Implementing Approximate Computing Techniques by Automatic Code Mutation

Domenico Amelino, Mario Barbareschi, Antonino Mazzeo, Antonio Tammaro and Alberto Bosio
Email: bosio@lirmm.fr

WAPCO 2017, Stockholm, Sweden
January, 25th 2017

DIPARTIMENTO DI INGEGNERIA ELETTRICA E DELLE TECNOLOGIE DELL'INFORMAZIONE
UNIVERSITA' DEGLI STUDI DI NAPOLI FEDERICO II
LIRMM

Motivation

The Era of Approximate Computing

- Energy-efficiency, computational speed, low-overhead implementation are paramount concern in digital system design:

Heavy Processing

- Big-data;
- High-definition multimedia;

Performance and Cost

- Energy Vs. Speed;
- General Vs. Special Purpose;

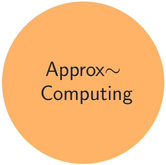
- has been becoming extremely hard to trade-off such features;
  - Literature is introducing post-Moore's Law era, non Von Neumann architectures, and so on...
- Even though... a *perfect* result is often not necessary:
  - an approximate of a less-than-optimal result is sufficient;
  - approximation opens the opportunity to deal with tight performance constraints;

Outline

Outline for Introduction

Approximate Computing in a Nutshell

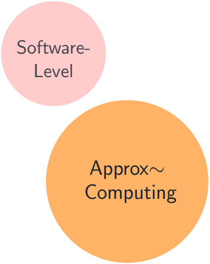
Towards the trading performance off for the quality



- Rather than the best possible result, approximate computing exploits inaccurate outputs to outperform classical elaboration approaches:

Approximate Computing in a Nutshell

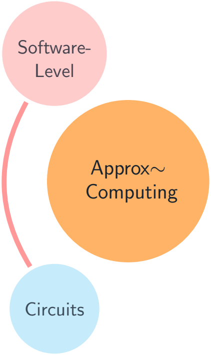
Towards the trading performance off for the quality



- Rather than the best possible result, approximate computing exploits inaccurate outputs to outperform classical elaboration approaches:
  - by altering software execution;
    - e.g. loop perforation;

Approximate Computing in a Nutshell

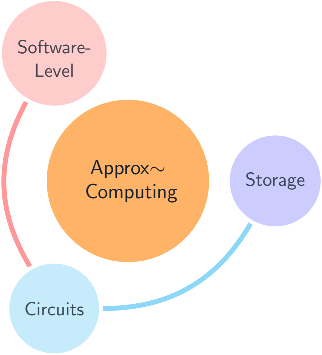
Towards the trading performance off for the quality



- Rather than the best possible result, approximate computing exploits inaccurate outputs to outperform classical elaboration approaches:
  - by altering software execution;
    - e.g. loop perforation;
  - by employing inexact custom circuits;
    - e.g. speculative arithmetic operations;

Approximate Computing in a Nutshell

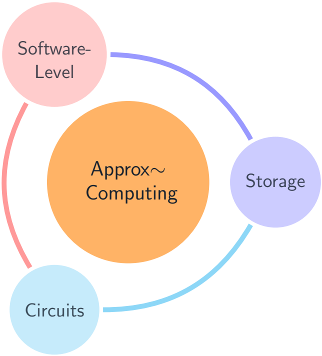
Towards the trading performance off for the quality



- Rather than the best possible result, approximate computing exploits inaccurate outputs to outperform classical elaboration approaches:
  - by altering software execution;
    - e.g. loop perforation;
  - by employing inexact custom circuits;
    - e.g. speculative arithmetic operations;
  - by approximately storing data;
    - e.g. DRAM with lower refresh rate.

Approximate Computing in a Nutshell

Towards the trading performance off for the quality



- Rather than the best possible result, approximate computing exploits inaccurate outputs to outperform classical elaboration approaches:
  - by altering software execution;
    - e.g. loop perforation;
  - by employing inexact custom circuits;
    - e.g. speculative arithmetic operations;
  - by approximately storing data;
    - e.g. DRAM with lower refresh rate.

Approximate Computing

- Several domains may benefit from approximation:
  - signal processing: image, video, audio, speech;
  - machine learning,
  - data mining;
  - search;
  - ...
- In the past, Approximate Computing was often considered an effect of the “human interpretation”:
  - the literature has demonstrated the crucial role of the **inherent application resiliency**.

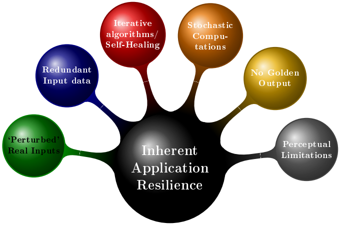
The Inherent Application Resiliency

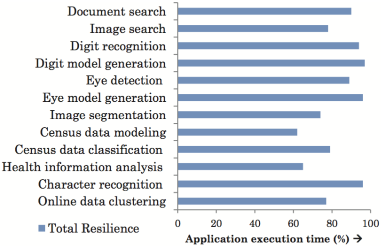
- The Inherent Application Resiliency is a property for an algorithm to return **acceptable outcomes** despite some of its inner computations being **approximate or imprecise**;
- Approximate Computing exploits a design approach that **leverages the inherent resiliency** through optimizations which **trade the outputs quality off for enhanced performance**, such as time, energy consumption, occupied area, and so on;

The Inherent Application Resiliency

Main sources

- The property mainly inherits from:





83% of run time spent in resilient computations

1

[1] Venkataramani, S. et al. Approximate computing and the quest for computing efficiency. 52nd Annual Design Automation Conference

Approximate Design... Possible?

A new design parameter

- Establish how much approximate;
- Trade resources saved off for inaccuracy.

Integration in classical flow

- Algorithms, tools, languages, models ...
- Ensemble of Approximate Computing techniques.

Outline for Research Effort

Precimonious

Tuning Assistant for Floating-Point Precision

- It is an automatic tuning tool for the IEEE 754 floating point types of C/C++ code by means of annotation that gives the user-desired accuracy;
- It uses the LLVM IR (so it works on compiled codes) to generate and evaluate different configurations using an algorithm based on the delta-debugging;
- The results are in the form of LLVM IR and they must be manually mapped to the original source code, in order to give the actual result to the programmer;
- The search algorithm is based on the delta-debugging with the adoption of an heuristic pruning technique.

ACCEPT

A Programmer-Guided Compiler Framework for Practical Approximate Computing

- Adaptation for C/C++ of EnerJ (from the same authors) which works in the Java environment:
  - It aims to help the programmers to *approximate* their own code, because authors claim automatic approaches loses practicality and controllability;
  - It supports the main approximation techniques such as loop perforation or synchronization relaxation;
  - It works on the LLVM Intermediate Representation, propagating the source code annotations to manage the *real* approximation at this level;
- To support the new dialect it introduces a new compiler (enerc and enerc++) for the LLVM infrastructure;
- It considers approximation effects which are linear! There is no a proper exploration algorithm.

REACT

A Framework for Exploration of Approximate Computing Techniques

- It extends ACCEPT, using it to *intercept* at LLVM IR level the instructions that has been marked as *approximate*;
- This hook mechanism is used to redirect those instructions to *user-defined* approximate techniques that they have implemented to quickly compare their effect;
- A simple linear energy model is presented in order to estimate the energy savings available through approximation.

Outline for IDEA Tool

Mutate And Explore

- Available tool and methodologies do not approach Approximate Computing with a general technique:
  - dependency to the language target;
  - software or hardware;
  - handling of only one technique or few variants of one;
  - quality/error estimation is not flexible;
  - access to the source code or customization of the tool.
- We tried to deal with such problems by introducing a new methodology and a corresponding tool:
  - The novelty stands in make an algorithm approximate by mutation;
  - The whole approach is totally user-defined.

Towards Automatic Exploration of Variants

Hypotheses

- The error is directly related to the **magnitude of the approximation** applied on involved operations;

Towards Automatic Exploration of Variants

Hypotheses

- The error is directly related to the **magnitude of the approximation** applied on involved operations;
- The application tolerates at most a **certain error threshold**;

Towards Automatic Exploration of Variants

Hypotheses

- The error is directly related to the **magnitude of the approximation** applied on involved operations;
- The application tolerates at most a **certain error threshold**;
- The error cannot be *universally* defined.

Towards Automatic Exploration of Variants

Hypotheses

- The error is directly related to the **magnitude of the approximation** applied on involved operations;
- The application tolerates at most a **certain error threshold**;
- The error cannot be *universally* defined.

Requirements

- *Tuning* process over every approximable inner operation;

Towards Automatic Exploration of Variants

Hypotheses

- The error is directly related to the **magnitude of the approximation** applied on involved operations;
- The application tolerates at most a **certain error threshold**;
- The error cannot be *universally* defined.

Requirements

- *Tuning* process over every approximable inner operation;
- **Customizable** quality function with respect to the original version;

Towards Automatic Exploration of Variants

Hypotheses

- The error is directly related to the **magnitude of the approximation** applied on involved operations;
- The application tolerates at most a **certain error threshold**;
- The error cannot be *universally* defined.

Requirements

- *Tuning* process over every approximable inner operation;
- **Customizable** quality function with respect to the original version;
- **Controllable** amount of error;

Towards Automatic Exploration of Variants

Hypotheses

- The error is directly related to the **magnitude of the approximation** applied on involved operations;
- The application tolerates at most a **certain error threshold**;
- The error cannot be *universally* defined.

Requirements

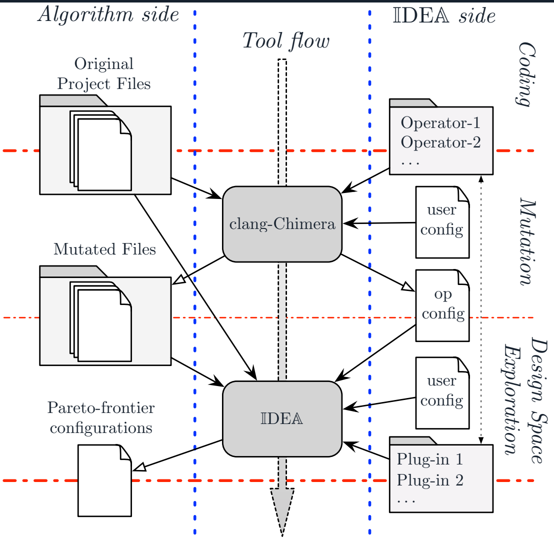
- *Tuning* process over every approximable inner operation;
- **Customizable** quality function with respect to the original version;
- **Controllable** amount of error;
- **Automatic** exploration of approximate configurations.

Approximate C/C++ Algorithms

We have got an 𝕀DE𝔸!

- We devised 𝕀DE𝔸 (**I**IDEA**A** **I**s a **D**esign **E**xploration tool for **A**pproximating **A**lgorithms), a novel approach which explores possible inaccuracies that can be applied on an algorithm written in C/C++:
  - 𝕀DE𝔸 integrates clang-Chimera, which is a source-to-source transformation tool, which can be configured with customizable transformation rules;
  - the generation of approximate *mutants* is: fully automatic, generic (ideally, 𝕀DE𝔸 would implement every proposed Approximate Computing technique), and is runnable in software;
- 𝕀DE𝔸 explores design solutions with a B&B approach:
  - Solutions define inherently a **Pareto frontier**;
  - They can be used to configure both <u>hardware and software</u> algorithms;
- 𝕀DE𝔸 tools are open source and can be freely downloaded and extended.

IDEA Flow


Algorithm side
Tool flow
IDEA side
Coding
Mutation
Design Space Exploration
Original Project Files
Operator-1
Operator-2
...
clang-Chimera
user config
Mutated Files
op config
user config
IDEA
Pareto-frontier configurations
Plug-in 1
Plug-in 2
...

Clang-Chimera

- Clang-Chimera is part of the **Chimera Tools**, a set of tools for mutating C/C++;

Clang-Chimera

- Clang-Chimera is part of the **Chimera Tools**, a set of tools for mutating C/C++;
  - Free available (GPLv3) at: https://git.io/vKOZK

Clang-Chimera

- Clang-Chimera is part of the **Chimera Tools**, a set of tools for mutating C/C++;
  - Free available (GPLv3) at: https://git.io/vKOZK
- It is implemented using the Chimera Design, that is based on the *Mutation Template* concept, an efficient and flexible way to generate mutants from an abstract syntax tree (AST) representation of the source code;

Clang-Chimera

- Clang-Chimera is part of the **Chimera Tools**, a set of tools for mutating C/C++;
  - Free available (GPLv3) at: https://git.io/vKOZK
- It is implemented using the Chimera Design, that is based on the *Mutation Template* concept, an efficient and flexible way to generate mutants from an abstract syntax tree (AST) representation of the source code;
- The idea is to avoid unnecessary AST visits to produce multiple mutants of the same target file, so a *mutation template* can be seen as a "list" of locations on the AST, in which it will be necessary to apply some kind of mutations.

IDEA Supporting Loop-Perforation

- Let us consider two well-known Approximate Computing techniques:

Loop First

```
for(i = 0; i < n; i += stride){
   body
}
```

Loop Second

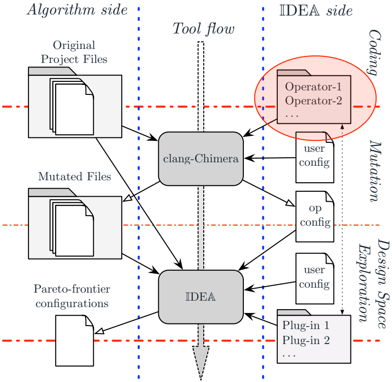
```
for(i = 0; i < n; i ++){
   if(i % stride != 0)
      body
}
```

- They are defined to *skip* some iterations of a loop, aiming at save time execution worsening the output result;
- Both of them requires the definition of the *stride* value.

IDEA Supporting Loop-Perforation

Defining Operators in clang-Chimera

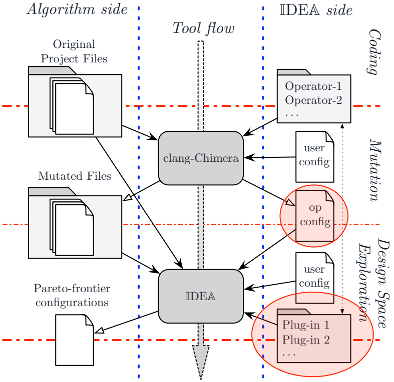
- In order to define a new mutator (or *operator*), the user needs to *inherit* the mutator class and has to define 3 methods:
  1. `getStatementMatcher`: coars-grain match over the AST;
  2. `match`: fine-grain match, which prepares variables for mutate;
  3. `mutate`: modifies involved AST nodes;



- W.r.t. loop performation, the first method looks for the *forStm* nodes, while the others applies the mutation whenever there are proper initialization and termination conditions.

IDEA Supporting Loop-Perforation

Defining IDEA plugins



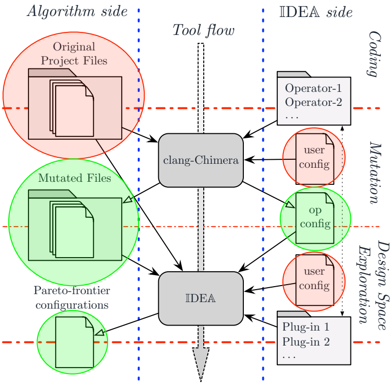
- IDEA has to be configured with proper plug-ins which are able to
  1. handle mutated source files and configurations;
  2. modify mutation parameters (e.g. strides), accordingly to the approximate computing technique;

- Once the mutators and the plug-ins are defined, the tool-flow is able to explore approximate variants of an arbitrary source code written in C/C++.

IDEA Supporting Loop-Perforation

Getting approximate variants

- In order to mutate a source code, the user has to provide
  1. the source code itself;
  2. a configuration file for clang-Chimera reporting target C/C++ functions and the list of operators that have to be applied to the source code;
- The output is the mutated source code and a report, which contain the list of mutated files and the operators applied.



- IDEA uses this file together with the error/quality function, provided by the user, to evaluate each approximate variant.

Outline for Result

Loop Perforation

- As last step, we use two algorithms as target of the IDEA flow:

Taylor series expansion

- Expansion of function $e^x \cdot ln(1 + x)$;
- Function evaluated up to the 250 power;
- Error calculated over 10^4 points as absolute difference with an oracle.

Inverse Discrete cosine transformation

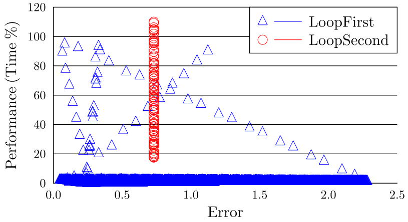
- Decompression of 10 gray scale images;
- 8x8 pixels block;
- Error computed as average PSNR between original and decompressed images.

Loop Perforation

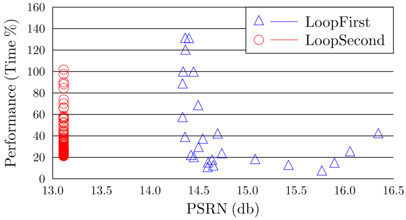
Experimental evidences

- As last step, we use two algorithms as target of the $\mathbb{IDEA}$ flow:

- Taylor required the exploration of 10^6 variants for the LoopFirst and 10^5 for the LoopSecond.



- IDCT required the exploration of 200 variants for the LoopFirst and 3000 for the LoopSecond.



Focus on the IDCT

Original DCT Compression Vs. IDEA Configuration

- PSNR: 14.637

Outline for Conclusion

Conclusion and Remarks

- IDEA is an extensible Approximate Computing tool as it relies on the concept of the mutation code, performed by clang-Chimera tool:
  - It is able to handle any kind of approximate computing technique;
  - the mutation algorithm is written in C/C++;
  - IDEA provides Pareto-solutions that can be use for both hardware and software projects;
- We proved the approach with experimental result and we also demonstrated its feasibility by case studies.

Implementing Approximate Computing Techniques by Automatic Code Mutation

Domenico AMELINO, Mario BARBARESCHI, Antonino MAZZEO, Antonio TAMMARO and Alberto BOSIO
Email: bosio@lirmm.fr

WAPCO 2017, Stockholm, Sweden
January, 25th 2017

DIETI. UNIVERSITA' DEGLI STUDI DI NAPOLI FEDERICO II
DIPARTIMENTO DI INGEGNERIA ELETTRICA E DELLE TECNOLOGIE DELL'INFORMAZIONE
LIRMM