Implementing Approximate Computing Techniques by Automatic Code Mutation

Domenico Amelino, Mario Barbareschi,
Antonino Mazzeo, Antonio Tammaro
DIETI, University of Naples Federico II
Via Claudio, 21 - 80125 Naples (NA), Italy
Email:{firstname.lastname}@unina.it

Alberto Bosio
LIRMM, University of Montpellier
161 rue Ada, 34095 Montpellier Cedex 05, France
Email: bosio@lirmm.fr

***Abstract*—Approximate computing has emerged as one of the most important breakthrough in many scientific research areas. It exploits the inherent tolerance of algorithms against computational errors in order to outperform the original versions by worsening the result quality. The research community demonstrated the effectiveness of the trade-off between accuracy and performance, such as energy consumption, time and occupied area for integrated circuits, and many approximate computing methodologies were proposed. Unfortunately, introduced approaches fit in specific application domain and a general and systematic methodology to automatically define approximate algorithms is still an open challenge. Bearing in mind such a lack of generality, in previous works we introduced a methodology which makes use of software mutation to obtain approximate versions of a software defined algorithm. Based on this concept, we developed $\mathbb{I}$DE$\mathbb{A}$, a design exploration tool that relies on a source-to-source manipulation technique, implemented by an open-source tool called clang-Chimera, in order to apply code transformations that approximate the computation of a C/C++ algorithm. In this paper, we detail the methodological flow introduced by $\mathbb{I}$DE$\mathbb{A}$ and we describe every step needed to have available any approximate computing technique by a walk-through. Furthermore, we provide experimental results which highlight the effectiveness of the approach, in particular by searching for approximate variants applying the loop perforation technique.**

I. Introduction

The Approximate Computing term has been introduced for indicating a design paradigm that implements efficient hardware circuits and software components by tolerating inexact outputs. It has demonstrated by the literature the effectiveness of imprecise computation for efficient design of software codes or hardware components that implement inexact algorithms thanks to their inherent resiliency. This property characterizes algorithms such that they return acceptable outcomes despite some of its inner computations being approximate or imprecise [1], [2]. The inherent resiliency is prevalent for domains in which outputs of algorithms have to be interpreted by humans, such as digital signal processing of images and audio, but also data analytics, web search and wireless communications exhibit an equivalent property [3], [4], [5].

Mainly, the Approximate Computing design exploits approaches that leverages inherent resiliency through optimizations which trade outputs quality off for better performance, such as time, energy consumption, occupied area, and so on [6]. As for the result quality, several metrics can be adopted to estimate the loss of precision with respect to the exact, or generally the best, result.

Many research papers in the literature explored the possibilities given by the Approximate Computing, as well as they introduced new methodologies to automatically define best trade-off configurations between result quality and performance. However, there are still open challenges that hold back Approximate Computing for a wider employment, especially for the implementation of custom Integrated Circuits (ICs). In particular, the key point is a lack of a general design space exploration method that searches for possible variants of an algorithm w.r.t. the grade of the inaccuracy.

Most of the proposed techniques try to define new methods to generate alternative versions of specific operations with less resources. For instance, there are several proposals of approximate arithmetic operations [7], [8], [9]. Such variants differ from speculative implementations, because are not focusing on generate alternatives and restoring the possible introduced error [10], [11], [12]. Other techniques generate variant by considering a high level description of the algorithm or their implementation at low-level [13], [3].

Aiming at define a general method for considering arbitrary approximate computing techniques, we developed $\mathbb{I}$DE$\mathbb{A}$, first presented in [14], which is a design exploration tool able to find approximate versions for an algorithm, coded in C/C++, by mutating the original code. $\mathbb{I}$DE$\mathbb{A}$ employs an extensible source-to-source transformation tool, the clang-Chimera, which analyzes the Abstract Syntax Tree (AST) to apply user-defined code mutators. Being extensible, clang-Chimera gives the possibility to implement any approximate computing technique.

In this paper, we describe the methodology introduced by the $\mathbb{I}$DE$\mathbb{A}$ approach and, to this aim, we detail the extension of clang-Chimera with an approximate computing technique. In particular, we implement the loop perforation technique and we give some experimental results.

The remainder of the paper is structured as follows. In Section II we briefly report the state of the art of approximate computing tools available in the literature. In Section III we describe the mutation code approach for obtaining approximate variants, while Section IV illustrates some experimental

results. Section V draws the conclusion of this manuscript.

II. Related Work

In the literature, several papers presented Approximate Computing methodologies and tools, such as ACCEPT [3] and its extension REACT [15]. ACCEPT is an adaptation for C/C++ of EnerJ [16], which has been designed to work in the Java environment. ACCEPT introduces an extension of the C/C++ language to provide programming annotations.

Such tools give to designers a programmed guided technique for the inner-implemented approximation techniques. The implementation of ACCEPT involves the LLVM Intermediate Representation (IR) [17], providing a modified compiler to support the language extension. REACT uses a compiler-infrastructure to create a framework for the rapid exploration of well-known approximate computing techniques, proposing an energy model to evaluate the actual power saving for different approaches. REACT extends ACCEPT, so it exploits the LLVM IR to manipulate algorithms written in C/C++.

The ABACUS tool, described in [13], directly manipulates hardware circuits, which are coded in Hardware Description Languages (HDL). It adopts a greedy algorithm to perform a design exploration to find Pareto-optimum solutions with a trade off between accuracy and power consumption.

Authors of [18] introduced Precimonious, which provides an automatic tuning tool for IEEE 754 Floating Point standards. Indeed, Precimonious uses the LLVM IR to perform a float-type exploration by means of annotations. This way allows the user to specify the maximum loss in accuracy.

As for the approximation of operations which involve numeric types, authors of [19] introduced a power-aware design methodology by exploiting a word-length optimization. They proposed a tool, called PowerCutter, to dynamically analyze the range of variables and reduce the precision of arithmetic operations for C/C++.

In [20] the authors illustrated the CMUFloat, a C++ library, that redefines operations for integers, shorts and floating point types in order to simulate reduced bit-width. They used the library to simulate digital signal processing units realized in an approximate manner.

III. Extensible Design Exploration for Approximating Algorithms

Existing approximate computing tools consider specific transformations and specific domains, ss previously discussed in sections II, . Furthermore, existing tools mostly are not fully automatic and provide a guided approach for approximation. Conversely, we aim to define a general approach considering the following observations:

1) the error introduced by approximation strictly depends on the algorithm and on the application, hence it has to be specifically defined by the designer;
2) Approximate Computing techniques have to be automatically applied by analyzing the algorithm, in order to generate approximate variants of it;

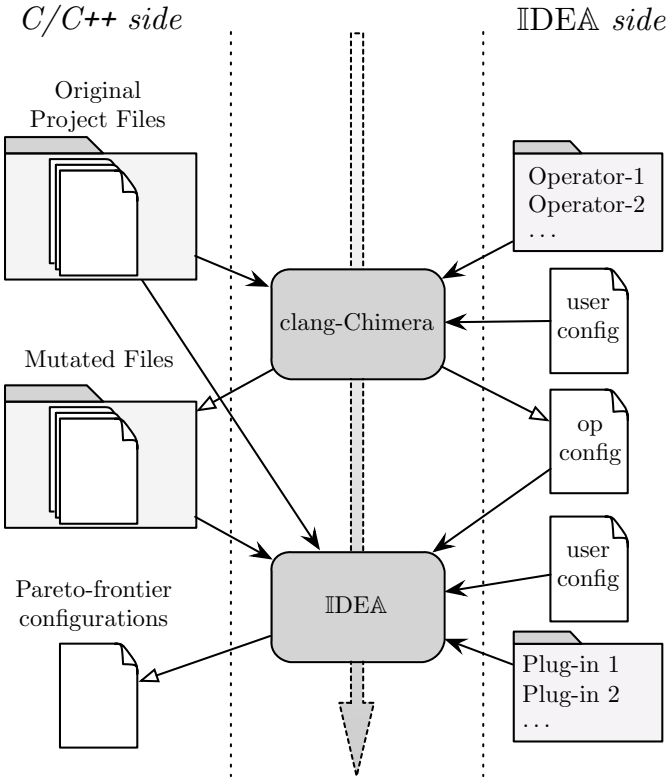

Fig. 1. Execution flow of 𝕀DE𝔸 tool.

3) each variant has to be characterized by a quality value w.r.t the original output; in particular, generated variants should be as close as possible to an error threshold defined by the designer;
4) each Approximate Computing technique has to be applied on the original code of the algorithm, in order to execute each variant in software.

Estimation of the error introduced by the approximation is a crucial operation. Indeed, whenever it is estimated by considering differences between the outcomes of the original algorithm and the approximate versions, the user has to define significant input to make it an effective value. Moreover, the error definition is itself not a trivial task, as many metrics can be employed to evaluate how far are approximate results from the original one.

The exhaustive search of every realizable approximate version is not feasible since the number of combinations explodes very soon, even taking into account a limited subset of Approximate Computing approaches. Furthermore, these approximate versions have to be evaluated in terms of performance, in order to establish the best trade-off between approximation and performance gain.

A. How 𝕀DE𝔸 works

The 𝕀DE𝔸 workflow, introduced in [14], [21], is illustrated in Figure 1. The process considers algorithms encoded in C/C++ language: this way the original algorithm and its approximate variants can be easily simulated in software.

As one can notice, the entire process involved 2 tools: clang-Chimera and IDEA.

Clang-Chimera tool is a mutation engine for C/C++ code based on the Clang compiler and written in C++. It is provided as a framework to rapidly develop source-to-source transformations on C/C++ code by means of Clang facilities, such as `ASTMatcher` and `Rewriter`. Clang-Chimera borrows the terminology from the mutation testing technique and introduces the concept of mutator and mutation operator. A mutator encapsulates a specific transformation logic on the C/C++ code and it is is composed of **matching rules**, the rules to match specific AST patterns, and **mutation rules**, the rules to modify the matched AST patterns. Chimera adopts the AST to analyze and manipulate the source code of a given algorithm. Indeed, the AST is a tree-based representation and each node (AST node) of the tree denotes a language construct of the analyzed code. An ensemble of AST Nodes defines an AST pattern, which corresponds to a specific structure of the code. A mutation rule modifies an AST pattern that matches a specific matching rule. A mutation operator, or simply operator, is a collection of mutators and provides a facility to group up related mutators.

The most important aspect of the tool is the ease of extension. Indeed, to extend clang-Chimera it is required the inheritance from the `Mutator` class to create new mutators and new `Operator` objects which collect newly created mutators. This way, clang-Chimera can integrate any Approximate Computing technique by defining mutation operators. Moreover, for the Approximate Computing technique in which the mutations are isolated onto the AST (and hence in the code), clang-Chimera allows to define a so called FOM operator (First Order Mutation). Differently, a High Order Mutation (HOM) operator has to be used when the Approximate Computing techniques has to accumulate mutations onto the AST before obtain the proper approximate code.

Once the operators are defined and configured by the user, clang-Chimera is able to perform mutations over a C/C++ project. The outputs of clang-Chimera (as reported in Figure 1) are: (i) C/C++ project files mutated by configured operators; (ii) a configuration file which reports what has been mutated w.r.t. the original files and which kind of operator has been applied.

The second tool in the flow is IDEA, which performs a branch and bound (B&B) exploration. The design space exploration is accomplished by manipulating the approximation grades of an applied approximation technique. Indeed, IDEA internally compiles the target variant of the algorithm and, thanks to the information contained in the clang-Chimera reports, is able to modify its in-memory code to enable a particular approximation technique and to modify the approximation grade of such technique. Each step of the B&B approach uses a configuration incrementing the approximation grade of the Approximate Computing Technique or activating it. It follows that, at each iteration, IDEA generates an approximate version of the target algorithm, which exhibits an error greater than or equal to the error that has been observed in previous iterations. The error is estimated through an user-provided error function. Without loss of generality, we can claim that this function is a monotonic function of the approximation grade and it is equal to 0 when is executed over the original version of the algorithm.

The execution of the B&B proceeds as a depth-first-search in a tree, namely it explores as far as possible along each branch before back-tracking. Whenever a configuration is characterized by an error that crosses the threshold, it is pruned from the tree and a different Approximate Computing technique is considered. Once the exploration terminates, the leaves of the execution tree are Pareto-optimum configurations of the original algorithm, such that they cannot be further approximated without crossing the error threshold. It follows that the tree depth, and hence the execution time, is proportional to the threshold, while the tree breadth depends on how many Approximate Computing techniques are applied by clang-Chimera.

In order to manipulate a given Approximate Computing technique, IDEA needs a *plugin* which has the role of interpreting the outputs produces by an operator. Indeed, an operator implemented in clang-Chimera produces a report listing every possible location in which a mutation can take place.

B. Extending IDEA *with the Loop Perforation technique*

In order to give a complete example of IDEA set-up, in this paper we illustrate the development of two new operators, called *LoopFirst* and *LoopSecond*, which implement the loop perforation techniques described by Lee&Jain in [22]. The loop perforation works by reducing or skipping some cycles of an iteration, obtaining an approximate version with better time performance. In [22] the loop perforation is introduced in two different forms, hence we name the first operator *LoopFirst*, which applies the first described loop perforation technique, that is

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

while the second defined operator, named *LoopSecond*, applies the other technque, that is

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

These two operator are realized as HOM ones and they are realized in clang-Chimera by means of inheritance of the `Mutator` class. The inheritance requires to override three functions, namely the *getStatementMatcher*, *match* and *mutate*. The first function defines the matching rule with a coarse grain approach by using the AST-Matcher, which is provided by the Clang compiler suite. Conversely, the function *match* is responsible for the fine grain match. As the operators are defined as HOM, the function is invoked each time an AST node passes the coarse grain match. The main role of the match is to define temporary variables, which has their scope within the current node of the AST, useful to obtain the mutation on the code. Moreover, whenever the fine grain rule applies

a transformation, the concern of the coarse grain match is to invoke the textitmutate function. Hence, such function is actually invoked each time a mutation is successfully applied onto an AST node. The *mutate* function uses the Rewriter, provided by Clang compiler suite, and saves a report for each accomplished mutation.

Hence, in order to correctly implement a loop perforation technique, the function *getStamentMatcher* filters out every node which is not a child of a *forStmt* node onto the whole AST. As for the *LoopFirst*, the *getStamentMatcher* takes into account nodes which properly defines a termination condition and an increment statement, while the *LoopSecond* takes into account nodes which defines the init statement and the termination condition. These conditions, which are children nodes of a *forStmt*, are passed to the *match* function, which is responsible to modify the condition, accordingly to the operator that has to be applied. For each mutated loop, a new variable is added onto the AST, which is called *stride*. Such a variable assumes the same type of the variable involved into the for statement and is named with a unique number concatenated each time. For instance, the following loop:

```
for(i=0; i<n; i++)
```

after the mutation by the operator *LoopFirst*

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

otherwise, in case of *LoopSecond*:

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

Once a new mutator has been defined, clang-Chimera requires the registration of such mutator by invoking the *registerMutationOperator* function.

IV. Experimental Result

In this Section, we aim to demonstrate the efficacy of the methodology introduced by the IDEA approach by performing approximation on some algorithms. In particular, we target some mathematic functions to be evaluated through the Taylor series. To retrieve the value of a given function by using the Taylor expansion requires the summation of infinite terms. Such summation is nested within others in the case of multi-variable function. Such example significantly stresses the clang-Chimera ability to perform automatic mutation and the B&B approach implemented by IDEA.

We code in C/C++ the computation of the function $f(x, y) = e^x \cdot \log(1 + y)$ by iterating the expansion in Taylor series. Then, we run the flow described in Figure 1 to obtain the Pareto frontier in terms of approximate configurations applying the loop perforation technique. As for the quality of the output, we configure a function which calculates the average proximity of some real points for the function $f(x, y) = e^x \cdot \log(1 + y)$. The quality threshold is fixed to 3 for both loop perforation technique.

Running IDEA with the *FirstLoop* causes the exploration of about 1×10^6 variants and less than 1×10^4 are leaves of the B&B, i.e. Pareto configurations. Conversely, IDEA configured with the *FirstLoop* performs the exploration of about 1×10^5 variants and less than 1×10^2 are Pareto configurations. Both campaigns executes in less than 1 minute.

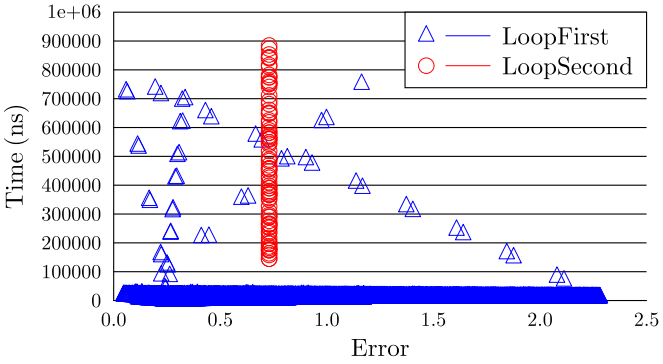

Fig. 2. Graphs of execution time over the error for each experiment executed by IDEA with both loop performation techniques

In Figure IV we report a graph with the error value and execution time of each experiments. As for the *LoopFirst* operator, most of the experiments (blue triangles) reside near the error axis, evidencing the inadequacy of the technique to generate configurations with lower time execution. Other solutions results give worse configurations in terms of trade-off.

As for the *LoopSecond* operator, the Pareto frontier is clearly visible as a vertical line, meaning that the technique is able to remove cycles which do not contributes to obtain better results, even though the best configuration is obtained by the *LoopFirst*.

V. Conclusion

So far, we demonstrated the feasibility of the IDEA tool in performing a design space exploration by taking into account Approximate Computing techniques and a target algorithm, coded in C/C++. To the best of our knowledge, this is the first attempt to approach Approximate Computing with a source-to-source transformation tool which makes use of mutation operators. To this aim, IDEA relies on clang-Chimera, a source-to-source transformation tool, which can be extended to support any Approximate Computing technique. By means of a B&B exploration approach, IDEA applies and accumulates approximations which are configured in clang-Chimera and for each obtained configuration it performs an error estimation, trimming ones which exhibit an error value greater than a configure threshold.

This paper fills the gap of the setup phase by detailing the implementation of a well-known Approximate Computing technique, which is the loop perforation. We provided an exhaustive description of what is required to implement a new operator, describing each required step. We also proved the efficacy of the loop perforation approach by running IDEA on a case study.

REFERENCES

[1] V. K. Chippa, S. T. Chakradhar, K. Roy, and A. Raghunathan, "Analysis and characterization of inherent application resilience for approximate computing," in *Proceedings of the 50th Annual Design Automation Conference*. ACM, 2013, p. 113.

[2] S. Venkataramani, S. T. Chakradhar, K. Roy, and A. Raghunathan, "Approximate computing and the quest for computing efficiency," in *Proceedings of the 52nd Annual Design Automation Conference*. ACM, 2015, p. 120.

[3] A. Sampson, A. Baixo, B. Ransford, T. Moreau, J. Yip, L. Ceze, and M. Oskin, "Accept: A programmer-guided compiler framework for practical approximate computing."

[4] J. Han and M. Orshansky, "Approximate computing: An emerging paradigm for energy-efficient design," in *Test Symposium (ETS), 2013 18th IEEE European*. IEEE, 2013, pp. 1–6.

[5] X. Wu, X. Zhu, G.-Q. Wu, and W. Ding, "Data mining with big data," *Knowledge and Data Engineering, IEEE Transactions on*, vol. 26, no. 1, pp. 97–107, 2014.

[6] S. T. Chakradhar and A. Raghunathan, "Best-effort computing: re-thinking parallel software and hardware," in *Proceedings of the 47th Design Automation Conference*. ACM, 2010, pp. 865–870.

[7] V. Gupta, D. Mohapatra, S. P. Park, A. Raghunathan, and K. Roy, "Impact: imprecise adders for low-power approximate computing," in *Proceedings of the 17th IEEE/ACM international symposium on Low-power electronics and design*. IEEE Press, 2011, pp. 409–414.

[8] V. Gupta, D. Mohapatra, A. Raghunathan, and K. Roy, "Low-power digital signal processing using approximate adders," *Computer-Aided Design of Integrated Circuits and Systems, IEEE Transactions on*, vol. 32, no. 1, pp. 124–137, 2013.

[9] J. Liang, J. Han, and F. Lombardi, "New metrics for the reliability of approximate and probabilistic adders," *Computers, IEEE Transactions on*, vol. 62, no. 9, pp. 1760–1771, 2013.

[10] A. Cilardo, "A new speculative addition architecture suitable for two's complement operations," in *Design, Automation Test in Europe Conference Exhibition, 2009. DATE '09.*, April 2009, pp. 664–669.

[11] ——, "Modular inversion based on digit-level speculative addition," *Electronics Letters*, vol. 49, no. 25, pp. 1609–1610, December 2013.

[12] ——, "Variable-latency signed addition on fpgas," in *Field Programmable Logic and Applications (FPL), 2015 25th International Conference on*, Sept 2015, pp. 1–6.

[13] K. Nepal, Y. Li, R. Bahar, and S. Reda, "Abacus: A technique for automated behavioral synthesis of approximate computing circuits," in *Proceedings of the conference on Design, Automation & Test in Europe*. European Design and Automation Association, 2014, p. 361.

[14] M. Barbareschi, F. Iannucci, and A. Mazzeo, "Automatic design space exploration of approximate algorithms for big data applications," in *IEEE International Conference on Advanced Information Networking and Applications (AINA-2016). IEEE*, 2016.

[15] M. Wyse, A. Baixo, T. Moreau, B. Zorn, J. Bornholt, A. Sampson, L. Ceze, and M. Oskin, "React: A framework for rapid exploration of approximate computing techniques."

[16] A. Sampson, W. Dietl, E. Fortuna, D. Gnanapragasam, L. Ceze, and D. Grossman, "Enerj: Approximate data types for safe and general low-power computation," *ACM SIGPLAN Notices*, vol. 46, no. 6, pp. 164–174, 2011.

[17] C. Clang, "language family frontend for llvm," 2005.

[18] C. Rubio-González, C. Nguyen, H. D. Nguyen, J. Demmel, W. Kahan, K. Sen, D. H. Bailey, C. Iancu, and D. Hough, "Precimonious: Tuning assistant for floating-point precision," in *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*. ACM, 2013, p. 27.

[19] W. G. Osborne, J. Coutinho, W. Luk, and O. Mencer, "Power-aware and branch-aware word-length optimization," in *Field-Programmable Custom Computing Machines, 2008. FCCM'08. 16th International Symposium on*. IEEE, 2008, pp. 129–138.

[20] F. Fang, T. Chen, R. Rutenbar *et al.*, "Floating-point bit-width optimization for low-power signal processing applications," in *Acoustics, Speech, and Signal Processing (ICASSP), 2002 IEEE International Conference on*, vol. 3. IEEE, 2002, pp. III–3208.

[21] M. Barbareschi, F. Iannucci, and A. Mazzeo, "An extendible design exploration tool for supporting approximate computing techniques," in *2016 International Conference on Design and Technology of Integrated Systems in Nanoscale Era (DTIS)*. IEEE, 2016, pp. 1–6.

[22] J. Lee and N. Jain, "Loop perforation for approximate computing," 2014.