THESIS FOR THE DEGREE OF DOCTOR OF PHILOSOPHY

# Program Analysis for Performance and Reliability

JACOB LIDMAN

*Division of Computer Engineering*
*Department of Computer Science and Engineering*
CHALMERS UNIVERSITY OF TECHNOLOGY
Göteborg, Sweden 2017

**Program Analysis for Performance and Reliability**
*Jacob Lidman*
*Göteborg, Sweden, 2017*
ISBN: 978-91-7597-667-9



Doktorsavhandlingar vid
Chalmers tekniska högskola
Ny serie Nr 4348
ISSN 0346-718X

Technical Report No. 149D
Department of Computer Science and Engineering
Chalmers University of Technology

**Contact Information:**

Division of Computer Engineering
Department of Computer Science and Engineering
Chalmers University of Technology
SE-412 96, Göteborg, Sweden
Phone: +46 (0)31-772 10 00
`http://www.chalmers.se/cse/`

Printed by Chalmers Reproservice
Göteborg, Sweden 2017

# Program Analysis for Performance and Reliability

Jacob Lidman
*Department of Computer Science and Engineering*
*Chalmers University of Technology, Sweden*

**Abstract**

The increased demand for computing power has lead designers to put an ever increasing number of cores on processor dies. This advance has been made possible through miniaturization and effectivization of the underlying semi-conductor technology. As a by-product, however, the resulting computer systems are more vulnerable to interference. This has made reliability a first-order concern and is treated both in software and hardware through some form of redundancy. Redundancy is however detrimental to performance leading to more resources spent re-computing. Efficient use of hardware requires software that can take advantage of the computer system.

Compilers are responsible for translating high-level source-code into efficient machine-code. Transformations in the compiler can improve performance and/or reliability of the software. Prior to applying such transformation the compiler needs to verify the legality and benefit of this optimization through program analysis.

This thesis develops program analyses for reasoning about performance and reliability properties and show how these synthesize information that could not be made available from previous approaches.

First, I present an analysis based on abstract interpretation to determine the impact of a finite number of faults. An analysis based on abstract interpretation guarantees logical soundness by construction, and I evaluate its applicability by deducing the fault susceptibility of kernels and how a program optimization affect reliability.

Second, I present the fuzzy program analysis framework and show that it admits a sound approximation in the abstract interpretation framework. Fuzzy sets allow non-binary membership and, in extension, a qualitative static program analysis that can perform common-case analyses. Furthermore this framework admits a dynamic analysis based on fuzzy control theory that refines the result from the static analysis online. Using the framework I show improvement on a code motion algorithm and several classical program analyses that target performance properties.

Third, I present an analysis based on geometric programming for deciding the minimal number of redundant executions of an program statement while maintaining a reliability threshold. Often a fixed number of redundant executions per statement is employed throughout the whole program. To minimize performance overhead I exploit that some statements are naturally more reliable, and more costly, than others. Using the analysis I show improvement in reliability and performance overhead due to use of a redundancy level that is tailored for each statement individually.

**Keywords:** static/dynamic program analysis, performance, reliability, abstract interpretation

# List of Publications

Parts of the contributions presented in this thesis have previously been accepted or submitted to conferences or workshops. Reference to the papers will be made using the Roman numerals below.

I. **Jacob Lidman**, Daniel J. Quinlan, Chunhua Liao, Sally A. McKee, "ROSE::FTTransform – A Source-to-Source Translation Framework for Exascale Fault-Tolerance Research" in *Proceedings of International Conference on Dependable Systems and Networks Workshops, Fault-Tolerance for HPC at Extreme Scale (FTXS)* , Boston, USA, June 25-28, 2012, p.1–6.

II. **Jacob Lidman**, Sally A. McKee, Daniel J. Quinlan, Chunhua Liao, "An Automated Performance-Aware Approach to Reliability Transformations" in *Proceedings of Euro-Par 2014: Parallel Processing Workshops, Resilience*, Porto, Portugal, August 25-26, 2014.

III. **Jacob Lidman**, Sally A. McKee, "Verifying reliability properties using the Hyperball abstract domain" to appear in *ACM Transactions on Programming Languages and Systems*,

IV. **Jacob Lidman**, Josef Svenningsson, "Bridging static and dynamic program analysis using fuzzy logic" in *Proceedings of Quantitative Aspects of Programming Languages and Systems (QAPL)* , Uppsala, Sweden, April 23, 2017

V. **Jacob Lidman**, Josef Svenningsson, "Fuzzy set abstraction" to appear in *Proceedings of Numerical and Symbolic Abstract Domains (NSAD)* , New York City, NY, USA, August 29, 2017

The following manuscript is under review but results are relevant to this work and are so included.

VI. **Jacob Lidman**, Josef Svenningsson, "Fuzzy program analysis and its application in speculative optimization" in *ACM Transactions on Computational Logic (TOCL), 2017*, Submitted for Publication.

# Acknowledgments

First, I would like to thank Josef Svenningsson for his feedback and support. I would also like to thank Sally A. McKee for her encouragement and her help in presenting my ideas clearly. I'm grateful to Lars Svensson for the technical discussions. In addition I'm thankful to the remaining members of the follow-up committee, Per Stenström, Jan Jonsson and Agneta Nilsson for their advice in structuring this thesis.

I would also like to thank all colleagues at the Computer Engineering division for making the workplace interesting and especially Anurag, Dmitry, Alen, Angelos, Kasyab, Stavros, Alirad, Magnus, Risat, Behrooz, Fatemeh, Vinay, Jochen, Viktor and in particular Bhavishya and Madhavan for the numerous discussions/advice and support during these years.

Last and most importantly I thank my family and Maria, without their hearten support this endeavor would not have been possible.

Jacob Lidman
Göteborg, November 2017

*ACKNOWLEDGMENTS*

# Contents

# 1 Thesis Summary

## 1 Introduction

Modern society depends on vast amounts of computing power. To this end processors are designed with increasing number of cores and accelerators. This is made possible through increasingly densely packed transistors with ever smaller feature sizes. As a byproduct, however, the computer systems are more susceptible to interference, from neighboring transistors and outside environment, such as by high-energy particles. When a sufficiently high amount of charge is induced on the transistor it can manifest as transient faults (i.e., soft errors or bit flips). This causes non-permanent alteration to the machine state, such as flipping a binary 0 to a 1. As a consequence, executions sometimes terminate with incorrect results (or do not terminate at all).

Soft errors have an measurable impact on High-Performance Computing (HPC) applications and embedded systems in hazardous environments making reliability a first order concern when designing and using these computer systems. To combat this negative effect some form of redundancy is employed. Redundancy is however negative to performance and striking a balance between the two is problematic. Furthermore the increasing number of computers imply a larger demand for efficiency in terms of power, performance and reliability to maximize return on investment.

The compiler is responsible for turning source-code into efficient machine-code and can be used to improve reliability and performance. Transforming the software however requires consideration into legality and benefit. These questions are answered though program analyses to extract information from the program under consideration. An important aspect of program analysis is the accuracy and validity of the result. Program

analyses that are performed statically produce conservative result as an implication of Rice's theorem and since it often can not impose assumptions on the input set or environment. Dynamic program analysis in contrast is accurate but has limited scope (and in extension validity) because of the limited set of dynamic executions that are considered.

Much of the current work on program analysis is done by the verification community that focuses on verifying functionality which is closely related to legality but less so to benefit. Compiler optimizations for performance are motivated based on improvement in the common-case as opposed to the worst-case/best-case. The difference in analysis type when determining benefit and the problem with limited accuracy in program analysis result motivates focused studies on these properties.

This thesis considers the problem of building program analyses for reliability and performance properties. The verification community has made significant progress in scaling and allowing expressive and yet correct-by-construction analyses. We introduce some underlying themes from their work that we rely on to decide reliability and performance properties efficiently and yet accurate. We will then state explicitly the scope/problem statement (Sections 1.1 - 1.3) and contribution (Sections 3.1 - 3.6) of this thesis.

**Verification and program analysis** Typical verification tasks involve proving a specification over a very large search-space, sometimes even infinite. Early attempts relied heavily on implicit descriptions of the specification and state-space.

The Reduced Ordered Binary Decision Diagrams (ROBDD) [24] is a canonical representation of a boolean function hence making equality checking tractable. By merging equivalent components of the function it can furthermore achieve exponential reduction in storage space, allowing verification of pipelined arithmetic units with $5 \times 10^{20}$ states [25]. The canonicity property is retained when allowing any finite number of terminals. Arithmetic Decision Diagrams (ADD) [8] uses fixed-point numbers and can be used to verify probabilistic specifications when representing a finite congruence set of the unit interval [50]. Bayesian Networks [131] are probabilistic graphical models that rely on independence assumptions between random variables to represent large probability distributions. The distributions of the random variables can be represented by ADD:s to exploit redundancies. Breaking down a large model into simple combinations of smaller models allows exploiting context-sensitivity to achieve tractable inference.

Decomposed verification is commonly used today. Rather than trying to prove an invariant for each input of a component *assume-guarantee verification* [133] iteratively weakens the pre-condition of a sub-component as it makes progress proving properties for its predecessors. But dependence between sub-models is not always binary and some sub-models contribute more to the output. Hence it is possible that we can ignore some assignments/relations between sub-models and still get a sufficiently accurate output [31]. This is the motivation for introducing *abstract models*, and approaches based on abstraction [42] and approximate inference [107], that have allowed the verification of a pipelined ALU with $10^{1300}$ states [38]. An abstract model is a simpler approximation of its *concrete* counterpart. This simplicity is often achieved by reducing

the accuracy of the model whereby some properties cannot be decided as true or false. Finding a suitable abstraction is often non-trivial since it is, a priori, not clear which properties that are essential and more expressive approximations are often less tractable to compute. Verification on abstract models typically try to find the strongest/weakest property before checking if it satisfies the specification and is therefore an optimization process.

The success of methods based in abstraction depends on how much of the state-space that can be *abstracted away* and only explored with very low accuracy. Counter-example guided abstraction refinement (CEGAR) [37] frameworks use equivalence partitions as abstract models and refine these based on the counter-examples obtained if the model does not satisfy the specification. It has been used successfully in a variety of situations, for instance to verify a Fujitsu multimedia-assist processor which was not possible using ROBDD based methods [37]. However, CEGAR is problematic to apply in probabilistic verification as the number of partitions of an infinite state-space is not finite [60].

The verification community has shown that abstraction, modular reasoning and efficient data-representations allow for tractable verification with very large, even infinite state-spaces. Hence they have shown that exploring very large models is possible if we are only interested in a less precise property than the full result.

**Limits of program analysis: Predictability** As mentioned above, even in a very simple scenario it is unreasonable to expect to find highly accurate information statically on the value of a dynamic property. This is in part due to lack of information and furthermore due to the conservative assumptions that are presented to the compiler. Limited information severely hampers the accuracy of static program analyses such as alias analysis [121] and dependence analysis [132]. Dynamic program analysis approaches the problem from the opposite direction by finding a predictor from a set of dynamic traces. This approach aims for logically completeness as opposed to that of its static counterpart which target logical soundness. Dynamic analysis based on machine learning has emerged as a feasible approach to obtain more accurate analysis results and has been used to analyze the best in-lining heuristic [97] and compilation sequence [98]. These frameworks collect a training set of feature assignments and property values online and generate a classifier offline. Deciding on the classifier type and parameter values is non-trivial and similar in spirit to the problem of deciding on a good abstraction but where the input set (i.e, training set) is noisy.

In conclusion, its important to make use of more accurate input data when performing program analysis. For this reason we would like to refine the results from a static analysis when runtime information is available.

## 1.1 Thesis overview/restriction

In this thesis we aim to advance the state of the art of deciding program properties related to performance and reliability. These properties form a class of non-functional properties [40] that make up a *derived relation* (or *view*) of the computation in contrast

to functional properties (e.g. *safety* or *liveness* properties). The performance gain of a transformation is motivated based on improvement in the average case while reliability is motivated based on worst-case results. To this end we consider common-case [139] and worst-case analyses for performance and reliability properties respectively. Furthermore, we consider methods to improve the accuracy of these analyses by allowing qualitative results that can be refined and updated when more information is available.

We specifically look at general performance and reliability properties (as opposed to focusing on properties that are specific to a certain computer system). We aim at producing program analyses that admit tractable and sound approximations.

To this end we consider analyses on flow-graphs where the control structure is known and the semantics of the program can be described by a basic low-level machine, e.g. MIPS R3000. Furthermore we only consider transient faults where the fault description is possibilistic/probabilistic.

## 1.2 Problem statement

This thesis considers the problems of:

I. How to decide program properties related to performance or reliability when considering transient faults and assuming a possibilistic model (where a bounded number of faults occur) or assuming a probabilistic model (where statements fail according to a given probability distribution).

II. How to construct a framework for deciding a qualitative estimate of a property statically and then refining this result dynamically.

III. Based on derived abstract properties, how to transform the program automatically or semi-automatically to improve its reliability and/or performance.

## 1.3 Main Contributions

We list the contributions of this thesis:

I. For Problem I we present an abstraction to quantify the quality degradation of a finite number of faults in a possibilistic fault model. (**Section 3.3**)

II. For Problem I we introduce an analysis to decide the smallest number of redundant statements, to minimize performance overhead, while still maintaining a target reliability. (**Section 3.2**)

III. For Problem I and Problem II we present the fuzzy program analysis framework that allow us to reason about the common-case and worst-case value of a property. We apply this framework to decide properties of interest to speculative compiler optimizations and refine its result dynamically when more information is available. We show how the analysis framework improve the results of a lazy code motion algorithm and constant folding. (**Section 3.5** and **3.6**)

IV. For Problem III we introduce a source-to-source translation framework for improving the fault-tolerance of an application. Using this framework we evaluate performance overhead and reliability improvement of redundant excutions. (**Section 3.1** and **3.2**)

# 2 Background

This section will review the fundamental concepts that this thesis rely upon. Its aim is to convey the basics such that the reader can understand related works in Section 2.3 and later the summaries of the papers in Section 3.1 - 3.6.

For a precise descriptions of these topics we refer to standard textbooks, i.e.. Nielson et al. [123] (in particular Chapter 4 and Appendix A) for abstract interpretation and Dubois et al. [56] for fuzzy sets.

## 2.1 Abstract interpretation

Approximation is central to most engineering fields. Taylor series, which are known from calculus, can be used to approximate arbitrary continuous functions by a polynomial of fixed order. The error of the approximation can be bounded by an error term $\pm$ X. Hence, for any input value we can deduce a range where the output value is guaranteed to exist.

The extension to approximating programs is less straightforward. What should be guaranteed by such approximation?

Abstract interpretation is a framework for deducing *sound* approximations. Soundness implies that a property that is true in the approximation (the abstract program) is guaranteed to be true in the original program (the concrete program). In contrast, if the property is false in the approximation then the analysis is inconclusive. The framework defines an abstraction as a order-theoretic relation between two ordered domains (e.g. representing states or state transformers) where every set of elements have a unique maximum (and often minimum) value. The order formalizes information content of each domain element and the relation allows *transferring* a result from the first domain to the second in such a way that the result preserve or over-approximate information. The first domain describe the *concrete* results and the second describe the approximated, or *abstract*, results. In the special case when we consider computations over numerical domains (i.e. states/state transformers over $\mathbb{R}^n$) the relation amounts to specifying that the results of a computation in the abstract domain is always a superset of a set of computations in the concrete domain.

Our work rely on abstract interpretation to crystallize the meaning of approximation, even when considering very complex or less concrete concepts (such as redundancy and memory sharing between statements). This allows us to reuse a large number of abstract domains and methods for combining and improving approximations. Approximation tend to become complicated when using several abstract domains so having a framework that guarantees correct-by-construction is very convenient.

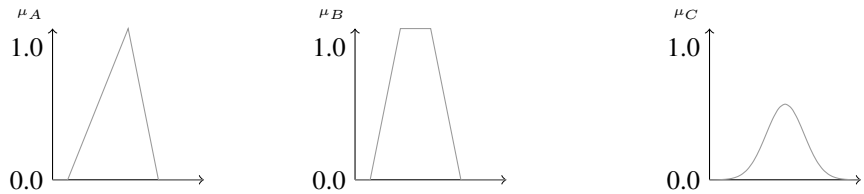


Figure 1.1: Triangular, trapezoidal and gaussian fuzzy sets

## 2.2 Fuzzy set theory

Elements are either members or non-members in classical sets. For fuzzy sets membership is graded and measured by a value of $[0, 1]$ (i.e. the unit interval). Fuzzy sets find applications in reasoning about concepts where membership can be measured but still not assigned a crisp value, for instance *tallness* of a person: is 168cm enough to be considered tall? or 187cm?

Distance to definite values, non-membership $(0)$ or membership $(1)$, determine bias and distance away from the middle-value $(0.5)$ determine uncertainty of the bias. A membership of $0.25$ is *biased* towards non-membership but a membership of $0.1$ has a stronger bias towards non-membership. Figure 1.1 exemplify three common types of fuzzy sets over some linearly ordered domain (e.g. $\mathbb{R}$) where the membership function is a triangular, trapezoidal and gaussian.

If the domain is length of a person the fuzzy sets can be considered to quantify the membership degree of a tall person given his/her height. For the triangular and gaussian fuzzy sets there is a single height with maximal membership and both lower and higher heights is considered to denote, in this categorization, a *less* tall person. Other fuzzy concepts (e.g. short and gigantic) could overlap with tall but have the opposite membership degree, i.e. a person that is unlikely to be considered tall would be more likely to be considered short or gigantic.

Fuzzy sets can also be used to define various reasoning frameworks,

- Fuzzy logic defines logical connectives over fuzzy memberships to allow reasoning over graded truth-values, i.e. logical bias. Fuzzy logic is an extension of Kleene's 3-valued logic which is heavily used in computer science to reason about programs where undefined values are possible.
- Possibility theory is a non-classical probability theory based on fuzzy sets that in contrast to conventional probability theory employ both a possibility and necessity measure to reason about the uncertainty of a statement. Possibility theory can hence reason about non-deterministic uncertainty from combinations of information from two mutually exclusive sources.

The truth-value of a statement should not be confused with the plausibility of a statement although they are sometimes related as shown by the bi-lattice framework [70]. A bi-

lattice employ a truth and information order over the same domain of values. Bi-lattices can be used to relate seemingly different reasoning frameworks.

Our work use bi-lattices and fuzzy logic to reason about bias of a statement. We also leverage other concepts from the fuzzy set community like fuzzy control to define a fuzzy program analysis framework that allow for a solid runtime system where fuzzy classifiers can refine the analysis result dynamically.

## 2.3 Related work

This thesis touches on several areas of program analysis and its relation to compiler optimizations. In particular we present its relation to approximating machine semantics and performance and reliability. Furthermore we present its relation to previous work on hybrid and qualitative program analysis.

**Static analysis of machine semantics** A large body of the abstract interpretation literature focus on verification of high-level code where numbers are elements from the set of reals, $\mathbb{R}$. Approximating execution of low-level code in contrast requires other considerations. Digital processors can only represent finite numbers and therefore compute results over a finite congruence domain of the set of reals, $\mathbb{Z}_{2^W}$ for some $W \in \mathbb{N}$ (typically $W$ is 8, 16, 32 or 64). Here $W$ is the *word length* of the *bit-vector*.

Approximation of numerical properties are in general often performed in the convex polyhedral abstract domain [45] or (because of the high computational complexity in using this domain) a limited subset thereof. The $\mathbb{Z}$-polyhedra domain [124] can be used to approximate machine semantics if each variable is bounded by a constraint (which is possible to express by the polyhedron). However it is hard to implement operations where wrap-around can occur in this domain. Intervals provide a tractable but often too inaccurate abstraction but have been used for machine semantics [67]. Karr [86] showed how hyperplanes (i.e. affine expression) can be used to form an abstraction. This was extended, and refined to restrict attention to modular arithmetic, by Müller-Olm and Seidl [122] who instead considered using an intersection of several hyperplanes (i.e. an affine congruence system of generators). Although programs that only rely on linear equations can easily be expressed in such a domain its harder to define the abstract meaning of non-linear expressions in such a domain, even if its possible to express all state transitions and no state transition in such a domain. King and Søndergaard [88, 89] instead considered using an affine congruence system of relations and showed how this could be done for the boolean case ($W = 1$) using a SAT solver. The abstract meaning is then defined by incrementally adding new satisfying models of the concrete expression until no new models can be found (hence the process is limited by the expressiveness of affine congruence systems). Elder et al. [57] extended this domain to the case of also include non-boolean cases ($W > 1$).

Besides numerical properties there are other aspects of low-level code where approximations is used.

- (*Memory*) The bit-vector memory domain of Sharma et al. [153] approximate memory references by uninterpreted functions. Each memory accessing instruction is provided with a fresh memory symbol that are constrained according to the accessing instruction. Following each update of the value of any symbol in a memory address all pairs of memory symbols are checked if they potentially overlap, in which case an equality between the memory symbols is added. Having a single memory symbol for each static program instruction that access memory is a course approximation when dealing with memory accesses in a loop. Tropical polyhedrons has been used to approximate memory references [3], but as the tropical semi-ring are defined over $\mathbb{R} \cup \{\infty\}$ it is not obvious how to extend these result to low-level semantics.

- (*Cache*) Ferdinand [62] presented an approximation for caches intended to be used when deciding worst-case execution time. The approximation allows computing if an access always hit/miss given a finite, a priori defined number of cache blocks.

As far as we know we are the first to consider approximating machine semantics where the computer system is affected by transient faults (Section 3.3). This abstraction in turn rely on the King and Søndergaard (KS) abstract domain. We consider extensions to our domain and the KS domain in Section 4.1.

**Static analysis of reliability and performance properties** Program analyses of interest to performance properties either decide if a compiler optimization is *legal* (i.e. do not harm performance) or if the compiler optimization is *beneficial* (i.e. improve performance). In improving performance through a transformation we're interested in properties that suggest minimal performance degradation or maximal performance improvement. In using the result of these program analyses the compiler hence inhibit a transformation based on legality properties and favor a transformation based on benefit properties[1].

Several data-flow analysis have been presented that decide legality of a program transformation. Many well-known classical data-flow analyses [123] are used for this purpose. *Constant propagation* decide if a program variable/expression is constant (and if so its value). This information is used by the *constant folding* optimization to compute results already at compile-time. Similarly, *Liveness analysis* deduce which variables that hold values that will be read before being written. Registers allocation rely on this information to decide which variables that need to be assigned to a register [134]. However these optimizations (and analyses) are machine-independent and although legal, given the correct analysis result, can be harmful to performance if applied. For instance in applying register allocation it is beneficial to consider the frequency that variables are accessed when deciding which variables to place in registers or memory (e.g. the spill heuristic).

[1] Although legality/benefit analysis can seem similar to may/must analysis we here differentiate analyses based on the type of property and not the type of collector function

Recent results on program analysis for legality however focus on more complex analyses such as dependency analysis in loops and alias analysis of pointers. Loops where any two iterations are independent can be executed in parallel. Dependency analysis decides if there are two iterations, the first assigning a variable and the second reading from the same variable. The polyhedron model allows such analysis to performed on an implicit description of a loop nest with affine transformation/array access. This description can be used to drive scheduling optimizations. Acharya et al. [1] consider the problem of deciding which loop transformations to apply given a polyhedral description of the dependencies.

In presenting the fuzzy program analysis framework (Section 3.4 - 3.6) we have exemplified the framework using a lazy code motion algorithm, alias analysis and several classical program analysis. We elaborate on extending this framework to consider machine-dependent program analysis in Section 4.3.

Reliability in a probabilistic fault model is hard to accurately approximate at a high (or functional) level since even small change to a low (or implementation) level can have a measurable impact [83]. At the same time it is intractable to compute this for a large and complex model. The Probability Transfer Matrix (PTM) framework [94] defines algebraic operations of connecting faulty circuits in parallel and serial. It uses the ADD description to describe the faulty circuits. But even with this description the framework requires consideration into the order in which algebraic operations are performed and the ADD variable order [95] as well as simplifications [162] to be considered tractable. For this reason we have considered using formal reasoning that admits approximations. But in contrast to the previous works we have considered a possibilistic fault model where a finite number of faults occur. The Single-Event Upset (SEU) model is the most common fault model in this class. Seshia et al. [150] used model checking to verify a ESA SpaceWire communication protocol when it was functioning in an environment affected by the SEU model. Similarly, type checking in Faulty Logic [114] can be used to validate an the Triple Modular Redundancy (TMR) design pattern for the SEU model. However these approaches does not allow any notion of quality to be associated to the results. The trend to approximate computing have shown that quality is an important aspect when considering reliability [120]. The Hyperball abstraction (Section 3.3) allows for an analysis where the quality of a faulty program can be decided.

**Hybrid and qualitative program analysis** The trade-off between safety and benefit of applying a compiler optimization motivates different uses and interpretations of program analysis results. Commonly, production compilers (e.g. GCC) only consider a program transformation if it can decide that the transformation will preserve the semantics of the original program. Motivated by conservative static program analysis results speculative compilation [157] instead delay this check to run-time. The advent of approximative computing has also motivated relaxing the semantics preserving requirement (e.g. guarantee that the result is within some bound) or sometimes provide the user with a knob to decide this level [119].

A promising middle-ground with respect to this trade-off is more expressive program

analysis frameworks where the results is non-binary.

Qualitative program analysis is done either statically or dynamically and associate some notion of confidence with the deduced results. This provides more information to the compiler in deciding if it should apply a transformation or not. Hybrid analyses employ a dynamic analysis to refine and update results obtained statically. Qualitative and hybrid program analysis are orthogonal concepts but has a synergistic effect when combined. The dynamic analysis part of a hybrid analysis can benefit from a static analysis if it includes qualitative information since it more aggressively can focus attention on a smaller subspace.

We next consider qualitative program analysis based on probability theory. Uncertainty quantification (UQ) can give a quantitative estimate of the output of a function given some specification of the input. Intuitively many approaches to UQ are based on probability theory. (Quasi-)Monte-carlo simulation is a probabilistic UQ approach that rely on repeated random sampling to construct an estimate of the distribution of a target function. One major problem with simulation-based approaches is the often large number of samples that are required. The use of meta-models/surrogate models (e.g. Kriging) can alleviate this problem at the expense of a model specific approximation error. Non-sampling based methods instead try to circumvent this problem by deriving algebras to describe the evolution of the quantitative state representation. The distribution arithmetic [128] describe such a evolution in term of a weighted set of random variables. More generally, polynomial chaos expansions [129] is an orthogonal series of a stochastic process that is used for UQ [51]. However, similarly to Fourier series for deterministic processes, there are corner cases where the Polynomial chaos representation become intractable. Often, probabilistic approaches to UQ have problems dealing with discontinuous transitions which are common in programs where information from mutual exclusive control paths needs to be merged. The problem with discontinuities can be handled by instead collecting information using a weighted average. Ramalingam [139] showed that the meet-over-paths (MOP) solution exists for such confluence operator (when the transfer function is composed of min and max functions) over the unit interval. Hence its possible to give a well-defined qualitative data-flow framework inspired by probabilistic reasoning. Several approaches have been introduced to approximate probabilistic deduction or distributions. Cousot and Monerau [47] introduced a unifying framework for probabilistic abstract interpretation. Their framework considers approximating the underlying measurable space in probability theory. Adje et al. [2] introduced an abstraction based on the zonotope abstraction to instead approximate Dempster-Shafer structures and P-boxes[2] that allows also representing non-deterministic uncertainty.

[2]Lower and upper bounds on a cumulative probability distribution functions

# 3 Summary of papers

This thesis is composed of six papers with a common theme. We next summarize these papers and their contributions.

## 3.1 ROSE::FTTransform

**Paper I** makes the following contributions:

I. We introduce a method for semi-automatically transforming a program to a more resilient version based on a specification.

II. We evaluate the applicability of redundant execution as a fault-tolerance construct based on performance overhead and reliability.

We introduce ROSE::FTTransform, a framework for building translators that semi-automatically add fault-tolerance based on a specification. The framework supports different schemes (e.g., redundant executions, replay) at the statement level and could easily be extended to work on different granularities. Furthermore, schemes can be combined as build blocks using *control structures* as glue (e.g., repeating a scheme N times, if scheme X fails perform scheme Y). Due to the existence of a large selection of basic control structures we expect low development cost for many reliability transformations if they are implemented in this framework[3].

We implemented translators based on redundant executions with several voting algorithms. Compared to a programming language extension, minimizing the human factor decreases the risk of introducing errors into the program. Although the current work of the framework targets single core systems with vector/out-of-order execution we expect that it should feasible to implement control structures for multi-core systems. Assuming the overhead of parallelization (e.g. forking/joining threads) is kept low we expect minimal performance overhead since the redundant computations execute without interaction. Our framework enables compositional verification. We also evaluated the feasibility of using redundant executions in HPC kernels with respect to performance and reliability. Our hypothesis was that it should be possible to overlap stall cycles when waiting for memory with computation from a redundant instruction stream.

Compilers need to make sure that no dependencies break in a statement before applying a re-ordering optimization. Imperative languages allow pointers. If pointers are being used in the considered statement the compiler needs to perform alias analysis. This analysis computes *points-to sets* which contain all memory locations that a pointer could reference. Precisely computing points-to sets is difficult and algorithms therefore over-approximate the resulting set. Many compiler optimizations remove redundancy to improve execution time. To allow compiler optimizations in our evaluation, we therefore manipulated inputs through redundant pointers in the HPC kernels. The compiler

[3]The $1-m.n$ voting system in Paper II was implemented without adding any additional control structures

was unable to deduce that the points-to sets were the same and hence kept all statements operating on these pointers. We showed that we can enable aggressive compiler optimizations, such as SIMDization and versioning, even with redundant executions. Because of this we were able to show a low 18% increase in execution time for one of the kernels with double modular redundancy.

## 3.2 An Automated Perf.-Aware Approach to Rel. Transformations

**Paper II** makes the following contributions:

I. We introduce an algorithm for computing the smallest number of redundant executions per instruction.

II. We present a voting system which performs a variable (in a pre-specified interval) number of redundant executions based on necessity.

Approaches based on redundant executions often assign a fixed number of executions per operation. Some operations are, however, naturally more reliable than other. Logical operations is often more reliable than arithmetic operations due to lower circuit complexities. Given the same input domain, an operation that only outputs a boolean value (e.g., the MIPS *set on less than* instruction) has a higher probability of being correct then one outputting a 32-bit word. Similarly, for groups of operations, an operation whose result is consumed frequently is more critical to reliability than one whose result is consumed infrequently.

**Paper II** investigates a method for automatically deducing the number of redundant executions per instruction while maintaining a reliability threshold. To keep performance overhead low, we want to minimize these numbers. We denote the performance impact with an integer weight and assume performance and reliability to be linearly proportional. Based on these assumptions we create an algorithm using geometric programming which is decidable and tractable, admitting solutions to problems with 1000 variables and 10000 constraints in less than a minute on a small desktop computer [19]. The number of redundant executions suggested by our algorithm is often very high (in one case 39 iterations), making the performance overhead unreasonable. We therefore introduced the $1-m.n$ voting system that performs between $m$ and $m+(n-1)\times(m-1)$ redundant executions, as needed. The voter performed a fixed number of stages ($n$), and in each stage adjudicates on a fixed number of results ($m$). After performing $m$ executions ($m+1$ in the first stage) we check the bit-wise majority element of the output of the executions. If this check returns true for all bits we return the majority as the final result. If not, we compute the bit-wise majority element (which will be at least partially incorrect) and perform another $m$ executions and repeat the process up to $n$ times. Using a $1-2.n$ system, with $n$ determined by our algorithm, we showed improvement over a scheme with a fixed number of redundant executions or even a fixed $n$ by upto $58.25\%$.

## 3.3 Verifying reliability properties using the hyperball abstract domain

**Paper III** makes the following contributions:

I. We introduce an abstract domain for computing the set of reachable states after a bounded number of bit flips.

II. We present an abstract domain to quantify the severity of bit flips.

III. We introduce an parametrized abstract domain for maintaining precision as we combine information from disjunctive paths.

**Paper III** investigates an analysis that can be used to deduce how essential an instruction is to the overall results. The analysis computes an over-approximation of the set of states reachable in a program if a fixed number of faults occur when executing an instruction. Paper II focused on improving all instructions of an application. In our formulation of a geometric programming problem we used at least one variable per instruction. Although geometric programming for sparse problems scales to millions of variables, large software can include billions of instructions. Our analysis is formulated in the abstract interpretation framework. Here an analysis is by construction guaranteed to be deciable and a sound over-approximation of the set of reachable states.

To compute the reachable states we extended an abstract domain for low-level code. The extension alters the approximated semantics to include the set of states reachable after a finite number of faults in a pre-specified set of fault sites. We need a measure to assess the severity of an increase in the reachable set of states. This measure quantifies the degradation due to bit flips. Consider an analogy to image processing where degradation due to noise is often measured by some statistical error measure. The set of values is then represented by a hyperball in the case of mean squared error. We therefore created a hyperball abstraction to quantify the worst-case degradation. The hyperball represents a state as a high-dimensional vector and the maximum distance between any state among those actually reached (i.e., our analysis result) and the state we expected to reach (i.e., the center) are reflected by the radius. We lose precision during analysis since we follow both the fault free path and the faulty path. To maintain precision we introduce the scale abstraction. This abstraction stores a bounded set of alternative results as elements. As the number of alternatives grows the abstraction uses mathematical programming to minimize the loss incurred by combining two alternatives. We use our framework to quantify the quality degradation of a set of sorting networks and show how a recently proposed algorithm for improving a cryptographic primitive is affected.

## 3.4 Briding static and dynamic program analysis using fuzzy logic

**Paper IV** makes the following contributions:

I. We introduce a static program analysis framework to reason about the average value of a property.

II. We apply our framework to program analysis of interest to a compiler optimization, lazy code motion, and show that we unveil opportunities classical approaches missed.

III. We present a mechanism based on fuzzy control theory that refines the result online thereby increasing the precision as more information becomes available.

Static program analysis is often conservative, in part due to overly pessimistic assumptions about the input state and in part due to inherit over-approximation to guarantee decidability of the analysis. Dynamic program analysis in contrast is precise but only account for a limited set of executions. Compiler optimizations are justified based on improvement in the common case. Hence results from conventional static program analysis is not suitable as they are often too pessimistic. Similarly the results from dynamic program analysis has limited scope. The aim of this work is to find a middle-ground that yields a static analysis that benefit from dynamic information and a runtime that can improve the analysis result online.

Fuzzy sets was introduced to model vaguely defined concepts such as “tall”, “warm” etc. In contrast to classical sets, element membership is gradual in [0,1] rather than binary in {0,1}. Fuzzy logic define logical connectives on fuzzy sets to reason about truth in the presence of vagueness. Our fuzzy program analysis framework compute the fix-point of a system of fuzzy logic formulas. By using weighted average as collector function, to weight information from different control paths, the results represent the *common value*. Here the weights are constants and can be deduced from profile runs or provided from other source. This approach allowed us to find opportunities classical frameworks would miss, e.g. statements that very likely were loop invariant because their operands was very unlikely to update inside the loop. Furthermore using an analysis on second-order fuzzy sets (i.e., fuzzy sets of fuzzy sets) we showed that we can separate inaccuracies introduced due to the framework from that which is given to the framework. Measuring the uncertainty of the analysis results makes it easier to know when to rely on the results and justify a speculative optimization. Finally the result from the fuzzy program analysis framework can be used in a fuzzy regulator to improve the accuracy of the analysis online. The adaptive neuro fuzzy inference system (ANFIS) regulator consists of a set of fuzzy IF-THEN rules which are weighted to produce classifications. The antecedent of the rules denotes the fuzzy regions where the consequent applies and is the result from our static analysis. The consequent is a polynomial function that initially is set to a default value but can be updated dynamically using polynomial least square regression. As far as we know this is the first program analysis approach that rely on fuzzy classifiers.

## 3.5 Fuzzy set abstraction

**Paper V** makes the following contributions:

I. We introduce the fuzzy set abstraction that generalize the 3-valued logic abstract domain of Sagiv et al. [147].

II. We present a static analysis that deduce the maximal value of a property.

III. We present a dynamic analysis based on monotonically converging fuzzy control systems.

Fuzzy data-flow analysis introduces several new techniques based on fuzzy logic and fuzzy control systems to quantify the logical bias of a property. Given such complex techniques its easy to question the soundness of the results. In **Paper V** we present an abstract domain based on fuzzy sets that generalize the well-known 3-valued logic abstract domain of Sagiv et al. [147]. Our abstraction relies on both fuzzy logic and possibility theory from the fuzzy set community, where fuzzy logic is used to describe how a statement transform input values to output values (as in the fuzzy data-flow analysis) and possibility theory gives some intuition into how values are combined from mutually exclusive sources.

We present a static analysis that compute the maximum value of each property and relate this to fuzzy data-flow analysis of Paper IV. Furthermore we show how converging fuzzy control systems relates to the dynamic refinement mechanism of Paper IV. Although this procedure improves the classification accuracy of the analysis result (i.e. completeness) it may sacrifice soundness. The motivation for this is that often when performing static analysis we conservatively make assumptions on the input information. These assumptions add to the uncertainty of the static analysis result. Hence a dynamic analysis that disregarded some of these assumptions may not introduce errors with respect to the intended input information.

The fuzzy set abstraction show that the techniques in Paper IV can provide logically sound results of the maximum values. Hence the result from fuzzy data-flow analysis can be related to an interval of truth values in $[0, 1]$ which in turn can be related to a value in classical logic or a value representing unknown (i.e. as in three-valued logic).

## 3.6 Fuzzy program analysis and its application in speculative opt.

**Paper VI** makes the following contributions:

I. We extend the fuzzy set abstraction to allow for a common-case analysis.

II. We present a static common-case analysis of two classical program analyses, constant propagation and shape analysis, and show an example of how the analyses benefit from this formulation.

III. We present a fuzzy alias analysis and relate this to a previously proposed speculative alias analysis algorithm.

**Paper VI** extends the fuzzy set abstraction and establish soundness of the common-case analysis from **Paper IV** together with presenting additional applications in speculative analysis/optimization. This puts the common case analyses on solid ground and shows, similarly to the analysis in Paper V that computes the maximum value of each property, that its possible to soundly compute an average value of each property.

The previous considered fuzzy data-flow analyses operate on a domain where all values are comparable, making it easy to compute an average. A large class program analysis operate on domains where some or none of the values are comparable. **Paper VI** present the maximum bias analysis that combines the computation of the maximum and average value. Using this analysis we present a fuzzy version of the classical constant propagation analysis and a shape analysis algorithm and show that the added level of expressiveness offered by the fuzzy version provides additional insights.

**Paper VI** additionally presents a speculative alias analysis using fuzzy logic/possibility theory and relates this to a previously presented probabilistic alias analysis showing that in a frequently occurring situation the two analyses are equivalent. However, its important to stress that this should not be interpreted as a claim of general equivalence between probability theory and fuzzy logic/possibility theory but that the two can output the same result, calling for additional research into the general case.

# 4 Future work

This section presents interesting future work on abstractions for low-level code and computer systems (Section 4.1), reliability and performance (Section 4.2) and possible improvements to the fuzzy program analysis framework (Section 4.3). We also present future work on using program analyses to drive decisions in the compiler optimization engine and revisit the optimal compilation problem in Section 4.4.

## 4.1 Approximations for machine semantics

Performance and reliability are both dependent on the values under consideration. Our work rely on approximating the value-set using a affine congruence system as in the KS abstract domain [88, 89]. The limited expressiveness of this domain is an advantage when performing analyses with faulty semantics (as in the $k$-fault domain) but less so when approximating the value-set which is later used with a reliability or performance property. This because the over-approximation of the value-set will spread to any approximation using it (e.g. approximating a performance property).

We are therefore considering extending the KS domain to approximate piece-wise affine congruence system as a possible future work. The extension would allow for a sound and complete approximation that in contrast to the powerdomain does not represent all solutions explicitly. Since low-level semantics have well-defined bit-length and hence bounded number of variables there is no issues with decidability of such a program analysis and in extension no need for widening/narrowing which can decrease accuracy.

## 4.2 Approximations for performance and reliability

Finding accurate approximations for a given program is often non-trival due to the complex function of the program. This problem is worsened when we allow faulty semantics

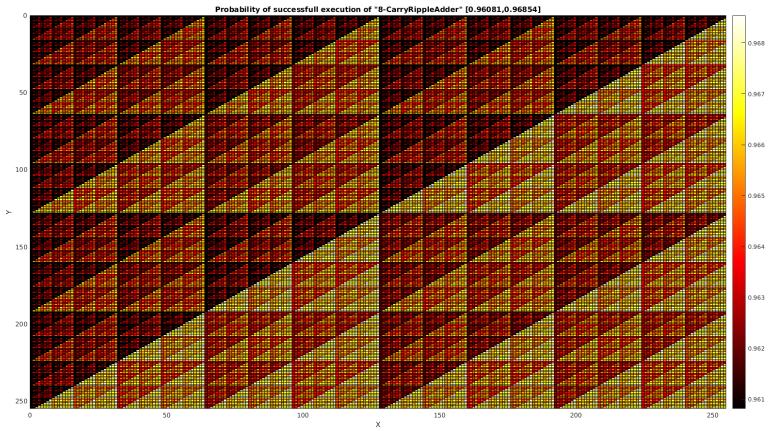


Figure 1.2: The probability that a 8-bit carry-ripple adder produce correct result as a function of the operand values (X and Y axis) if the probability that a gate is correct is p=0.9999

since the function becomes less structured. However even if faults add to the complexity they do not remove the structure of the function altogether.

The Fourier series is a powerful concept in electrical engineering (and statistics, in particular the study of random processes). One of its defining properties is that it describes transformed values in terms of a weighted neighborhood of the original function. Recently Xue [166] presented work on inference in Bayesian networks where the Fourier description of the underlying probability distribution was used, in particular they argued that such a description is smaller in many cases. We are considering an approximation where such a Fourier description can be used together with an abstraction refinement process using smoothing to gradually decide the full probability distribution of a faulty circuit. From simulations we have noted that describing the probability of a fault is often easier done when looking at neighborhoods since faults tend to preserve at least some structure (i.e. symmetries). Figure 1.2 show an example from the result of applying the PTM framework to calculate the probability of a fault in an 8-bit carry-ripple adder. The figure color encode the probability of failure from low (white) to high (black) as a function of the value of the operands (X and Y axis). As seen there are considerable redundancies that could be better exploited if the description was transformed to a different domain.

## 4.3 Fuzzy program analysis

The fuzzy program analysis framework is composed of a static and dynamic analysis that work in synergy. We however specify extensions to both parts separately but the

improvement in one should aid the analysis framework as a whole.

Current work on fuzzy static analysis do not specialize the domain ($S$) or impose any restriction on the structure of the fuzzy set ($S \rightarrow [0, 1]$). As such its possible to form a fuzzy set over abstract elements, e.g. the set of convex polyhedrons. For numerical approximations the underlying set is a subset of $\mathbb{R}^n$. As numerical approximations most often are performed over a lattice domain where the height is large (or sometimes infinite) its important to find a suitable implicit description of the domain as opposed to our current work where this is represented explicitly. Reductions over a pair of abstract domains are easier to design if the domains are similar. We are working on a fuzzy static analysis where we require the structure of the fuzzy set to be a linear fuzzy set, e.g. triangular/trapezoidal/piece-wise linear. The aim is to use this analysis with the piece-wise affine congruence domain mentioned in Section 4.1.

We have presented the theoretical foundation of the fuzzy dynamic analysis and are currently working on a practical implementation. The analysis can be applied to a very general class of problems considering that the structure of the ANFIS can be updated. Although expressive it is less obvious how to harness this ability. We are working on policies for refining the ANFIS classifier such that it improves classification accuracy but possibly sacrificing soundness since this seems more useful given our target application: compiler optimizations.

## 4.4 Optimal compilation

One of the major problems for contemporary compiler research is *optimal compilation* for heterogeneous systems [79], i.e. finding the sequence of compiler transformations that improve performance (or some other objective) of a program the most. Although this problem is undecidable in the general, theoretical, case it should be understood that we restrict attention to approaches that terminate. Even in this case the problem is however intractable where the hardness is often attributed to:

I. The combinatorial complexity [79] and the highly non-linear and discontinuous objective function [90].

II. The limited predictability due to lack of information about the program input and state of the execution environment [35].

We elaborate on an approach to design a compiler optimization engine based on abstractions for performance and reliability properties. First we review lessons from earlier attempt of solving the optimal compilation problem.

### 4.4.1 Model-driven optimization

Early attempts to resolve the optimal compilation problem tried to mimic how expert programmers would optimize code. The BLISS-11 compiler project [164], developed by experienced compiler designers, was well-known for producing high quality machine code from BLISS source-code for the PDF-11 processor [21]. Its front-end performed

syntax/semantical analysis, global data-flow analysis and normalizations to produce the resulting operator tree of the whole program. A complex decision system then decided on simplifications and transformations. In comparison, a recent compiler framework such as the Low-Level Virtual Machine (LLVM) [99] prioritize a modular design over global decision functionality. This makes it easier to create and replace modules (e.g., data-flow analyses of various strengths) but harder to coordinate the modules into producing efficient code. The Production Quality Compiler-Compiler (PQCC) [100] was an attempt to retarget the BLISS-11 compiler to other systems, e.g. the VAX processor. However abstracting the low-level details from the decision function was problematic and the heuristic approach was hard to extend to new architectures [125]. Much later it was shown empirically that the best compiler optimization sequence depends heavily on the underlying architecture [90].

Most contemporary production compilers such as GCC and LLVM employ decision functions that only use local information. For instance, the GCC flag "*-fvect-cost-model*" enables the vectorization cost model that assigns weights determined from experiments to alternative implementations.

For a given processor, compiler optimizations can be characterized by which *processor events* (e.g. cache misses) they induce. Not all state transitions generate processor events, but only the subset which cause degradation of performance. Compiler optimizations are in practise motivated based on a subset of situations where they improve performance. Examples include spatial/temporal locality [16] and cache misses for loop optimizations or instruction parallelism for instruction scheduling [39]. These situations can be assigned an event. Hence the improvement of compiler optimizations can in general be reasoned about by considering events. We can improve performance by iteratively detecting and transforming the instructions that cause events [130]. Although this process is always limited by the accuracy of the underlying processor model, it is always possible to extend the model and consider additional events. Thus, code transformations utilized by this process are "*optimal*" up to the expressiveness of the considered set of event types.

Modern processors include performance monitor counters (PMCs) that count events and aid the programmer in this optimization process. The result of this approach has been shown to be comparable to choosing the best sequence of compiler optimizations based on runs with execution time [168]. Furthermore, the process of improving performance by compiler optimizations is usually limited by the lack of transformations that can further remove events of similar types [96]. This explains why many choices of parameter values for compiler optimizations, e.g. loop tiling factor, produce code with similar performance [90]. As many architectures have common traits they also induce similar events allowing for a modular approach to optimal compilation.

A model-driven approach does not change the combinatorial complexity. However, there are general approaches to optimization that are tractable despite the large search space.

### 4.4.2 Program synthesis, system verification and mathematical optimization

In mathematical optimization it is common to solve large-scale problems in an iterative fashion where each iteration increases the detail of the problem and accuracy of the result, e.g. as in graduated [80] and surrogate optimization [135]. Such approaches operate on gradually increasing the problem complexity starting from a very simple problem. This shares many similarities with the abstraction refinement process and many techniques used in computer-aided verification as mentioned above.

But to use advances from the verification community we need to re-phrase optimal compilation as a verification problem. For this we turn to program synthesis.

Program synthesis is the process of generating a complete program given input-output examples, constraints or program templates [76]. The process can be interpreted as a generalization of program verification that generate three sets of conditions [156]:

I. Safety conditions to ensure *partial correctness*, i.e. requirements that if a result is produced it will be correct.

II. Progress conditions to ensure termination.

III. Well-formedness conditions to guarantee that the synthesized program is valid syntactically.

In the setting of a program template the synthesis engine is tasked with completing a partial program that includes missing values [32] or functions [77] such that the application maximize/minimize some objective function. Abstraction refinement has been used in program synthesis to produce string and matrix transformations [165] upto 90x faster than previous work where abstractions is not used. It is implicitly assumed that the application is semantically-correct for all values of the feasible region of the optimization problem. Many algorithms for compiler optimizations do not produce correct code for all parameter values [79]. Angelic non-determinism through Floyd's *choose* operator is useful for solving this miss-match [15]. The operator non-deterministically picks a value such that the output is correct. Any synthesized deterministic program needs to satisfy this invariant. Angelic non-determinism hence allows us to incorporate compiler optimizations in a program synthesis framework where it is used to enforce the synthesis of the decision function that controls which compiler optimization to apply to which program statement.

As shown Gao et al. [68], the process of optimization, given a decision procedure, could be achieved by a branch-and-bound algorithm that incrementally try to verify that there is no better solution than the currently best known option up to some minimum difference. More generally Optimization Modulo Theory (OPT) allows the optimization problem to solved as a dedicated decision procedure. This has for example been used to solve worst-case execution time problems [81].

# 5 Conclusion

This thesis have considered program analyses for deciding reliability and performance properties. We believe advances in abstractions for these properties are key to enabling optimal compilation which would aid in producing modular compiler optimizations for heterogeneous systems.

We introduce the fuzzy program analysis framework that allows us in a sound way to approximate qualitative information and in particular performance properties; an approach to reason about reliability properties based on three new domains, the hyperball, the $k$-fault and the scale domains; and finally, an approach based on geometric programming that can decide the minimal reliability of a program required to reach at least a given threshold on the output while taking into account the performance cost of the program instructions.

Using these contributions we show how the fuzzy program analysis framework applied to lazy code motion and constant propagation uncover opportunities classical frameworks would not; how to deduce the quality degradation of program in the presence of a bounded number of faults; and finally, how we can reduce the performance impact on fault-tolerant software that use redundant computations.