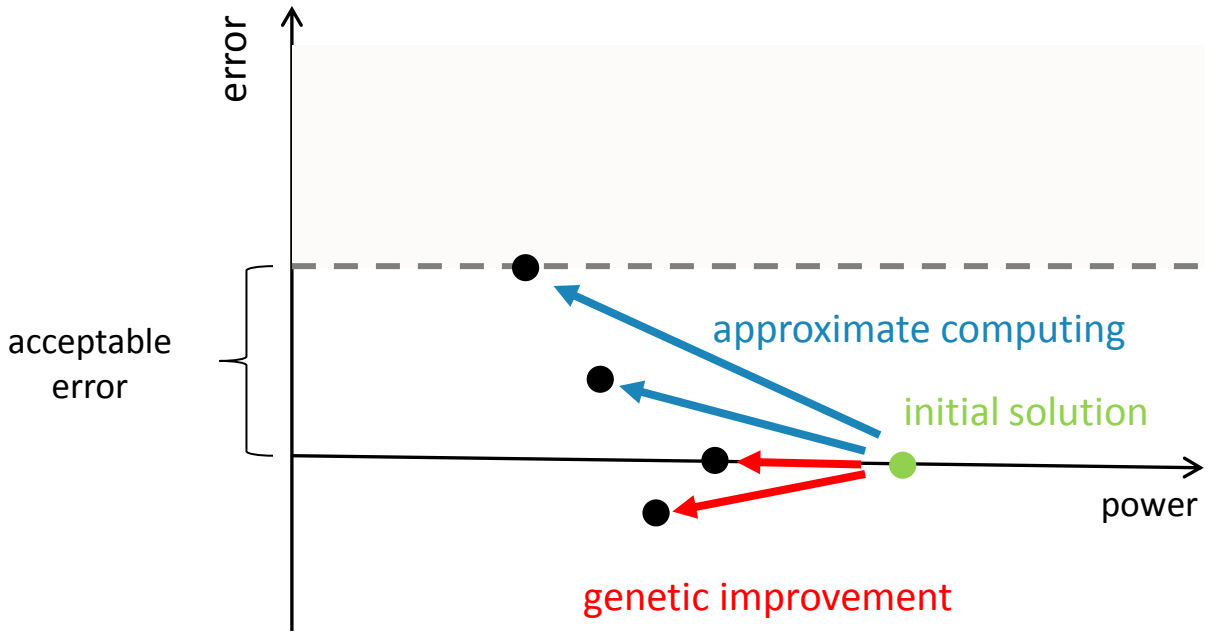
Genetic improvement of SW: Examples

- automatic bug fixing (real bugs in real C programs)
  - *W. Weimer, et al. Automatic program repair with evolutionary computation. Communications of the ACM, vol. 53, no. 5, pp. 109–116, 2010.*
- an improved version of C++ code from multiple versions of a program written by different domain experts (e.g. improved MiniSAT)
  - *J. Petke, et al. Using genetic improvement and code transplants to specialise a C++ program to a problem class. In 17th European Conference on Genetic Programming, LNCS, vol. 8599. Springer, 2014, pp. 137–149*
- improved CUDA programs (DNA analysis SW)
  - *W. Langdon. Improving CUDA DNA Analysis Software with Genetic Programming. In Genetic and Evolutionary Computation Conference (GECCO 2015): 1063-1070*
- Bowtie2, a widely-used DNA sequencing system, consisting of 50k lines of C++ code, was reduced by GI to 20k lines of code; with an average 70 times faster execution than the original code
  - *W. Langdon and M. Harman: IEEE Tr. on Evol. Computing. 19(1), 2015*



Genetic improvement vs. Approximate computing

The Genetic improvement method does not usually accept solutions increasing the error (w.r.t the original implementation)
In AC, genetic improvement can tolerate some errors.



How are EAs used in Approximate computing?

- Evolutionary optimization
  - optimization of the data type (bit width) to variable assignment - in the source code (e.g. GRATER)
  - selection of operations to be approximated – in the source code (e.g. ExpAX)
- Genetic improvement with errors enabled
  - existing solutions (SW or HW) are approximated by GP to get a suitable trade-off between error and power/performance (e.g. ABACUS, EvoApprox8b, …)
- Evolutionary design (from scratch)
  - GP is used to evolve desired approximate solutions from scratch (e.g. CGP in the multipliers approximatio, median, image filter …)

Tutorial Outline

- Approximate Computing
  - Motivation
  - Error resilience
  - Sensitivity analysis and error metrics
  - Overview of approximation techniques
- Evolutionary algorithms and genetic improvement
- EA in SW approximations
  - Extension of Java - ExpAX
  - Median
- EA in HW approximations
  - Approximations at the hardware description language level
  - Approximate multipliers in ANN
  - Library of approximate components
- Formal relaxed equivalence checking in approximate computing
  - Binary decision diagrams
  - Approximate circuits with formal error guarantees
- Conclusions

SW approximation

Software is annotated in order to introduce approximations

```
@Approx int foo (
    @Approx int x[][],
    @Approx int y[]) {
  @Approx int sum := 0;
  for i = 1 .. x.length
    for j = 1 .. y.length
      sum := sum + x[i][j] * y[j];
  return sum;}
```

(a) EnerJ [21]

```
int <0.90*R(x, y)> foo (
      int <R(x)> x[][] in urel,
      int <R(y)> y[] in urel) {
  int sum := 0 in urel;
  for i = 1 .. x.length
    for j = 1 .. y.length
      sum := sum +. x[i][j] *. y[j];
  return sum;}
```

(b) Rely [4]

```
int foo (int x[][], int y[]) {
  int sum := 0;
  for i = 1 .. x.length
    for j = 1 .. y.length
      sum := sum + x[i][j] * y[j];
  accept magnitude(sum) < 0.10;
  return sum;
}
```

(c) ExpAX

- ExpAX
  - A new programming framework that employs **error expectations.**
  - HW/CPU supporting approximate computing is assumed.
  - A static safety analysis is performed that uses the high-level expectations to automatically infer a safe-to-approximate set of program operations

SW approximation: GA in ExpAX

- Programming model with expectations (*v* is variable)
  - *accept* **rate**(*v*) < 0.2
  - *accept* **magnitude**(v) > 0.9 with **rate** < 0.3
- Finding possible safe-to-approximate variables
  - Unsafe-to-approximate variables are variables violating memory safety or functional correctness
- GA used to find a subset of safe-to-approximate operations
  - Fitness = min. (W_1.error + W_2.energy)
  - Chromosome: a bit vector representing a subset (approximate ('0') or precise('1'))
- Greedy algorithm used to refine the GA result.
- Significant energy savings (up to 35%) with large reduction in programmer effort (3x to 113x less annotations w.r.t EnerJ) while providing formal safety and statistical quality-of-result guarantees.

Approximate HW

Operation	Technique	
Integer Arithmetic/Logic	Voltage Overscaling	Timing Error Probability Energy Saving
Floating Point Arithmetic	Bit-width Reduction	Mantissa Bits Energy Saving
Double Precision Arithmetic	Bit-width Reduction	Mantissa Bits Energy Saving
SRAM Read (Reg File/Data Cache)	Voltage Overscaling	Read Upset Probability Energy Saving
SRAM Write (Reg File/Data Cache)	Voltage Overscaling	Write Failure Probability Energy Saving
DRAM (Memory)	Reduced Refresh Rate	Per-Second Bit Flip Prob Memory Power Saving

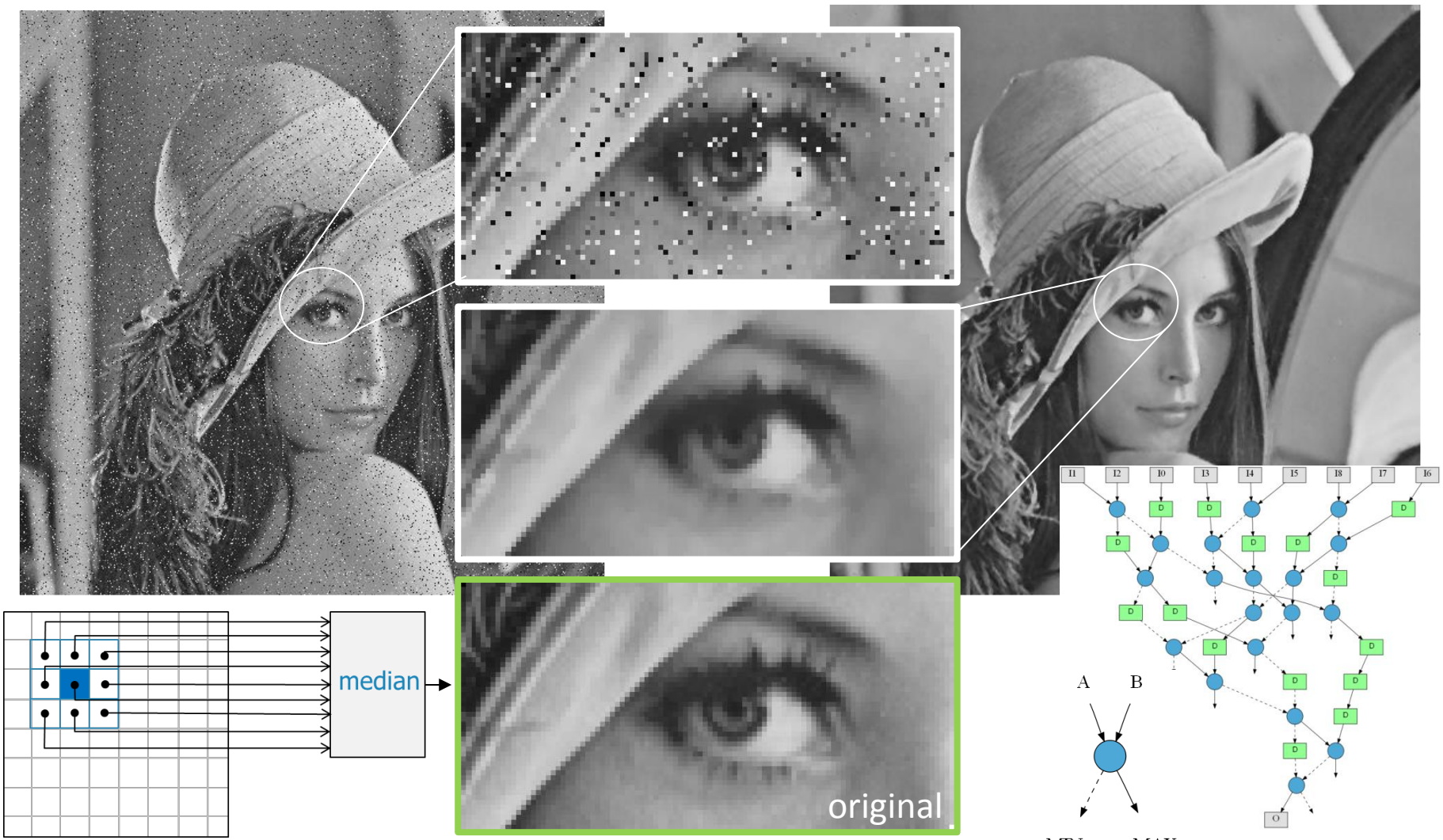
Example

```
float to_grayscale(Pixel p) {
  float luminance;
  luminance = p.r * 0.30 + p.g * 0.59 + p.b * 0.11;
  accept magnitude[luminance] > 0.9 with rate < 0.35;
  return luminance;
}
float sobel(float[][] p) {
  float x, y, gradient;
  x =  (p[0][0] + 2 * p[0][1] + p[0][2]);
  x += (p[2][0] + 2 * p[2][1] + p[2][2]);
  y =  (p[0][2] + 2 * p[1][2] + p[2][2]);
  y += (p[0][0] + 2 * p[1][1] + p[2][0]);
  gradient = sqrt(x * x + y * y);
  gradient = (gradient > 0.7070) ? 0.7070 : gradient;
  accept rate[gradient] < 0.25;
  return gradient;
}
```

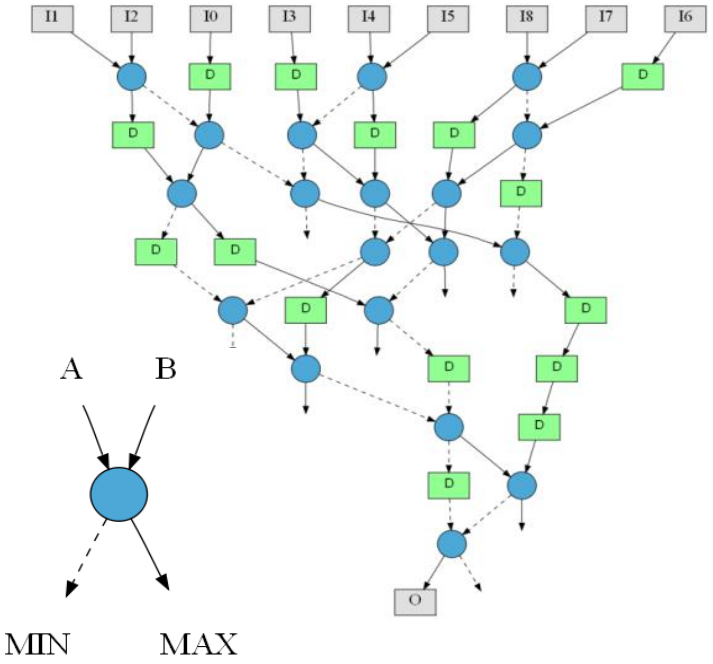
Median in image filters

corrupted image
(10% pixels, impulse noise)

filtered image
(9-input exact median filter)



9-input median function




```
#define PIX_SORT(a,b) {
 if ((a)>(b))
   PIX_SWAP((a),(b));
}
```

```
pixelvalue opt_med9 (pixelvalue * p)
{
    PIX_SORT(p[1], p[2]) ; PIX_SORT(p[4], p[5]) ; PIX_SORT(p[7], p[8]) ;
    PIX_SORT(p[0], p[1]) ; PIX_SORT(p[3], p[4]) ; PIX_SORT(p[6], p[7]) ;
    PIX_SORT(p[1], p[2]) ; PIX_SORT(p[4], p[5]) ; PIX_SORT(p[7], p[8]) ;
    PIX_SORT(p[0], p[3]) ; PIX_SORT(p[5], p[8]) ; PIX_SORT(p[4], p[7]) ;
    PIX_SORT(p[3], p[6]) ; PIX_SORT(p[1], p[4]) ; PIX_SORT(p[2], p[5]) ;
    PIX_SORT(p[4], p[7]) ; PIX_SORT(p[4], p[2]) ; PIX_SORT(p[6], p[4]) ;
    PIX_SORT(p[4], p[2]) ; return(p[4]) ;
}
```

Source: http://ndevilla.free.fr/median/median.pdf

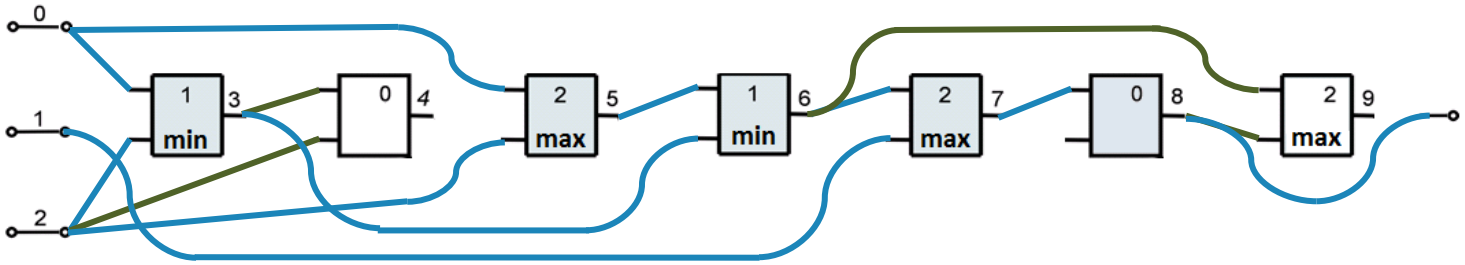
Approximate median using CGP

Cartesian Genetic Programming (CGP) for median approximation

- Median network (consisting of up to N operations) is represented by means of an one-dimensional array of N nodes.
- Each node can act as: identity (0), minimum (1), maximum (2)
- Each candidate solution is encoded using $3N + 1$ integers.
- Fitness function (single objective)

$$error = \sum_{i \in S} \left| O_{candidate}(i) - O_{reference}(i) \right|$$

- Example for 3-input median:



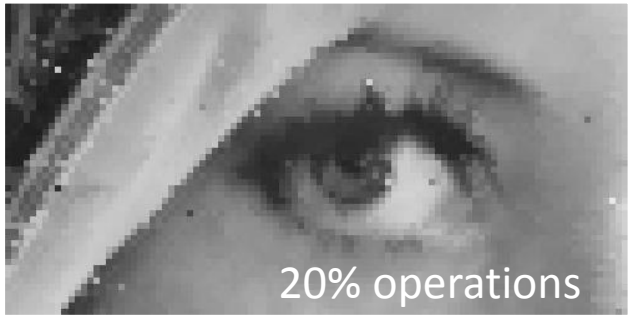
Chromosome: 0, 2, 3; 3, 2, 0; 0, 2, 2; 5, 3, 1; 6, 1, 2; 7, 0, 0; 6, 8, 2; 8

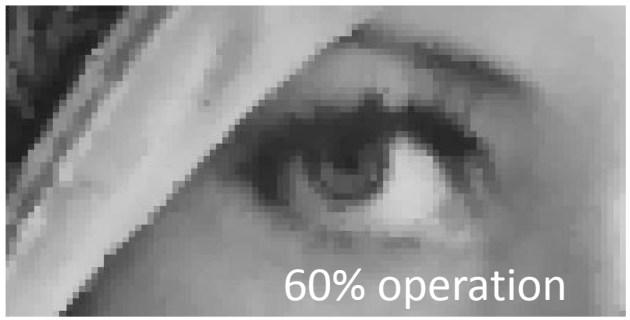
Approximate median using CGP

Experimental setup

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

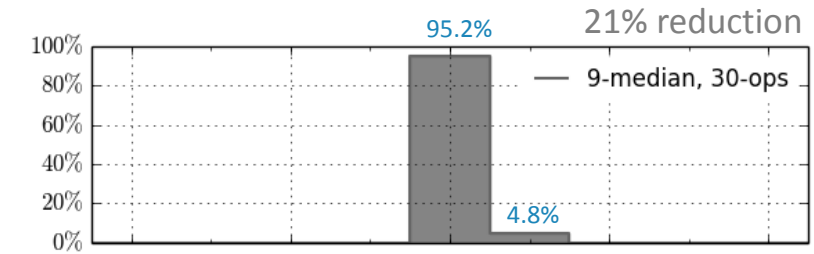
	Median-9	Median-25
Inputs	9	25
Outputs	1	1
Generations	3×10^6 (3 hours)	3×10^5 (3 hours)
Training vectors	1×10^4	1×10^5
Reference (exact) solution	38 operations	220 operations
Number of nodes	6 – 34 operations	10 – 200 operations

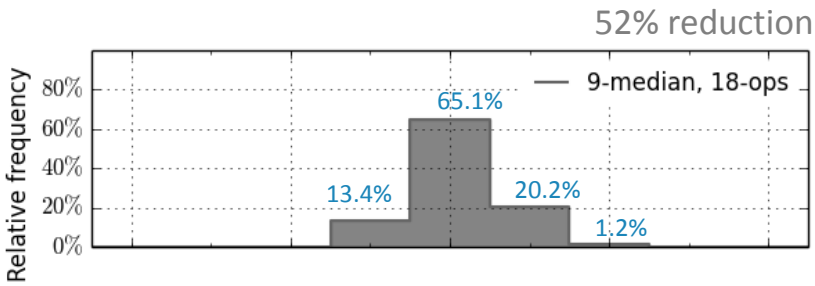


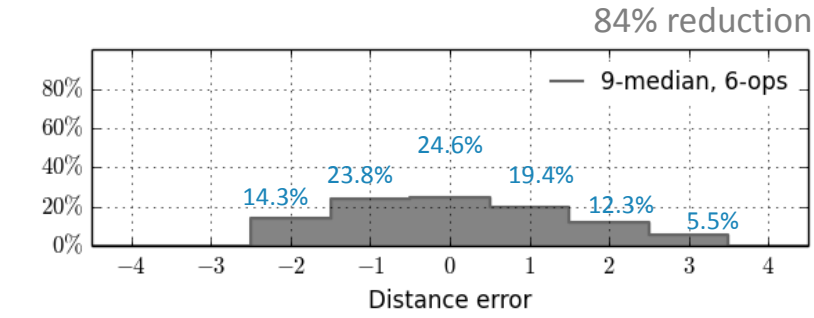


Approximate median: Quality

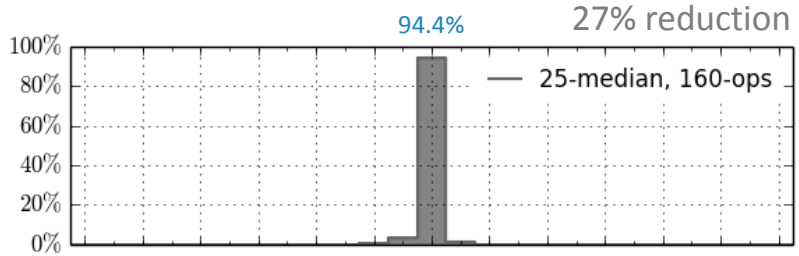
9-input median
fully-working: 38 operations

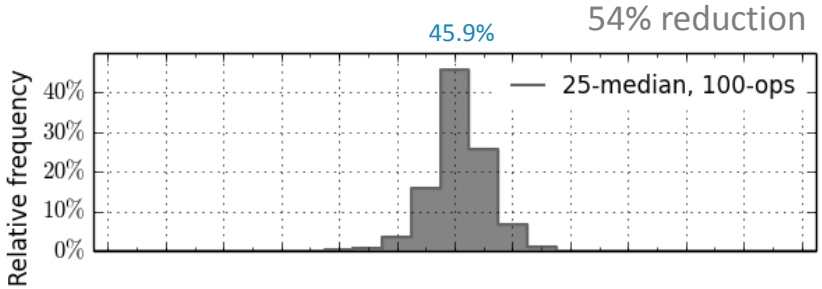


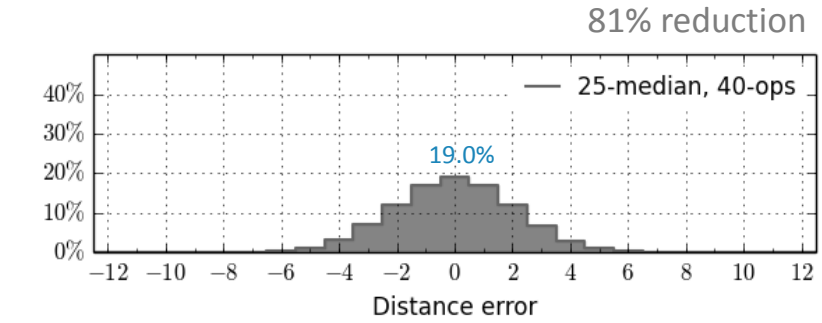




25-input median
fully-working: 220 operations







Approx. 9-median as SW for microcontrollers

Impl.	Time [μs]			Energy [nWs]		
	STM32	PIC24	PIC16	STM32	PIC24	PIC16
6-ops	2.8	54.5	170.5	86	377	342
10-ops	3.3	70.8	251.5	102	490	504
14-ops	3.9	86.8	336.5	118	600	674
18-ops	4.5	104.5	424.1	138	723	850
22-ops	5.0	116.7	487.8	151	808	978
26-ops	5.9	130.0	558.0	179	900	1118
30-ops	6.0	142.0	627.4	181	983	1257
34-ops	6.4	154.0	819.7	196	1066	1643
38-ops	6.9	165.5	885.0	210	1145	1774
qsort	28.5	1106.2	—	869	7655	—

34.9% error prob.,
max. error dist. 2
52% power reduction

4.8% error prob.,
max. error dist. 1
21% power reduction

fully-working median

ops = operations in the source code.

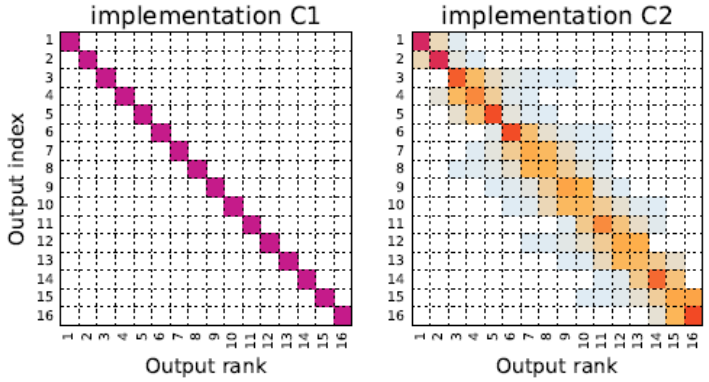
```
#define PIX_SORT(a,b) {
 if ((a)>(b))
   PIX_SWAP((a),(b));
}
```

V. Mrazek, Z. Vasicek and L. Sekanina. GECCO GI Workshop, 2015

Energy-efficient implementation of sorting networks

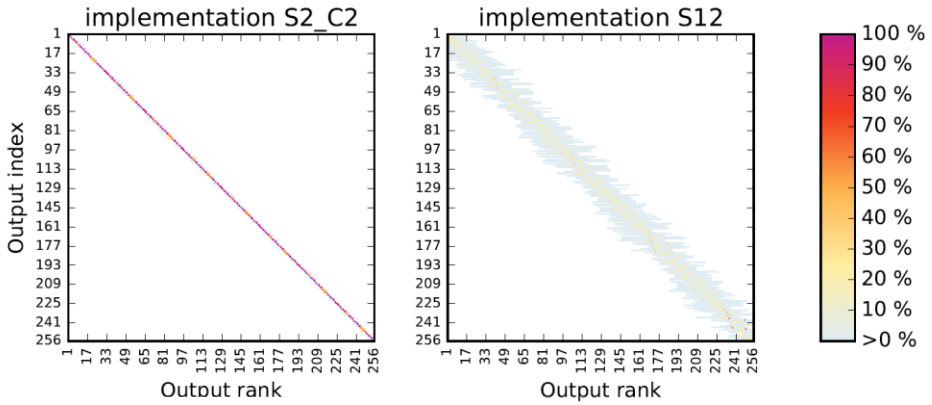
- Key idea: reduce the number of compare-and-swap operation in sorting networks to improve energy-efficiency
- To model the error introduced by the approximations in sorting networks, the distance between the rank of the returned element and rank given by the specification is measured.

16-input sorting network



power reduction: 9%
error: <2 for 99% cases

256-input sorting network



power reduction: 52%
error: <14 for 99% cases

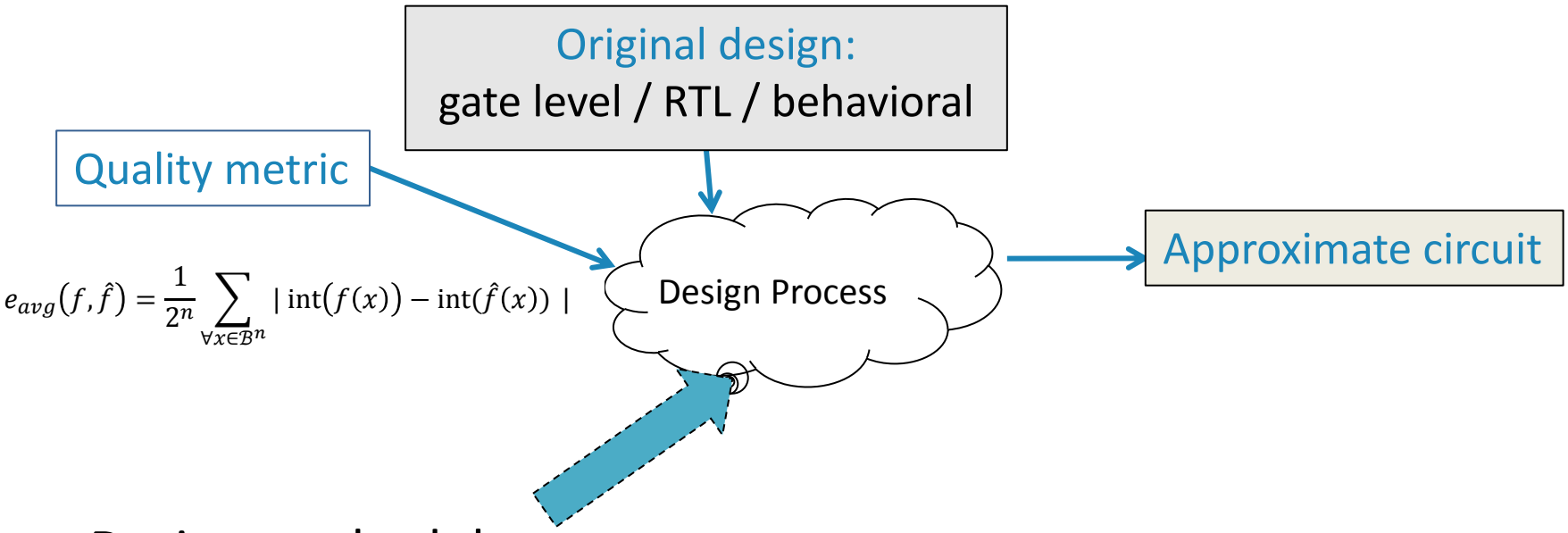
MRAZEK V., VASICEK Z.: Automatic Design of Arbitrary-Size Approximate Sorting Networks with Error Guarantee. In: PATMOS 2016, pp. 221-228

VASICEK Z., MRAZEK V.: Trading between Quality and Non-functional Properties of Median Filter in Embedded Systems. Genetic Programming and Evolvable Machines, 2017, 18:45–82

Tutorial Outline

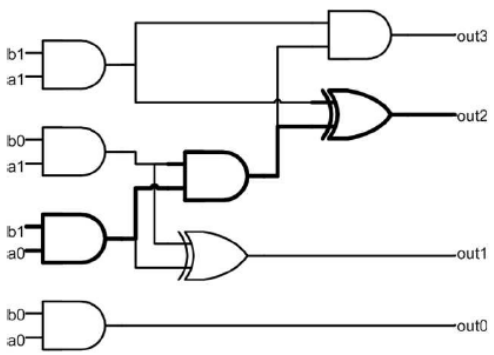
- Approximate Computing
  - Motivation
  - Error resilience
  - Sensitivity analysis and error metrics
  - Overview of approximation techniques
- Evolutionary algorithms and genetic improvement
- EA in SW approximations
  - Extension of Java - ExpAX
  - Median
- EA in HW approximations
  - Approximations at the hardware description language level
  - Approximate multipliers in ANN
  - Library of approximate components
- Formal relaxed equivalence checking in approximate computing
  - Binary decision diagrams
  - Approximate circuits with formal error guarantees
- Conclusions

Functional circuit approximation



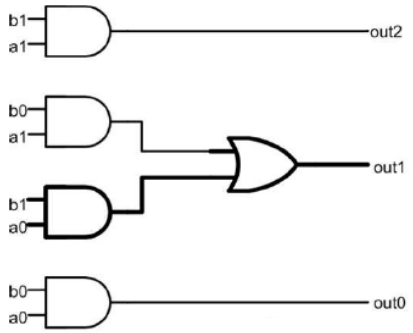
- Design methodology
  - Manual [Kulkarni et al.: J. Low Power Electronic 2011]
  - Automatic (= some heuristics used)
    - Heuristic algorithms: SALSA (DAC 2012), SASIMI (DATE 2013), ABACUS (DATE 2014), ASLAN (DATE 2014), AIG-REV (ICCAD 2016) …
    - GP-based methods: CGP (ICES 2013, DDECS 2014, EuroGP 2015, IEEE Tr. on EC 2015, FPL 2016, GENP 2016), GP (ABACUS with NSGA-II 2017)

Manual approximation: Multipliers



accurate

MSB removed!



approximated

AxB	0	1	2	3
0	0	0	0	0
1	0	1	2	3
2	0	2	4	6
3	0	3	6	7

- Correct results for 15 out of 16 input combinations (almost 50% area reduction, lower delay).
- Used as a building block for larger multipliers and then in image processing applications.

Error probability

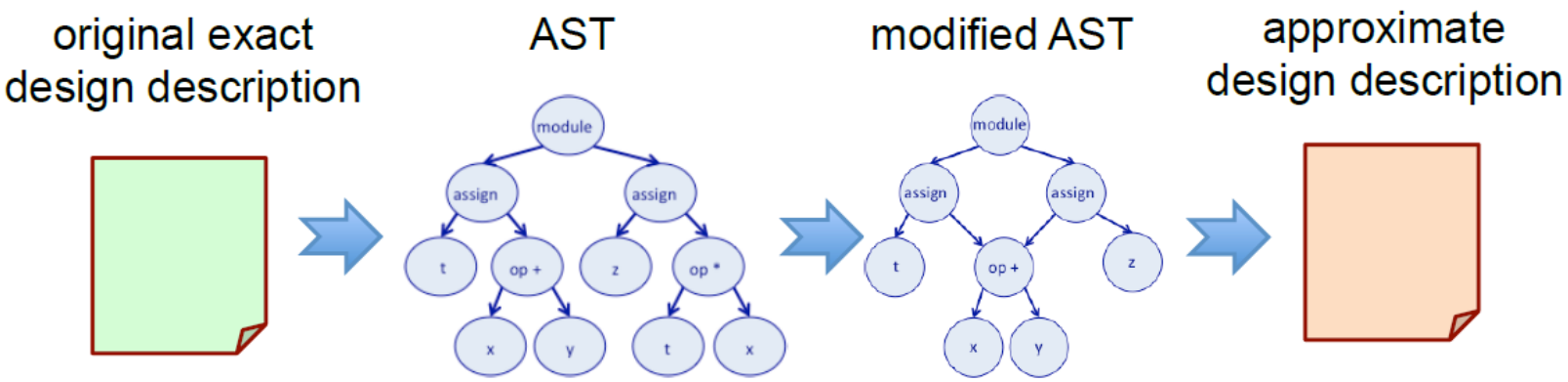
Bit-Width	Error-Prob	Mean-Error	Max-Error
2	0.0625	1.39%	22.22%
4	0.19	2.60%	22.22%
8	0.46	3.25%	22.22%
12	0.675	3.31%	22.22%
16	0.81	3.32%	22.22%

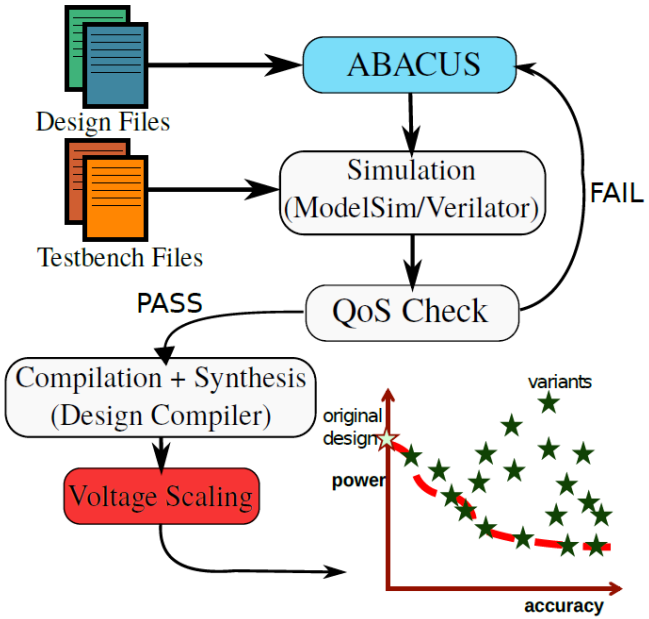
Dynamic power reduction for various frequencies

Bit	F (%)	1.25F (%)	1.5F (%)	1.75F (%)	2F (%)	Avg. (%)
2	44.9	42.1	42.1	48.9	48.9	45.4
4	13.7	31.6	44.8	44.7	46.5	36.3
8	33.1	40.4	26.3	48.8	58.9	41.5
16	25.6	29.6	32.4	33.8	37.4	31.8

Kulkarni et al. Trading Accuracy for Power in a Multiplier Architecture, VLSI Design, 2011

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: NSGA-II – based
- Fitness is obtained by circuit simulation and combines the error & power

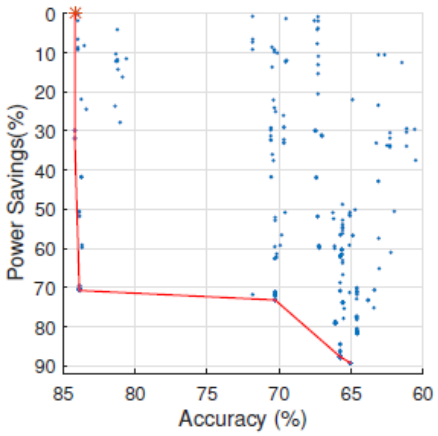
Nepal K. et al.: Automated High-Level Generation of Low-Power Approximate Computing Circuits, Trans. on Emerging Topics in Computing, 2017

ABACUS: Results

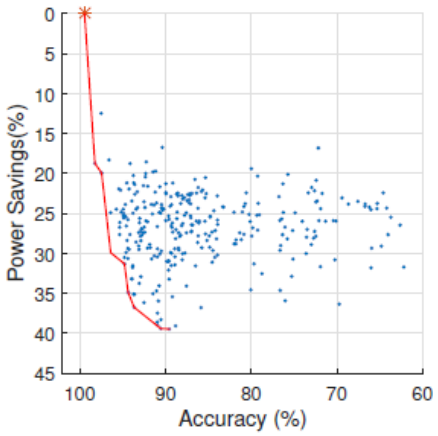
Benchmark problems:

Design	Class of Application	#Lines	Area (um^2)	Power (mW)	Quality Measure	Quality
perceptron	Machine Learning	188	37775.16	2.74	classification error	82.9%
FIR filter	Signal Processing	265	40390.20	6.89	MSE	99.45%
FFT	Signal Processing	255	18480.96	2.07	MSE	100%
block matching	Computer Vision	1277	80272.44	30.42	PSNR	30.66 dB

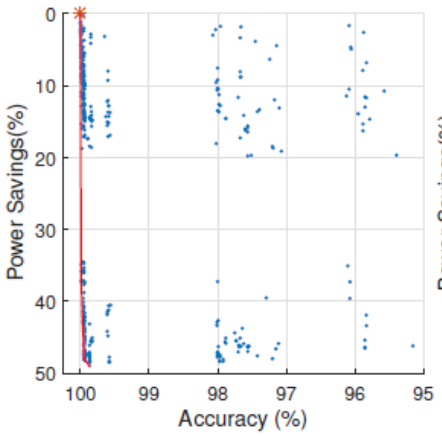
Results of evolutionary approximation:



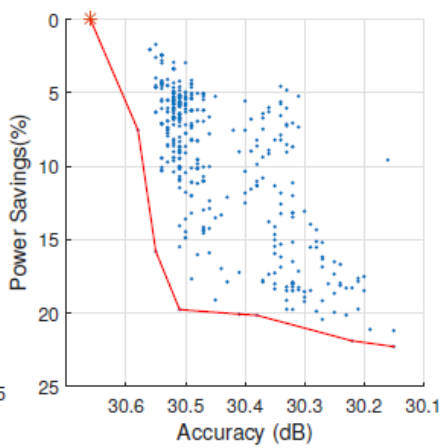
(a) Perceptron



(b) FIR Filter



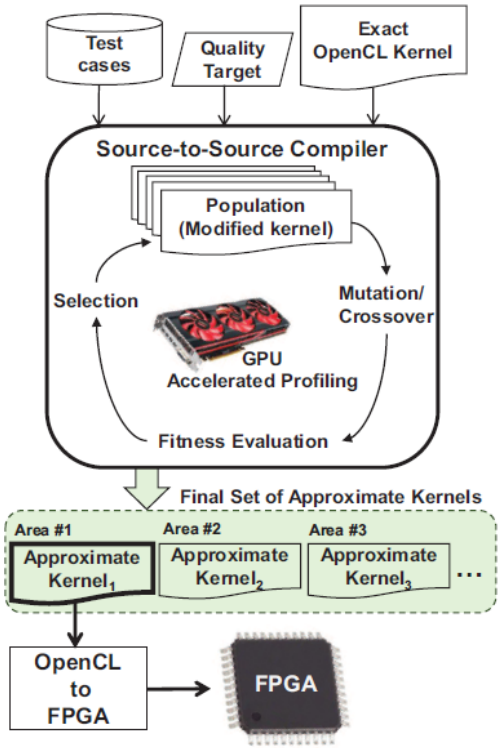
(c) FFT

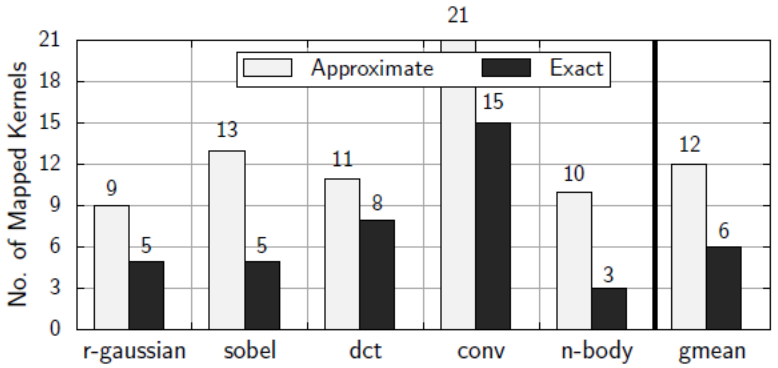


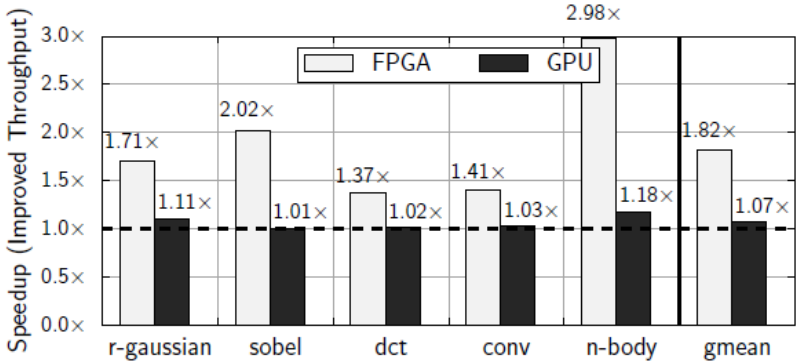
(d) Block Matching

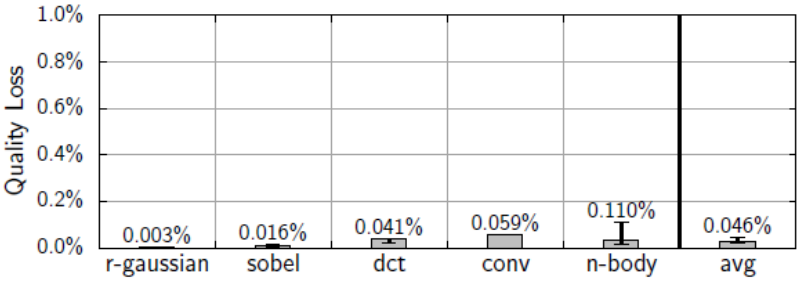
GRATER: GA-based optimization of data types

- Sensitivity analysis performed to find safe-to-approximate variables (AV) in OpenCL kernel.
- Chromosome: *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.



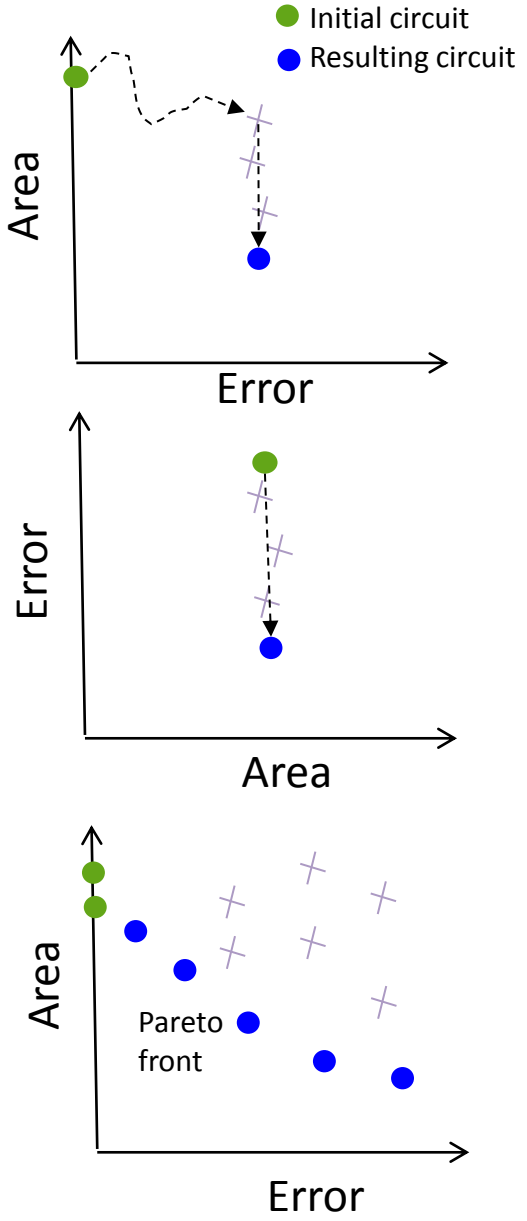




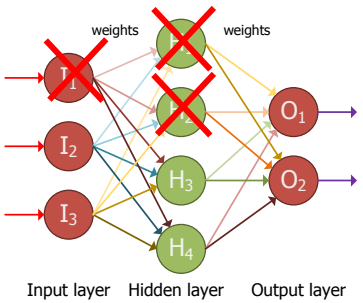


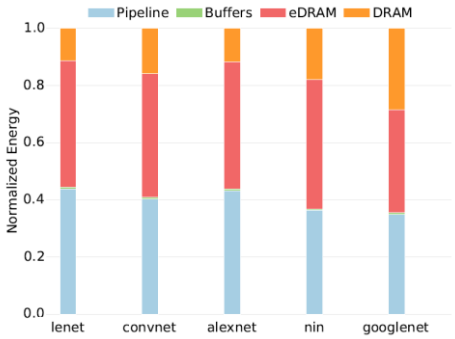
CGP for circuit (functional) approximation

- 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.

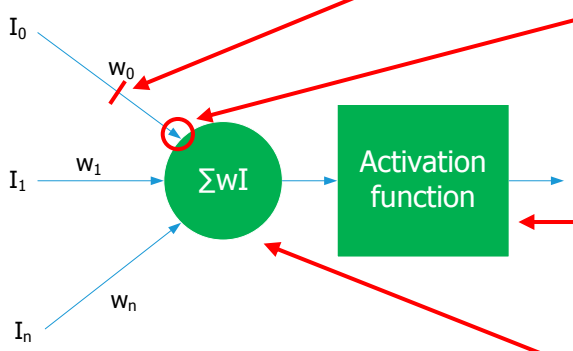


Energy-efficient implementation of ANNs





[Judd et.al. WAPCO'16]



Approximations introduced in:

- ANN structure – removing nodes of NN
  [Venkataramani et al. ISLPED'14]
- Learning algorithm
- Memory – approximate Load/Store
  [Srinivasan et al. DATE'16]

- Pipeline
  - Reducing data bit-width
    [Judd et al. ICS'16]
  - Multiplication (approx. 45% of total power)
    Multiplierless multiplication
    [Sarwar et al. DATE'16]
  - Activation function
    [Du et al. ASP-DAC'14]
  - Sum function

Energy-efficient implementation of ANNs

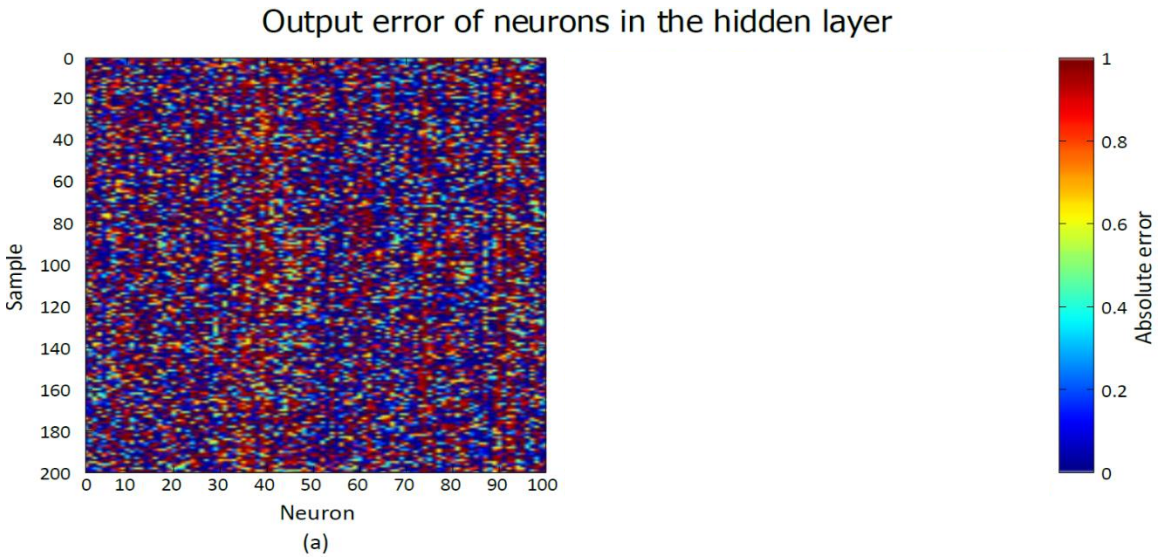
MNIST dataset classification: 32x32 – 100 – 10 MLP network (classification accuracy 94.16% with accurate implementation). We introduced an approximate multiplier by adding a jitter function $\Delta(a, b)$, resulting in a 5.2% error for multiplication.

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
  $$m'(a, b) = \begin{cases} 0 & if\ a = 0 \vee b = 0 \\ a \cdot b + \Delta(a, b) & otherwise \end{cases}$$
- Classification accuracy : 94.20%



Mrazek, Sarwar, Sekanina, Vasicek, Roy: "Design of power-efficient approximate multipliers for approximate artificial neural networks," ICCAD 2016

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. $\forall a, b: |m(a,b) - a * b| \leq \varepsilon \cdot 2^{n_o}$
  2. $\forall a: m(a,0) = m(0,a) = 0$
- Fitness function:

$$C(m) = \begin{cases} -GatesCount(m) & if\ constraints\ (1)\ and\ (2)\ met, \\ -\infty & otherwise \end{cases}$$

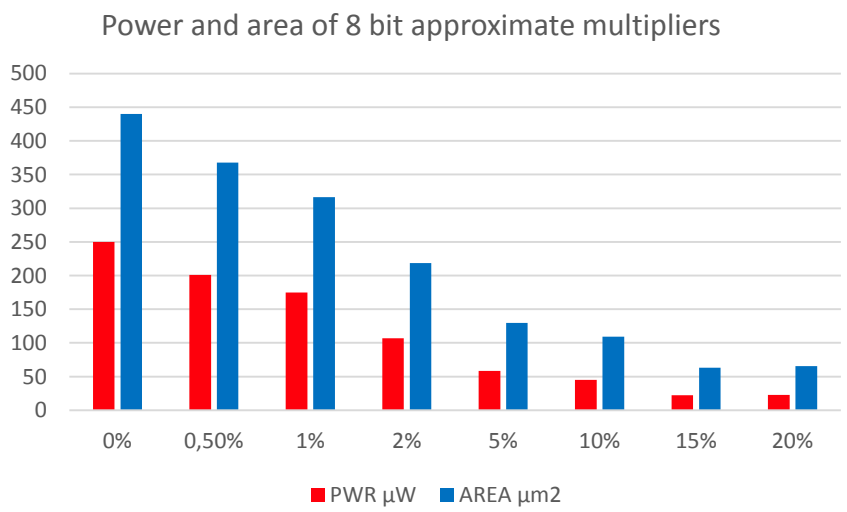
Mrazek, Sarwar, Sekanina, Vasicek, Roy: "Design of power-efficient approximate multipliers for approximate artificial neural networks," ICCAD 2016

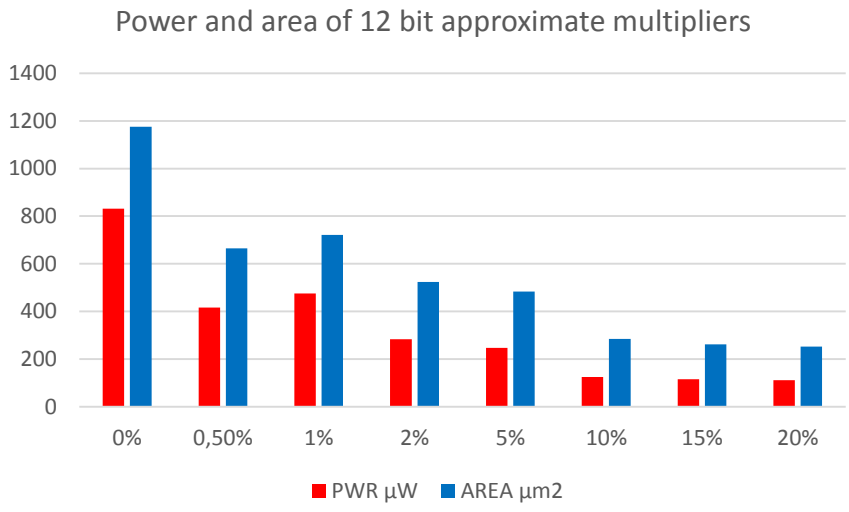
CGP in approx. multiplier design for ANNs

- 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 backpropagation algorithm.
- Approximate multipliers showing the best trade off between power and accuracy in NN were selected (for different error targets).

Mrazek, Sarwar, Sekanina, Vasicek, Roy: "Design of power-efficient approximate multipliers for approximate artificial neural networks," ICCAD 2016

Evolved approximate multipliers for ANNs





Results of synthesis of sign-extended multipliers with Synopsys DC

45 nm technology

Timing:

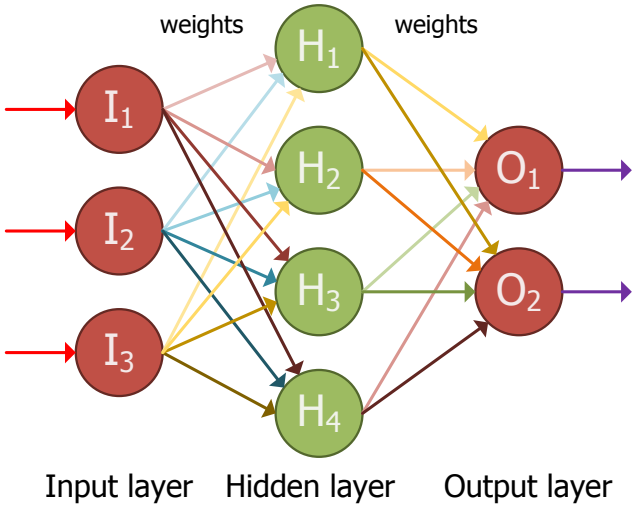
- 8-bit multipliers: 2.5 GHz
- 12-bit multipliers: 2 GHz

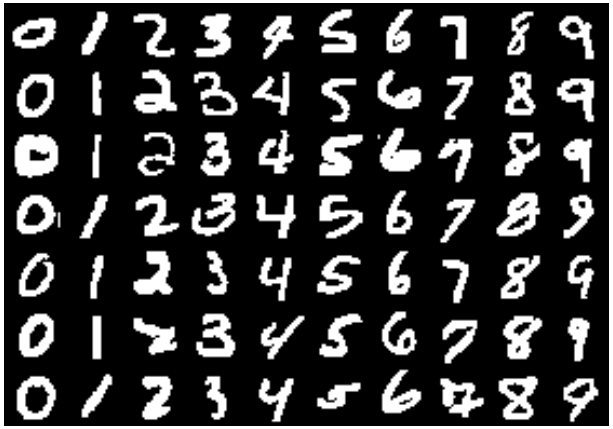
Accurate multiplier was implemented in Verilog using standard * arithmetic operator

Mrazek, Sarwar, Sekanina, Vasicek, Roy: "Design of power-efficient approximate multipliers for approximate artificial neural networks," ICCAD 2016

Energy-efficient implementation of ANNs: MLP

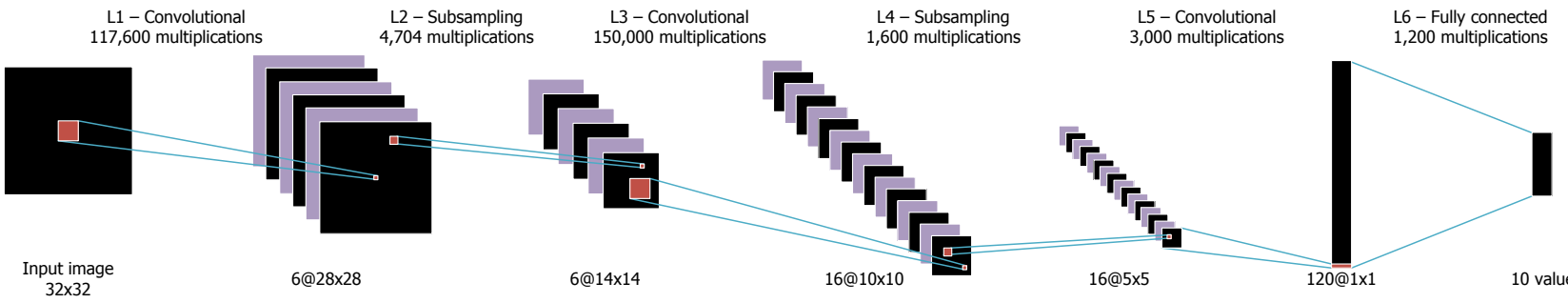
- Handwritten number dataset (dataset used for benchmarking)
- Fully connected MLP network
- 28x28 inputs, 300 hidden neurons, 10 outputs
- 60k training images
- 10k testing images
- More than 238k multiplications for approximation
- Initial classification accuracy:
  - 8b: 97.67%
  - 12b: 97.70%



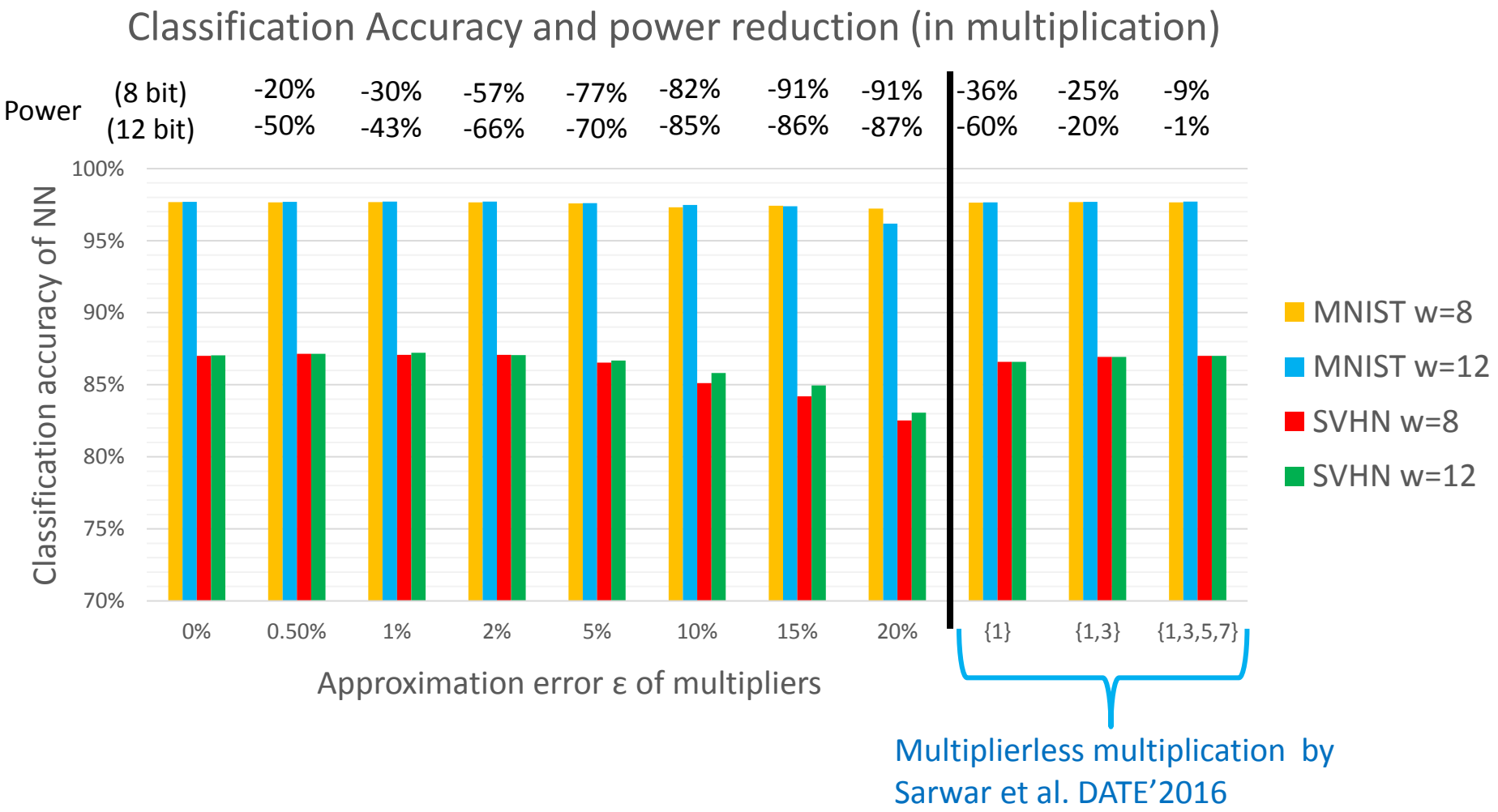


Energy-efficient implementation of ANNs: LeNet

- 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%



Energy-efficient implementation of ANNs: Summary



Mrazek, Sarwar, Sekanina, Vasicek, Roy: "Design of power-efficient approximate multipliers for approximate artificial neural networks," ICCAD 2016

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

$$f_{\mathrm{mre}} := \frac{\sum_{\forall i} \frac{\left|O^{(i)}_{\mathrm{orig}} - O^{(i)}_{\mathrm{approx}}\right|}{\max(1, O^{(i)}_{\mathrm{orig}})}}{2^{N_\mathrm{i}}}$$

$O^{(i)}$ is the i-th circuit output
$i = 1 \ldots 2^{Ni}$

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 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

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

13 conventional 8-bit adders

TA = Tree Adder

HVTA = Higher Valency Tree Adder

Architecture	Power	Area	Delay
Ripple-Carry Array	**100.00%**	**100.00%**	**100.00%**
Carry-Save Array using RCA	102.30%	100.00%	71.16%
Carry-Save Array using CSA	108.42%	106.16%	62.03%
Wallace Tree using RCA	104.29%	107.39%	68.91%
Wallace Tree using CLA	116.10%	148.48%	51.26%
Wallace Tree using CSA	120.12%	122.35%	53.28%

6 conventional 8-bit multipliers

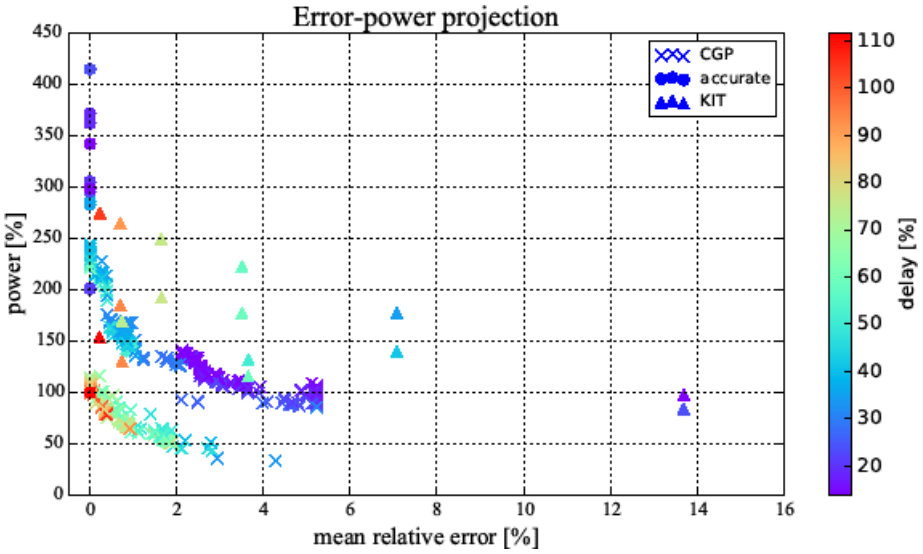
RCA = Ripple-Carry Adder

CSA = Carry-Save Adder

CLA = Carry-Lookahead Adder

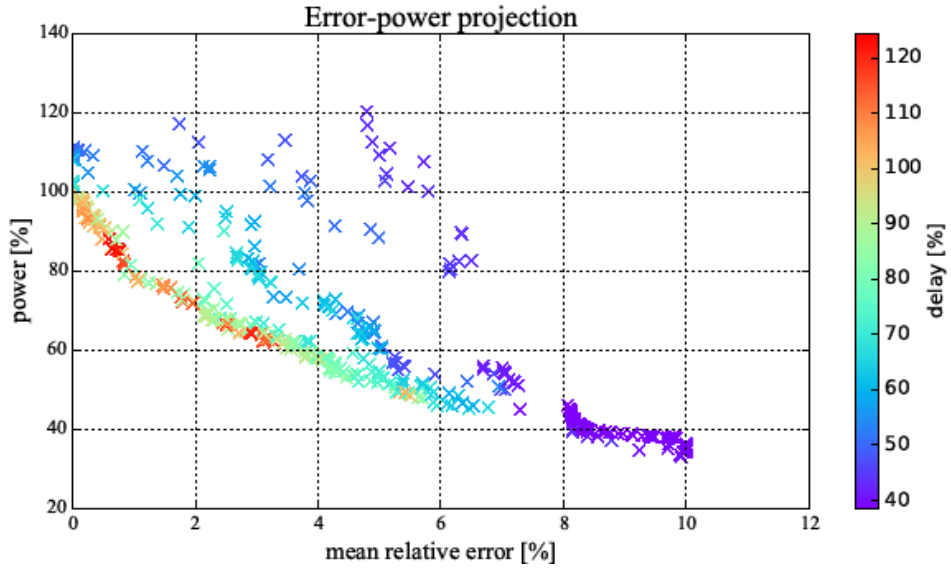
Library of 8-bit approx. adders and multipliers

- Large 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)

V. Mrazek, R. Hrbacek, Z. Vasicek, L. Sekanina: EvoApprox8b: Library, DATE 2017, p. 1-4

KIT: M. Shafique, W. Ahmad, R. Hafiz, and J. Henkel: A low latency generic accuracy configurable adder, DAC 2015, pp. 86:1–86:6.

Library of 8-bit approx. adders and multipliers

Approximate adders (430), exact adders (43)

Circuit	Est. area	Est. delay	Est. power	Nodes	HD	MAE	MSE	MRE	WCE	WCRE	EP	OPS
add8_000	820 μm^2	1.314 ns	194.31 μW	10	138496	1.71875	6.00000	0.88 %	7	100 %	71.875 %	Verilog C Matlab
add8_001	2040 μm^2	0.718 ns	681.20 μW	42	0	0.00000	0.00000	0.00 %	0	0 %	0.000 %	Verilog C Matlab
add8_002	836 μm^2	1.282 ns	194.75 μW	13	140448	1.69531	5.85938	0.88 %	7	100 %	71.484 %	Verilog C Matlab
add8_003	912 μm^2	0.379 ns	266.66 μW	20	192640	9.64844	138.25000	5.21 %	24	100 %	96.875 %	Verilog C Matlab
add8_004	708 μm^2	1.213 ns	205.54 μW	9	134528	1.37500	3.25000	0.75 %	5	200 %	76.562 %	Verilog C Matlab

.....

Approximate multipliers (471), exact multipliers (28)

Circuit	Est. area	Est. delay	Est. power	Nodes	HD	MAE	MSE	MRE	WCE	WCRE	EP	OPS
mul8_000	9224 μm^2	3.015 ns	4933.22 μW	137	176134	98.52710	27520.00000	1.99 %	820	560 %	86.490 %	Verilog C Matlab
mul8_001	5200 μm^2	3.566 ns	2524.84 μW	91	310752	239.95550	108908.84375	5.36 %	1671	100 %	98.169 %	Verilog C Matlab
mul8_002	6715 μm^2	2.086 ns	2789.47 μW	132	339806	329.88147	207883.35278	6.70 %	2193	700 %	98.482 %	Verilog C Matlab
mul8_003	4172 μm^2	1.963 ns	1816.06 μW	79	376002	624.46875	679898.57422	10.00 %	2911	700 %	98.984 %	Verilog C Matlab
mul8_004	5034 μm^2	1.944 ns	1893.73 μW	104	382402	639.22653	709554.15625	9.76 %	3143	253 %	99.071 %	Verilog C Matlab

.....

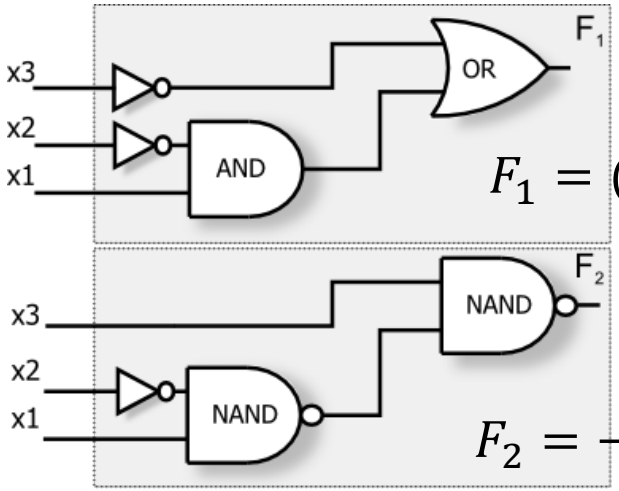
Synthesis results for 45 nm and 180 nm technology (Cadence Encounter RTL Compiler), 7 error metrics

http://www.fit.vutbr.cz/research/groups/ehw/approxlib/

Tutorial Outline

- Approximate Computing
  - Motivation
  - Error resilience
  - Sensitivity analysis and error metrics
  - Overview of approximation techniques
- Evolutionary algorithms and genetic improvement
- EA in SW approximations
  - Extension of Java - ExpAX
  - Median
- EA in HW approximations
  - Approximations at the hardware description language level
  - Approximate multipliers in ANN
  - Library of approximate components
- Formal relaxed equivalence checking in approximate computing
  - Binary decision diagrams
  - Approximate circuits with formal error guarantees
- Conclusions

Are F_1 and F_2 functionally equivalent?



$$F_1 = (x_1 \wedge \neg x_2) \vee \neg x_3$$

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

x3	x2	x1	F1	F2
0	0	0	**1**	**1**
0	0	1	**1**	**1**
0	1	0	**1**	**1**
0	1	1	**1**	**1**
1	0	0	**0**	**0**
1	0	1	**1**	**1**
1	1	0	**0**	**0**
1	1	1	**0**	**0**

- 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!

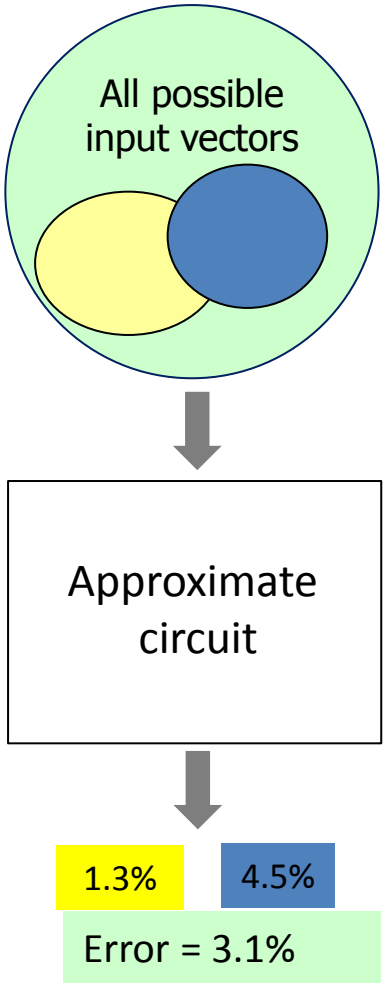
How to determine the error?

Error “estimation”

- Simulation
- Probabilistic models, e.g. Li at al., DAC 2015

Exact error calculation

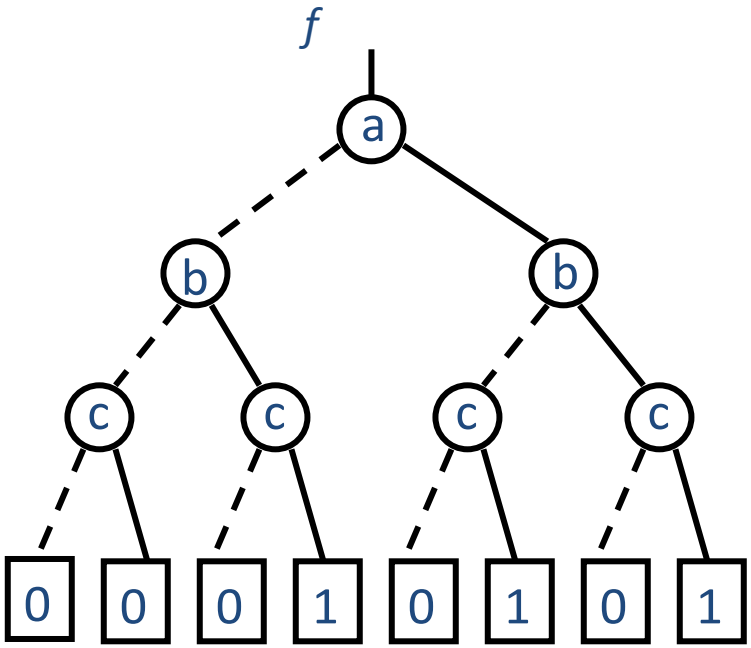
- Exhaustive Simulation – small problem instances only
- Analysis of Binary decision diagrams
  - Average error, worst case, error rate …
    - M. Soeken, D. Grosse, A. Chandrasekharan, and R. Drechsler: BDD minimization for approximate computing, ASP-DAC 2016
  - Average Hamming distance:
    - Z. Vasicek and L. Sekanina: Circuit approximation using single- and multi-objective cartesian GP. Gen. Prog. Evol. Mach., 17(2), 2016
  - Not scalable for some problems such as multipliers
- Transforming to SAT problem
  - Worst case error
    - S. Venkataramani et al. : SALSA: systematic logic synthesis of approximate circuits, DAC 2012



$f = ac + bc$

a	b	c	f
0	0	0	0
0	0	1	0
0	1	0	0
0	1	1	1
1	0	0	0
1	0	1	1
1	1	0	0
1	1	1	1

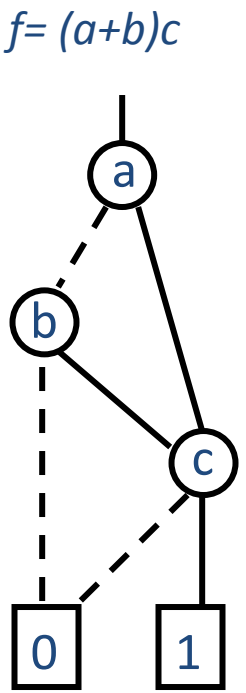
Truth table



Decision tree

1 edge

0 edge



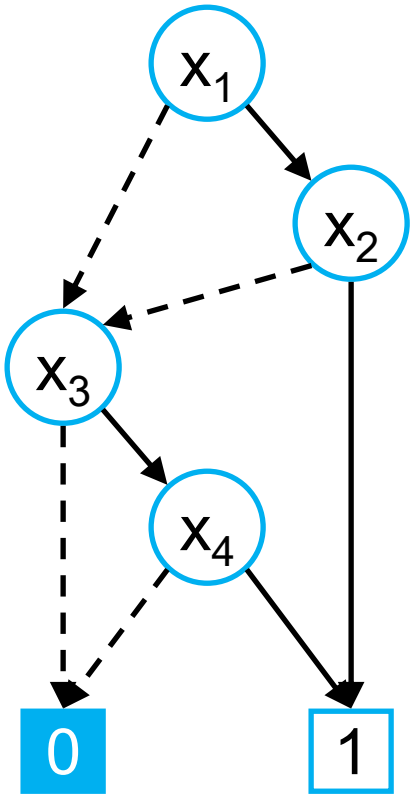
Reduced Ordered BDD (ROBDD)

(canonical form)

Operations over (RO)BDDs implemented by many libraries, e.g. Buddy.

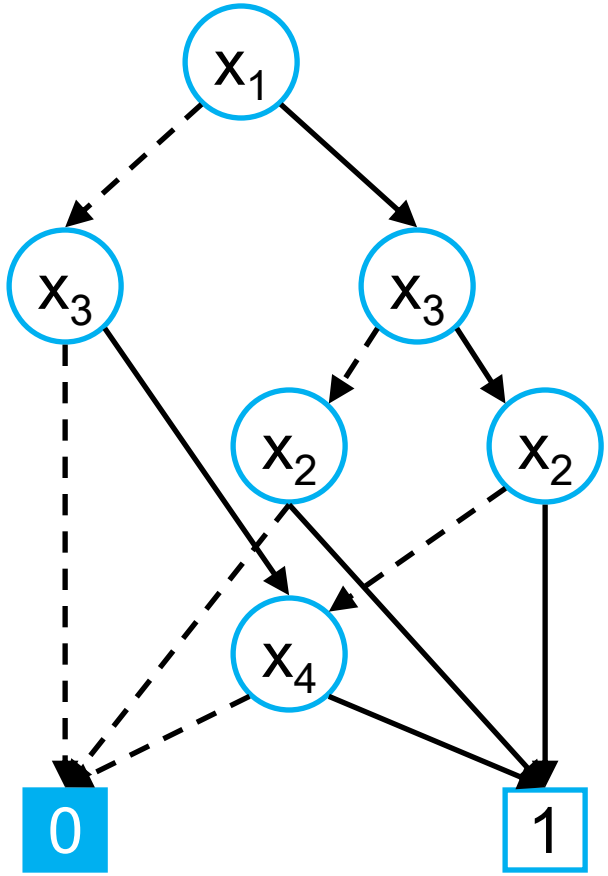
Pitfalls of Binary Decision Diagrams

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



$x_1x_2 + x_3x_4$

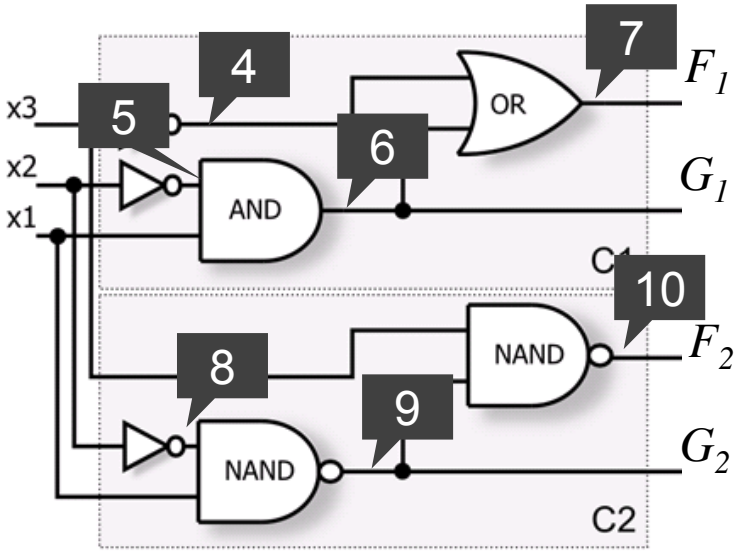
$x_1 < x_2 < x_3 < x_4$
(optimal)



$x_1 < x_3 < x_2 < x_4$

Equivalence checking using ROBDDs

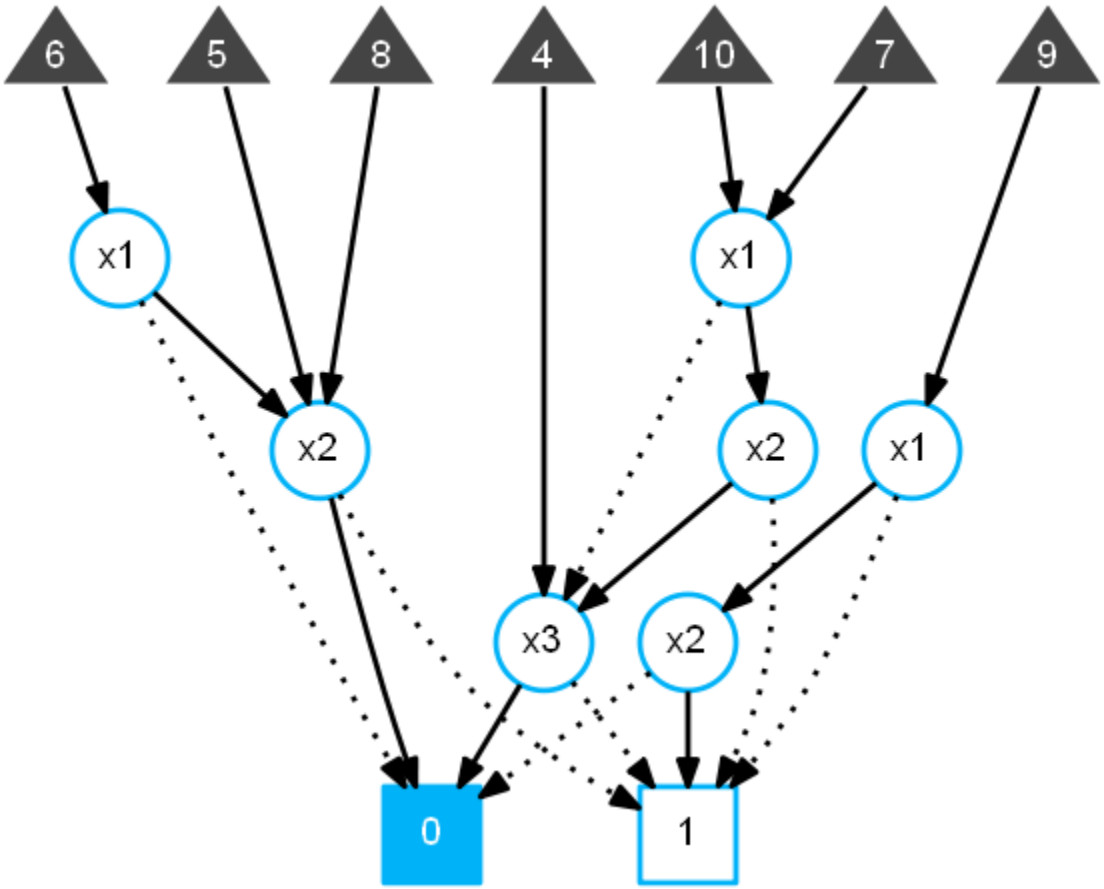
Are circuits C1 and C2 functionally equivalent?



The decision procedure is trivial and reduces to pointer comparison.

ROBDD construction:

Apply (*op*, *a*, *b*) – creates ROBDD representing logic function *op* over two ROBDDs *a* and *b*



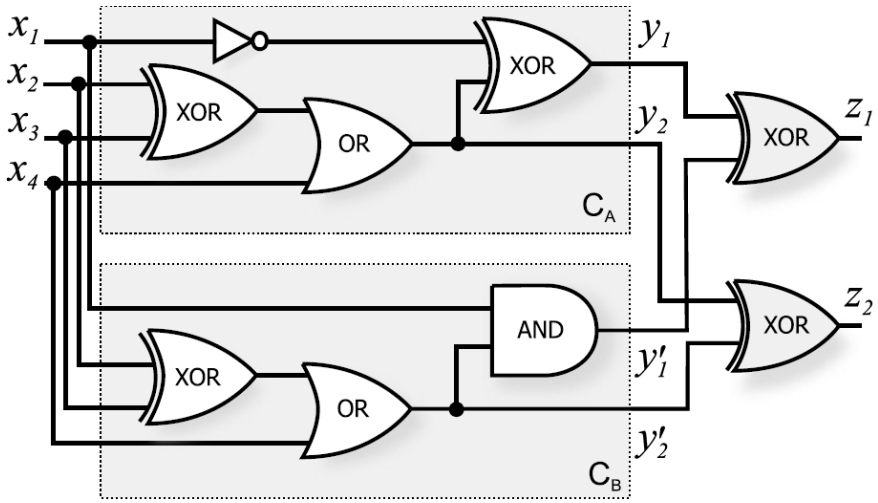
Other operations on ROBDDs

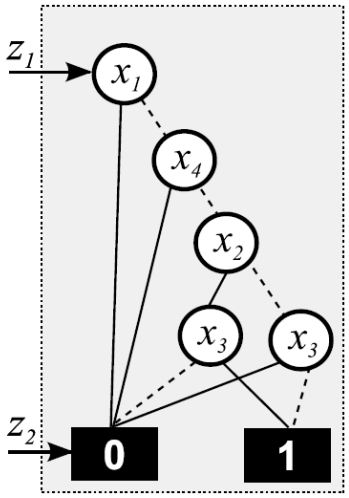
- Many logic operations can be performed efficiently on BDDs
  - usually linear in size of result
  - tautology and complement are constant time

Procedure	**Result**	**Time Complexity**
Reduce	G reduced to canonical form	$O(\|G\| \cdot \log\|G\|)$
Apply	f_1 <op> f_2	$O(\|G_1\| \cdot \|G_2\|)$
Restrict	$f\|_{x_i=b}$	$O(\|G\| \cdot \log\|G\|)$
Compose	$f_1\|_{x_i=f_2}$	$O(\|G_1\|^2 \cdot \|G_2\|)$
Satisfy-one	some element of S_f	$O(n)$
Satisfy-all	S_f	$O(n \cdot \|S_f\|)$
Satisfy-count	$\|S_f\|$	$O(\|G\|)$

Bryant R.: Graph-Based Algorithms for Boolean Function Manipulation. IEEE Trans. on Comp. 1986

Hamming distance using BDDs





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

$$SatCount(z_1) = 2$$
$$SatCount(z_2) = 0$$

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

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

x_1	x_2	x_3	x_4	# combinations
0	0	0	0	1
0	1	1	0	1

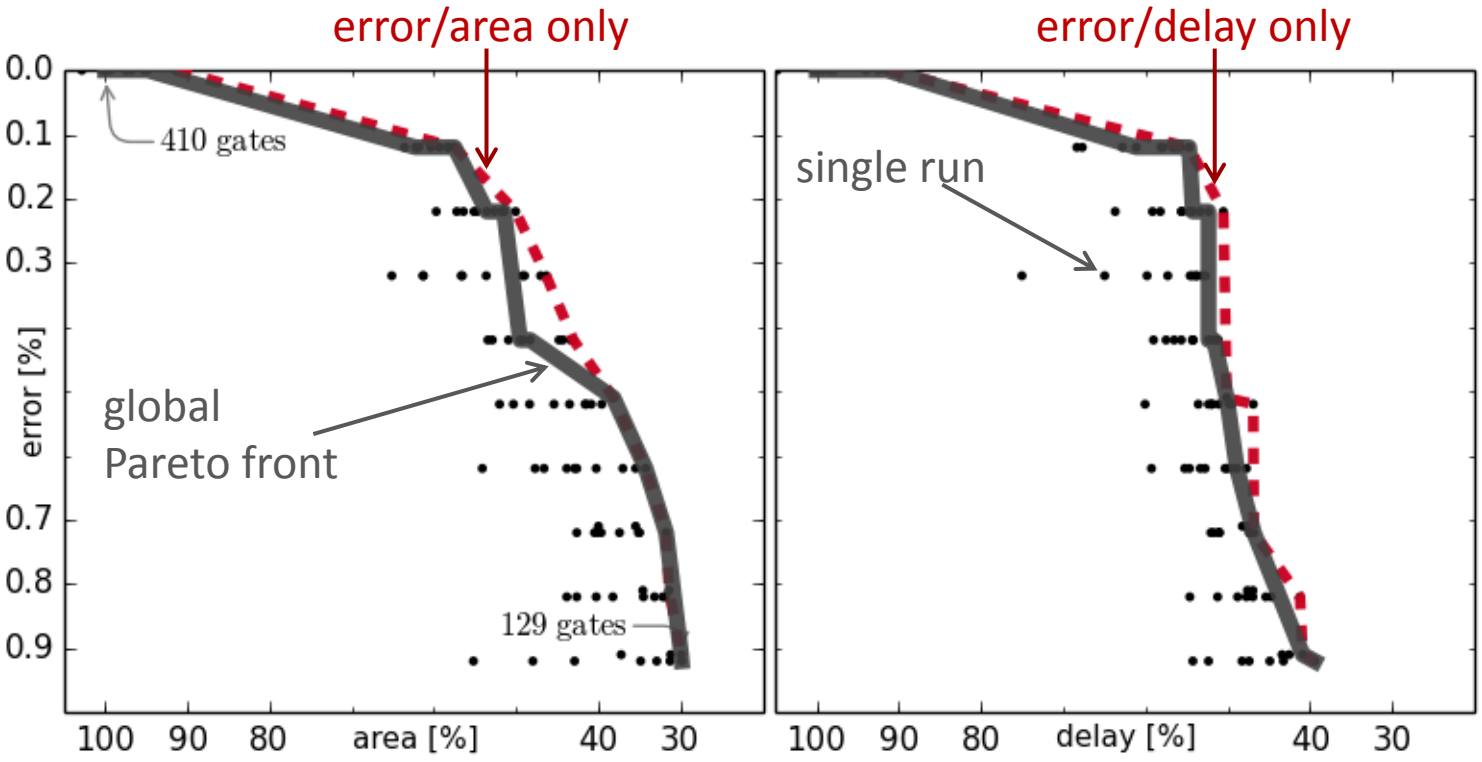
Circuit approximation with CGP and BDD

- 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

Vasicek Z., Sekanina L.: Evolutionary Design of Complex Approximate Combinational Circuits. Gen. Prog. and Evol. Mach. 17(2), 2016

CGP with BDD in the fitness function: Example



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

Properly optimize before doing approximations!

Vasicek Z., Sekanina L.: Evolutionary Design of Complex Approximate Combinational Circuits. Gen. Prog. and Evol. Mach. 17(2), 2016

Average-case error analysis

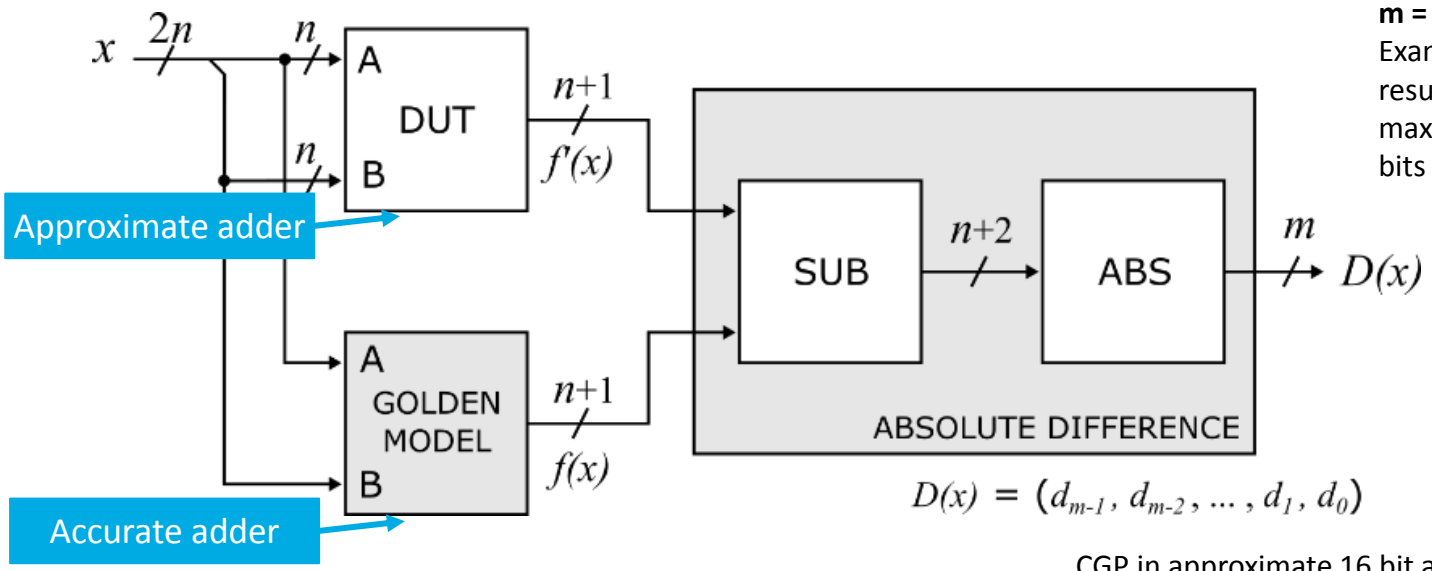
- Let $f: \mathcal{B}^n \rightarrow \mathcal{B}^m$ be a Boolean function that describes correct functionality and $\hat{f}: \mathcal{B}^n \rightarrow \mathcal{B}^m$ an approximation of it. The average-case error is defined as the sum of absolute differences in magnitude between the original and approximate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

- No practically useful method capable of establishing the average-case error using a SAT-based solver has been proposed up to now. The BDDs seem to be the only viable option how to calculate this error metrics.

Average-case error analysis using BDD



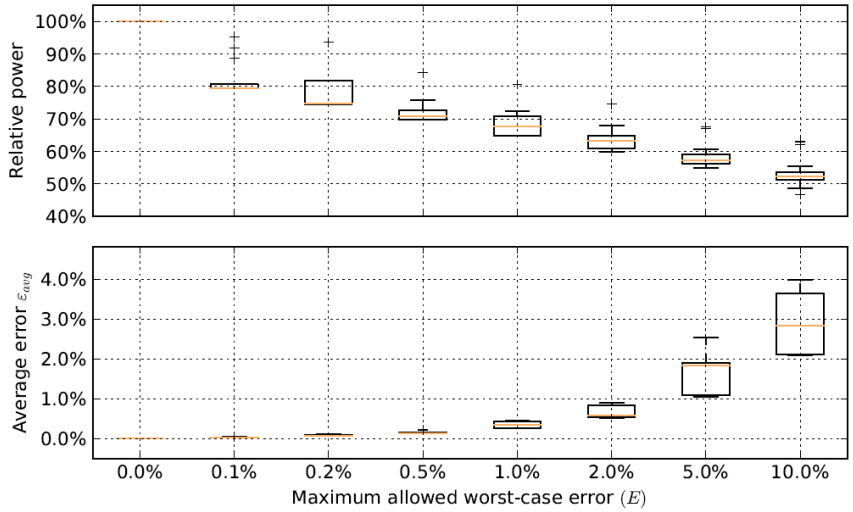
m = n + 2
Example for *n* = 4: Because the result of SUB is -32 … +31, the max absolute value is 32 and 6 bits are needed for *m*.

Algorithm 2: average-case error analysis

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

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

CGP in approximate 16 bit adder design



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

BDD vs exhaustive simulation

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

bit-width	*inputs*	*parallel simulation*	*BDD-based method*		*speedup*	
		$\mathcal{E}_{max} + \mathcal{E}_{avg}$	$\mathcal{E}_{max}$	$\mathcal{E}_{avg}$	$\mathcal{E}_{max}$	$\mathcal{E}_{avg}$
4-bit	8	4.5 us	10.3 us	14.0 us	0.43 ×	0.32 ×
8-bit	16	1.9 ms	3.5 ms	4.6 ms	0.54 ×	0.42 ×
12-bit	24	682.4 ms	127.9 ms	312.7 ms	5.33 ×	2.18 ×
16-bit	32	140.9 s	1.38 s	2.93 s	102.3 ×	48.09 ×

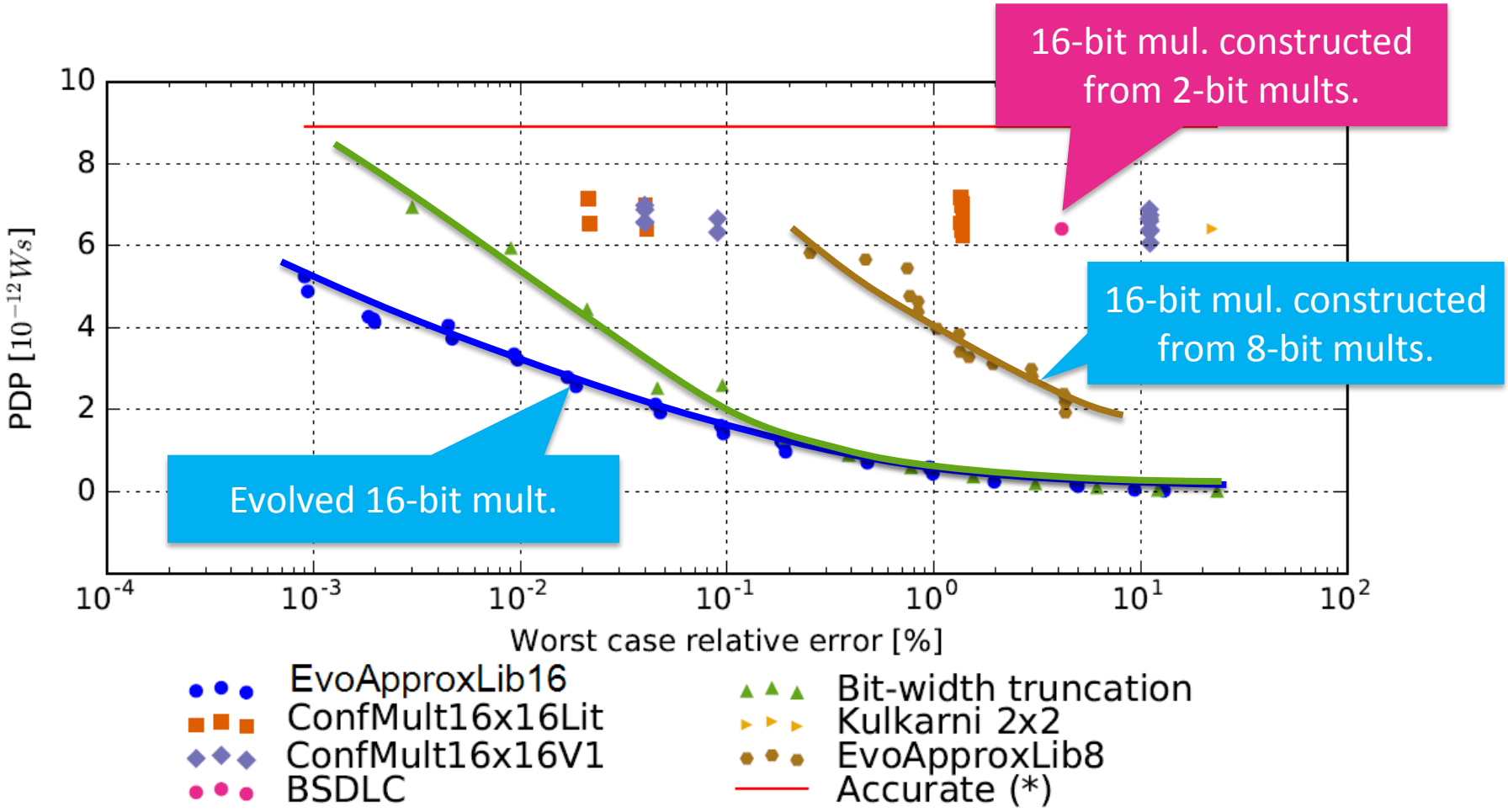
Notes

1) More than 100 randomly generated approximate adders were evaluated for each bit-width.
2) Time required to construct a BDD for a virtual circuit is included.

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

Is the search-based approximation competitive?

- Approximate 16 bit multipliers created using different methods
- PDP = Power Delay Product



Approx. circuits with formal error guarantees

List of the most complex arithmetic circuits that were successfully approximated and whose error is formally guaranteed.

Aprox. circuit	e_{wst}	e_{avg}	e_{prob}	e_{bf}	e_{wstac}	e_{HD}	method	paper	authors	conference
32-bit adder **!analysed only!**	X						BDDs	MACACO: Modeling and Analysis of Circuits for Approximate Computing	Venkatesan, Agarwal, Roy, Raghunathan	ICCAD 2011
8-bit multiplier	X						BDDs	MACACO: Modeling and Analysis of Circuits for Approximate Computing	Venkatesan, Agarwal, Roy, Raghunathan	ICCAD 2011
8-bit multiplier	X	X	X				simulation	Evolutionary Design of Approximate Multipliers Under Different Error Metrics	Vasicek, Sekanina	DDECS 2014
64-bit adder			X				BDDs	Analyzing Imprecise Adders Using BDDs - A Case Study	Yu, Ciesielski	ISVLSI 2016
16-bit adder				X			SAT, bin search	Approximation-aware Rewriting of AIGs for Error Tolerant Applications	Chandrasekharan, Soeken, Grosse, Drechsler	ICCAD 2016
16-bit adder	X						BDDs	Approximation-aware Rewriting of AIGs for Error Tolerant Applications	Chandrasekharan, Soeken, Grosse, Drechsler	ICCAD 2016
16-bit adder					X		PDR	Precise Error Determination of Approximated Components in Sequential Circuits with Model Checking	Chandrasekharan, Soeken, Grosse, Drechsler	DAC 2016
8-bit ALU c3540			X				BDDs	BDD Minimization for Approximate Computing	Soeken, Grosse, Chandrasekharan, Drechsler	ICCAD 2016
12-bit multiplier	X	X					simulation	Design of power-efficient approximate multipliers for approximate artificial neural networks	Mrazek, Sarwar, Sekanina, Vasicek, Roy	ICCAD 2016
8-bit mutiplier	X	X	X	X		X	simulation	Automatic Design of Approximate Circuits by Means of Multi-Objective Evolutionary Algorithms	Hrbacek, Mrazek, Vasicek	DTIS 2016
16-bit adder	X	X					BDDs	Towards Low Power Approximate DCT Architecture for HEVC Standard	Vasicek,Mrazek,Sekanina	DATE 2017

Conclusions

- Approximate computing is a hot topic!
  - It addresses one of the most critical challenges of our society -- energy efficiency.
- The roots of approximate computing:
  - energy-efficient computing is needed
  - high variability in current/future technology nodes
  - many applications are error resilient
- The approximation problem can be formulated as a multi-objective design/optimization problem
  - A holistic approach is needed.
  - A great opportunity for EAs!
- Current use of EA in Approximate computing
  - Optimization tasks (selection of types, variables, …)
  - Genetic improvement (with errors enabled)
  - Evolutionary design from scratch

References

- 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
  - H. Esmaeilzadeh, A. Sampson, L. Ceze, D. Burger, “Neural acceleration for general-purpose approximate programs,” Commun. ACM, 58(1): 105-115, 2015
  - S. Mittal, “A survey of techniques for approximate computing,” ACM Computing Surveys, 48(4), 1–34, 2016.
  - Q. Xu, T. Mytkowicz, N. S. Kim. “Approximate Computing: A Survey,” IEEE Design and Test, 33(1), 8-22, 2016.
  - L. Sekanina, “Introduction to Approximate Computing” (embedded tutorial). IEEE International Symposium on Design and Diagnostics of Electronic Circuits, DDECS 2016
  - Z. Vasicek, “Relaxed equivalence checking: a new challenge in logic synthesis” (embedded tutorial). IEEE International Symposium on Design and Diagnostics of Electronic Circuits, DDECS 2017

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 projects
  - IT4Innovations Centre of Excellence – National supercomputing center
  - 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)

Thank you for your attention!

http://www.fit.vutbr.cz/research/groups/ehw

EHW @FIT