NTNU
Kunnskap for en bedre verden

Department of Computer Science

TDT4501 - Computer Science, Specialization Project

Reliability Evaluation of Approximate Computing Techniques through Fault Injection

Number of words (excluding front page and references)	3229

Author:
Lars Andreassen Jaatun

Supervisors:
Leonardo Montecchi & Stefano Cherubin

Date: 11.12.2024

Table of Contents

1 Introduction

Approximate computing is a computing paradigm that is gaining traction in both hardware and software applications, due to potential performance gains combined with lower power consumption. This is further incentivized by the end of Dennard scaling; in the 1970's Dennard theorized that together with increasingly smaller transistors, other aspects of processors would scale with it, such as power consumption through a lower required supply voltage (?). This is currently falling apart (?), and so researchers, hardware designers and programmers are starting to look elsewhere for performance and efficiency gains. One limiting factor to the usage of approximate computing is that there is no common method for comparing the reliability of different approximate computing methods. Different techniques may have a different level of inherent reliability, due to achieving performance and energy gains in very different ways. This project aims to investigate how reliability varies within different techniques of approximate computing through fault injection.

This work will focus on measuring the fault tolerance of software implemented approximate computing through faults injected in software. This is due to physical fault injection being a topic that is already extensively covered, and software implemented fault injection allows for a wider range of injected faults. A tool that does not require learning a whole new domain-specific language, and that can be implemented in off-the-shelf hardware, is more likely to be used in practice. For that reason this paper focuses on software tools that can be implemented in existing projects.

This project aims to quantify fault tolerance in approximate computing programs through fault injection, and to enable the comparison of fault tolerance between different kinds of approximate computing methods.

2 Background

This section introduces prerequisite knowledge for this project within approximate computing and reliability.

2.1 Approximate Computing

Approximate computing is a method for obtaining better performance in programs and better power efficiency, at the cost of a reduction in the precision in the computation or of the results. There are multiple ways of performing approximate computing, both in hardware and software (?). ? names some categories within approximate computing, including precision scaling, loop perforation and memoization. What these techniques have in common, is that they do not require specialized hardware to obtain performance and power efficiency benefits. There are other techniques, but they either depend on specialized hardware, such as neural networks used to approximate complex functions (?), or are entirely implemented in hardware (?) and therefore falls outside the scope of this project.

Precision scaling is a broader category, encompassing allocating fewer mantissa bits in floating point variables (?), as well as converting floating point variables to fixed point variables as in the tool TAFFO (?).

Loop perforation means selecting a loop, and not running all of the iterations of the loop, thereby performing less work at the cost of precision. This method of approximate computation cannot always be used due to losing too much accuracy, and therefore requires verifying precision loss in the loop (?).

Memoization means re-using and approximating results for an expensive function, thereby incurring a lower amount of cycles (?).

The implementations of the aforementioned techniques vary in how much extra work they place on the user of the implementation, and can be provided in multiple ways such as a compiler pass that only requires adding annotations to a standard precise program (**??**) or a program library (**??**).

Some examples of approximate computing frameworks are TAFFO (**?**), which uses annotations made by the programmer to convert floating point values into fixed point variables, FlexFloat (**?**) that is provided as an API to allow users to tune precision of floats through decreasing the mantissa and exponent bit width, and Green (**?**), which among other things performs loop perforation, i.e., removes some iterations of an expensive loop to lessen the amount of work. These techniques all increase power efficiency of the computation as well as reducing the accuracy of the result, but they achieve it in different ways.

The acceptable accuracy and required fault tolerance varies between different projects because the requirements can be quite different depending on what domain the code is targeting. The consequences of a home automation light sensor reporting incorrect values is far less than, e.g., a computer that controls a water dam.

Approximate computing is today mostly used within applications where incorrect output does not necessarily make the results useless. In image processing, errors in output can be produced willfully such as for image compression, and while heavy compression will eventually render the image unusable, lesser compression will produce a "faulty" image compared to the original photo data, but to the human eye the differences will be negligible (**?**).

This project aims to investigate how approximate computing can be used in applications where correct output matters to a higher degree, through analyzing the application's *reliability* requirements, and the difference in reliability with or without approximate computing.

2.2 Reliability

Reliability is defined by **?** as continuity of correct service, and this project uses the definitions therein.

Fault tolerance is one way of attaining reliability. Fault tolerance is defined in **?** as the avoidance of service failure in the presence of faults. The other definitions that are relevant for this project are listed below:

- a fault is defined as something that may or may not lead to an error, such as an incorrect comparison operator, or a corrupted database entry.
- An error is defined as an internal state in which a part of the entire system deviates from correct service, but not necessarily affecting the external state of the system. An example of this could be a router dropping packets. This is an error, because the router is not providing the packets it should to the endpoint, but given the packets are transferred using tcp, the packets will be re-sent, thus not resulting in a service failure.
- A failure is defined as an event in which the service a system provides does not correspond with the functional specification of the system. An example of this would be getting "HTTP 404 error not found" screen when trying to access a website.

? defines many different types of faults, all of which can be put into two categories: operational and developmental. Operational faults are faults that occur during the operation of the service and encompasses environmental and physical interactions with the service, while developmental faults concern the faults that appear in the design and construction phase of the service.

2.3 Fault Injection

One method of evaluating whether a system or an application is reliable is through fault injection. Fault injection can be performed on the hardware level or on the software level. Hardware fault injection attempts to simulate environmental interference with the system, i.e., simulate operational faults, through magnetic interference, ionizing radiation, or physically applying external voltage to pins to turn a logical 0 to a 1 (?). Software fault injection concerns injecting faults emulating human non-deliberate faults not covered by test cases to simulate a scenario in which production code is shipped with a fault (?). In addition to this, there are also tools that emulate hardware faults (bit flips), such as Xception (?), as well as tools that can simulate how these emulated hardware faults propagate through a program when run on specific instruction sets (?), and these can be useful in combination with the aforementioned fault injection method.

Mutation testing is a method of doing fault injection where the goal is to verify the quality of your test suite (?). Mutation testing is done by creating ”mutants” of your code by deliberately changing vital parts of the code base to see whether these changes are still caught by the tests. If the tests still pass given mutated code, this ”mutant” is said to survive (which is bad). To ”kill” the mutant, the tests need to be changed so that the test catches both the mutated code as well as the original code.

3 Literature study

The main purpose of the pre-project was to perform a literature study of the state of the art within assessment of reliability in approximate computing studies. The search consisted of an initial reliability-focused snowballing-phase based on the seminal paper on reliability written by ?, and the paper on representative fault injection by ?.

The search was split into two parts: one with a focus on methods for conducting fault injection to obtain information about a systems' fault tolerance, and one with a specific focus on fault tolerance measurements within approximate computing. The search for fault tolerance measurements in approximate computing was conducted on scopus with the following search term: ”approximate computing” AND (”accuracy” OR ”precision”) AND (tool OR technique OR algorithm) AND software AND fault AND tolerance. This yielded 9 papers that I triaged according to the relevance to the project. The search for fault tolerance in general was conducted with the following search term: fault AND tolerance AND injection. This produced 1905 results, which is more than I could go through within the time limits of the project. Therefore I sorted the results according to citations and selected papers from the first 10 pages. The relevant papers were then selected through reading the title and the abstract.

3.1 Fault tolerance

? presents an effective approach for performing software fault injection to inject representative faults in software. They claim that not only is what kind of fault injected important, but also the location. Injecting faults that would be caught by the test suite in any case is not helpful, and therefore they created an overview of representative fault locations for three different application types. The amount of faults that are representative vary between the three application types.

Other tools such as Xception (?) have been proposed to emulate hardware faults (such as bit flips) within software, as opposed to software faults (injecting software bugs). Other projects also look at hardware faults, and compares between different instruction set architectures (??).

? proposes a method for measuring fault tolerance they call fault tolerance threshold, which is the threshold where an error goes from being an error to a failure. This threshold is set arbitrarily dependent on the application domain. They propose a method in which they inject faults into a program, and record how the fault propagates throughout the system, and register whether the fault results in an acceptable result, a silent data corruption or a crash. The faults injected are all bit-flips, and represent operational errors, either with corruptions in memory or radiations induced faults.

? presents a comparison of reliability in different hardware architectures when running different types of benchmarks that stresses different parts of the system. The focus is not generalized processor architectures, but rather accelerators such as GPUs, and hardware you either see in a datacenter or in edge computing systems. They show that different programs and different architectures will benefit differently from adjusting the acceptable error. The way they measure reliability is with the use of failure in time (FIT), which is defined as amount of failures per billion operating hours (**?**), and with MBTF.

3.2 Fault tolerance in approximate computing systems

As approximate computing is becoming more common, more investigations into the reliability aspect of approximate computing are being performed. This is necessary in order to enable the usage of approximate computing within fields that typically do not lend themselves to approximate computing, such as critical infrastructure or scientific high-performance computing (**??**).

? suggest using probabilistic error functions to define error boundaries within approximate computing, which can then be used to evaluate the degree of approximation that the application can tolerate. They apply this to iterative linear solvers accelerated using hardware implemented approximate computing, and achieve mostly more efficient processing while retaining the necessary amount of accuracy.

There are also multiple papers describing fault injection within novel approximate computing hardware (**??**), but this falls outside the scope of this paper.

? covers methods of using approximate computing indirectly through redundancy mechanisms to increase the fault tolerance of a system, though does not delve into the intrinsic fault tolerance of different methods and strategies of approximate computing.

Gem5-approxylizer is a tool created to inject errors into dynamic instructions bit by bit, to evaluate how faults propagate throughout the program and how they affect the accuracy of the program (**?**). The paper performs fault injection on approximate computing applications, and allows the user to map locations in code which are extra sensitive to faults. This does not take in account software faults (i.e. bugs) and therefore the fault tolerance in the developmental phase, but is nevertheless interesting for fault tolerance in the operational phase. Moreover gem5-approxilyzer is significantly newer than the fault injection tools mentioned earlier.

4 Research Gap and Project Objectives

Existing literature explores different kinds of fault tolerance within specialized software systems, both in hardware and software. Papers tend to have a narrow focus, either on the operational or the developmental phase, and do not present a general applicability regardless of the domain. The goal for this paper is to enable the direct comparison of strengths and weaknesses within approximate computing methods that are implemented in tools or frameworks.

Measuring both operational and developmental fault tolerance is important to get the full picture of how a system behaves. To properly gauge *developmental fault tolerance*, we can use G-SWFIT (**?**). G-SWFIT can be used to inject faults that are representative for software in general (faults that appear in real world production code, and that is missed by the test suite). Importantly, **?** makes the point that using fault types that represent the occurrence of faults in the real world is vital for making a trustworthy evaluation of fault tolerance. With all the progress that has been made in the programming world since 2013, especially regarding AI code generation, it is therefore important to verify that the fault categories mentioned are still representative.

Operational faults can be injected at runtime using Gem5-Approxilyzer (**?**), this to gain an overview of how changes in the code affect the reliability of the finished executable. This tool allows the user to follow how operational errors propagate throughout the code unlike anything that can practically be performed only in hardware.

4.1 How to measure fault tolerance

Measuring fault tolerance is done through the concept of coverage. Coverage refers to the degree in which errors are detected, and how well the system recovers from these errors.

Measuring fault tolerance in approximate computing requires some extra considerations:

1. In approximate computing techniques, not all parts of the system is approximate; for example in loop perforation, only the loop is affected, precision scaling only operates on certain variables that exist in certain contexts, etc.
2. Not only operational faults may affect the approximate parts differently, but also the developmental faults. The literature does not cover whether developmental faults have a greater effect on approximate computing systems than precise computing systems.

There are many metrics in the studies that have previously been mentioned that can be applied for approximate computing applications. In addition to using preexisting metrics, the project will explore new metrics to gauge different aspects of fault tolerance within approximate computing.

The project will include performing mutation testing to evaluate a test suite. The mutation testing will be performed using **?** as a guide, to evaluate whether the approximate computing strategies have different effects on a test suite.

To see the effect that the approximate blocks have on the rest of the system the project will include injecting faults in the code blocks that contain approximate constructs:

$$\frac{\text{faults that cause error in approximate block}}{\text{faults that create errors that propagate to the result}}$$

This will be measured both with operational faults through the gem5-approxilyzer(**?**), as well as with injected software faults.

These metrics will be used in combination with the G-SWFIT technique for injection of representative faults(**?**). This is not an exhaustive list. As the project progresses additional metrics may be developed.

One thing that these metrics lack, is the ability to directly compare between different methods of approximate computing. This stems from the fact even though source code for different types of approximate computing may be similar, in practice faults may affect similar source code locations in very different ways.

A goal for the project is therefore to combine and revise existing metrics to allow for better comparisons between approximate computing systems, providing researchers and developers with an overview of the reliability of an approximate computing tool in addition to performance and energy efficiency.

5 Further work

Reliability is not a static measurement that can be forecasted with 100% accuracy; the reliability of a system is an expression of the probability of a service experiencing a failure. Quantifying the reliability of a system can however aid you in taking precautions, and help you decide whether the risks of some arbitrary software are small enough to justify the usage in a specific domain. There exists strategies that allow programmers to make informed decisions about the required accuracy of their program, such as **?**, but only in the operational phase. In addition to this, not all outputs of functions or systems can be explicitly categorized as inside or outside a numeric error tolerance limit as they propose in **?** and **?**.

Approximate computing is special in the sense that the criteria for "correct service" is already relaxed through accepting a wider range of outputs with the same domain as an equivalent nonapproximating system. This itself affects the reliability of a system, and to realistically gauge the reliability of an approximate system compared to a precise equivalent requires a targeted effort.

For the M.Sc. project the goal would be to design a model or system that can be applied to multiple generalized approximate systems. The performance aspect of approximate computing systems is already heavily covered in other papers, without relating the results to reliability.

The masters' project will start with a brief literature review, to ensure that no important papers have been missed.

The next two weeks will be focused on assembling/adapting a benchmark that represents a common workload in an approximate computing tool, such as TAFFO(**?**). Further two weeks will be spent on assembling the fault tolerance metrics, and finally one month each for:

1. Fault injection experiments.
2. Extended fault injection experiments, adjusting and refining the metrics.
3. Finalizing the thesis report.