Toward Floating-Point Run-time Variable Precision in CPU-based Architectures

Noureddine Ait Said

Communauté
UNIVERSITÉ Grenoble Alpes

THÈSE

Pour obtenir le grade de

DOCTEUR DE LA COMMUNAUTÉ UNIVERSITÉ GRENOBLE ALPES

Spécialité : **Nano Électronique et Nano Technologies**

Arrêtée ministériel : 25 mai 2016

Présentée par

Noureddine AIT SAID (doctorant)

Thèse dirigée par **Katell MORIN-ALLORY (directeur)**
et codirigéee par **Mounir BENABDENBI (co-directeur)**

préparée au sein du **Laboratoire Techniques de l'Informatique et de la Microélectronique pour l'Architecture des systèmes intégrés (TIMA)**
dans **École Doctorale Électronique, Électrotechnique, Automatique et Traitement du Signal (EEATS)**

Toward Floating-Point Run-time Variable Precision in CPU-based Architectures

Thèse soutenue publiquement le **Mercredi 24 Novembre 2021**,
devant le jury composé de :

M. Fréderic ROUSSEAU
Professeur des Universités, Université Grenoble Alpes, Président et Examinateur

M. Alberto BOSIO
Professeur des Universités, Université de Lyon, Rapporteur

M. François PÊCHEUX
Professeur des Universités, Sorbonne Université, Rapporteur

M. Arnaud VIRAZEL
Professeur des Universités, Université de Montpellier, Examinateur

Mme. Katell MORIN-ALLORY
Maitre de Conférences, Université Grenoble Alpes, Directrice de thèse

M. Mounir BENABDENBI
Maitre de Conférences, Université Grenoble Alpes, Co-Directeur de thèse

To my mother Zahra,
my father Ahssine,
my sister Hasnaa,
my brother Youssef,
my grandparents,
my friends,
and to the memory of my great-uncle, Houssa,
from whom I learned the meaning of determination and self-discipline.

Acknowledgments

Je tiens tout d'abord à remercier mes encadrants Mounir Benabdenbi et Katell Morin-Allory de m'avoir si bien encadré et supporté pendant ces trois dernières années et d'avoir été toujours disponible. Mounir, je suis reconnaissant d'avoir cru en moi dès mon stage PFE et de m'avoir fait confiance. Merci aussi pour le soutien moral et de m'avoir permis de découvrir un métier aussi noble que l'enseignement. Katell, je te remercie également pour ton soutien continu. Grâce à nos nombreux échanges j'ai appris de beaux arts: rigueur, analyse, précision et bien d'autres.

Vous étiez tous les deux toujours gentils et bienveillants envers moi. Avec vous j'ai appris à prendre du recul, à être bienveillant, et à faire confiance en moi. Sans oublier de nombreuses qualités personnelles et connaissances culturelles que j'ai pu aquérir à travers nos discussions. Je vous en serai éternellement reconnaissant!

Je remercie également les membres du jury: M. Alberto Bosio et M. François Pêcheux qui m'ont fait l'honneur d'être les rapporteurs de cette thèse. Je remercie aussi M. Arnaud Virazel et M. Fréderic Rousseau d'avoir accepté d'être examinateurs de cette thèse. Merci également à ... d'avoir présidé le jury.

Mes vifs remerciements à tous mes collègues du Laboratoire TIMA: doctorants, professeurs, chercheurs et administratifs. J'adresse mes remerciements à tous les membres de l'équipe AMfoRS pour leur accueil chaleureux et pour leurs échanges constructifs. Je leur souhaite à tous et toutes une bonne continuation. Mes remerciements vont également à M. Laurent Fesquet et aux membres de l'équipe CDSI pour leurs précieux conseils et leurs échanges enrichissants, surtout dans le côté technique. Merci à M. Rodrigo Iga pour son aide technique lors de l'implémentation ASIC. Je remercie aussi mes collègues du CIME Nanotech M. Mohamed Ben Jrad, M. Abdelhamid Aitoumeri et M. Robin Rolland de leur support technique et leur serviabilité.

Enfin, j'adresse ma sincère gratitude à ma chère mère Zahra et mon cher père Ahssine qui ont œuvré pour ma réussite, de part leur amour, leur soutien, tous les sacrifices consentis et leurs précieux conseils, pour toute leur assistance et leur présence dans ma vie, reçoivent à travers ce travail aussi modeste soit-il, l'expression de mes sentiments et de mon éternelle gratitude.

Mes remerciements s'adressent aussi à mes anciens chers professeurs Oussama El Issati et Karima El Otmani, et à mes chers ami.e.s Hajar, Abdelhamid, Soukaina, Mourad, Samih, Youssef(s), Zakariyae, Imane, Assia et toutes les autres personnes qui par leurs paroles, leurs écrits, et leurs conseils ont guidé mes réflexions et m'ont aidé à la réalisation de cette thèse ainsi que toutes celles qui ont contribué, de part leur sympathie, à son bon déroulement. Qu'elles soient toutes assurées de ma reconnaissance et gratitude.

Merci à vous tous.

Contents

1 Introduction

Contents

1.1 Context and Motivations

1.1.1 The road to emerging computing paradigms

Moore's law: the beginning of the end In 1975, Gordon Moore, the co-founder and former CEO of Intel made a prediction that has become considered as a "law":

> *"The complexity for minimum component costs has increased at a rate of roughly a factor of two per year. Certainly over the short term this rate can be expected to continue, if not to increase. Over the longer term, the rate of increase is a bit more uncertain, although there is no reason to believe it will not remain nearly constant for at least 10 years"* [1]

This observation states that the number of transistors in a dense **Integrated Circuit** (IC) doubles about every two years. Although it was more of an intuition- and experience-based observation, rather than an empirically-backed formal law of physics, this statement held since then and has long been used as research and development road-map guideline in the semiconductor industry. Figure 1.1 depicts the evolution of processor trends in terms of transistor count, single-thread performance, frequency, and other parameters during the last 40 years.

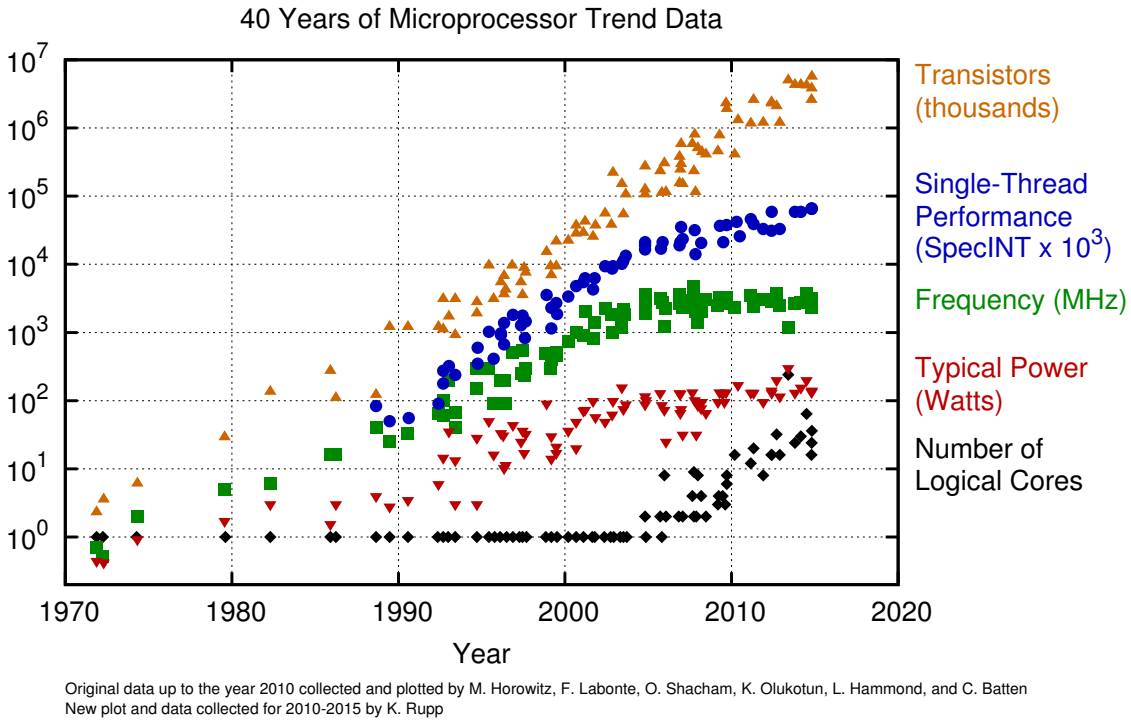


Figure 1.1: Evolution of processor trends.[1]

The various advancements in this domain have led to a democratization of semiconductors (*i.e.*, production cost optimization and economic growth) while

[1] Raw data is available at https://www.karlrupp.net/2015/06/40-years-of-microprocessor-trend-data/

increasing quality *e.g.*, computational performance, memory capacity, sensors' precision *etc.* In the last decade, technologists and computer architects started reporting that the advancements' speed has slowed-down. However, there is no consensus on when the pace predicted by Moore will no longer be applicable, especially with the arrival of new technologies characterized by transistor lengths as low as 5nm.

> *"While scaling computing performance has never been easy, a number of factors have made scaling increasingly difficult this past decade, and have caused power to become the principal constraint on performance."* [2]

More dense technologies have been announced recently with increased transistor density, increased speed and reduced power consumption. For example, in May 2021, IBM announced it had produced a 2nm process node [3]. However, the term "X nm" is not necessarily correlated with the transistor's length, but instead, it has become a marketing term [4] that refers to an improved version of the node. For instance, TSMC's 3nm FinFET chips will reduce power consumption by 25 to 30 percent at the same frequency constraint, increase speed by 10 to 15 percent at the same power constraint and increase transistor density by about 33 percent compared to its previous 5nm FinFET chips [5].

This means that the slowness of Moore's law is now being masked by innovation in the process node technology domain. However, instead of only betting on the technology factor, there is a need for more techniques to drive computer architecture innovation.

Dennard scaling doesn't make things simpler Figure 1.1 also exposes another concept: the *"Dennard scaling"* law [6], which is another factor that impacts the evolution of the semiconductor industry. It states that transistors' power density stays constant while their size shrinks. This means that their power usage is proportionate to their area, while voltage and current decrease *i.e.*, as transistors' length decreases, we are able to reach higher frequencies using the same amount of power. The reduction of transistor power allowed IC manufacturers to increase clock frequencies drastically from one generation to the next without significantly increasing overall circuit power consumption.

However, although performance increases when frequency is increased, power dissipation also increases, hence generating heat. And exceeding a certain frequency threshold requires appropriate and complex cooling systems (which also consume an important amount of energy), especially with other technologies such as multi-processing having emerged. The end of this law is represented with the frequency curve (Fig. 1.1), stagnating between 3 GHz and 4 GHz starting from around the year 2006.

Emerging paradigms to the rescue The end of the *Dennard Scaling* law and the slowness (or arguable end) of *Moore's* law marked the beginning of a new quest: increasing computing performance and energy efficiency through disruptive computing paradigms targeting mainly computing and memory / storage.

On the memory side, advancements in emerging memories have led to technologies such as Phase Change Memories (PCM), Resistive Random-Access Memories (RRAM), Magneto-resistive Random-Access Memories (MRAM), ... which

saw the light in the context of modern use cases such as **Internet of Things** (IoT), **Machine Learning** (ML), especially for edge computing applications. However, their reliability and longevity is still fragile, which explains why CMOS-based memories (Flash, DRAM, SRAM) are still in use in current ICs despite their energy cost.

On the computing side, a framework of *emerging computing paradigms* has also emerged. This thesis falls under this framework, which encompasses new disruptive techniques that have been introduced in the **State of the Art**. The objective is to tackle the energy efficiency and the power wall problem caused by the end of the Dennard Scaling law using drastically new approaches.

Figure 1.2 presents some trending emerging computing paradigms explained below:

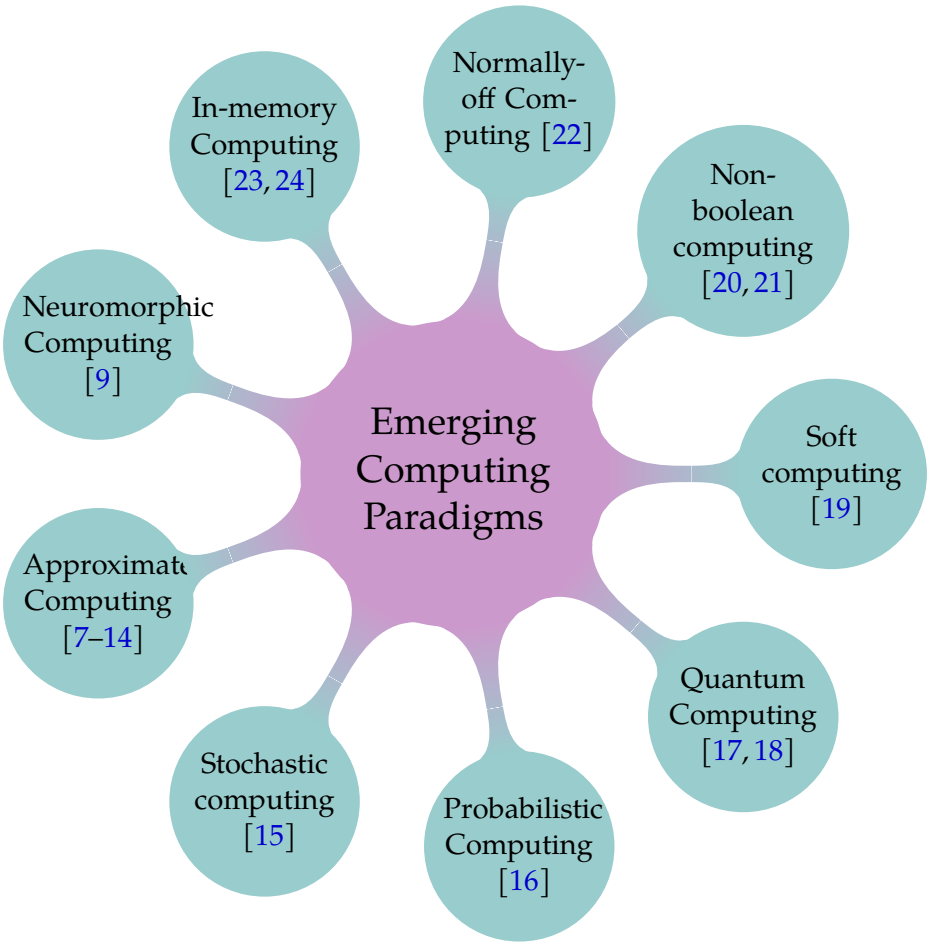


Figure 1.2: Emerging computing paradigms

- **Stochastic computing** [15]: consists of representing continuous values by carefully-crafted random bit streams and performing computations between these streams using simple bit-wise operations (AND, OR, ...).
- **Probabilistic Computing** [16, 25]: this paradigm introduces the concept of probabilistic bits or "p-bits" to represent data using tiny magnetic memories. Knowing that magnets tend to become unstable when miniaturized, this instability has been turned into a feature to encode the data.

In such computers, p-bits evolve from an initial state to a final state, transiting through one of many possible intermediate unstable states. Which path the computer takes depends on pure chance, with each path having a certain probability. By accounting for the probabilities of all possible paths, one can estimate the probability of getting to a specific final state. Probabilistic Computing is a paradigm that essentially paves the way toward the Quantum Computing paradigm, since it is used to simulate the behaviour of quantum computers and they both start from similar mathematical foundations [25].

- **Quantum Computing** [17,18]: Quantum computing is a paradigm that relies on quantum states' properties such quantum entanglement and superposition to perform computations. Quantum computers are known to be able to solve complex computational problems more efficiently and substantially faster. The most cited example is the RSA encryption that relies on integer factorization. Recent trends in this field are shifting toward real-world applications in biology, pharmaceutical and security domains.

- **Soft computing** [19]: a set of approaches that take advantage from applications' intrinsic resilience against imprecision to improve tractability, robustness, efficiency and performance. This field is based on fuzzy logic, genetic algorithms, information theory, machine learning, and probabilistic reasoning.

- **Non-boolean computing** [20,21]: non-boolean computing systems leverage the continuous analog nature of some electronic components such as nanomagnets (as it is the case with probabilistic computing) to represent, transmit and perform computations on data. For example, in [21], low-power energy-efficient nanomagnetic systems were used to solve some quadratic optimization problems usually found in computationally-intensive computer vision algorithms, which shows the potential of such alternative computing method.

- **Normally-off Computing** [22]: yet another computing paradigm that consists of aggressively powering-off power systems' components when they are not needed for the current operation. The need for this paradigm comes from the limitations of traditional clock-gating techniques, especially in the memory design domain, which are tackled by using new **Non-Volatile Memories** (NVMs).

- **Non-Von Neumann Architectures**: these are architectures that make a disruptive leap between the usual Von-Neumann-based traditional computing systems:

  - **In-memory Computing** [23,24]: consists of bringing the memory closer to compution units or vice versa. The objective of such paradigm is to alleviate the cost of memory-to-CPU transfers. Different kinds of architectures have been explored in the literature allowing to perform all or part of the computations closer to memory. The proposed architectures are usually enhanced with principles from Non-Boolean Computing, where analog components, such as resistive memories are used to perform computations.

- **Neuromorphic Computing** [9]: this biologically inspired paradigm encompasses a variety of brain-inspired computers, devices, and models that contrast the pervasive Von Neumann computer architecture.

One may notice that there are many common research interests between these fields, and sometimes it is not obvious to find the thin border separating them. Thus, the difference in naming conventions is mainly due to the communities working on these concepts.

Approximate Computing (AxC) as a solution: among all the mentioned paradigms, we chose to explore AxC [7–14], which has recently emerged as a promising approach to energy-efficient design of digital systems. Approximate computing relies on the ability of many systems and applications to tolerate some loss of quality or optimality in the computed result. This paradigm is more suitable for inherently error-resilient applications such as most modern computer vision algorithms, neural networks, machine learning and artificial intelligence in general. A taxonomy of existing AxC techniques across the system stack will be discussed in section 2.2.

1.1.2 From Approximate Computing to Transprecision Computing: toward variable FP precision

The modern applications such as edge computing, IoT, and ML have driven the evolution of AxC toward **Transprecision Computing** (TC). The reason for that is the fact that these applications are very intensive computationally. And since they are also error-resilient applications, the standard full precision (*e.g.*, floating-point single-precision and double-precision) supported in most CPUs is shown to be very over-designed. This means that there is an interesting opportunity for AxC and TC to optimize their energy-efficiency by acting on both computational resources and memory bandwidth.

However, in the context of general-purpose CPUs, the standard precisions should still be supported. Meaning that, there is a need for systems or architectures that are able to change their internal operating precision at run-time, on-demand, depending upon the application's need. TC is usually used in a general-purpose CPU context, with a special focus on FP arithmetic, since studies [26] have shown that approximate computing based on integer arithmetic is limited in such contexts (more details in Paragraph §5).

In the remaining of this thesis report, **Transprecision Computing** (TC) will refer to the sub-field of AxC that specifically targets software **Floating-Point** applications within a CPU-based architecture context.

Figure 1.3 graphically illustrates the idea behind TC through an example: Sub-figure (a) depicts a software FP application that uses a 64-bit wide FP format and operates at a high average computation power. The horizontal axis represents the computation progress in terms of executed instructions. The left vertical axis measures the precision of the executed instructions, whereas the right axis measures power consumption. The operating precision variation is represented by the red, orange and cyan curves. The total dissipated energy E at the end of execution

is the integral of power consumption $P(t)$ over the execution time T *i.e.,* the area under the precision variation curves:

$$E = \int_T P(t)\, dt$$

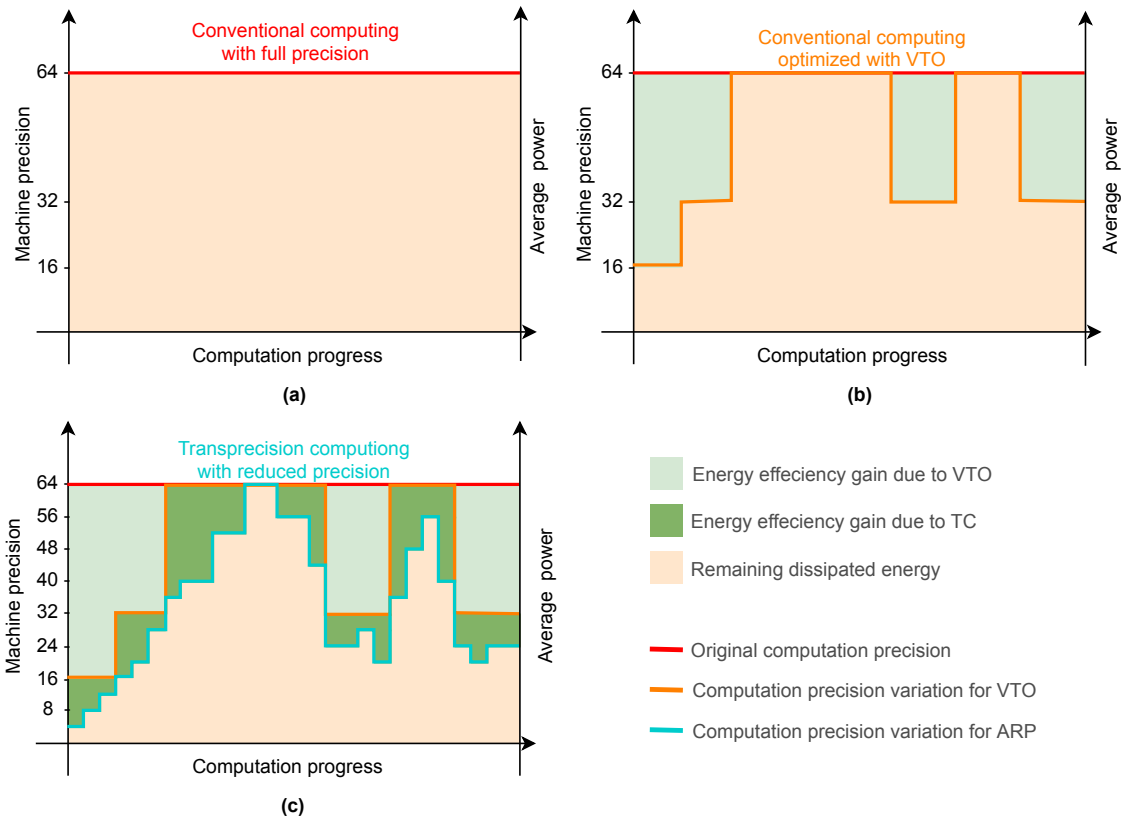


Figure 1.3: Conventional computing to approximate computing (inspired from [27])

Without any optimization (Subfigure a), the application operates using only entirely 64-bits **Floating-Point** variables. A first energy optimization step can be reached using adequate standard format for each variable. This can be done for example, by migrating some variables in an application' source code to lower precision (`float` data type in C instead of `double`). This process is called **Variable Type Optimization** (VTO) and it performs a first optimization pass in order to reduce computation power as well as memory transfers (Subfigure (b)) and can also be performed on most traditional computing systems (CPUs and GPUs). As can be be seen in Subfigure (b), using VTO enables non-negligible energy gains. In the next and final step, **Arbitrary Reduced Precision** (ARP) is used for further power consumption reduction using variables with arbitrary reduced precisions (Subfigure (c)). This allows a more fine-grained and aggressive bit-level optimization in terms of power. Moreover, as a side effect, the total execution time T also depends on the computation precision, meaning that fine-grained precision reduction can also lead to less execution time, and hence more aggressive energy optimization. As depicted in Subfigure (c), more gains are obtained thanks to the bit-level precision reduction.

Through this example, we showed how the granularity of TC has evolved over time and how it has brought more aggressive fine-grained bit-level optimization. Up to now, these optimizations are manual and static *i.e.,* the programmer should decide manually on the precisions that should be used and hard code them in source code. The future directions point toward self-adaptive precision computing and **Variable Precision in Time** (VPT) in order to enable applications to automatically (self) adjust their precision depending on their environment at run-time without or with little intervention from the user.

1.1.3 Guiding Principles for Transprecision Computing

This thesis is a contribution to **Transprecision Computing**, and more specifically, on the computation precision aspects. Traditional TC techniques related to precision reduction suffer from many limitations that can be overcome following a set of guiding principles:

1. **The less intrusive it is, the more adopted it will be**: most existing TC techniques are very intrusive to source code. This means that their use leads to the alteration of the existing software infrastructures. For example, using precision reduction in software needs to rewrite the applications' source code. Sometimes it also needs to implement support at the compiler level. This means that existing SW and HW stacks such as scientific computation libraries, **Application Programming Interfaces** (API), **Operating Systems** (OSes), should also implement support for such techniques and they could not be reused as they are. This constitues a huge loss, especially when it comes to loosing the support for most scientific libraries upon which most of the high-level libraries are based.

   $\Longrightarrow$ There is a need for non-intrusive or at least minimally-intrusive techniques that conserve the existing HW/SW technology bricks.

2. **Simulation only is not sufficient**: many SoA techniques were only developed for simulation purposes *i.e.,* they are designed to simulate the impact of reduced precision on applications' **Quality of Result** (QoR). This is useful to drive designers' decisions when it comes to dimensioning arithmetic units in a high-level way. However, in a general-purpose CPU-based context, such techniques are not production-ready, since they are not backed by specialized hardware that would take advantage from the approximations to reduce energy consumption. Moreover, such techniques introduce significant overheads due to the simulation of approximate behaviour: significant simulation time overhead is introduced since the simulation host processes on full-precision hardware. This brings us to the next limitation.

   $\Longrightarrow$ There is a need for comprehensive HW/SW co-designed techniques that allow high-level and fast simulation of approximate behaviour as well as HW-level support to leverage the benefits of reduced precision in terms of energy-efficiency.

3. **Arbitrary precision hardware support is a must**: most software SoA techniques implement **Mixed-Precision** (MxP) using standard **Floating-Point**

formats such as IEEE 754 [28] *single-precision* and *double-precision* formats. Some GPUs also support the *half-precision* formats. However, to the best of our knowledge, support for arbitrary reduced precision has never been implemented in CPU-based architectures, except for higher precisions [29]. Traditional SoA techniques simulate reduced precision only in software, and these simulation still use standard full-precision hardware, meaning that they are only dedicated for prototyping and they are not production-ready.

$\Longrightarrow$ There is a need for HW-level support for *arbitrary* variable precision achieve aggressive energy-efficiency savings.

4. **Cross-layer and context-agnostic techniques are better**: even when the aforementioned simulation-only techniques are used, most of them only operate at the higher levels of the stack *i.e.,* they enable approximations mostly in the user-level applications, meaning that they are not supported in the low levels of the stack such as the OS level.

   $\Longrightarrow$ There is a need for techniques that enable approximations across all the system layers from low-level HW to high level SW.

5. **Generality is preferred over application- and data-dependancy**: many SoA techniques are specific to particular applications or class of applications, meaning that they cannot be easily generalized to other kinds of applications. Many of them are also data-dependant *i.e.,* the impact on application QoR depends on the inputs on which it is applied.

   $\Longrightarrow$ There is a need for general-purpose techniques that leverage the potential of TC independently of the application or input.

6. **Dynamic is better than static**: many techniques proposed in the literature are static *i.e.,* reduced precision instructions should be decided off-line, at design-/synthesis- time for hardware-based techniques or at compile-time for software-based techniques.

   $\Longrightarrow$ There is a need for dynamic techniques that allow programmers to vary the computation precision dynamically. This dynamic feature can be set either in space or in time. In space: usually referred to as MxP arithmetic, the idea is to allow the programmer to instantiate variables with mixed precisions in the same source code. In time: the operands' precision for a given instruction I may vary during execution time. As will be shown in Chapter 6, this is particularly useful for iterative algorithms.

7. **Automation is yet another driver for mass adoption** many proposed techniques necessitate manually rewriting the source code, or manually exploring the design space exploration. When a methodology is very tedious to apply to existing applications, or when automated flows are not provided, the mass adoption of the methodology is very unlikely. Self-adaptive techniques that enable precision to vary automatically during execution might be of interest.

   $\Longrightarrow$ there is a need for automated or even self-adaptive techniques that enable automatic selection of precision or that allow automatic precision variation during execution. Instead of the programmer manually choosing

the adequate data-types for a specific application, we need algorithms that choose automatically which precision to use at each instant of the program execution timeline under some QoR constraint.

Obviously, building tools and solutions that satisfy all these principles is not straightforward. In our case, we worked on first eliminating the intrusive aspect in FP impact simulation tools (Chapters 3 and 4) *i.e.*, tools that explore the effects of approximate computation on the outputs of a given application. We also ensure that the technique we propose is hardware-friendly and not only useful for software simulation, by proposing an adequate hardware **Floating-Point Unit** (FPU) with support for multiple reduced arbitrary precisions (Chapter 5) for energy efficiency boost. The dynamic aspect is also enabled by design, since the proposed hardware architecture enables precision to vary at run-time. This capability is exploited for the particular use case of iterative algorithms in Chapter 6. We also ensured that the technique we proposed is applicable at different levels of the software stack (OS / RTOS / bare-metal / user-level) hence facilitating usage and combination with other SoA techniques.

1.2 Thesis Outline

This thesis is organized as follows:

Chapter 2 presents a literature review of several **Approximate Computing** techniques at different levels of the system stack, starting from the circuit level, up to the software higher level. We also present how the current trends shifted from traditional AxC to **Transprecision Computing** with an emphasis on **Floating-Point** reduced precision. Next, we present a typical FP algorithm optimization flow and the different tools and libraries existing in the literature and we conclude with a brief comparison between existing works and the ones proposed in this thesis.

Chapter 3 sets the background on **Floating-Point** arithmetic and the terminology employed in the remaining of the manuscript. Then we present a formalized problem statement along with the non-intrusive approach we proposed for the introduction of FP approximations in a non intrusive way.

Chapter 4 is dedicated to the presentation of the AxQEMU simulator: a non-intrusive simulator that implements the approach described in Chapter 3. The technical details of the simulator will be explained as well as two main use cases: the first one is the use of AxQEMU to instrument a well-known benchmark of approximate applications, and the second is about interfacing AxQEMU with other techniques available in the **State of the Art**.

Chapter 5 features the details of the hardware implementation: a multi-precision FPU that enables approximate **Floating-Point** computation and variable precision at run-time synthesized on a 28nm FD-SOI technology. The details of its internal blocks will be then showed. Finally, primary evaluation of hardware-level gains for use cases from Chapter 4 will be discussed.

Chapter 6 focuses on **Variable Precision in Time** (VPT) *i.e.*, the variation of

reduced arbitrary precision during run-time, for fine-grained energy-efficiency optimization. In this chapter we target two iterative methods (Jacobi and Gauss-Seidel) that solve linear systems. After explaining the mathematical foundations of such algorithms, we propose two new versions that take advantage of the multi-precision FPU presented in Chapter 5. A detailed statistical evaluation across many inputs is performed to study the hardware-level gains (power, execution time, and energy) of the proposed algorithms.

Finally, Chapter 7 paves the way to future research directions and concludes this thesis.

2

State of the Art on Floating-Point Approximate Computing

Contents

2.1 Introduction

As explained in the previous chapter, the slowing of Moore's law and the end of the Dennard scaling law have initiated a quest for new energy efficiency optimization techniques instead of only betting on the technology evolution.

We saw that many new computing paradigms have emerged. Among these emerging paradigms, we focus on **Approximate Computing**, which is a promising approach to energy-efficient design of digital systems. AxC relies on the ability of many systems and applications to tolerate some loss of quality or optimality in the computed result [7–14].

Through the last years, many works have targeted the domain of AxC at several levels of the system stack and for several levels of granularity. This chapter presents a cross-layer review of SoA techniques proposed in this field. Then we show how AxC has evolved toward TC that closely relates to FP arithmetic. The need for variable precision floating-point techniques will be illustrated by reviewing the typical FP algorithms' optimization techniques available in the SoA. We'll show that these techniques are well over-designed for error-resilient applications, hence rooting for reduced variable precision in CPU-based architectures.

Then we conclude the chapter with a summarized comparison between the different SoA techniques and define our research contributions' road-map.

2.2 Approximate Computing Techniques: a Cross-layer overview

AxC can be exploited at many levels of the system stack (Figure 2.1), starting from circuit / device level, up to the software application level, through micro-architecture, **Instruction Set Architecture** (ISA), compiler, and language layers.

The development of AxC has started by introducing ad-hoc application-specific techniques in very niche domains, experimenting with concepts such as reduced precision, approximate logic synthesis and approximate boolean functions, approximate functional units, undervolting and underclocking. Most of these works were only software-oriented or only hardware-oriented. These works led to interesting local gains. However, when integrated within complex **System on Chip** (SoC) systems, those gains were proven to be very limited. The ad-hoc and application-specific aspect of AxC was starting to fade away since the scientific community realized that the gains obtained by such techniques were very limited when evaluated at the system level [30].

Since then, a more transverse / comprehensive cross-layer approach started to appear in the AxC community, by proposing more complex and involved methodologies taking advantage of the advances in HW/SW co-design. For example, in [13, 31, 32], the authors have combined many traditional AxC ideas from domains such as ISA extensions, micro-architecture, storage, and circuit in one SoC architecture.

This section presents a detailed taxonomy of existing AxC techniques excerpted

from [7–14].

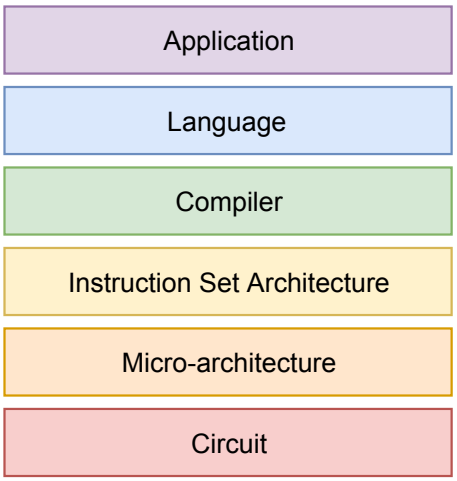


Figure 2.1: System stack

2.2.1 Key Comparison Points

In [12], the authors have presented a taxonomy of general-purpose AxC techniques according to three different criteria:

1. **Correctability**: this criterion evaluates to which extent an approximation technique is able to introduce error and to which extent the introduced error is visible at the architectural level.

   For example, some AxC techniques can produce *silent errors* that have no effects on program execution nor side-effects on the architectural state of the machine (*e.g.*, registers content, caches / memory elements' content, ...). However, others introduce *latent errors*, which do not affect the result directly but affect the architectural state of the executing machine. The latter are dangerous since they may alter the behaviour of the machine later on if not corrected [33]. There are other kinds of errors that might also be introduced, and which should be taken into consideration when choosing an AxC technique to apply.

2. **Reproducibility**: indicates to which extent an approximate execution is deterministic. A reproducible technique is one that yields a constant error every time starting from the same initial state when given the same input.

3. **Error Control**: characterizes the granularity of the impact brought by the AxC approach subject to study both on approximated resources and the output result.

   Although some fine-grained techniques have a local influence on resources (*e.g.*, single assembly instructions, elementary functional units, *etc.*), they can lead to a massive impact on output results and energy efficiency. Hence, it is important to keep track of error propagation throughout the execution.

To these criteria, we can add an other important one:

4. **Intrusiveness**: which characterizes to which extent is the introduced technique intrusive to existing "normal" computing systems. A non-intrusive technique can be applied transparently on at least 1 layer of the system stack. For example, most software-oriented techniques are intrusive to software source code but non-intrusive to the underlying hardware platform. Intrusiveness to existing system stacks influences directly the adoption of AxC in the industry.

It is hard to design AxC approaches that excel on all these criteria simultaneously. Designers should find compromises by trading-off some against others depending on the context and the use case.

2.2.2 Metrics

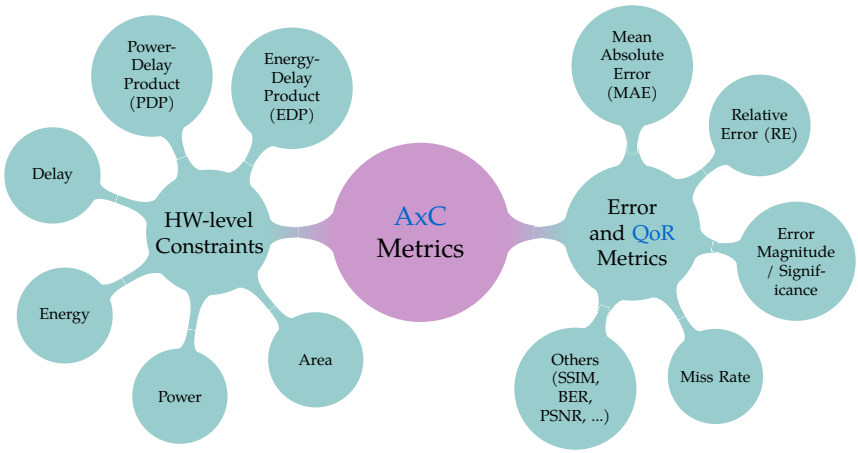


Figure 2.2: AxC related metrics

In order to guide the designer while choosing which AxC technique, approach, or architecture is the most adequate to their specific application, the designer should take many aspects into consideration:

- **Quality of Result evaluation metrics:** a numerical quantity that measures how close is the result under approximation to the accurate result that would have been obtained with accurate hardware. One can also use the term "*error metric*" interchangeably to refer to QoR. There are many standard error metrics. Table 2.1 depicts some of these metrics [34].

  Most of these metrics (Error Rate (ER), Mean Absolute Error (MAE), Relative Error (RE), *etc.*) are only useful for characterizing elementary blocks such as approximate adders / multipliers. However, at the high application-level, designers usually use other metrics that quantify the amount of error introduced in the final output. Application-level metrics depend on the application *e.g.*, Peak Signal-to-Noise Ratio (PSNR) for signal processing applications, Structural Similarity Index Measure (SSIM) for image processing, Top-1 / Top-5 accuracy for **Convolutional Neural Networks** (CNNs) and ML classification applications. Other metrics can also be carefully-crafted to fit the studied use case.

$$\mathrm{ER} = \frac{\sum_{\forall i: O^{(i)}_{\mathrm{approx}} \neq O^{(i)}_{\mathrm{orig}}} 1}{2^{n_\mathrm{i}}},$$

$$\mathrm{MAE} = \frac{\sum_{\forall i} \left| O^{(i)}_{\mathrm{approx}} - O^{(i)}_{\mathrm{orig}} \right|}{2^{n_\mathrm{i}}},$$

$$\mathrm{MSE} = \frac{\sum_{\forall i} \left| O^{(i)}_{\mathrm{approx}} - O^{(i)}_{\mathrm{orig}} \right|^2}{2^{n_\mathrm{i}}},$$

$$\mathrm{MRE} = \frac{\sum_{\forall i} \frac{\left| O^{(i)}_{\mathrm{approx}} - O^{(i)}_{\mathrm{orig}} \right|}{\max(1, O^{(i)}_{\mathrm{orig}})}}{2^{n_\mathrm{i}}},$$

$$\mathrm{WCE} = \max_{\forall i} \left| O^{(i)}_{\mathrm{approx}} - O^{(i)}_{\mathrm{orig}} \right|,$$

$$\mathrm{WCRE} = \max_{\forall i} \frac{\left| O^{(i)}_{\mathrm{approx}} - O^{(i)}_{\mathrm{orig}} \right|}{\max(1, O^{(i)}_{\mathrm{orig}})}.$$

Table 2.1: QoR metrics (Extracted from [34]).

- **HW-level parameters:** introducing AxC does not only affect the QoR, but also HW-level parameters such as average power consumption (measured in Watt), total dissipated energy (measured in Joules), delay (nanoseconds) / maximum frequency (MHz) and circuit area (μm^2). Designers should find trade-offs that fulfil their needs in terms of QoR while guaranteeing maximal energy efficiency.

2.2.3 Circuit level AxC

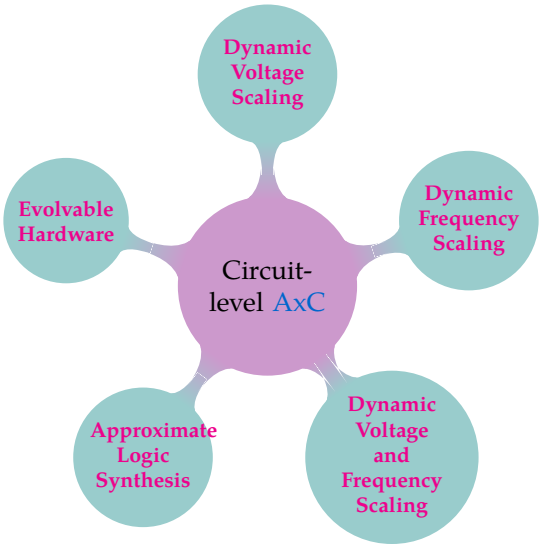


Figure 2.3: Circuit-level AxC

In this category, AxC is applied by deriving an approximate circuit from an

original (precise) one, by voluntarily violating its design constraints in a careful and controlled way. This technique can be applied to different kinds of circuits *e.g.,* arithmetic operator, control circuit, memory, *etc.* Such constraint relaxation techniques usually lead to potential incorrect behaviour after circuit fabrication. For example, some timing constraints (*e.g.,* hold / setup violations, maximum frequency (slack) violation *etc.*) can be relaxed to allow significant performance gains while risking run-time errors.

Among the first techniques deployed in this context are **Dynamic Voltage Scaling** (DVS) and **Dynamic Frequency Scaling** (DFS) (which can be used jointly for **Dynamic Voltage and Frequency Scaling** (DVFS)). This technique consists of relaxing the safety margins of the digital circuitry in order to optimize power, execution time, or total energy.

Of course, this should be done carefully, since the function of the target circuit is altered. The total power dissipated in a digital circuit is composed of two main components: static power and dynamic power.

While the static power (leakage power) is independant of the switching activity of the circuit, DVFS mainly focuses on optimizing the dynamic part, since it can be written as follows

$$P_{dynamic} = a\, C_{load}\, V_{DD}^2\, f$$

Where f is the operating clock frequency, V_{DD} is the supply voltage, C_{load} is the effective capacitance, and a represents the circuit switching activity. Dynamic power can be lowered by reducing either the clock frequency f (underclocking), or the supply voltage V_{DD} (undervolting), or the switching activities of synchronous circuits by removing the clock signal when the circuit is not in use (clock-gating) [35]. As depicted in the previous equation, the supply voltage V_{DD} has more effect on power than the frequency f.

In [36], the authors have proposed a microprocessor system based on ARM V4 with DVFS capability: the supply voltage and clock frequency can be dynamically tuned so that the system can deliver high throughput when required while significantly extending battery life during the low speed periods. In this paper, DVS is used as a mere optimization technique rather than an AxC technique, since the QoR at the output does not change.

Authors of [37] have proposed a similar system called Razor which incorporates an error detection and correction mechanism to recover from timing errors. In this system, the supply voltage is automatically reduced while still ensuring correct operation to achieve an optimal trade-off between energy savings from voltage reduction and energy overhead from increased error detection and correction activity.

Authors of [38] have explored, for the first time, the use of DVS for interconnection networks, where the frequency and voltage of links are dynamically adjusted to minimize power consumption with a history-based policy that judiciously adjusts link frequencies and voltages based on past utilization, hence realizing an average $\times 4.6$ power savings against a moderate impact on performance.

Other works such as [39] presented an approach to find the minimum supply voltage for an overclocked 16-bit or 32-bit ripple-carry adder with an output error reduction reaching as much as $2.58\times$ compared to SoA approximate adders. In

[40] the authors explored the effects of DFS on server workloads, whereas [41] used DFS for dynamic thermal management of **Field-Programmable Gate Array** (FPGA)-based soft-core processors.

Undervolting and underclocking can also be referred to as **Near-threshold Computing** (NTC) [42]. For example in [43], an open-source multi-core RISC-V design is specifically designed with NTC capability allowing it to achieve important speed ($3.5\times$ faster) and energy efficiency ($3.2\times$) gains while operating with a supply voltage of 0.6 to 1.2 V.

Another technique that has been explored in the SoA is **Approximate Logic Synthesis** (ALS). It consists of developing algorithms that synthesize approximate circuits while satisfying some hardware constraints. For example [44] proposed a new logic synthesis approach for minimizing a circuit's area given an error rate threshold. In this work, relaxing the circuit's error rate to 1% leads to an average of 9.43% area reduction across many benchmarks. Similarly, in [45], the authors proposed a design space exploration methodology that trades-off QoR against cost to implement high performance and low energy embedded systems. These works are an attempt to address Moore's law challenges by shifting from the current-day deterministic design paradigm to statistical and probabilistic designs. The major limitation that is common between these last two works is the focus on error rate / frequency only. In fact, error rate should be studied along with error magnitude in order to evaluate the criticity of the errors introduced and to which extent they alter the behaviour of the system under study.

Other ALS-related works have proposed methodologies that tackle this problem by considering the error magnitude constraint [46,47]. The advantage of these methods is that they do not alter the conventional logic synthesis flow. Their main objective is to develop general algorithms for the generation of approximate Boolean functions starting from an exact one. One suitable use case of such methodologies is the design of arithmetic units and HW-accelerated signal processing pipelines.

Evolvable Hardware (EH) is one of the very prominent branches of circuit-level AxC. It consists of starting from exact computing systems and transform them by applying concepts from **Evolutionary Algorithms** (EAs) to generate approximate computing systems. Such concepts are inspired by biological evolution mechanisms, such as reproduction, mutation, recombination, and selection.

In this branch, the cost-QoR trade-off problem is formulated as an optimization problem where candidate circuit configurations play the role of individuals in a population. The fitness function in this case is modeled as a function of the circuit cost in terms of area / power / QoR. The algorithms automatically explore the quality of the candidate circuit configurations and the population individuals evolve iteratively. Currently, this branch is led by Prof. Sekanina from University of Brno [34,48–52]

2.2.4 Compiler- and Language-level AxC

The AxC philosophy can also be implemented in compilers and languages, mainly by proposing new languages, specialized compilers, or expanding existing compilers with AxC-inspired features.

Given a program P, some studies [53–55] proposed tools and approaches to

evaluate the effect of approximations at different points of the program P. Such studies conclude that different instructions tend to have different effects on the result output and that it makes sense to distinguish between critical and non-critical program points. This conclusion has later driven the development of new AxC-aware compilers such as the Java-based EnerJ AxC-aware language and compiler [56, 57].

EnerJ separates critical and non-critical program components and allows the programmer to manually annotate each program component (variable, function, ...). This feature is very important and will be referred to as **Selective Approximation** (SA) in the remaining of this manuscript. Statements are then interpreted by the compiler following a set of type rules and operational semantics that guarantee that precise data remains precise. For example, approximate values cannot be directly assigned to precise variables unless promoted (or endorsed) to a precise type. Operators such as + are overloaded for approximate operations on approximate data. Assembly instructions generated for an exact operation get executed on exact hardware whereas approximate instructions get executed on approximate hardware. Table 2.2 depicts an example of AxC-aware source code transformation using EnerJ. Starting from an original source code (on the left), the programmer can declare some variables as approximate *e.g.,* the integer variable a declared in line 2. Associated instructions will be interpreted differently *e.g.,* the addition operation (line 4) as well as the division operation (line 9) would be executed in an approximate way. The same goes for the variable z which is annotated as an approximate integer variable (line 5). The assignment in line 6 is an implicit approximate operation that will also be executed on approximate hardware.

```
1  int p = 5;
2  int a = 7;
3  for (int x = 0..) {
4      a += func(2);
5      int z;
6      z = p * 2;
7      p += 4;
8  }
9  a /= 9;
10 p += 10;
11 socket.send(z);
12 write(file, z);
```

```
int p = 5;
@Approx int a = 7;
for (int x = 0..) {
    a += func(2);
    @Approx int z;
    z = p * 2;
    p += 4;
}
a /= 9;
p += 10;
socket.send(z);
write(file, z);
```

Table 2.2: AxC-aware code transformation using EnerJ

A similar work, ACCEPT [58] was built on EnerJ and dedicated to C/C++ programs instead of Java. It includes a comprehensive framework for approximation that balances automation with programmer guidance. In ACCEPT, the SA process has been automated in order to guide the programmer while defining which parts of a given program would have to be approximate and which should be precise.

Rely [59] is yet another AxC-aware programming language. It enables developers to predict the quantitative reliability of an application *i.e.,* the probability that it produces the correct result when executed on approximate hardware. Rely allows developers to specify the reliability requirements for each value that a function produces.

Researchers from Intel [60] went beyond simply providing ways to describe approximate behaviours. They introduce a comprehensive open source toolkit named iACT (Intel's Approximate Computing Toolkit) that analyzes and studies the scope of approximations in applications. The toolkit consists of a compiler, a set of libraries, and a simulated hardware test bench. These contributions' main objective is to accelerate AxC mainstream adoption.

Microsoft also contributed to this domain by introducing `Uncertain<T>` [61]: a programming language abstraction for uncertain data. It presents a new type system along with a set of operators that allows developers to explicitly expose and reason about uncertainty.

2.2.5 Algorithm- and application-level AxC

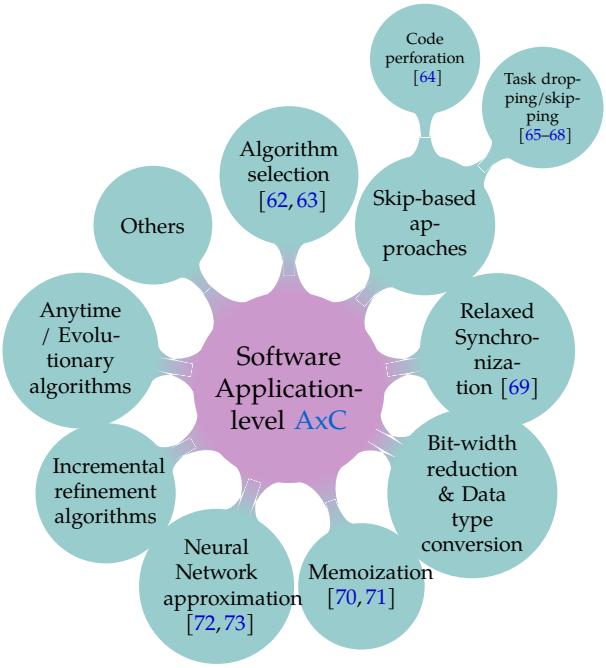


Figure 2.4: Algorithm- and application-level AxC

There have been also many works that targeted the application level, in the following we shortly decribe some of them:

- **Algorithm selection [62, 63]:** traditional AxC techniques are often ad-hoc, data-dependant and do not offer any guarantees on the QoR. [62] tempts to tackle this problem using a framework called Green, that allows programmers to take advantage of approximation opportunities in a systematic manner while providing statistical QoR guarantees. Green enables programmers to approximate expensive functions and loops. First Green builds a model of the studied approximation technique, *e.g.*, loop perforation. This model describes the variation of applications' QoR due to approximation. Based on this model, Green generates a new program that is able to make decisions as to whether or not run-time approximations should be performed. The model built earlier is updated after each run, hence improving the statistical QoR guarantees while achieving energy efficiency gains.

- **Skip-based approaches :** these methodologies are based on suppressing some parts of an application in order to reduce execution time and improve energy efficiency at the expense of some accuracy loss. For example code perforation [64] consists of skipping some iterations when the QoR constraints are relaxed. Similar techniques apply the same approach on threads and tasks [68].

- **Memoization :** consists of pre-computing results of expensive function calls and caching them for future use. If the function is called again with the same inputs, the results are retrieved from the cache directly instead of re-computing them. Such techniques can also be used in hardware (mainly on FPGA platforms) in the form of look-up tables.

  In [70], instruction memoization is applied to reduce the power consumption and increase the performance of low-end/mobile multimedia systems by reusing instances of an already executed operation. However, this technique is not always worth the effort due to the power consumption and area impact of the required look-up tables. This is why the authors proposed a fuzzy memorization technique that associates entries with similar inputs to the same output at the expense of a small QoR loss (more than 30dB Signal-to-Noise ratio) and achieving 12% energy saving instead of 3% using traditional memoization techniques.

  Similarly [71] proposed a simplified software memoization technique dedicated to dynamically linked applications. The methodology has been applied on computationally expensive transcendental functions[1]. This technique is based on intercepting calls to the traditional transcendental functions implemented in the standard C math library `libm`. These calls are replaced by intermediate calls with the following behaviour: the look-up table containing the pre-computed results is checked, if the corresponding result is already present then it is returned, otherwise, the argument is forwarded to the original `libm` library function to be computed and stored back for future use. Such techniques result in an additional execution time overhead in the beginning because the look-up tables should be populated. However, at the end it achieves a performance gain of up to 50% on Intel Ivy Bridge and up to 10% on ARM Cortex-A9. The technique presented in the paper is not approximate since the look-up table stores exact output results. However, the authors presented a way to further increase performance gains using AxC: returning similar outputs for inputs that are close enough.

- **Approximation of functions using Artificial Neural Networks (ANNs):** in this category, ANNs are used to approximate the behaviour of complex software functions. The ANNs can be implemented either in software or hardware. For example, in [72], the authors have proposed an approach that selects and trains a HW Neural Network (NN) topology that mimics a portion of imperative code. A learning phase is first carried out, where

[1] Transcendental functions are those that cannot be expressed in terms of a finite sequence of algebraic operations (trigonometric functions, exponential functions, etc.) and hence are computationally expensive.

the compiler replaces the original code with an invocation of a tightly coupled low-power accelerator called a **Neural Processing Unit** (NPU). The authors have proposed a programming model that allows developers to define "approximable" code regions (*i.e.*, regions that can tolerate errors) to be accelerated with the NPUs. This technique provides results more quickly and efficiently than executing the original code and leads to a speedup of $2.3\times$ and energy savings of $3.0\times$ on average while maintaining an accuracy greater than 90% in all cases. Similarly, [73] pushes the previous idea further by accelerating the neural network inference using a configurable network of NPUs implemented on FPGAs that are available on off-the-shelf programmable SoCs. For a similar accuracy loss as in [72], measurements on a Xilinx Zynq FPGA show an average of $3.8\times$ speedup and $2.8\times$ energy savings.

2.2.6 Architecture level AxC

AxC at the architecture level encompasses a set of methodologies that exploit the inherent error-resilience ability of some applications to leverage energy efficiency, performance and area optimization for computers' main parts: memory, storage, cache, interconnects, **Functional Units** (FUs), and ISAs.

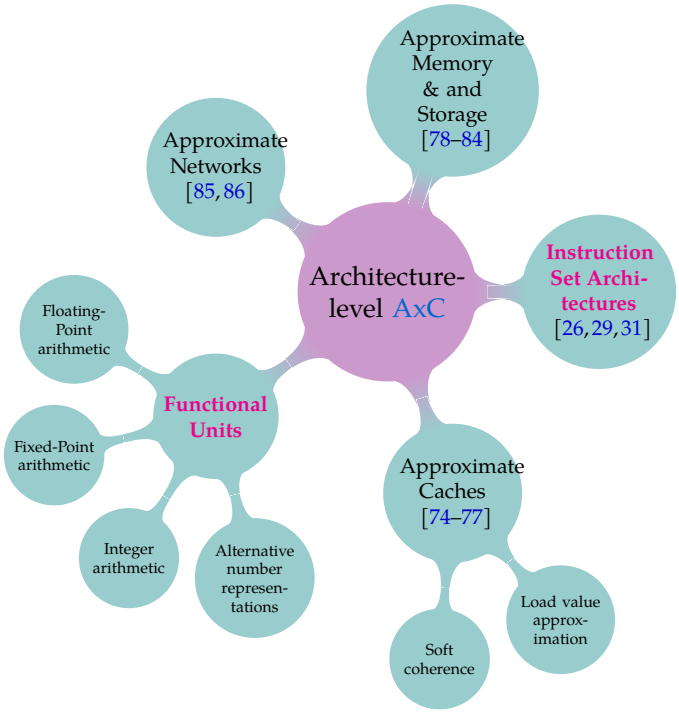


Figure 2.5: Architecture-level AxC

2.2.6.1 Approximate Caches

A CPU cache is a hardware unit intended to minimize the cost of accessing data stored in the main memory. Caches are small and fast memories that are very close to the processor core. They store copies of frequently used data from the

main memory. There are many levels of caches (*L1*, *L2*, ...), but they are all based on two principles:

- **Spatial locality:** when an instruction is executed, it is more likely that the nearby instructions will be executed soon. Similarly, when a data is accessed, it is more likely that the nearby data elements will be accessed soon too.
- **Temporal locality:** when an instruction is executed, it is likely that it will be executed again. This is mainly the case when dealing with loops, since the same instructions are executed multiple times.

An approximate cache is proposed in [74], which approximates similar values and saves energy in the L2 cache of **General-Purpose Graphics Processing Units** (GPGPUs). The paper is based on the observation that threads within the same warp[2] tend to write values into the memory that are very close arithmetically. Hence, similar values are identified at run-time and are written only once, which leads to data size reduction. This process is referred as *data deduplication*. A dynamic energy reduction of about 52% is measured with minimal quality degradation while maintaining performance across a diverse set of applications.

In [76], the authors exploited the inherently noisy or imprecise data that some error-resilient applications operate on using two approximate cache architectures: 1) one that forgoes loading data from caches when the processor can make a reasonable estimate of the value that is needed, and 2) another that selectively determines which values to store in the cache through approximate deduplication of data. The first technique allows the processor to achieve execution time savings by fetching an approximate value instead of going through the cache miss process. The second one helps increase the effective capacity of the cache memory by storing approximate values instead of precise ones. Results show a clear increase in the effective cache capacity.

Authors of [77] worked on improving the data deduplication process by associating the tags of multiple similar blocks with a single data array entry to reduce the amount of data stored, hence achieving $1.55\times$, $2.55\times$ and $1.41\times$ reductions in circuit area, dynamic energy and leakage energy without harming performance nor incurring high application error.

Such methodologies can also be applied to other kinds of caches such as packet classification caches. For example, in [75], accuracy is introduced as a design space exploration parameter in addition to time and space locality. The paper shows nearly an order of magnitude cost savings in terms of memory capacity at the expense of misclassifying one billionth of packets for IPv6-based caches.

2.2.6.2 Approximate Networks

Network on Chips (NoCs) did not escape from the attention of the AxC community. For example, [85] presents an approximate NoC architecture named APPROX-NoC: a hardware data approximation framework with an online data error control mechanism for high performance NoCs. This work also uses data dedupli-

[2] A collection of threads, typically 32, that are executed simultaneously.

cation for aggressive compression thereby reducing the volume of data transfers across the chip. An evaluation shows up to 9% latency reduction and 60% throughput improvement compared with State of the Art NoC data compression mechanisms, while maintaining low application error. On average, the technique achieves a QoR that is always above 99% across the studied benchmarks even though a 10% error threshold is allowed. Another example is presented in [86], where an optical NoC supporting the transmission of approximate data is proposed. This time the approximation is introduced when transmitting a programmable amount of least significant bits of Floating-Point numbers with low power optical signals. Simulations results show that approximations lead to up to 42% laser power reduction for an image processing application with a limited degradation at the application level, compared to an interconnect involving only robust communications. For instance, for a Mean Squared Error threshold of 4.8×10^{-4}, an edge detection application can still provide acceptable results in exchange of 20% laser power saving.

2.2.6.3 Approximate Memory and Storage

Memory and storage technologies consume a lot of energy due to off-chip read-/write operations and data retention. Several works targeting this area studied ways for trading-off energy efficiency against the reliability of stored data.

Static Random-Access Memory (SRAM) spends significant static power on retaining data, so they represent an opportunity for energy versus reliability trade-offs [78–80].

Similarly, Dynamic Random-Access Memory (DRAM) structures use a huge amount of energy on refresh cycles. One can define "approximable" or non-critical memory regions (*e.g.*, containing image data, CNN weights, integers' LSB bits *etc.*) that can be refreshed less frequently [81, 82].

Approximations can also be applied to non-volatile Solid-State Storage (SSS). For example, in [84], the authors proposed mechanisms that enable applications to store data approximately and show that doing so can improve the performance, lifetime, or density of solid-state memories.

Other disruptive ideas have been proposed in the literature. For example, [83] presents high density non-volatile Spintronic memories with near-zero leakage. The major challenge facing such devices is the energy consumed by read and write operations. The storage on these devices can be optimized in a configurable way by leveraging the capability of many applications to tolerate impreciseness in their underlying computations and data. The memory has been integrated as a scratch-pad in the memory hierarchy of a programmable vector processor and exposed to software by introducing quality-aware load/store instructions within the ISA. The ISA of the associated vector processor has been enhanced with new load/-store instructions that provide a field for programmable quality, enabling application software to control which instructions are to be approximated and which ones should be executed precisely on precise hardware. The paper demonstrates 40% and 19.5% improvement in memory energy and overall application energy respectively, for negligible (<0.5%) QoR loss across a suite of recognition and vision applications.

2.2.6.4 Functional Units

A FU or **Execution Unit** (EU) is a part of the CPU dedicated to execute assembly instructions such as computational operations (*e.g.*, addition, subtraction, multiplication, *etc.*) or others (load, store, ...). FUs can be configuration or data registers, internal control unit or other internal units such as **Arithmetic Logic Units** (ALUs), **Floating-Point Units** (FPUs), **Load-Store Units** (LSU) and **Accelerated Processing Units** (APUs). Modern processors usually contain many parallel EU that are often pipelined.

§1 Integer Arithmetic Units Throughout the years, many integer arithmetic unit architectures have been developed. One of the most used architectures in the industry is the **Ripple-Carry Adder** (RCA) depicted in Figure 2.6. It is composed of n interconnected 1-bit full adders, where each adder i receives as an input the carry of the previous full adder $(i-1)$. Although it is a small and accurate adder, this architecture suffers from a limitation: the critical path is very long, since the carry should be propagated from LSB bits up to MSB bits. The critical path increases with the bit-width of the adder.

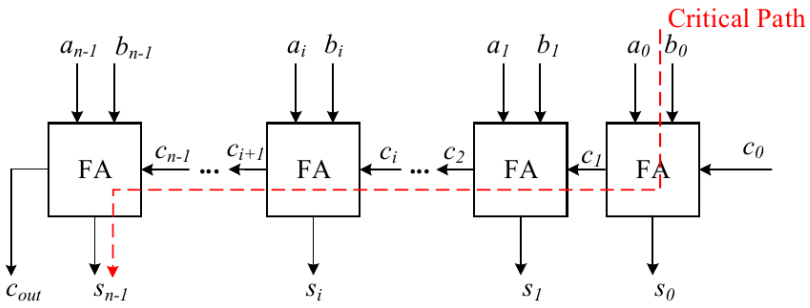


Figure 2.6: Architecture of an n-bit RCA. FA: a 1-bit full adder.

The latency limitation of this adder is known from the mid 1800s and a new category of adders has since been introduced: the **Carry-Lookahead Adder** (CLA). This architecture (Figure 2.7) consists of calculating single (or groups of) intermediate carry bits, which reduces the wait time to calculate the final sum and obtain the final result. The price to pay for this CLA is essentially circuit area and complexity overheads.

To overcome these limitation and to reduce the circuit's hardware complexity, many approximation schemes have been proposed in the literature. For example, in [87], authors have proposed a speculative approach where the carry chain is broken before the propagation is finished, hence allowing to optimize execution time. However, since the speculated results could be different than the results at the end of propagation, there might be corrections to perform. Authors show that the overall performance sees an increase in spite of the overhead caused by recovering from speculation mistakes. A similar approach has been overtaken in [88], where authors proposed an exponentially faster adder architecture that produces incorrect results for a very small fraction of input combinations, along with error detection and recovery mechanisms. Both of these works propose variable-latency adders paving the way toward variable integer arithmetic.

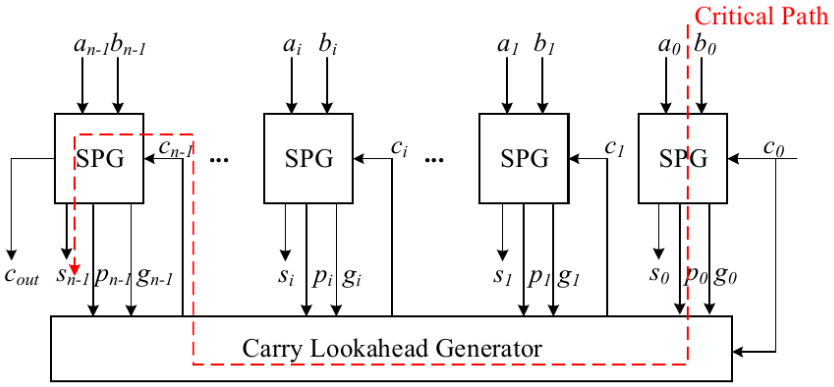


Figure 2.7: Architecture of an n-bit CLA. SPG: the cell used to produce the sum, generate ($g_i = a_i b_i$) and propagate ($p_i = a_i + b_i$) signals.

An alternative technique to leverage approximation for speedup and energy efficiency is to use redundant arithmetic [89, 90]. This number system allows to start computation from MSB bits or LSB bits independently without waiting for the carry to propagate. Using this technique, one can, for example, start computation from the MSB and compute only the necessary bits depending on the criticality of the application being executed. However, there are two downsides to this approach: 1) conversions at FU's IOs from/to a redundant number representation to the usual binary representation is expensive in terms of delay and area, and 2) each bit is computed in one cycle, meaning that an exact 32-bit wide addition would take at least 32 cycles to finish in contrast to the RCA or CLA adders which only take one to two cycles to accomplish. This implies that the redundant number system is more suitable when multiple operations are performed in parallel or when a set of instructions is executed entirely in the redundant number system to avoid intermediate conversions from binary to redundant and vice-versa, or for SIMD instructions.

§2 Fixed-Point (FxP) Arithmetic Units the FxP representation is a number system that uses integer arithmetic hardware to represent fractional values often in base 2. Each number x is represented with a sign bit, an integer part of size m bits, and a fractional part of size n bits. The number x can be represented as an integer number p such that

$$x = p \times K, \text{ where } K \text{ is a fixed scale factor such that } K = 2^{-n}$$

FxP uses speed-, power-, and cost-efficient hardware integer arithmetic units. However, FxP engineering is a time-consuming task, typically demanding 25% to 50% of the total hardware design time [91]. Moreover, it is a challenging task, since the designer should keep track of the virtual decimal point and carefully decide on the dimensions n and m: where n represents the dynamic range and m represents the precision of the representation. This decision is data- and application-dependant and is a complicated process. For that, many works have proposed tools and methodologies that find the best word-length w, where $w = m + n$, adequate for a given application. If m is very short, the dynamic range of repre-

sentable numbers might not be sufficient and overflows might occur during computations. Otherwise, if n is very short, precision will be lost during computation.

This problem is referred to as **Word-Length Optimization** (WLO) and it can be formulated as follows:

> Given an application A and an input dataset D, what is the optimal triplet (w, m, n) that minimizes the circuit cost under a given QoR constraint?

Many works have tried to tackle this problem in the past using genetic algorithms [92], flow graphs [93], geometric programming [94] and others. However, since such applications usually contain dozens of different FxP variables, such a problem needs a scalable approach. To that end, authors of [95] proposed a method to improve the scalability of WLO for large image processing applications that use complex application-level metrics such as the **Structural Similarity Index Measure** (SSIM) metric[3]. The method consists of decomposing the application into smaller kernels, then characterize the impact of approximating each kernel on QoR/cost while taking interactions between these kernels into account. The approach improves the scalability while finding better solutions for image processing pipelines against a 0.9 to 0.99 SSIM **Quality of Result**.

§3 FP Arithmetic Units the fixed scale factor K defined earlier for the FxP number system makes the range of representable numbers uniform but very limited. For example, if a program manipulates numbers that have different orders of magnitude, usingFxP representation would not be suitable. This is why a system with more flexible range and precision is needed. The FP representation was introduced to solve this problem. Moreover, hardware and software implementations enable a faster design time, a considerable performance boost, and an enhanced software portability and flexibility.

The underlying details of the representation will be presented in Section 3.2 as it is at the heart of this thesis. Thereafter we only give a brief summary to facilitate the understanding of the FP related works.

FP represents a real number x with an approximate number $\tilde{x}$ which has a sign bit s, a mantissa field (of size M bits), an exponent field (of size E bits), and considering some base (usually 2 or 10), such that :

$$\tilde{x} = (-1)^{\text{sign}} \times (1 + \text{mantissa}) \times \text{base}^{\text{exponent}}$$

Since its introduction, FP representation enabled programmers to manipulate numbers with different orders of magnitude or non-uniform precision within the same context (*e.g.*, the distance between galaxies, the electric charge of an electron, ...). It allows using the same bit-width without worrying about the position of the decimal point since all the processing/computing is offloaded to a specialized FPU hardware.

The de facto standard that has defined the basics of such number system is the IEEE 754-2008 [28]. It defined many standard formats, such as the 32-bit single-precision format (called `float` in the C language) and the 64-bit double-precision

[3] SSIM measure the similarity between two images: a reference one and an approximated one.

format (called `double` in C), and the 128-bit quadruple-precision format (called `__float128` in C). Over the years, other formats have emerged, such as the 80-bit extended-precision from Intel (called `long double` in C).

Although FP has many advantages, it comes with some limitations: 1) FPUs are very power-consuming for compute-intensive applications, 2) they add a considerable area overhead, 3) the standard types are over-designed for the majority of modern applications (*e.g.*, computer vision, ML, CNNs), and 4) they have a fixed bit-width that prevents the programmers from actually tuning the computations' precision at run-time.

To overcome some of these limitations, many industrials introduced their own custom non-standard formats suitable for specific applications. For example, Intel has proposed an architecture called Nervana Flexpoint [96] which is suitable for **Deep Neural Network** (DNN) applications. The Microsoft Brainwave project [97] introduced a 9-bit float format for the same application class within a datacenter context. Similarly, Google introduced a 16-bit format called "bfloat" in their **Tensor Processing Unit** (TPU) architecture [98]. NVIDIA also introduced its custom 19-bit format called "TF32", implemented in dedicated tensor core accelerators [99]. In academia, works such as [100, 101] proposed FPUs with multiple-precision capabilities.

Although these works have solved many problems, there is still room for substantial improvement, especially in the field of variable precision or transprecision. Further details will be discussed in Chapter 3.

§4 Alternative number representations similarly to integer arithmetic, alternative number systems have been proposed to address a specific limitation of either FxP or FP.

For example, in order to define a FP format with inherent variable precision, the **Universal Number** (UNUM) number system was proposed by John L. Gustafson [102, 103]. The first version **Type I** UNUM had a variable-width storage format for both the exponent and mantissa and an additional bit used to determine if the number is an exact number or if it is an interval between two consecutive exact UNUMs. Subsequent computations on **Type I** UNUM numbers were performed using interval arithmetic. The second version **Type II** led to a redesign of the number system that made it incompatible with the IEEE 754 standard. Then came the third generation (**Type III**) that introduced two concepts: **POSITs** for FP-like operations, and **VALIDs** for interval arithmetic. **POSIT** is the most hardware-friendly version of the UNUM system. This format came to tackle a set of challenges faced during hardware implementation of the first two types. In fact, the feature that first motivated UNUMs, namely the variable bit-width, became the main bottleneck later during hardware implementation.

Given a FP format and a POSIT format with the same bit-width, POSITs perform equally or better than **Floating-Point** in terms of accuracy [104], since it enables a larger dynamic range and more fractional precision. However, in terms of hardware cost, authors of [105] have demonstrated that IEEE-754 FP operators remain between 30% and 60% faster and smaller than their POSIT counterparts. This is mainly due to the extra encoding and decoding steps required to unpack variable-size POSITs. This is also due to the higher accuracy that requires wider data-paths.

Other number systems such as the **Logarithmic Number System** (LNS) can also be used to represent real numbers in computer and digital hardware. This system is especially practical for digital signal processing [106, 107]. LNS is very efficient in terms of area and delay for multiplication and division. However, other operations such as addition, subtraction, and conversion are too complex, too costly. The overall global penalties prevents the use of LNS for general-purpose CPUs, except for a small niche of algorithms (when at least 70% operations are multiplications or 50-60% are divisions) or in an FPGA context. Otherwise, the impact on area and latency overhead is too important [108].

More disruptive ideas such as redundant arithmetic (as seen in Paragraph §1) have found their way to be part of FP computation. For example, in [109], the concept of "anytime instructions" has been introduced: enabling FP assembly instructions to be executed at a programmable accuracy, accuracy specified in the instructions' binary opcode itself, as opposed to fixed-length **Floating-Point**. The presented approach leads to an average 15% energy savings when computing a **Floating-Point** addition with an acceptable relative error of less than 0.1% and up to 39% energy savings for larger applications such as the Jacobi iterative algorithm

§5 ISA and cross-layer AxC approaches in the previous paragraphs we showed that ad-hoc individual AxC techniques can lead to very cost- and energy-efficient hardware and software systems when compared to standard and conventional components.

However, when doing a holistic and comprehensive system-level evaluation of these approaches, it results in more limited gains. For example, in [110], an error-tolerant multiplier achieving up to 46% energy savings has been proposed. Similarly, in [111], an approximate 16-bit adder is shown to achieve 30% energy savings. To study the overall system-level gains, these components are integrated in a RISC-V processor: the core has been augmented with an approximate adder and multiplier by extending the ISA with additional instructions and proper compiler support. For instance, exact addition operations use the assembly instruction "`add`" and are executed using the exact adder, whereas approximate additions are processed through approximate adders using the approximate assembly instruction "`a.add`. When estimating the energy dissipated in the entire processor, the authors found that it only achieves a maximum of 7.5% energy saving against an acceptable output image degradation, hence concluding that the benefits of using approximate *integer operators only* in embedded systems processors, have yet to be proved [26].

Others have taken more drastic approaches, by combining a set of different AxC techniques at once. One of the most prominent related works is done at the University of Washington [31]. The authors integrated DVFS with various other AxC techniques (*e.g.*, ISA extension, FP bit-width reduction, near-threshold computation, ...) into a single SoC model in order to evaluate the order of magnitude of the potential gains. The paper proposed an efficient and careful mapping of AxC programs onto hardware. An ISA extension including approximate operations and storage has been developed as well as a dedicated architecture, called Truffle, supporting that ISA extension. This architecture puts in practice the DVS concept since two voltage levels are used: a high (nominal) voltage for precise operations and a low voltage for approximate operations. Two processor micro-

architecture configurations have been proposed: an in-order one and an out-of-order one. Several applications have been evaluated demonstrating energy savings up to 43%.

The same team also worked on specialized languages and compilers (*e.g.*, EnerJ [56], ACCEPT [112]) that translate source code into specific binary code that is able to run on unreliable hardware. They also integrated approximate FP and integer arithmetic Units that are used when an instruction is required to execute in approximate mode. Other software techniques such as the skip-based approaches presented earlier have also been implemented in software. Such cross-layer approaches are the ones that leverage AxC for aggressive energy efficiency gains.

These works and others showed to which extent applying AxC only on elementary blocks (*e.g.*, adders, multipliers, ...) is not much effective at the system-level unless agressive cross-layer approaches are put together [13,32].

2.3 From AxC to Transprecision Computing: Toward Variable Precision Floating-Point

The **Approximate Computing** paradigm has known a clear shift in the last few years toward more adaptive, variable, and mixed-precision computing referred to as TC [27,113]. This paradigm targets mainly **Floating-Point** computations and storage, where the goal is to design more flexible, run-time variable precision and efficient architectures that trade accuracy against cost and resource savings.

2.3.1 The cost of Floating-Point Arithmetic

As mentioned in section 2.2.6.4, FPUs are ubiquitous in modern architectures, including **General-Purpose Processors** (GPPs) and **Application-Specific Instruction-Set Processors** (ASIPs), since they boost the performance of computationally intensive applications. However, FPUs occupy a significant part of CPU core area, and can be responsible for extensive power consumption and high memory bandwidth usage.

From a circuit area point of view, Figure 2.8 from [114] shows the area distribution of an entire RISC-V core called Ariane [115], including an FPU containing both standard formats (FP64, FP32) and transprecision reduced formats (FP16 and FP8) FMA. The FPU occupies a little more than 30% of the entire core with cache memories excluded.

Figure 2.9 presents the area distribution among the different components of a standard FPU[4], which clearly shows that the **Fused Multiplication-Addition** (FMA) blocks and the division/square root block take up to 81.4% of the total cell

[4]The RV64FD RISC-V standard extension architecture from [101,114] open-sourced at [116], synthesized for a 28nm FD-SOI technology, with a target frequency of 200MHz and automatic clock-gating enabled.

area of the FPU. The rest is occupied by the control logic and conversion/comparison blocks.

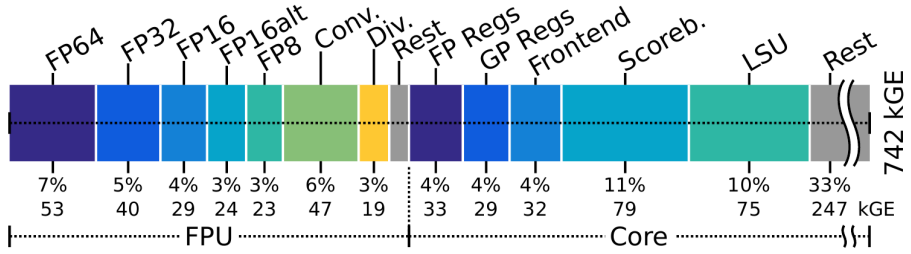


Figure 2.8: Area distribution of the entire Ariane RISC-V core, excluding cache memories (in kGE, 1 GE (gate equivalent) ≈ 0.199 μm^2) [114]

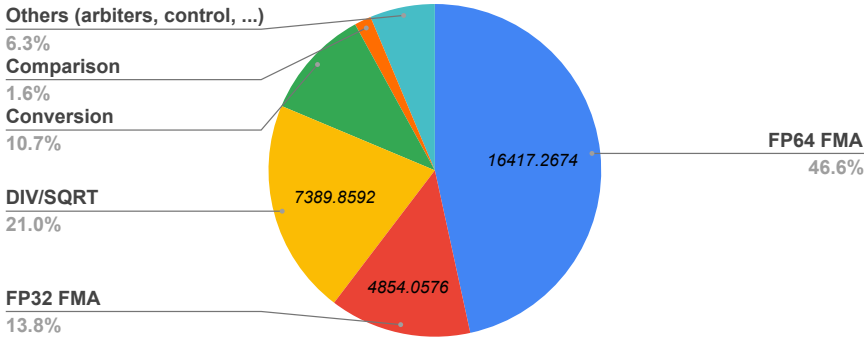


Figure 2.9: Cell area distribution of a standard RISC-V RV64FD FPU

From an energy efficiency point of view, the energy consumption associated with FP arithmetic and memory operations are known to be greater than their integer counterpart [117], making FPU optimization a priority. Figure 2.10 (extracted from [114]) depicts the contribution of each core element, including the LSU, FPU, ALU, caches *etc.* into the overall energy consumption of each RISC-V instruction (left axis). For example, 26% (yellow horizontal bars) of the energy dissipated during a double-precision FMA operation (*i.e.*, the RISC-V instruction `fmadd.d`) is consumed by the FPU. One can see also the great impact of other components such as instruction and data caches on the overall energy consumption (static and dynamic).

The need for wide dynamic range, as well as the software portability and flexibility ensured by FP have motivated the selection of **Floating-Point** over other number systems. However, there is a serious need for aggressive optimization of FP memory and computation usage.

2.3.2 The Need for Variable and Reduced Precision Floating-Point

In order to spot the limitations of existing works in the literature, especially in terms of reduced variable precision, we first present the typical FP algorithm implementation flow as shown in Figure 5.9. It is a five-step process.

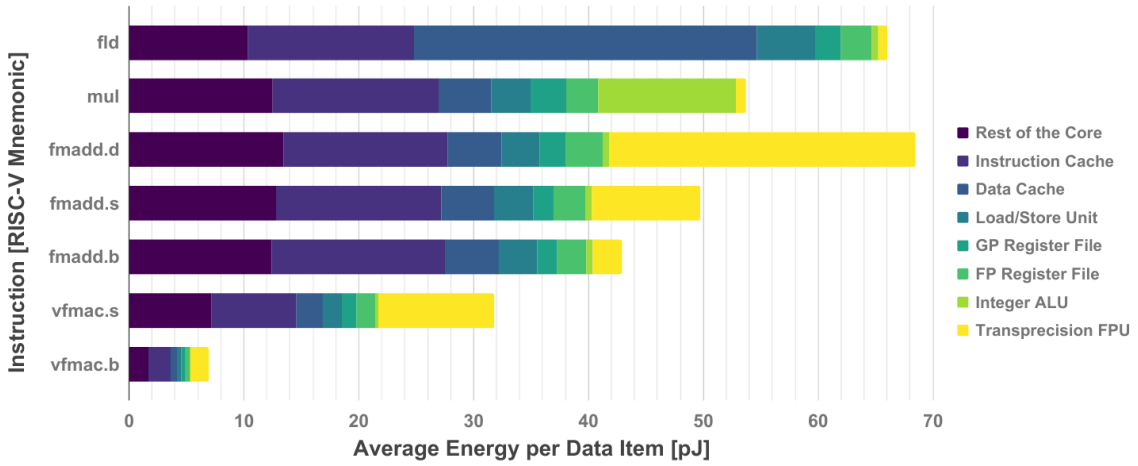


Figure 2.10: Energy dissipated by RISC-V instructions in the RISC-V Ariane core [114].

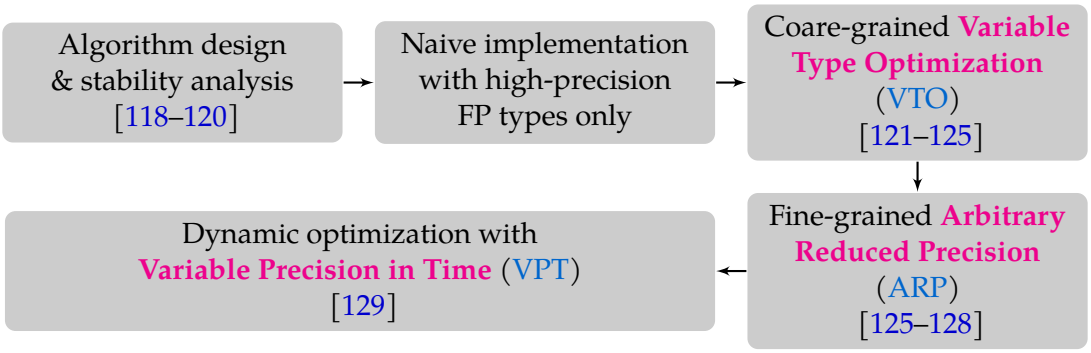


Figure 2.11: Typical FP application implementation flow.

§1 Algorithm design and stability analysis: this consists of constructing the mathematical foundations of the algorithm and verify its numerical stability *i.e.*, how it behaves vis-à-vis different inputs and edge cases. For example, some formulas can sometimes be accurate for some point in some intervals whereas they can be subject to rounding errors and cancellations in other intervals.

To this end, many tools have been proposed. For example Herbie [118] detects inaccurate expressions and finds more accurate replacements for each input interval. For instance, consider the following line:

$$\sqrt{x+1} - \sqrt{x} \longrightarrow \frac{1}{\sqrt{x+1} + \sqrt{x}}$$

The expression in the left side is inaccurate when $x > 1$; it is hence replaced by the equivalent but more accurate expression for all x values in the right side. These rewrites are usually done manually by FP experts. Herbie came to automate this process based on a search heuristic that estimates and localizes rounding error using sampled points, then applies a set of of rules stored in a database to generate improvements. It then takes series expansions, and combines improvements for different input intervals. Accuracy is improved overall while imposing a median performance (execution time) overhead of 40%. Herbgrind [119] is a tool based on Herbie and the famous dynamic binary instrumentation and profiling tool Valgrind [130]. Herbgrind analyzes binaries to find inaccurate FP expressions and

point out their places in the source code. It can also identify and characterize the inputs that produce unstable results. FpDebug [120] is another dynamic binary instrumentation tool based on Valgrind that supports the programmer in finding accuracy problems. FpDebug uses the concept of shadow computation: every FP computation is performed in higher precision in the background and compared to the actual result computed by the program. This helps to track errors' evolution and to detect catastrophic cancellation problems [131, 132].

§2 Variable Type Optimization (VTO) the next step is often a naive implementation, where all variables are declared in high precision formats *e.g.*, using `double` or `long double` type variables in C/C++. Next, the process of VTO [121–125] is overtaken. For a given constraint, its objective is to migrate as many variables as possible from high precisions to lower ones. For example, by carefully changing some `double` variables to `float` or `long double` variables to `double`, either manually or automatically, one can still satisfy a given QoR constraint on the output.

In this area, tools such as PROMISE [121] have been proposed. PROMISE is a tool that optimizes the numerical types in a program by taking into account the requested QoR constraint on the computed results. It is based on a technique called Discrete Stochastic Arithmetic (DSA) [133] that enables the estimation of round-off errors. This technique is coupled with a search heuristic called Delta Debugging [134], which enables a fast exploration of all the possible combinations of variable types in a source code. Many similar tools exist, such as Precimonious [123] which is also based on Delta Debugging. It is a dynamic program analysis tool that assists developers in tuning the precision of FP programs leading to speedup improvements as high as 41%. In [124], the authors augmented Precimonious with a new approach called Blame Analysis. The latter is a form of shadow computation, where FP instructions are performed in different levels of precision to determine the most optimized data type that enables a minimum level of QoR. This technique leads to up to 40% execution time gain and up to $38\times$ faster analysis time compared to Precimonious. Within the same context, we can find CRAFT [122], which migrates 64-bit double-precision operations to 32-bit single-precision by modifying the applications' binary executable instead of modifying their source code. Precision tuning is performed following a search algorithm that identifies code regions that can be migrated to lower precision and carefully replaces the double-precision FP instructions with their single-precision equivalent. Despite a $10\times$ execution time overhead spent on analysis compared to other SoA works, the tool produces modified programs that run twice as fast in average.

§3 Arbitrary Reduced Precision (ARP) the VTO step is a limited coarse-grained optimization process since it does not enable to optimize data bit-width and precision down to the bit level. To overcome this limitation, ARP can be used [135]. Its goal is to perform a fine-grained bit-width optimization. It consists of reducing the exponent and/or mantissa bit-widths to either narrower standard bit-widths such as the 16-bit `binary16` format [28], or custom reduced arbitrary formats such Intel Nervana's Flexpoint format [96], Microsoft Brainwave's 9-bit floats [97], Google

TPU's 16-bit BFloats [98], and NVIDIA's 19-bit format [99], or even craft new custom arbitrary formats.

To this end, tools such as fpPrecisionTuning [125] share the same goal as VTO but with a different approach: a distributed algorithm analyzes the variables and transforms **Floating-Point** signal processing programs to their arbitrary precision **Fixed-Point** equivalent, hence enabling arbitrary bit-width reduction. This leads to about 82% and 66% average reduction in hardware resources when implemented on FPGA when compared to the standard double-precision and single-precision versions with no quality degradation, respectively.

Other SoA tools and libraries mainly focused on the software simulation of the impact of such approximate behaviour on applications' QoR. However, these tools / libraries do not provide hardware architectures that take advantage from their results, so they still run on full-precision hardware since the programs are only simulated in software. Among these works, we find the GNU MPFR (Multiple Precision Floating-Point Reliable) library [126] and GNU Multiple Precision Arithmetic (GMP) library [136]. These are C libraries for multiple-precision **Floating-Point** computations. They provide special data types with configurable precision as well as computational functions and they are more suitable to simulate extremely high precisions *e.g.,* hundreds of mantissa bits. Other libraries such as FlexFloat [128] also provide both specific C data types for arbitrary precision simulation and elementary functions. However, it only provides arbitrary reduced precisions and does not support arbitrary higher ones. Similarly, FloatX [127] is an improved C++ version that leverages modern programming language ideas such as templates, operator overloading, *etc.* to simplify its integration in existing applications and minimize manual source code modification. Both FlexFloat and FloatX are more efficient than MPFR and GMP for reduced precisions in terms of simulation time. The common point for all these libraries is that they all need manual rewriting of the source code in order to simulate the effects of FP approximations on software applications' **Quality of Result**.

In [29], a variable precision framework that accelerates scientific computing applications has been introduced. Specifically, the authors presented a **Floating-Point** RISC-V coprocessor able to reach up to 512 bits of precision based on the UNUM format (Paragraph §4) along with an extension of the RISC-V ISA and a corresponding programming model that enables the acceleration of applications written with the GMP library. A speed up of up to $18\times$ has been observed while keeping computational errors within the same order of magnitude as with GMP. The advantage of this work is that it addresses many of the limitations mentioned earlier, especially by providing hardware acceleration for variable precision. However, this accelerator is only dedicated to higher precisions (64 to 512 bits) and does not support reduced ones.

§4 Variable Precision in Time (VPT): VPT is a further fine-grained FP cost optimization technique that started to emerge for CPU-based architectures. It allows precision to vary arbitrarily not only statically in source code (spacial variation), but also at run-time (temporal variation). For example, let's assume a **Floating-Point** `double` (64-bit double-precision format) variable x that is declared inside a loop. If VTO is applied, the variable x would be demoted to `float` for example (32-bit single-precision format). If we further optimize using ARP, we

could reduce its size to, let's say 20 bits, but this size would be fixed during the whole execution time. VPT can further optimize computations, by choosing lower precisions depending on the computed results during execution time. We can imagine starting with a very short format *e.g.*, 10 bits, then increase it to 12, then 15, and so on and so forth until reaching 20 bits, hence achieving more aggressive optimization. If the application is error-resilient and computational errors are tolerated, AxC can be leveraged to further reduce the precision and hence improve the energy efficiency.

To this end, authors of VPREC [129] have proposed an automatic tool that evaluates the effect of precision variation for each computational operation (*e.g.*, addition, multiplication, ...) over time, which is particularly useful in iterative schemes. The proposed tool emulates standard FP operations in lower precision. It has been applied for a large **High-Performance Computing** (HPC) application, achieving 28% to 67% reduction in communication volume. However, as we will see in Chapter 6, this methodology has also its limitations. The most relevant one here is the fact that it is only intended for software simulation and there is no proper hardware to support such behaviour.

2.3.3 Summary comparison of Transprecision Computing SoA techniques

Table 2.3 presents a short comparison between the different **Transprecision Computing** SoA tools presented previously. The comparison is based on six criterias hereafter discussed.

Localization all the presented tools, except VPREC, inject the **Floating-Point** approximations in the source code or in the executable binary *i.e.*, only the spatial dimension is explored. This means that, no assembly instruction can change its precision at run-time. VPREC on the other hand enables exploring the temporal dimension *i.e.*, an instruction can vary its precision in time. In order to also explore the spatial aspects with VPREC, the latter should be used after having performed VTO with another tool such as Promise for example.

Support for mixed-precision and arbitrary precision all the presented tools have support for mixed-precision *i.e.*, they allow to have mixed single- and double-precision operations and variables in one program. Arbitrary precision signifies fine-grained bit-level precision variation. VPREC and Promise are the only tools based on Discrete Stochastic Arithmetic (DSA). DSA is a technique that consists on injecting small rounding error variations at the level of instructions in order to evaluate the overall resiliency of applications.

Support for automatic search automatic tools are able to provide search algorithms to find the best suitable variable types' configuration that optimizes cost. Whereas non-automatic tools require rewriting the application and either implementing specific search methods or relying on experts' knowledge to decide which variables can be migrated to low precisions.

Granularity in terms of granularity, we distinguish between coarse-grained and fine-grained tools, which is directly related to the support for arbitrary precision. The first five ones are coarse-grained due to the fact that they only allow applications to use standard FP precisions (32 / 64 / 80 / 128 bits). The remaining tools are fine-grained since they enable the user to control the precision at the bit level.

Intrusiveness the presented tools are very intrusive to source code or to the binary executable. However, this intrusiveness is masked for the ones that are automatic, since the tools handle the transformation process instead of the programmer. Although, from our experience, some manual tweaking is always needed to get the tools and the scripts working, but this is a purely technical limitation.

Tool	Localization	Mixed prec.	Arbitrary prec.	Round. Error	Automatic	Granularity
Precimonious [123]	Spatial	✓				Coarse
Blame Analysis [124]	Spatial	✓			✓	Coarse
Promise [121]	Spatial	✓		✓		Coarse
CRAFT [122]	Spatial	✓			✓	Coarse
fpPrecisionTuning [125]	Spatial	✓			✓	Coarse
MPFR / GMP [126, 136]	Spatial	✓	✓[1]			Fine[1]
FlexFloat [128]	Spatial	✓	✓[1]			Fine[1]
FloatX [127]	Spatial	✓	✓[1]			Fine[1]
VPREC [129,137]	Temporal	✓	✓[1]	✓	✓	Fine[1]

[1] Behaviour supported in simulation only.

Table 2.3: Comparison of **State of the Art** *tools (Inspired from [129])*

From the previous comparative study we conclude that the most **State of the Art** technique is VPREC [129] . This work will constitute the SoA baseline against which our contributions will be compared, especially in Chapter 6. Although VPREC was proposed and used for the **High-Performance Computing** context and for scientific computing applications, our work builds on top of the existing **Variable Precision in Time** concept. Our work aims to alleviate some of its limitations but in the context of TC applied to embedded and application-class **General-Purpose Processors** for error-resilient applications.

2.4 Conclusion

In this chapter, we saw an overview of the existing SoA tools and techniques in the AxC topic in general. Then we saw how the field evolved to **Transprecision Computing** with a focus on **Floating-Point** computation and storage. After that, a summary of existing tools related to FP program optimization techniques was presented.

Through this brief overview, we conclude that there is still room for improvements. In this manuscript, we will focus on the following improvements :

- Proposing both a hardware-friendly FP approximation approach and a tool that enables non-intrusive simulation of applications' QoR using reduced arbitrary precision (Chapters 3 and 4)
- Providing a hardware floating-point unit that takes advantage from the proposed approach and tool to achieve significant energy efficiency gains (Chapter 5).
- As a direct application of the whole developed features, proposing an iterative algorithm scheme that enables precision to vary in time in a self-adaptive way without programmer intervention (Chapter 6).

3

A Non-intrusive Approach for Floating-Point Approximation

Contents

3.1 Introduction

Approximate Computing and **Transprecision Computing** applications are by definition error-resilient. Many SoA works have proposed leveraging this aspect for energy efficiency, performance, and complexity/area optimization. However, most existing techniques are very hard to use because of many limitations highlighted in the SoA (Section 2.3.3).

First, existing tools are mostly static, *i.e.*, they allow approximations at design-time or compilation-time only and they are fixed during execution time. Once the software application is compiled to binary code, it is no longer possible to adjust these approximations. For example, Precimonious [123] takes a software application source code as an input in C/C++ and generates a source code with an optimized number of high-precision variables with respect to a given input dataset. This means that these approximations are injected at design time (during the development). Once the generated source code is compiled, the approximations will be static and fixed during all executions, regardless of the variations in inputs.

Second, other tools allow approximations to be dynamic, *i.e.*, the precision can vary at run-time. For instance, the FlexFloat [128] (and its C++ successor FloatX [127]) library allows programmers to use approximate **Arbitrary Reduced Precision** FP formats instead of the standard `float`, `double`, and `long double` formats in C/C++. These libraries offer two versions of custom data structures: some with static/fixed precision and others with dynamic precision. The ones with static precision are more efficient to simulate. The dynamic ones take slightly more space in memory and more execution time to be simulated, but the simulation cost is affordable. However, these tools are only made for approximation impact simulation, *i.e.*, they only help understand the effects of precision reduction on the applications' QoR by simulation on top of exact FPU hardware. The reason is that they don't provide an appropriate hardware implementation that can take advantage of its conclusions and actually evaluate energy efficiency.

In summary, most of the existing approaches go against the guiding principles explained earlier in Section 1.1.3: 1) most of them are intrusive to source code, 2) the ones supporting arbitrary reduced precision are only adequate for simulation purposes with no adequate underlying hardware at hand, 3) they are application-specific and data-dependant, and 4) they are static.

In this chapter, we propose a new approach that overcomes some of the SoA limitations. The main idea of the approach is that, instead of operating at the design stage by rewriting/transforming the source code or even the binary executable, the approximations are introduced at run-time. With such an approach, the methodology can be implemented either for software simulation (in this case, we talk about interpretation, *C.f.*, Chapter 4) or hardware (in this case, we talk about execution, *C.f.*, Chapter 5).

Before going through the details of the approach, we will first overview the **Floating-Point** model and underline the required features necessary to a clear understanding of the approach (Section 3.2). Then a description of the approach will be presented in Section 3.3. After that, we will discuss a formalization of the proposed method in the form of a **Finite-State Machine** model in Section 3.4.

3.2 Background: The Floating-Point (FP) model

This section will present the basics of the **Floating-Point** representation, starting first from the standard IEEE 754-2008 [28] formats and generalizing to arbitrary precision arithmetic. Finally, we will end this section with a presentation of the terminology used in this manuscript and the problem statement.

3.2.1 Definitions

Floating-Point arithmetic is a way to represent real numbers with a finite number of bits approximately. It has been introduced to provide flexibility between range and precision, especially when numbers of different magnitudes are used in the same system. That is why it has found its way to mainstream **General-Purpose Processors**.

A binary **Floating-Point** number x can be written in the form:

$$\begin{cases} (-1)^s \times (1 \,.m) \times 2^{e-b} & \text{if } x \text{ is normal.} \\ (-1)^s \times (0 \,.m) \times 2^{1-b} & \text{if } x \text{ is subnormal.} \end{cases} \tag{3.1}$$

where :

- s is the sign bit,
- e is the exponent. Its bit-width is denoted E.

$$e = e_0 \, e_1 \quad \dots \quad e_{E-2} \, e_{E-1}$$

The exponent e is biased, , *i.e.,* the value stored is equal to the actual exponent plus a positive number called the bias b. The practical reason behind that is simplifying the comparison of exponents in hardware by representing them in an always positive form. The exponent bias b depends on the bit-width of the exponent:

$$b = 2^{E-1} - 1$$

Let $\mathcal{E}$ denote the (actual) unbiased exponent *i.e.,* $\mathcal{E} = e - b$.

- m is the mantissa[1] (or fraction). Its bit-width, denoted M, is called the *precision* of the representation.

$$m = m_0 \, m_1 \quad \dots \quad m_{M-2} \, m_{M-1}$$

The dot in the expressions $(1.m)$ and $(0.m)$ represents the decimal point. The leading bit in these expressions is called the hidden bit and is denoted h. It is equal to 1 in normal FP numbers and equal to 0 in subnormal numbers.

[1] Please note that in our case, the mantissa does not include the hidden bit.

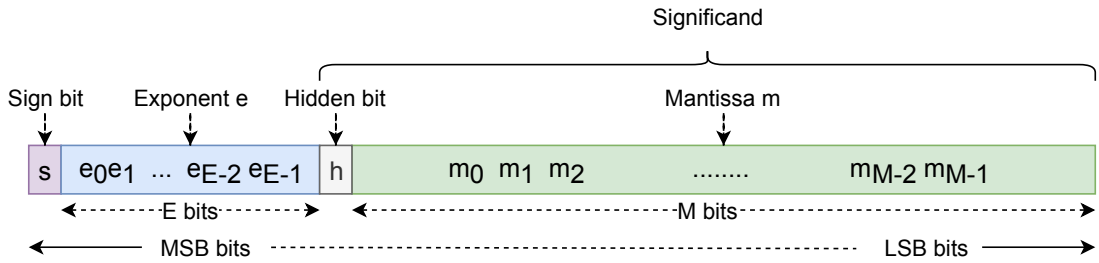


Figure 3.1: **Floating-Point** *number representation layout.*

Definition 3.2.1. The **hidden bit** h is a bit that specifies the nature of the FP number being handled. When the exponent and the hidden bit are both equal to zero, the number is called a **subnormal** or **denormal number**; otherwise, it is called a **normal** number.

Figure 3.1 depicts the layout of FP numbers in memory. Typical hardware implementations contain an additional bit called the "hidden bit" h defined as follows:

The significand includes the hidden bit h and the fraction m

$$\text{significand} = h.m_0\, m_1 \quad \ldots \quad m_{M-2}\, m_{M-1}$$

To avoid storing the hidden bit in memory and use its space to gain more precision, the **Floating-Point** standard uses the exponent and the mantissa to deduce the value of the hidden bit; when the exponent e is equal to zero and the mantissa m is non-zero, the hidden bit is set to zero. Otherwise, it is set to 1.

Definition 3.2.2. A **Floating-Point** format is defined by the pair (E, M), where E is the bit-width of its **exponent** and M is the bit-width of its **mantissa** field.

For each format (E, M), a set of parameters can be derived:

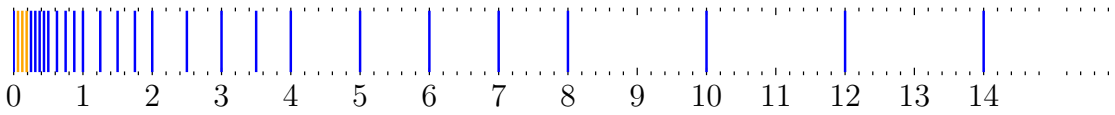
- The largest unbiased exponent value: $\mathcal{E}_{max} = 2^{E-1} - 1$
- The smallest unbiased exponent value: $\mathcal{E}_{min} = 1 - \mathcal{E}_{max}$
- The largest representable normal number: $x_{max} = (2 - 2^{-M)}).2^{\mathcal{E}_{max}}$
- The smallest representable normal number: $x_{min} = 2^{\mathcal{E}_{min}}$
- The smallest representable denormal number: $2^{\mathcal{E}_{min} - m}$

Table 3.1 summarizes how to interpret an FP number depending on the value of the exponent e and the mantissa m according to Equation 3.1. For example, if e equals zero and the mantissa is non-zero, the represented number would be denormal, and the hidden bit h is set to zero. In this case, the number would be equal to $(-1)^s \times 2^{(1-b)} \times (0.m)$. As a second example, when the exponent is equal to $2^E - 1$, *i.e.*, $111\ldots1$ in binary, the number would represent either $\pm\infty$ if the fraction is equal to zero, or a NaN (Not a Number) if the fraction is non-zero.

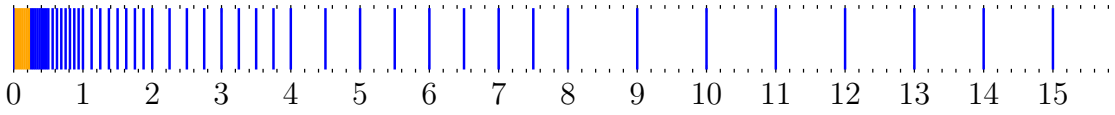
We will use the notation (E, M) in the remainder of this thesis to describe the formats supported within a given FP system.

Exponent e	fraction = 000 ... 0	fraction $\neq$ 000 ... 0	Equation
000 ... 0	zero	subnormal number	$(-1)^s \times 2^{1-b} \times 0.m$
$0 < e < 2^E - 1$	normal number		$(-1)^s \times 2^{e-b} \times 1.m$
111 ... 1	$(-1)^s \times \infty$	NaN (Not a Number)	—

Table 3.1: Interpretation of FP numbers



(a) Numbers distribution in the (3, 2) example FP system.



(b) Numbers distribution in the (3, 3) example FP system.

Figure 3.2: Effect of precision on numbers' distribution

3.2.2 FP numbers through a simple case

In this paragraph, we show through a simple case how the different parameters (exponent bit-width, mantissa bit-width) influence numbers' distribution.

Consider a binary FP system with the FP format $(E, M) = (3, 2)$. Table 3.2 depicts the encoding of all the representable numbers in this format and their type and corresponding decimal values. Since the FP system is symmetric, we only represent the positive numbers.

This format's bias would be $b = 2^{3-1} - 1 = 3$, the largest unbiased exponent value $\mathcal{E}_{max}$ is equal to 3, and the smallest $\mathcal{E}_{min}$ is equal to -2. The smallest normal value would be $x_{min} = 2^{\mathcal{E}_{min}}$ which is equal to 0.25 in this case. The largest normal value would be $x_{max} = (2 - 2^{-M}).2^{\mathcal{E}_{max}}$ which is equal to 14.0. Supporting subnormal numbers adds three additional numbers between 0 and x_{min}.

When including the symmetric negative space, one can see that there are two representations for ∞ and $2 \times (2^M - 1)$ (in this case 6) representations for NaN, which is obviously a waste of encoding space. Some custom FP systems in the State of The Art eliminate redundant NaN representations to encode more values or convey specific information.

Figure 3.2 represents the distribution of the finite numbers ($\pm\infty$ and NaNs not included) that are representable in two example FP formats. Subfigure 3.2a depicts the numbers representable in the (3, 2) FP format. Subfigure 3.2b depicts the numbers representable in the (3, 3) FP format. Normal numbers are represented in blue, whereas subnormal numbers are plotted in orange.

As you can see, the numbers 4.5, 5.5, 9, 11, 13, 15 (and others) cannot be represented in the (3, 2) format. However, the (3, 3) format can represent them exactly. Increasing precision from M to $M + 1$ fills the gap between two consecutive FP numbers with an additional number, increasing the distribution density. More-

over, the numbers that are represented in a format (E, M_1) are also representable in formats with higher precisions (E, M_2), where $M_2 > M_1$.

Exponent e	mantissa m	Type	Decimal value
000	00	zero	±0.0000
000	01	subnormal	±0.0625
000	10	subnormal	±0.1250
000	11	subnormal	±0.1875
001	00	normal	±0.2500
001	...		
001	11	normal	±0.4375
010	00	normal	±0.5000
010	...		
010	11	normal	±0.8750
011	00	normal	±1.0000
011	...		
011	11	normal	±1.7500
100	00	normal	±2.0000
100	...		
100	11	normal	±3.5000
101	00	normal	±4.0000
101	...		
101	11	normal	±7.0000
110	00	normal	±8.0000
110	...		
110	11	normal	±14.0000
111	00	$\pm\infty$	
111	01	NaN	
111	10	NaN	
111	11	NaN	

Table 3.2: Example: the (3,2) example FP system.

3.2.3 Standard formats Vs. arbitrary formats

The IEEE 754-2008 [28] proposes many standard formats starting from the 16-bit `binary16` format up to 256-bit format `binary256`. Table 3.3 presents some IEEE standard formats and other specific arbitrary formats proposed in the SoA with their characteristics.

As you can see, the bit-widths E and M can be adjusted arbitrarily depending on the target application and the need in terms of dynamic range and desired precision, especially when some loss of precision is tolerated.

Type	Name	E	M	Total bit-width $1+E+M$	Exponent bias $b=2^{E-1}-1$	Number of decimal digits $\log_{10}(2^{M+1})$
Standard IEEE formats [28]						
Half-precision	`binary16`	5	10	16	15	~3.3
Single-precision	`binary32`	8	23	32	127	~7.2
Double-precision	`binary64`	11	52	64	1023	~15.9
Quad-precision	`binary128`	15	112	128	16383	~34.0
Octuple-precision	`binary256`	19	237	256	262143	~71.6
Other formats [97,99,100]						
8-bit smallFloat	`Xf8`	5	2	8	15	~0.9
Minifloats	–	4	3	8	7	~1.2
Brain Floating Point	`bfloat16`	8	7	16	127	~2.4
Tensor Float 32	`TF32`	8	10	19	127	~3.3
x86 extended precision	–	15	64	80	16383	~19.2

Table 3.3: **Floating-Point** *standard formats*

These formats' precisions are fixed since they are implemented in hardware, and their associated software is also fixed. In contrast our technique aims to support multiple precisions in a flexible way as will be described later. Ultimately, software applications should be able to select the operating precision at run-time depending on the application's needs in terms of accuracy.

3.2.4 Rounding and extension

As our approach is based on FP format reduction and extension (to be detailed in the next subsection), we first introduce how the standards define these operations. In addition, when multiple FP formats are often used in the same application, conversions from a format to another are then needed.

Rounding

Definition 3.2.3. We call **rounding** the operation of converting a number from a source format to a destination format with less precision.

For example, in base 10, rounding the elementary charge q_e, which is equal to $1.602176634 \times 10^{-19}$ to 3 digits after the decimal point with result in either 1.602×10^{-19} (round down or truncate) or 1.603×10^{-19} (round up). Many other rounding modes can be imagined depending on the fourth digit after the decimal point.

The IEEE 754-2008 [28] defines the following four rounding modes :

- **Round towards $+\infty$ (or round up)**: rounds the source number to the closest number representable in the destination format that is larger or equal.

- **Round towards $-\infty$ (or round down)**: rounds the source number to the closest number representable in the destination format that is smaller or equal.
- **Round towards 0 (or truncate)**: rounds the source number to the closest number representable in the destination format that has equal or smaller absolute value.
- **Round to nearest (or unbiased rounding)**: rounds the source number to the closest number representable in the destination format. If the destination number is exactly halfway between two numbers, a tie-breaking rule should be applied. By default, a tie-to-even rule is applied, which chooses the one with a pair mantissa.

Table 3.4 depicts some rounding examples in base 10, where numbers are rounded from 2 digits after the decimal point to 1 digit.

	Round towards $+\infty$	Round towards $-\infty$	Round towards 0	Round to nearest
1.52	1.6	1.5	1.5	1.5
-1.52	-1.5	-1.6	-1.5	-1.5
3.15	3.2	3.1	3.1	3.2

Table 3.4: Rounding examples

Many other rounding modes can be found in the SoA, for example, rounding away from zero, faithful rounding, stochastic rounding [126, 133, 136] , *etc.*

Extension We define an extension as the opposite operation of rounding: converting from a source format to a destination format with different mantissa bit-width M and different exponent bit-width E. In this case, the mantissa is padded with trailing zeroes. As for the exponent, it should first be unbiased by computing

$$\mathcal{E} = e_{\text{src}} - b_{\text{src}}$$

and then re-biased by computing the final destination exponent

$$e_{\text{dest}} = \mathcal{E} + b_{\text{dest}}$$

Where $\mathcal{E}$ is the unbiased exponent of the represented number, e_{src} (resp. b_{src}) is the exponent represented in the source format (resp. the bias of the source format). Similarly, e_{dst} (resp. b_{dst}) is the exponent represented in the destination format (resp. the bias of the destination format). The result represented in the destination format will then have the same sign bit, the destination exponent e_{dest}, and the zero-padded mantissa.

Other conversions In addition to conversions from a Floating-Point format to another, the IEEE standard and many Instruction Set Architectures also define FP to integer and integer to FP conversions. These are invoked when an FP variable is cast to an integer variable. Another case where these conversions intervene

is when an integer variable is part of an FP operation. In this case, the integer variable is first cast to an FP format before performing the operation. These conversions can be introduced when needed either by the programmer or automatically by the compiler.

3.3 Presentation of the Approach

Our approach overcomes some challenges facing existing **Transprecision Computing** techniques by operating a the interpretation/execution stage rather than at the design stage. To illustrate our methodology, we will start from the simple example source code depicted on the left side of Table 3.5 and compare our technique with a classic tool. Here we chose the GNU MPFR library [126] .

Intrusiveness of existing techniques MPFR and other libraries such as FlexFloat [128] and FloatX [127] allow designers to express variable precision behaviour. MPFR in particular is popular in the domain of scientific computing where hundreds of bits of precision are needed to compute extremely accurate results. Nonetheless, it can also be used in the context of AxC where errors are tolerated to simulate ARP.

```
// Declare variables
float  a = 2.69;
double b = 3.65364294;
float  c = 6.2451;
double d ;

// Perform FMA operation
d = a * b + c;
```

```
// Declare variables
mpfr_t a, b, c, d;
mpfr_t temp; // temporary intermediate variable

// Set mantissa bit-widths
mpfr_init2 (a,    15);
mpfr_init2 (b,    42);
mpfr_init2 (c,    20);
mpfr_init2 (d,    50);
mpfr_init2 (temp, 47);

// Initialize variables
mpfr_set_d (a, 2.69,       MPFR_RNDN);
mpfr_set_d (b, 3.65364294, MPFR_RNDN);
mpfr_set_d (c, 6.2451,     MPFR_RNDN);

// Perform operations
mpfr_mul (temp, a, b,    MPFR_RNDN);
mpfr_add (d,    d, temp, MPFR_RNDN);
```

Table 3.5: Code transformation using the GNU MPFR [126] library.

Table 3.5 shows a manually-transformed source code using MPFR on the right side. The transformation process consists of declaring the variables `a`, `b`, `c` and `d` with a special data type `mpfr_t` provided by the library. For complex algorithms, there is always a need for temporary variables to store intermediate computation results. The programmer should also assign a precision to each variable based

on the needed output accuracy. After that, each variable should be assigned an initial value with some rounding mode (here, we use the round-to-nearest mode specified by the macro `MPFR_RNDN`). Finally, each complex FP operation should be divided into multiple elementary operations organized such that the dependencies and the order of the original operations is satisfied.

This example illustrates to which extent existing techniques can be very tedious and complex for larger applications if done manually. This task should be performed each time the original application's source code is modified. It also involves the choice of the appropriate precision for each individual variable. Although this process can be automated to some extent, from our experience, manual tweaking is always needed to get the scripts and the final application working, which is not obvious when dealing with auto-generated source code. As shown in Section 2.3.3, some tools propose automation tools and search heuristics that help designers find the adequate precision for each variable. Nonetheless, these techniques do not scale well for applications with dozens of variables nor more complex algorithms or libraries.

Approximation at the execution / interpretation level from this perspective, we conclude that an approach that rather operates at the execution stage would offload many tasks that, otherwise, should have been performed manually by the programmer.

Figure 3.3 illustrates our proposed methodology. We present the general flow on the right side, and on the left side, we provide an example of an application to an FMA operation.

Our flow consists of compiling the original source code normally using a typical C/C++ compiler. This generates a binary executable with standard assembly instructions. In the example presented here, the FMA statement results in the RISC-V assembly instruction `fmadd.d` which takes as inputs the registers `RS1`, `RS2`, `RS3` and as an output the register `RD`. In standard precision, this instruction would normally be interpreted at run-time by computing `RS1` $\times$ `RS2` + `RS3` and storing the result in the output register `RD`. This is the exact interpretation of the assembly instruction. In our case, we also define a new execution / interpretation[2] mode: the approximate mode.

In the approximate mode, the computation is executed with a reduced precision to optimize power consumption. To do that, the inputs should first be reduced, by rounding the inputs `RS1`, `RS2`, `RS3` down to the target reduced precision (using the function `Reduce()`). Then performing the reduced-precision computation. Finally, the results should be extended (using the function `Extend()`) from the intermediate reduced precision to the original standard precision of the inputs.

Support for multiple precisions or approximation modes we can further improve the proposed approach to support multiple reduced precisions at run-time

[2]Interpretation and execution are used interchangeably here to reflect the fact that our methodology can be used either for simulation (interpretation) or a real-world hardware FPU implementation (execution).

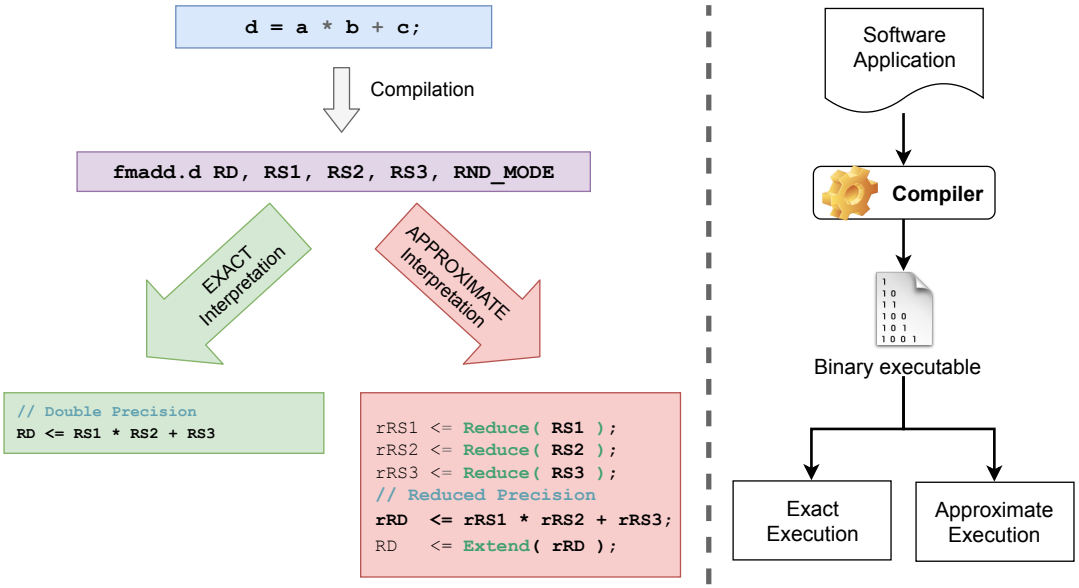


Figure 3.3: The proposed approach.

or **Variable Precision in Time** (VPT). This is achieved by enabling the exact interpretation mode and multiple approximation modes instead of only one, as illustrated in Figure 3.4. For each approximation mode i, we associate a reduced precision M_i. This will enable programmers to modify the computations' precision at run-time.

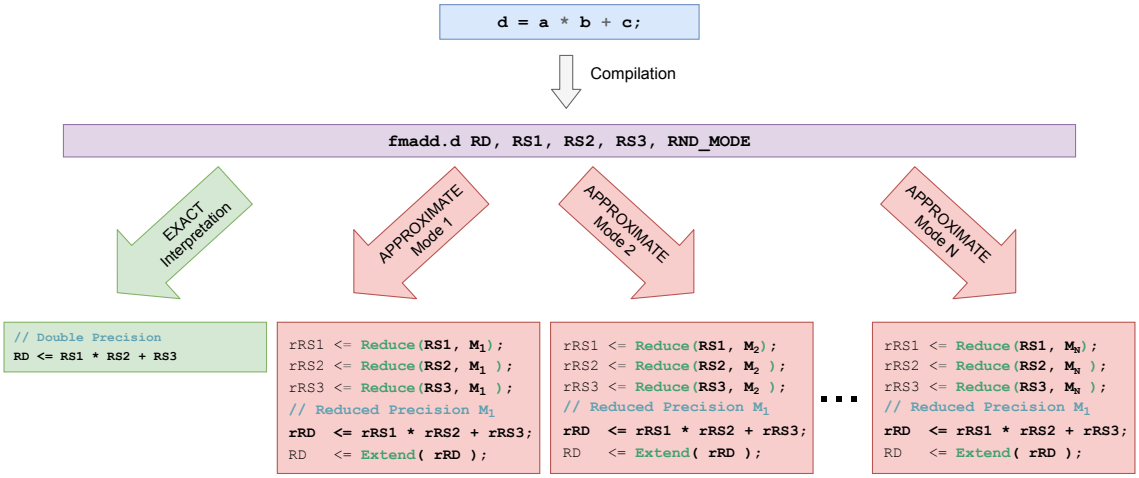


Figure 3.4: The proposed approach with support for multiple precisions.

Choosing the interpretation / execution mode the execution mode is to be specified by the programmer and can be changed at run-time. Programmers can specify which functions can be approximated and in which mode they are supposed to be executed. This process is called **Selective Approximation** (SA) and it will be detailed in Section 3.4.4.

Software Vs. Hardware implementation in our methodology, all the approximations are introduced at the interpretation level at run-time. In this thesis, we

implemented the approach for two purposes :

- For impact simulation: in this case, the approach is implemented in software using the QEMU functional simulator. This will enable programmers to evaluate the impact of reduced precision on applications' QoR while being non-intrusive or at least minimally intrusive to source code. We will discuss details of this implementation in Chapter 4.
- For energy efficiency optimization: in this case, the approach is implemented in hardware as a multi-precision FPU. This will enable the evaluation of energy and execution time savings brought by the FP reduced precision approximations. We will discuss details of this implementation in Chapter 5.

In the following section, we will present a formalization of the approach described earlier in the form of an FPU FSM model.

3.4 Formalization of the Approach

This section introduces some definitions and a formalization of an **Floating-Point Unit** in the form of an FSM model. We also give insights into the selective approximation and the problem statement.

3.4.1 Definitions & Notations

Consider a **Floating-Point Unit**, let $\mathcal{A}$ be the set of its registers and $\mathcal{R}$ be the set of available rounding modes.

Let $\mathcal{X}_{SP} = \{L_1, L_2, ...\}$ be the set of Standard Precision formats supported (*e.g.*, `binary64`, `binary32`), and $\mathcal{X}_{ARP} = \{l_1, l_2, ...\}$ be a set of non-standard **Arbitrary Reduced Precision** FP formats. In the following, we denote L a format in $\mathcal{X}_{SP}$, l a format in $\mathcal{X}_{ARP}$ and ℓ a format in $\mathcal{X}_{SP} \cup \mathcal{X}_{ARP}$. We can also use L, l, and ℓ to refer to the total bit-width of the format *e.g.*, an L-bit format is a format whose total bit-width $(1 + E + M)$ is equal to L.

We augment the FPU with an additional set of **Control and Status Registers** (CSRs) denoted $\mathcal{A}_{\text{ARP}} = \{\text{PREC}_{L_1}, \text{PREC}_{L_2}, ...\}$ that will be dedicated to precision selection; each operation that is originally supposed to be executed on a format L_i will instead be executed on a precision selected by the register PREC_{L_i}. A function Select_Prec that maps high precision formats L_i to their corresponding reduced precision formats l_i pointed to by the CSRs PREC_{L_i} is defined as follows:

$$\text{Select\_Prec} : \mathcal{X}_{SP} \rightarrow \mathcal{X}_{ARP}$$

Let $\mathbb{F}_\ell$ denote the set of FP numbers that can be represented in the ℓ format.

Let $\mathcal{F}$ be the set of FP instructions available for a given ISA. This set is partitioned in two sets $\mathcal{F}_{approx}$ and $\mathcal{F}_{exact}$, where $\mathcal{F}_{exact}$ represents the non-computational FP instructions (comparison, conversion, loads/stores, and sign injection), whereas

$\mathcal{F}_{approx}$ represents the FP computational instructions: addition, subtraction, multiplication, fused multiplication-addition, square root, and division[3].

We define the precision reduction function

$$Reduce^{R}_{L,l} : \mathbb{F}_L \rightarrow \mathbb{F}_l$$

that reduces an L-bit number to an l-bit one using rounding mode $R \in \mathcal{R}$. We also define the reverse function

$$Extend_{L,l} : \mathbb{F}_l \rightarrow \mathbb{F}_L$$

that extends an l-bit number to produce an L-bit one. While $Reduce^{R}_{L,l}$ is a many-to-one function, $Extend_{L,l}$ is a one-to-one function.

Let $F\_inst^{R}_{\ell}\ rd, rs_1, \ldots, rs_n$ be a FP instruction, where F_inst is in $\mathcal{F}$, and operates on $n+1$ registers $rd, rs_1, \ldots, rs_n$, of ℓ-bit values, using the rounding mode R. With each $F\_inst^{R}_{\ell}$ we associate an arithmetic operator

$$F\_op^{R}_{\ell} : (\mathbb{F}_\ell)^n \rightarrow \mathbb{F}_\ell$$

which performs the computation in the ℓ format using the rounding mode R.

To introduce **Selective Approximation** (SA), as a means to apply approximations to a specific set of instructions, let $\mathcal{B}$ be the Boolean domain and `enable_approx` $\in \mathcal{B}$ an enable signal that indicates whether the instruction should be approximated at run-time.

3.4.2 Approach formalization

The FPU can be modeled by an FSM $\{\mathcal{I}, \Gamma, \gamma_0, \delta\}$ with no output, where $\mathcal{I} = \mathcal{F} \times \mathcal{R} \times \mathcal{X}_{SP} \times \mathcal{A}^{n+1} \times \mathcal{B}$ is the input of the FSM, representing an instruction instance characterized by its name, its rounding mode, its original FP format, its destination and source registers, and an approximation enable signal. Γ is the set of states[4], γ_0 is the initial state where all the registers are set to zero, and δ the state-transition function, which is defined as follows:

$$\begin{aligned} \delta' :\ & \mathcal{I} \times \Gamma && \longrightarrow \Gamma \\ & (F\_inst^{R}_{L}\ rd, rs_1, \ldots, rs_n, \texttt{enable\_approx}), \gamma && \longmapsto \gamma' \end{aligned}$$

such that $\forall r \in \mathcal{A}$:

$$\gamma'(r) = \begin{cases} \gamma(r), & \text{if } r \neq rd; \\ Extend_{L,l}\big(F\_\text{op}^{R}_{l}(\tilde{v}_1, \ldots, \tilde{v}_n)\big), & \text{if } r = rd \\ & \text{and } Finst \in \mathcal{F}_{approx} \\ & \text{and } \texttt{enable\_approx} == 1, \\ F\_op^{R}_{L}(v_1, \ldots, v_n), & \text{otherwise.} \end{cases}$$

[3]Please note that in contrast to our definition, the IEEE 2008-754 Standard's [28] computational instructions also include comparisons. We follow the same definition as the RISC-V ISA [138]

[4]A state is defined by the content of the FPU registers $r_i \in \mathcal{A} \cup \mathcal{A}_{\text{ARP}}$.

where,

$\gamma(r), \gamma'(r)$		refer to the content of the register r before and after the transition.
l	$= \text{Select\_Prec}(L)$	the selected target reduced precision l corresponding to L
v_i	$= \gamma(rs_i) \in \mathbb{F}_L$	high-precision values.
$\tilde{v}_i$	$= Reduce^{R'}_{L,l}(\gamma(rs_i)) \in \mathbb{F}_l$	reduced precision values.

Precision selection: before starting the process, the appropriate reduced precision l to execute the operation should be selected. We distinguish two schemes:

- A simplified scheme would associate to each standard precision format L_i a unique reduced format l_i. In this case, the hardware architecture would contain as many reduced precision formats as standard precision formats.
- A more elaborate scheme would allow to have more reduced precision formats than standard ones. In this case, the target reduced format l_i that will approximate the operation is encoded in the register PREC_{L_i}. The latter being a typical CSR register that can be programmed at run-time, *i.e.*, in this case, the computation precision would be programmable at run-time.

Precision reduction: in this step, we perform a cast using a rounding mode R' equal to or different from R, depending on how it is implemented. In most ISAs, R is encoded either in the instruction binary code (fixed) itself or in an FPU control status register (dynamic). Hence, an implementation can either define a fixed reduction rounding mode R' or support dynamic rounding.

We chose to leave the choice to the implementation because rounding hardware logic is generally expensive in an FPU. In addition, double rounding, (*i.e.*, casting numbers to the reduced format plus rounding the final result) may drastically affect the QoR. When the original format L and the selected target reduced format l have similar exponent bit-widths, the $Reduce^{R'}_{L,l}$ function is simplified to a rounding operator instead of a complete conversion operator. Otherwise, a complete cast operator should be implemented, which may be expensive in terms of circuit area.

Computation in ARP: the assembly-level approximation is introduced by performing the computation using the ARP operator $F\_\text{op}^R_l$, which operates on ARP l-bit operands, with a bit-width shorter than L. This is why inputs should first be reduced from L to l before performing the computation and then extended back from l to L after the computation.

Extension: once the computation $F\_\text{op}^R_l$ has been performed, the result should be converted back to the original format using the $Extend_{L,l}$ function, which converts the result from l back to the L format. This operation is intended to guarantee consistency with the non-computational FP instructions $\mathcal{F}_{exact}$, which are not approximated. The final result is then stored in the destination register rd.

3.4.3 The case of iterative operators

Some FPUs implement iterative operators $F\_\mathrm{op}_\ell^R$, particularly for division and square root calculations [139] [101], to allow circuit area optimization. For such implementations, the reduction and extension stages are unnecessary since the precision of their computations can be set at run-time. This means that the reduction and extension blocks are only necessary for **Fused Multiplication-Addition** (FMA) operations from an implementation point of view. More details will be provided along with the hardware architecture in Chapter 5.

3.4.4 Selective Approximation (SA)

A programmer needs to apply approximations to specific parts of an application and exclude others. For instance, in a software application, the functions that compute error metrics should be executed exactly to avoid compromising the results with approximations. Hence, the developer should be able to tag such functions or instructions.

From an implementation point of view, the signal `enable_approx` was added to support SA. This signal can be inferred using three different schemes:

1. The first one is *static in space*, by defining **non-approximable memory regions** where all instructions are executed exactly and **approximable memory regions** where all instructions are executed approximately.

   The enable signal can then be inferred from the **Program Counter** (PC) address of the current executing FP assembly instruction $F\_inst_\ell^R$. If the PC address belongs to the non-approximable memory region, then it is set to zero; otherwise, it is set to one.

2. The second one is also *static in space*, by encoding the enable signal in the FP assembly instruction's binary code itself using non-reserved ISA opcodes. Again, however, this will necessitate support from the compiler.

3. The third scheme is *dynamic in time* by mapping the enable signal to another custom CSR that is added to the FPU and then assign it using CSR read-/write assembly instructions.

As shown in Section 2.3.3, hardware-level support for **Arbitrary Reduced Precision** (ARP) and **Variable Precision in Time** (VPT) is mandatory if one needs to go beyond impact simulation towards achieving real-world energy efficiency. This means that intrusiveness in hardware cannot be avoided. However, guaranteeing non-intrusiveness or at least minimal intrusiveness to source code without the need for a specialized compiler is possible in the first and the third schemes. The third adds one feature; the ability for applications to change their operating precision at run-time. A practical use case (iterative algorithms) will be demonstrated in Chapter 6.

3.4.5 Problem statement

The choice of the "approximable" instructions/memory regions and the choice of the intermediate reduced precisions' set $\mathcal{X}_{ARP}$ can be formulated as the following optimization problem:

> Given an application that takes as an input a dataset I, providing a numerical output result O executed on top of an FPU supporting a set of standard precisions $\mathcal{X}_{SP}$, our objective is to construct an optimal FPU configuration, *i.e.*, a set of supported reduced precisions $\mathcal{X}_{ARP}$ that minimizes power consumption, execution time, and overall energy consumption, subject to a **Quality of Result** (QoR) constraint on the output O, while being non- or minimally-intrusive to source code.

To solve this problem, we propose a two-step process:

1. Software design space exploration in simulation using the QEMU-based simulator. This will allow us to select a set of candidate FPU bit-width configurations ($\mathcal{X}_{ARP}$) satisfying the QoR constraint target.

   This is achieved through the software implementation and is detailed in Chapter 4.

2. Hardware-level design space exploration: for each candidate FPU configuration, an estimation of the overall execution time and energy savings is made to select the best final configurations among the $\mathcal{X}_{ARP}$ set.

   This is achieved through the hardware implementation and is detailed in Chapter 5.

This process will be presented in more detail in the remainder of this thesis.

3.5 Conclusion

This chapter introduced the proposed approach that leverages FP approximations for energy efficiency in error-tolerant applications. We presented a background on the **Floating-Point** model, the basic idea and finally the theoretical formalization of the proposed methodology in the form of a FSM. This model will constitute the specification of our future implementations either in SW (for simulation) or in HW (for execution).

In the next two Chapters (4 and 5) we will present two complementary implementations: 1) a software-level one that allows the simulation of FP approximations' impact on the QoR of applications, and 2) a hardware-level implementation in the form of a multi-precision FPU that allows cultivating important energy savings. Finally, in chapter 6, we will present a use case that shows how the proposed approach enables **Variable Precision in Time**.

4

AxQEMU: a Non-intrusive Floating-Point Approximation Simulator

Contents

4.1 Introduction

Floating-Point Units FPUs are ubiquitous in modern hardware architectures, including **General-Purpose Processors** GPPs and **Application-Specific Instruction-Set Processors** ASIPs, where they boost the performance of computationally intensive applications. Unfortunately, FPUs are responsible for a significant proportion of power consumption and memory bandwidth. Furthermore, the energy consumption associated with FP arithmetic is known to be higher than that of its integer counterpart [117] , making FPU optimization a priority.

One of the techniques used to optimize FPUs is bit-width reduction, where exponent and/or mantissa lengths are reduced to either standard bit-widths (defined in the IEEE 754 standard [28]), or custom arbitrary bit-widths. Over the years, many techniques/tools/libraries have been proposed to explore the impact of using arbitrary precision FP arithmetic [126–128,140] in computational kernels targeting many platforms (GPPs, GPUs, FPGAs, *etc.*).

In chapter 3 we presented a new methodology that leverages **Floating-Point** approximations for energy efficiency. The methodology consists on injecting approximations at the low level by changing the interpretation of FP assembly instructions at run-time during execution. This means that the approach is non-intrusive to the software application source code nor the binary executable.

In this chapter, we leverage this technique for rapid **Design Space Exploration** (DSE) of **Arbitrary Reduced Precision** FPUs in a CPU-based architecture context. The objective is to help designers select the most optimized FPU configuration (exponent and mantissa bit-widths) satisfying a **Quality of Result** (QoR) threshold set for the application and input dataset provided by the designer, without the need to transform / rewrite / modify the source code.

The approach led to the development of a software implementation which we called AxQEMU (Approximate QEMU). The simulator is built on top of the well-known functional simulator QEMU [141], which performs dynamic binary translation of assembly instructions.

AxQEMU can be used alone with software applications and it can also be used in conjunction with other optimization tools from the SoA.

First, Section 4.2 will introduce our main target ISA RISC-V and more specifically the aspects related to FP. Then, in 4.2.2, we will present the (vanilla) QEMU [141] binary translator and its internal architecture. Moreover, in Section 4.3 we will present how approximations have been integrated inside QEMU to produce AxQEMU as well a brief presentation of its capabilities. Finally, in Section 4.4 we present a use case illustrating how AxQEMU can be used for DSE.

The next Chapter will present the hardware-level aspects of the proposed methodology and a more complex use case that demonstrates how AxQEMU can be used in conjunction with other SoA tools for fine-grained **Variable Type Optimization** (VTO).

4.2 Background

To implement a proof of concept of the approach proposed in the previous chapter, a functional ISA simulator (*e.g.*, gem5, or1k, Spike, *etc.*) can be used. Here, we targeted the free and open-source QEMU multi-ISA dynamic binary translator [141] .

4.2.1 The RISC-V Instruction Set Architecture

In this section, we briefly present some key aspects about the RISC-V ISA. An extended description with more details can be found in Appendix B.

RISC-V (pronounced "risk-five") is an ISA that was originally designed to support computer architecture research and education. It is now also becoming a standard free and open architecture for industry implementations [138, 142, 143].

We chose RISC-V in our implementations for many reasons, of which we cite the following:

- A completely *open* ISA that is freely available to academia and industry.
- A *real* ISA suitable for direct native hardware implementation, not just simulation or binary translation.
- Support for the revised 2008 IEEE-754 floating-point standard [28].
- Well supported and actively maintained HW/SW tools thanks to the open-source HW/SW community.
- Availability of many open-source SoC and processors' implementations such as Rocket Chip [144], Pulpino [145], Pulpissimo [146], Ariane / CVA6 [115, 147] , Ibex [148], lowRISC Chip [149], and many others.

Although all the ideas explained in this manuscript, such as **Arbitrary Reduced Precision** and **Variable Precision in Time**, were only implemented on RISC-V for pragmatic and practical reasons, they are also applicable to other architectures.

4.2.1.1 Target ISA

An ISA defines which instructions and registers are available in hardware. It is a sort of contract that bonds HW to SW and defines which/how assembly instructions are generated from high-level source code, which registers can be used, how special cases or interrupts are handled, *etc.*

RISC-V is a modular ISA that enables designers to support one or many features depending on the target application. Since we target **General-Purpose Processors**, we will use by default the architecture `RV64IMAFDC` in all our hardware and software experiments unless otherwise stated. This architecture is also referred to as `RV64GC`, where the "G" stands for general-purpose. [138]

When only the FPU of the architecture is concerned, the name `RV64FD` will refer to the standard baseline FPU architecture with support for the standard FP precisions `binary32` and `binary64`.

4.2.1.2 Target Application Binary Interface (ABI)

An ABI is an interface that defines how different software modules communicate with each other. It can be assimilated to a protocol that defines how arguments are passed to functions, where functions' return values are stored, and how memory is organized.

The ISA architecture and the ABI should both be specified to the RISC-V GCC cross-compiler using the following arguments, respectively [150]:

- `-march=<ISA>` selects the architecture to target. This controls which instructions and registers are available for the compiler usage.
- `-mabi=<ABI>` selects the ABI to target. This controls the calling convention (which arguments are passed in which registers) and data layout in memory.

For all our applications and experiments, we only deal with `float` and `double` variables. So we use the `lp64d` ABI along with the `RV64IMAFDC` ISA.

4.2.1.3 Floating-Point in RISC-V

In this section, we will present the ISA aspects related to FP arithmetic in RISC-V. Namely, we will briefly present which registers and instructions are added to the base ISA when the FP extensions are handled.

FP extensions Until now, RISC-V supports three FP extensions **F, D**, and **Q** as depicted in table B.1. Since RISC-V is designed with extensibility and scalability in mind, other non-standard extensions in the **State of the Art** also support **H**, an extension dedicated for 16-bit half-precision FP arithmetic [100, 101, 114].

FP General-Purpose Registers When a **Floating-Point** extension is supported, a total of 32 GPRs are added ($\texttt{f}_0, \texttt{f}_1, \ldots, \texttt{f}_{31}$). Their bit-width, denoted `FLEN`, is the widest FP format bit-width supported in the system.

For instance, the typical RV64IMAFD architecture supporting 32-bit single-precision and 64-bit double-precision contains 32 base integer registers $x_0, x_1, \ldots, x_{31}$ which are 64-bit wide (since XLEN equals 64, *C.f.*, Section B.2.2). It also contains an additional 32 registers $f_0, f_1, \ldots, f_{31}$ which are 64-bit wide, since $FLEN = max(32, 64) = 64$.

In addition to the GPRs, a CSR named fcsr is also added by FP extensions. Table B.3 depicts its layout.

FP formats To indicate the FP format of each assembly instruction, a field called fmt is embedded in the binary code of the instruction. This field is 2-bit wide, meaning that RISC-V is technically able to support up to four FP formats. These formats are listed in table 4.1 [138, Section 11.6].

FP format	Full name	Format mnemonic	*fmt* field	Corresponding extensions
binary32	32-bit single-precision	S	00	**F**, **D**, **Q**
binary64	64-bit double-precision	D	01	**D**, **Q**
binary16	16-bit half-precision	H	10	–[1]
binary128	128-bit quad-precision	Q	11	**Q**

[1] Non-standard extension in the SoA [100, 101, 114].

Table 4.1: Format field encoding.

FP instructions There are five types of instructions that are added when supporting an FP extension:

- Load instructions: FLW, FLD, FLQ
- Store instructions: FSW, FSD, FSQ
- Computational instructions: FADD.x, FSUB.x, FMUL.x, FDIV.x, FMADD.x, FMSUB.x, FNMADD.x, FNMSUB.x, FSQRT.x, FMIN-MAX.x where x ∈ {S, D, Q}.

  In our case, all computational instructions are approximable except FMIN-MAX since they are the most energy consuming operators.
- Conversion and Move instructions: FCVT.int.fmt, FCVT.fmt.int, FCVT.S.D, FCVT.D.S, FMV.X.W, FMV.W.X ...*etc.*

  Instructions that move or convert numbers from a source FP format to another, or from FP to integer formats and vice versa.
- Comparison instructions: FCMP.x where x ∈ {S, D, Q}.
- Classify instructions: examine the value of a FP register and return a 10-bit mask defining the type of the number, *e.g.*, normal/denormal, finite/infinite/NaN, positive/negative, zero/non-zero.

All these instructions and their specifications are described in detail in the official RISC-V user-level specification [138].

FP emulation when an FPU is absent Even when an ISA does not support hardware-level IEEE 2008-754 **Floating-Point** arithmetic, programmers can still use data types such as `float`, `double`, and `long double` and FP computations. However, at that moment, it is necessary to include an FP emulation library that emulates FP operations in software using integer arithmetic. In the case of RISC-V, emulation is performed using SoftFloat [151], a library written by J. Hauser, one of the co-authors of the RISC-V privileged specification [142].

For example, consider the `RV64IMAFD` architecture. The architecture name means that an FPU is available in the system with support for both `binary32` and `binary64` standard FP formats. Meaning that software instructions involving operations on `float` and `double` variables will be performed in hardware. However, `long double` variables will be associated with the `binary128` IEEE standard format. Since the `RV64IMAFD` does not support this type in hardware, operations on such variables will be emulated in software based on the SoftFloat library shipped with the standard C library and the GCC Toolchain.

4.2.1.4 Performance Vs. hardware-level overhead

Computer architects should decide which extensions to support depending on the intended application domain. This choice brings significant performance boosts, but it affects both the software toolchain and the processors' hardware. For example, if the **M** extension is supported in a given RISC-V processor, it means that its underlying micro-architecture contains a hardware integer multiplier and divider. Otherwise, multiplication operations will be emulated using the base integer **I** instructions. This is technically feasible since multiplications can be emulated with a series of additions, although they are inefficient in terms of performance (execution time and throughput). On the other hand, supporting extensions means adding hardware components to the processor (ALUs, FUs, decoder logic, *etc.*), leading to more circuit area. Hence, computer designers should find trade-offs that are adequate depending on the target application.

4.2.2 The QEMU Binary Translator

QEMU is a fast and portable emulator, developed by the French programmer Fabrice Bellard back in 2005. When a software application is compiled for a given architecture, the resulting binary executable cannot be executed on other architectures. For example, RISC-V binary code cannot run on ARM processors. QEMU solves this problem by emulating some ISAs on top of others, using the concept of **Dynamic Binary Translation** (DBT) [141, 152]. DBT in particular, and more generally QEMU, are useful for the following use cases, among others:

- Enabling binary executable portability

  For instance, this is useful when the source code of the application is not available, cross-compilation is not possible, or a lack of software libraries or low-level drivers in the target architecture.

- Enabling early virtual prototyping and hardware/software codesign

This enables computer/platform/SoC architects, hardware designers, toolchain engineers, and application engineers to rapidly co-develop new processor architectures in parallel without waiting for the actual processor to be manufactured.

- Enabling **Design Space Exploration** (DSE) of new ideas

  Such tools can help computer architects make architecture design decisions based on performance estimations. For example, in our case, the development of the final hardware architecture of the multi-precision FPU (Chapter 5) was guided by feedback from the AxQEMU software virtual prototype.

One of the most recent industrial contexts where DBT showed its usefulness was the transition of Apple from Intel x86-64-based processors to their own ARM-based Apple silicon in 2020. To speed up the transition, Apple announced Rosetta 2, a tool that permits many applications compiled exclusively for execution on x86-64-based processors to be translated for execution on Apple silicon while waiting to port their native source code [153].

However, there is a price to pay in terms of performance when using such tools instead of native execution. Typically, execution with QEMU is about 5 to 20× slower than native code execution [152]. However, given its benefits, this execution time overhead is still acceptable compared to other SoA simulators.

4.2.2.1 Overview of QEMU's architecture

In QEMU, we distinguish two kinds of machines:

1. the guest (or target) machine: the processor emulated, for which an application has been compiled, and

2. the host machine: the processor executing QEMU itself, which simulates the execution of the target code, even if the two processors have different architectures

Figure 4.1 depicts QEMU's operation principle [152]. Efficient emulation is achieved thanks to the **Tiny Code Generator** (TCG), a sort of run-time compiler embedded in QEMU. This generator dynamically translates blocks of target instructions known as **Translation Blocks** (TBs) to TCG operations (TCGops) or micro-operations (micro-ops), which constitute the machine-independent **Intermediate Representation** (IR). Subsequently, the TCGops will be translated into host instructions.

A TB is a set of consecutive elementary (RISC-V) assembly instructions from the guest binary executable. Any jump/call assembly instruction or system call marks the end of a TB.

When a TB is translated into its corresponding host code, the translated block is cached for later use in a Translation Cache. Thus, for example, in the case of a loop, the TB is translated only once but executed multiple times. This caching capability is one of several optimization procedures that increase the performance of this simulator when compared to others. When the next TB is fetched, QEMU first checks if its first PC address has already been seen (left part, Fig. 4.1). If

this is the case, it means that its corresponding translation already exists in the Translation Cache. Hence it is fetched and executed. Otherwise, the TB should go through the translation process and the caching process.

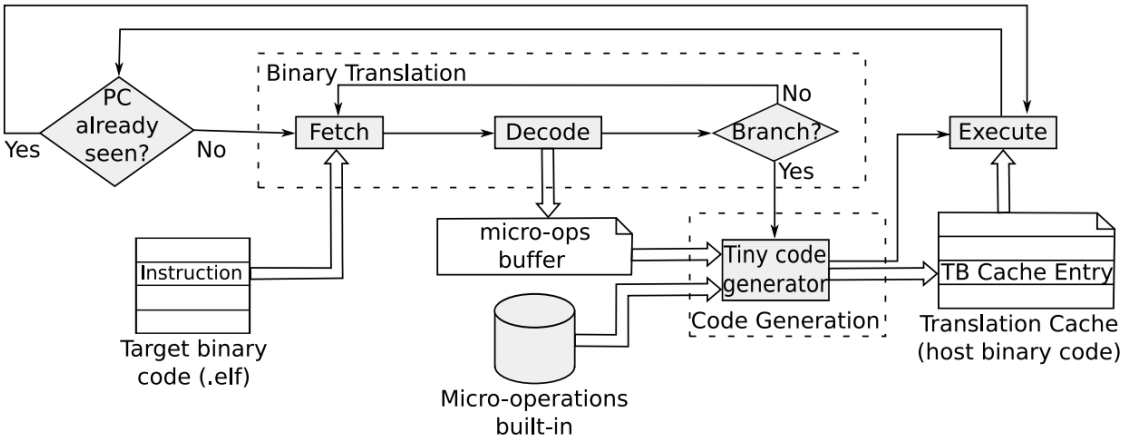


Figure 4.1: The architecture of the QEMU binary translator [152].

4.2.2.2 QEMU's Intermediate Representation

In QEMU, DBT is enabled by introducing a unified machine-independent IR layer (Fig. 4.2). This IR is an intermediate form of binary code that is used as an interface between the guest machine architecture and the host machine architecture:

```
...
fld     fa5, 8(sp)
fld     fa4, 16(sp)
fadd.d dyn, fa5, fa5, fa4
fsd     fa5, 24(sp)
ld      a1, 24(sp)
jal     ra, 1232
...
```

```
...
mov_i64 tmp2, sp
movi_i64 tmp3, $0x10
add_i64 tmp2, tmp2, tmp3
qemu_ld_i64 fa4, tmp2
movi_i32 tmp0, $0x7
call set_rounding_mode, $0x2, $0, env, tmp0
call fadd_d, $0x1, $1, fa5, env, fa5, fa4
...
```

Table 4.2: Example: code translation from RISC-V assembly instructions to QEMU IR

Table 4.2 provides an example of code translation of the RISC-V binary assembly instructions `fld` and `fadd.d` (left) to the QEMU Intermediate Representation (right).

4.2.2.3 QEMU's Dynamic Binary Translation process

Figure 4.3 depicts the DBT process in more detail:

Step ① For each newly fetched assembly instruction, the TCG find the corresponding TB. If the TB has already been translated, it is retrieved from the translation cache and is executed in Step ⑥. Otherwise, the TB gets translated in Step ②.

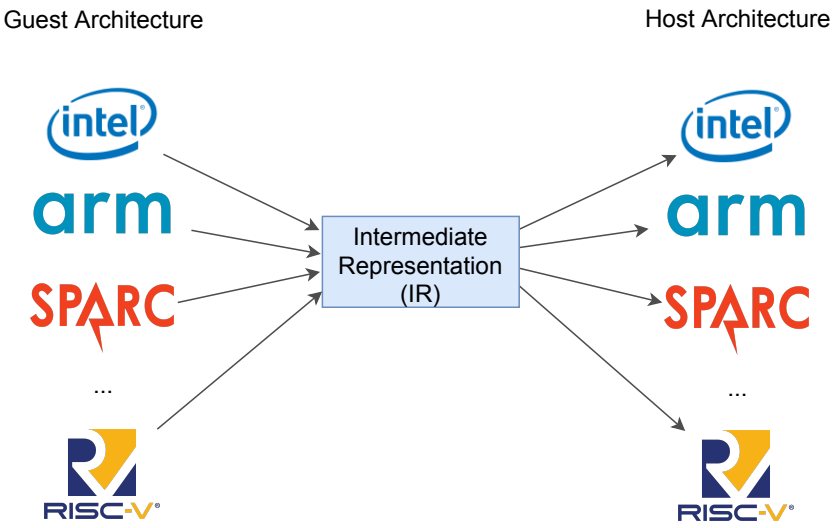


Figure 4.2: The QEMU **Intermediate Representation**.

Step ② The assembly instruction is converted to QEMU's IR. In this step, a mapping between the guest machine ISA and the IR is performed. It associates to each assembly instruction one or more TCG micro-ops. Each translated block is prefixed with a prologue and suffixed with an epilogue IR code.

For example, in Table 4.2, the RISC-V assembly instruction `fadd.d` which performs addition of two FP numbers is translated into three TCG-ops: a) first `movi_i32` which performs a 32-bit move immediate instruction, then b) a call to a C helper function named `set_rounding_mode()` which configures the rounding mode of the following FP operation. And finally, c) a call to another C helper function named `fadd_d` that will perform the FP addition operation.

Step ③ The IR of the translated block is then translated to host machine binary code. Thus, each TCG-op can be translated into either a single (or a set of) host assembly instructions or a C helper function that will eventually be compiled to produce several host instructions.

Considering the same example, the `movi_i32` TCG-op can be translated into a single x86-64 assembly instruction, whereas the two other function calls will eventually be compiled into many assembly instructions.

Step ④ The resulting host binary code from the previous step is then cached in a translation cache for future use leading to significant performance gains in QEMU.

Step ⑤ After the execution of each translated block, QEMU uses the PC and information related to the CPU state to find the next TB. Since this process is costly, a direct jump that points to the next TB is appended to the TB. This leads to a zero run-time overhead during transitions from one TB to the next. This process is called Direct Block Chaining [141].

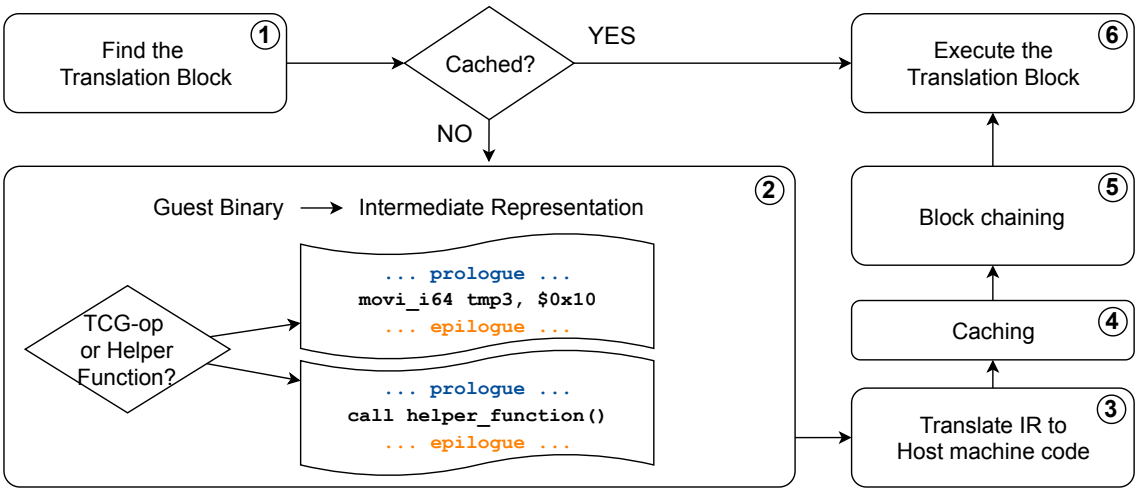


Figure 4.3: QEMU's **Dynamic Binary Translation** *process*

4.3 AxQEMU: A Floating-Point Approximation-aware Emulator

In this section we will present some implementation details regarding the AxQEMU emulator. The tool has been open-sourced and is accessible at the link below.

`https://github.com/noureddine-as/axqemu`

The most up-to-date branch is `v5.0-variable-prec-in-time`. More technical details about the modifications made to the original QEMU project are presented in Appendix C.1. In total, 26 files have been modified, with more than 3846 insertions and 114 deletions[2].

4.3.1 Approximate Floating-Point operations in AxQEMU

The behavior of **Floating-Point** instructions is originally emulated in software using the SoftFloat [151] library mentioned earlier inside QEMU. It is a reference C library used in many industrial systems to emulate FP operations. Please note that we are talking about the implementation of FP arithmetic inside QEMU without regard to the RISC-V application itself.

In other words, the RISC-V software application itself is compiled for the `RV64IMAFDC` ISA and the `lp64d` ABI. This means that all FP C/C++ statements will generate instructions from the **F** and **D** extensions, *e.g.*, `FADD.x, FSUB.x, FMUL.x ...`, where `x` $\in$ `{S, D}`. If the application is run on top of a hardware CPU, these instructions will be executed inside the FPU. However, when these instructions are emulated with QEMU and AxQEMU, their behavior is simulated in software, using the SoftFloat library through calls from the C helper functions `helper_fadd_x()`.

TCG intermediate operations can be translated into either a single host instruction or a C helper function (Fig. 4.4) that will eventually be compiled to produce

[2]A full list of modifications (diff file) is available in this link.

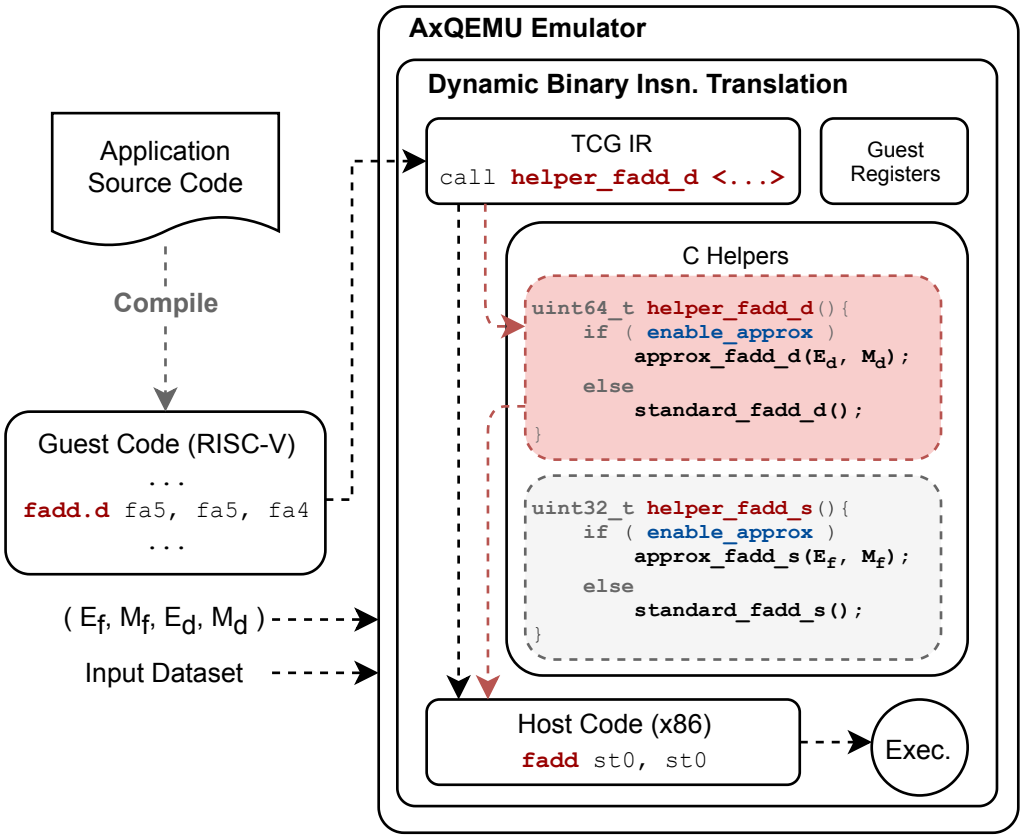


Figure 4.4: AxQEMU overview

several host instructions. Furthermore, when translating FP instructions, the corresponding TCGops can either be executed directly on the host hardware's FPU or emulated using C helper functions exploiting a backend FP software emulation library [126–128, 151]. In other words, each guest FP computational assembly instruction is associated with a custom C function which simulates the reduced precision behavior in the back-end.

Figure 4.4 shows an application source code that has been compiled for a RISC-V target machine. Each assembly instruction from the executable binary will be fed to the TCG to generate its corresponding intermediate representation (TCGop). For example the RISC-V FP addition instruction (`fadd.d fa5, fa5, fa4`), which should perform the operation `fa5 = fa4 + fa5`, is mapped to the C function `helper_fadd_d()`[3], which operates on the values contained in emulated guest registers `fa5` and `fa4`. In AxQEMU the helper functions were modified to implement the behavior explained in Section 3.4.

4.3.2 Selective Approximation

In Section 3.4.4 we have proposed two schemes to support SA. The first one consists on defining two or many memory regions with different approximation modes at compile time. The second enables support for dynamic variable preci-

[3]The "**d**" in `helper_fadd_d` refers to double-precision, whereas the "**s**" in `helper_fadd_s` refers to single-precision.

sion at run-time and allows the precision to change at run-time within the same memory region or even the same function. In this section, we discuss the implementation aspects of both schemes.

4.3.2.1 Static SA based on memory partitioning

This scheme allows developers to specify functions that should be executed precisely to facilitate integration with existing flows. In our case, SA is supported by splitting the (instruction) memory address space across two regions:

1. **Non-Approximable address space**: a memory region where approximations are deactivated (*i.e.*, `enable_approx = 0`), and
2. **Approximable address space**: where approximations are enabled using ARP (*i.e.*, `enable_approx = 1`).

To implement this split, the non-approximable functions requiring precise execution are annotated using a C macro. For instance, Listing 2 depicts a function `compute_QoR`, destined to compute some QoR metric using a numerical result `result` and a reference value `ref`. It has been annotated with the `PRECISE` macro defined in lines [1-2]. This macro tells the linker to place the annotated function in a specific memory section, the "`.precise`" section. Functions that belong to this memory region will be executed precisely, whereas functions stored in other regions, such as `compute_fft()` in the example below, are by default executed approximately.

```
1  #define PRECISE __attribute__((__section__(".precise"))) \
2                  __attribute__((noinline))
3
4  double PRECISE compute_QoR(float result, double ref){
5          // Function body here
6  }
7
8  void compute_fft(float input[], float coefficients[], double output[]){
9          // Function body here
10 }
```

Listing 1: Example of a C macro for function annotation

At run-time, the address of each assembly instruction instance is fetched from the Program Counter register (PC). If an assembly function belongs to the non-approximable address space (*i.e.*, the `.precise` memory section) then `enable_approx` is set to zero, and the instruction is emulated using the standard SoftFloat [151] FP emulation library. Otherwise, `enable_approx` is asserted, and the instruction is emulated using an ARP library [126–128], taking the FPU configuration specified by the user into account.

In the previous example and the remaining of this chapter, we only consider two execution modes: precise and approximate. It is also possible to develop other scenarios where we have many memory regions: `PRECISE`, `APPROX_1`, `APPROX_2` ...*etc.* Where each region has a different FP precision.

As for all similar techniques, the source code must be modified and re-compiled when using SA. However, when using our approach, the implementation is much simpler and light-weight than other current techniques and minimally-intrusive.

This scheme can also be extended to data memory, where approximable data regions are mapped to approximate/unreliable cost-effective approximate memories or storage devices (*C.f.*, Section 2.2.6.3). However, this aspect is out of the scope of this thesis.

4.3.2.2 Dynamic SA at run-time based CSR operations

This scheme allows developers to modify operating precision at run-time. This is achieved by augmenting the FPU with a set of CSRs that control computations' precision. These CSRs can be modified using the usual CSR Read / Write assembly instructions provided by RISC-V. In addition, this scheme allows more fine-grained precision tuning by overriding the default operating precision defined within the first scheme.

This scheme will be further explained later in Chapter 5 when we introduce the multi-precision VPT-FPU. In the remaining parts of this chapter, we will only use the first scheme to define approximable and precise instruction memory regions. Moreover, for simplicity, all instructions belonging to the approximable region will be executed using the same fixed reduced precision.

4.3.3 Key engineering decisions

For implementation, since SoftFloat (the library used within QEMU to simulate FP behavior) only supports standard precisions, we replaced it with the FlexFloat [128] library. This library was chosen for approximation emulation, since it demonstrates an increase in speed of up to 2.8x compared to other mixed-precision libraries according to [128]. The library provides functions for casting from full-precision types to reduced-precision (reduction stage) types and vice versa (extension stage) and functions to perform arithmetic operations in reduced precision.

The reduction rounding mode R' is dynamic and is the same as the rounding mode R of the approximated instruction. The RISC-V ISA [138] was chosen for its design simplicity. We implemented the approach for all the computational instructions of single-precision (**F**) and double-precision (**D**) extensions of the ISA except the `fmin-max.x` instruction, hence a total of 9 instructions per extension.

The C helpers were modified to accept two additional arguments: the exponent and the mantissa bit-widths, which define the precision of the internal computations. These arguments have been exposed so that the user can define them at launch time. This facilitates the use of the simulator when dealing with search algorithms to explore several FPU configurations.

4.3.4 AxQEMU usage example

Listing 4.1 depicts an excerpt of AxQEMU's arguments along with their description. For the full listing, please refer to Appendix C.1. The arguments `expbitsf`, `fracbitsf`, `expbitsd`, `fracbitsd` refer to E_f, M_f, E_d, M_d, respectively.

Listing 4.1: Excerpt of AxQEMU's arguments

```
$ qemu-riscv64 --help
usage: qemu-riscv64 [options] program [arguments...]
Linux CPU emulator (compiled for riscv64 emulation)

Options and associated environment variables:

Argument                          Description
...
-expbitsd <EXP_BITS_d>             The FPU exponent bit-width
                                   for the D extension. Default is 11
-fracbitsd <FRAC_BITS_d>           The FPU fraction bit-width
                                   for the D extension. Default is 52
-expbitsf <EXP_BITS_f>             The FPU exponent bit-width
                                   for the F extension. Default is 8
-fracbitsf <FRAC_BITS_f>           The FPU fraction bit-width
                                   for the F extension. Default is 23
-non_approx_region_start <@ADDR>   The Start address of a non-approximable
                                   (.precise) region.
-non_approx_region_size <SIZE>     The Size (in Bytes) of a non-approximable
                                   (.precise) region.
```

Ax(QEMU) supports two execution modes: user emulation mode and full system emulation mode. On the one hand, the application should be cross-compiled for a RISC-V Linux-based execution environment in the user emulation mode. It is launched as a Linux process with full access to the host machine's file system. On the other hand, the application is cross-compiled in full system emulation mode for a bare-metal-like execution environment with no OS. In the latter case, the application is emulated on top of a virtual device that simulates a real-world RISC-V platform[4], including some peripherals, network interfaces, and others.

Usage example Listing 4.2 shows an example where a C application source code `application.c` is cross-compiled using the `riscv64-linux-gcc` GCC compiler and run on top of AxQEMU in User Emulation mode. In this example, `float` operations are approximated on (6, 10) format, whereas `double` operations are approximated on (9, 40) format.

Listing 4.2: Compiling an application and emulation in User Emulation mode.

```
$ # Cross-Compilation using RISC-V Linux GCC (Glibc Standard Library)
$ riscv64-linux-gcc -march=rv64imafd -mabi=lp64d -static application.c \
                    -o application.elf
$ # Simulation using AxQEMU
$ qemu-riscv64 --expbitsd 9 --fracbitsd 40 \
               --expbitsf 6 --fracbitsf 10 \
               -non_approx_region_start 0x1d238 \
               -non_approx_region_size 0x428 \
```

[4]Supported RISC-V board models are available here: https://qemu.readthedocs.io/en/latest/system/target-riscv.html

```
application.elf
```

Listing C.2 (Appendix C) presents the same example but in full system emulation mode.

4.3.5 Supporting other ISAs in AxQEMU

For now, the variable precision in the AxQEMU tool only supports the RISC-V ISA. However, the methodology can also be ported to other ISAs. To support static SA in other **Instruction Set Architectures**, the following steps should be performed :

1. Locate the **Floating-Point** arithmetic C helpers corresponding to the new target architecture:

   These C helper functions map FP assembly instructions to functions from SoftFloat.

2. Replace these calls with the logic described in Section 4.3.1:

   Instead of directly executing the FP operation in standard precision, check if the PC address of the current assembly instruction belongs to the approximate region. If this is the case, set `enable_approx` to 1 and execute the operation using the corresponding function from FlexFloat instead of SoftFloat. The function from FlexFloat should be called with arguments specifying the chosen reduced format. Once executed, extend back the result to the original format and return the result.

   Otherwise, if the instruction belongs to the non-approximable address space, then execute the original function provided by the SoftFloat library.

Extending the support to dynamic SA in other **Instruction Set Architectures** requires the following steps in addition to the previous ones:

1. Locate the FPU registers in the new architecture's CPU model inside QEMU.
2. Extend the FPU with two CSR registers, one that will hold the precision of `float` operations, and the other holds the precision of `double` operations.
3. Map these CSR registers to non-used addresses.

   At run-time, the content of these registers can be updated using CSR Read-/Write instructions, hence overriding the current precision.
4. Override precision selection inside the C helper functions by fetching it from the new CSRs registers.

4.4 Use Case 1: Direct Application to Fixed-Precision Applications

In Chapter 3 we presented a methodology that leverages FP approximations for energy efficiency of approximate applications that tolerate some loss in their

QoR. Then, in the previous sections of this chapter, we presented in detail a software implementation of this methodology in the form of a simulation tool called AxQEMU that enables designers to explore the impact of FP approximations on applications' QoR.

This section presents a first use case where AxQEMU is directly applied to a set of benchmark applications from state-of-the-art.

4.4.1 Design Space Exploration flow

This section presents the DSE flow used to evaluate the impact of FP precision on the output QoR of applications.

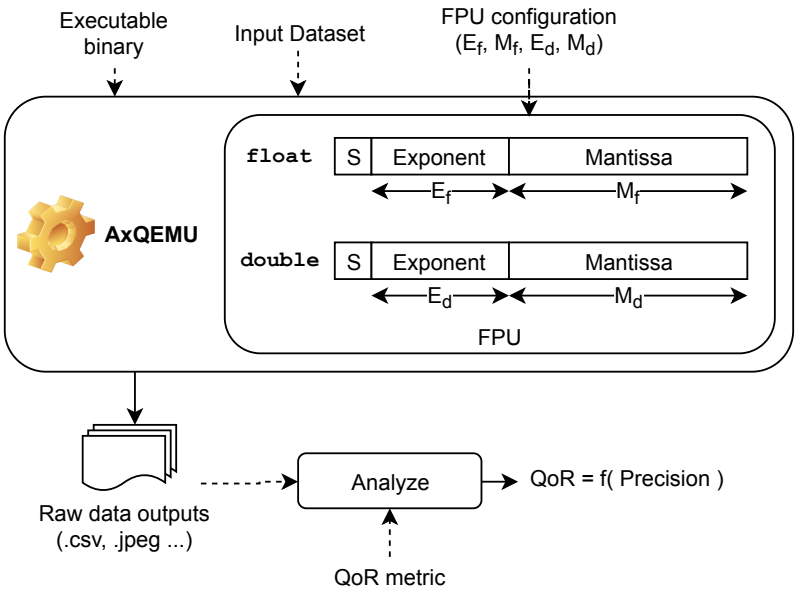


Figure 4.5: **Design Space Exploration** *methodology*

Let A be an application, and D its input dataset. Let us denote $C = \{c_0, c_1, ...\}$ the set of candidate hardware configurations (E_f, M_f, E_d, M_d) to be studied. Our objective is to find the most optimized FPU bit-width configuration $c_i \in C$ that satisfies a target QoR specified by the user for the couple (A, D). The DSE process is depicted in Figure 4.5. It includes two steps :

§1 Simulation phase the application A is executed for each configuration $c_i \in C$, using the QEMU-based variable precision simulator we developed. The raw output results are stored for further analysis in the next phase. This phase can be more or less time-consuming depending on the size of the dataset, the execution time of the application itself, and the number of FPU configurations to be studied.

§2 QoR analysis phase the user chooses a reference configuration c_{ref} according to A and D (typically the highest bit-width available in C). Its results will serve as the application output golden reference. The user can define a custom

error evaluation metric[5] or use standard ones, *e.g.*, **Mean Squared Error** (MSE), **Mean Relative Error** (MRE), **Structural Similarity Index Measure** (SSIM), *etc.* An analysis tool iterates over the numeric results issued for each FPU configuration. The final result of this process (Figures 4.6, 4.7, 4.8, 4.9) is a heat-map representing the computed error values for each application A, using its associated metric for each FPU configuration w.r.t the reference configuration c_{ref}.

The applications presented in this section are all written using one precision (fixed-precision or unmixed-precision) to simplify the analysis and demonstrate the capability of the tool, *i.e.*, all variables and FP operations are performed in the double-precision (`double`) format. Meaning that for these applications, we are targeting the optimization of the 64-bit data path (11, 52) in particular. The 32-bit data path (8, 23) is fixed during these experiments. More complex mixed-precision applications (containing both `float` and `double` variables) will be presented in the next chapter with an emphasis on the hardware-level aspects.

4.4.2 Evaluation benchmarks and QoR metrics

Application (A)	**Input dataset** (D)	**Error Metric**
Black-Scholes [154]	100,000 options (FP numbers)	MRE
FBench [155]	4x4 FP numbers matrix	RMSE
FFT [154]	2048 integer numbers	RMSE
Jmeint [154]	100,000 pairs of FP 3D triangle coordinates	Miss rate

Table 4.3: The list of studied applications and their corresponding error metrics

We applied the proposed DSE methodology to several applications from the literature [154, 155]. Table 4.3 depicts the evaluation benchmarks selected based on their extensive use of FP arithmetic, along with their corresponding input datasets and error metrics.

For each couple (A, D) and each $c_i \in C$, a set of N raw data output results $R = \{r_0, r_1, \ldots, r_{N-1}\}$ is obtained after execution. The MRE, RMSE, and Miss Rate metrics are computed for each configuration c_i on the output results using the following definitions [34, 154].

$$\begin{aligned} \text{MRE}(c_i) &= \frac{1}{N}\sum_{r_k \in R} \frac{\left|r_k(c_{ref}) - r_k(c_i)\right|}{max\left(1, r_k(c_{ref})\right)} \\ \text{RMSE}(c_i) &= \sqrt{\frac{1}{N}\sum_{r_k \in R}\left(r_k(c_{ref}) - r_k(c_i)\right)^2} \\ \text{Miss Rate}(c_i) &= \frac{1}{N}\sum_{r_k \in R} OnesCount\left(r_k(c_{ref}) \oplus r_k(c_i)\right) \end{aligned}$$

[5]The terms "**Quality of Result**" and "error metric" are equivalent and can be are used here interchangeably.

Where $\oplus$ represents binary XOR. $OnesCount()$ counts the number of elements that are equal to one *i.e.*, how many elements were misclassified with respect to the reference configuration c_{ref}.

4.4.3 Target architecture

For this case study, our target ISA is RISC-V [138], with the standard `RV64IMAFDC` instruction extensions *i.e.* 64-bit, I (integer), M (mul/div), A (atomic), F (single-precision FP), D (double-precision FP), and C (compressed) standard instruction extensions. To simplify the study, all the variables are initially expressed in double precision (but the tool also supports single-precision), and all applications have been compiled statically with the standard RISC-V GCC cross-compiler, using the (`-O3`) optimization level and the `lp64d` ABI. The Newlib standard C library implementation is used by default for these experiments.

4.4.4 Results and Discussion

Instruction types	Black-Scholes	FBench	FFT	Jmeint
FP arith. insns.	12.74%	15.68%	13.67%	10.71%
FP mem. insns.	25.74%	26.20%	22.66%	21.37%
Other FP insns.	8.34%	10.80%	7.72%	8.77%
Integer mem. insns.	13.35%	10.38%	11.67%	19.73%
Integer arith. insns.	9.50%	5.45%	11.83%	9.94%
Others insns.	30.33%	31.50%	32.45%	29.49

Table 4.4: Dynamic instruction execution breakdown per benchmark

§1 Dynamic instructions' breakdown To measure FPU usage accurately, we augmented the simulator with the ability to report an accurate dynamic instruction execution histogram. Table 4.4 shows the breakdown of all executed instruction instances for each application and its dataset, with the reference FPU configuration (8, 23, 11, 52).

Table 4.4 also shows that among all executed instructions, the FP-related operations count as a fraction equal to 46.82%, 52.60%, 44.05%, and 40.85% for Black-Scholes, FBench, FFT, and Jmeint, respectively. The majority of these FP-related operations are memory operations (loads and stores). This is due to the large input datasets that are loaded at startup. The second most significant part is arithmetic FP instructions. The remaining FP instructions include FP-to-integer/integer-to-FP conversions, comparisons, sign injections, *etc.*

§2 Simulation time A complete, exhaustive simulation has been performed on an x86-64 Intel(R) Xeon(R) E-2176M CPU machine with 16 GB RAM running a

Simulation time	Black-Scholes	FBench	FFT	Jmeint
Starndard Precision	4m22.2s	2.6s	24.0s	0m21.3s
Arbitrary Reduced Precision	8m23.9s	31.6s	26.9s	3m42.0s

Table 4.5: Total simulation time per benchmark.

Linux operating system. Table 4.5 depicts the total simulation time of all configurations when performed in **Arbitrary Reduced Precision** and in standard full precision (SP). The overhead added by the variable precision simulation is non-negligible.

Conclusion: *Thanks to the inherent QEMU optimizations (e.g. instruction simplification, TB caching, and chaining), the simulation time overhead is still affordable. More computationally intensive benchmarks can also be coupled with search algorithms to explore specific configurations and reduce this simulation overhead.*

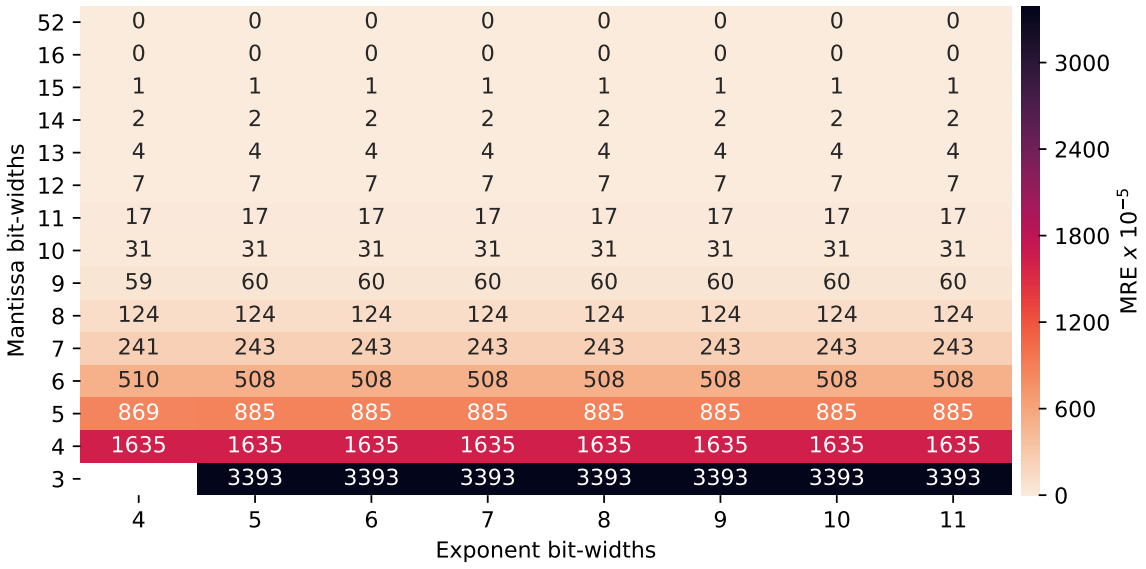


Figure 4.6: Resulting QoR corresponding to each FPU configuration for Black-Scholes (MRE)

§3 QoR results Since we only consider double-precision formats in this experiment, the optimization only concerns E_d (the exponent bit-width) and M_d (the mantissa bit-width) of the double-precision datapath. The simulations cover all arbitrary FPU configurations ($4 \leq E_d \leq 11$ and $3 \leq M_d \leq 52$), corresponding to a total of 400 simulations for each (A, D) couple. The QoR has been computed for all the studied configurations, and the results are summarized as heatmaps in Figures 4.6, 4.7, 4.8, and 4.9). The reference configuration used is (8,23, 11, 52).

Figure 4.6 and 4.8 show that QoR varies with mantissa bit-width and does not depend on the exponent bit-width. Meaning that a 4-bit (*resp.* 6-bit) exponent

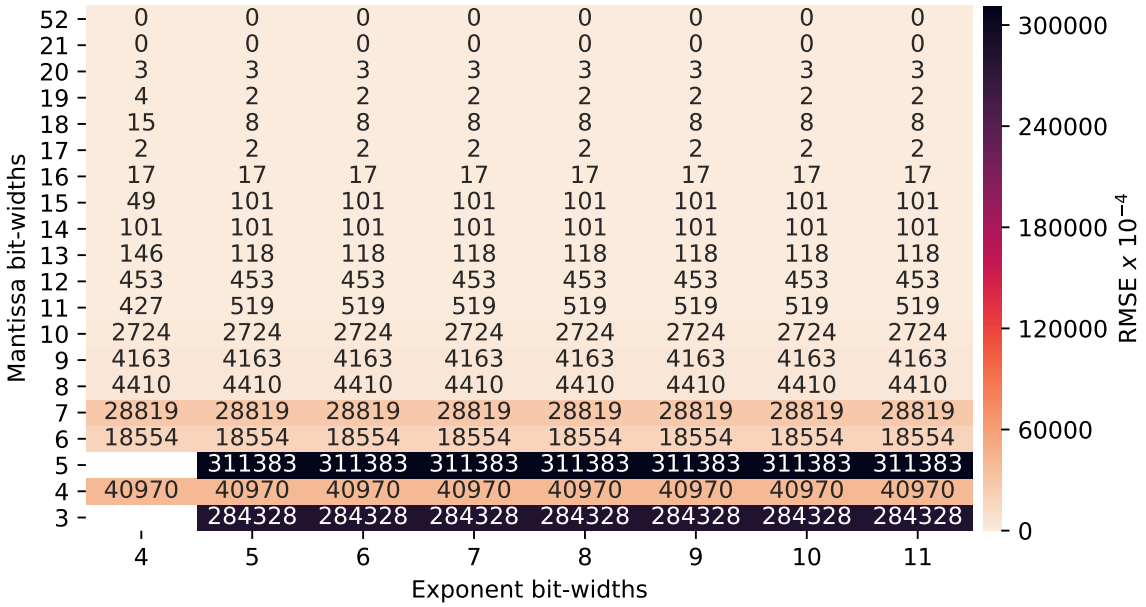


Figure 4.7: Resulting QoR corresponding to each FPU configuration for FBench (RMSE)

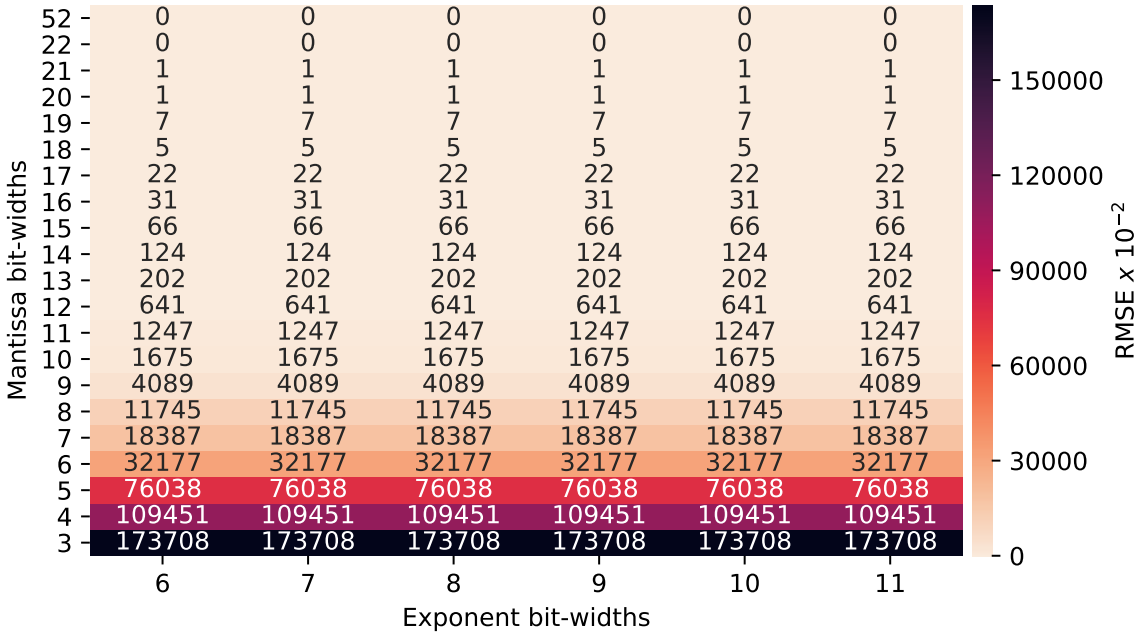


Figure 4.8: Resulting QoR corresponding to each FPU configuration for FFT (RMSE)

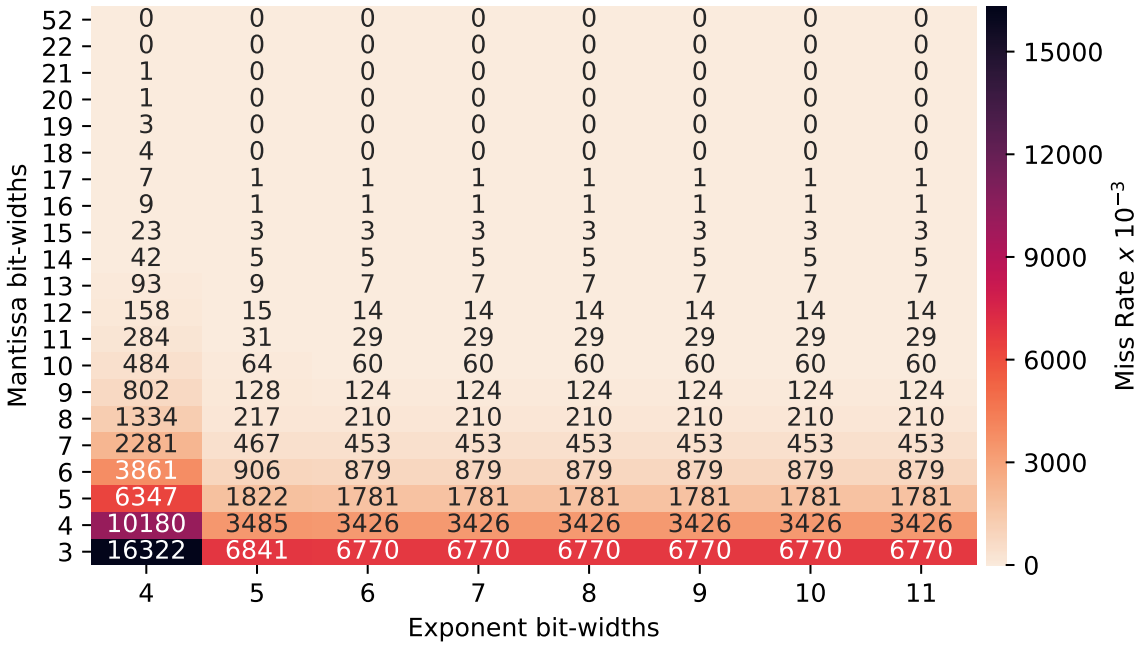


Figure 4.9: Resulting QoR corresponding to each FPU configuration for Jmeint (Miss Rate)

is sufficient to cover the dynamic range of all internal FP computations for Black-Scholes (*resp.* FFT).

Conclusion: *the QoR does not depend on the exponent bit-width E_d for all these applications except Jmeint.*

We also observe that the QoR is maximal, and its value does not change after a given mantissa bit-width threshold: 16, 21, 22, and 22 for Black-Scholes, FBench, FFT, and Jmeint, respectively. This means that executing Black-Scholes using a 16-bit mantissa provides the same QoR as the standard 52-bit double-precision FPU mantissa. However, this threshold is not guaranteed for all possible datasets. Hence, a pertinent and representative dataset should be chosen by the designer.

Conclusion: *the* `double` *data type is over-designed for all these applications.*

For some over-reduced configurations, the output results are invalid, *i.e.*, NaN, inf, *etc.* Hence the corresponding QoR is then not relevant, and these configurations are discarded (blank boxes in Fig 4.6 and 4.7).

Conclusion: *overflows/underflows are more likely to be noticed when the chosen FP format's bit-width is very small for some applications, leading to potential corrupted results.*

FBench (Fig. 4.7) demonstrates that extending the FPU bit-width does not always increase the output accuracy. Individual intermediate computations can lead to errors due to catastrophic cancellation, insufficient precision, rounding errors .... [119,120] The resulting errors either accumulate or cancel out depending on the application. So in some cases, even when the bit-width is reduced, individual intermediate instructions are less precise. Still, the errors cancel out overall, and thus the output result accidentally becomes closer to the accurate reference value. These phenomena do not appear when dealing with more conservative standard formats (32-bit Vs. 64-bit) because of the big gap between the two, which

renders the phenomenon unlikely to be observed.

Conclusion: *at a fine-grained level, a reduced precision can still lead to higher QoR at the output, due to FP phenomena such as error cancellation.*

For FFT, as shown in Fig. 4.8, the QoR highly depend on the dynamic range of the dataset since FFT consists of several intermediate multiplication-accumulations. Configurations with $E_d = 4$ and $E_d = 5$ give invalid results (NaNs, Inf), and hence they have been discarded.

Table 4.6 also shows the most optimized configurations total FPU bit-width $(1 + E_d + M_d)$ for each benchmark without QoR loss based on the performed analysis. Of course, lowering the QoR constraint will lead to more optimization and further hardware overhead reduction. Table 4.7 depicts the optimized configurations for a set of arbitrary QoR thresholds.

	Black-Scholes	FBench	FFT	Jmeint
Optimized FPU bit-width	21 bits	26 bits	29 bits	24 bits

Table 4.6: Optimized FPU bit-widths per benchmark (No QoR loss).

	Black-Scholes	FBench	FFT	Jmeint
Error Metric	RMSE	RMSE	RMSE	Miss Rate
Error Threshold	0.0005	0.005	0.5	0.05%
Optimized FPU bit-width	15 bits	19 bits	20 bits	17 bits

Table 4.7: Optimized FPU bit-widths per benchmark.

4.4.5 Challenges & Limitations

§1 Challenges Approximating individual assembly instructions comes with a few drawbacks. Since approximation is applied locally, error propagation is not controlled globally because the algorithmic structure of the original program is practically lost. However, our approach considers the final production-ready executable binary as well as compiler optimizations. This feature is unique in that it reflects the expected production-stage behavior.

§2 Limitations The approach described in Chapter 3 and implemented in this chapter is agnostic to the language (C, Fortran, *etc.*), to the context (bare-metal, OS/RTOS *etc.*), and to the FP analysis tools used upstream. It is therefore very powerful when compared to existing strategies. However, for a given FP program using complex functions, the results will depend on the underlying standard C

library implementation. Indeed, the QoR variation behavior vis-a-vis the precision reduction might differ slightly between two implementations. For instance, transcendental functions are implemented in different ways across distinct C standard library implementations (*e.g.* Newlib, Glibc, MUSL, *etc.*). Their internal use of elementary FP operations (*e.g.*, addition, multiplication) will differ, which explains the non-identical numerical results. The proposed methodology is thus C library-dependant when complex functions are invoked.

Another limitation of this approach is its data-dependency. As for all other existing SoA techniques, our strategy provides no guarantees that the application will produce an equivalent QoR for all possible input datasets. Hence, designers should carefully select representative data inputs to cover all intended application behaviors.

4.5 Conclusion

In this section, we presented AxQEMU, a tool that simulates the impact of FP approximations inserted at the assembly interpretation level on the output of applications. The proposed tool inherits many qualities from the well-known QEMU functional simulator. A use case demonstrating how it can be applied to unmixed-precision applications was presented.

In the next chapter, we will present the hardware implementation of the approach presented in Chapter 3: a multi-precision FPU that supports precise and multi-precision operation in hardware. AxQEMU will be there employed in a second use case with this time multiple precisions and for applications with mixed-precision (float and double). This will enable the selection of a set of optimal FPU configurations, for a given application and dataset couple. Then, a final step will be performed to evaluate the hardware savings and costs brought by the different options.

5 Approximate-aware Multi-precision FPU

Contents

5.1 Introduction

Approximate Computing and specifically **Transprecision Computing** applications may necessitate different **Floating-Point** computation precision requirements. Traditional full-precision is shown to be over-designed through many SoA works. This idea was also confirmed in Chapter 4 using the proposed AxQEMU approximate-aware FP simulation tool.

The previous chapters presented a new methodology that leverages FP approximations at the interpretation/execution level, ensuring non-intrusive or minimally intrusive integration with existing SoA tools (Chapter 3). In addition, Chapter 4 presented a software implementation in the form of a simulator called AxQEMU that allows studying the effects of precision variation on application-level **Quality of Result**.

However, this tool only serves as an impact simulation tool. Therefore, it is still mandatory to provide a hardware implementation that matches the SW model simulated using AxQEMU to achieve energy efficiency. For example, for a given application, an input dataset, and a QoR constraint, AxQEMU can conclude that a reduced FP format, say $(11, 36)$ for instance, is sufficient. However, to achieve an actual energy saving, the final hardware processor that will execute the application should support the reduced format $(11, 36)$ inside its FPU, which is not possible in traditional CPU cores.

Therefore, there is a need for a hardware FPU architecture that supports the reduced precision FP formats simulated by AxQEMU following the behavior described and formalized in Chapter 3. In this context, we talk about **Arbitrary Reduced Precision** (**Arbitrary Reduced Precision**) support at synthesis time, meaning that the hardware has a reduced precision (fixed at run-time). A more advanced aspect is the **Variable Precision in Time** (VPT), which supports many reduced hardware formats and allows the software to switch precisions as applications continue their execution.

This chapter presents a hardware FPU architecture, called VPT-FPU, that supports both ARP and software-controlled VPT. The proposed architecture implements the specification given in Chapter 3. The design is inspired by ARM's "big.LITTLE" CPU architectures, which combine high-performance CPU cores for computationally-heavy applications and power-efficient CPU cores for applications with lower performance needs. Similarly, the VPT-FPU architecture combines both the usual energy-demanding standard-precision (*e.g.*, `binary32`, `binary64`) computational blocks dedicated for precise computation as well as reduced precision FP formats that enable power, execution time, and energy consumption optimization.

In this chapter, the proposed architecture is presented in detail (Section 5.2), then VPT support within the proposed architecture is discussed (Section 5.3). Moreover, we show that the architecture is globally parametric and how supported reduced FP formats can be configured at synthesis time (Section 5.4). Then, we explain how the power consumption of a given application is evaluated when executed on top of a given VPT-FPU architecture (Section 5.5). Finally, two use cases will be presented; The first (Section 5.6) consists of a HW-level evaluation of the Jmeint application presented earlier in Chapter 4. The second use case (Section 5.7) illustrates the full SW and HW FP optimization process explained previously

in Section 2.3.2, including the use of our ARP technique along with SoA tools such as Promise [121] and the AxQEMU simulator to find the most optimized FPU architecture configuration that reduces power consumption while still satisfying a QoR requirement.

5.2 Proposed Architecture

We propose a hardware FPU architecture VPT-FPU enabling configurable runtime precision and a lightweight software library that facilitates integration in an existing CPU core.

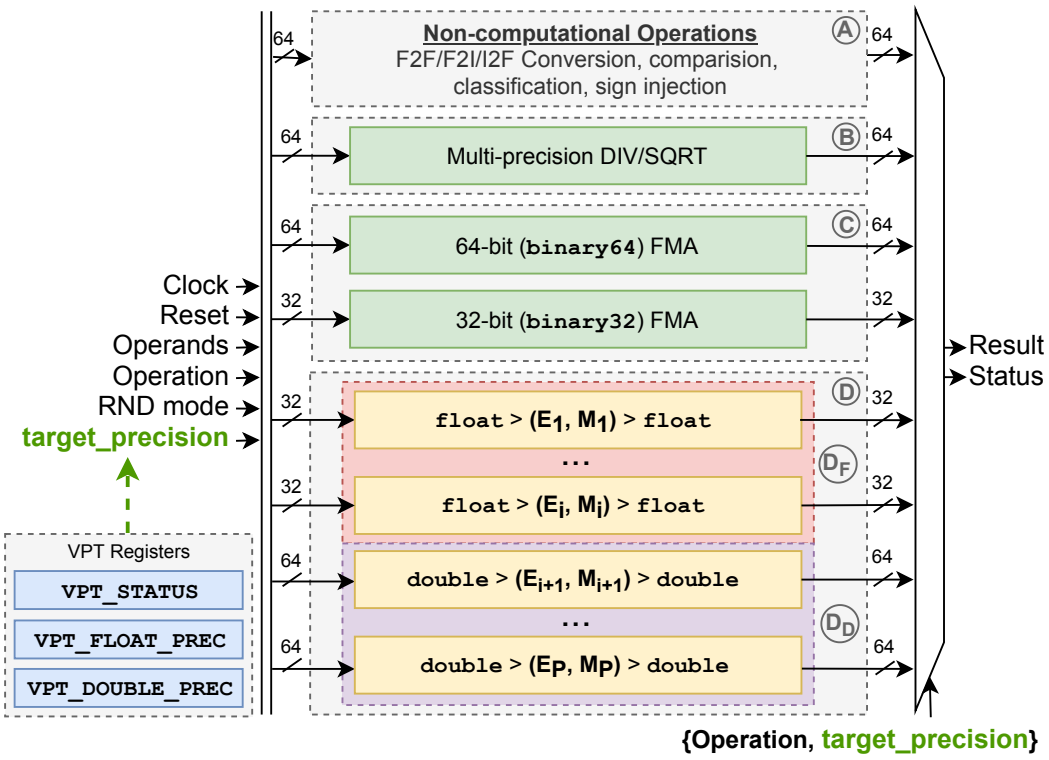


Figure 5.1: Hardware architecture of the Approximate Aware FPU

VPT-FPU contains two types of datapaths: a) precise datapaths, which contain standard-precision arithmetic operators, and b) approximate datapaths, which contain reduced arbitrary precision operators. The proposed architecture is depicted in Figure 5.1, which is composed of :

Ⓐ **Non-computational operations datapath:** performs non computational operations such as FP-to-integer and integer-to-FP conversion, comparison, input classification *etc.* In practice, the cost of these instructions is negligible when compared to computational instructions, so we chose to not optimize them.

Ⓑ **Multi-precision DIV/SQRT datapath** [156] : a block that computes division

and square root operations using an iterative algorithm implemented in hardware [157] . The precision of this block is adjustable at run-time.

Ⓒ **Precise Fused Multiply Add (FMA) datapath:** contains the precise FMA operators that perform addition, subtraction, multiplication, and multiplication-accumulation in `binary32` and `binary64` formats.

Ⓓ **Approximate FMA datapaths:** contains two sets of reduced arbitrary precision datapaths (D$_F$) and (D$_D$). Approximated *float* computations are executed in (D$_F$), whereas approximated *double-precision* computations are executed in (D$_D$).

For each supported non-standard FP format (E_p, M_p), $1 \leq p \leq P$, an approximate block is integrated either inside (D$_F$) or (D$_D$) at the RTL level at synthesis time. This means that the set of supported precisions is fixed, once synthesized. Each of these approximate blocks contains:

- A reduction block that converts the inputs from the original format (`binary32` or `binary64`) to the target reduced format (E_p, M_p).
- An FMA computational operator in (E_p, M_p) format.
- An extension block that converts the result from (E_p, M_p) back to the original format.

The blocks Ⓐ, Ⓑ, and Ⓒ constitute the original standard RV64FD FPU. The block Ⓑ containing DIV/SQRT operators do not need to be duplicated or surrounded by reduction and extension blocks since their precision can be adjusted via a precision selection input signal that varies at run-time. This is why only FMA blocks are duplicated.

Computation of the status register In addition to the numerical result computed, an FPU also returns a 5-bit status register explained in detail in Appendix B.3.3. Since the FPU is modified to consider approximations, the five flags are redefined as follows:

- **Inexact bit (NX)**

  If the operation is executed on the exact datapath Ⓒ, this flag is set when the final result cannot be represented precisely in the target full-precision format.

  Otherwise, if the instruction is executed on the approximate datapath Ⓓ, the inexact bit is set if at least one of the reduction operations yields an inexact reduced operand $\tilde{v}_i$, or if the result of $F\_\mathrm{op}_l^R$ is inexact. The extension phase does not affect this flag.

- **Invalid bit (NV)**

  If the operation is executed on the exact datapath Ⓒ, this flag signals an invalid FP operation (*e.g.*, multiplying ∞ and zero).

  Otherwise, if the instruction is executed on the approximate datapath Ⓓ, this flag is set if $F\_\mathrm{op}_l^R$ performed an invalid FP operation. The reduction and extension stages do not affect this flag.

- **Division by zero (DZ)**

  If the operation is executed on the exact datapath (C), this flag indicates an operation involving a division by zero.

  Otherwise, if the instruction is executed on the approximate datapath (D), it is set to the same value as the DZ flag resulting from $F\_\mathrm{op}_l^R$.

- **Underflow (UF)**

  If the operation is executed on the exact datapath (C), this flag indicates if the computed result is too small to be represented in the target full-precision format.

  Otherwise, if the instruction is executed on the approximate datapath (D), it is set if at least one of the reduced operands $\tilde{v}_i$ or the result of $F\_\mathrm{op}_l^R$ is too small to be represented in the l format.

- **Overflow (OF)**

  If the operation is executed on the exact datapath (C), this flag indicates if the computed result cannot be represented as a finite value in the target full-precision format.

  Otherwise, if the instruction is executed on the approximate datapath (D), it is set if at least one of the reduction blocks or the $F\_\mathrm{op}_l^R$ operator yielded an overflow.

5.3 VPT Support

5.3.1 Custom VPT Registers

Register Name	Address	Description
`VPT_STATUS`	`0x800`	Stores the actual status of the VPT. When it is set to zero the VPT is deactivated, otherwise it is enabled.
`VPT_FLOAT_PREC`	`0x801`	Stores the actual precision of `float` operations, as a one-hot encoded value[1]. If the `VPT_FLOAT_PREC` is all zeros then the precision is default (23-bit mantissa).
`VPT_DOUBLE_PREC`	`0x802`	Stores the actual precision of `double` operations, as a one-hot encoded value. If the `VPT_DOUBLE_PREC` is all zeros then the precision is default (52-bit mantissa).

Table 5.1: VPT registers

To support VPT operation, three custom Control and Status Registers (CSR) have been added to the FPU: the `VPT_STATUS` register, which is used to enable/disable the VPT feature, and the two registers `VPT_FLOAT_PREC` and `VPT_DOUBLE_PREC`,

[1] When operating in M bits, the M^{th} bit is set to 1, others are set to 0.

which respectively configure the precision settings for float and double operations.

5.3.2 VPT Software Support

The added registers are memory-mapped when the FPU is integrated into a CPU core. This enables lightweight software support since read/write (R/W) operations can then be performed using the usual CSR assembly instructions, *i.e.*, no special compiler modification is needed. The additional CSRs belong to the custom R/W user-level address space [138] , which means that no particular machine- or supervisor-level privileges are needed to perform R/W operations.

A small set of **Hardware Abstraction Layer** (HAL) functions has been developed to enable the programmer to select float and double operations' precision via software. Of course, at the SW level, the programmer can only select a precision among the ones supported in hardware (E_p, M_p), $1 \leq p \leq P$. If the selected precision is not available in hardware, the VPT-FPU defaults to standard precision without raising exceptions. The proposed HAL functions are wrappers for the CSR read/write assembly instructions. For a seamless integration in existing applications, the functions are bundled as a header-only library.

Listing 2 depicts a few HAL examples. For instance, `vpt_set_prec_float` (resp. `vpt_set_prec_double`), sets the precision of float (resp. double) operations *i.e.*, by configuring the content of the register `VPT_FLOAT_PREC` (resp. `VPT_DOUBLE_PREC`) to `1 << M`, where M is the target arbitrary reduced precision.

```
void vpt_enable(void);  /* enable  VPT */
void vpt_disable(void); /* disable VPT */
uint64_t vpt_set_prec_float(uint8_t M);
uint64_t vpt_set_prec_double(uint8_t M);
```

Listing 2: VPT-related HAL function prototypes

Listing 3 depicts an implementation example of the `vpt_set_prec_float()` function. This function uses the `csrrw` RISC-V assembly instruction which reads the original value of a given register and sets its content to the new value. These HAL functions are all marked as `inline`, meaning that, at compile-time, their source code will be directly embedded inside the caller function, so there is no extra call penalty.

```
inline uint64_t vpt_set_prec_float(uint8_t M){
    uint64_t returned_value = 0;
    __asm__ volatile ("csrrw %0, 0x801, %1" /* CSR Read/Write to VPT_FLOAT_PREC */
                      : "=r" (returned_value) /* output: register \%0 */
                      : "r"  (0x01 << M)      /* input: register \%1 */
                      :                       /* clobbers: none */      );
    return returned_value; // Return old register value
}
```

Listing 3: Implementation of the `vpt_set_prec_float()` *function.*

5.4 HW Customization

The architecture was implemented mainly in SystemVerilog. The source files contain packages that provide a set of configuration parameters. These packages are generated automatically from a set of Python scripts to automate the HW **Design Space Exploration** of many architectural parameters.

Figure 5.2 depicts the internal architecture of the approximate blocks inside the (D_F), and (D_D) paths and how the internal reduced format (E_p, M_p) as well as the pipelines are configured.

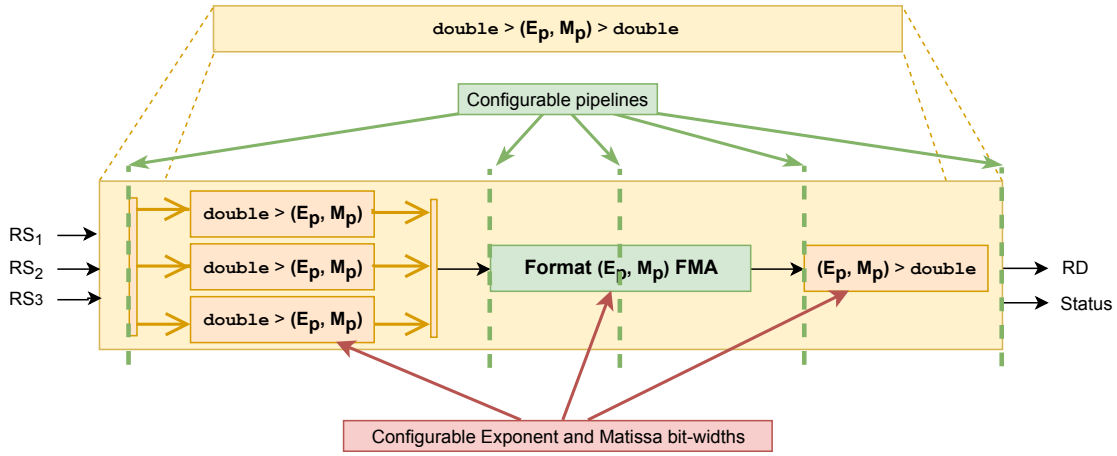


Figure 5.2: Architecture customization at synthesis time.

The implementation allows the configuration of two HW parameters at synthesis time, which cannot vary afterward:

- **Customizing the set of supported reduced FP formats**[2]**:** the python script automates the generation of the proper package parameters that allow EDA tools to generate multiple-precision hardware.

  This step allows to find a trade-off between power consumption and area occupation: supporting more formats leads to higher flexibility and finer-grained QoR control, but supporting more formats than needed leads to an increasing area overhead.

- **Customizing the pipeline registers of supported reduced formats:** the python script also defines how many registers should be inserted inside the reduced precision blocks. The inserted register blocks are then automatically distributed and retimed[3] using EDA tools such as Synopsys Design Compiler.

  This step optimizes execution time: more registers inside the custom precision blocks lead to higher maximum frequency (shorter critical path) but more cycles to execute the assembly instruction.

The Python automation scripts and the architecture's flexibility helped us quickly find interesting trade-offs for all these parameters.

[2]Denoted as $\mathcal{X}_{ARP}$ in Section 3.4.

[3]Register retiming is a technique used by EDA tools to redistribute registers inside an implementation to minimize critical path, improve performance, reduce area/power consumption.

5.5 Synthesis and Evaluation Flow

In this section, the synthesis conditions of the hardware implementation will first be introduced. Then we will explain how the area, power, execution time, and overall energy consumption are evaluated for a given software application and hardware architecture.

5.5.1 Synthesis conditions

The implementation was synthesized as an ASIC, on a 28-nm FD-SOI technology node, in the typical corner (Regular V_t, 1.00V, 25°C, No Body Biasing) for a 200-MHz frequency target.

Synthesis has been performed on Synopsys Design Compiler® with automatic clock-gating enabled and default effort levels. If the number of registers inside the reduced precision paths (D$_F$) and (D$_D$) is more than three, the automatic register re-timing of Synopsys Design Compiler® is applied.[4]

Post-synthesis gate-level simulations were performed using Synopsys VCS®, and power consumption was estimated by considering both static power and dynamic switching activity associated with the application studied using Synopsys PrimeTime®.

5.5.2 HW-level evaluation flow

Evaluation at the hardware level is performed for a given SW application and a hardware configuration as depicted in Figures 5.3 and 5.4. This flow combines output from the AxQEMU SW simulator tool and hardware **Electronic Design Automation** (EDA) tools. It consists of three steps:

Test vector generation (Fig. 5.3): by simulating the execution of the SW application on top of AxQEMU with the corresponding architecture parameters (supported reduced FP format). After the simulation, AxQEMU generates a test vector containing the execution trace of the application. This execution trace reports FP operations' inputs and expected outputs in addition to the usual numerical outputs of the application. This test vector is a binary file that will be used later as a stimulus to the gate-level netlist to evaluate the power consumption through nodes activity.

Gate-level netlist generation (Fig. 5.3): a gate-level netlist is generated using the RTL description provided in SystemVerilog plus the configuration packages generated using the Python HW configuration script. The RTL is fed to Synopsys Design Compiler® with other TCL automation scripts to generate the final gate-level netlist described in Verilog. The circuit area is evaluated at this level.

[4] All experiments presented in this manuscript have 3 or less internal registers.

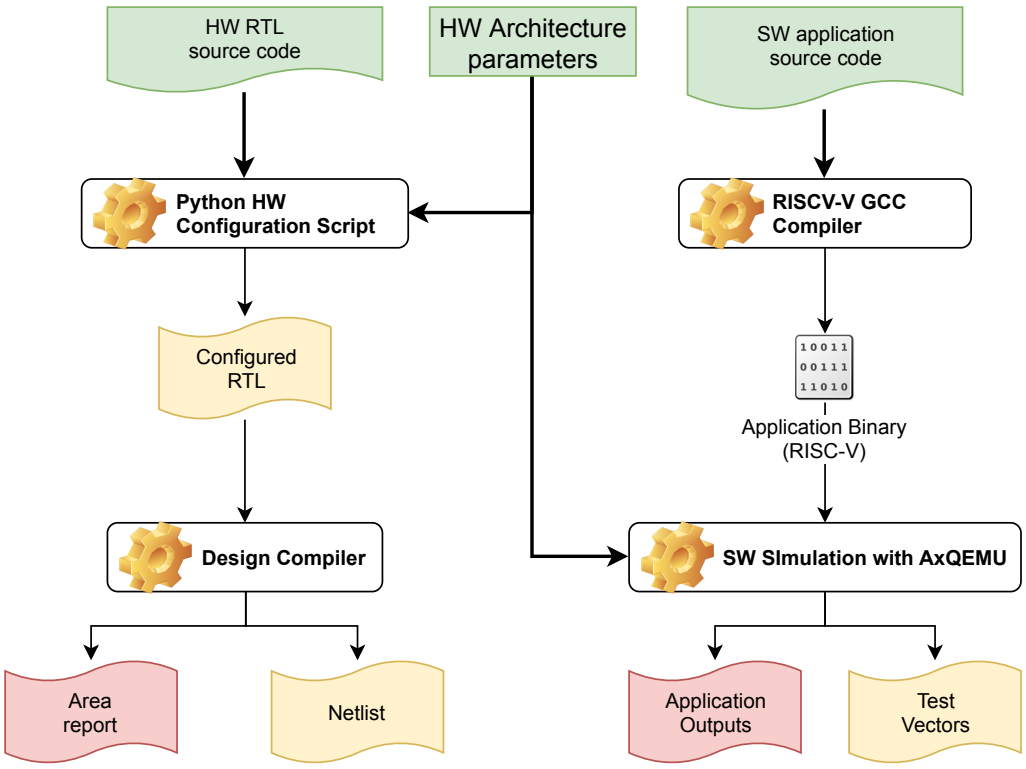


Figure 5.3: Test vector and gate-level netlist generation.

Parallel post-synthesis simulations (Fig. 5.4): in this step, the Synopsys VCS® tool is used to apply the test vectors generated from AxQEMU to generate activity files (SAIF file types). However, since the test vectors can sometimes be huge (several to hundreds of megabytes, *i.e,* millions of instructions) and might take hours to days to finish, we first split the test vectors into N smaller mini test vectors containing $\sim 100K$ to $\sim 500K$ instructions. Each piece is then initialized with a CPU architectural state (initial FPU register values, precision configuration, ...) and then launched separately in a parallel VCS process. This way, we can launch N smaller simulations that take around 1 hour or a few dozens of minutes instead of days.

Post-synthesis power/energy estimation (Fig. 5.4): After the parallel post-synthesis simulations are executed, each intermediate activity file is fed to a Synopsys PrimeTime® instance, along with the gate-level netlist to estimate the average power consumption for the N small test vectors.

Similarly to VCS®, PrimeTime® simulations are also parallelized. At the end, the results of the N test vectors are combined using another Python script to compute the total execution time, the average power consumption of the VPT-FPU hardware configuration, and the overall energy consumption of the application when executed on top of the studied HW architecture.

Please note that the post-synthesis simulations only focus on evaluating the power consumption and execution time spent by the VPT-FPU, *i.e.,* the amount of energy spent on FP computations. Other microprocessor and system parameters such as cache memory latency, memory bus contention, branch prediction, external peripherals, *etc.* are beyond the scope of this thesis.

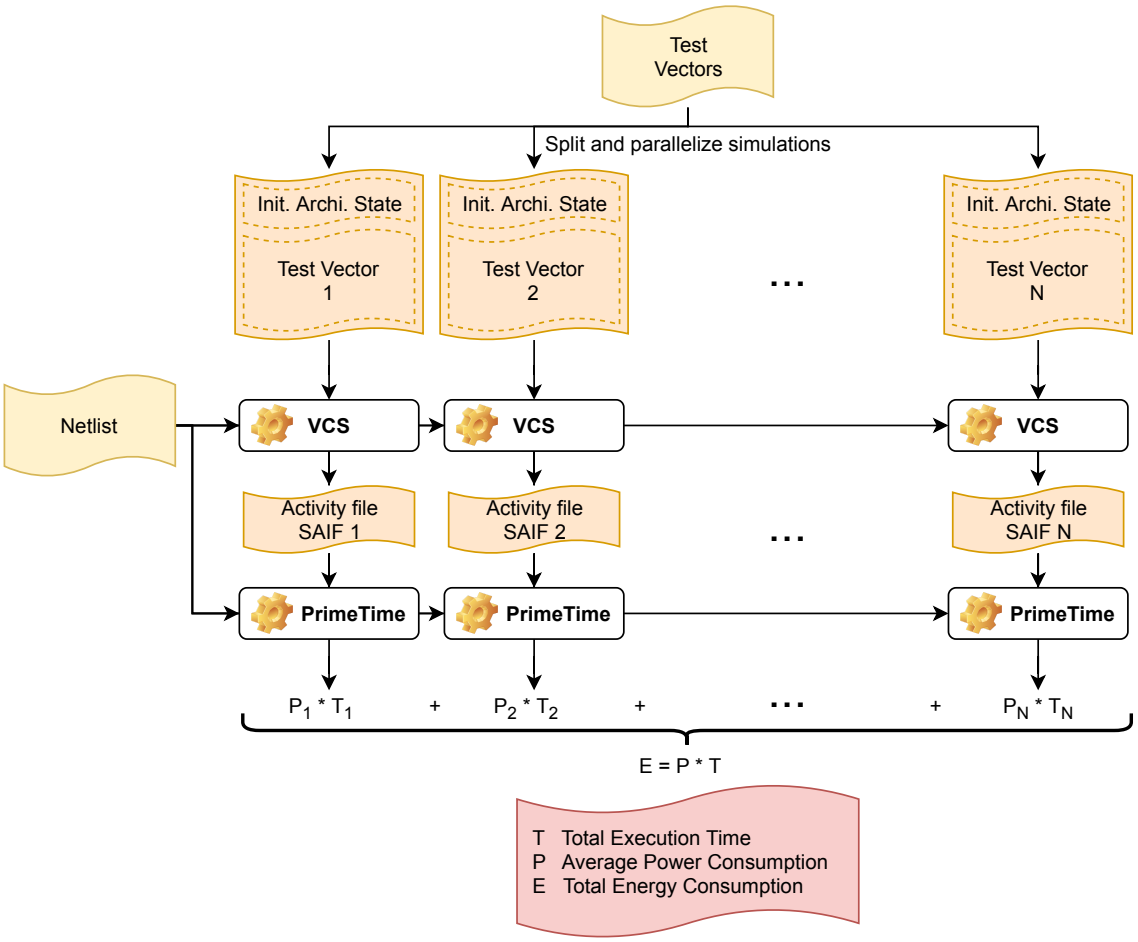


Figure 5.4: Parallel post-synthesis simulations and estimation of execution time, power, and energy consumption.

5.6 Use Case 1: HW-level Evaluation of the Fixed-Precision Jmeint

In the previous chapter (Section 4.4), we demonstrated how AxQEMU could be used to evaluate the impact of FP approximations on applications' QoR by directly applying AxQEMU to a set of benchmark applications. In other words, this previous section explored precision Vs. application-level QoR trade-offs.

In contrast, this section will explore another effect of precision reduction which is power consumption reduction and energy efficiency optimization, *i.e.*, precision Vs. power consumption trade-offs [135].

For conciseness, here we only present results regarding the Jmeint application among the AxBench benchmark [154]. This application belongs to the 3D gaming domain. It detects the intersection of two triangles in space and takes 100,000 random pairs of 3D triangle coordinates as an input.

§1 Simulation context All the variables of this fixed-precision application are declared as doubles, and the QoR metric is the intersection detection miss rate. The source code was instrumented with **Selective Approximation** (SA) macros, as explained in Section 4.3.2. After compilation, the generated binary executable

has been simulated with AxQEMU, as presented in Section 4.4.

§2 Precision-QoR trade-offs (revisited) These results show that the QoR is maximal and constant for $M_d \geq 22, E_d = 4$ as well as for $M_d \geq 18, E_d \geq 5$. Therefore, from a software perspective, the `float` type is over-designed for this application. If the hardware running the application supports other formats, such as the IEEE half-precision 16-bit format [28] which has a 5-bit exponent and a 10-bit mantissa, and the user tolerates a miss rate of 6.4% then significant energy savings and reductions in memory footprint can be reached. Using ARP, it is possible to trade accuracy against hardware overheads in a more fine-grained way.

§3 Generated HW configurations From the raw output results used in the first study (Figure 4.9), we selected a set of precisions/HW configurations each leading to a different QoR. We generated the corresponding test vectors for power-accuracy trade-offs following the evaluation flow described in Section 5.5.

In this relatively simple fixed-precision use case, we only consider a VPT-FPU containing the standard blocks (A), (B), (C) and the approximate 64-bit (D_D) data path. Moreover, the generated HW architectures only contain one reduced format $(11, M_d)$ inside the (D_D) data path. To simplify the study, the exponent bit-width is fixed at 11 (similar to `binary64`'s exponent bit-width), and only the precision M_d varies.

§4 Power / Execution time / Energy results Figures 5.5, 5.6, 5.7, 5.8 represent the variation of Jmeint's average power consumption, execution time, total energy consumption, and cell area, respectively, as a function of precision.

The bottom horizontal axis shows the different HW architectures' precision. The vertical axis depicts the power, execution time, energy, and area values normalized by the baseline `RV64IMAFD` FPU architecture (denoted `RV64FD` here). The percentages written in blue represent the **Quality of Result** expressed as the miss rate achieved by each architecture extracted from Figure 4.9. These figures allow us to have two aspects under sight: the precision Vs. QoR aspect and the precision Vs. HW-level parameters aspect.

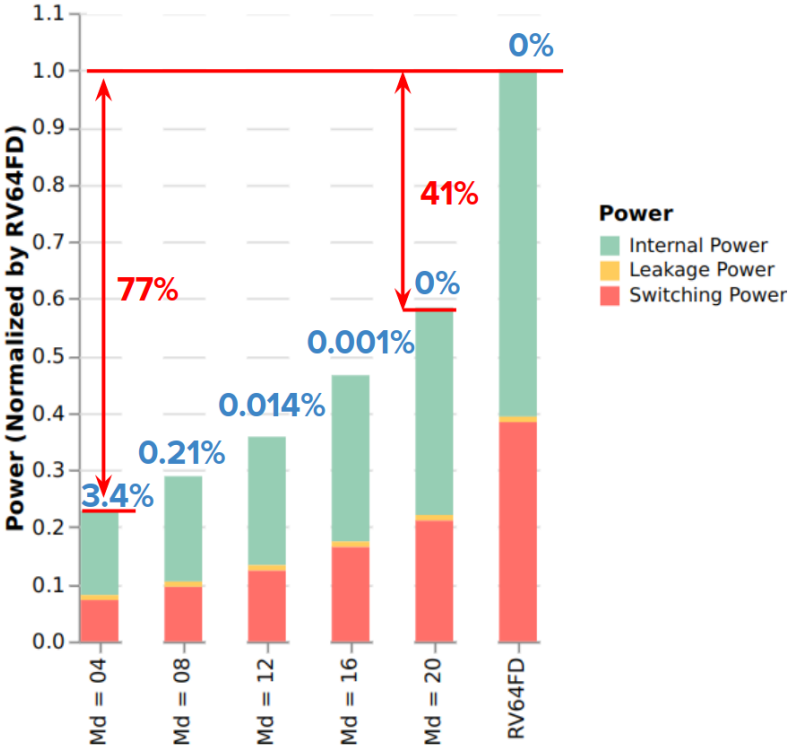


Figure 5.5: Jmeint power consumption per architecture and precision.

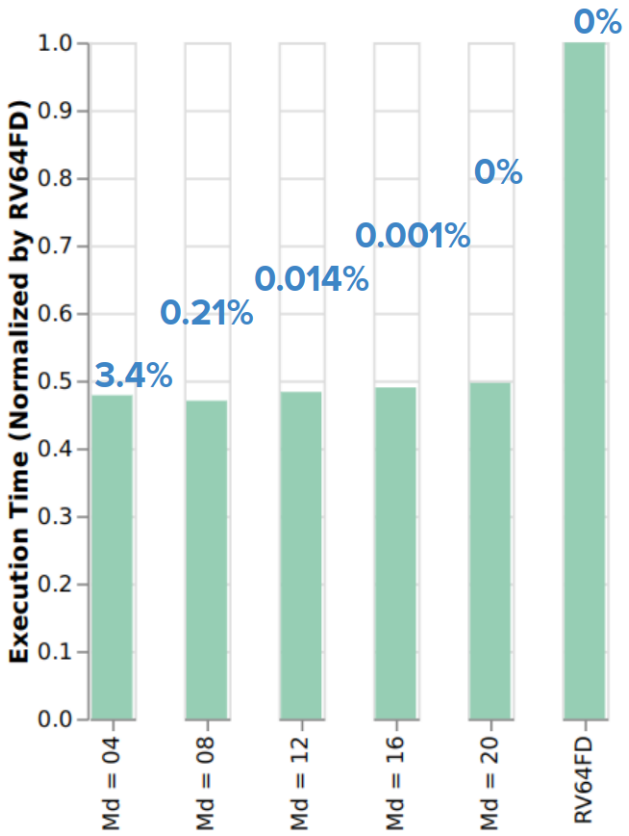


Figure 5.6: Jmeint execution time per architecture and precision.

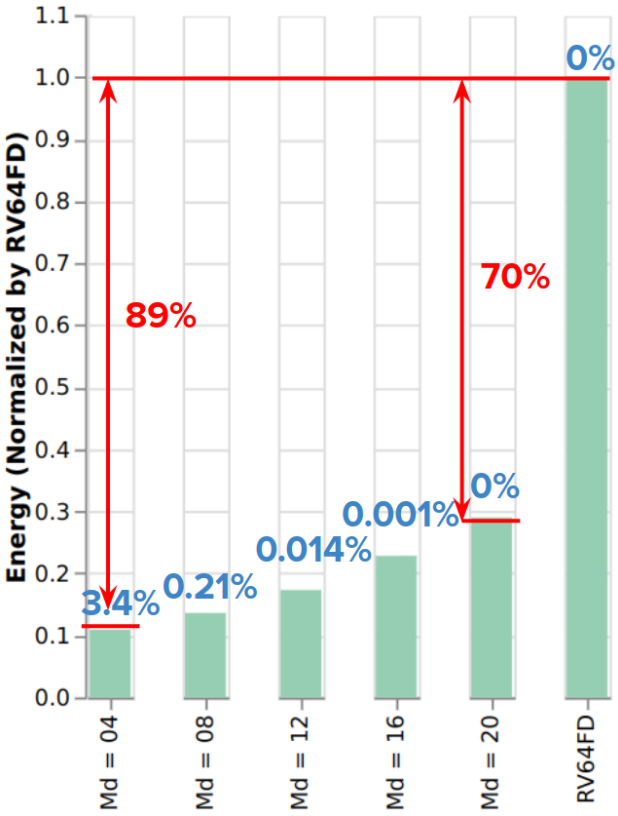


Figure 5.7: Jmeint energy consumption per architecture and precision.

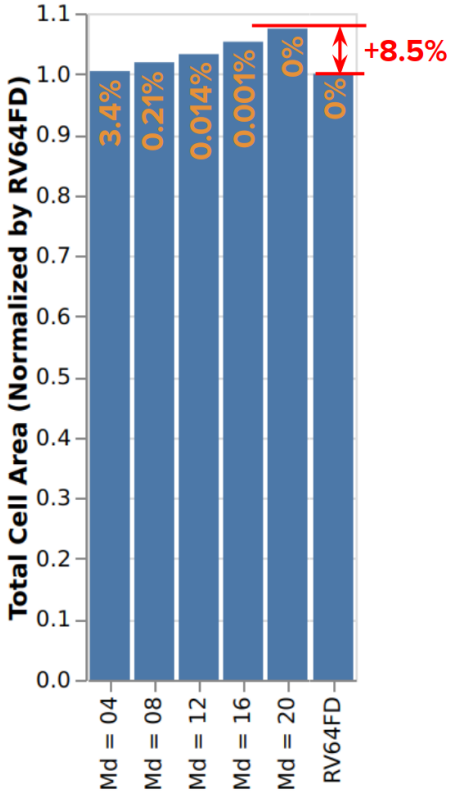


Figure 5.8: Area for each studied architecture.

Optimization with no QoR loss All the 100,000 Jmeint's inputs are correctly classified using the arbitrary reduced FP format $(11, 20)$ while achieving 41% less average power consumption, 50% less execution time and 70% less FP computation energy. However, including this reduced format's hardware in the D_D data path requires an additional area overhead of $+8.5\%$ compared to the baseline `RV64IMAFD` architecture.

Please note that the execution time gain is explained by the reduction of the number of registers inside the approximate blocks (D) of the FPU. This means that, approximated operations are executed in less cycles compared to full-precision ones, whereas the target frequency of the circuit is always fixed at 200MHz for fair comparison.

Optimization for a relaxed QoR threshold When QoR loss is tolerated, which is the case of such inherently approximate applications, the energy efficiency can further be optimized while paying less area overhead. If a QoR miss rate of 3.4% is allowed, one can achieve 77% less average power, ∼51% less execution time, and up to 89% less total energy. The advantage, in this case, is that only a $+3.4\%$ area overhead is added when compared to the baseline reference architecture.

5.7 Use Case 2: Application to Mixed-Precision applications

This section will demonstrate the following assumptions through the study of three applications from the **State of the Art**:

- AxQEMU supports existing SoA tools and mixed-precision applications, *i.e.*, ones that contain both `float` and `double` variables.
- **Arbitrary Reduced Precision** (ARP) can be used in conjunction with other SoA techniques such as **Variable Type Optimization** (VTO) to further refine FP optimization. For this cooperation use case, a detailed HW-level analysis is done to evaluate how efficient is the symbiosis in terms of precision Vs. power/execution time/energy analysis.

5.7.1 Background: Mixed-precision DSE flow

This section presents the context and some definitions that will be used in the following use cases.

As explained in Section 2.3.2, a typical SW FP algorithm implementation scenario consists on multiple steps. The first three are well explored in the **State of the Art**:

1. **Algorithm design and numerical stability analysis** [118–120] **:** establishing the mathematical foundations of the algorithm and their stability w.r.t the inputs. The main goal is to avoid instabilities such as round-off errors.

2. **Naive implementation with high-precision FP formats only:** a software implementation using all high precision formats.

3. **Coare-grained Variable Type Optimization (VTO) [121–125] :** the process of migrating as much variables as possible from high-precision to lower precisions (*e.g.*, changing `double` variables to `float`) in a given application source code while satisfying a QoR constraint. The process output is a (software) ***type configuration*** defined as follows.

Definition 5.7.1. A **type configuration** is a version of the original application, possibly with mixed-precision, where some variables are declared as floats and others as doubles.

Although VTO is necessary to optimize the memory footprint and energy consumption of a program, it is a coarse-grained optimization process. As a result, it usually produces solutions (type configurations) that are over-designed for many application classes. In our approach, we have therefore included a fourth step to minimize hardware FPU implementation through a fine-grained optimization:

4. **Fine-grained optimization using fixed Arbitrary Reduced Precision (ARP) [125–128]:** this approach takes advantage of non-standard reduced operators.

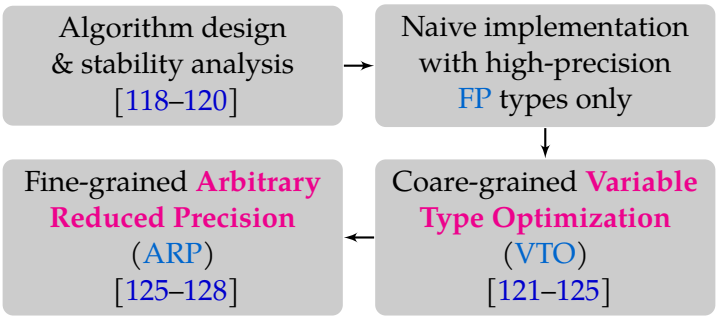


Figure 5.9: Typical FP application implementation flow.

In the remaining of this chapter, we will use the following definitions:

Definition 5.7.2. An **FPU configuration** is a 4-uple (E_f, M_f, E_d, M_d), where E_f (resp. E_d) is the exponent bit-width for the single-precision (resp. double-precision) operator, and M_f (resp. M_d) is the mantissa bit-width of the single-precision (resp. double-precision) computational operator.

For example, an FPU that supports `binary32` and `binary64` is represented as $(8, 23, 11, 52)$.

5.7.2 Problem statement

Given an application that takes as an input a dataset I, and providing a numerical output result O, our objective is to identify the optimal FPU configuration (E_f, M_f, E_d, M_d) in terms of power consumption, execution time, and overall energy consumption, subject to a **Quality of Result** (QoR) constraint on the output O.

This goal can be reached in three phases:

Phase ① Starting from an application source code, steps 1 to 3 (Section: 5.7.1) were performed using SoA tools to generate a set of valid type configurations for each QoR level constraint.

Phase ② Software design space exploration: for a given type configuration, the design space is rapidly explored to select a set of FPU bit-width configurations satisfying the QoR constraint target.

Phase ③ FPU hardware assessments: for each configuration, an estimation of the overall energy savings was made to select the best final configuration.

For this experiment, we selected the tool described in [121] which is briefly introduced in Section 5.7.3. First, the software type configurations described in [121] (Phase ①) were instrumented with AxQEMU to select a set of candidate hardware FPU configurations according to each QoR constraint (Phase ②). Then these candidates will be evaluated in hardware to select the best final optimized architecture (Phase ③).

These phases can be automated using specific search algorithms. However, in this manuscript, for the sake of clarity, all simulations were performed exhaustively.

5.7.3 The PROMISE tool

PROMISE [121] [5] was selected as the primary tool for VTO (Phase ①). Given an application and an input dataset, it applies a delta debugging algorithm [134] to find a configuration that minimizes the number of high-precision (double-precision) variables while nevertheless satisfying a given QoR constraint. Internally, PROMISE uses the CADNA library [158] , which implements Discrete Stochastic Arithmetic (DSA) [133] , a technique that estimates round-off error propagation and detects numerical instabilities in a program.

5.7.4 Evaluation and Discussion

The approach was evaluated using three applications from [121] : `arclength`, `rectangle`, and `squareroot`. For each application, an optimized type configuration is generated by PROMISE for various QoR thresholds (Phase ①, Section 5.7.2).

Benchmark QoR Metric In the remainder of this section, we use the same QoR metric as in [121] *i.e.*, the number of significant digits S computed as follows:

$$S = -\log_{10} \left| \frac{Result_{(8,23,11,52)} - Result_{(E_f,M_f,E_d,M_d)}}{Result_{(8,23,11,52)}} \right|$$

Where $Result_{(E_f,M_f,E_d,M_d)}$ is the numerical output computed when simulating the application with the (E_f, M_f, E_d, M_d) FPU configuration on AxQEMU, and $Result_{(8,23,11,52)}$ is the golden reference result obtained using the standard

[5] http://promise.lip6.fr

$(8, 23, 11, 52)$ configuration. The numerical results for these applications are non-zero.

5.7.4.1 Phase (1)

Table 5.2 lists the benchmarks studied alongside the type configurations generated. The 3rd (resp. 4th) column depicts the number of `double` (resp. `float`) variables required for each type configuration.

Every {type configuration, benchmark} pair achieves a maximum QoR (shaded cells in the 5th column) when executed exclusively in full precise mode using the standard configuration $(8, 23, 11, 52)$. In other words, if one is limited to standard FP types, only the conservative QoR thresholds that are highlighted can be obtained.

The last column depicts, for each {application, type configuration, QoR threshold} tuple, a set of candidate FPU configurations that also satisfy the given QoR constraint. Candidate FPU configurations (the 6th column) were selected based on a design space exploration process which is explained in the next paragraph.

Type Configurations	Application	# Doubles	# Floats	QoR Threshold	Candidate FPU Configurations
V10	arclength	8	1	10	(8, 3, 11, 44)
				8	(8, 3, 11, 36)
				6	(8, 3, 11, 32)
				4	(8, 3, 11, 28)
	rectangle	4	3	10	(8, 3, 11, 36)
				8-6	(8, 3, 11, 28)
				4	(8, 3, 11, 16)
	squareroot	6	2	10	(8, 3, 11, 32)
				8	(8, 3, 11, 28)
				6	(8, 3, 11, 20)
				4	(8, 3, 11, 8)
V6	arclength	7	2	6	(8, 22, 11, 32)
				4	(8, 13, 11, 28)
	rectangle	3	4	8-6	(8, 3, 11, 28)
				4	(8, 3, 11, 16)
	squareroot	0	8	6	(8, 15, 11, 4)
				4	(8, 7, 11, 4)
V4	arclength	2	7	4	(8, 22, 11, 28)
	rectangle	0	7	4	(8, 16, 11, 4), (8, 13, 11, 4)

Table 5.2: Benchmark summary

5.7.4.2 Phase (2)

An exhaustive DSE was performed for each type configuration and each application. Let us denote $C = \{c_0, c_1, ...\}$ a set of FPU configurations to be studied. Our objective was to determine the most optimized FPU configuration $c_i \in C$ satisfying the target QoR threshold. The DSE is a two-step process:

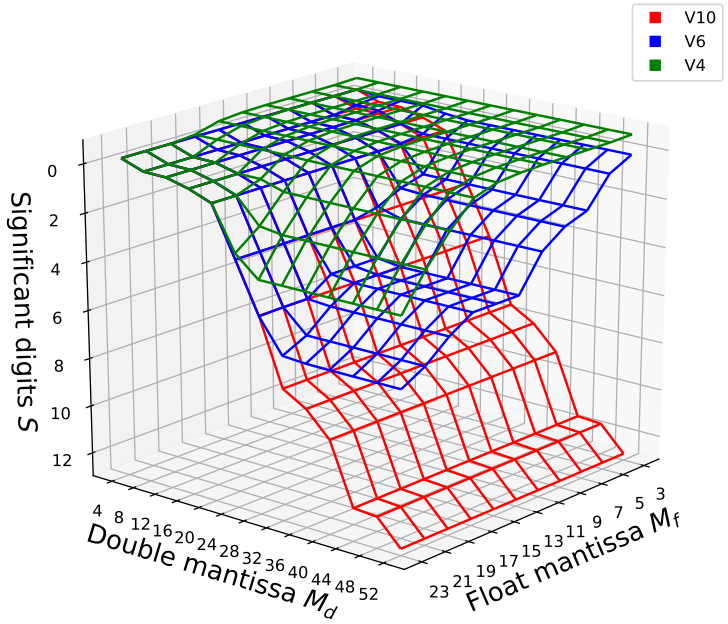
§1 Simulation First the source code corresponding to a type configuration (*e.g.* the V10 configuration of `arclength`) is compiled. Then, the binary is executed using AxQEMU for each FPU configuration $c_i \in C$. The raw QoR values (*i.e.*, the number of significant digits) for all c_i are stored in an array $\mathcal{M}$ for processing in the next step. This simulation step can be more or less time-consuming depending on the size of the input dataset, the execution time for the application itself, and the number of FPU configurations to be studied.

§2 QoR normalization Once all the simulations have been performed and the QoR array $\mathcal{M}$ constructed, a processing algorithm is applied to filter out anomalous local optimums caused mainly by error cancellation. Indeed, reducing the operator-level precision may lead to errors in the output of an operation. However, sometimes, when multiple operations are combined, the errors may cancel each other out, leading to an accidental increase in overall QoR rather than a decreased QoR. To keep results consistent, we considered the worst-case error detected (*i.e.*, the lowest QoR) among all the configurations that have an equal or greater mantissa bit-width. Thus, for each $c_i = (E_{fi}, M_{fi}, E_{di}, M_{di})$ we considered the minimum QoR value between the raw original value obtained for c_i and the QoR values determined for any higher-precision configuration $c_j = (E_{fj}, M_{fj}, E_{dj}, M_{dj})$ that satisfies $M_{dj} \geq M_{di}$ or $M_{fj} \geq M_{fi}$.

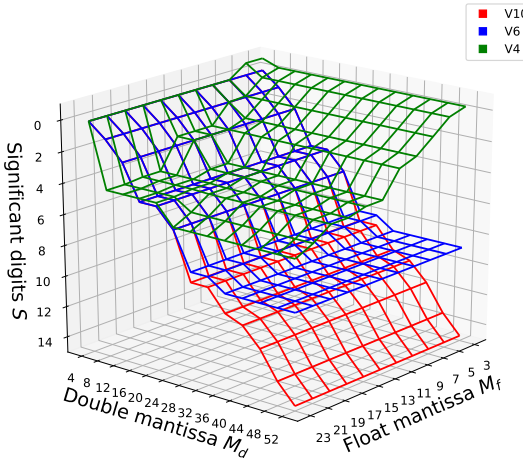
§3 Phase (2) results Phase ② (from Section 5.7.2) consists on performing a series of simulations using AxQEMU for each benchmark and each type configuration (V10, V6, V4). Figure 5.10 shows the variation of the QoR against `float` mantissa bit-width M_f and `double` mantissa bit-width M_d. Thus, Phase ② provides a set of candidate FPU configurations that produce intermediate QoR levels.

§4 Phase (2) analysis Figure 5.10 summarizes the effects of operator-level precision (M_f and M_d) on application-level results. For the same application, the impact of bit-width sizing on overall accuracy differs between type configurations. The gradient of variation of the QoR reveals some information about the sensitivity of each type configuration to low-level precision. For example, for `rectangle`, the output accuracy of the V10 type configuration is independent of the `float` mantissa bit-width (M_f). Indeed, this type configuration considers only one `float` variable, the impact of which on the QoR is negligible.

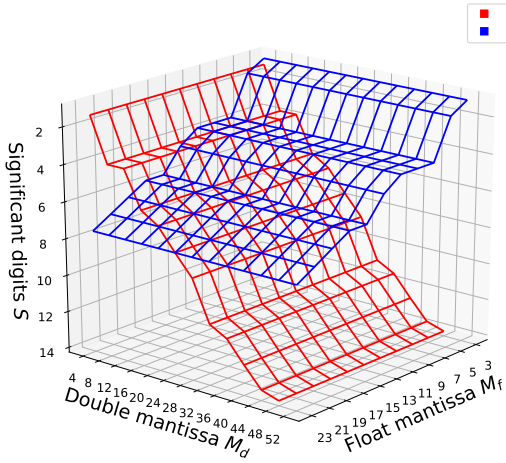
Design engineers could use Figure 5.10 to guide their selection of the best options for application implementation. Starting from a type configuration (V10, for example), the QoR threshold (and hence the consumed energy) can be selected by varying the underlying operator-level precision rather than changing the variable types. This allows more fine-grained control of the accuracy vs. energy comparison.



(*a*) `arclength`



(*b*) `rectangle`



(*c*) `squareroot`

Figure 5.10: QoR results for `arclength`, `rectangle`, *and* `squareroot`.

The last column of Table 5.2 shows the selected candidate FPU configurations. The most appropriate one can be selected based on a qualitative comparison of the bit-widths. However, a detailed hardware-level evaluation is necessary to determine which bit-width minimizes the overall energy consumption quantitatively.

In the previous section, we applied AxQEMU to some benchmarks to study the effects of FP approximations on application-level accuracy. In this section, we present a hardware-level evaluation of the selected candidates, which aimed to estimate the HW savings in terms of energy consumption thanks to ARP (Phase ③).

The methodology was applied to many applications, but for the sake of concision, we only present the `arclength` study here.

5.7.4.3 Phase (3)

§1 HW evaluation methodology The approximate-aware hardware FPU presented in the previous sections was implemented in SystemVerilog to perform the HW-level evaluation. The HW design is globally parametric so that the four parameters of the FPU configuration (E_f, M_f, E_d, M_d) are variable at design time, as explained in Section 5.2.

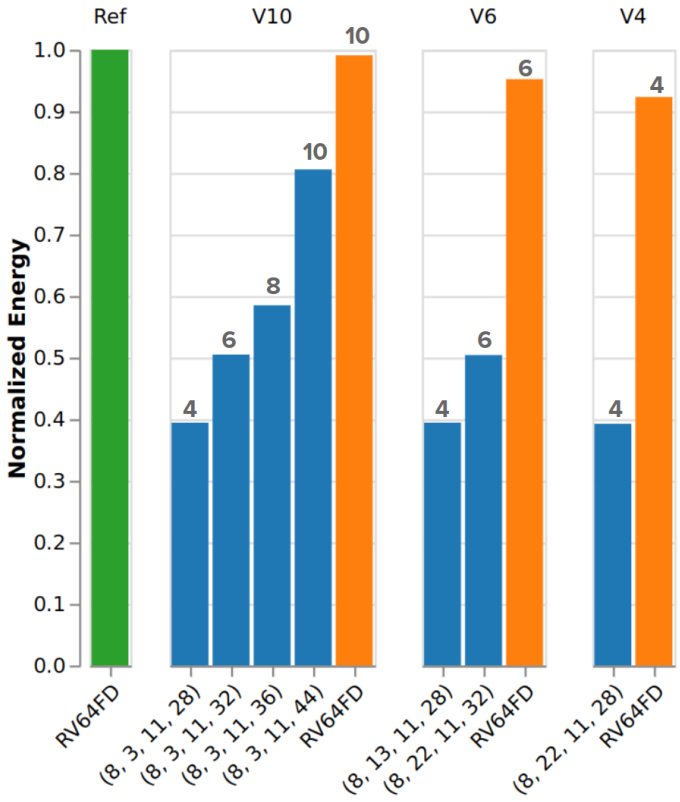


Figure 5.11: Energy vs. QoR trade-offs

§2 Phase (3) results Figure 5.11 shows the estimated energy consumed by the FPU, for each `arclength` type configuration (V10, V6, V4). To evaluate and compare the savings provided by VTO alone (Phase ①, represented in orange) vs. VTO + ARP (Phase ②, represented in blue), they are compared to a reference implementation *Ref* (represented in green). The latter is a conservative implementation (*i.e.*, a naive one, where all variables are in double-precision). The energy values are normalized by the *Ref* type configuration when executed on the `RV64IMAFD` (denoted `RV64FD`) standard architecture $(8, 23, 11, 52)$. The numbers on top of the bar plots indicate the number of significant digits S associated with each {type configuration, FPU configuration} pair.

For a fair comparison, the metric evaluation functions executed in full precision were not considered. Therefore, the results presented only relate to the computational part *i.e.*, the amount of energy spent on FP computations.

§3 Phase (3) analysis Figure 5.11 shows that the type configurations generated by PROMISE only provided 0.9%, 4.8%, and 7.7% savings on FP computa-

tion energy compared to the *Ref* conservative implementation when producing results with 10, 6, and 4 significant digits, respectively. Therefore, even when the accuracy is drastically reduced to just 4 digits, the energy-saving does not exceed 7.7%. In contrast, by using ARP, it becomes possible to save 19.4%, 49.5%, and 60.7% energy with the pairs {V10, (8, 3, 11, 44)}, {V6, (8, 22, 11, 32)}, {V4, (8, 22, 11, 28)}, respectively, without degrading QoR.

For some QoR constraints, VTO tools may not be able to identify an appropriate optimized type configuration. This was the case for all the applications presented in Case Study 2. For example, no optimized type configuration can satisfy an 8-digit QoR constraint, and the designer will therefore have to select an over-designed configuration that produces a higher QoR, such as V10. This effect is due to the coarse granularity of the standard types. However, with ARP, fine-tuning can be performed until a near-threshold, or in other words "good enough", configuration is found satisfying the constraint defined.

When quality constraints are relaxed, more interesting savings are possible by considering both software (type configurations) and hardware (FPU configurations). For example, if only 4 digits are required, the pairs {V10, $(8, 3, 11, 28)$}, {V6, $(8, 13, 11, 28)$}, and {V4, $(8, 22, 11, 28)$} are all good candidates. These combinations provide equivalent energy savings: 60.7%. Other criteria, such as the circuit area or the memory footprint, can be considered to guide the choice of the final FPU configuration. According to our synthesis results, the best compromise for this application would be the (8, 3, 11, 28) configuration, since it has the lowest circuit area overhead, which is estimated to be +27% higher than the standard `RV64IMAFD` architecture.

5.8 Conclusion

In this chapter, we presented a hardware FPU architecture, called VPT-FPU, that supports ARP and VPT with multiple reduced-precision FP formats in hardware. This VPT-FPU architecture combines the usual standard-precision computational blocks as well as reduced precision FP formats that enable power, execution time, and energy consumption optimization.

The proposed HW architecture and its software support aspects were presented in detail. We also presented the power/execution time/energy consumption evaluation flow based on EDA tools. Finally, we presented two use cases that illustrate the SW and HW FP optimization process explained previously in Section 2.3.2 including the use of our ARP technique along with SoA tools such as Promise [121] and the AxQEMU simulator to find the most optimized FPU architecture configuration that reduces power consumption while still satisfying a QoR requirement.

The uses cases presented in this chapter conclude that traditional VTO is suitable for primary coarse-grained memory optimization, whereas our technique enhance (**Arbitrary Reduced Precision**) FP computational energy optimization.

In this chapter, we only took into consideration the ARP aspect and we consid-

ered precision fixed at run-time. In contrast, the next chapter will explore the VPT aspects, where hardware supports multiple reduced-precision FP formats and the software varies its precision at run-time automatically. We will focus in particular on Iterative Methods (*e.g.*, Jacobi, Gauss-Seidel, ...) as a use case since these are inherently approximate applications.

6

Variable Precision in Time for Stationary Iterative Methods

Contents

6.1 Introduction

To achieve power efficiency, Chapter 5 presented a hardware implementation called VPT-FPU, which is configurable at design/synthesis time and can host one or many reduced precision FP formats. We demonstrated that the computation precision can be selected either statically at compilation time through memory partitioning or at run-time through a lightweight HAL library. However, the use cases presented in Chapter 5 only demonstrated static precision selection. In contrast, this chapter presents a practical use case where the VPT capability of the VPT-FPU hardware architecture allows the application to automatically tune the computation precision through the low-level HAL functions.

This chapter presents a new method that enables self-adaptive precision automatically at run-time. A theoretical and statistical study is performed in this chapter to demonstrate the drastic energy reduction in the case of iterative algorithms. These algorithms are the cornerstone of **Computer Vision** (CV) applications which are widely used in the context of **Artificial Intelligence**. Reducing the power consumption of such applications is crucial to meet the current international trends.

In this work [159, 160], we focus on applying VPT to stationary iterative algorithms such as Jacobi and Gauss-Seidel. For such applications, the requirements in terms of precision vary at run-time. This means that standard FP formats are over-designed for these workloads. Moreover, it is shown that even fixed reduced precision formats are over-designed for some parts of these applications. Hence, there is a need for FPU architectures that enable fine-grained VPT.

The concept of VPT is demonstrated by proposing for each iterative algorithm (Jacobi and Gauss-Seidel) two transformed iterative algorithms that self-adapt their computation precision automatically. These algorithms do not need intervention from the user or the programmer. The first is a conservative version and the second is a relaxed (approximate) version. Programmers can choose one of them depending on their application; the first provides more robust guarantees while achieving interesting energy gains, while the second has relaxed guarantees but with a higher power-efficiency. The software implementation of these algorithms is discussed and a set of statistical studies is performed to evaluate the effectiveness of the two approaches across many inputs.

This chapter is organized as follows:

- Application of VPT to iterative methods (Section 6.4) and proposition of two modified VPT-enabled algorithms with self-adaptive run-time precision (Section 6.5);
- An in-depth statistical verification study of the VPT's impact on Jacobi algorithm's behavior (Section 6.6); and
- ASIC implementation of a set of multi-precision hardware configurations of the proposed VPT-FPU on a 28nm FD-SOI technology node and a post-synthesis evaluation of the proposed VPT algorithms' power consumption, execution time, and overall dissipated energy (Section 6.7).

Before diving into the details, some related works are presented in Section 6.2, and the motivations behind this work are introduced in Section 6.3.

6.2 Related Works

The proposed approach takes advantage of non-standard arbitrary reduced operators [135, 161] to achieve significant gains in terms of execution time, power consumption, and overall energy consumption. The following are some related works from the literature.

§1 FP Variable Type Optimization (VTO) For a given triplet {application, input dataset, QoR constraint}, tools such as [121, 123, 124] perform coarse-grained VTO using the delta-debugging search heuristic [134]. The authors' objective is to minimize the number of high-precision variables and maximize the number of low-precision variables. For some of these tools [123, 124], the objective is to optimize for speed, whereas for others, such as Promise [121], the goal is to maximize the number of `float` variables.

§2 Non-standard/Arbitrary precision support The mentioned tools support standard IEEE 754 [28] formats only, except Precimonious [123], which also supports Intel's 80-bit format implemented as `long double` in C. The authors of fpPrecisionTuning [125] proposed an arbitrary precision impact simulation methodology based on an automatic source code transformation tool. Libraries such as FlexFloat and FloatX [127, 128] enable developers to simulate the impact of reduced precision on application-level **Quality of Result**. These tools support arbitrary precision only in simulation to help designers decide which precision is adequate for their application. However, they do not provide hardware-level support to leverage these decisions for power, execution time, or energy savings.

§3 Mixed-precision for Linear Algebra There are two categories of linear system solvers: direct solvers (*e.g.*, Gaussian Elimination, Cholesky decomposition) and iterative solvers. Carson and Higham [162, 163] have proposed a general algorithm for solving linear systems based on iterative refinement [164] with three standard FP precisions. However, this method only uses standard precisions that traditional processors support. Moreover, the technique is mixed-precision in space but not in time, *i.e.*, contains mixed-precision instructions. Still, each instruction keeps its precision for all iterations. [165] have built on the previous work and proposed a solution with five different precisions. In contrast, [166] introduced new FP data types supported in NVIDIA GPUs. Finally, others [167] proposed an adaptive scheme to reduce communication overhead by selectively storing parts of the system preconditioner in different precision formats (half, single, or double). In this chapter, we focus specifically on classical stationary iterative solvers such as Jacobi, Gauss-Seidel, Richardson, Successive Over-Relaxation, [168, 169] *etc.*

§4 Arbitrary Reduced Precision In this thesis we proposed a simulator (AxQEMU) (source code available at [170]) that simulates the impact of arbitrary reduced FP precision on applications in a non-intrusive way without modifying the source code. This approach is complementary to VTO since the tools mentioned before can be used as a starting point. Furthermore, using AxQEMU allows primary

memory footprint optimization and enables fine-grained accuracy/energy trade-offs (Chapter 5).

§5 Variable Precision in Time VPREC [129] is a software back-end component that enables non-intrusive run-time variable precision simulation in the Verificarlo [137] software toolchain. This tool simulates run-time variable precision, specifically for iterative algorithms. However, VPREC is designed for impact simulation only in software and does not provide a hardware implementation for power reduction, whereas this work targets both software and hardware-level implementations. Moreover, VPREC evaluates the needed reduced precision for each iteration through off-line and data-dependent studies performed after the execution. In contrast, we propose a technique to automatically select the adequate precision for each iteration online at run-time in this work.

6.3 Motivation

This section presents the reasons that motivated this work. First, the fact that **Floating-Point** computations constitute an essential part of iterative workloads is demonstrated. Second, we show to which extent the standard FP double-precision is over-designed for such applications using the AxQEMU **Arbitrary Reduced Precision** simulator [135, 161].

In the rest of this chapter, let P be the maximum number of precisions supported by the VPT-FPU (Section 5.2, Figure 5.1) and p an integer index such that $1 \leq p \leq P$. Then, let $(E_1, M_1), \ldots, (E_P, M_P)$ denote the list of reduced precision FP formats supported in hardware.

6.3.1 FP computation usage in Jacobi and Gauss-Seidel

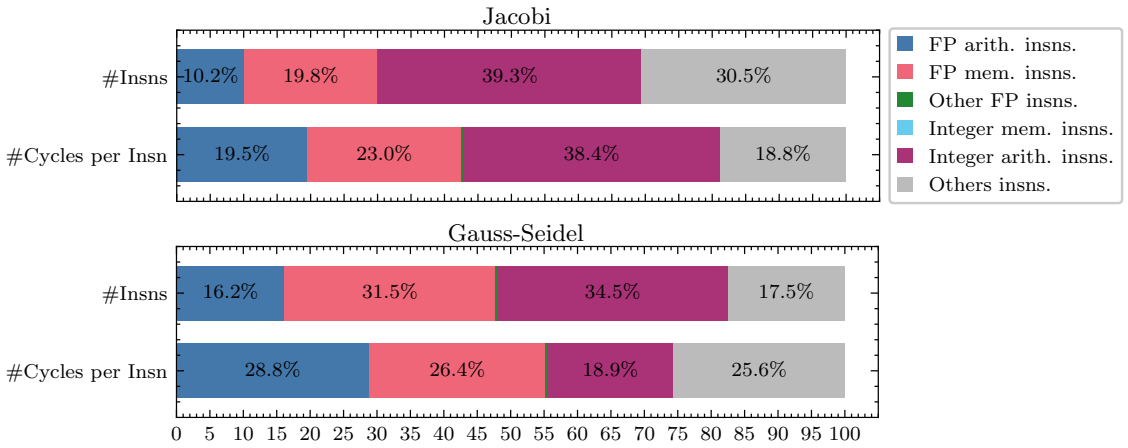


Figure 6.1: Dynamic instructions breakdown and number of cycles per instruction for Jacobi and Gauss-Seidel. Both applied to the same random input with a tolerance threshold $\texttt{TOL} = 10^{-12}$.

The first motivation behind this work is the fact that iterative algorithms such as Jacobi and Gauss-Seidel spend a lot of time performing **Floating-Point** (FP) computations. To evaluate the execution time associated with such computational operations, a cycle-accurate simulation is performed.

To do that, a cycle-accurate processor model (called CVA6 [171]) running at 200MHz is used. The Jacobi and Gauss-Seidel algorithms have been executed on top of this model with a randomly generated input. Figure 6.1 depicts dynamic assembly instructions' breakdown for both Jacobi and Gauss-Seidel, *i.e.*, how many instances of assembly instructions have been issued through the processor pipeline during the execution of the application.

As depicted in Figure 6.1, for a 10^{-12} tolerance threshold, 10.2% (resp. 16.2%) of the total assembly instructions executed are FP arithmetic instructions, and 19.8% (resp. 31.5%) are FP memory instructions for Jacobi (resp. Gauss-Seidel). The remaining portions concern instructions manipulating integers (arithmetic and memory) as well as other instructions such as branch and control flow instructions, system instructions that manipulate CSRs, *etc.*

Although executed FP arithmetic and memory instructions are limited to 10.2% (resp. 16.2%) and 19.8% (resp. 31.5%), they constitute 19.5% (resp. 28.8%) and 23.0% (resp. 26.4%) in terms of how many cycles are actually spent on each instruction for Jacobi (resp. Gauss-Seidel).

These statistics might vary slightly from one input to another and depend on the SoC parameters (cache parameters, memory bus / interconnect, ...). However, **it is safe to say that targeting FP computation optimization for these applications is a good decision**. Moreover, when the approach is combined with classic SoA techniques, designers can also benefit from low memory overhead in addition to the computation gains afforded by our technique.

6.3.2 The limitation of Fixed Arbitrary Reduced Precision

The second motivation behind this work is the fact that even fixed ARP is over-designed for iterative algorithms. Figure 6.2 shows the impact of precision variation on the **Convergence Profile** (variation of the computed solution's accuracy at each iteration *C.f.*, definition 6.4.1) of Jacobi when operating on different **Arbitrary Reduced Precision**s. The reference application is simulated using the AxQEMU simulator [135, 161] for multiple precisions (4, 8, ..., 48, 52). Each of the colored lines shows the evolution of the error metric when the application is executed with ARP. For each simulation, the precision is defined at launch time and stays constant at run-time.

This example demonstrates that it is unnecessary to compute all iterations with double-precision. For instance, for Jacobi to reach a target 10^{-10} error threshold, the designer only needs an FP format with a 32-bit mantissa, *i.e.*, the FP format (11, 32), which has a total bit-width of 44 bits. Similarly, for Gauss-Seidel, a 28-bit mantissa, *i.e.*, a total bit-width of 40 bits is sufficient to reach a target error threshold of 10^{-10}.

For Jacobi, even though the (11, 32) format will dissipate less power than the original `binary64` format, there is still room for significant improvement. For instance, the (11, 32) format is still over-designed for iterations 0 to 457. A similar case could be made for Gauss-Seidel.

As a conclusion, the standard FP formats are over-designed for these applications and ARP optimizes the energy consumption compared to the original reference precision. However, there is still room for improvement using an **Arbitrary Reduced Precision** if it is variable at run-time. This would allow precision to be varied during the execution depending on the error threshold needed. In this chapter, the VPT-FPU presented in Chapter 5, is used with two modified self-adaptive algorithms that automatically vary the computation precision without user intervention.

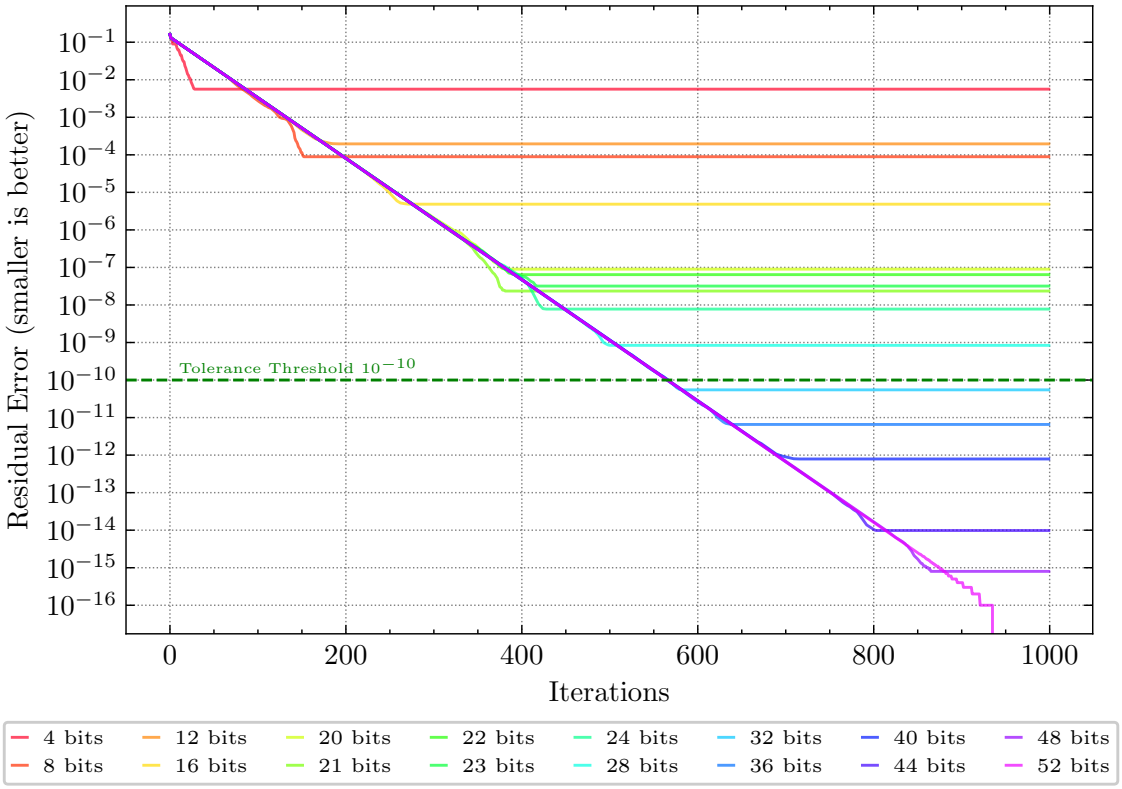


Figure 6.2: The variation of solution accuracy when Jacobi is executed for each fixed **Arbitrary Reduced Precision**.

6.4 Iterative Methods: Mathematical Foundations

The Jacobi and Gauss-Seidel iterative algorithms are used as case studies to evaluate the gains and the limitations of the proposed VPT-FPU. The original algorithms are presented as well as two VPT-enabled versions that benefit from the run-time variable precision capability.

6.4.1 Presentation of Jacobi and Gauss-Seidel iterative methods

Stationary iterative methods are algorithms that determine the solution of a square $n \times n$ system of linear equations, in the form :

$$\mathbf{A}\vec{x} = \vec{b} \tag{6.1}$$

Given the $n \times n$ real coefficient matrix $\mathbf{A}$ and the right-hand side n-vector $\vec{b}$, an iterative algorithm aims to find the unknown vector $\vec{x}$ satisfying equation 6.1. This equation can be transformed to a fixed-point iteration [168]. The system is solved by computing, at each iteration $(k+1)$, the $(k+1)^{th}$ approximation of the vector $\vec{x}^{(k+1)}$ as a function of the previous result $\vec{x}^{(k)}$. This chapter focuses on two main iterative algorithms to solve this problem based on the fixed-point method:

- The Jacobi method (published in 1845), whose formula can be written as follows:

$$x_i^{(k+1)} = \frac{1}{a_{ii}} \left(b_i - \sum_{j \neq i} a_{ij} x_j^{(k)} \right), \quad i = 1, 2, \ldots, n. \tag{6.2}$$

  where $x_i^{(k+1)}$ is the i^{th} element of the vector $\vec{x}^{(k+1)}$.

- The Gauss-Seidel method (published in 1874), which uses a slightly different equation:

$$x_i^{(k+1)} = \frac{1}{a_{ii}} \left(b_i - \sum_{j=1}^{i-1} a_{ij} x_j^{(k+1)} - \sum_{j=i+1}^{n} a_{ij} x_j^{(k)} \right) \tag{6.3}$$

These methods and others are well documented in the literature [131, 168].

6.4.2 Convergence of Iterative Algorithms

To solve this problem iteratively, the convergence of the system should first be mathematically ensured. In the remaining of this chapter, we assume $\mathbf{A}$ to be **strictly diagonally dominant**: the elements of matrix $\mathbf{A}$ satisfy the following condition $|a_{ii}| > \sum_{j \neq i} |a_{ij}|$. For both Jacobi and Gauss-Seidel, this is a sufficient (but not necessary) condition to ensure the convergence for any initial guess vector $\vec{x}^{(0)}$ [168, Theorem 4.5, p. 111].

The iterative algorithm reaches convergence when the computed approximation $\vec{x}^{(k)}$ is very close to the exact solution of the system $\vec{x}^*$, *i.e.*, the *forward error* $\vec{e}^{(k)}$ defined below is small enough

$$\vec{e}^{(k)} = \vec{x}^{(k)} - \vec{x}^* \tag{6.4}$$

However, the exact solution $\vec{x}^*$ is unknown. This is why the error of the computed solution is estimated in an indirect way using a *convergence metric*. The latter is a scalar that represents the evolution of the solution accuracy at each iteration k. The idea is to compute the convergence metric at the end of each iteration and

stop the iterative process when progress is no longer being made, *i.e.*, when the computed metric is below a given user-defined threshold denoted TOL. The result of this comparison is called the *stopping criterion*.

For the sake of simplicity, the **distance between two consecutive results** is chosen as metric. It is defined as follows:

$$||\vec{x}^{(k+1)} - \vec{x}^{(k)}|| = \sqrt{\sum_{i=1}^{n}(x_i^{(k+1)} - x_i^{(k)})^2} \tag{6.5}$$

where $||.||$ refers to the Euclidean norm. The metric is computed at the end of each iteration. Other metrics (*e.g.*, **residual error** $||\vec{b} - \mathbf{A}\vec{x}||$**,**) can also be used.

At the end of each iteration, the stopping criterion is evaluated to 1) identify when the *forward error* $\vec{e}^{(k)}$ is small enough to stop iterations [169, p. 63], 2) detect when the error is no longer decreasing or decreasing too slowly, and 3) limit the maximum amount of time spent iterating.

The following definitions are provided for the remaining of this chapter:

Definition 6.4.1. The **Convergence Profile** (CP) is the curve representing the variation of the convergence metric (*e.g.*, $||\vec{x}^{(k+1)} - \vec{x}^{(k)}||$) through iterations k. An **average** CP is an average curve computed across many inputs.

Definition 6.4.2. The **Precision Variation Profile** (PVP) is the curve representing the variation of operating precision (M_p) through iterations k. An **average** PVP is an average curve computed across many inputs.

6.5 Implementation of VPT-enabled Iterative Methods

This part presents the implementation details of iterative algorithms.

6.5.1 The original algorithm

The following algorithm (Algo. 1) depicts a typical implementation of an iterative method with only standard precision, (*i.e.*, single-precision/double-precision) with no custom variable precision involved.

The original algorithm contains a main loop (lines 2-6), where the elements $x_i^{(k+1)}$ of the solution vector $\vec{x}^{(k+1)}$ are computed in line 3, using either Equation 6.2 for Jacobi, or Equation 6.3 for Gauss-Seidel. Then, the convergence metric is computed according to Equation 6.5 (line 4). After that, the stopping criterion evaluation is performed by checking if the computed metric is lower than the tolerance threshold TOL or if the number of iterations k has reached its limit MAX_ITER. If the stopping criterion is satisfied, the algorithm stops and returns the last computed result. Otherwise, the algorithm continues until reaching convergence in the next iterations or potentially reaching the maximum iterations limit MAX_ITER.

Algorithm 1: general structure of iterative methods

Inputs : $\mathbf{A}$: a diagonally dominant matrix of size $n \times n$,
$\vec{b}$: the right-hand side vector of size $n \times 1$,
$\vec{x}^{(0)}$: an initial guess vector of size $n \times 1$,
`MAX_ITER` : maximum number of iterations,
`TOL` : original global error threshold.

Output: $\vec{x}^{(k+1)}$: the solution of the linear system.

1 $k \leftarrow 0$
2 **repeat**
3 Compute $\vec{x}^{(k+1)}$ ▷ Implements equation 6.2 or 6.3
4 $\texttt{metric}^{(k+1)} = \texttt{compute\_metric}(\vec{x}^{(k)}, \vec{x}^{(k+1)})$ ▷ Compute convergence metric (Eq. 6.5)
5 $k \leftarrow k + 1$ ▷ Continue until convergence is reached
6 **until** $\texttt{metric}^{(k+1)} \leq \texttt{TOL}$ **or** $k \geq \texttt{MAX\_ITER}$;
7 **return** $\vec{x}^{(k+1)}$

6.5.2 The transformed algorithm

To take advantage of the VPT-FPU presented in Section 5.2 of Chapter 5, Jacobi and Gauss-Seidel algorithms should be manually transformed. Algorithm 2 depicts the general structure of an iterative method with VPT enabled. The regions added to the original baseline implementation (depicted in Algo. 1) are colored in blue.

Our transformation methodology consists of starting the process with the lowest possible precision and increasing it gradually until convergence. In this example, both the original and the VPT-enabled algorithms are applied to the same input matrix $\mathbf{A}$ and vector $\vec{b}$ for comparison. These inputs are generated randomly with a randomization seed equal to zero. The steps of the transformation process are:

1. **Define a list of available precisions supported in hardware** $\{M_1, \ldots, M_P\}$ (input of Algo. 2).

2. **Define a list of intermediate tolerance thresholds** $\texttt{TOL}_p$ (input of Algo. 2).

   Since the objective is to increase precision gradually, the intermediate thresholds $\{\texttt{TOL}_0, \texttt{TOL}_1, \texttt{TOL}_2, ...\}$ should be defined for each one of the intermediate precisions $\{M_0, M_1, M_2...\}$.

3. **Enable VPT at the beginning of the algorithm** (line 2 of Algo. 2).

4. **Iterate over the supported precisions to gradually improve the accuracy of the solution.**

   The outer loop (lines 3-11) iterates over the available precisions M_p. At the beginning of each outer loop iteration, the precision is set to M_p with the function `vpt_set_precision(`M_p`)` which configures the internal registers of the FPU to use the precision M_p.

   Inside the inner loop iteration (lines 6-10), the vector $\vec{x}^{(k+1)}$ as well as the metric are computed using the intermediate precision M_p:

a) If the intermediate tolerance threshold $\texttt{TOL}_p$, corresponding to the precision M_p, is reached, the algorithm exits the inner loop and moves on to the next higher precision M_{p+1}.

b) If the maximum number of iterations `MAX_ITER` is reached, the algorithm exits both loops and returns the last computed $\vec{x}^{(k+1)}$ vector.

c) Otherwise, the inner loop continues with the same precision M_p until convergence or until `MAX_ITER` is reached, and the outer loop continues until scanning all available precisions or until reaching `MAX_ITER`.

This process is illustrated through the example depicted in Figure 6.3. It shows the evolution of the **Convergence Profile** (left axis), with iterations (horizontal axis) for the double-precision reference original algorithm (continuous blue line) and the VPT-enabled algorithm (continuous red line) of Jacobi.

In this example, a set of 13 supported precisions $\{4, 8, ..., 52\}$ is considered (right axis) to which 13 tolerance thresholds $\{2^{-4}, 2^{-8}, ..., 2^{-52}\}$ are associated. The figure also shows how the computation precision M_p of the VPT-enabled algorithm increases at run-time (orange dashed line, right axis). The intermediate thresholds $\texttt{TOL}_p$ are marked in green.

As shown in Figure 6.3, the precision is increased from M_p to M_{p+1} when the convergence metric reaches the intermediate threshold $\texttt{TOL}_p$. Through this example, it is shown that the VPT-enabled algorithm follows the same trend and provides the same accuracy at the input while operating with much lower and auto-adaptive precision.

The presented methodology does not alter the convergence of the algorithm, and programmers can apply it to other applications, (*e.g.*, Successive Over-Relaxation (SOR), Richardson method, *etc.* [168]) as long as its convergence is guaranteed mathematically. In strict logic, we verify that the input couple $(\mathbf{A}, \vec{b})$ remains strictly diagonally dominant for each reduced precision. The variation of the **Convergence Profile** for Gauss-Seidel is depicted in Figure D.1 of Appendix D.

The choice of intermediate thresholds is critical: the smaller they are, the harder they can be reached. On one hand, choosing thresholds that are easy to reach will lead to premature precision increment; hence more iterations will be spent on higher precisions, and power consumption will be increased. On the other hand, if the thresholds are very difficult to reach for a given precision, it can cause convergence stagnation. In this chapter, two threshold policies are proposed for choosing these thresholds.The first one provides conservative thresholds and is explained in Paragraph 6.5.2.1. The second generates smaller thresholds and handles stagnation cases. The latter is detailed in Paragraph 6.5.2.2.

6.5.2.1 Details of Threshold Policy (1): conservative thresholds

An intermediate threshold should be computed for each precision M_p. With this threshold policy, the computed thresholds are more conservative, *i.e.*, they are sufficiently high, which makes them more reachable with a given precision. They are mathematically computed according to the smallest distance between two consecutive points in a given FP format.

In the case where the distance metric is used for convergence, an upper bound can be computed in the precision M_p, by assuming that the distances between

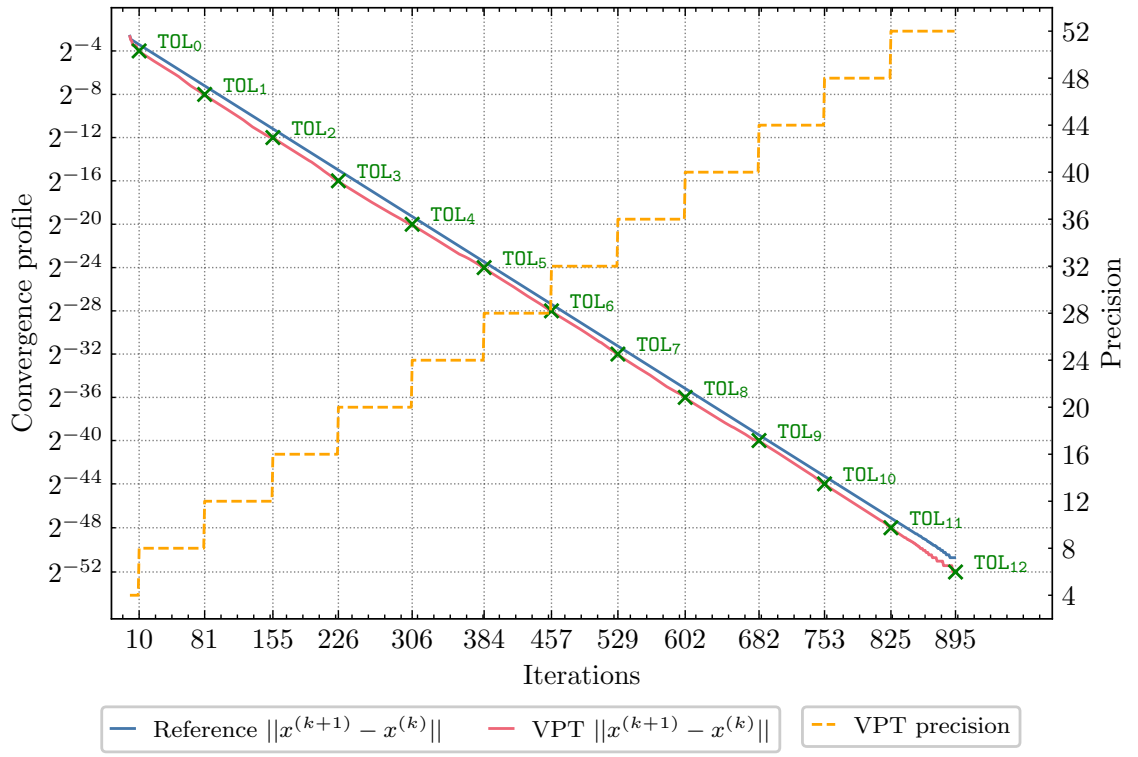


Figure 6.3: The **Convergence Profile** *of the original and the VPT-enabled Jacobi applied to one randomly generated input.*

Algorithm 2: VPT-enabled iterative methods

Inputs : $\mathbf{A}$: a diagonally dominant matrix of size $n \times n$,
$\vec{b}$: the right-hand side vector of size $n \times 1$,
$\vec{x}^{(0)}$: an initial guess vector of size $n \times 1$,
`MAX_ITER` : maximum number of iterations,
`TOL` : original error threshold.
$M_1 \dots M_P$: available precisions,
$\texttt{TOL}_1 \dots \texttt{TOL}_P$: intermediate error thresholds.

Output: $\vec{x}^{(k+1)}$: the solution of the linear system.

```
1  k ← 0, p ← 0
2  vpt_enable()                                   ▷ Enable VPT
3  repeat
4      p ← p + 1                                  ▷ Increase precision index
5      vpt_set_precision( M_p )                   ▷ Set precision to M_p
6      repeat
7          Compute x⃗^(k+1)                        ▷ Implements equation 6.2 or 6.3
8          metric^(k+1) = compute_metric(x⃗^(k), x⃗^(k+1))   ▷ Compute convergence
           metric (Eq. 6.5)
9          k ← k + 1                              ▷ Continue until convergence is reached.
10     until metric^(k+1) ≤ TOL_p or k ≥ MAX_ITER ;
11 until p == P or k ≥ MAX_ITER;
12 return x⃗^(k+1)
```

all elements of $\vec{x}^{(k+1)}$ and $\vec{x}^{(k)}$ *i.e.*, $x_i^{(k+1)} - x_i^{(k)}$ are as low as some small positive Floating-Point value u.

$$x_i^{(k+1)} - x_i^{(k)} \leq u \iff \sum_{i=1}^{n} (x_i^{(k+1)} - x_i^{(k)})^2 \leq n\,u^2 \tag{6.6}$$

$$\iff \sqrt{\sum_{i=1}^{n} (x_i^{(k+1)} - x_i^{(k)})^2} \leq u\sqrt{n} \tag{6.7}$$

This assumption provides an upper bound on the tolerance threshold equal to $u\sqrt{n}$, as explained in Equation 6.6 stated before.

This upper bound has to be computed and rounded to the precision M_p. For example, if u is set to ϵ_{M_p}, then $\texttt{TOL}_\texttt{p} = \mathrm{round}(\sqrt{n}\,\epsilon_{M_p},\ M_p)$, where $\mathrm{round}(\mathrm{X}, \mathrm{M_p})$ is a function that rounds the result of a FP number X to M_p mantissa bits following one of the standard FP rounding modes, typically the round-to-nearest rounding mode.

As the convergence metric is computed in low precision, the designer should also take the rounding errors associated with the metric computation into consideration [169, Section 4.2.5, p. 56], since the metric computation involves n multiplications, $(n-1)$ additions, plus a final square root operation (see Equation 6.5). All these computations will contribute to the final computed $\texttt{TOL}_p$ values.

To compute an estimation of the thresholds, a proof assistant called Gappa [172] is used to compute the thresholds for all possible precisions $M_p \in \{1, 2, ..., 52\}$. Listing 4 of the Appendix (D) shows the Gappa script used to compute the thresholds. In this case, it is applied to a precision M_p equal to 40 bits. Furthermore, this script is run offline only once for each precision. This means that it does not add an overhead to the iterative application itself at run-time.

Example: Consider $n = 50$, $M_p = 40$, and $u = \epsilon_{40} = 2^{-40}$. If the threshold is computed ideally, the result would be $\texttt{TOL}_\texttt{p} = 2^{-40} \times \sqrt{50} = 6.43109\ldots 10^{-12}$, which is similar to Gappa's result. However, for $M_p = 4$, the ideal result would be $\texttt{TOL}_\texttt{p} = 0.44194\ldots$, whereas using Gappa, it provides the result 0.5. The latter is more conservative and takes rounding error into account.

6.5.2.2 Details of Threshold Policy (2): relaxed thresholds with stagnation detection

This threshold policy is more "relaxed" in the sense that the chosen intermediate thresholds are as low as possible to maximize the number of iterations spent on lower precisions.

Definition of convergence stagnation Choosing very low thresholds is riskier since there are no guarantees that the convergence metric can actually go as low as the specified intermediate $\texttt{TOL}_p$ thresholds, *i.e.*, there is a chance that the convergence metric will stagnate at a specific value or oscillate around it. Hence, it is important to consider this effect when choosing to lower down the selected convergence thresholds. This process will be referred to as "stagnation detection". The stagnation behavior has been observed only for a small subset of the tested inputs, yet it is important to consider when the thresholds are selected.

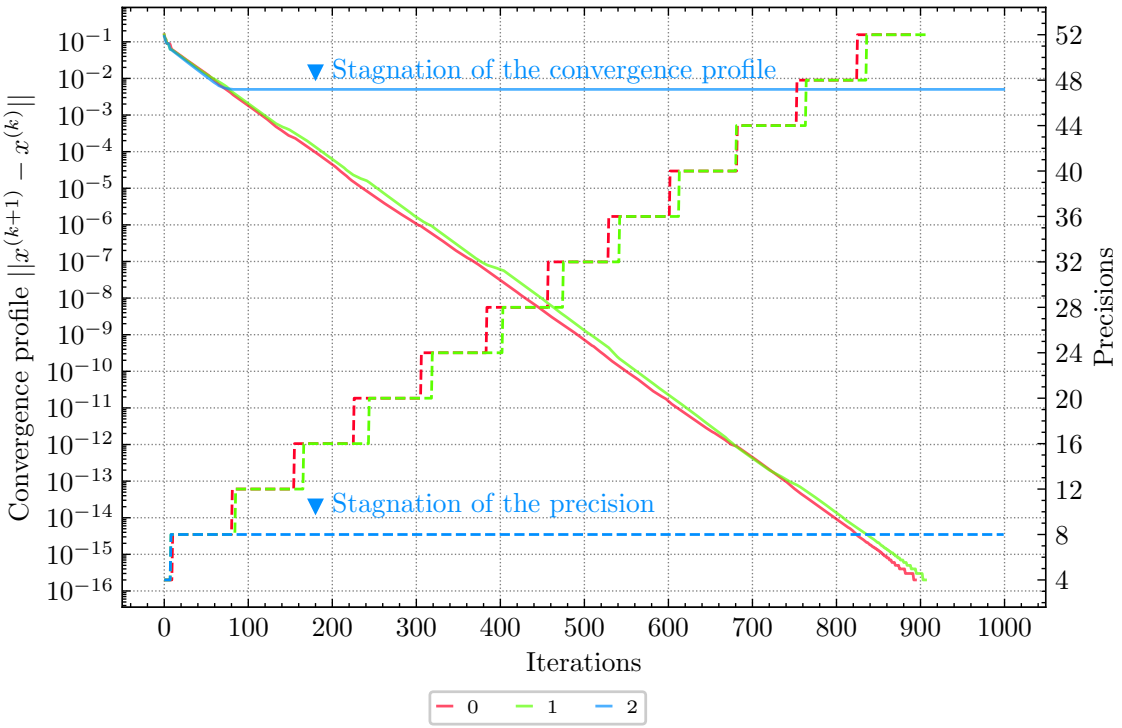


Figure 6.4: Convergence profiles (left axis, continuous lines) and their corresponding precision variation profiles (right axis, dotted lines) for three inputs (Matrix IDs 0, 1, and 2). The **Convergence Profile** *here is in terms of the distance metric* ($||\vec{x}^{(k+1)} - \vec{x}^{(k)}||$).

Example of Convergence stagnation Figure 6.4 depicts the **Convergence Profile** and the **Precision Variation Profile** for three separate inputs 0, 1, and 2 (each one has a randomly generated matrix $\mathbf{A}$ and vector $\vec{b}$). In this case, $\texttt{TOL}_p$ is set to ϵ_{M_p} for each precision M_p and a discrete subset of precisions *i.e.*, $M_p \in \{4, 8, 12, ..., 48, 52\}$ is chosen so that the total bit-width $(1+11+M_p)$ of the reduced VPT-FPU formats are multiples of 4 bits.

As shown in Fig. 6.4, input matrices 0 and 1 converge normally without problems. However, the distance metric associated with input matrix 2 oscillates between two values 0.14312744140625 (which is equal to $2^{-6.1265}$) and 0.13732910156250 (which is equal to $2^{-6.1862}$) while operating in 8-bit precision. The residual error $\vec{r}^{(k)}$ (i.e., $||\vec{b}^{(k)} - \mathbf{A}\vec{x}^{(k)}||$) also stagnates at the value 0.0000889301300049. For an 8-bit mantissa, the expected convergence threshold is normally 2^{-8}, which seems to be difficult to reach for input matrix 2. Hence the importance of being able to detect such stagnation cases.

Figure D.2 (Appendix D) shows the same phenomenon for the residual error metric.

The proposed convergence stagnation detection mechanism: To avoid the stagnation of the **Convergence Profile**, a condition to the stopping criterion should be added to detect this stagnation or oscillation phenomenon [169, Section 4.2.4, p. 56]. For that, we define the **Stagnation Maximum Iterations** (denoted `SMI`) as the maximum number of iterations for which stagnation can be tolerated. If the convergence metric stagnates at a fixed level or oscillates, the precision should be increased if higher precision is available. Otherwise, if there is no higher available

precision, then the algorithm is stopped.

A comprehensive VPT-enabled iterative algorithm is proposed (Algo. 3) to account for the convergence stagnation effect. In addition to the standard version of the algorithm (black-colored statements) and the original VPT statements (colored in blue), additional instructions (colored in red) have been added to implement stagnation detection.

From an implementation point of view, an additional integer variable should be added to keep stagnation iterations count, as well as one subtraction instruction and one comparison to check if two consecutive iterations have similar or very close distance metric ($||\vec{x}^{(k+1)} - \vec{x}^{(k)}||$) values (line 11). This overhead is negligible compared to the cost of the main iteration computations, which is confirmed by the energy consumption study.

Algorithm 3: VPT-enabled iterative algorithm with stagnation detection

Inputs : **A** : a diagonally dominant matrix of size $n \times n$,
$\vec{b}$: the right-hand side vector of size $n \times 1$,
$\vec{x}^{(0)}$: an initial guess vector of size $n \times 1$,
`MAX_ITER` : maximum number of iterations,
`TOL` : original error threshold.
$M_1 \dots M_P$: available precisions,
$\texttt{TOL}_1 \dots \texttt{TOL}_P$: intermediate error thresholds.
`SMI` : stagnation maximum iterations.

Output: $\vec{x}^{(k+1)}$: the solution of the linear system.

1 $k \leftarrow 0$, $p \leftarrow 0$
2 `vpt_enable()` ▷ Enable VPT
3 **repeat**
4 $p \leftarrow p + 1$ ▷ Increase precision index
5 `stag_counter` $\leftarrow 0$ ▷ Initialize stagnation counter
6 `vpt_set_precision(` M_p `)` ▷ Set precision to M_p
7 **repeat**
8 Compute $\vec{x}^{(k+1)}$ ▷ Implements equation 6.2 or 6.3
9 $\texttt{metric}^{(k+1)} = \texttt{compute\_metric}(\vec{x}^{(k)}, \vec{x}^{(k+1)})$ ▷ Compute convergence metric (Eq. 6.5)
10 $k \leftarrow k + 1$ ▷ Continue until convergence is reached
11 **if** $|\texttt{metric}^{(k+1)} - \texttt{metric}^{(k)}| \leq$ `DBL_MIN` [1] **then**
12 `stag_counter` $\leftarrow$ `stag_counter` $+ 1$ ▷ Increase stagnation counter
13 **end**
14 **else**
15 `stag_counter` $\leftarrow 0$ ▷ Reset stagnation counter
16 **end**
17 **until** $\texttt{metric}^{(k+1)} \leq \texttt{TOL}_p$ **or** $k \geq$ `MAX_ITER` **or** `stag_counter` $==$ `SMI` ;
18 **until** $p == P$ **or** $k \geq$ `MAX_ITER`;
19 **return** $\vec{x}^{(k+1)}$

[1] The value of the macro `DBL_MIN` is the minimum normalized positive Floating-Point number that is representable in double-precision format. It is provided by the header `float.h` from the standard C library.

6.5.2.3 Comparing different threshold policies

The different kinds of intermediate thresholds resulting from each threshold policy are studied and compared to understand how conservative or relaxed they are. Figure 6.5 depicts a comparative study of these thresholds.

For each precision $M_p \in \{1, 2, \ldots, 52\}$ three versions of intermediate thresholds $\texttt{TOL}_p$ are computed:

- **Relaxed thresholds**: by considering ϵ_{M_p} for each $M_p \in \{1, 2, \ldots, 52\}$.
- **Theoretical conservative thresholds**: computed using the formula $\sqrt{n}\,\epsilon_{M_p}$ and rounded to the nearest.
- **Conservative thresholds computed with Gappa**: these are generated using the Gappa proof assistant as explained in Section 6.5.2.1 to take rounding errors into account.

Figure 6.5 shows no meaningful difference between the two conservative threshold versions, except for very low precisions ($M_p < 6$). Thus, in practice, using one or the other does not change the results since it only affects the threshold associated with the precision $M_p = 4$.

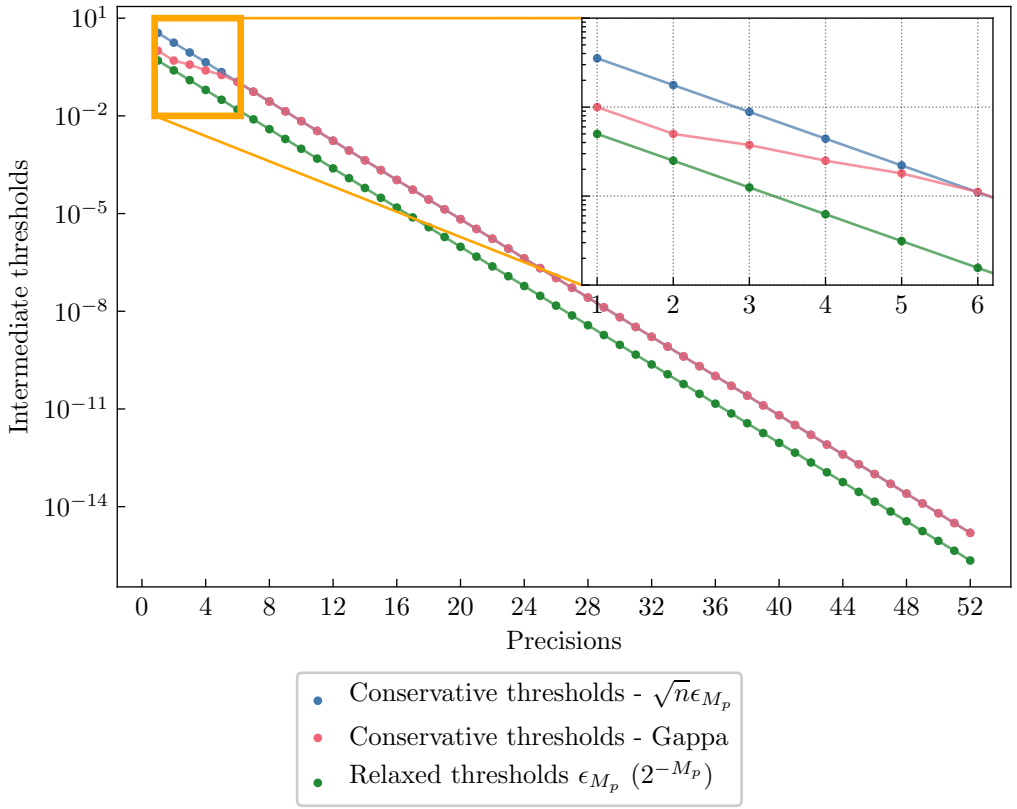


Figure 6.5: Thresholds comparison

6.6 Statistical Analysis

Section 6.5.2, demonstrated the effect of VPT by applying it to a single randomly generated input. In this section, the behavior of VPT and its gains will be evaluated across a set of 1000 randomly generated input $(\mathbf{A}, \vec{b})$.

The first objective of this section is to statistically study the effects of VPT on the generated 1000-input set. The second is to study and compare the influence of each convergence threshold policy (conservative thresholds Vs. relaxed thresholds) on the **Convergence Profile**, the **Precision Variation Profile**, the total number of iterations, the distribution of iterations across the intermediate precisions.

To perform the statistical study, each input is generated using a randomization seed ranging from 0 to 999. This section only studies the Jacobi iterative method, but the conclusions were also verified for Gauss-Seidel. These concepts can then be applied to similar iterative methods such as Newton-Raphson, Richardson, and others.

6.6.1 Software implementation aspects

In the following sections, an open-source Jacobi implementation [] written in C is considered. It takes as an input a randomly generated 50x50 diagonally dominant matrix $\mathbf{A}$, with pseudo-random values between 0 and 1. The default target tolerance threshold is set to 10^{-10} unless otherwise is specified. Gauss-Seidel is also implemented following the same structure of the original Jacobi application.

The software applications have been cross-compiled for the Proxy Kernel execution environment (a lightweight bare-metal-like OS dedicated to RISC-V-based systems) [173], using the RISC-V GCC compiler in a similar fashion to the cycle-accurate study performed earlier in Section 6.3.1.

6.6.2 Effects of VPT on the Convergence Profile and Precision Variation Profile

Considering the two threshold policies and their respective parameters presented in Section 6.5.2, five different use cases are established to be evaluated and compared:

1. **Standard baseline results.**

   In this case, the original algorithm (Algo. 1) double-precision implementation is applied to the randomly generated 1000-input set. Figure 6.6 depicts the average **Convergence Profile** (continuous blue line). The area highlighted in light-blue covers the range of possible convergence profiles obtained for the 1000 inputs. This case is represented by the blue line (Figures 6.6, 6.7 and 6.8).

2. **VPT-enabled results, with threshold policy (1), using $\sqrt{n}\,\epsilon_{M_p}$ conservative thresholds.**

   In this case, the VPT-enabled algorithm (Algo. 1) is implemented. Here, the thresholds are computed as $\mathrm{round}(\sqrt{n}\,\epsilon_{M_p}, M_p)$ as explained in Paragraph 6.5.2.1.

   The corresponding **Convergence Profile** is depicted in Figure 6.7, which overlaps with the original standard baseline **Convergence Profile**. This means that

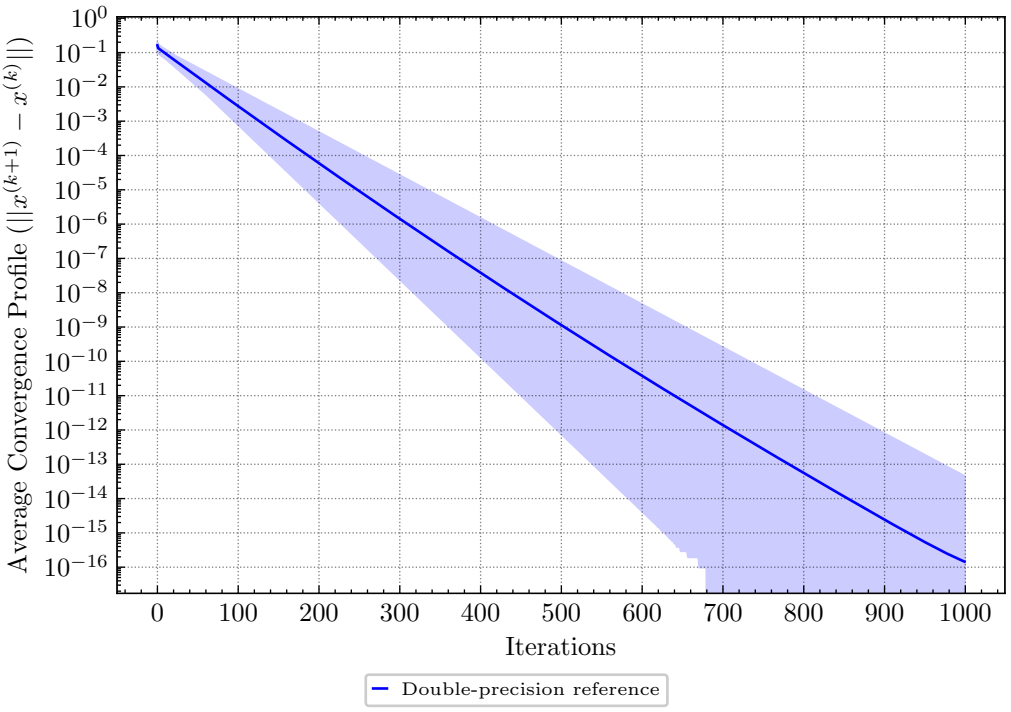


Figure 6.6: The average **Convergence Profile** *of the reference double-precision Jacobi.*

in this case, the convergence speed does not change compared to the reference even though lower precisions are used. This case is represented by the orange line (Fig. 6.7, 6.8).

3. **VPT-enabled results, with threshold policy (1), using conservative thresholds generated with Gappa.**

   In this case, the VPT-enabled algorithm (Algo. 1) is implemented. Here the thresholds computed using the Gappa proof assistant are used (Paragraph 6.5.2.1). Please note that these thresholds are similar to the latter case except for low precisions $1 \leq M_p \leq 5$. This case is represented by the green line (Fig. 6.7, 6.8).

4. **VPT-enabled, with threshold policy (2), along with ϵ_{M_p} thresholds and the Stagnation Maximum Iterations set to 2 (`SMI = 2`).**

   In this case, the VPT-enabled algorithm (Algo. 2) is implemented. The convergence thresholds used here are more relaxed, and the stagnation detection mechanism presented in Paragraph 6.5.2.2 is activated. For this experiment, stagnation is tolerated for at most two consecutive iterations before incrementing the precision. This case is represented by the red line (Fig. 6.7, 6.8).

5. **VPT-enabled, with threshold policy (2), along with ϵ_{M_p} thresholds and the Stagnation Maximum Iterations set to 4 (`SMI = 4`).**

   This case is represented by the purple line (Fig. 6.7, 6.8).

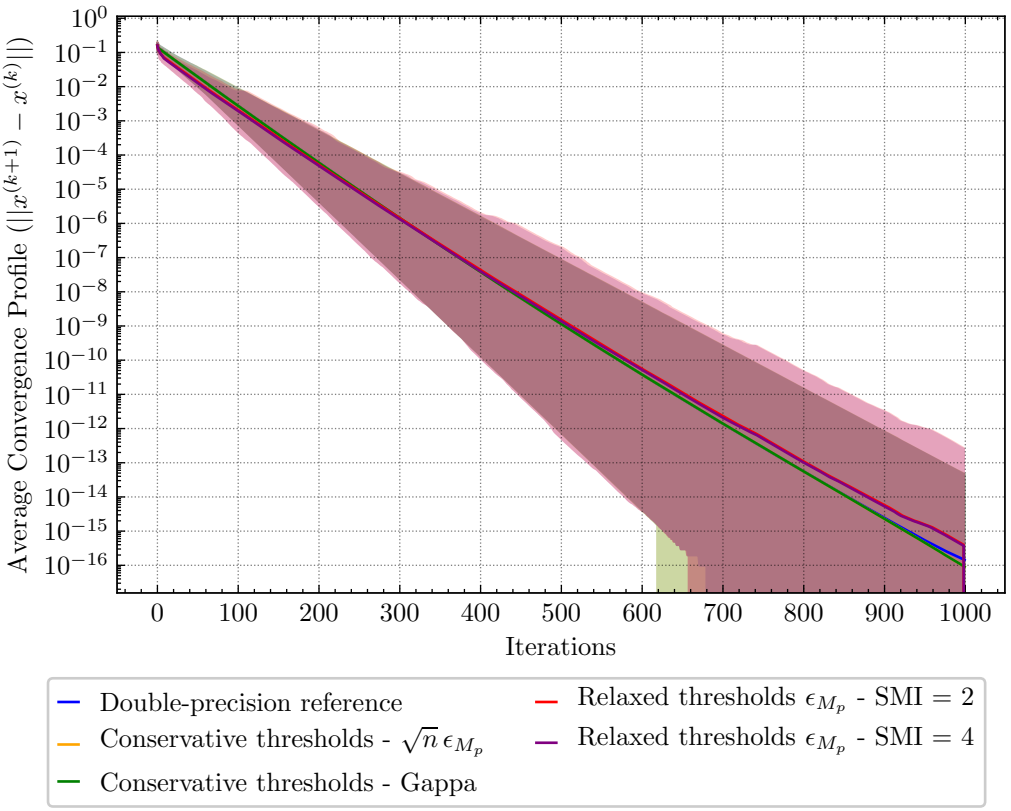


Figure 6.7: Comparison of average **Convergence Profiles** *for each use case.*

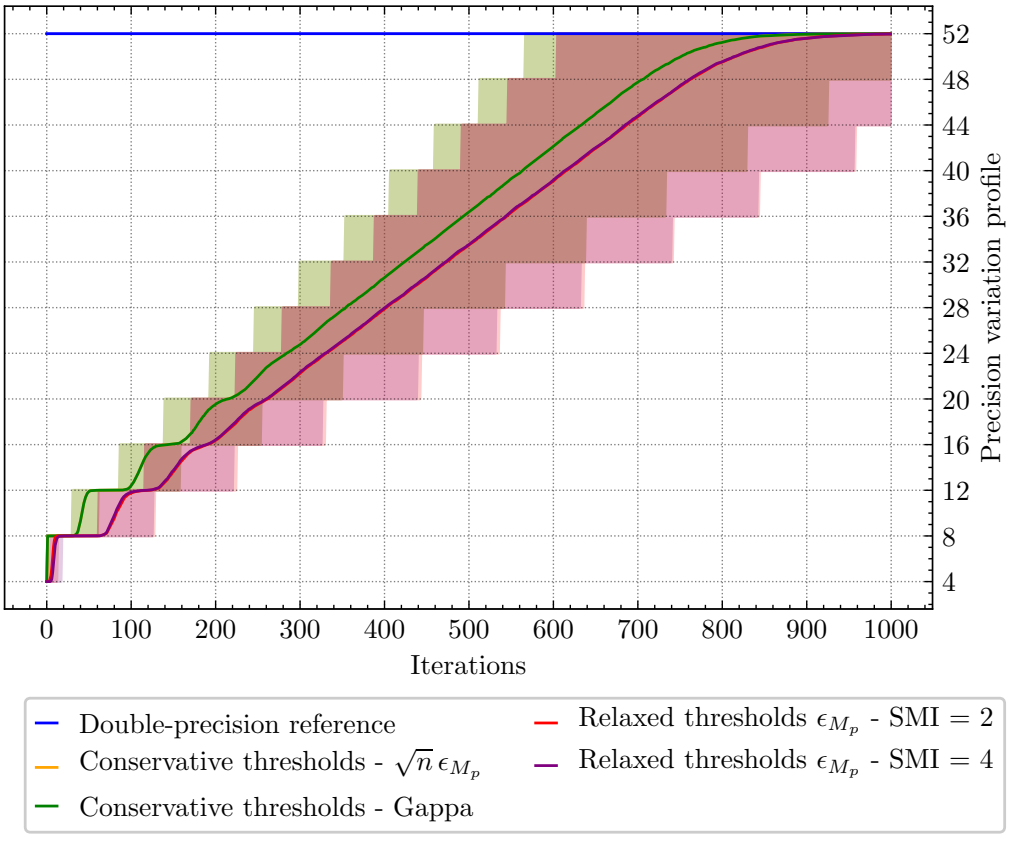


Figure 6.8: Comparison of average **Precision Variation Profiles** *for each use case.*

Figure 6.7 depicts the average CPs for each one of the five use cases. The conservative thresholds (overlapping orange and green) produce a **Convergence Profile** similar to the standard baseline (blue). The relaxed thresholds (overlapping red and purple) also result in similar convergence profiles on average, although its speed slows down compared to the baseline starting from iteration 700.

Figure 6.8 shows the **Precision Variation Profile** for the five cases. For the baseline reference, the operating precision is fixed at 52 bits, *i.e.*, double-precision format for all iterations. Both conservative threshold sets ($\sqrt{n}\,\epsilon_{M_p}$ and Gappa) produce a similar overlapping **Precision Variation Profile** (green and orange). Relaxed thresholds also produce similar overlapping profiles (red and purple) indifferent to the value of `SMI`.

6.6.3 Effects of VPT on the total number of iterations

In this paragraph, the effect of VPT on the total number of iterations is studied by analyzing each one of the five use cases stated previously. Figure 6.9 and 6.10 show the distribution of the total number of iterations (vertical axis) for each use case (horizontal axis), for two target thresholds 10^{-4} and 10^{-12}, respectively. To get these data, each version of Jacobi has been executed on the 1000 inputs. The figure represents the medians (red lines in the middle of the box plot) and the means (black dot). The blue crosses (+) represent non-significant outliers.

For higher target thresholds, and specifically 10^{-4} in this case (Fig. 6.9), there is a clear effect on iterations' number distribution: using the relaxed thresholds induces less time to converge. It is also clear that an `SMI` value of 4 achieves slightly less number of iterations compared to an `SMI` of 2. On the other hand, threshold policy (1) (conservative thresholds) shows no noticeable effect on the number of iterations.

When the target tolerance threshold is set to 10^{-12} (Fig. 6.10), no significant effect is observed. However, the relaxed thresholds lead to slightly higher iteration numbers on average compared to the double-precision reference.

There seems to be an effect on the outliers (+). Although this is not statistically significant, a worst-case and best-case HW-level comparison will be performed to evaluate to which extent this increase affects the power, energy, and execution time savings.

Figures 6.11 and 6.12 present the distribution of the overheads brought by each threshold policy with respect to the total number of iterations for the whole 1000-input set for a 10^{-04} and 10^{-12} tolerance threshold, respectively. Positive percentage values signify an increase in the number of iterations, whereas negative ones mean that there was a reduction in the number of iterations. Furthermore, the conservative policies do not alter the number of iterations much compared to the double-precision reference: they achieve a bit less than $\pm 1.5\%$. However, the relaxed policies have a slightly important but scattered effect on iterations' count. For example, for a threshold of 10^{-4} (Fig. 6.11), relaxed policies achieve a -17% to -1% reduction in the number of iterations for more than 75% of inputs. At 10^{-12} (Fig. 6.12), nearly half of the input dataset achieves a variation between -1% and $+2.5\%$.

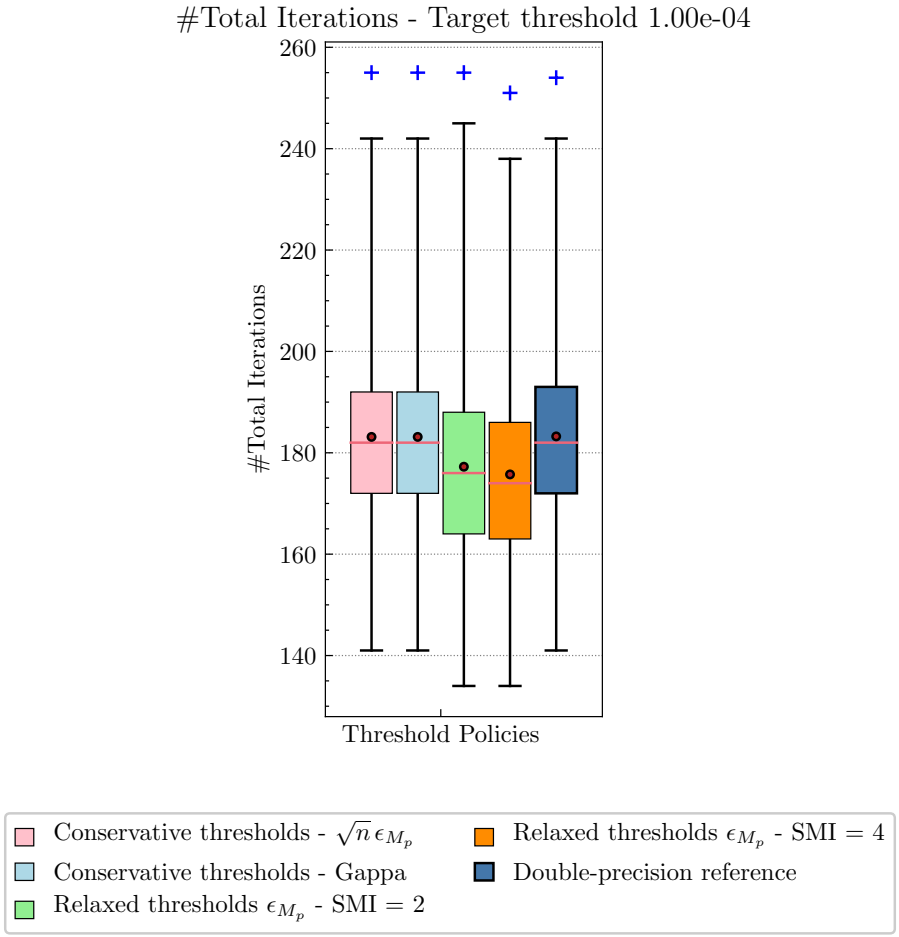


Figure 6.9: Total number of iterations for TOL $= 10^{-4}$

6.6.4 Effects of VPT on iterations' distribution

Studying the effects on the total number of iterations alone is insufficient to understand the consequences of applying VPT to such algorithms fully. Hence the necessity of also studying the distribution of iterations across the intermediate precisions. The reason for that is that, even though two use cases have the same total number of iterations, some cases will tend to over-use lower precisions more than the higher ones or vice-versa, hence leading to potentially different power/energy consumption.

Figures 6.13 and 6.14 illustrate a study similar to the one performed in the last section, this time by considering the distribution of the number of iterations per each precision for the 1000 inputs.

Figure 6.13 depicts the study when the target threshold is set to 10^{-4}. As you can see, threshold policy (2) (relaxed thresholds) tends to maximize iterations at the lower precisions, whereas threshold policy (1) (conservative thresholds) tend to maximize iterations at the higher precisions. Thus, if the two policies have the same total number of iterations, it is more likely that most of these iterations will be skewed towards lower precisions for threshold policy (2) and skewed to higher precisions for threshold policy (1). This will translate into an important difference in terms of power consumption.

Figure 6.14 provides a similar study with a target threshold of 10^{-12} to examine

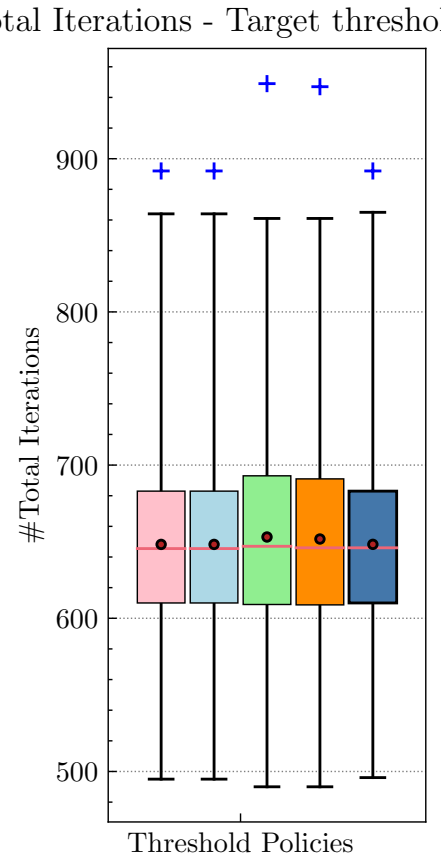


Figure 6.10: Total number of iterations for TOL $= 10^{-12}$

the previous trend in the long term. The figure shows that the earlier conclusion holds since threshold policy (1) seems to minimize iterations at lower precisions and maximize them at higher ones. For example, at $M_p = 44$, most cases already finished their execution when operating with threshold policy (2), whereas they have around 45 iterations left when working with threshold policy (1). Moreover, those 45 iterations are executed in higher precision, which means that there will be significant overhead in terms of energy compared to threshold policy (1).

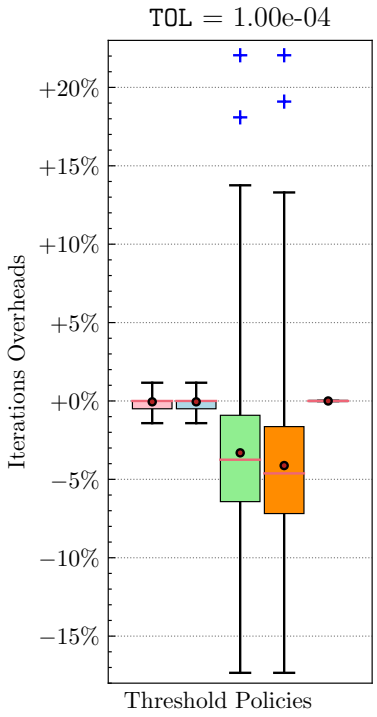


Figure 6.11: Total number of iterations' overhead w.r.t each policy for TOL $= 10^{-04}$

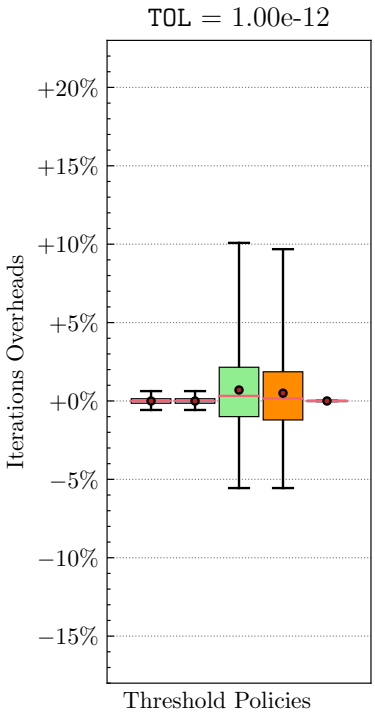


Figure 6.12: Total number of iterations' overhead w.r.t each policy for TOL $= 10^{-12}$

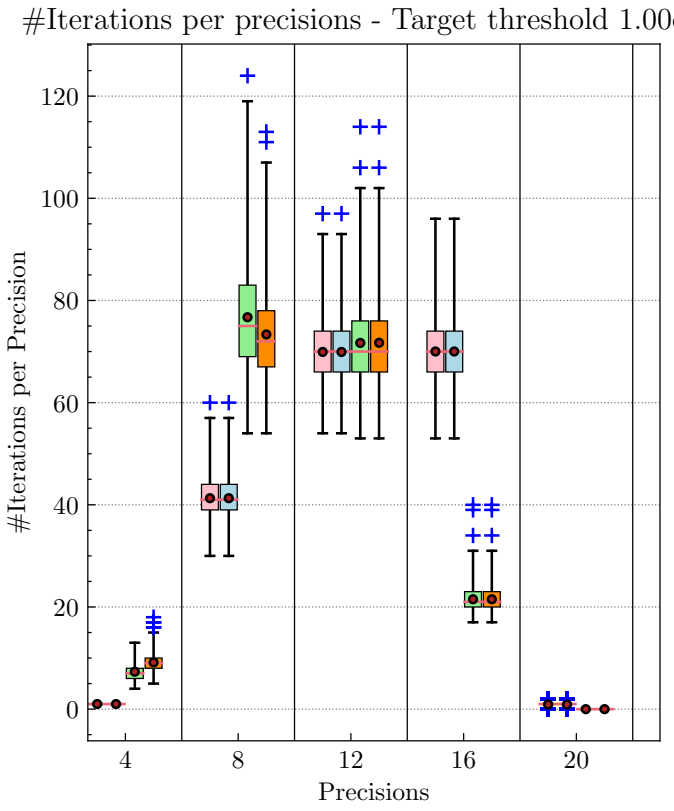


Figure 6.13: Iterations' distribution per precision for TOL $= 10^{-4}$

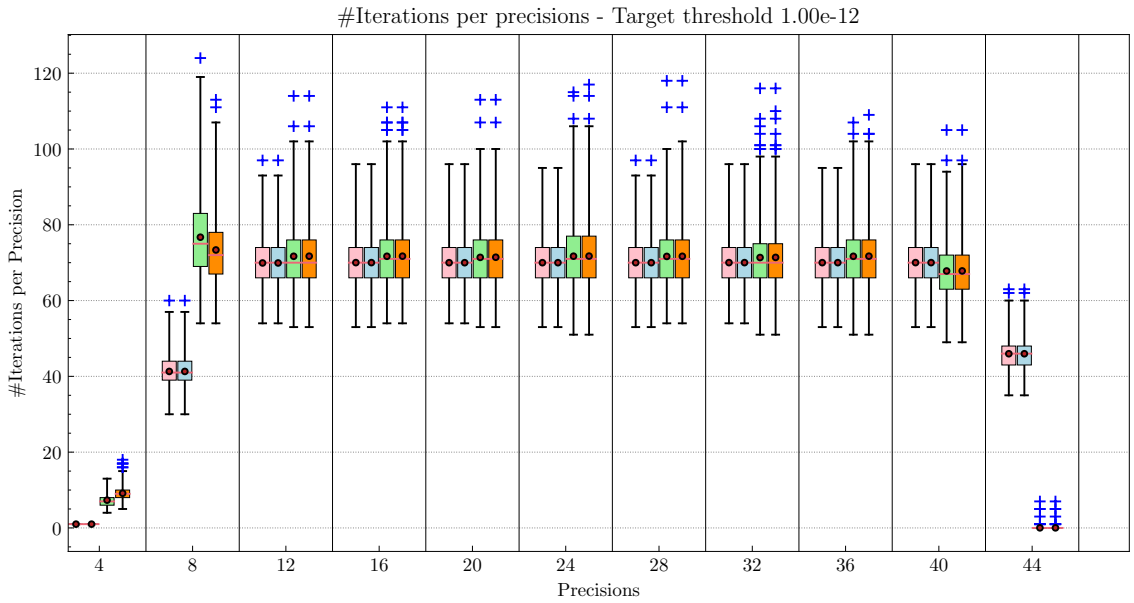


Figure 6.14: Iterations' distribution per precision for TOL $= 10^{-12}$

Conclusion The conservative thresholds provide a similar Convergence Profile to the baseline reference, whereas the relaxed thresholds tend to be slightly slower at the end of convergence. However, the relaxed thresholds tend to use less precision, which will translate to lower power consumption.

In the next section, a hardware-level power, execution time, and total computational energy evaluation will be performed for a single (typical) input and then on two cases among the 1000 random inputs.

6.7 Hardware-level Evaluation & Discussion

After assessing the effectiveness of the proposed approach from a statistical and software point of view, this section presents a hardware-level evaluation of the power, execution time, and energy savings related to computations occurring inside the FPU.

6.7.1 Hardware synthesis conditions

The VPT-FPU presented in Chapter 5 was implemented in SystemVerilog, based on an open-source parametrized FPU [101]. The hardware design is globally parametric so that the list of supported formats $\{(E_1, M_1), \ldots, (E_P, M_P)\}$ in hardware are configurable at synthesis time.

The following results were obtained in the same synthesis conditions and following the same HW evaluation flow explained previously in Section 5.5.

6.7.2 HW-level evaluation with one input and relaxed thresholds (nominal scenario)

This section will quantitatively evaluate the hardware savings and overheads introduced by VPT in terms of execution time, average power consumption, and overall energy. Here, only one randomly generated matrix $\mathbf{A}$ and vector $\vec{b}$ is considered (the same one presented in Section 6.5.2).

6.7.2.1 Evaluation methodology

Table 6.1 depicts the evaluation results for each error threshold $\texttt{TOL} \in \{10^{-4}, 10^{-6}, 10^{-8}, 10^{-10}, 10^{-12}\}$. Experiments are performed for each {SW configuration, HW configuration} pair, defined as follows:

Definition 6.7.1. A **SW configuration** specifies which algorithm is implemented and what variable types are used in the source code implementation.

Definition 6.7.2. A **HW configuration** specifies the set of supported reduced Floating-Point formats $\{(E_0, M_0), (E_1, M_1), ..., (E_P, M_P)\}$ and to which approximate datapath they belong (D_D) or (D_F), as explained in Section 5.2 and depicted in Figure 5.1.

For each tolerance threshold, post-synthesis results are provided for three different SW configurations of each iterative algorithm:

1. *Ref double*: an implementation of the original algorithm (Algo. 1) in double-precision format. FP computations are executed in `binary64` (11, 52) format within the (C) data-path.

2. VPT `double`: an implementation of the VPT-enabled algorithm (Algo. 2) in double-precision format. FP computations are executed within the (D_D) data-path of the VPT-FPU, which is populated with the supported precisions (4, 8, 12, ...). All corresponding exponents are 11-bit.

3. VPT `float`: an implementation of the VPT-enabled algorithm (Algo. 2) in single-precision format. FP computations are executed within the (D_F) data-path of the VPT-FPU, which is populated with the supported precisions (3, 7, 11, ...). All corresponding exponents are 8-bit. This SW configuration takes advantage of both VPT and the classic VTO technique.

Seven different HW configurations: VPT_A, VPT_B, ... VPT_G are provided (Table 6.1).

Baseline results The baseline performance results are obtained by executing the reference SW configuration *Ref double* on the reference `RV64FD` hardware FPU. This experiment is repeated for each tolerance threshold. All upcoming results will be normalized w.r.t this one.

VPT results For each tolerance threshold and each {SW configuration, HW configuration} pair, a gate-level simulation is performed. The absolute values of execution times, average power, and the dissipated energy spent on FP computations are reported in Table 6.1 for Jacobi and Gauss-Seidel. The results were obtained by applying the algorithm on a single input matrix $\mathbf{A}$ and vector $\vec{b}$ auto-generated with a randomization seed equal to zero. Column-wise normalized values (%) are reported by dividing the estimated values by the reference ones for each target error threshold, *i.e.,* there is a reference for each target threshold column.

6.7.2.2 Discussion

VPT double* Vs. *Ref double Table 6.1 shows that the VPT-enabled implementations always achieve better performance with no accuracy loss compared to the reference. For example, the *VPT double* SW configuration of Jacobi achieves power consumption savings ranging from 59.71% up to 74.31%, execution time savings ranging from 39.17% up to 59.77%, and energy savings between 77.21% and 87.89%. Meanwhile, the same SW configuration for Gauss-Seidel application achieves power consumption savings ranging from 50.83% up to 78.18%, execution time savings ranging from 31.70% up to 51.70%, and energy savings between 66.40% and 89.46%.

Target error thresholds		10^{-4}		10^{-6}		10^{-8}	10^{-10}	10^{-12}
VPT SW Configurations		VPT float	VPT double	VPT float	VPT double	VPT double	VPT double	VPT double
M_p precisions	D_D ($E_p = 11$)	-	4, 8, 12, 16	-	4, 8, 12, 16, 20	4, 8, 12, 16, 20, 24, 28	4, 8, 12, 16, 20, 24, 28, 32, 36	4, 8, 12, 16, 20, 24, 28, 32, 36, 40
	D_F ($E_p = 8$)	3, 7, 11, 15	-	3, 7, 11, 15, 19	-	-	-	-
VPT-FPU Config. name		VPT_A	VPT_B	VPT_C	VPT_D	VPT_E	VPT_F	VPT_G
Area overhead		**1.19**×	**1.27**×	**1.30**×	**1.39**×	**1.78**×	**2.30**×	**2.63**×
Jacobi Results								
Average Power (W)	Switching	2.44E-04	3.01E-04	2.95E-04	4.24E-04	4.61E-04	4.36E-04	4.77E-04
	Internal	4.97E-04	6.14E-04	5.83E-04	8.03E-04	8.74E-04	8.72E-04	9.39E-04
	Leakage	2.90E-05	3.10E-05	3.20E-05	3.40E-05	4.30E-05	5.60E-05	6.40E-05
	Total	7.70E-04	9.46E-04	9.10E-04	1.26E-03	1.38E-03	1.36E-03	1.48E-03
Total power savings (%)		**79.09%**	**74.31%**	**75.3%**	**65.77%**	**62.55%**	**62.9%**	**59.71%**
Execution time (ps)		5.96E+09	5.35E+09	1.00E+10	1.00E+10	1.56E+10	1.32E+10	1.65E+10
Execution time savings (%)		**47.49%**	**52.85%**	**45.72%**	**45.72%**	**39.17%**	**59.77%**	**58.70%**
Energy (pJ)		4.59E+06	5.06E+06	9.12E+06	1.26E+07	2.15E+07	1.80E+07	2.45E+07
Energy savings (%)		**89.02%**	**87.89%**	**86.59%**	**81.46%**	**77.21%**	**85.07%**	**83.35%**
Gauss-Seidel Results								
Average Power (W)	Switching	2.11E-04	2.20E-04	2.83E-04	3.26E-04	3.74E-04	4.50E-04	5.89E-04
	Internal	4.40E-04	4.93E-04	5.57E-04	6.59E-04	7.47E-04	8.81E-04	1.09E-03
	Leakage	2.90E-05	3.10E-05	3.20E-05	3.40E-05	4.30E-05	5.60E-05	6.40E-05
	Total	6.80E-04	7.44E-04	8.72E-04	1.02E-03	1.16E-03	1.39E-03	1.74E-03
Total power savings (%)		**80.06%**	**78.18%**	**74.97%**	**70.75%**	**66.92%**	**60.79%**	**50.83%**
Execution time (ps)		1.68E+08	1.68E+08	2.54E+08	2.83E+08	4.10E+08	5.38E+08	7.12E+08
Execution time savings (%)		**51.61%**	**51.70%**	**51.19%**	**45.76%**	**41.00%**	**38.03%**	**31.70%**
Energy (pJ)		1.14E+05	1.25E+05	2.22E+05	2.88E+05	4.77E+05	7.46E+05	1.24E+06
Energy savings (%)		**90.35%**	**89.46%**	**87.78%**	**84.15%**	**80.49%**	**75.72%**	**66.40%**

Table 6.1: Post-synthesis and gate-level simulation results for Jacobi and Gauss-Seidel applied to 1 input with relaxed thresholds (ϵ_{M_p}).

VPT float* Vs. *Ref double Analyzing the two first columns (10^{-4} and 10^{-6}) reveals the importance of traditional **Variable Type Optimization** (VTO) techniques in **Floating-Point**. For example, in the case of Jacobi, when `TOL` equals 10^{-4}, the *VPT double* SW configuration saves 74.31% of the original total power and 87.89% of the energy consumed by the *Ref double* configuration. However, if VPT is applied along with VTO, *i.e.*, if FP variables are migrated from `double` to `float` in the VPT-enabled algorithm, the power savings are further optimized up to 79.09%, and the energy savings up to 89.02%. Similarly, at 10^{-6} error threshold, optimizing variable types increases energy savings brought by VPT from 81.46% up to 86.59%. This observation also holds for Gauss-Seidel.

Conclusion To sum up, starting from an already optimized standard software version (using VTO) instead of a double-precision, one can achieve high power and energy savings. Using the VPT approach allows to refine and enhance VTO savings one step further. Even though the additional VTO gains shown may seem limited to a few percentage points in terms of computations, starting from an already optimized version (using VTO) guarantees subsequent memory footprint savings too. However, this is only feasible for 10^{-4} and 10^{-6} target error thresholds which are reachable using single-precision.

6.7.3 Worst case / Best case HW-level evaluation

With everything said in Section 6.6 in mind, it is important to perform an empirical HW-level evaluation. Only edge cases where VPT affects either negatively or positively the energy consumption of the algorithm are considered since it would take months to perform the HW-level study for all the 1000 inputs. Moreover, for the sake of concision, only the conservative thresholds generated with Gappa in the case of threshold policy (1) are considered. And for threshold policy (2), only the case where `SMI` is set to 4 is evaluated.

The edge cases have been chosen from the 1000-input dataset based on the software simulation presented in Section 6.6. They correspond to the input couples $(\mathbf{A}, \vec{b})$ that produce the lowest and the biggest iterations' overhead for each threshold in Figures 6.11 and 6.12. Table 6.2 lists the nomenclature that will be used in the remaining of this chapter. For example, BC-04 refers to the input couple $(\mathbf{A}, \vec{b})$ that represents the best case for a 10^{-4} tolerance threshold.

To ensure a fair comparison from a hardware standing point, the following experiments are all run on top of a VPT-FPU hardware configuration that will be referred to as **VPT_H**. This configuration supports the following precisions $\{4, 8, ..., 48\}$ within the $(\mathrm{D_D})$ data-path (with an exponent bit-width maintained at 11). Post-synthesis power simulations are performed with the same synthesis conditions as explained in Section 5.5.

Threshold Policy	1.00E-04		1.00E-12	
	Best Case	Worst Case	Best Case	Worst Case
Conservative thresholds - Gappa	BC-04	WC-04	BC-12	WC-12
Relaxed thresholds - SMI = 4	BR-04	WR-04	BR-12	WR-12

Table 6.2: Worst cases' and best cases' nomenclature per tolerance threshold and threshold generation policy - Jacobi.

Baseline HW-level results for all edge cases Table 6.4 shows the baseline average power, execution time, and energy consumption results for the standard double-precision version of Jacobi applied to each of the selected cases and executed on top of the reference baseline `RV64FD` architecture. This data will constitute the baseline against which all the following studies will be compared.

Overhead of VPT-FPU when operating in full precision Table 6.5 depicts HW results when executing the application on the precise part of the VPT-FPU for each selected edge case. This evaluates the overhead brought by the static energy dissipated in the approximate part of the VPT-FPU circuitry when execution mode is fully precise.

As shown in the table, execution time is not affected. Only power increases due to static power (leakage) dissipated in the non-active approximate parts, leading to a 3.72% (input BC-12) up to 3.85% (input WR-12) energy increase, which is negligible compared to the savings exposed in the following sections.

Gains using the conservative thresholds Table 6.3 depicts the HW-level gains for the conservative threshold policy (1). In addition, the table presents the results of the best case and worst case inputs at 10^{-4} and 10^{-12} target thresholds.

As depicted in the table, there is not much difference between best-case and worst-case scenarios in terms of computational energy, *i.e.*, 84.06% (best case for 10^{-4}) Vs. 83.43% (worst case for 10^{-4}) then 63.13% (best case for 10^{-12}) Vs. 62.33% (worst case for 10^{-12}). This can be explained by the fact that using VPT threshold policy (1) does not affect the number of iterations executed.

Upon a closer look to Figures 6.11 and 6.12, we conclude that this is due to the negligible effect on the number of iterations when using conservative threshold policies.

Target Error Thresholds		1.00E-04				1.00E-12			
Input		**BC-04**	**WC-04**	**BR-04**	**WR-04**	**BC-12**	**WC-12**	**BR-12**	**WR-12**
VPT SW Configs	Type	Double	Double	Double	Double	Double	Double	Double	Double
Average Power (W)	Switching	1.428E-03	1.433E-03	1.431E-03	1.434E-03	1.426E-03	1.426E-03	1.433E-03	1.420E-03
	Internal	2.219E-03	2.226E-03	2.222E-03	2.227E-03	2.213E-03	2.214E-03	2.225E-03	2.206E-03
	Leakage	2.600E-05	2.600E-05	2.600E-05	2.600E-05	2.600E-05	2.600E-05	2.600E-05	2.600E-05
	Total	3.673E-03	3.685E-03	3.679E-03	3.687E-03	3.665E-03	3.666E-03	3.684E-03	3.652E-03
Execution Time (ps)		1.099E+10	1.180E+10	1.007E+10	1.157E+10	4.033E+10	3.206E+10	3.547E+10	4.056E+10
Energy (pJ)		4.038E+07	4.350E+07	3.703E+07	4.266E+07	1.478E+08	1.175E+08	1.306E+08	1.481E+08

Table 6.4: Power, execution time, and energy consumption of the standard double-precision version of Jacobi executed on the `RV64FD` *reference FPU*

Target Error Thresholds		1.00E-04				1.00E-12			
Input		**BC-04**	**WC-04**	**BR-04**	**WR-04**	**BC-12**	**WC-12**	**BR-12**	**WR-12**
VPT SW Configs	Type	Double	Double	Double	Double	Double	Double	Double	Double
Average Power (W)	Switching	1.399E-03	1.403E-03	1.401E-03	1.404E-03	1.395E-03	1.396E-03	1.403E-03	1.390E-03
	Internal	2.331E-03	2.338E-03	2.335E-03	2.339E-03	2.325E-03	2.326E-03	2.337E-03	2.318E-03
	Leakage	8.300E-05	8.300E-05	8.300E-05	8.300E-05	8.300E-05	8.300E-05	8.300E-05	8.300E-05
	Total	3.813E-03	3.824E-03	3.819E-03	3.826E-03	3.803E-03	3.805E-03	3.823E-03	3.791E-03
Total power overhead		**3.81 %**	**3.77 %**	**3.81 %**	**3.77 %**	**3.77 %**	**3.79 %**	**3.77 %**	**3.81 %**
Execution Time (ps)		1.099E+10	1.180E+10	1.007E+10	1.157E+10	4.033E+10	3.206E+10	3.547E+10	4.056E+10
Execution time overhead		**0.00 %**	**0.00 %**	**0.00 %**	**0.00 %**	**0.00 %**	**0.00 %**	**0.00 %**	**0.00 %**
Energy (pJ)		4.192E+07	4.514E+07	3.845E+07	4.428E+07	1.533E+08	1.220E+08	1.356E+08	1.538E+08
Energy overhead		**3.81 %**	**3.77 %**	**3.83 %**	**3.80 %**	**3.72 %**	**3.83 %**	**3.83 %**	**3.85 %**

Table 6.5: Power, execution time, and energy consumption ***overheads*** *of the standard double-precision version of Jacobi executed on the VPT_H configuration of the VPT-FPU*

Target Error Thresholds		1.00E-04		1.00E-12	
Input		Best case (BC-04)	Worst case (WC-04)	Best case (BC-12)	Worst case (WC-12)
VPT SW Configs	Type	VPT double	VPT double	VPT double	VPT double
Average Power (W)	Switching	3.410E-04	3.510E-04	7.140E-04	7.230E-04
	Internal	7.890E-04	8.050E-04	1.312E-03	1.325E-03
	Leakage	8.100E-05	8.100E-05	8.200E-05	8.200E-05
	Total	1.211E-03	1.237E-03	2.108E-03	2.130E-03
Total power savings		**67.03 %**	**66.43 %**	**42.48 %**	**41.90 %**
Execution Time (ps)		5.315E+09	5.828E+09	2.586E+10	2.079E+10
Execution time savings		**51.64 %**	**50.61 %**	**35.88 %**	**35.15 %**
Energy (pJ)		6.436E+06	7.209E+06	5.449E+07	4.426E+07
Energy savings		**84.06 %**	**83.43 %**	**63.13 %**	**62.33 %**

*Table 6.3: Power, execution time, and energy consumption **gains** of VPT-enabled version of Jacobi (with theoretical conservative thresholds) executed on the VPT_H configuration of the VPT-FPU*

Target Error Thresholds		1.00E-04		1.00E-12	
Input		Best case (BR-04)	Worst case (WR-04)	Best case (BR-12)	Worst case (WR-12)
VPT SW Configs	Type	VPT double	VPT double	VPT double	VPT double
Average Power (W)	Switching	2.900E-04	2.940E-04	6.420E-04	6.480E-04
	Internal	7.080E-04	7.130E-04	1.207E-03	1.215E-03
	Leakage	8.100E-05	8.100E-05	8.200E-05	8.200E-05
	Total	1.079E-03	1.088E-03	1.931E-03	1.945E-03
Total power savings		**70.67 %**	**70.49 %**	**47.58 %**	**46.74 %**
Execution Time (ps)		4.048E+09	6.687E+09	2.099E+10	2.803E+10
Execution time savings		**59.80 %**	**42.20 %**	**40.82 %**	**30.89 %**
Energy (pJ)		4.368E+06	7.275E+06	4.052E+07	5.450E+07
Energy savings		**88.20 %**	**82.95 %**	**68.97 %**	**63.20 %**

*Table 6.6: Power, execution time, and energy consumption **gains** of VPT-enabled version of Jacobi (with relaxed thresholds and SMI = 4) executed on the VPT_H configuration of the VPT-FPU*

Gains using the relaxed thresholds Results for this experiment are depicted in Table 6.6. For a target threshold of 10^{-4}, this threshold policy achieves energy savings between 82.95%(worst case) up to 88.20% (best case), mainly due to 42.20% up to 59.80% savings in terms of execution time. Similarly, for a target of 10^{-12}, the energy gain stands between 63.20% (worst case) and 68.97% (best case). These percentages are slightly more interesting than in threshold policy (1), especially in the best case.

Please remember that, in this situation, the VPT relaxed threshold policy (2) has a more substantial impact on the number of iterations, as shown in Figures 6.11 and 6.12. For example, input BR-12 sees a decrease in the number of iterations from 612 to 578 (-5.5% variation), translating to a 68.97% energy gain. On the other hand, WR-12 sees an increase from 700 to 763 iterations ($+9.0\%$ increase) which translates to 63.20% total energy gain, which is still an important achievement despite the extra iterations' overhead.

6.7.4 Circuit area results

Table 6.1 reports the total cell area overhead for each HW configuration w.r.t the baseline standard `RV64FD` FPU.

The area overhead depends on the additional supported reduced precision formats. Overheads can range from $1.19\times$ (using the VPT_A architecture) up to $2.63\times$ (using the VPT_G architecture).

When VTO is also applied along with VPT, (*i.e.*, when the *VPT float* SW configuration is considered), this technique can attenuate the area overhead. For example, overhead can be reduced from $1.27\times$ (obtained for VPT_B) down to $1.19\times$ (obtained for VPT_A) for a 10^{-4} threshold.

These ratios also depend on the number of intermediate pipeline registers inserted in each approximate datapath, configurable at design time. The area overhead is the only significant disadvantage of the proposed architecture.

6.7.5 Limitations

These results demonstrate that the VPT-enabled software implementation is a one-size-fits-all solution, *i.e.*, using the same software implementation along with our proposed VPT-FPU, the designer can drastically reduce consumed resources by using only the needed precision instead of an over-designed solution such as standard FPUs. The price to pay is in terms of circuit area overhead. Designers can also use standard VTO techniques jointly to enhance the energy savings of our technique and reduce its area overhead.

To choose between the two threshold policies for a given problem, the user/designer should use representative datasets and especially evaluate the memory-related aspects, especially for threshold policy (2). Furthermore, the memory-related aspects should be studied further to evaluate whether the cost of the potential added iterations (hence more load and store operations) is lower than the gain associated with computation optimization.

The proposed methodology is application-dependent, *i.e.*, the designer should transform the algorithm manually, ensure convergence after modification, and select adequate intermediate tolerance thresholds depending on the convergence metric used in the application.

6.8 Conclusion

This chapter proposes a new method based on the VPT-FPU architecture, presented in Chapter 5, that enables designers to dynamically tune FP computations' precision automatically at run-time called **Variable Precision in Time** (VPT). In spite of the circuit area overhead, the proposed approach simplifies the integration of variable precision in existing software workloads at any level of the software stack (OS, RTOS, or application-level): it only requires light-weight software support and solely relies on traditional assembly instructions, without the need for a specialized compiler or custom instructions.

The technique was applied here on the Jacobi and the Gauss-Seidel iterative methods taking full advantage of the suggested VPT-FPU. For each algorithm two threshold policies were proposed: a conservative policy (1) and a relaxed policy (2).

The last two sections presented a statistical study that explored the effects of each VPT threshold policy on many aspects of the application: impact on the total number of iterations, impact on the iterations' distribution across different precisions, impact on HW-level estimations such as execution time, power and the overall energy consumption.

The implementations demonstrate up to 70.67% power consumption saving, up to 59.80% execution time saving, and up to 88.20% total energy saving w.r.t the reference double-precision implementation, with no QoR loss.

To conclude, the threshold policy (1) is a conservative approach that brings some predictability and safety along with all the run-time variable precision benefits. On the other hand, threshold policy (2) is a relaxed approach that further optimizes power and energy consumption. However, it tends to alter the total number of iterations, sometimes favorably and sometimes negatively. However, even when the total number of iterations is increased, there is still a very interesting energy gain. Thus, threshold policy (1) is the safest solution, and threshold policy (2) represents a bit risky but an optimized solution.

7

Conclusion and Perspectives

7.1 Thesis Contributions

The landscape of modern software applications is dominated by computionally intensive and data-centric applications such as **Machine Learning** (ML), **Neural Networks** (NNs), **Computer Vision** (CV), ...*etc.* Such applications are known for their built-in resilience against errors. In other words, even when part of their intermediate computations is performed only approximately, the final result can still be considered acceptable.

As these modern applications feature an inherent ability to tolerate precision loss, new computing paradigms have emerged: **Approximate Computing** (AxC) and **Transprecision Computing** (TC). The idea behind these paradigms is to trade-off precision against some kind of improvement. Potential improvements include reducing power consumption, execution time, circuit area, boosting performance, throughput, and energy efficiency.

AxC and TC can be applied to computations as well as memory/storage. In this thesis we chose to only focus on the computational part, and particularly on **Floating-Point** computations. This choice was motivated by the fact that full-precision **Floating-Point Units** (FPUs) can be a source of extensive hardware overhead in CPU-based architectures. In addition, for this class of applications, the standard IEEE **Floating-Point** (FP) formats were shown to be often over-designed. Hence, hardware FPU architectures need reduced and variable precision capabilities to trade-off energy efficiency or performance against accuracy when the latter is not needed.

Several tools and techniques that allow the **Design Space Exploration** of such trade-offs have been proposed in the context of TC. The objective is to help designers to find the most optimized FPU architecture adequate for a given application.

However, existing techniques require developers to rewrite part or all of their software stacks (applications, libraries, operating systems, ...), which is often infeasible, complex or at least a very time-consuming development effort. In addition to their intrusiveness, most TC tools are only designed for simulation, and they do not provide corresponding hardware architectures that achieve real-world energy gains. Furthermore, most of these works target static precision reduction, *i.e.*, precision is fixed during the execution.

This thesis proposes many contributions that alleviate some **State of the Art** limitations:

- A minimally-intrusive approach that leverages FP approximations for power, execution time, and overall energy consumption reduction.

  The approach, described in Chapter 3, introduces approximations at the assembly instruction execution/interpretation level. This allows approximating virtually all kinds of executable binaries (bare-metal applications, single-/multi-threaded user applications, OS/RTOS, *etc.*) without the need for a specialized compiler or custom instructions. The methodology supports **Arbitrary Reduced Precision** (ARP) (*i.e.*, non-standard reduced-precision FP formats) as well as **Variable Precision in Time** (VPT) (*i.e.*, the ability for applications to tune their operating precision at run-time).

- A software implementation based on the well known QEMU dynamic binary translator -called AxQEMU- was developed.

It simulates the impact of such approximations on high-level applications' **Quality of Result** (QoR). The tool was presented in Chapter 4.

- A hardware multi-precision FPU, referred to as VPT-FPU, was designed to support VPT in hardware.

  This was achieved mainly by instantiating multiple ARP in hardware, hence achieving real-world energy efficiency gains. The architecture was synthesized on a 28nm FD-SOI ASIC technology node (*C.f.*, Chapter 5).

- Self-adaptive precision was demonstrated through the particular case of Iterative Methods (Chapter 6)

  In this contribution we proposed a joint HW/SW approach that leverages the VPT-FPU architecture for aggressive energy savings when applied to stationary iterative methods. In our case, we targeted particularly Jacobi and Gauss-Seidel algorithms. For each algorithm, two modified versions are proposed : a conservative version and a relaxed one. Both algorithms are analyzed and compared statistically to understand the effects of VPT on iterative applications. The implementations demonstrated up to 70.67% power consumption saving, up to 59.80% execution time saving, and up to 88.20% total energy saving w.r.t the reference double-precision implementation, and with no accuracy loss.

In this thesis, the proposed approaches and their implementations were evaluated jointly in software and hardware against a set of error-resilient benchmarks with a particular emphasis on iterative methods. The experiments show how fine-grained energy/accuracy trade-offs can be made thanks to ARP and VPT, leading to drastic computational energy savings.

7.2 Future Perspectives

The contributions brought by this thesis allowed to solve several limitations in the **State of the Art** and paved the way towards self-adaptive reduced variable FP precision in CPU-based architectures. However, there are many areas than can be improved:

In the short term many technical and implementation aspects could be improved. For instance, it should be interesting to have a comprehensive system-level evaluation of the energy gains brought by our architecture since in this work only the FPU is considered. For that the HW-level evaluation flow should take also into consideration memory and system parameters (*e.g.*, processor parameters, cache parameters, interconnect, ...)

Furthermore, the accuracy of the HW-level estimations could be refined by performing a physical implementation (Place and Route). Of course, more realistic evaluations can be obtained by fabricating the circuit.

The area of the current VPT-FPU circuit can be optimized by combining some common blocks such as the reduction/extension/rounding blocks. Further optimization could be achieved by supporting only reduced precision in hardware and precise computations in software (using SoftFloat). However, this process will have to be automated, since it is more intrusive to source code and might need some development efforts by the programmer. This would allow the VPT-FPU to get rid of the single-precision and double-precision FMAs.

On the SW simulation side, enabling AxQEMU to simulate applications written in other languages (*e.g.*, Python, Java, ...) would allow a more seamless integration with a wide set of SoA applications especially in the ML/NNs/CV domains.

In the long term We recommended to continue exploring new emerging FP computation paradigms such as redundant arithmetic [90] for ajustable precision. This seems an interesting solution to implement hardware variable precisions operators in the FPU with potentially less area overhead. It is also important to extend the proposed methodology to memory optimization. This would bring a further fine-grained improvement of memory footprint.

In this thesis, all implementations targeted the RISC-V ISA. However, it is possible to port these works to other architectures such as ARM, MIPS, *etc.*. Exploring the effect of VPT on vector and multi-core architectures would be of interest for modern computationally intensive applications.

Another aspect that could be improved is the automation of the **Design Space Exploration** flow presented in Chapter 4. This can be done by developing search algorithms capable of selecting the most adequate FPU architecture for a given {application, input dataset} pair. For instance, the traditional delta debugging algorithm [134] can be adapted to achieve that.

It would be of great interest also to generalize VPT to other algorithms. For example, the Gradient Descent algorithm is yet another iterative method that is often used in ML/NN training. This would be interesting for resource-constrained computing platforms that perform online NN (re-)training and data processing on the edge.

To generalize VPT to other classes of applications, the programmer needs to define when precision is tuned and which precision to select. The tools we developed allow the programmer to do that manually. However, automating this task could be achieved using ML to predict the needed precision. It is worth mentioning that the use of ML for TC seems to be the new trend in this area [174]. A typical use case is the following: for a given application, a **Neural Network** could be trained offline using many inputs and several precisions to be able to predict which precision is needed to achieve a given QoR. At run-time, the trained ML algorithm could be used to periodically update the computation precision.

Finally, the ideas brought by this thesis can be transposed to other computing architectures such as GPUs and FPGAs. With the current advanced topics such as adaptive and reconfigurable computing, the issues related to circuit area can be resolved. For example, an FPGA-based system containing an FPU could reconfigure its precision, at runtime, depending on the needs of the application being executed.

7.3 Scientific Communications

Peer-reviewed International Journals

1. [160] N. Ait Said, M. Benabdenbi, and K. Morin-Allory, "**Self-adaptive Runtime Variable Floating-Point Precision for Iterative Algorithms : a Joint HW/SW Approach**."

   To appear soon in the MDPI Electronics Journal. Special Session on *Approximate Computing: Design, Acceleration, Validation and Testing of Circuits, Architectures and Algorithms in Future Systems*.

2. [135] N. Ait Said, M. Benabdenbi, and K. Morin-Allory, "**Arbitrary Reduced Precision for Fine-grained Accuracy and Energy Trade-offs**."

   In *Microelectronics Reliability*, 2021, 120, 114099.

   DOI: https://doi.org/10.1016/j.microrel.2021.114099.

Peer-reviewed International Conferences

1. [159] N. Ait Said, M. Benabdenbi, and K. Morin-Allory, "**FPU Reduced Variable Precision in Time: Application to the Jacobi Iterative Method.**"

   In 2021 *IEEE Computer Society Annual Symposium on VLSI* (*ISVLSI*) 2021, pp. 170-175,

   DOI: https://www.doi.org/10.1109/ISVLSI51109.2021.00040.

   Finalist for the Best Paper Award.

2. [161] N. Ait Said, M. Benabdenbi, and K. Morin-Allory, "**FPU Bit-Width Optimization for Approximate Computing: A Non-Intrusive Approach.**"

   In 2020 15th *Design Technology of Integrated Systems in Nanoscale Era* (*DTIS*), 2020.

   DOI: https://www.doi.org/10.1109/10.1109/DTIS48698.2020.9080931

Bibliography

[1] Moore, Gordon E., "Cramming more components onto integrated circuits, Reprinted from Electronics, volume 38, number 8, April 19, 1965, pp.114 ff." *IEEE Solid-State Circuits Society Newsletter*, vol. 11, no. 3, pp. 33–35, 2006.

[2] Horowitz, Mark, "1.1 Computing's energy problem (and what we can do about it)," in *2014 IEEE International Solid-State Circuits Conference Digest of Technical Papers (ISSCC)*, 2014, pp. 10–14.

[3] "IBM unveils 2-nanometer chip technology for faster computing," May 2021. [Online]. Available: https://www.reuters.com/technology/ibm-unveils-2-nanometer-chip-technology-faster-computing-2021-05-06/

[4] "TSMC's 7nm, 5nm, and 3nm "are just numbers… it doesn't matter what the number is"." [Online]. Available: https://www.pcgamesn.com/amd/tsmc-7nm-5nm-and-3nm-are-just-numbers

[5] Writer, Staff, "TSMC details its future 5nm and 3nm manufacturing processes—here's what it means for Apple silicon." [Online]. Available: https://www.macworld.com/article/234529/tsmc-details-its-future-5nm-and-3nm-manufacturing-processesheres-what-it-means-for-apple-silicon.html

[6] Dennard, R.H. and Gaensslen, F.H. and Yu, Hwa-Nien and Rideout, V.L. and Bassous, E. and LeBlanc, A.R., "Design of ion-implanted MOSFET's with very small physical dimensions," *IEEE Journal of Solid-State Circuits*, vol. 9, no. 5, pp. 256–268, 1974.

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

[8] Q. Xu and T. Mytkowicz and N. S. Kim, "Approximate Computing: A Survey," *IEEE Design Test*, vol. 33, no. 1, pp. 8–22, Feb. 2016.

[9] Schuman, Catherine D and Potok, Thomas E and Patton, Robert M and Birdwell, J Douglas and Dean, Mark E and Rose, Garrett S and Plank, James S, "A survey of neuromorphic computing and neural networks in hardware," *arXiv preprint arXiv:1705.06963*, 2017.

[10] Mittal, Sparsh, "A Survey of Techniques for Approximate Computing," *ACM Comput. Surv.*, vol. 48, no. 4, Mar. 2016. [Online]. Available: https://doi.org/10.1145/2893356

[11] Venkataramani, Swagath and Chakradhar, Srimat T and Roy, Kaushik and Raghunathan, Anand, "Approximate computing and the quest for computing efficiency," in *2015 52nd ACM/EDAC/IEEE Design Automation Conference (DAC)*. IEEE, 2015, pp. 1–6.

[12] Moreau, T. and Miguel, J. San and Wyse, M. and Bornholt, J. and Alaghi, A. and Ceze, L. and Jerger, N. Enright and Sampson, A., "A Taxonomy of General Purpose Approximate Computing Techniques," *IEEE Embedded Systems Letters*, vol. 10, no. 1, pp. 2–5, Mar. 2018.

[13] Shafique, M. and Hafiz, R. and Rehman, S. and El-Harouni, W. and Henkel, J., "Invited: Cross-layer approximate computing: From logic to architectures," in *2016 53nd ACM/EDAC/IEEE Design Automation Conference (DAC)*, Jun. 2016, pp. 1–6.

[14] Aponte-Moreno, A. and Moncada, A. and Restrepo-Calle, F. and Pedraza, C., "A review of approximate computing techniques towards fault mitigation in HW/SW systems," in *2018 IEEE 19th Latin-American Test Symposium (LATS)*, Mar. 2018, pp. 1–6.

[15] Gaines, Brian R, "Stochastic computing," in *Proceedings of the April 18-20, 1967, spring joint computer conference*, 1967, pp. 149–156.

[16] Rejimon, Thara and Bhanja, Sanjukta, "Scalable probabilistic computing models using Bayesian networks," in *48th Midwest Symposium on Circuits and Systems, 2005.* IEEE, 2005, pp. 712–715.

[17] Gruska, Jozef and others, *Quantum computing*. McGraw-Hill London, 1999, vol. 2005.

[18] National Academies of Sciences, Engineering, *Quantum Computing: Progress and Prospects*, Dec. 2018. [Online]. Available: https://www.nap.edu/catalog/25196/quantum-computing-progress-and-prospects

[19] Zadeh, Lotfi A, "Soft computing and fuzzy logic," in *Fuzzy Sets, Fuzzy Logic, and Fuzzy Systems: Selected Papers by Lotfi a Zadeh*. World Scientific, 1996, pp. 796–804.

[20] Sharad, Mrigank and Fan, Deliang and Aitken, Kyle and Roy, Kaushik, "Energy-efficient non-Boolean computing with spin neurons and resistive memory," *IEEE Transactions on Nanotechnology*, vol. 13, no. 1, pp. 23–34, 2013.

[21] Bhanja, Sanjukta and Karunaratne, D. K. and Panchumarthy, Ravi and Rajaram, Srinath and Sarkar, Sudeep, "Non-Boolean computing with nanomagnets for computer vision applications," *Nature Nanotechnology*, vol. 11, no. 2, pp. 177–183, Feb. 2016. [Online]. Available: https://www.nature.com/articles/nnano.2015.245

[22] Nakada, Takashi and Nakamura, Hiroshi, "Normally-off computing," in *Normally-Off Computing*. Springer, 2017, pp. 57–63.

[23] Verma, Naveen and Jia, Hongyang and Valavi, Hossein and Tang, Yinqi and Ozatay, Murat and Chen, Lung-Yen and Zhang, Bonan and Deaville, Peter, "In-memory computing: Advances and prospects," *IEEE Solid-State Circuits Magazine*, vol. 11, no. 3, pp. 43–55, 2019.

[24] Ielmini, Daniele and Wong, H-S Philip, "In-memory computing with resistive switching devices," *Nature Electronics*, vol. 1, no. 6, pp. 333–343, 2018.

[25] GMT, Posted 31 Mar 2021 |15:00, "Waiting for Quantum Computing? Try Probabilistic Computing - IEEE Spectrum." [Online]. Available: https://spectrum.ieee.org/computing/hardware/waiting-for-quantum-computing-try-probabilistic-computing

[26] Ndour, Geneviève and Jost, Tiago Trevisan and Molnos, Anca and Durand, Yves and Tisserand, Arnaud, "Evaluation of Approximate Operators Case Study: Sobel Filter Application Executed on an Approximate RISC-V Platform," in *Proceedings of the 18th International Conference on Embedded Computer Systems: Architectures, Modeling, and Simulation*, ser. SAMOS '18. New York, NY, USA: ACM, 2018, pp. 146–149, event-place: Pythagorion, Greece. [Online]. Available: http://doi.acm.org/10.1145/3229631.3229646

[27] OpenPOWER Foundation, "OpenPOWER Summit Europe 2018: Transprecision Computing," Nov. 2018. [Online]. Available: https://www.youtube.com/watch?v=JqbBIO9nCC4

[28] "IEEE Standard for Floating-Point Arithmetic," *IEEE Std 754-2008*, pp. 1–70, Aug. 2008.

[29] Jost, Tiago T and Bocco, Andrea and Durand, Yves and Fabre, Christian and de Dinechin, Florent and Cohen, Albert, "Variable Precision Floating-Point RISC-V Coprocessor Evaluation using Lightweight Software and Compiler Support," in *CARRV 2019 - Third Workshop on Computer Architecture Research with RISC-V*, Phoenix, AZ, United States, Jun. 2019, pp. 1–6. [Online]. Available: https://hal.inria.fr/hal-02161621

[30] "Closed Problems in Approximate Computing." [Online]. Available: https://www.cs.cornell.edu/~asampson/blog/closedproblems.html

[31] Esmaeilzadeh, Hadi and Sampson, Adrian and Ceze, Luis and Burger, Doug, "Architecture support for disciplined approximate programming," in *International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2012.

[32] Gong, Yu and Liu, Bo and Ge, Wei and Shi, Longxing, "ARA: Cross-Layer approximate computing framework based reconfigurable architecture for CNNs," *Microelectronics Journal*, vol. 87, pp. 33–44, May 2019. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0026269218307055

[33] Wali, Imran, "Circuit and system fault tolerance techniques," Theses, Université Montpellier, Mar. 2016. [Online]. Available: https://tel.archives-ouvertes.fr/tel-01807927

[34] V. Mrazek and R. Hrbacek and Z. Vasicek and L. Sekanina, "EvoApprox8b: Library of approximate adders and multipliers for circuit design and benchmarking of approximation methods," in *Design, Automation Test in Europe Conference Exhibition (DATE), 2017*, Mar. 2017, pp. 258–261.

[35] Strollo, A.G.M. and Napoli, E. and De Caro, D., "New clock-gating techniques for low-power flip-flops," in *ISLPED'00: Proceedings of the 2000 International Symposium on Low Power Electronics and Design (Cat. No.00TH8514)*, 2000, pp. 114–119.

[36] Burd, T.D. and Pering, T.A. and Stratakos, A.J. and Brodersen, R.W., "A dynamic voltage scaled microprocessor system," *IEEE Journal of Solid-State Circuits*, vol. 35, no. 11, pp. 1571–1580, 2000.

[37] Das, S. and Roberts, D. and Seokwoo Lee and Pant, S. and Blaauw, D. and Austin, T. and Flautner, K. and Mudge, T., "A self-tuning DVS processor using delay-error detection and correction," *IEEE Journal of Solid-State Circuits*, vol. 41, no. 4, pp. 792–804, 2006.

[38] Li Shang and Li-Shiuan Peh and Jha, N.K., "Dynamic voltage scaling with links for power optimization of interconnection networks," in *The Ninth International Symposium on High-Performance Computer Architecture, 2003. HPCA-9 2003. Proceedings.*, 2003, pp. 91–102.

[39] Kedem, Zvi M. and Mooney, Vincent J. and Muntimadugu, Kirthi Krishna and Palem, Krishna V., "An approach to energy-error tradeoffs in approximate ripple carry adders," in *IEEE/ACM International Symposium on Low Power Electronics and Design*, 2011, pp. 211–216.

[40] Oi, Hitoshi, "Case study: Effectiveness of dynamic frequency scaling on server workload," in *2014 International Symposium on Integrated Circuits (ISIC)*, 2014, pp. 332–335.

[41] Chafi, Poorya Raoofi and Moradi, Morteza and Rahmanikia, Navid and Noori, Hamid, "A platform for dynamic thermal management of FPGA-based soft-core processors via Dynamic Frequency Scaling," in *2015 23rd Iranian Conference on Electrical Engineering*, 2015, pp. 1093–1097.

[42] Ulya R. Karpuzcu and Ismail Akturk and Nam Sung Kim, "Accordion: Toward soft Near-Threshold Voltage Computing," in *International Symposium on High-Performance Computer Architecture (HPCA)*, 2014.

[43] Gautschi, Michael and Schiavone, Pasquale Davide and Traber, Andreas and Loi, Igor and Pullini, Antonio and Rossi, Davide and Flamand, Eric and Gürkaynak, Frank K. and Benini, Luca, "Near-Threshold RISC-V Core With DSP Extensions for Scalable IoT Endpoint Devices," *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, vol. 25, no. 10, pp. 2700–2713, 2017.

[44] Shin, Doochul and Gupta, Sandeep K., "Approximate logic synthesis for error tolerant applications," in *2010 Design, Automation Test in Europe Conference Exhibition (DATE 2010)*, 2010, pp. 957–960.

[45] Palem, Krishna V. and Chakrapani, Lakshmi N.B. and Kedem, Zvi M. and Lingamneni, Avinash and Muntimadugu, Kirthi Krishna, "Sustaining Moore's Law in Embedded Computing through Probabilistic and Approximate Design: Retrospects and Prospects," in *Proceedings of the 2009 International Conference on Compilers, Architecture, and Synthesis for Embedded Systems*, ser. CASES '09. New York, NY, USA: Association for Computing Machinery, 2009, p. 1–10. [Online]. Available: https://doi.org/10.1145/1629395.1629397

[46] Miao, Jin and He, Ku and Gerstlauer, Andreas and Orshansky, Michael, "Modeling and Synthesis of Quality-energy Optimal Approximate Adders," in *IEEE–ACM International Conference on Computer-Aided Design (ICCAD)*, 2012.

[47] Gupta, Vaibhav and Mohapatra, Debabrata and Park, Sang Phill and Raghunathan, Anand and Roy, Kaushik, "IMPACT: Imprecise Adders for Low-Power Approximate Computing," in *Proceedings of the 17th IEEE/ACM International Symposium on Low-Power Electronics and Design*, ser. ISLPED '11. IEEE Press, 2011, p. 409–414.

[48] Sekanina, Lukas, *Evolvable Components: From Theory to Hardware Implementations*, ser. Natural Computing Series. Berlin Heidelberg: Springer-Verlag, 2004. [Online]. Available: https://www.springer.com/gp/book/9783540403777

[49] Sekanina, Lukáš and Han, Jie, "Approximate Computing with Approximate Circuits: Methodologies and Applications," p. 85.

[50] Mrazek, Vojtech and Hanif, Muhammad Abdullah and Vasicek, Zdenek and Sekanina, Lukas and Shafique, Muhammad, "autoAx: An Automatic Design Space Exploration and Circuit Building Methodology utilizing Libraries of Approximate Components," *arXiv:1902.10807 [cs]*, Feb. 2019, arXiv: 1902.10807. [Online]. Available: http://arxiv.org/abs/1902.10807

[51] Ceska, Milan and Matyas, Jiri and Mrazek, Vojtech and Sekanina, Lukas and Vasicek, Zdenek and Vojnar, Tomas, "Approximating complex arithmetic circuits with formal error guarantees: 32-bit multipliers accomplished," in *2017 IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*. Irvine, CA: IEEE, Nov. 2017, pp. 416–423. [Online]. Available: http://ieeexplore.ieee.org/document/8203807/

[52] Sekanina, Lukas and Mrazek, Vojtech and Vasicek, Zdenek, "Design Space Exploration for Approximate Implementations of Arithmetic Data Path Primitives," in *2018 25th IEEE International Conference on Electronics, Circuits and Systems (ICECS)*. Bordeaux: IEEE, Dec. 2018, pp. 377–380. [Online]. Available: https://ieeexplore.ieee.org/document/8618048/

[53] Andreas Heinig and Vincent John Mooney and Florian Schmoll and Peter Marwedel and Krishna V. Palem and Michael Engel, "Classification-Based Improvement of Application Robustness and Quality of Service in Probabilistic Computer Systems," in *International Conference on Architecture of Computing Systems (ARCS)*, 2012.

[54] Darshan D. Thaker and Diana Franklin and John Oliver and Susmit Biswas and Derek Lockhart and Tzvetan S. Metodi and Frederic T. Chong, "Characterization of Error-Tolerant Applications when Protecting Control Data," in *IEEE International Symposium on Workload Characterization (IISWC)*, 2006.

[55] Anna Thomas and Karthik Pattabiraman, "LLFI: An Intermediate Code Level Fault Injector For Soft Computing Applications," in *Workshop on Silicon Errors in Logic: System Effects (SELSE)*, 2013.

[56] Sampson, Adrian and Dietl, Werner and Fortuna, Emily and Gnanapragasam, Danushen and Ceze, Luis and Grossman, Dan, "EnerJ: approximate data types for safe and general low-power computation," in *ACM Conference on Programming Language Design and Implementation (PLDI)*, 2011.

[57] A. Sampson and W. Dietl and E. Fortuna and D. Gnanapragasam and L. Ceze and D. Grossman, "EnerJ: Approximate Data Types for Safe and General Low-Power Computation — Full Proofs," University of Washington, Tech. Rep. UW-CSE-10-12-01, 2011.

[58] Sampson, Adrian and Baixo, André and Ransford, Benjamin and Moreau, Thierry and Yip, Joshua and Ceze, Luis and Oskin, Mark, "ACCEPT: A programmer-guided compiler framework for practical approximate computing," *University of Washington Technical Report UW-CSE-15-01*, vol. 1, 2015.

[59] Michael Carbin and Sasa Misailovic and Martin C. Rinard, "Verifying Quantitative Reliability for Programs that Execute on Unreliable Hardware," in *ACM Conference on Object-Oriented Programming, Systems, Languages, and Applications* (*OOPSLA*), 2013.

[60] Mishra, Asit K and Barik, Rajkishore and Paul, Somnath, "iACT: A Software-Hardware Framework for Understanding the Scope of Approximate Computing," p. 6.

[61] James Bornholt and Todd Mytkowicz and Kathryn S. McKinley, "Uncertain<T>: A First-Order Type for Uncertain Data," in *International Conference on Architectural Support for Programming Languages and Operating Systems* (*ASPLOS*), 2014.

[62] Baek, Woongki and Chilimbi, Trishul M, "Green: A Framework for Supporting Energy-Conscious Programming using Controlled Approximation," p. 12.

[63] Jason Ansel and Cy P. Chan and Yee Lok Wong and Marek Olszewski and Qin Zhao and Alan Edelman and Saman P. Amarasinghe, "PetaBricks: a language and compiler for algorithmic choice," in *ACM Conference on Programming Language Design and Implementation* (*PLDI*), 2009.

[64] Sidiroglou-Douskos, Stelios and Misailovic, Sasa and Hoffmann, Henry and Rinard, Martin C., "Managing performance vs. accuracy trade-offs with loop perforation," in *ACM SIGSOFT Symposium on the Foundations of Software Engineering* (*FSE*), 2011.

[65] Byna, Surendra and Meng, Jiayuan and Raghunathan, Anand and Chakradhar, Srimat and Cadambi, Srihari, "Best-Effort Semantic Document Search on GPUs," in *Proceedings of the 3rd Workshop on General-Purpose Computation on Graphics Processing Units*, ser. GPGPU-3. New York, NY, USA: Association for Computing Machinery, 2010, p. 86–93. [Online]. Available: https://doi.org/10.1145/1735688.1735705

[66] Meng, Jiayuan and Raghunathan, Anand and Chakradhar, Srimat and Byna, Surendra, "Exploiting the forgiving nature of applications for scalable parallel execution," in *IEEE International Parallel & Distributed Processing Symposium*, 2010.

[67] Goiri, Inigo and Bianchini, Ricardo and Nagarakatte, Santosh and Nguyen, Thu D., "ApproxHadoop: Bringing Approximations to MapReduce Frameworks," in *Proceedings of the Twentieth International Conference on Architectural Support for Programming Languages and Operating Systems*, ser. ASPLOS '15. New York, NY, USA: Association for Computing Machinery, 2015, p. 383–397. [Online]. Available: https://doi.org/10.1145/2694344.2694351

[68] Samadi, Mehrzad and Lee, Janghaeng and Jamshidi, D. Anoushe and Hormati, Amir and Mahlke, Scott, "SAGE: Self-tuning Approximation for Graphics Engines," in *IEEE/ACM International Symposium on Microarchitecture* (*MICRO*), 2013.

[69] Jiayuan Meng and Chakradhar, Srimat and Raghunathan, Anand, "Best-effort parallel execution framework for Recognition and mining applications," in *IEEE International Parallel & Distributed Processing Symposium*, 2009.

[70] Alvarez, Carlos and Corbal, Jesus and Valero, Mateo, "Fuzzy Memoization for Floating-Point Multimedia Applications," *IEEE Transactions on Computers*, vol. 54, no. 7, 2005.

[71] Suresh, Arjun and Swamy, Bharath Narasimha and Rohou, Erven and Seznec, André, "Intercepting Functions for Memoization: A Case Study Using Transcendental Functions," *ACM Transactions on Architecture and Code Optimization*, vol. 12, no. 2, p. 23, Jul. 2015. [Online]. Available: https://hal.inria.fr/hal-01178085

[72] Esmaeilzadeh, Hadi and Sampson, Adrian and Ceze, Luis and Burger, Doug, "Neural acceleration for general-purpose approximate programs," in *IEEE/ACM International Symposium on Microarchitecture* (*MICRO*), 2012.

[73] Thierry Moreau and Mark Wyse and Jacob Nelson and Adrian Sampson and Hadi Esmaeilzadeh and Luis Ceze and Mark Oskin, "SNNAP: Approximate Computing on Programmable SoCs via Neural Acceleration," in *International Symposium on High-Performance Computer Architecture* (*HPCA*), 2015.

[74] Atoofian, Ehsan, "Approximate Cache in GPGPUs," *ACM Trans. Embed. Comput. Syst.*, vol. 19, no. 5, Sep. 2020. [Online]. Available: https://doi.org/10.1145/3407904

[75] Chang, F. and Wu-chang Feng and Kang Li, "Approximate caches for packet classification," in *IEEE INFOCOM 2004*, vol. 4, 2004, pp. 2196–2207 vol.4.

[76] Jerger, Natalie Enright and Miguel, Joshua San, "Approximate Cache Architectures," in *Approximate Circuits: Methodologies and CAD*, S. Reda and M. Shafique, Eds. Cham: Springer International Publishing, 2019, pp. 399–416. [Online]. Available: https://doi.org/10.1007/978-3-319-99322-5_20

[77] San Miguel, Joshua and Albericio, Jorge and Moshovos, Andreas and Enright Jerger, Natalie, "Doppelganger: A Cache for Approximate Computing," 12 2015.

[78] Ik Joon Chang and Mohapatra, D. and Roy, K., "A Priority-Based 6T/8T Hybrid SRAM Architecture for Aggressive Voltage Scaling in Video Applications," *IEEE Transactions on Circuits and Systems for Video Technology*, vol. 21, no. 2, pp. 101–112, 2011.

[79] Kumar, Animesh and Rabaey, Jan and Ramchandran, Kannan, "SRAM supply voltage scaling: A reliability perspective," in *International Symposium on Quality Electronic Design (ISQED)*, 2009.

[80] Shoushtari, Majid and BanaiyanMofrad, Abbas and Dutt, Nikil, "Exploiting Partially-Forgetful Memories for Approximate Computing," *IEEE Embedded Systems Letters*, vol. 7, no. 1, pp. 19–22, Mar. 2015.

[81] Song Liu and Karthik Pattabiraman and Thomas Moscibroda and Benjamin G. Zorn, "Flikker: Saving Refresh-Power in Mobile Devices through Critical Data Partitioning," in *International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2011.

[82] Jan Lucas and Mauricio Alvarez Mesa and Michael Andersch and Ben Juurlink, "Sparkk: Quality-Scalable Approximate Storage in DRAM," in *The Memory Forum*, 2014.

[83] Ranjan, Ashish and Venkataramani, Swagath and Fong, Xuanyao and Roy, Kaushik and Raghunathan, Anand, "Approximate Storage for Energy Efficient Spintronic Memories," in *Design Automation Conference (DAC)*, 2015.

[84] Sampson, Adrian and Nelson, Jacob and Strauss, Karin and Ceze, Luis, "Approximate Storage in Solid-state Memories," in *IEEE/ACM International Symposium on Microarchitecture (MICRO)*, 2013.

[85] Boyapati, Rahul and Huang, Jiayi and Majumder, Pritam and Yum, Ki Hwan and Kim, Eun Jung, "APPROX-NoC: A data approximation framework for Network-on-Chip architectures," in *2017 ACM/IEEE 44th Annual International Symposium on Computer Architecture (ISCA)*, 2017, pp. 666–677.

[86] Lee, Jaechul and Killian, Cédric and Beux, Sébastien Le and Chillet, Daniel, "Approximate Nanophotonic Interconnects," in *Proceedings of the 13th IEEE/ACM International Symposium on Networks-on-Chip*, ser. NOCS '19. New York, NY, USA: Association for Computing Machinery, 2019. [Online]. Available: https://doi.org/10.1145/3313231.3352365

[87] Shih-Lien Lu, "Speeding up processing with approximation circuits," *Computer*, vol. 37, no. 3, pp. 67–73, 2004.

[88] Verma, Ajay K. and Brisk, Philip and Ienne, Paolo, "Variable Latency Speculative Addition: A New Paradigm for Arithmetic Circuit Design," in *2008 Design, Automation and Test in Europe*, 2008, pp. 1250–1255.

[89] Behrooz Parhami, *Computer arithmetic - algorithms and hardware designs*. Oxford University Press, 2000.

[90] Ali Skaf and Mona Ezzadeen and Mounir Benabdenbi and Laurent Fesquet, "On-Line Adjustable Precision Computing," in *14th International Conference on Design & Technology of Integrated Systems In Nanoscale Era, DTIS 2019, Mykonos, Greece, April 16-18, 2019*. IEEE, 2019, pp. 1–5. [Online]. Available: https://doi.org/10.1109/DTIS.2019.8735067

[91] "Accelerating Fixed-Point Design for MB-OFDM UWB Systems." [Online]. Available: https://www.design-reuse.com/articles/9559/accelerating-fixed-point-design-for-mb-ofdm-uwb-systems.html

[92] Arslan, T. and Horrocks, D.H., "A genetic algorithm for the design of finite word length arbitrary response cascaded IIR digital filters," in *First International Conference on Genetic Algorithms in Engineering Systems: Innovations and Applications*, 1995, pp. 276–281.

[93] Cantin, M.-A. and Savaria, Y. and Prodanos, D. and Lavoie, P., "An automatic word length determination method," in *ISCAS 2001. The 2001 IEEE International Symposium on Circuits and Systems (Cat. No.01CH37196)*, vol. 5, 2001, pp. 53–56 vol. 5.

[94] Chan, S. C. and Tsui, K. M., "Wordlength Optimization of Linear Time-Invariant Systems With Multiple Outputs Using Geometric Programming," *IEEE Transactions on Circuits and Systems I: Regular Papers*, vol. 54, no. 4, pp. 845–854, 2007.

[95] Ha, Van-Phu and Yuki, Tomofumi and Sentieys, Olivier, "Towards Generic and Scalable Word-Length Optimization," in *2020 Design, Automation Test in Europe Conference Exhibition (DATE)*, 2020, pp. 1668–1673.

[96] Köster, Urs and Webb, Tristan J. and Wang, Xin and Nassar, Marcel and Bansal, Arjun K. and Constable, William H. and Elibol, Ouguz H. and Gray, Scott and Hall, Stewart and Hornof, Luke and Khosrowshahi, Amir and Kloss, Carey and Pai, Ruby J. and Rao, Naveen, "Flexpoint: An Adaptive Numerical Format for Efficient Training of Deep Neural Networks," in *Proceedings of the 31st International Conference on Neural Information Processing Systems*, ser. NIPS'17. Red Hook, NY, USA: Curran Associates Inc., 2017, p. 1740–1750.

[97] Chung, Eric and Fowers, Jeremy and Ovtcharov, Kalin and Papamichael, Michael and Caulfield, Adrian and Massengill, Todd and Liu, Ming and Lo, Daniel and Alkalay, Shlomi and Haselman, Michael and Abeydeera, Maleen and Adams, Logan and Angepat, Hari and Boehn, Christian and Chiou, Derek and Firestein, Oren and Forin, Alessandro and Gatlin, Kang Su and Ghandi, Mahdi and Heil, Stephen and Holohan, Kyle and El Husseini, Ahmad and Juhasz, Tamas and Kagi, Kara and Kovvuri, Ratna K. and Lanka, Sitaram and van Megen, Friedel and Mukhortov, Dima and Patel, Prerak and Perez, Brandon and Rapsang, Amanda and Reinhardt, Steven and Rouhani, Bita and Sapek, Adam and Seera, Raja and Shekar, Sangeetha and Sridharan, Balaji and Weisz, Gabriel and Woods, Lisa and Yi Xiao, Phillip and Zhang, Dan and Zhao, Ritchie and Burger, Doug, "Serving DNNs in Real Time at Datacenter Scale with Project Brainwave," *IEEE Micro*, vol. 38, no. 2, pp. 8–20, 2018.

[98] Jouppi, Norman P. and Young, Cliff and Patil, Nishant and Patterson, David and Agrawal, Gaurav and Bajwa, Raminder and Bates, Sarah and Bhatia, Suresh and Boden, Nan and Borchers, Al and Boyle, Rick and Cantin, Pierre-luc and Chao, Clifford and Clark, Chris and Coriell, Jeremy and Daley, Mike and Dau, Matt and Dean, Jeffrey and Gelb, Ben and Ghaemmaghami, Tara Vazir and Gottipati, Rajendra and Gulland, William and Hagmann, Robert and Ho, C. Richard and Hogberg, Doug and Hu, John and Hundt, Robert and Hurt, Dan and Ibarz, Julian and Jaffey, Aaron and Jaworski, Alek and Kaplan, Alexander and Khaitan, Harshit and Killebrew, Daniel and Koch, Andy and Kumar, Naveen and Lacy, Steve and Laudon, James and Law, James and Le, Diemthu and Leary, Chris and Liu, Zhuyuan and Lucke, Kyle and Lundin, Alan and MacKean, Gordon and Maggiore, Adriana and Mahony, Maire and Miller, Kieran and Nagarajan, Rahul and Narayanaswami, Ravi and Ni, Ray and Nix, Kathy and Norrie, Thomas and Omernick, Mark and Penukonda, Narayana and Phelps, Andy and Ross, Jonathan and Ross, Matt and Salek, Amir and Samadiani, Emad and Severn, Chris and Sizikov, Gregory and Snelham, Matthew and Souter, Jed and Steinberg, Dan and Swing, Andy and Tan, Mercedes and Thorson, Gregory and Tian, Bo and Toma, Horia and Tuttle, Erick and Vasudevan, Vijay and Walter, Richard and Wang, Walter and Wilcox, Eric and Yoon, Doe Hyun, "In-Datacenter Performance Analysis of a Tensor Processing Unit," *SIGARCH Comput. Archit. News*, vol. 45, no. 2, p. 1–12, Jun. 2017. [Online]. Available: https://doi.org/10.1145/3140659.3080246

[99] Xie, Shaolin and Davidson, Scott and Magaki, Ikuo and Khazraee, Moein and Vega, Luis and Zhang, Lu and Taylor, Michael B., "Extreme Datacenter Specialization for Planet-Scale Computing: ASIC Clouds," *SIGOPS Oper. Syst. Rev.*, vol. 52, no. 1, p. 96–108, Aug. 2018. [Online]. Available: https://doi.org/10.1145/3273982.3273991

[100] Tagliavini, Giuseppe and Mach, Stefan and Rossi, Davide and Marongiu, Andrea and Benini, Luca, "Design and Evaluation of SmallFloat SIMD extensions to the RISC-V ISA," in *2019 Design, Automation Test in Europe Conference Exhibition (DATE)*, 2019, pp. 654–657.

[101] S. Mach and D. Rossi and G. Tagliavini and A. Marongiu and L. Benini, "A Transprecision Floating-Point Architecture for Energy-Efficient Embedded Computing," in *2018 IEEE International Symposium on Circuits and Systems (ISCAS)*, 2018, pp. 1–5.

[102] Gustafson, J.L., *The End of Error: Unum Computing*, ser. Chapman & Hall/CRC Computational Science. Taylor & Francis, 2015. [Online]. Available: https://books.google.fr/books?id=W2ThoAEACAAJ

[103] Gustafson and Yonemoto, "Beating Floating Point at Its Own Game: Posit Arithmetic," *Supercomput. Front. Innov.: Int. J.*, vol. 4, no. 2, p. 71–86, Jun. 2017. [Online]. Available: https://doi.org/10.14529/jsfi170206

[104] Besseling, Johan and Renström, Anders, "A comparative study of IEEE 754 32-bit Float and Posit 32-bit floating point format on precision. : Using numerical methods." p. 36, 2020.

[105] Forget, Luc and Uguen, Yohann and de Dinechin, Florent, "Comparing posit and IEEE-754 hardware cost," Apr. 2021, working paper or preprint. [Online]. Available: https://hal.archives-ouvertes.fr/hal-03195756

[106] Lee, S.C. and Edgar, A.D., "Addendum to "The Focus Number System"," *IEEE Transactions on Computers*, vol. C-28, no. 9, pp. 693–693, 1979.

[107] N.G. Kingsbury, "Digital filtering using logarithmic arithmetic," *Electronics Letters*, vol. 7, pp. 56–58(2), January 1971. [Online]. Available: https://digital-library.theiet.org/content/journals/10.1049/el_19710039

[108] M. Haselman and M. Beauchamp and Aaron Wood and S. Hauck and K. Underwood and K. Hemmert, "A comparison of floating point and logarithmic number systems for FPGAs," *13th Annual IEEE Symposium on Field-Programmable Custom Computing Machines (FCCM'05)*, pp. 181–190, 2005.

[109] Brand, Marcel and Witterauf, Michael and Bosio, Alberto and Teich, Jürgen, "Anytime Floating-Point Addition and Multiplication-Concepts and Implementations," in *2020 IEEE 31st International Conference on Application-specific Systems, Architectures and Processors (ASAP)*, 2020, pp. 157–164.

[110] Khaing Yin Kyaw and Wang Ling Goh and Kiat Seng Yeo, "Low-power high-speed multiplier for error-tolerant application," in *2010 IEEE International Conference of Electron Devices and Solid-State Circuits (EDSSC)*, 2010, pp. 1–4.

[111] Kahng, Andrew B. and Kang, Seokhyeong, "Accuracy-configurable Adder for Approximate Arithmetic Designs," in *Design Automation Conference (DAC)*, 2012.

[112] Sampson, Adrian and Baixo, Andre and Ransford, Benjamin and Moreau, Thierry and Yip, Joshua and Ceze, Luis and Oskin, Mark, "ACCEPT: A Programmer-Guided Compiler Framework for Practical Approximate Computing," p. 14.

[113] A. C. I. Malossi and M. Schaffner and A. Molnos and L. Gammaitoni and G. Tagliavini and A. Emerson and A. Tomás and D. S. Nikolopoulos and E. Flamand and N. Wehn, "The transprecision computing paradigm: Concept, design, and applications," in *2018 Design, Automation Test in Europe Conference Exhibition (DATE)*, Mar. 2018, pp. 1105–1110.

[114] Mach, Stefan and Schuiki, Fabian and Zaruba, Florian and Benini, Luca, "FPnew: An Open-Source Multi-Format Floating-Point Unit Architecture for Energy-Proportional Transprecision Computing," *arXiv:2007.01530 [cs]*, Jul. 2020, arXiv: 2007.01530. [Online]. Available: http://arxiv.org/abs/2007.01530

[115] Zaruba, Florian and Benini, Luca, "The Cost of Application-Class Processing: Energy and Performance Analysis of a Linux-Ready 1.7-GHz 64-Bit RISC-V Core in 22-nm FDSOI Technology," *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, vol. 27, no. 11, pp. 2629–2640, 2019.

[116] "FPNEW: Parametric floating-point unit," https://github.com/pulp-platform/fpnew, 2019.

[117] Matoussi, Oumaima and Durand, Yves and Sentieys, Olivier and Molnos, Anca, "Error Analysis of the Square Root Operation for the Purpose of Precision Tuning: a Case Study on K-means," in *ASAP 2019 - 30th IEEE International Conference on Application-specific Systems, Architectures and Processors*. New York, United States: IEEE, Jul. 2019. [Online]. Available: https://hal.inria.fr/hal-02183945

[118] Panchekha, Pavel and Sanchez-Stern, Alex and Wilcox, James R. and Tatlock, Zachary, "Automatically Improving Accuracy for Floating Point Expressions," *SIGPLAN Not.*, vol. 50, no. 6, p. 1–11, Jun. 2015. [Online]. Available: https://doi.org/10.1145/2813885.2737959

[119] Sanchez-Stern, Alex and Panchekha, Pavel and Lerner, Sorin and Tatlock, Zachary, "Finding Root Causes of Floating Point Error with Herbgrind," *arXiv:1705.10416 [cs]*, Jun. 2018, arXiv: 1705.10416. [Online]. Available: http://arxiv.org/abs/1705.10416

[120] Benz, Florian and Hildebrandt, Andreas and Hack, Sebastian, "A Dynamic Program Analysis to find Floating-Point Accuracy Problems," p. 10.

[121] Graillat, Stef and Jézéquel, Fabienne and Picot, Romain and Févotte, François and Lathuilière, Bruno, "Auto-tuning for floating-point precision with Discrete Stochastic Arithmetic," *Journal of computational science*, 2019. [Online]. Available: https://hal.archives-ouvertes.fr/hal-01331917

[122] Lam, Michael O. and Hollingsworth, Jeffrey K. and de Supinski, Bronis R. and Legendre, Matthew P., "Automatically Adapting Programs for Mixed-precision Floating-point Computation," in *Proceedings of the 27th International ACM Conference on International Conference on Supercomputing*, ser. ICS '13. New York, NY, USA: ACM, 2013, pp. 369–378. [Online]. Available: http://doi.acm.org/10.1145/2464996.2465018

[123] Rubio-González, Cindy and Nguyen, Cuong and Nguyen, Hong Diep and Demmel, James and Kahan, William and Sen, Koushik and Bailey, David H. and Iancu, Costin and Hough, David, "Precimonious: Tuning Assistant for Floating-point Precision," in *International Conference for High Performance Computing, Networking, Storage and Analysis (SC)*, 2013.

[124] Rubio-González, Cindy and Hough, David and Nguyen, Cuong and Mehne, Benjamin and Sen, Koushik and Demmel, James and Kahan, William and Iancu, Costin and Lavrijsen, Wim and Bailey, David H., "Floating-point precision tuning using blame analysis," in *Proceedings of the 38th International Conference on Software Engineering - ICSE '16*. Austin, Texas: ACM Press, 2016, pp. 1074–1085. [Online]. Available: http://dl.acm.org/citation.cfm?doid=2884781.2884850

[125] Ho, N. and Manogaran, E. and Wong, W. and Anoosheh, A., "Efficient floating point precision tuning for approximate computing," in *2017 22nd Asia and South Pacific Design Automation Conference (ASP-DAC)*, Jan. 2017, pp. 63–68.

[126] Fousse, Laurent and Hanrot, Guillaume and Lefèvre, Vincent and Pélissier, Patrick and Zimmermann, Paul, "MPFR: A Multiple-precision Binary Floating-point Library with Correct Rounding," *ACM Trans. Math. Softw.*, vol. 33, no. 2, Jun. 2007. [Online]. Available: http://doi.acm.org/10.1145/1236463.1236468

[127] Flegar, Goran and Scheidegger, Florian and Novaković, Vedran and Mariani, Giovani and Tomás, Andrés E. and Malossi, A. Cristiano I. and Quintana-Ortí, Enrique S., "FloatX: A C++ Library for Customized Floating-Point Arithmetic," *ACM Trans. Math. Softw.*, vol. 45, no. 4, pp. 40:1–40:23, Dec. 2019. [Online]. Available: http://doi.acm.org/10.1145/3368086

[128] Tagliavini, G. and Marongiu, A. and Benini, L., "FlexFloat: A Software Library for Transprecision Computing," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, pp. 1–1, 2018.

[129] Chatelain, Yohan and Petit, Eric and de Oliveira Castro, Pablo and Lartigue, Ghislain and Defour, David, "Automatic Exploration of Reduced Floating-Point Representations in Iterative Methods," in *Euro-Par 2019: Parallel Processing*, R. Yahyapour, Ed. Cham: Springer International Publishing, 2019, pp. 481–494.

[130] Nethercote, Nicholas and Seward, Julian, "Valgrind: a framework for heavyweight dynamic binary instrumentation." in *PLDI*, J. Ferrante and K. S. McKinley, Eds. ACM, 2007, pp. 89–100. [Online]. Available: http://dblp.uni-trier.de/db/conf/pldi/pldi2007.html#NethercoteS07

[131] Higham, Nicholas J., *Accuracy and stability of numerical algorithms*, 2nd ed. Philadelphia: Society for Industrial and Applied Mathematics, 2002.

[132] Cheney, E. and Kincaid, David, *Numerical Mathematics and Computing*. Cengage Learning, Aug. 2007, google-Books-ID: ZUfVZELlrMEC.

[133] Vignes, Jean, "Discrete Stochastic Arithmetic for Validating Results of Numerical Software," vol. 37, no. 1, pp. 377–390. [Online]. Available: https://doi.org/10.1023/B:NUMA.0000049483.75679.ce

[134] A. Zeller and R. Hildebrandt, "Simplifying and isolating failure-inducing input," *IEEE Transactions on Software Engineering*, vol. 28, no. 2, pp. 183–200, 2002.

[135] Noureddine Ait Said and Mounir Benabdenbi and Katell Morin-Allory, "Arbitrary Reduced Precision for Fine-grained Accuracy and Energy Trade-offs," *Microelectronics Reliability*, vol. 120, p. 114099, 2021. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0026271421000652

[136] Granlund, Torbjrn and Gmp Development Team, *GNU MP 6.0 Multiple Precision Arithmetic Library*. London, GBR: Samurai Media Limited, 2015.

[137] Christophe Denis and Pablo de Oliveira Castro and Eric Petit, "Verificarlo: Checking Floating Point Accuracy through Monte Carlo Arithmetic," in *23nd IEEE Symposium on Computer Arithmetic, ARITH 2016, Silicon Valley, CA, USA, July 10-13, 2016*, 2016, pp. 55–62. [Online]. Available: http://dx.doi.org/10.1109/ARITH.2016.31

[138] Waterman, Andrew and Asanović, Krste, "The RISC-V Instruction Set Manual, Volume I: User-Level ISA, Document Version 2019121," 2019. [Online]. Available: https://riscv.org/specifications

[139] Yamin Li and Wanming Chu, "Implementation of single precision floating point square root on FPGAs," in *Proceedings. The 5th Annual IEEE Symposium on Field-Programmable Custom Computing Machines Cat. No.97TB100186*), 1997, pp. 226–232.

[140] Lefèvre, Vincent, "Sipe: a Mini-Library for Very Low Precision Computations with Correct Rounding," Sep. 2013, working paper or preprint. [Online]. Available: https://hal.inria.fr/hal-00864580

[141] Bellard, Fabrice, "QEMU, a Fast and Portable Dynamic Translator," in *Proceedings of the Annual Conference on USENIX Annual Technical Conference*, ser. ATEC '05. USA: USENIX Association, 2005, p. 41.

[142] Waterman, Andrew and Asanović, Krste and John Hauser, "The RISC-V Instruction Set Manual, Volume II: Privileged Architecture," 2019. [Online]. Available: https://riscv.org/specifications

[143] "RISC-V Foundation official website," https://riscv.org/, 2019.

[144] "Rocket Chip Generator," Aug. 2021, original-date: 2014-09-12T07:04:30Z. [Online]. Available: https://github.com/chipsalliance/rocket-chip

[145] "Pulpino," Aug. 2021, original-date: 2016-02-18T18:15:49Z. [Online]. Available: https://github.com/pulp-platform/pulpino

[146] "PULPissimo," Aug. 2021, original-date: 2018-02-09T10:24:02Z. [Online]. Available: https://github.com/pulp-platform/pulpissimo

[147] "CVA6 RISC-V CPU," Aug. 2021, original-date: 2018-01-23T18:36:12Z. [Online]. Available: https://github.com/openhwgroup/cva6

[148] "Ibex RISC-V Core," Aug. 2021, original-date: 2017-08-08T12:16:36Z. [Online]. Available: https://github.com/lowRISC/ibex

[149] "lowRISC/lowrisc-chip: The root repo for lowRISC project and FPGA demos." [Online]. Available: https://github.com/lowRISC/lowrisc-chip

[150] "All Aboard, Part 1: The -march, -mabi, and -mtune arguments to RISC-V Compilers - SiFive." [Online]. Available: https://www.sifive.com/blog/all-aboard-part-1-compiler-args

[151] John R. Hauser, "SoftFloat release 3," https://github.com/ucb-bar/berkeley-softfloat-3, 2019.

[152] Fournel, Nicolas and Pétrot, Frédéric, "Using Binary Translation in Event Driven Simulation for Fast and Flexible MPSoC Simulation," 10 2009, pp. 71–80.

[153] Warren, Tom, "Apple is switching Macs to its own processors starting later this year," Jun. 2020. [Online]. Available: https://www.theverge.com/2020/6/22/21295475/apple-mac-processors-arm-silicon-chips-wwdc-2020

[154] Yazdanbakhsh, A. and Mahajan, D. and Esmaeilzadeh, H. and Lotfi-Kamran, P., "AxBench: A Multiplatform Benchmark Suite for Approximate Computing," *IEEE Design Test*, vol. 34, no. 2, pp. 60–68, Apr. 2017.

[155] John Walker, "FBENCH: Floating Point Benchmark," https://www.fourmilab.ch/fbench/fbench.html.

[156] Li, L. and Gautschi, M. and Benini, L., "Approximate DIV and SQRT instructions for the RISC-V ISA: An efficiency vs. accuracy analysis," in *2017 27th International Symposium on Power and Timing Modeling, Optimization and Simulation (PATMOS)*, Sep. 2017, pp. 1–8.

[157] K. Jun and E. E. Swartzlander, "Modified non-restoring division algorithm with improved delay profile and error correction," in *2012 Conference Record of the Forty Sixth Asilomar Conference on Signals, Systems and Computers (ASILOMAR)*, 2012, pp. 1460–1464.

[158] Fabienne Jézéquel and Jean Marie Chesneaux, "CADNA: a library for estimating round-off error propagation," *Comput. Phys. Commun.*, vol. 178, no. 12, pp. 933–955, 2008. [Online]. Available: https://doi.org/10.1016/j.cpc.2008.02.003

[159] N. Ait Said, M. Benabdenbi, and K. Morin-Allory, "FPU Reduced Variable Precision in Time: Application to the Jacobi Iterative Method," in *2021 IEEE Computer Society Annual Symposium on VLSI (ISVLSI) (ISVLSI-2021)*, Tampa, USA, Jul. 2021, pp. 170–175.

[160] ——, "Self-adaptive Run-time Variable Floating-Point Precision for Iterative Algorithms : a Joint HW/SW Approach," in *MDPI Electronics Journal. Special Session on Approximate Computing: Design, Acceleration, Validation and Testing of Circuits, Architectures and Algorithms in Future Systems*, 2021.

[161] N. Ait Said and M. Benabdenbi and K. Morin-Allory, "FPU Bit-Width Optimization for Approximate Computing: A Non-Intrusive Approach," in *2020 15th Design Technology of Integrated Systems in Nanoscale Era (DTIS)*, 2020, pp. 1–6.

[162] E. Carson and N. J. Higham, "A New Analysis of Iterative Refinement and Its Application to Accurate Solution of Ill-Conditioned Sparse Linear Systems," *SIAM Journal on Scientific Computing*, vol. 39, no. 6, pp. A2834–A2856, Jan. 2017. [Online]. Available: https://epubs.siam.org/doi/abs/10.1137/17M1122918

[163] ——, "Accelerating the Solution of Linear Systems by Iterative Refinement in Three Precisions," *SIAM Journal on Scientific Computing*, vol. 40, no. 2, pp. A817–A847, Jan. 2018. [Online]. Available: https://epubs.siam.org/doi/10.1137/17M1140819

[164] C. B. Moler, "Iterative Refinement in Floating Point," *J. ACM*, vol. 14, no. 2, p. 316–321, Apr. 1967. [Online]. Available: https://doi.org/10.1145/321386.321394

[165] P. Amestoy, A. Buttari, N. Higham, J.-Y. L'Excellent, T. Mary, and B. Vieuble, "Five-Precision GMRES-based iterative refinement," Apr. 2021, working paper or preprint. [Online]. Available: https://hal.archives-ouvertes.fr/hal-03190686

[166] P. Blanchard, N. J. Higham, F. Lopez, T. Mary, and S. Pranesh, "Mixed Precision Block Fused Multiply-Add: Error Analysis and Application to GPU Tensor Cores," *SIAM Journal on Scientific Computing*, vol. 42, no. 3, pp. C124–C141, Jan. 2020. [Online]. Available: https://epubs.siam.org/doi/10.1137/19M1289546

[167] H. Anzt, J. Dongarra, G. Flegar, N. J. Higham, and E. S. Quintana-Ortí, "Adaptive precision in block-Jacobi preconditioning for iterative sparse linear system solvers: Adaptive precision in block-Jacobi preconditioning for iterative solvers," *Concurrency and Computation: Practice and Experience*, vol. 31, no. 6, p. e4460, Mar. 2019. [Online]. Available: http://doi.wiley.com/10.1002/cpe.4460

[168] Saad, Yousef, *Iterative Methods for Sparse Linear Systems*, ser. Other Titles in Applied Mathematics. Society for Industrial and Applied Mathematics, Jan. 2003. [Online]. Available: https://epubs.siam.org/doi/book/10.1137/1.9780898718003

[169] S., G. W. and Barrett, Richard and Berry, Michael and Chan, Tony F. and Demmel, James and Donato, June and Dongarra, Jack and Eijkhout, Victor and Pozo, Roldan and Romine, Charles and van der Vorst, Henk, "Templates for the Solution of Linear Systems: Building Blocks for Iterative Methods." *Mathematics of Computation*, vol. 64, no. 211, p. 1349, Jul. 1995. [Online]. Available: https://www.jstor.org/stable/2153507?origin=crossref

[170] N. Ait Said, "AxQEMU: a Floating-Point Approximation-aware Emulator," https://github.com/noureddine-as/axqemu, Aug. 2021, original-date: 2019-09-17T13:30:55Z. [Online]. Available: https://github.com/noureddine-as/axqemu

[171] "The Open-source Repository of CORE-V CVA6 CPU: an Application class 6-stage RISC-V CPU capable of booting Linux," https://github.com/openhwgroup/cva6, Jun. 2021, accessed 2021-06-13. [Online]. Available: https://github.com/openhwgroup/cva6

[172] F. de Dinechin and C. Lauter and G. Melquiond, "Certifying the Floating-Point Implementation of an Elementary Function Using Gappa," *IEEE Transactions on Computers*, vol. 60, no. 2, pp. 242–253, 2011.

[173] "RISC-V Proxy Kernel." https://github.com/riscv/riscv-pk, 2019.

[174] A. Borghesi, G. Tagliavini, M. Lombardi, L. Benini, and M. Milano, "Combining learning and optimization for transprecision computing," in *Proceedings of the 17th ACM International Conference on Computing Frontiers*, ser. CF '20. New York, NY, USA: Association for Computing Machinery, 2020, p. 10–18. [Online]. Available: https://doi.org/10.1145/3387902.3392615

[175] "Risc-v psabi documentation." [Online]. Available: https://github.com/riscv/riscv-elf-psabi-doc

List of Figures

List of Tables

Abbreviations

ABI Application Binary Interface . . . . . 60, 66, 74, XXXII ff.

AI Artificial Intelligence 106

ALS Approximate Logic Synthesis 19

ALU Arithmetic Logic Unit . . . . . 25, 32, 62, XXXIV

ANN Artificial Neural Network 22

API Application Programming Interface 8

APU Accelerated Processing Unit 25

ARP Arbitrary Reduced Precision 7, 34 f., 41, 48, 51, 53 f., 58 f., 68, 74, 82, 91, 94 f., 99 ff., 104, 107 ff., 139 f., XXVII

ASIP Application-Specific Instruction-Set Processor 31, 58

AxC Approximate Computing . 1, 6, 10, 13–25, 27, 29 ff., 33, 35, 37, 41, 48, 82, 139

CLA Carry-Lookahead Adder 26

CNN Convolutional Neural Network 16, 25, 29

CP Convergence Profile 104, 109, 112, 114, 117, 120, 122, 128

CSR Control and Status Register . . . . . 51, 53 f., 61, 69, 71, 109, XXIX f.

CV Computer Vision 106, 139, 141

DBT Dynamic Binary Translation . . . . . 62 ff.

DFS Dynamic Frequency Scaling 18

DNN Deep Neural Network 29

DRAM Dynamic Random-Access Memory 25

DSE Design Space Exploration . . . . . 57 f., 63, 72 f., 81, 86, 94, 97, 139, 141

DVFS Dynamic Voltage and Frequency Scaling 18, 30

DVS Dynamic Voltage Scaling 18, 30

EA Evolutionary Algorithm 19

EDA Electronic Design Automation 88, 101

EH Evolvable Hardware 19

EU Execution Unit 25

FMA Fused Multiplication-Addition . . . . . 31 f., 49, 53, 140

FP Floating-Point . . . . . 1, 6 ff., 10, 14, 24, 28–38, 40–48, 51–55, 57 f., 60 ff., 65–69, 71–75, 77 ff., 82 ff., 87–90, 93 f., 97, 99 ff., 104, 106–109, 114, 116, 118, 128 f., 135, 139 ff., XXIX–XXXIV

FPGA Field-Programmable Gate Array 18, 22, 29, 35, 58, 141

FPU Floating-Point Unit . . . . . 10, 25, 28 f., 31 f., 41, 49, 51–55, 58, 60–63, 66, 69, 71 f., 74 f., 77 ff., 81–86, 88–92, 94–102, 106, 108 ff., 112 f., 117, 128 f., 131 f., 134 f., 139 ff., XXIX, XXXII

FSM Finite-State Machine . . . . . 41, 51 f., 55

FU Functional Unit . . . . . 23, 25, 62, XXIX, XXXIV

FxP Fixed-Point . . . . . 27 ff., 35

GPGPU General-Purpose Graphics Processing Unit 24

GPP General-Purpose Processor . . . . . 31, 37, 42, 58, 60, XXVIII f.

GPR General-Purpose Register . . . . . 60 f., XXIX f.

GPU Graphics Processing Unit 58, 141

HAL Hardware Abstraction Layer 86, 106

HPC High-Performance Computing . . . . . 36 f., XXVIII

IC Integrated Circuit . . . . . 2 f.

IoT Internet of Things 3, 6

IR Intermediate Representation . . . . . 63 ff.

ISA Instruction Set Architecture . . 14, 23, 25, 30, 35, 47, 51, 53 f., 57–62, 64, 66, 69, 71, 74, 141, XXVI–XXX, XXXII ff.

LNS Logarithmic Number System 29

LSU Load-Store Unit 25, 32

ML Machine Learning 3, 6, 16, 29, 139, 141

MRE Mean Relative Error 72

MSE Mean Squared Error 72

MxP Mixed-Precision . . . . . 8 f.

NN Neural Network 22, 139, 141

NoC Network on Chip 24

NPU Neural Processing Unit 22

NTC Near-threshold Computing 19

NVMs Non-Volatile Memories 5

OS Operating System . . . . . 8 f., 70

PC Program Counter 54, 63, 65, 71

PVP Precision Variation Profile 104, 112, 117, 120, 123

QEMU Quick EMUlator . 10, 57 f., 60, 62–72, 74 ff., 78 f., 82, 88, 90, 94, 96, 98, 101, 107 ff., 139, 141, XXXVII, XXXIX

QoR Quality of Result . 8 f., 16, 18 f., 21 f., 24 f., 28, 34 f., 38, 41, 51, 55, 58, 71 f., 75, 77 ff., 82, 87, 90 f., 93–98, 100 f., 107, 139, 141

RCA Ripple-Carry Adder 26

SA Selective Approximation . . . . . 20, 40, 50, 52, 54, 57, 67 ff., 71, 90

SIMD Single Instruction, Multiple Data 26

SoA State of the Art . . 4, 8 ff., 13 f., 16, 18 ff., 22, 24, 26, 28, 30, 32, 34–38, 41, 45, 47, 58, 60, 63, 79, 82, 94 f., 101, 109, 139 ff., XXIX

SoC System on Chip . . . . . 14, 22, 30, 59, 62, 109, XXVII

SRAM Static Random-Access Memory 25

SSIM Structural Similarity Index Measure 28, 72

SSS Solid-State Storage 25

TB Translation Block . . . . . 63 ff.

TC Transprecision Computing . . . . . 1, 6–10, 13 f., 31, 33, 35 ff., 41, 48, 82, 139, 141

TCG Tiny Code Generator 63–67

TPU Tensor Processing Unit 29

UNUM Universal Number 29, 35

VPT Variable Precision in Time . . 7, 10, 35, 37, 49, 54 f., 59, 81 ff., 85 f., 89, 91, 101, 104, 106, 108 ff., 112–124, 126, 128–132, 134 ff., 139 ff., XXVII

VTO Variable Type Optimization . . . . . 7, 34 ff., 58, 94 ff., 100 f., 107, 129, 131, 135

WLO Word-Length Optimization 28

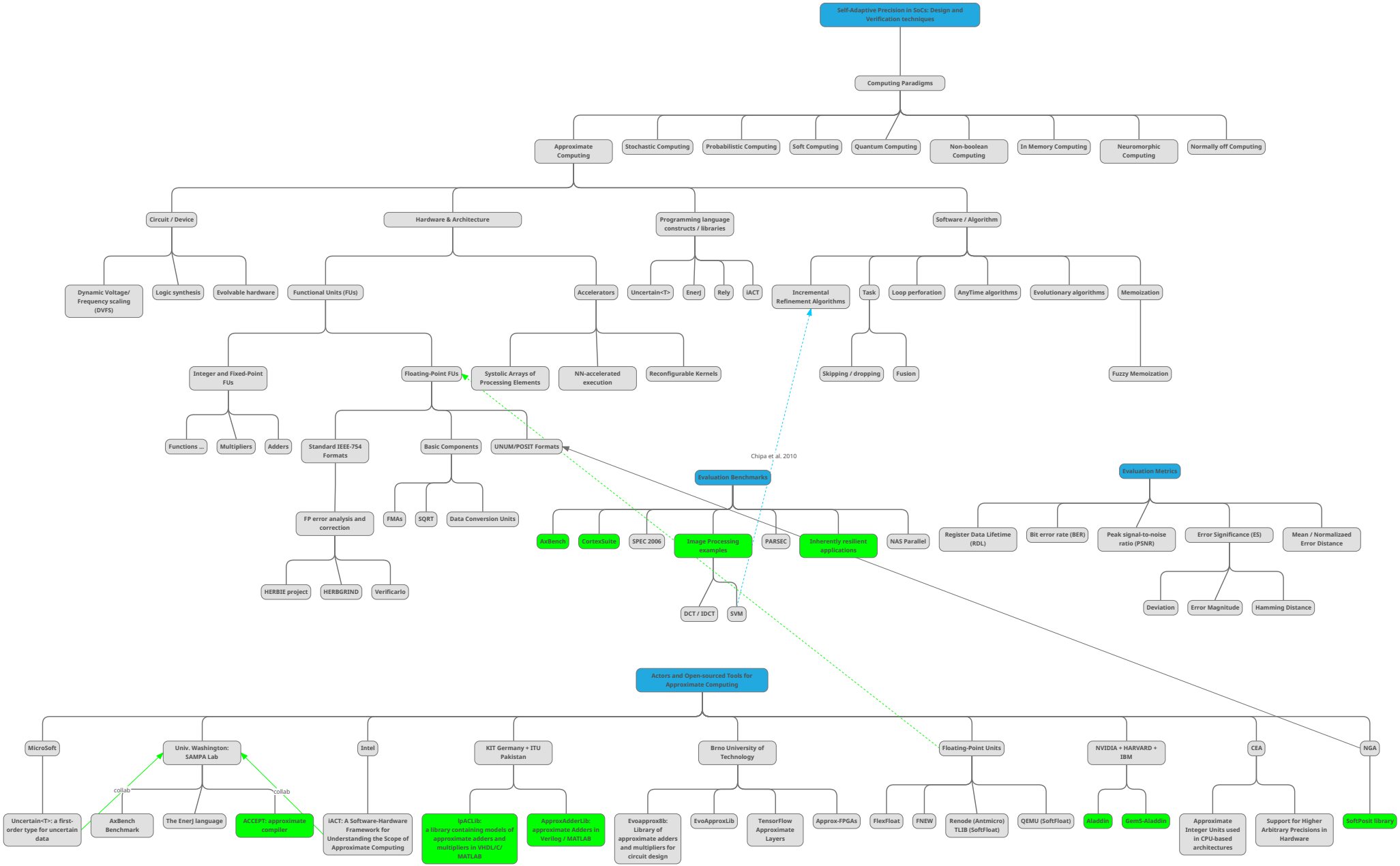
A

Mind Map of the State of the Art

This is the state of the art summarized on a mind map.

A.1 Mind Map: State of the Art

This is the state of the art summarized on a mind map.


Self-Adaptive Precision in SoCs: Design and Verification techniques
Computing Paradigms
Approximate Computing
Stochastic Computing
Probabilistic Computing
Soft Computing
Quantum Computing
Non-boolean Computing
In Memory Computing
Neuromorphic Computing
Normally off Computing
Circuit / Device
Hardware & Architecture
Programming language constructs / libraries
Software / Algorithm
Dynamic Voltage/ Frequency scaling (DVFS)
Logic synthesis
Evolvable hardware
Functional Units (FUs)
Accelerators
Uncertain<T>
Enerj
Rely
iACT
Incremental Refinement Algorithms
Task
Loop perforation
AnyTime algorithms
Evolutionary algorithms
Memoization
Integer and Fixed-Point FUs
Floating-Point FUs
Systolic Arrays of Processing Elements
NN-accelerated execution
Reconfigurable Kernels
Skipping / dropping
Fusion
Fuzzy Memoization
Functions ...
Multipliers
Adders
Standard IEEE-754 Formats
Basic Components
UNUM/POSIT Formats
Chipa et al. 2010
Evaluation Benchmarks
Evaluation Metrics
FP error analysis and correction
FMAs
SQRT
Data Conversion Units
AxBench
CortexSuite
SPEC 2006
Image Processing examples
PARSEC
Inherently resilient applications
NAS Parallel
Register Data Lifetime (RDL)
Bit error rate (BER)
Peak signal-to-noise ratio (PSNR)
Error Significance (ES)
Mean / Normalizaed Error Distance
HERBIE project
HERBGRIND
Verificarlo
DCT / IDCT
SVM
Deviation
Error Magnitude
Hamming Distance
Actors and Open-sourced Tools for Approximate Computing
MicroSoft
Univ. Washington: SAMPA Lab
Intel
KIT Germany + ITU Pakistan
Brno University of Technology
Floating-Point Units
NVIDIA + HARVARD + IBM
CEA
NGA
collab
collab
Uncertain<T>: a first-order type for uncertain data
AxBench Benchmark
The Enerj language
ACCEPT: approximate compiler
iACT: A Software-Hardware Framework for Understanding the Scope of Approximate Computing
IpACLib: a library containing models of approximate adders and multipliers in VHDL/C/ MATLAB
ApproxAdderLib: approximate Adders in Verilog / MATLAB
Evoapprox8b: Library of approximate adders and multipliers for circuit design
EvoApproxLib
TensorFlow Approximate Layers
Approx-FPGAs
FlexFloat
FNEW
Renode (Antmicro) TLIB (SoftFloat)
QEMU (SoftFloat)
Aladdin
Gem5-Aladdin
Approximate Integer Units used in CPU-based architectures
Support for Higher Arbitrary Precisions in Hardware
SoftPosit library

B

The RISC-V Instruction Set Architecture

This appendix presents the RISC-V **Instruction Set Architecture** *in detail.*

B.1 Introduction

RISC-V (pronounced "risk-five") is an ISA that was originally designed to support computer architecture research and education. It is now also becoming a standard free and open architecture for industry implementations [138, 142, 143].

We chose RISC-V in our implementations for many reasons, of which we cite the following:

- A completely *open* ISA that is freely available to academia and industry.
- A *real* ISA suitable for direct native hardware implementation, not just simulation or binary translation.
- Support for the revised 2008 IEEE-754 floating-point standard [28].
- An ISA that simplifies experiments with new architecture designs.
- Well supported and actively maintained HW/SW tools thanks to the open-source HW/SW community.
- Availability of many open-source SoC and processors' implementations such as Rocket Chip [144], Pulpino [145], Pulpissimo [146], Ariane / CVA6 [115, 147] , Ibex [148], lowRISC Chip [149], and many others.

Although all the ideas explained in this manuscript, such as **Arbitrary Reduced Precision** and **Variable Precision in Time**, were only implemented on RISC-V for pragmatic and practical reasons, they are also applicable to other architectures.

B.2 Modularity through extensions

the RISC-V ISA is a modular architecture since it enables designers to support several features in a modular and parametrized way. There are two levels of specifications that describe the different components of the standard: the user-level specification [138] and the priviliged specification [142]. These specifications can be implemented partly or entirely by designers and microprocessor implementers depending on the target applications.

RISC-V defines a set of basic assembly instructions called the base Integer instruction set (I). The base integer ISA is very similar to the early RISC processors; it contains basic computational assembly instructions such as addition, subtraction, basic bit manipulation, and others. Thus, all RISC-V systems should support this base. However, the base is carefully restricted to a minimal set of instructions sufficient to provide a reasonable target for compilers, assemblers, linkers, and operating systems, providing a convenient ISA and software toolchain "skeleton" around which more customized processor ISAs can be built.

Base	Description
RV32I	32-bit Base Integer instruction set.
RV32E	A lighter version of **RV32I** designed for embedded systems.
RV64I	64-bit Base Integer instruction set
RV128I	128-bit Base Integer instruction set
Standard extensions	
M	Integer multiplication and division
A	Atomic instructions
F	Single-precision **Floating-Point** (`FLEN` = 32)
D	Double-precision **Floating-Point** (`FLEN` = 64)
Q	Quad-precision **Floating-Point** (`FLEN` = 128)
C	Compressed instructions
Counters	Performance counters
L	Decimal **Floating-Point**
B	Bit manipulation
J	Dynamically translated languages
T	Transactional memory
P	Packed-SIMD instructions
V	Vector operations
Zicsr	**Control and Status Register** Instructions
Ztso	Total store ordering

Table B.1: Frequently used RISC-V extensions and their descriptions.

The standard also defines other sets of instructions bundled as *extentions*. Table B.1 presents some of the frequently used standard extensions. The standard [138, 142] defines these extensions, instructions, corresponding encoding, behavior, and potential side effects.

This design modularity was inherited from the current agile software trends, where modularity is key to rapid prototyping and product delivery. This aspect made the ISA suitable for many uses, from massively parallel data-centric **High-**

Performance Computing, through General-Purpose Processors, to mobile / embedded edge computing.

B.2.1 ISA naming

an ISA configuration is referred to by a string that starts with the base and contains the letters of its supported extensions. For example, the architecture `RV64IMAFD` is based on the `RV64I` and supports the extensions **M**, **A**, **F**, and **D**. The number 64 in `RV64I` stands for the bit-width of the internal general-purpose and CSR registers of the CPU. As a second example, the architecture `RV32IMAFC` contains 32-bit wide registers and supports the standard extensions **M**, **A**, **F**, and compressed 16-bit instructions (**C**).

Since we target General-Purpose Processors, we will use by default the architecture `RV64IMAFD` in all our hardware and software experiments unless otherwise stated. This architecture is also referred to as `RV64G`, where the "G" stands for general-purpose. [138]

B.2.2 GPRs and CSRs

the base integer instructions are supported through a set of 32 General-Purpose Registers (GPRs) named $\texttt{x}_0, \texttt{x}_1, \ldots, \texttt{x}_{31}$. These registers are used to handle the data fed into and produced by the internal processor's FUs. The width of these registers depends on the base integer instruction set and is denoted $\texttt{XLEN} \in \{32, 64, 128\}$.

Supporting other extensions can affect the number of GPR (and CSR) registers sometimes. For example, the `RV32E` base integer instruction set (E for embedded) only provides the registers $\texttt{x}_0, \texttt{x}_1, \ldots, \texttt{x}_{15}$, whereas supporting the single-precision FP extension (**F**) adds 32 more registers $\texttt{f}_0, \texttt{f}_1, \ldots, \texttt{f}_{31}$ dedicated for Floating-Point computations within the FPU.

B.3 Floating-Point in RISC-V

In this section, we will present the ISA aspects related to FP arithmetic in RISC-V. Namely, we will briefly present which registers and instructions are added to the base ISA when the FP extensions are handled.

B.3.1 FP extensions

Until now, RISC-V supports three FP extensions **F, D**, and **Q** as depicted in table B.1. Since RISC-V is designed with extensibility and scalability in mind, other non-standard extensions in the State of the Art also support **H**, an extension dedicated for 16-bit half-precision FP arithmetic [100, 101, 114].

B.3.2 FP General-Purpose Registers

When a Floating-Point extension is supported, a total of 32 GPRs, named $\texttt{f}_0, \texttt{f}_1, \ldots, \texttt{f}_{31}$ are added. Their width `FLEN` is the total bit-width of the widest FP format supported in the system.

For instance, the typical `RV64IMAFD` architecture supporting 32-bit single-precision and 64-bit double-precision contains 32 base integer registers $\texttt{x}_0, \texttt{x}_1, \ldots, \texttt{x}_{31}$ which are 64-bit wide, since `XLEN` equals 64. It also contains an additional 32 registers $\texttt{f}_0, \texttt{f}_1, \ldots, \texttt{f}_{31}$ which are 64-bit wide, since $\texttt{FLEN} = max(32, 64) = 64$.

The FP formats whose bit-width is less than `FLEN` are encapsulated through a mechanism called NaN boxing [138]. In the previous example, this mechanism extends the narrower formats with leading binary ones to fit them in the 64-bit registers.

B.3.3 FP Control and Status Register

Rounding Mode	Mnemonic	Meaning
000	RNE	Round to Nearest, ties to Even
001	RTZ	Round towards Zero
010	RDN	Round Down (towards $-\infty$)
011	RUP	Round Up (towards $+\infty$)
100	RMM	Round to Nearest, ties to Max Magnitude
101		*Reserved for future use.*
110		*Reserved for future use.*
111	DYN	In instruction's *rm* field, selects dynamic rounding mode; In Rounding Mode register, *reserved*.

Table B.2: Rounding mode encoding [138].

31 — 8	7 — 5	4	3	2	1	0
Reserved	Rounding Mode (`frm`)	Accrued Exceptions (`fflags`)				
		NV	DZ	OF	UF	NX
24	3	1	1	1	1	1

Table B.3: Floating-point control and status register `fcsr` *[138].*

In addition to the GPRs, a CSR named `fcsr` is also added by FP extensions. Table B.3 depicts its layout.

Bits 31–8 are reserved for other standard extensions.

Bits 7-5 define the default rounding mode `frm`. Table B.2 depicts five supported rounding modes and their corresponding `frm` values. These are compatible with the IEEE 2008-754 standard's rounding modes, as explained in subsection 3.2.4.

RISC-V implements two flavors of the round-to-nearest rounding mode. There is a 6^{th} rounding mode called dynamic rounding mode (DYN). If this mode is used at run-time, the rounding mode of executed FP instructions is encoded in the instruction's binary code itself. Otherwise, the rounding mode encoded in the `frm` field is used.

The remaining bits constitute the exception flags that arise after each executed FP instruction. A description of each flag is depicted in Table B.4 [138].

Flag	Flag Meaning	Flag Description
NV	Invalid Operation	Signals an invalid FP operation (*e.g.*, multiplying ∞ and zero).
DZ	Divide by Zero	Indicates an operation involving a division by zero.
OF	Overflow	Indicates if the result cannot be represented as a finite value in the current FP format.
UF	Underflow	Indicates if the result is too small to be represented in the current FP format.
NX	Inexact	Set if the result cannot be represented precisely in the current FP format.

Table B.4: Accrued exception flag encoding and description.

B.3.4 FP formats

To indicate the FP format of each assembly instruction, a field called `fmt` is embedded in the binary code of the instruction. This field is 2-bit wide, meaning that RISC-V is technically able to support up to four FP formats. These formats are listed in table B.5 [138, Section 11.6].

FP format	Full name	Format mnemonic	*fmt* field	Supported in which extensions
`binary32`	32-bit single-precision	`S`	00	**F**, **D**, **Q**
`binary64`	64-bit double-precision	`D`	01	**D**, **Q**
`binary16`	16-bit half-precision	`H`	10	–[1]
`binary128`	128-bit quad-precision	`Q`	11	**Q**

[1] Non-standard extension in the SoA [100, 101, 114].

Table B.5: Format field encoding.

B.3.5 FP instructions

There are five types of instructions that are added when supporting an FP extension:

- Load instructions: `FLW, FLD, FLQ`
- Store instructions: `FSW, FSD, FSQ`
- Computational instructions: `FADD.x, FSUB.x, FMUL.x, FDIV.x, FMADD.x, FMSUB.x, FNMADD.x, FNMSUB.x, FSQRT.x, FMIN-MAX.x` where `x` $\in$ `{S, D, Q}`.

  In our case, all computational instructions are approximable except `FMIN-MAX`.

- Conversion and Move instructions: instructions that move or convert numbers from a source FP format to another, or from FP to integer formats and vice versa.
- Comparison instructions: `FCMP.x` where `x` $\in$ `{S, D, Q}`.
- Classify instructions: examine the value of a FP register and return a 10-bit mask defining the type of the number, *e.g.*, normal/denormal, finite/infinite/NaN, positive/negative, zero/non-zero.

All these instructions and their specifications are described in detail in the official RISC-V user-level specification [138].

B.3.6 FP emulation when an FPU is absent

Even when an ISA does not support hardware-level IEEE 2008-754 Floating-Point arithmetic, programmers can still use data types such as `float`, `double`, and `long double` and FP computations. However, at that moment, it is necessary to include an FP emulation library that emulates FP operations in software using integer arithmetic. In the case of RISC-V, emulation is performed using SoftFloat [151], a library written by J. Hauser, one of the co-authors of the RISC-V privileged specification [142].

For example, consider the `RV64IMAFD` architecture. The architecture name means that an FPU is available in the system with support for both `binary32` and `binary64` standard FP formats. Meaning that software instructions involving operations on `float` and `double` variables will be performed in hardware. However, `long double` variables will be associated with the `binary128` IEEE standard format. Since the `RV64IMAFD` does not support this type in hardware, operations on such variables will be emulated in software based on the SoftFloat library shipped with the standard C library and the GCC Toolchain.

B.4 Application Binary Interfaces

B.4.1 Definition

The choice of the architecture also affects the whole toolchain; any modification on the micro-architectural level should be reflected in the parameters passed to the compiler/linker/profiler to guarantee HW/SW compatibility [150].

The technical term that refers to this aspect is ABI. It is an interface that defines how high-level software communicates with the lower levels of the stack. For example, an ABI can be assimilated to a protocol or a contract bonding HW to SW by defining how binary code is generated from C code, how arguments are passed to functions, where functions' return values are stored, and how memory is organized.

The ISA architecture and the ABI should both be specified to the GCC compiler using the following arguments, respectively [150]:

- `-march=<ISA>` selects the architecture to target. This controls which instructions and registers are available for the compiler to use.
- `-mabi=<ABI>` selects the ABI to target. This controls the calling convention (which arguments are passed in which registers) and data layout in memory.

B.4.2 Supported ABIs

RISC-V supports a total of 8 ABIs. For `RV64`-based architectures, we can cite the following:

- `lp64` : in this case, no FP arguments are passed in registers.
- `lp64f` : 32-bit FP arguments (*i.e.*, `float` arguments) are passed in 32-bit registers. This ABI requires support for the **F** extension which provides the FP registers.
- `lp64d` : 64-bit and shorter FP arguments (*i.e.*, `double` and `float` arguments) are passed in 64-bit FP registers. This ABI requires support for the **D** extension which provides the FP registers.
- `lp64q` : 128-bit and shorter FP arguments (*i.e.*, `long double`, `double` and `float` arguments) are passed in 128-bit FP registers. This ABI requires support for the **Q** extension which provides the FP registers.

B.4.3 ISA Vs. ABI

please note that the ISA is decoupled from the ABI. In fact, the ISA determines which instructions are supported in hardware and which registers are provided, whereas the ABI defines how functions get their input arguments and deliver their outputs.

For example, it is legal to have a system with the architecture `RV64IMAFD`, where **Floating-Point** arguments are passed through integer registers $x_0 - x_{31}$ instead of the dedicated FP ones $f_0 - f_{31}$. Of course, that would require additional move operations to transport FP operands from integer registers to FP to perform computations, which may affect efficiency negatively, but it is legal.

All these ABIs require support for the **RV64I** base integer instruction set. There are many other ABIs such as `lp64q`, `ilp32`, `ilp32f`, `ilp32d`, and `ilp32e`. For more details, please refer to [150, 175].

For all our applications and experiments, we only deal with `float` and `double` variables. So we use the `LP64D` ABI along with the `RV64IMAFD` ISA.

B.4.4 Endianness, instruction encoding, and memory layout

RISC-V is little-endian, *i.e.*, the RISC-V instruction set decodes starting at the lowest-addressed byte of the instruction. However, the specification leaves open the possibility of non-standard big-endian or bi-endian systems being implemented as extensions [138].

The base instruction set has a fixed length of 32-bit naturally aligned instructions. However, when the **C** extension is supported, the system can also support

variable-length instructions where each instruction could be any number of 16-bit parcels in length [138]. This enables compact code size when dealing with memory-constrained devices.

Type	Size (Bytes)	Alignment (Bytes)
`float`	4	4
`double`	8	8
`long double`	16	16
`float _Complex`	8	4
`double _Complex`	16	8
`long double _Complex`	32	16

Table B.6: Memory size and layout for some commonly-used FP C/C++ data types.

B.4.5 Data memory layout

Data size and memory layout depend on the ABI. Table B.6 depicts the memory layout of the frequently used C/C++ **Floating-Point** data types in `lp64`-based ABIs, and `lp64d` in particular [175].

B.5 Custom instructions for domain-specific applications

RISC-V has been designed to support extensive customization and specialization. As one gets further from the base ISA, the extensions become more and more domain-specific. For example, the **B** extension is dedicated to the acceleration of bit manipulation operations mostly used in security applications. Similarly, the **J** (still work in progress and not yet standardized) will be intended to accelerate languages that are implemented via dynamic translation, including Java and Javascript. Such languages can benefit from additional ISA support for dynamic checks and garbage collection [138].

With RISC-V being an open-source specification, computer designers are free to create and support their own custom instructions while reusing the HW/SW stack already put in place by the community. For example, authors of [26, 29] have designed and implemented ISA extensions that support approximate integer arithmetic and variable high-precision FP computations (*C.f.*, subsection §5)

B.6 Performance Vs. hardware-level overhead

Computer architects should decide which extensions to support depending on the intended application domain. This choice brings significant performance boosts, but it affects both the software toolchain and the processors' hardware. For example, if the **M** extension is supported in a given RISC-V processor, it means that its underlying micro-architecture contains a hardware integer multiplier and divider. Otherwise, multiplication operations will be emulated using the base integer **I** instructions. This is technically feasible since multiplications can be emulated with a series of additions, although they are inefficient in terms of performance (execution time and throughput). On the other hand, supporting extensions means adding hardware components to the processor (ALUs, FUs, decoder logic, *etc.*), leading to more circuit area.

C Contributions

C.1 Contributions to AxQEMU

AxQEMU has been open-sourced and is accessible at the link below. The most up-to-date branch is `v5.0-variable-prec-in-time`.

https://github.com/noureddine-as/axqemu

Figure C.1 depicts a diff statistics between the original QEMU tool (the `master` branch of the same repository) and the `v5.0-variable-prec-in-time` branch. This result can be obtained by executing the following command:

```
git diff --stat master ':(exclude).gitignore' ':(exclude)*.MD' \
                       ':(exclude)contributions.txt' \
                       ':(exclude)*_BCKP' \
                       ':(exclude)*.png'
```

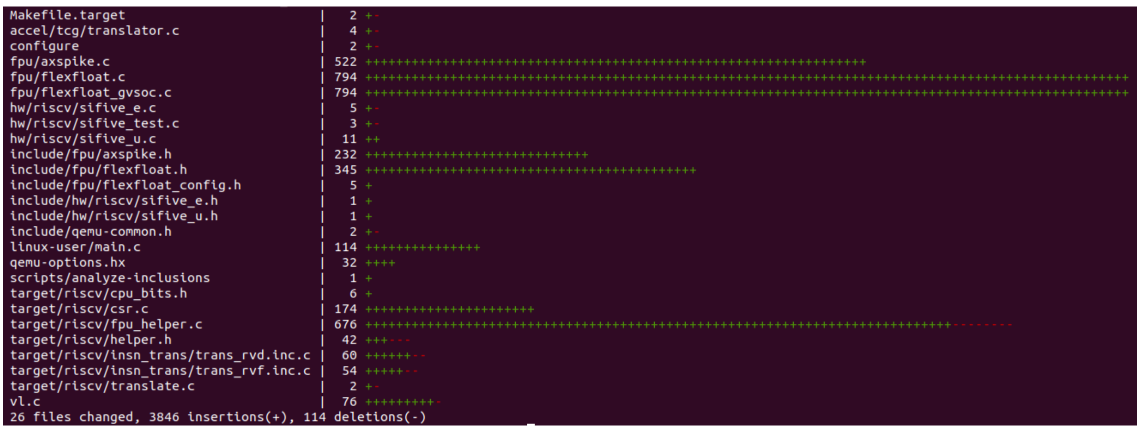


Figure C.1: Contributions to AxQEMU.

Listing C.1: Full list of AxQEMU's arguments

```
$ qemu-riscv64 --help
usage: qemu-riscv64 [options] program [arguments...]
Linux CPU emulator (compiled for riscv64 emulation)

Options and associated environment variables:

Argument               Env-variable      Description
-h                                       print this help
-help
-g port                QEMU_GDB          wait gdb connection to 'port'
-L path                QEMU_LD_PREFIX    set the elf
                                         interpreter prefix to 'path'
-s size                QEMU_STACK_SIZE   set the stack size to 'size' bytes
-cpu model             QEMU_CPU          select CPU (-cpu help for list)
-E var=value           QEMU_SET_ENV      sets targets environment
                                         variable (see below)
-U var                 QEMU_UNSET_ENV    unsets targets environment variable (see below)
-0 argv0               QEMU_ARGV0        forces target process
                                         argv[0] to be 'argv0'
-r uname               QEMU_UNAME        set qemu uname release string to 'uname'
-B address             QEMU_GUEST_BASE   set guest_base address to 'address'
-R size                QEMU_RESERVED_VA  reserve 'size'
                                         bytes for guest virtual address space
-d item[,...]          QEMU_LOG          enable logging of
                                         specified items (use '-d help' for a
                                         list of items)
-dfilter range[,...]   QEMU_DFILTER      filter logging based on address range
-D logfile             QEMU_LOG_FILENAME write logs to '
                                         logfile' (default stderr)
-p pagesize            QEMU_PAGESIZE     set the host
                                         page size to 'pagesize'
-singlestep            QEMU_SINGLESTEP   run in singlestep mode
-strace                QEMU_STRACE       log system calls
-seed                  QEMU_RAND_SEED    Seed for pseudo-random number generator
-trace                 QEMU_TRACE        [[enable=]<
                                         pattern>][,events=<file>][,file=<file>]
-version               QEMU_VERSION      display version information
                                         and exit
-expbitsd <EXP_BITS_d>                   The FPU exponent bit-width
                                         for the D extension. Default is 11
-fracbitsd <FRAC_BITS_d>                 The FPU fraction bit-width
                                         for the D extension. Default is 52
-expbitsf <EXP_BITS_f>                   The FPU exponent bit-width
                                         for the F extension. Default is 8
-fracbitsf <FRAC_BITS_f>                 The FPU fraction bit-width
                                         for the F extension. Default is 23
-non_approx_region_start <@ADDR>         The Start address of a non-approximable
                                         (.precise) region.
-non_approx_region_size <SIZE>           The Size (in Bytes) of a non-approximable
                                         (.precise) region.
```

Listing C.1 below depicts the full list of AxQEMU's arguments along with their description.

Example of AxQEMU usage in full system emulation mode.

Listing C.2: Compiling an application and emulation in Full System Emulation mode.

```
$ # Cross-Compilation using RISC-V Bare-metal GCC (Newlib Standard Library)
$ riscv64-unknown-elf-gcc -march=rv64imafd -mabi=lp64d -static \
                          application.c -o application.elf
$ # Simulation using AxQEMU
$ qemu-system-riscv64 --expbitsd 9 --fracbitsd 40 \
                      --expbitsf 6 --fracbitsf 10 \
                      -non_approx_region_start 0x1d238 \
                      -non_approx_region_size 0x428 \
                      application.elf
```

D

Appendix to Chapter 6

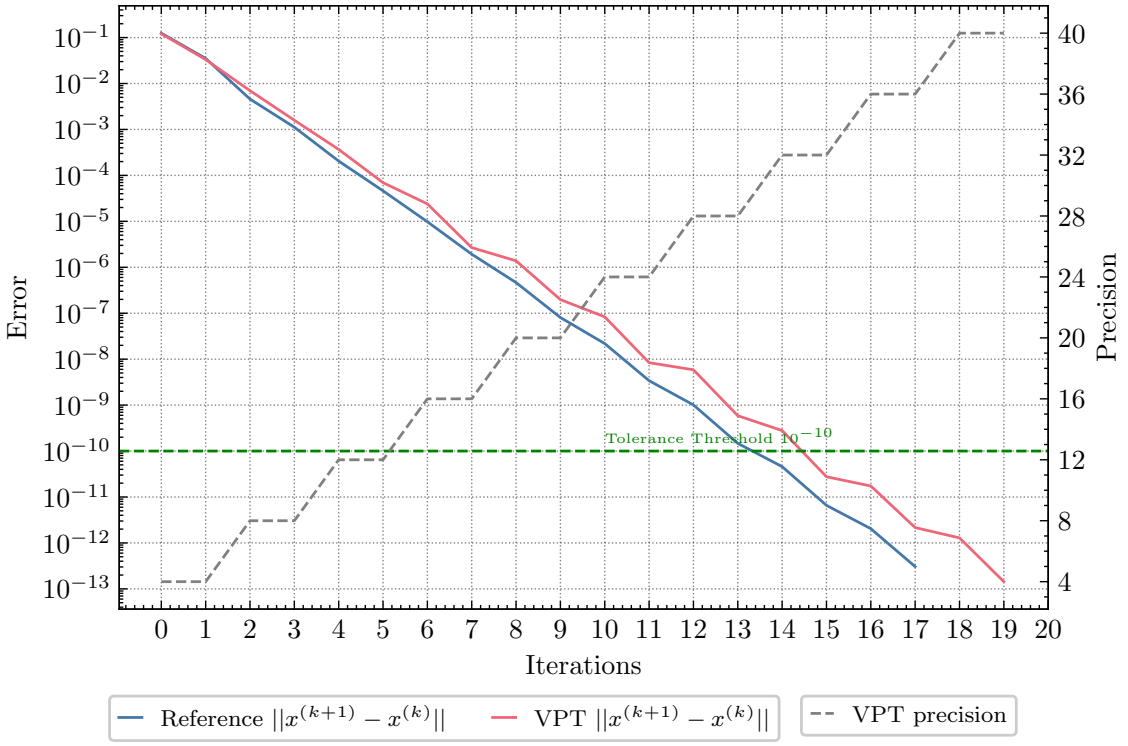


Figure D.1: The convergence profile of the VPT-enabled Gauss-Seidel.

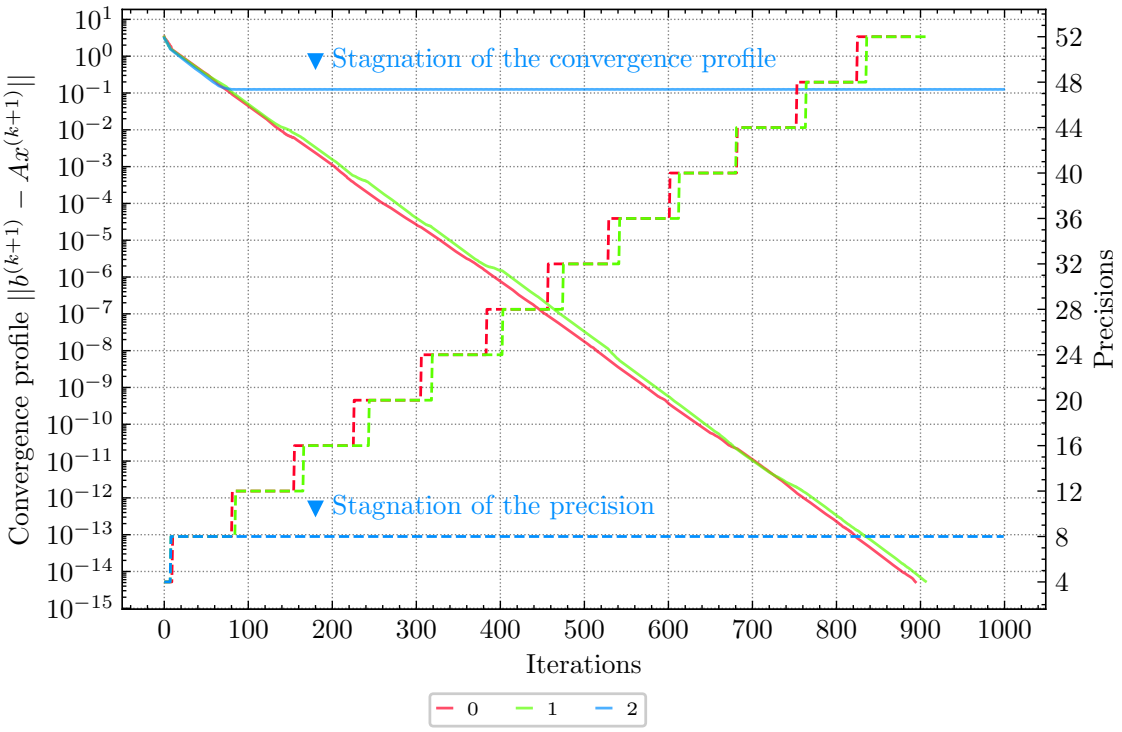


Figure D.2: Convergence profiles (left axis, continuous lines) and their corresponding precision variation profiles (right axis, dotted lines) for three inputs (Matrix IDs 0, 1, and 2). The convergence profile here is in terms of the residual error metric ($||\vec{b}^{(k+1)} - \mathbf{A}\vec{x}^{(k+1)}||$).

```
# Create a FP format with E = 11 (same as binary64) and M_p = 40
# Choose Round to Nearest, tie to even (ne)
@rnd = float<40, -1022, ne>;

# Declare a variable u, which is cast from a real value U
u = rnd(U);

# Compute expression with 40 bits of precision
square  rnd= u * u;

# Unrolled summation of squares, loops don't exist in Gappa
nsquare_1 rnd= square;
nsquare_2 rnd= nsquare_1 + square;
nsquare_3 rnd= nsquare_2 + square;
...
...
nsquare_49 rnd= nsquare_48 + square;
nsquare_50 rnd= nsquare_49 + square;

sqroot  rnd= sqrt( nsquare );
norm2      = rnd(sqroot);

{
# Hypotheses ----------------------------------------
# Given that n = 50, and U is in [0, 2^-40]
    n = 50  /\  U in [0 , 1b-40 ]
->
# The property to be proven ---------------------
    rnd(norm2) in ?
}
```



```
# Output of Gappa -----------------------------------------
Results:
float<40,-1022,ne>(norm2) in [0, 121480019999b-74, 6.431098710766519833e-12]
```

Toward Floating-Point Run-time Variable Precision in CPU-based Architectures

Résumé

De nos jours, les unités de calcul à virgule flottante (FPU) à précision standard consomment une partie importante des ressources énergétiques dans les architectures à base de processeurs. Beaucoup d'applications modernes ont une capacité inhérente à tolérer une perte de précision entraînant peu ou pas d'impact sur les résultats en sortie. Pour de telles applications, les formats de virgule flottante (FP) standardisés IEEE sont souvent surdimensionnés. Cette caractéristique a permis à un nouveau paradigme de conception d'émerger ces dernières années: le Calcul en Transprécision (TC).
Cette thèse propose plusieurs contributions qui pallient certaines limitations de l'état de l'art. Tout d'abord, nous proposons une approche minimalement intrusive qui exploite les approximations des calculs en FP pour la réduction de puissance, du temps d'exécution et de l'énergie consommée. Cette approche introduit des approximations au niveau de l'exécution/interprétation des instructions. Cela permet de traiter tous les types de binaires exécutables (applications bare-metal, applications utilisateur single-/multi-threads, OS/RTOS, etc.). La méthodologie prend en charge la précision réduite arbitraire (ARP) (formats de FP à précision réduite non standard) ainsi que la précision variable dans le temps (possibilité pour les applications d'ajuster leur précision opérationnelle au moment de l'exécution). Par la suite, une implémentation logicielle a été développée. Elle simule l'impact de ces calculs approchés sur la qualité du résultat (QoR) des applications. Enfin, une FPU matérielle multiprécision a été conçue en technologie FD-SOI ASIC de 28 nm et qui permet d'obtenir des gains d'efficacité énergétique importants.
La méthodologie proposée et ses implémentations ont été évaluées conjointement en logiciel et en matériel sur un ensemble d'applications de référence, avec un focus sur des méthodes itératives. Les expériences montrent comment des ajustements très fins de la précision peuvent être réalisés grâce à l'ARP et à la VPT, sans perte de qualité du résultat, pour générer des économies importantes en termes d'énergie consommée lors du calcul. Avec un niveau de perte de qualité acceptable sur les résultats en sortie, l'efficacité énergétique est encore accrue.

Mots-clés : Calcul approché, Calcul en Transprecision, optimisation de la largeur des bits, Précision Réduite Arbitraire (ARP), Précision Variable dans le Temps (VPT), RISC-V

Abstract

Full-precision Floating-Point Units (FPUs) can be a source of extensive hardware overhead in general-purpose and application-specific processors nowadays. As several modern applications feature an inherent ability to tolerate precision loss, a new computing paradigm has emerged: Transprecision Computing (TC). For such applications, the standard IEEE Floating-Point (FP) formats are over-designed.
This thesis proposes contributions that alleviate some State-of-the-Art limitations. First, we propose a minimally-intrusive approach that introduces approximations at the instruction execution/interpretation level, allowing the approximation of virtually all kinds of executable binaries (bare-metal applications, single-/multi-threaded user applications, OS/RTOS, *etc.*). The methodology supports Arbitrary Reduced Precision (ARP) (*i.e.,* non-standard reduced-precision FP formats) as well as Variable Precision in Time (*i.e.,* the ability for applications to tune their operating precision at run-time). Subsequently, a software simulator called AxQEMU was developed to simulate the impact of such approximations on applications' Quality of Result (QoR). Finally, a 28nm FD-SOI ASIC multi-precision FPU was designed to support ARP and VPT in hardware hence achieving real-world energy efficiency gains.
The proposed methodology and its implementations were evaluated against a set of error-resilient benchmarks with a particular emphasis on iterative methods. The experiments show how fine-grained energy/accuracy trade-offs can be made thanks to ARP and VPT, leading to drastic computational energy savings compared to traditional techniques while preserving the application's QoR. With an acceptable level of quality loss on the output results, energy efficiency is further increased.

Keywords : Approximate Computing (AC), Transprecision Computing (TC), bit-width optimization, Arbitrary Reduced Precision (ARP), Varible Precision in Time (VPT), RISC-V