Using artificial neural networks for error detection in unreliable computations

Vassilis Vassiliadis, Konstantinos Parasyris, Christos D. Antonopoulos,
Spyros Lalis, Nikolaos Bellas
University of Thessaly,
Volos, Greece
{**vasiliad**, koparasy, cda, lalis, nbellas}@uth.gr

Unreliable Computing

Unwanted causes

- Issues during the process of fabrication (e.g. variation)
- Novel technology which has not matured enough to truly be reliable
- Radiation
- Component aging

Deliberate Unreliable Computing

- Under-volting for improved energy/power efficiency
- **Over-clocking for improved performance**

Unreliable computing

Recipe

1. Opt for Significance-Aware Unreliable Computing
2. Detect errors before they propagate to the program output
3. Correct errors upon detection

Significance-Aware Unreliable Computing

Computations within a program are not equally important with respect to its final output quality.

The least important/significant parts of a program can be executed unreliably to improve performance (or energy/power efficiency) through CPU over-clocking (or CPU under-volting)

Error Detection

1. Domain expert wisdom
2. Patterns in the intermediate outputs
   - Artificial Neural Networks are pretty good at discovering patterns in data

Error Correction

Borrow a page from Approximate Computing.

Correct errors via:

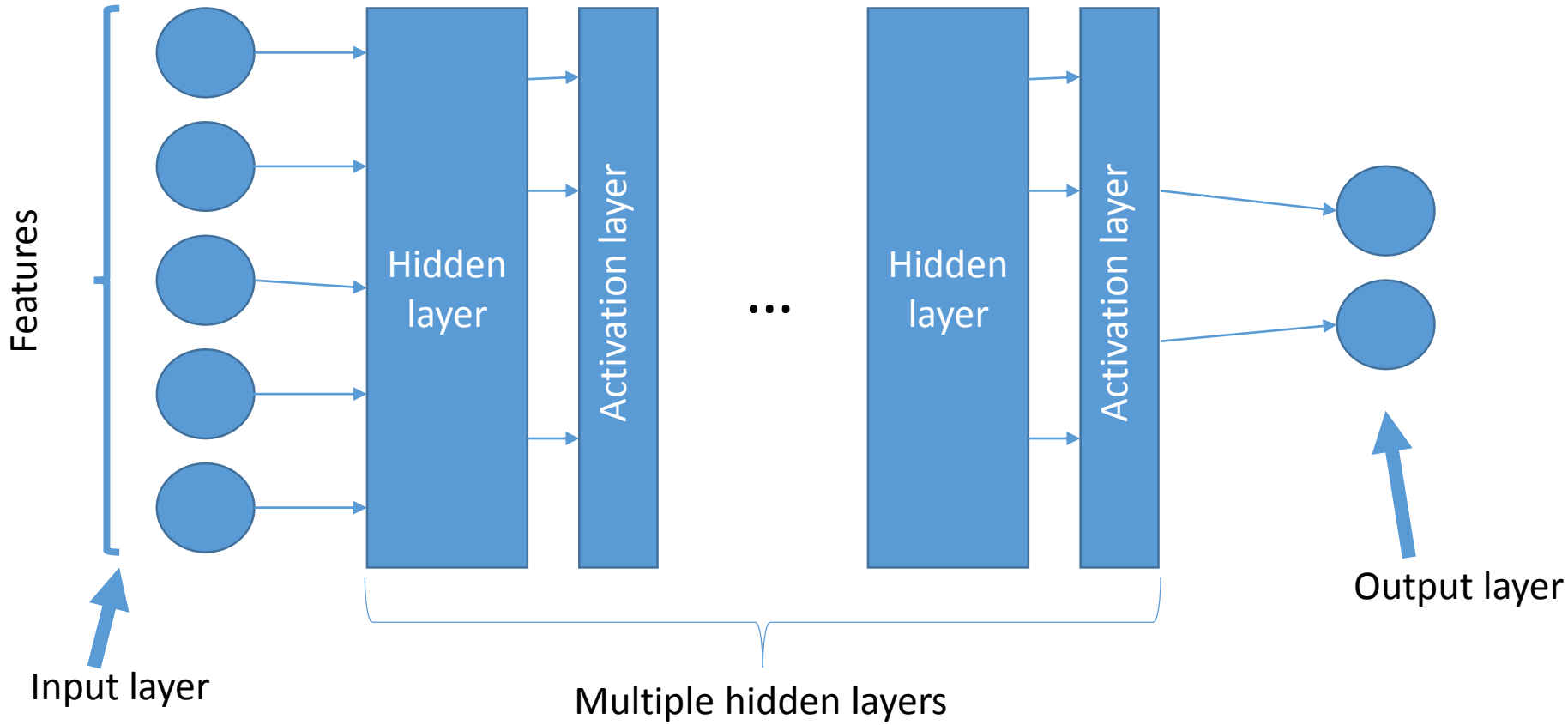
1. Re-execute the code in a reliable way
2. Approximate alternative implementations of the unreliably executed code
3. Default values

Execution cost and resulting quality DECREASE

Artificial Neural Networks 101

Artificial Neural Networks (ANNs) are great in identifying patterns in data. They even out-perform humans in certain problems.

Artificial Neural Network breakdown



This work focuses on ANNs for error detection

Algorithm 1: Error Detection using Artificial Neural Networks

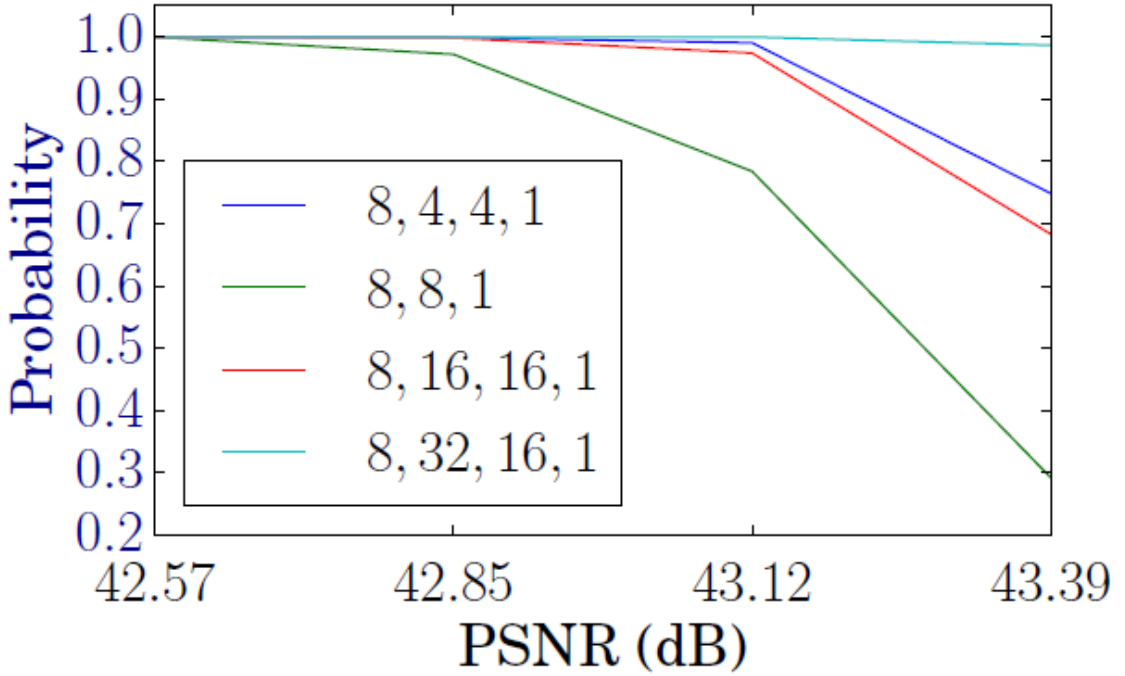
Input : Application source code, Src
Output: Hardened application source code

S1: Src_{tasks} = partitionCodeIntoTasks(Src)
S2: $outputs$ = performSoftwareInjection(Src_{tasks})
S3: $outputs_{labeled}$ = label($outputs$)
S4: $outputs_{normalized}$ = normalize($outputs_{labeled}$)
S5: $ANNS$ = generate()
S6: $ANNS_{trained}$ = train($ANNS$, $outputs_{normalized}$)
S7: $Functions$ = convertANNtoC($ANNS$)
S8 Src_{batch} = groupTasksInBatches(Src_{task})
S9: Src_{hardnd} = attachANNFnc(Src_{batch}, $Functions$)

Key challenges

- Choosing the input features
- Pre-processing data
  - Faulty values ?
  - Data balancing ?
  - Normalization ?
- Which ANN configuration is the best?
- When should the training phase stop?
- How are the ANNs going to be used to classify unreliable execution as “good” or “bad”?
  - “Executing” the ANN must be much faster than simply running the code unreliably

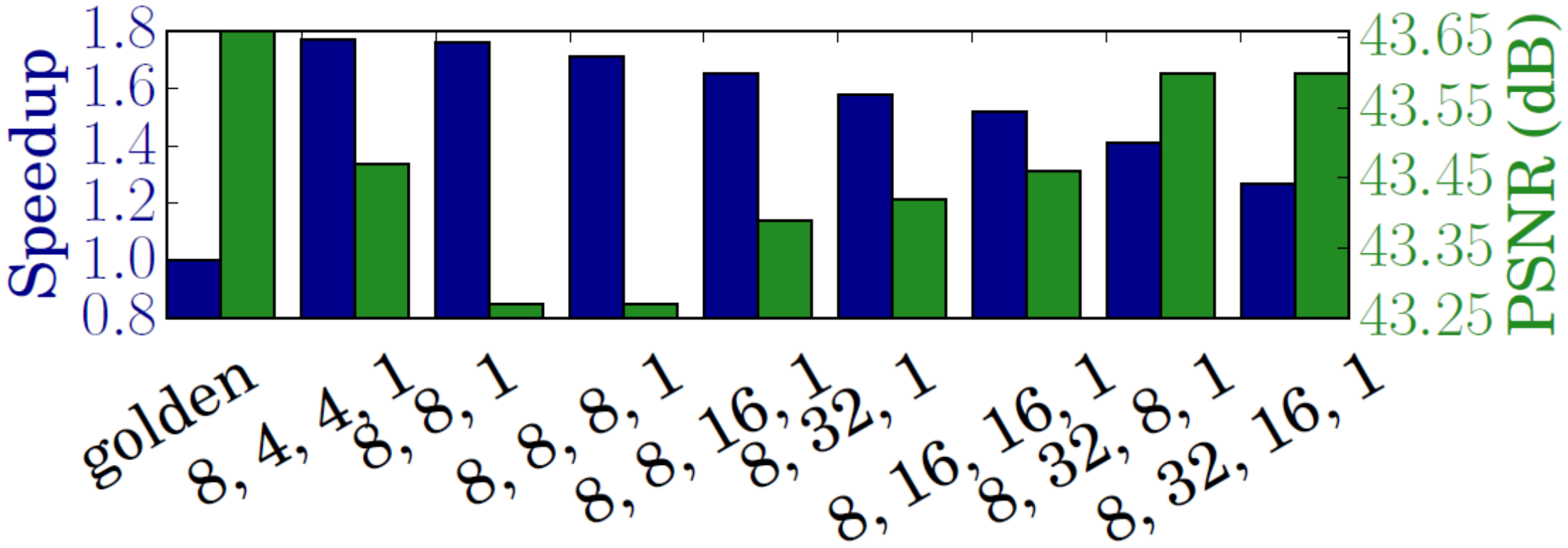
Running example DCT



The probability that an ANN results in images with at least a specific PSNR value. The first (8) and last layer (1) of each ANN corresponds to the input and output layers respectively. The ANNs are listed in decreasing execution time see Figure 2 for a more detailed speedup, output quality comparison

An error free execution produces a 43.67 dB output

Speedup vs Output quality



Best Artificial Neural Network for DCT

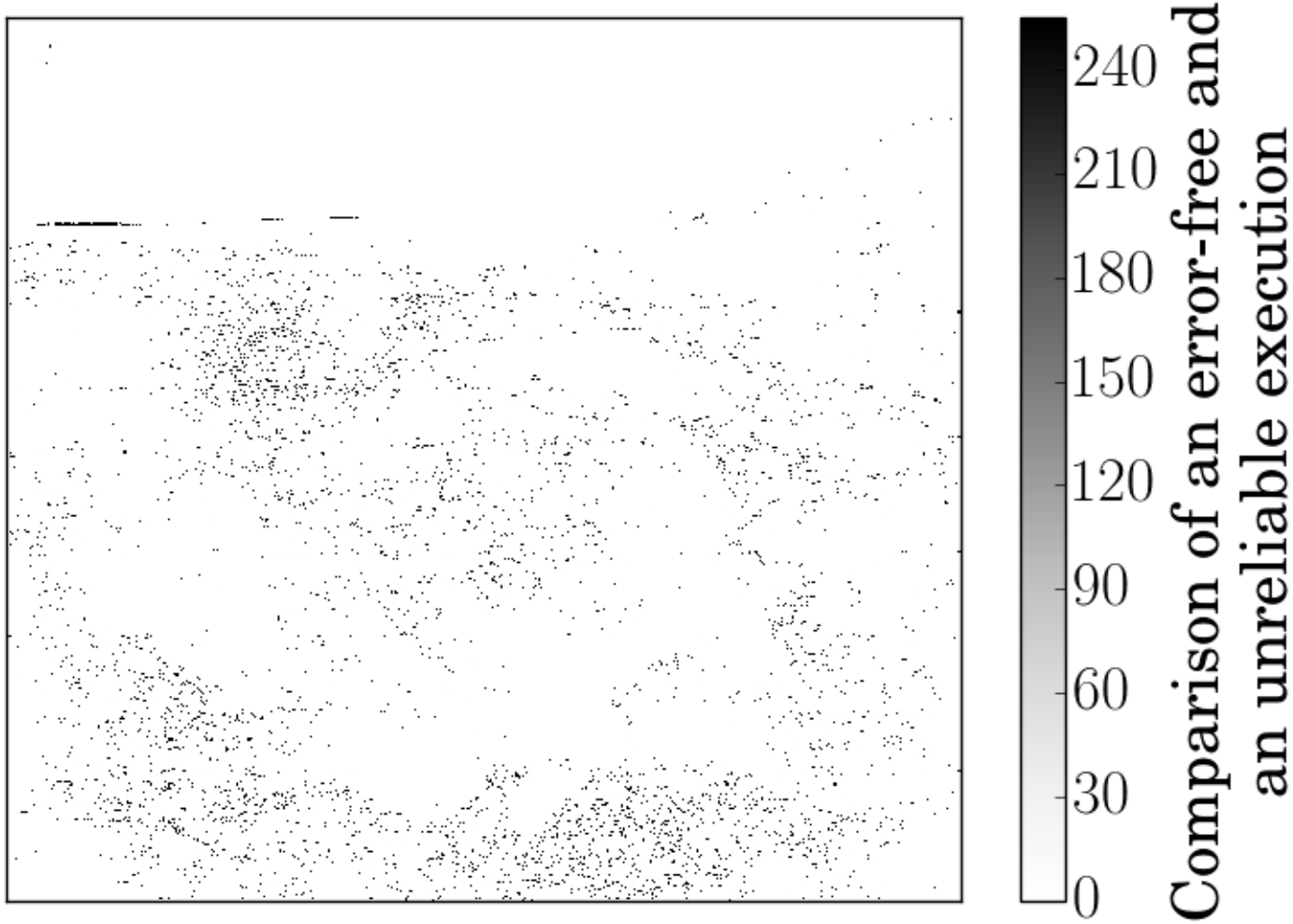
Runtime overhead	Reliable task execution	Unreliable task execution	Error detection & correction
2.9 %	19.1%	61.3%	16.7%

Speedup 1.77x

Quality degradation: PSNR down to 43.49 dB from 43.67 dB

Can you tell the difference?

SDCs even though an ANN was used to classify outputs as “good” and “bad”



Questions?

My e-mail is vasiliad@uth.gr