Approximate Computing for Low-power: Survey and Challenges

Prof. Dr. Akash Kumar

Chair for Processor Design

(Ack: my past and current students/PostDocs)

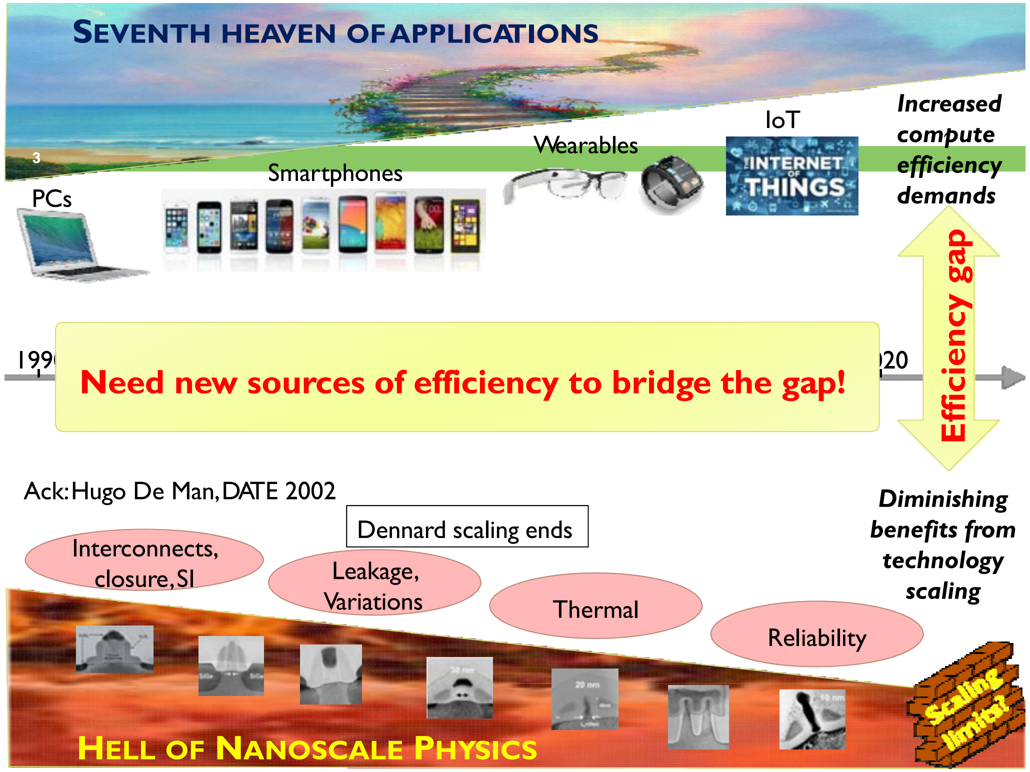
(Some slides adapted from Anand)

cfaed.tu-dresden.de

Outline

2

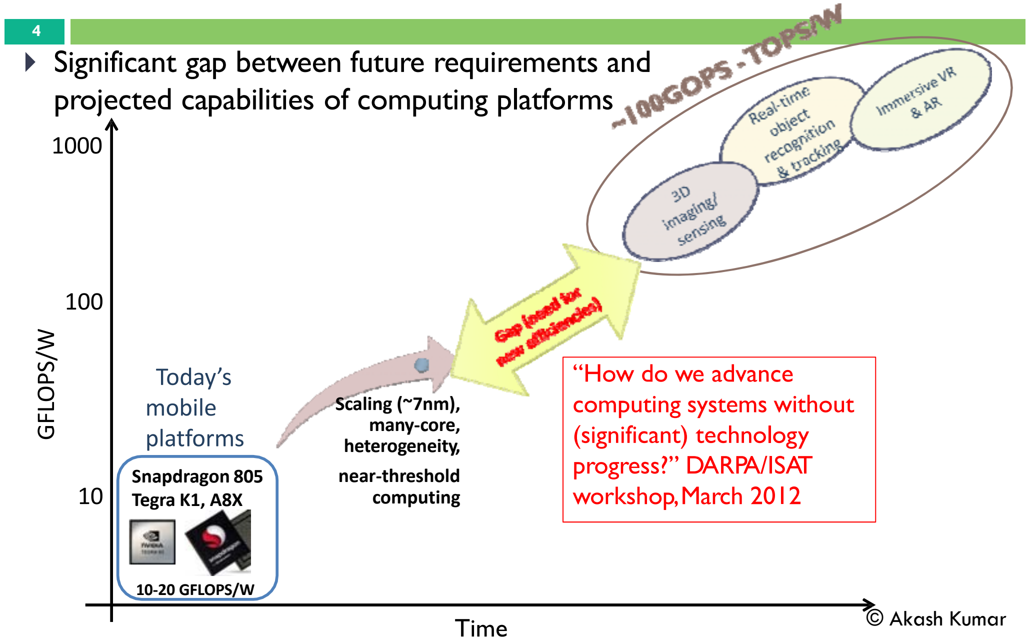
- Why?
  - Motivation for Approximate Computing
- What?
  - Approximate computing: Design philosophy and approach
- How?
  - Technologies for Approximate Computing



Efficiency Gap In Computing

4

- Significant gap between future requirements and projected capabilities of computing platforms



The Computational Efficiency Gap

5

IBM Watson playing Jeopardy, 2011

Humans Approximate

6

Task: Division

$$is\ \frac{923}{21} > 1.75?$$

$$is\ \frac{923}{21} > 45?$$

21) 923 (43 - - -
84
83
63
⋮

$$\frac{923}{21} = --.--?$$

Accuracy

~1Petaflop/W

Application context dictates required accuracy of results

Effort expended increases with required accuracy

But Computers DO NOT

7

$$\frac{923}{21} > 45$$

```
float x = 923;
float y = 21;
cout << (x/y > 45.0) ?
"YES":"NO";
```

NO

$$\frac{923}{21} > 1.75$$

```
float x = 923;
float y = 21;
cout << (x/y > 1.75) ?
"YES":"NO";
```

YES

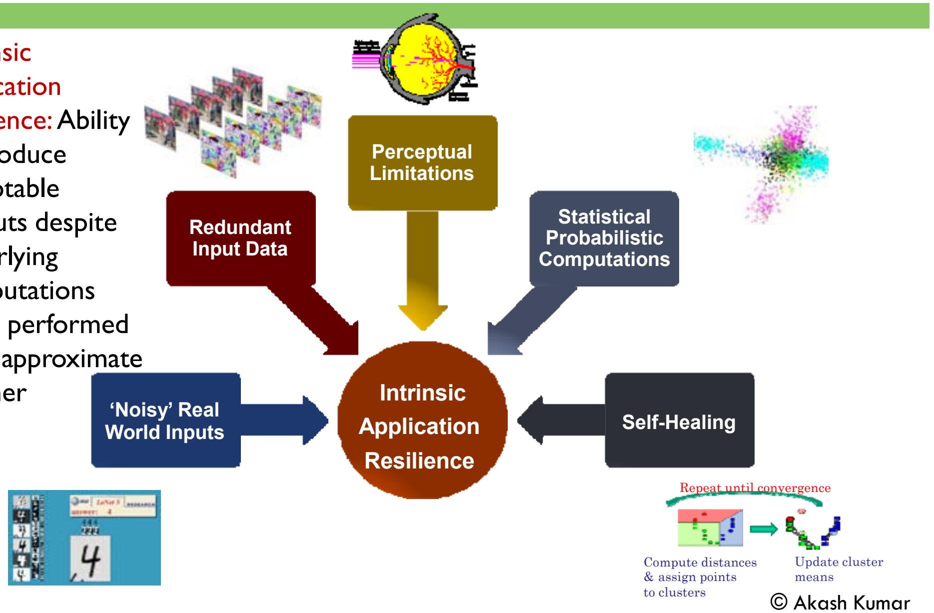
But, I worked harder than needed

- Overkill (for many applications)
- Leads to inefficiency
- Can computers be more efficient by producing "just good enough" results?

Intrinsic Application Resilience: Sources

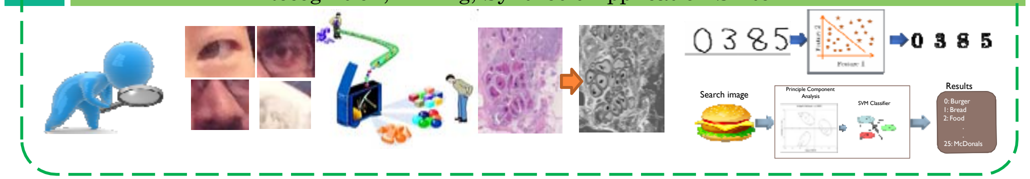
8

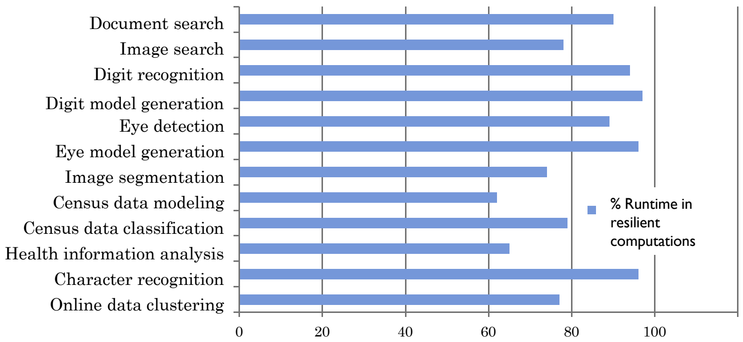
- Intrinsic application resilience: Ability to produce acceptable outputs despite underlying computations being performed in an approximate manner



Intrinsic Resilience In RMS Applications

Recognition, Mining, Synthesis Application Suite





Applications have a mix of resilient and sensitive computations

83% of runtime spent in computations that can be approximated

V.K. Chippa, S.T. Chakradhar, K. Roy and A. Raghunathan, "Analysis and characterization of inherent application resilience for approximate computing," DAC 2013.

Its an Approximate World … At the Top

- No golden answer (multiple answers are equally acceptable)
  - Web search, recommendation systems

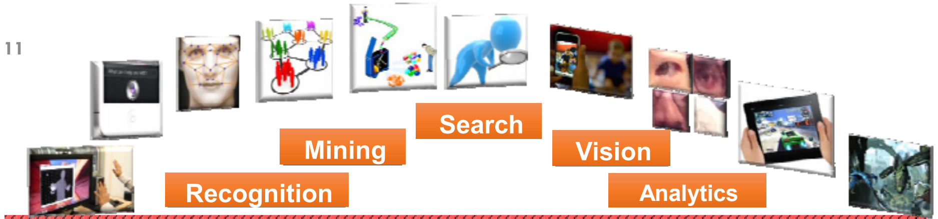
- Even the best algorithm cannot produce correct results all the time
  - Most recognition / machine learning problems

- Too expensive to produce fully correct or optimal results
  - Heuristic and probabilistic algorithms, relaxed consistency models, …

Miller-Rabin primality test

Eventual consistency

Yet, Computing Lives In A Utopian World!



Programming Languages, Compilers, Runtimes
Architecture
Logic
Circuits

Strict Numerical or Boolean Equivalence

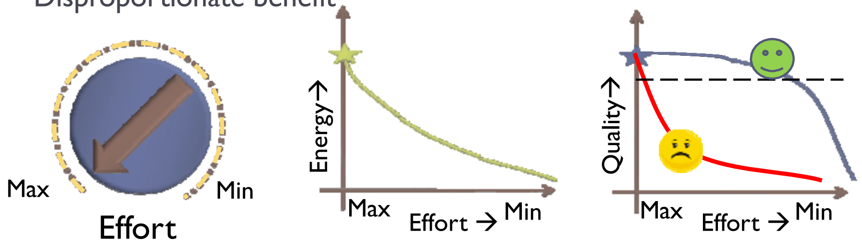
Outline

- Why?
  - Motivation for Approximate Computing
- What?
  - Approximate computing: Design philosophy and approach
- How?
  - Technologies for Approximate Computing

Approximate Computing: Design Philosophy

13

- Computing platforms that can modulate the effort expended towards quality of results
  - Higher effort → Higher quality but lower efficiency
- How do we get the best Q vs. E tradeoff?
  - Disproportionate benefit

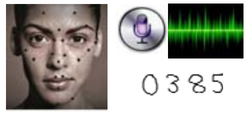


Its an Approximate World … At the Top

14

No golden answer

Perfect/correct answers not always possible



Too expensive to produce perfect/correct answers

Miller-Rabin primality test

Eventual consistency

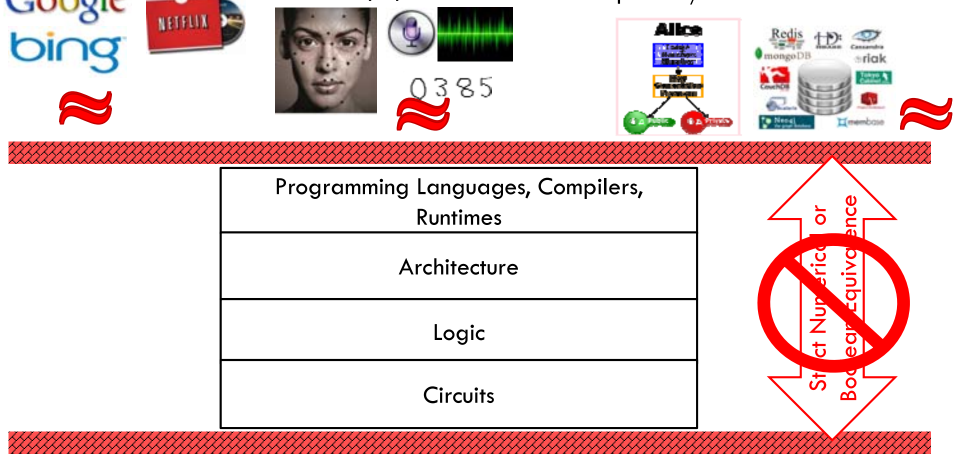
Approximate Computing Throughout the Stack

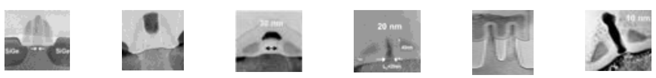
15

No golden answer

Perfect/correct answers not always possible

Too expensive to produce perfect/correct answers





Where did Approximate Computing come from?

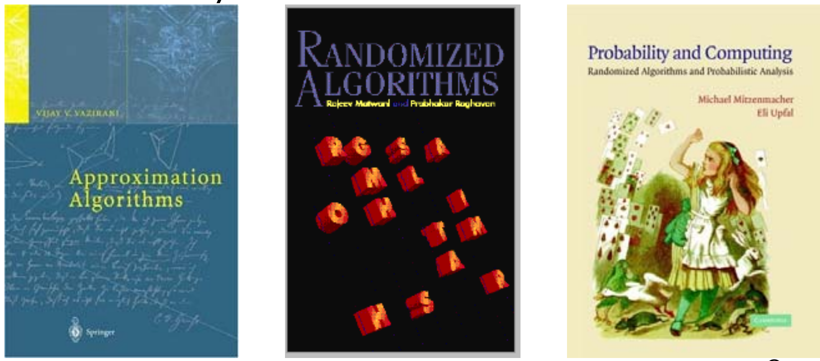
16

- Tradeoffs between Quality of Results and Efficiency are not new
  - Intellectual roots of approximate computing can be traced back to many fields

Where did Approximate Computing come from?

17

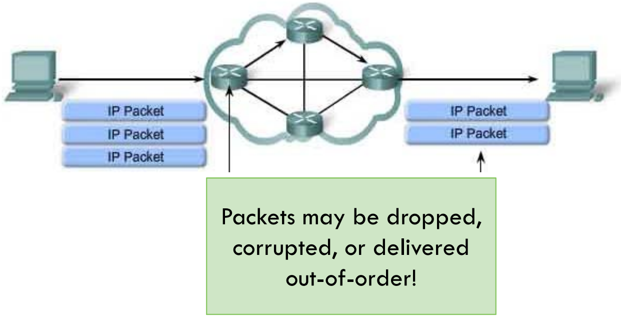
- Approximation, Heuristic, and Probabilistic algorithms
  - Tradeoff amount of work for sub-optimal or occasionally incorrect results



Where did Approximate Computing come from?

18

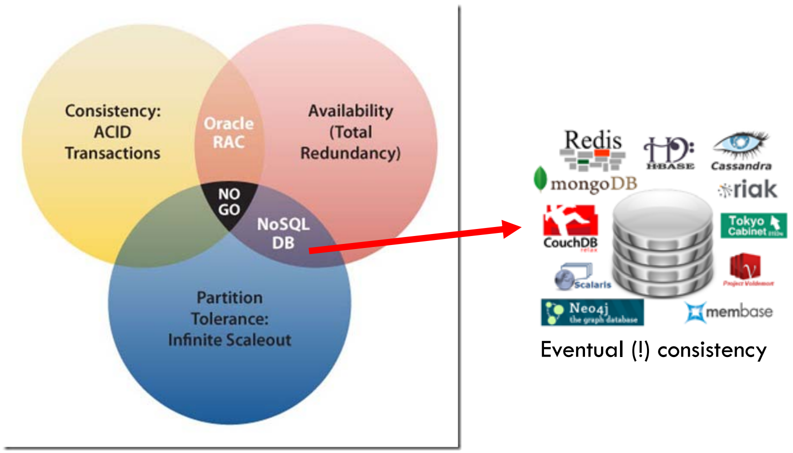
- Networking
  - Best-effort packet delivery (IP)
  - Reliability layered on top only when needed (TCP)
  - Many apps do not need or use reliable packet delivery!
    - Video, audio streaming



Where did Approximate Computing come from?

19

- Large-scale unstructured data storage



Where did Approximate Computing come from?

20

- Digital Signal Processing
  - Filter design (optimize taps, coefficients, and precision based on specifications)

Approximate Computing Now: Why?

21

- Arising from the application level
  - Inherent lack of notion or ability for a single 'correct' answer
  - 'Noisy' or redundant real-world data
  - Perceptual limitations

- Arising from the transistor level
  - Increasing fault-rates
  - Increased effort/resource to achieve fault-tolerance

Outline

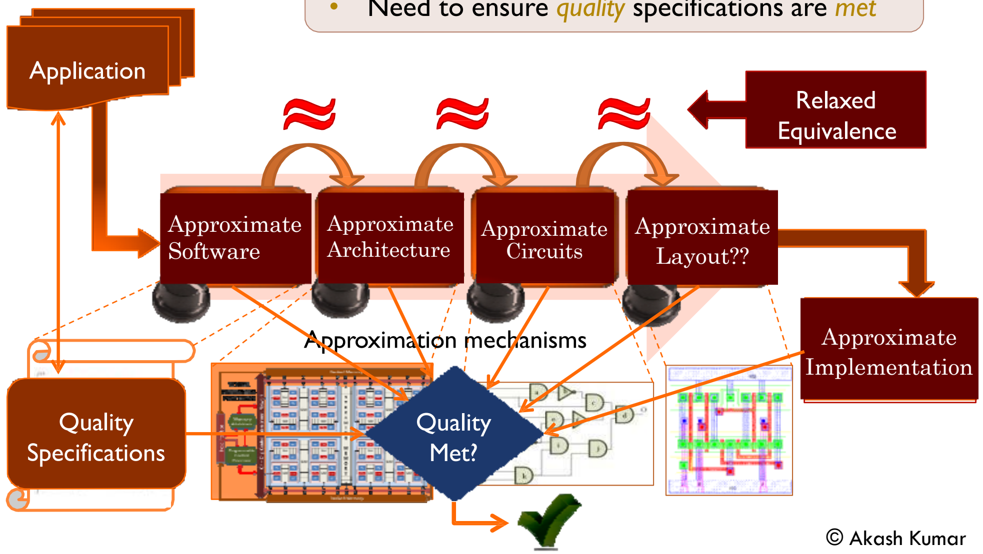
22

- Why?
  - Motivation for Approximate Computing
- What?
  - Approximate computing: Design philosophy and approach
- How?
  - Technologies for Approximate Computing

Approximate Computing Approach

23

- *Approximations* at various layers of abstraction
- Need to ensure *quality* specifications are *met*



Some Early Efforts in Approx. Computing*

24

- Approximate signal processing (Chandrakasan et. al, 1997)
- Voltage overscaling (Shanbhag et. al, ISLPED 1999)
- Probabilistic CMOS (Palem et. al, 2003)
- Manufacturing yield enhancement (Breuer et. al, 2004-)
- Energy-efficient, variation-tolerant approximate hardware (Roy et. al, 2006-)
- Probabilistic Arithmetic / Biased voltage overscaling (Palem et. al, CASES 2006-)
- Parallel runtime framework with computation skipping, dependency relaxation (Raghunathan et. al, IPDPS 2009; IPDPS 2010)
- Error-resilient / stochastic processors (Mitra et. al, 2010; Kumar et. al, 2010)
- Cross-layer, scalable-effort approximate HW design (Chippa et. al, 2010)
- Programming support for approximate computing (Chilimbi el. al, 2010; Misailovic et. al, 2010; Sampson et. al, 2011)
- …
- …
- http://timor.github.io/refgraph/ Dancing authors. ☺

Why some in red?

*** Not an exhaustive list!**

Approximate Software

Largely based on:

[1] Mittal, "A survey of techniques for approximate computing", ACM Computing Surveys 2016
[2] Shafique, Hafiz, Rehman, El-Harouni & Henkel, "Cross-layer Approximate Computing: From Logic to Architectures", DAC 2016

Approximate Software

- Techniques can be applied at
  - Compile-time OR
  - Run-time
- Frameworks that exploit multiple layers
  - Precision specification -> identify and specify what to approximate
  - Precision reduction implementation -> actually perform and control approximation
- Application at different layers (Better throughout!)
  - Language
  - Algorithm
  - Compiler

Compile-time vs Run-time

- Compile-time
  - Use information available before execution
  - Possibly lower execution overhead
  - Need analysis on accuracy bounds
- Run-time
  - More lenient towards incomplete accuracy analysis
  - Generally larger overhead
- Combination of the two
  - Runtime reconfigurable approximate systems

Precision Specification

1. Code annotation
2. Built-in Language support
3. Explicit algorithm techniques
4. Output quality monitoring

Precision Specification

29

1. Code annotation
   - using existing programming languages with "magic" markers
     - Comments, pragmas
   - ignored by regular compiler, but can be processed by special preprocessors
   - E.g. iACT

```
//axc_memoize for loops

# pragma axc_memoize [(0:5),(1:10)]out(z)
for ( i = 0; i < n; i = i + 1 ) {
        z = f(x, y);
}
```

```
//axc_memoize for functions

# axc_pragma [(0:5),(1:10)]{2}
foo(x, y, &ret);
```

```
float foo(float x, float y, &var_ret) {
    var_ret = x + y;
    return ret;
}
```

Mishra, Barik & Paul, "iACT: A software-hardware framework for understanding the scope of approximate computing", WACAS, 2014

Precision Specification

30

2. Built-in Language support
   - implemented by extending existing programming languages or designing a new programming language

EnerJ, Proposed by Sampson (2011)
- Use Type Qualifiers to indicate approximate data and operations
- Approximation-aware execution substrate can make use of this additional information

```
@Approx int a = ...;
int p;          // precise by default
p = a;          // illegal
```

Sampson, Dietl, Fortuna, Gnanapragasam, Ceze & Grossman, "EnerJ: Approximate Data Types for Safe and General Low-power Computation", PLDI 2011

Precision Specification

31

2. Built-in Language support

Rely, Proposed by Carbin (2013)
- allows to program explicitly on unreliable hardware, while giving guarantees on error probabilities
- incorporates Hardware Reliability Specification
- used for numerical calculations, e.g. computation kernels
- knowledge about intermediate reliablity constraints needed

- specify joint reliability of operations in signature `.. R(x,y)..`
- specify data in unreliable storage: … `in urel`

Carbin, Misailovic & Rinard, "Verifying Quantitative Reliability for Programs That Execute on Unreliable Hardware", OOPSLA, 2013

Precision Specification

32

```
#define nblocks 20
#define height 16
#define width 16
int <0.99*R(pblocks , cblock) > search_ref (
        int <R(pblocks) > pblocks(3) in urel ,
        int <R(cblock) > cblock(2) in urel)
{
        int minssd = INT_MAX , minblock = -1 in urel ;
        int ssd , t , t1 , t2 in urel ;
        int i = 0 , j , k ;
        repeat nblocks {
                ssd = 0; j = 0; ... i = i + 1;
        }
        return minblock ;
}
```

Precision Specification

33

2. Built-in Language support

Axilog, Proposed by Mahajan (2015)

- extend verilog syntax with annotations to declare arguments safe to approximate
- infer which other connections and gates are safe to approximate
- synthesis can either relax timing constraints or assume probabilistic gate models

```
module full adder(a, b, c in, c out, s);
    input a, b, c_in; output c_out;
    approximate output s;
    assign s = a ^ b ^ c_in;
    assign c_out = a & b + b & c_in + a & c_in;
    relax(s);
endmodule
```

Mahajan et al, "Axilog: Abstractions for Approximate Hardware Design and Reuse", Micro 2015

Precision Specification

34

3. Explicit Algorithm Techniques
   - careful analysis of algorithm and input data properties
   - manually optimize code with existing means
   - no automation

4. Output quality monitoring
   - measure output quality and adjust "control knobs" accordingly
   - role of quality metrics is most important
   - quality metrics often application-specific

Precision Reduction Implementation

35

1. Loop perforation
2. Precision Scaling
3. Memoization
4. Task Skipping
5. Program Selection
6. Neural Network Substitution
7. Approximate Storage

Precision Reduction Implementation

36

1. **Loop perforation –** identify loops where only a subset of iterations can be performed while maintaining acceptable accuracy
2. **Precision Scaling –** right-shift data or truncate
3. **Memoization –** use for functions with similar input/output pairs
4. **Task Skipping –** perform subset of tasks
5. **Program Selection –** select from multiple versions
6. **Approximate Storage –** allow data to degrade
7. Neural Network Substitution

Precision Reduction Implementation

- Neural Network Substitution
  - Replace part of the program with an accelerator based on neural network
  - NN needs to be trained with input/output data sets of original function

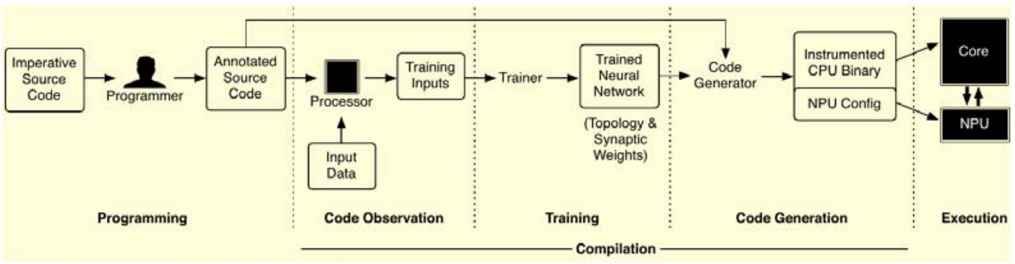


Figure 1: The Parrot transformation at a glance: from annotated code to accelerated execution on an NPU-augmented core.

Esmaeilzadeh, Sampson, Ceze & Burger, "Neural Acceleration for General-Purpose Approximate Programs", Micro, 2012

Overall Frameworks

1. **Green** — Baek & Chilimbi, "Green: A Framework for Supporting Energy-conscious Programming Using Controlled Approximation", PLDI, 2010
2. **iACT** — Mishra, A. K.; Barik, R. & Paul, S. iACT: A software-hardware framework for understanding the scope of approximate computing, WACAS, 2014
3. **GRATER** — Lofti, A.; Rahimi, A.; Yazdanbakhsh, A.; Esmaeilzadeh, H. & Gupta, R. K. GRATER: An Approximation Workflow for Exploiting Data-Level Parallelism in FPGA Acceleration, DATE 2016

Green

- Whole-stack flow
- Uses language extensions
- Supports loop termination and approximate function selection
- Generates necessary support code to perform adaptive QoS control at runtime
- Approximate functions have to be supplied by user

GREEN

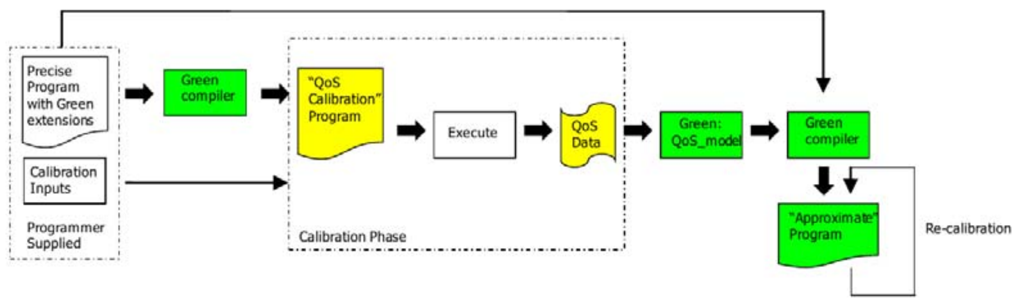


Figure 1. Overview of the Green system.

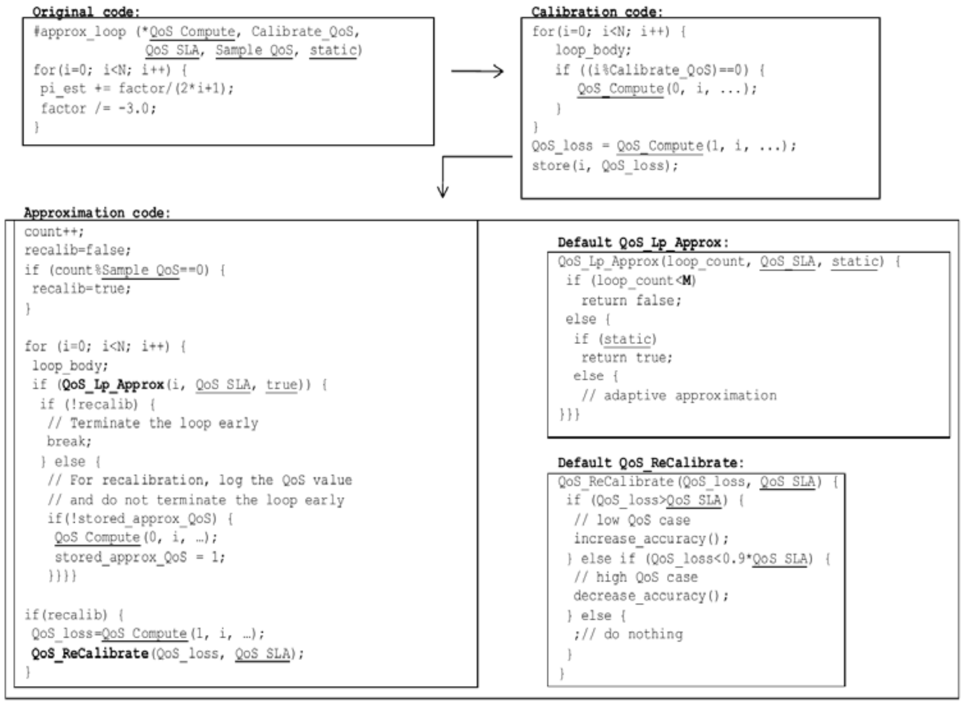


Figure 3. An end-to-end example of applying loop approximation to the Pi estimation program.

iACT

- Compiler, runtime and simulated hardware test bed
- Use pragmas to annotate approximation amenable functions
- Compiler performs static analysis, places **annotations in binaries**
- Supported transformations:
  - automated variable precision reduction
  - noisy ALU computations
  - approximate memoization

iACT Capabilities

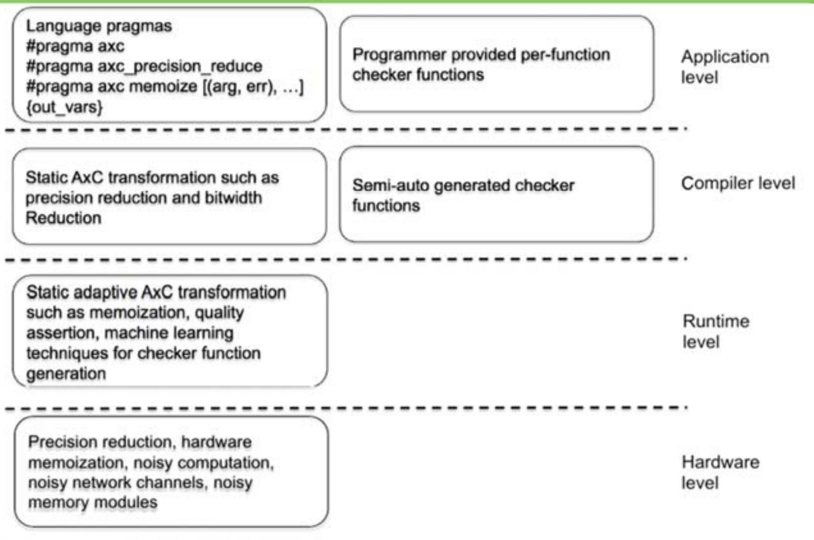


Figure 2. Summary of the capabilities of iACT.

GRATER

- Synthesize smaller hardware accelerators of OpenCL computation kernels automatically, exploiting inherent application error tolerance
- Uses genetic algorithm to find operations whose precision can be reduced safely
- Increases data-level parallelism by allowing to place more functional units
- Generate implementation for FPGA (in their case, Altera)

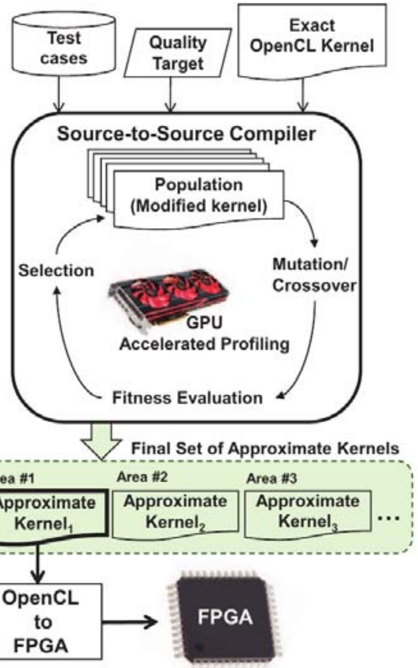


Fig. 2: Overview of GRATER, our approximation design workflow.

Approximate Computing in Software

- Templates allow programmers to easily specify mechanisms for computation skipping and dependency relaxation
  - Auto-tuning and runtime frameworks explore quality-speed tradeoff

Example: Iterative-convergence pattern

```
iterate {
  for (i = 1 to M) {
    …
  }
} until converged();
```

```
iterate {
  for ( j = 1 to M) {
    ..
  }
} until converged( );
```

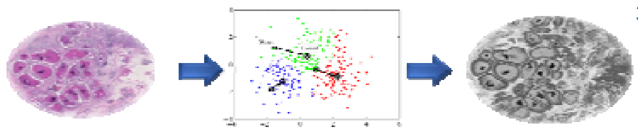
Approximate

When to stop iterations

```
float *[M] points;
float * [k] means;
int [M] memberships;
float *[M] distances;
K-means() {
    means = random_array(k);
    iterate {
        for (i= 1 to M) {
            distances[i] = calc_distances(means,
            points[i]);
            memberships[i]=compute_cluster(distance
            s);
        }
        means = calc_means(points, memberships);
    } until (no_change(means));
}
```


Approximate Computing in Software

- Image segmentation (K-means)

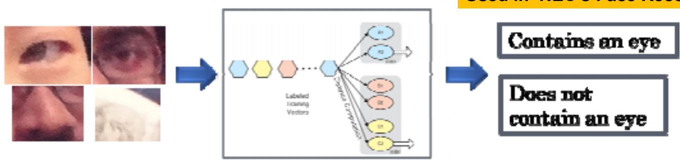


Dell 2950 (8-core Xeon, 32GB RAM), Intel TBB

5X speedup
99% accuracy

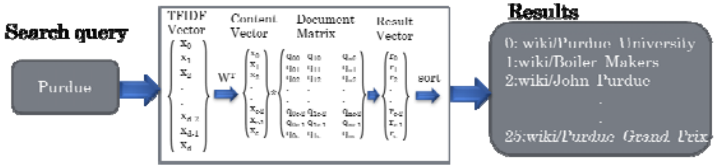
- Face detection (GLVQ)

Used in NEC's Face Recognition Products



4.9X speedup
0.1% error

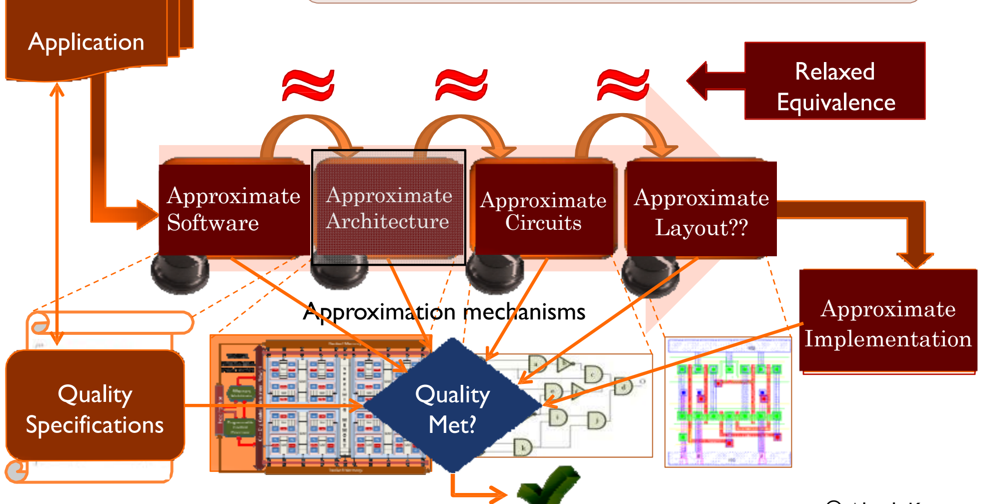
- Semantic Document Search



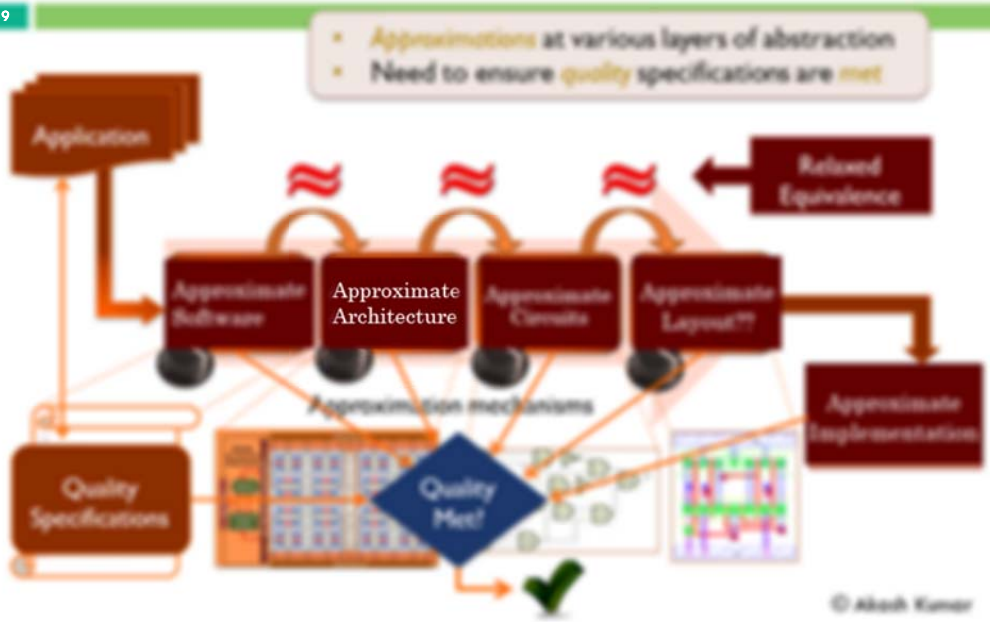
3X speedup
Iso-accuracy

Approximate Computing Approach

- *Approximations* at various layers of abstraction
- Need to ensure *quality* specifications are *met*



Approximate Computing Approach



Approximate Architecture

Approximate Architecture

Algorithm-specific accelerators → Domain-specific accelerators – image, video … → Programmable accelerators (GPGPUs, MIC) / Vector processors → General purpose processors/ Multicores

- ANT – Hedge *et. al.* – ISLPED 1999
- Significance driven computing – Mohapatra et. al. – ISLPED 2009
- Scalable effort hardware – Chippa et. al. – DAC 2010, DAC 2011

Application specific designs

- ERSA – Leem *et. al.* – DATE 2010
- Stochastic processor – Narayanan *et. al.* – DATE 2010

Cores of different reliabilities

- Truffle – Esmaeilzadeh *et. al.* – ASPLOS 2012
- EnerJ – Sampson *et. al.* – PLDI 2011

Accurate and approximate instructions

Approximate Architecture

Algorithm-specific accelerators → Domain-specific accelerators – image, video … → Programmable accelerators (GPGPUs, MIC) / Vector processors → General purpose processors/ Multicores

Pros:

☺ Large energy benefits

☺ Broader applicability

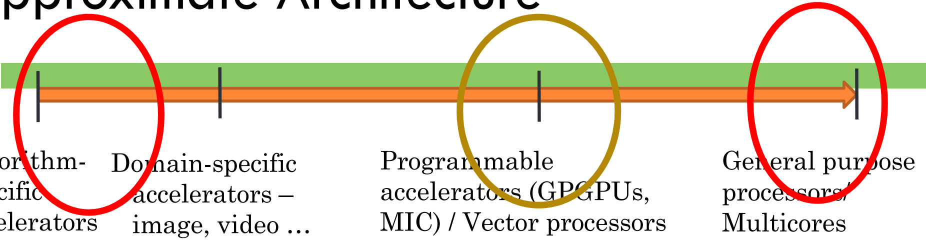
Challenges:

☹ Limited applicability

☹ Inherently limited energy benefit – Dominated by control front-ends that cannot be approximated

☹ Allow arbitrary errors in hardware – limits the fraction of computations that can be approximated

Approximate Architecture



Opportunity:

☺ Wide range of applications – fine grained parallelism

☺ SIMD:Control overheads amortized over many execution units

☺ Need quality guarantees from HW

Swagath Venkataramani, Vinay K. Chippa, Srimat T. Chakradhar, Kaushik Roy, and Anand Raghunathan, Quality programmable vector processors for approximate computing, MICRO 2013.

Designing Inexact Systems Efficiently using Elimination Heuristics

DATE 2015

Introduction

- Diminishing transistor sizes ⇨ increase in power density and errors
- Inexact computing can trade accuracy for significant gain in area/power
- Real-world examples where accuracy can be traded
  - Video streaming (errors in few pixels considered okay)
  - Brain-inspired computing architectures
  - Learning and decision systems

Background and Motivation

- Previous works:
  - Decreasing the voltage of operation significantly to reduce the power consumption albeit at the cost of reliable circuit operation [Kim, ACM JETC 2014][George, CASES 2006]
  - Reducing the number of transistors in order to save energy [Lingamneni, ACM TECS 2013]
  - Removing parts of circuit that have a lower probability of being active – probabilistic pruning [Lingamneni, DATE 2011]
- However, designing such inexact systems is expensive
- Exponential growth in search space

Background and Motivation

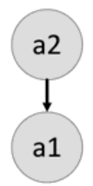
- Current inexact systems lack
  - Ability to estimate quickly the overall inexactness of a system
  - Identifying the best set of inexact components to use from a given set of components
- Having an overall design flow to construct such inexact systems with tunable parameters is the scope of this research

Contributions

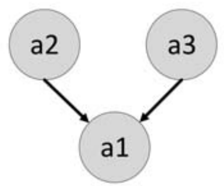
- Algorithm to quickly estimate the inexactness of the larger components
- A design-flow that uses the above algorithm to design the entire system under the area and power constraints
- A heuristic to reduce the design-space exploration time by eliminating the non-distinct points.
- Results of the design-flow applied to an ECG application of QRS detection.

Design Flow – Inexact Components

- Inexact components considered
  - Adders
  - Multipliers
- 2 types of configurations for adders and multipliers
  - Series
  - Parallel

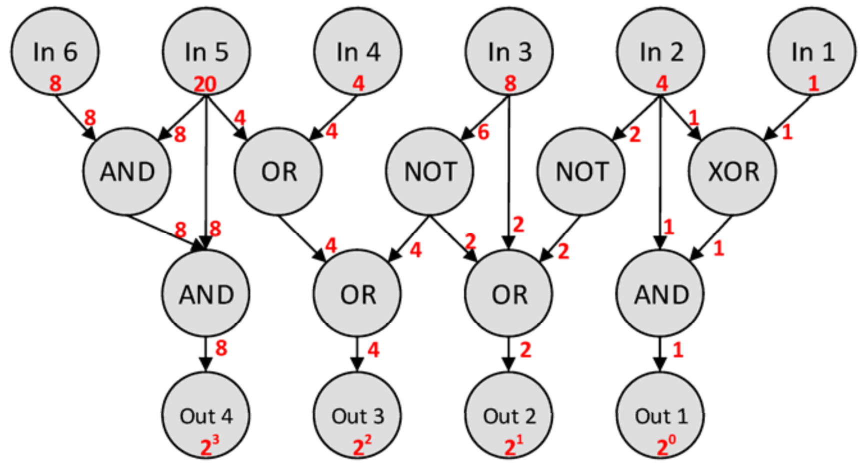


Adders in series



Adders in parallel

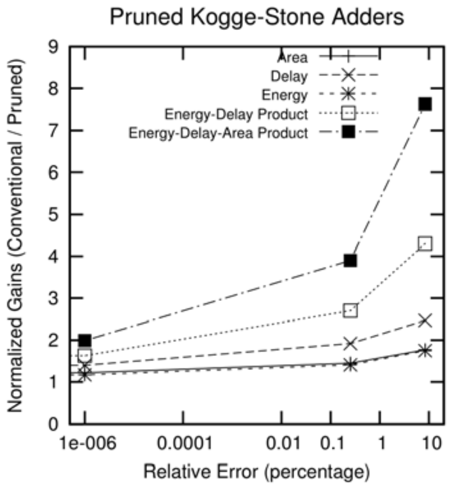
Probabilistic pruning



Probabilistic Pruning – Accuracy Tradeoff

61

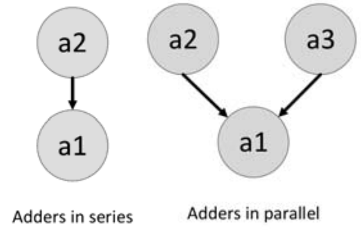
- Accuracy tradeoff for adder/multiplier
  - As more nodes pruned, gains in area, delay and energy increase
  - An order of magnitude improvement in energy-delay-area for 10% error



Design Flow – Parameter Computation

62

- Calculation of overall design parameters
  - Area - ∑
  - Power - ∑
  - Delay – path with maximum delay in given design
  - Relative error – path with maximum error in given design
  - Essentially identifying critical path



$$R_{P_i} = \prod_{k=1}^{n} 1 - R_{v_k}$$

Design Flow – Optimization Problem

63

- Given
  - Inexact versions for each adder and multiplier
- Objective
  - Choose the inexact versions for all components such that we get the most significant gains in power/delay/area with the least tradeoff in accuracy

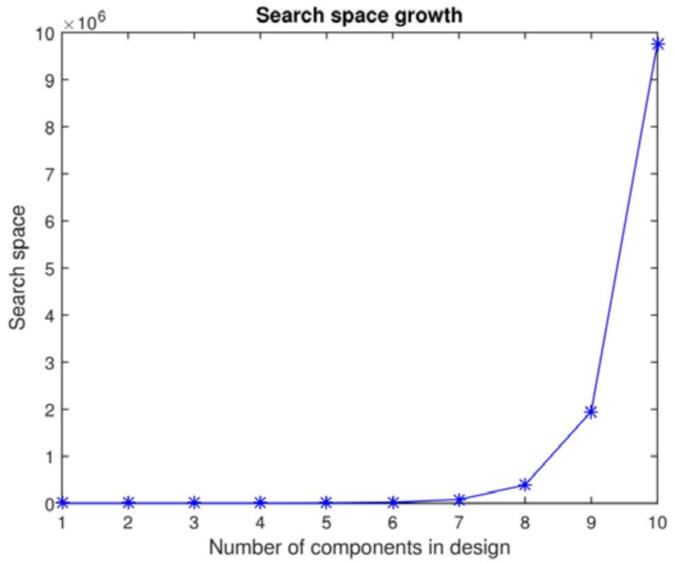
Design Flow – Optimization Problem

64

- Exhaustive search – exponential growth
  - For a system with 2 adders and 2 multipliers with 5 inexact versions of each – design search space is 625 points
  - For a system with 5 adders and 5 multipliers with 5 inexact versions of each – design search space is 9.7 million points
  - 37 years for simulating all options!

Design Flow – Optimization Problem

65



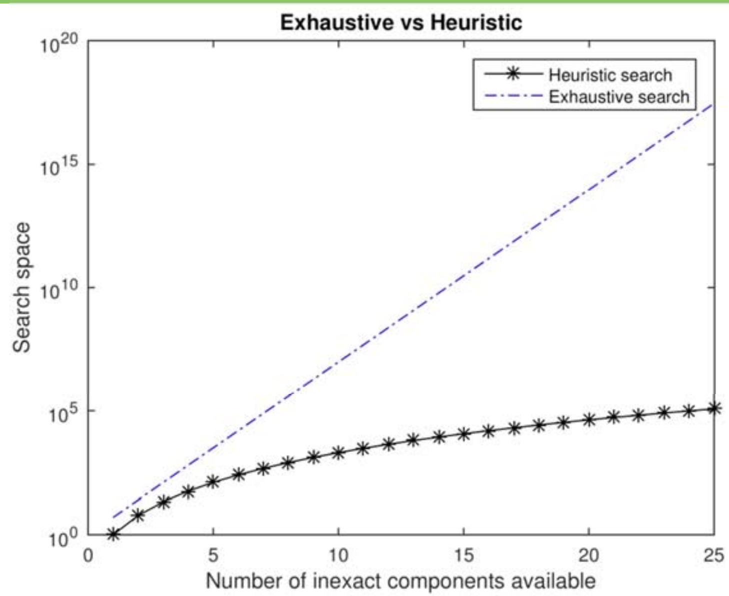
Design Flow – Heuristic Search

66

- Reduce design space exploration time
  - Order of inexact components does not matter (??!!)
  - Only designs which would result in distinct Pareto points considered
  - Design space compared to exhaustive search
    - 2 adders, 2 multipliers = 64 points (vs 625)
    - 5 adders, 5 multipliers = 16,384 points (vs 9.7mln) (Still 22 days!)
  - Orders of magnitude smaller than exhaustive search

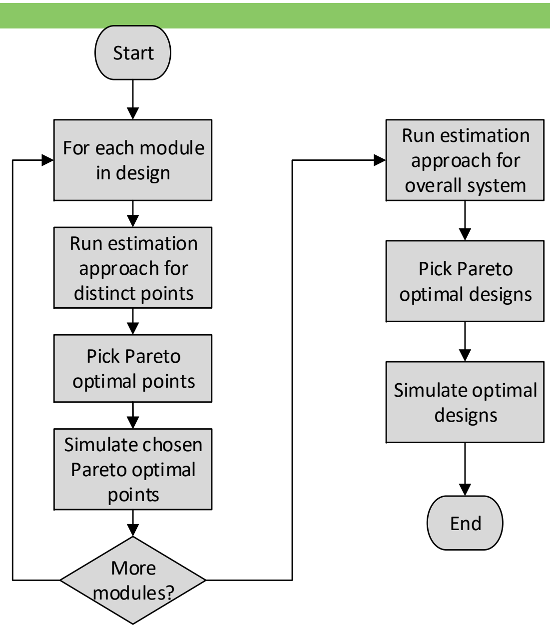
Design Flow – Heuristic Search

67

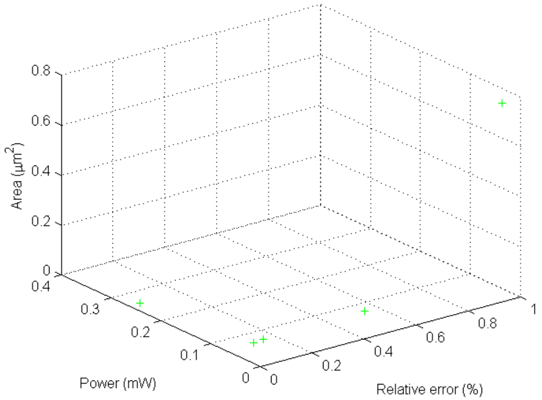


Design Flow – System level

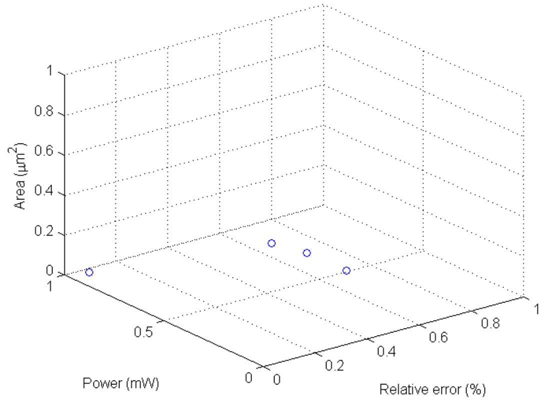
68



Module 1 → Module 2 → Module 3

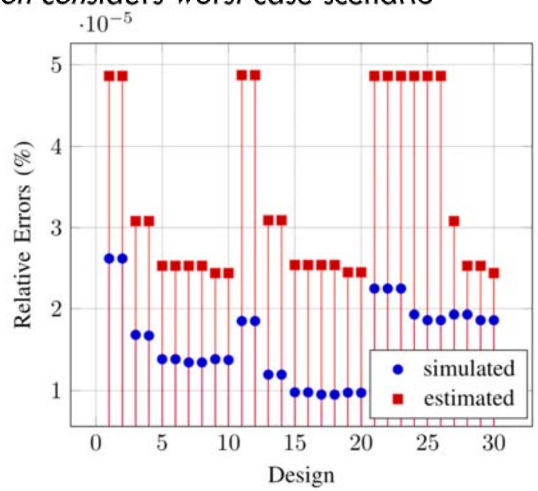


Module 1 → Module 2 → Module 3



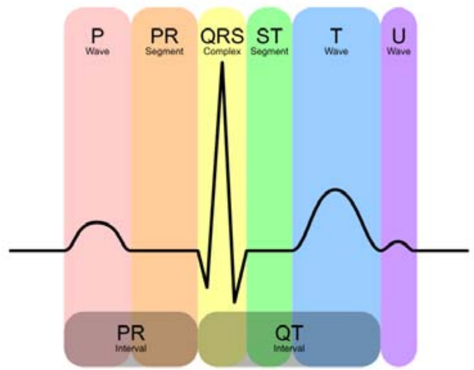
Results – Accuracy of Estimation

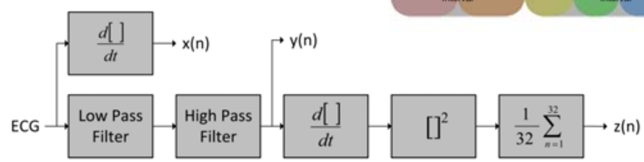
- Estimation and simulation results in similar trend
- Estimation considers worst case scenario



Case Study - Background

- QRS detection, one of the most important features of ECG considered
- Figure below shows steps required to process ECG signal before QRS can be detected

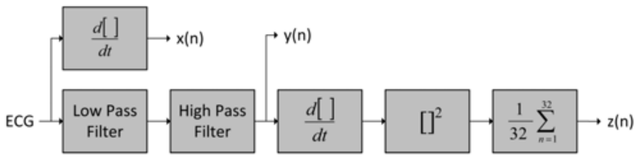




Case Study – Inexact QRS design

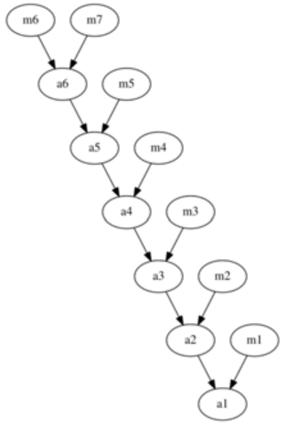
- 5 inexact adders and 5 inexact multipliers chosen to implement the different filters

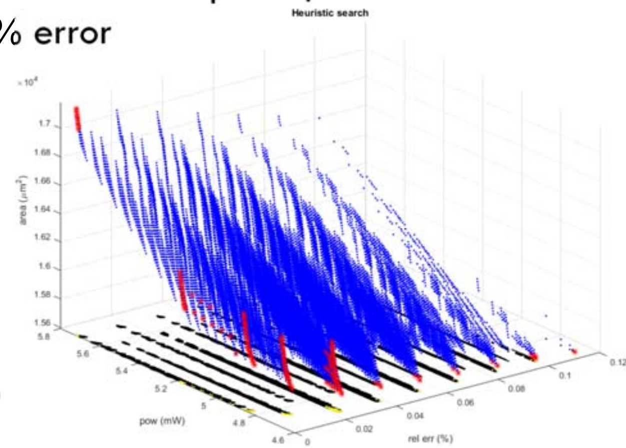
Sub-design	Number of taps	Number of adders	Number of Multipliers
Low pass filter	6	6	7
High Pass Filter	16	16	17
Differentiator	4	4	5
Squaring	0	0	1
Integrator	30	30	1



Case Study – Low Pass Filter

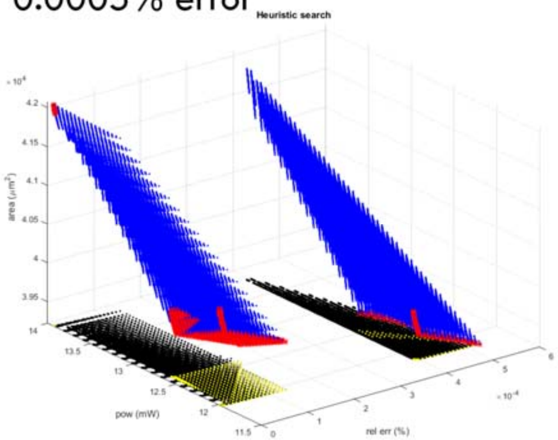
- Different points obtained for the low-pass filter using estimation approach – 20% power, 10% area savings for 0.1% error





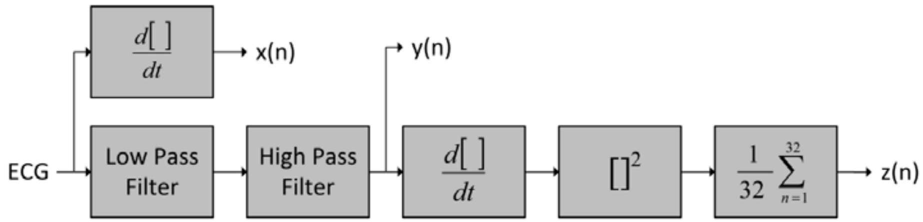
Case Study – High Pass Filter

- Different points obtained for the high pass filter using estimation approach – 15% power, 10% area savings for 0.0005% error



Case Study – Heuristic for entire system

- 5 Pareto optimal points chosen for each module
- Estimation on distinct points run for entire system
- 5 Pareto optimal designs finally chosen for simulation

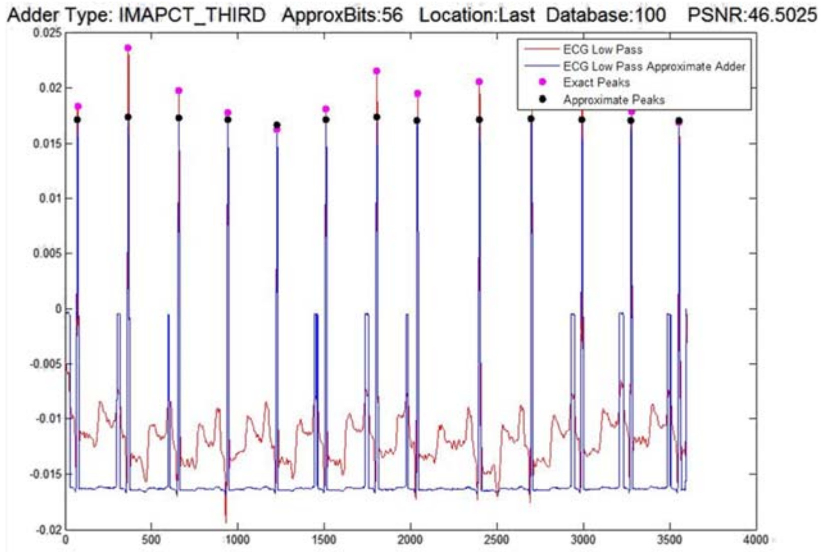


Case Study – Exact vs Inexact design

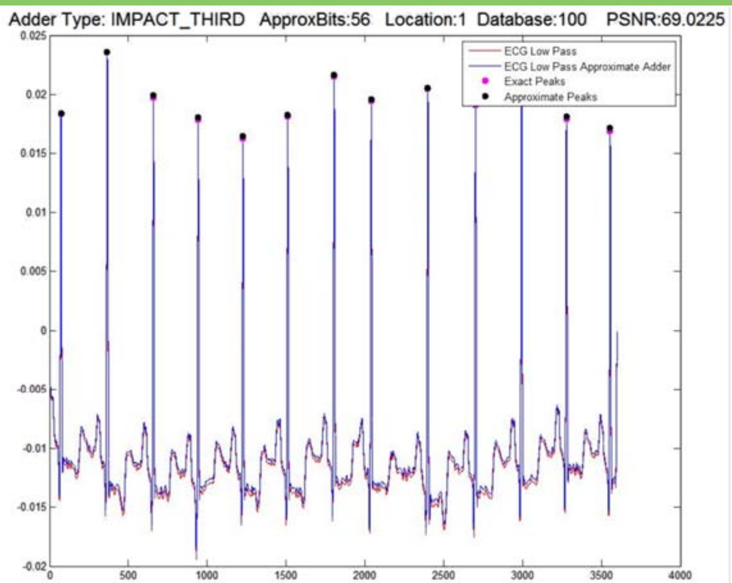
- None of the inexact designs missed a QRS signal
- Able to achieve good output with up to 15% power

Design	Power savings (%)	Area savings (%)	Relative Error (%)	Number of QRS signals missed
Exact design	0	0	0	0
Pareto optimal 1	12.6	4.5	0.18	0
Pareto optimal 2	14.6	4.9	0.96	0
Pareto optimal 3	12	5.3	1.92	0
Pareto optimal 4	12.8	5.9	3.03	0

QRS Peak detection with app. adders



QRS Peak detection with app. adders



Limitations and future works

- The error in estimation increases with the number of components although the trend remains the same – have better heuristics for estimation
- Heuristic for automated Pareto point selection rather than human input
- Designing co-efficient specific components for filters

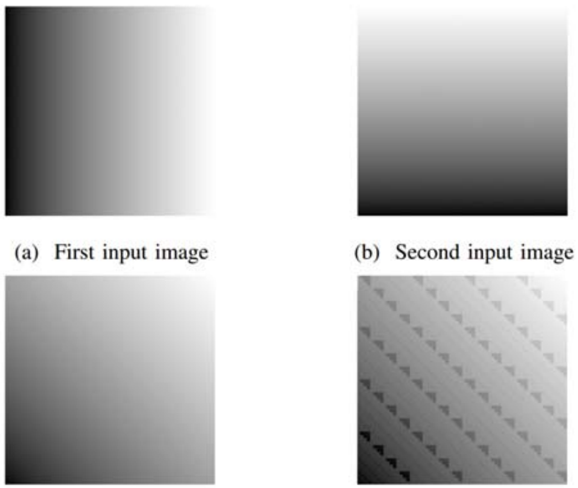
Conclusions

- Proposed overall design flow for constructing inexact systems using individual modules
- Heuristics to reduce search space
- Quick estimation of overall design parameters including relative error
- Case study with QRS detection flow shows the effectiveness of the overall design flow

Challenges

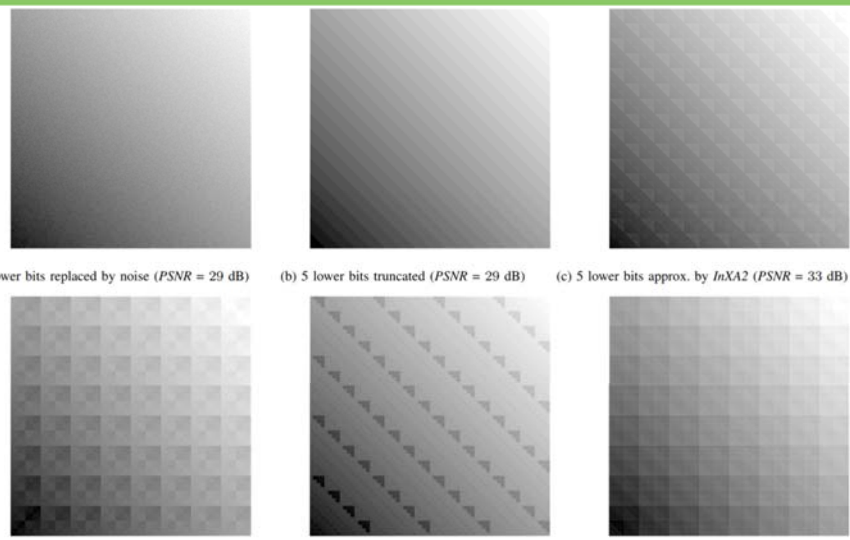
- Determine the precision
- Application designers are the best approximators!
- Defining the approx. metric for an application
- Which level to apply? Across the stack? Need a whole flow compatible with existing tools
- Run-time variation of the accuracy in the flow
- H/w support necessary
- Runtime reconf. approx. hardware
- Can we use the remaining hardware?

Approx. Addition of images



(a) First input image
(b) Second input image
(c) Exact result of image addition
(d) Image addition performed using an approximate adder configuration from [9]

Approx. Addition Result



(a) 5 lower bits replaced by noise (*PSNR* = 29 dB)
(b) 5 lower bits truncated (*PSNR* = 29 dB)
(c) 5 lower bits approx. by *InXA2* (*PSNR* = 33 dB)
(d) 5 lower bits approx. by *InXA1* (*PSNR* = 27 dB)
(e) approximated by *GeAr1* (*PSNR* = 28 dB)
(f) 5 lower bits approx. by *InXA3* (*PSNR* = 33 dB)

Summary

- Modern device/system level challenges forces us to rethink the design principles
- Approximate Computing is not new, but surely opens a new door
- Various mechanisms in various layers proposed to address the challenges and save power
- Can be applied at all levels, but the higher the layer, the bigger the gains

Questions and Answers

Email: *akash.kumar@tu-dresden.de*