Arbitrary Reduced Precision for Fine-grained Accuracy and Energy Trade-offs

N. Ait Said, Mounir Benabdenbi, Katell Morin-Allory

Arbitrary Reduced Precision for Fine-grained Accuracy and Energy Trade-offs

Noureddine Ait Said*, Mounir Benabdenbi, Katell Morin-Allory

*Univ. Grenoble Alpes, CNRS, Grenoble INP**, TIMA*
Grenoble, France

Abstract

Full-precision Floating-Point Units (FPUs) can be a source of extensive hardware overhead (power consumption, area, memory footprint, *etc.*). As several modern applications feature an inherent tolerance to precision loss, a new computing paradigm has emerged: Transprecision Computing (TC). TC proposes several tools and techniques that trade precision for energy efficiency. However, most of these tools require developers to rewrite part or all of their existing software stacks, which is often infeasible, complex, or requires extensive development efforts. In addition to their intrusiveness, TC tools can only simulate the impact of precision loss, and they do not provide corresponding hardware designs that take advantage of the simulations.

This work proposes a non-intrusive hardware-oriented approach, requiring no modification of source code that applies approximations at the low-level in assembly. The approach can be used to approximate virtually all types of executable binaries (bare-metal applications, single-/multi-threaded user applications, OS/RTOS, *etc.*). We introduce AxQEMU: a software based on the well known QEMU dynamic binary translator. We demonstrate how our approach can determine the effects of FP approximations on application-level Quality of Result (QoR), and how it interfaces with other tools from the literature. A hardware-level case study on a 28-nm FD-SOI implementation is presented, demonstrating how fine-grained energy/accuracy trade-offs can be made thanks to floating-point arbitrary reduced precision (ARP). For instance, considering the well-known `arclength` FP application, FPU computation energy savings of up to 19.4% were achieved with an accuracy threshold of 10 significant digits, and up to 60.7% with a 4-digit accuracy when using ARP. These savings compared favorably to the limited 7.7% saving afforded by using standard variable type optimization tools.

Keywords: Approximate Computing, Transprecision Computing, FPU bit-width optimization, RISC-V, ASIP, ASIC

1. Introduction

Approximate computing (AC) [1–3] is a paradigm based on the idea that the numerical accuracy of data and computations can be traded off against power savings, performance gains, and area/resource optimization. AC is suitable for non-critical RMS applications (Recognition, Mining, and Synthesis) that tolerate a certain reduction in their output precision (e.g. image/signal processing, computer vision, machine learning).

Current trends are pushing towards more adaptive, variable, and mixed-precision computing known as Transprecision Computing (TC) [4]. This paradigm targets mainly floating-point (FP) computations and storage, and the goal is to design more flexible, run-time variable precision and efficient architectures where accuracy is balanced against cost and resource savings.

In this paper, we focus on Floating-Point Units (FPUs), which are ubiquitous in modern hardware architectures, including General-Purpose Processors (GPP) and Application Specific Instruction-Set Processors (ASIP), where they serve to boost the performance of computationally-intensive applications. Unfortunately, an FPU occupies a significant proportion of the CPU's core area and can cause extensive power consumption and high memory bandwidth usage. The energy consumption associated with FP arithmetic is known to be higher than that of its integer counterpart [5], 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 [6]), 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 [7–10] in computational kernels targeting many platforms (GPPs, GPUs, FPGAs, *etc.*).

The study presented here contributes to this goal by introducing an extension to [11]. It presents a **non-intrusive methodology** enabling rapid design space exploration of reduced arbitrary precision FPUs in a CPU-based architecture context. The methodology proposed should help

** Institute of Engineering Univ. Grenoble Alpes
*Corresponding author.
Email addresses:
`noureddine.ait-said@univ-grenoble-alpes.fr` (Noureddine Ait Said), `mounir.benabdenbi@univ-grenoble-alpes.fr` (Mounir Benabdenbi), `katell.morin-allory@univ-grenoble-alpes.fr` (Katell Morin-Allory)

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. This implementation is based on the dynamic binary translation of assembly instructions, and is built on top of the well-known functional simulator QEMU. The methodology was assessed through three case studies: 1) a direct application of AxQEMU alone, 2) using AxQEMU in conjunction with an FP variable type optimization tool from the literature, and 3) a hardware-level estimation of the energy savings provided by our methodology.

Section 2 introduces our motivations, the terminology used in the remainder of the paper, and the problem statement. Section 3 presents a formalization of the idea. Section 4 shows the details of the software implementation. In Section 5, we assess the methodology developed through three detailed case studies. Section 6 presents a set of related works followed by a discussion.

2. Background & Motivation

2.1. Terminology

A binary floating-point number can be written in the form $(-1)^s \times (1+m) \times 2^e$, where s is the sign bit, m is the mantissa (also called significand or fraction), and e is the exponent. Each of these components can be encoded either following standard IEEE 754 [6] formats *e.g.,* `binary32` (32-bit single-precision format), `binary64` (64-bit double-precision format), or using a custom bit-width representation. Variations in the mantissa bit-width change the precision of the number representation, whereas alterations to the exponent bit-width change its dynamic range. Custom non-standard (arbitrary) formats can be defined when some loss of precision is tolerated, or when the numbers represented have a limited range.

For this work, we focused on the software Application Binary Interfaces (ABI) that support FP arithmetic in hardware (hard-float ABI). We assumed that the ABI and the hardware FPU support at least two standard types: `binary32` and `binary64`.

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). Typical hardware implementations contain an additional bit, or hidden bit, which is set to 1 by default and set to 0 when manipulating ± 0.0 or denormal numbers [6], when supported. The bit-widths M_f and M_d do not consider the hidden bit.

2.2. Context of the proposed approach

A typical FP algorithm implementation scenario consists mainly of three steps:

1. **Algorithm design and numerical stability analysis [12–14]:** 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. **Conservative (naive) implementation:** a software implementation using all high precision formats.
3. **Variable type optimization (VTO) [15–18]:** the process of migrating the maximum possible number of variables from high-precision to lower precisions (*e.g.,* changing *double* variables to *float*) in a given application source code while satisfying a Quality of Result (QoR) constraint. The process output is a (software) ***type configuration***: 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. It usually results in solutions 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 Arbitrary Reduced Precision (ARP):** this approach takes advantage of non-standard reduced operators.

In this work, we demonstrate that ARP can produce numerical outputs that are as accurate as standard formats while consuming less energy.

2.3. Problem statement

Given an application that takes as an input a dataset I, and providing a numerical output result O, our objective was 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.

To reach this goal, we adopted a three-step method:

① Starting from an application source code, steps 1 to 3 were performed using state-of-the-art tools [12–18] to generate a valid type configuration.
② Software design space exploration: for a given type configuration, the design space must be rapidly explored to select a set of FPU bit-width configurations satisfying the QoR constraint target.
③ FPU hardware assessments: for each configuration, an estimation of the overall energy savings was made to select the best final configuration.

These different steps can be automated using specific search algorithms. However, in this paper, for the sake of clarity, simulations were performed exhaustively.

3. Proposed Approach

In this section, we introduce some definitions along with a formalization of the approach in the form of an FSM model, the underlying details of status flag computation, and selective approximation.

3.1. Definitions & Notations

Let $\mathcal{A}$ be the set of FPU registers, $\mathcal{R}$ 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 ARP floating-point 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 $\mathcal{X}_{SP} \cup \mathcal{X}_{ARP}$. 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[1].

We define the precision reduction function $Reduce^R_{L,l} : \mathbb{F}_L \to \mathbb{F}_l$ that reduces an L-bit number to an l-bit one using rounding mode $R \in \mathcal{R}$. We also define the reverse function $Extend_{L,l} : \mathbb{F}_l \to \mathbb{F}_L$, that extends an l-bit number to produce an L-bit one.

Let $F\_inst^R_\ell\ rd, rs_1, \ldots, rs_n$ be an 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 $F\_op^R_\ell : (\mathbb{F}_\ell)^n \to \mathbb{F}_\ell$ which performs the computation in the ℓ format using the rounding mode R.

To introduce selective approximation, as a means to apply approximations to a specific set of instructions, let $\mathcal{B}$ be the Boolean domain and `do_approx` $\in \mathcal{B}$ an enable signal associated with $F\_inst^R_\ell$ which indicates whether the instruction should be approximated at run-time.

3.2. 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[2], 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:

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

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

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

where,

$$\begin{aligned} v_i &= \gamma(rs_i) \in \mathbb{F}_L && \text{high-precision values.} \\ \tilde{v}_i &= Reduce^{R'}_{L,l}(\gamma(rs_i)) \in \mathbb{F}_l && \text{reduced precision values.} \end{aligned}$$

The assembly-level approximation is introduced by performing the computation using the ARP operator $F\_op^R_l$, which operates on ARP l-bit operands , with a bit-width shorter than L. Consequently, inputs should first be reduced from L to l before performing the computation, and then extended back from l to L following the computation.

Precision reduction performs 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, and in addition double rounding (*i.e.,* rounding to reduced format plus rounding the final result) may drastically affect the QoR. When the original format L and the reduced format l have similar exponent bit-widths, the $Reduce^{R'}_{L,l}$ function is simplified to a rounding operator. Otherwise, a complete cast operator should be implemented, which may be expensive in terms of circuit area.

Once the computation $F\_op^R_l$ 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.

3.3. Computation of the status register

In addition to the numerical result computed, an FPU also returns a 5-bit status register. Since the FPU is modified to take approximations into consideration, the five flags are redefined as follows:

- **Inexact bit (NX)**: set if the final result cannot be represented precisely in the current representation. If the instruction is executed approximately, 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 $F\_op^R_l$ is inexact. The extension does not affect this flag.
- **Invalid bit (NV)**: signals an invalid FP operation (*e.g.,* multiplying ∞ and zero). This flag is set if $F\_op^R_l$ performed an invalid FP operation. The reduction and extension stages do not affect this flag.

[1] Please note that in contrast to our definition, the IEEE 2008-754 Standard's [6] computational instructions also include comparisons.

[2] A state is defined by the content of the FPU registers $r_i \in \mathcal{A}$.

- **Division by zero (DZ)**: indicates an operation involving division by zero. It is similar to the DZ flag resulting from $F\_op_l^R$.
- **Underflow (UF)**: indicates if at least one of the reduction results or the result of $F\_op_l^R$ is too small to be represented in the l format.
- **Overflow (OF)**: indicates if the result cannot be represented as a finite value in l. It has the same effect as UF.

3.4. The case of iterative operators

Some FPUs implement iterative operators $F\_op_\ell^R$, particularly for division and square root calculations [19] [20], 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 runtime.

3.5. Selective Approximation (SA)

The signal `do_approx` was added to support SA. This feature allows the programmer 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 precisely to avoid compromising the results with approximations. Hence, the developer should be able to tag such functions or instructions.

4. Implementation

To implement a proof of concept of the approach, 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 [21].

4.1. QEMU overview

QEMU supports most well-known architectures (RISC-V, ARM, x86-64, MicroBlaze, *etc.*). We distinguish two 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 1 depicts QEMU's operation principle. Efficient emulation is achieved thanks to the 'Tiny Code Generator' (TCG), which is a sort of run-time compiler embedded in QEMU. This generator dynamically translates blocks of target instructions known as Translation Blocks (TB) to TCG operations (TCGops), which constitute a machine-independent intermediate representation (IR). Subsequently, the TCGops will be translated into host instructions.

When a TB is translated into its corresponding host code, the translated block is cached for later use. 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.

4.2. AxQEMU: a floating-point approximation-aware emulator

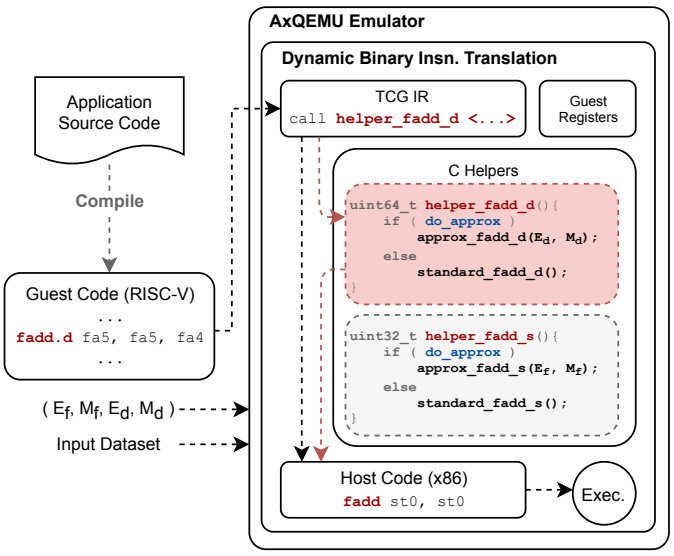


Figure 1: AxQEMU overview

TCG intermediate operations can be translated into either a single host instruction or a C helper function (Fig. 1) that will eventually be compiled to produce several host instructions. When translating floating-point 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 [7, 9, 10, 22]. 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 1 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`. The helper functions were modified to implement the behavior explained in Section 3.

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

4.3. Implementation of SA

SA 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 memory address space across two regions:

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

To implement this split, the non-approximable functions requiring precise execution are annotated using a C macro. For instance, Listing 1 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 particular memory section, the "`.precise`" section.

```
1 #define PRECISE __attribute__((__section__(".precise"))) \
2                 __attribute__((noinline))
3 double PRECISE compute_QoR(float result, double ref){
4         // Function body here
5 }
```

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 `do_approx` is set to zero and the instruction is emulated using the standard SoftFloat [22] FP emulation library. Otherwise, `do_approx` is asserted, and the instruction is emulated using an ARP library [7, 9, 10], taking the FPU configuration specified by the user into account.

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, as well as being minimally-intrusive.

4.4. Key engineering decisions

For implementation, the FlexFloat library was chosen for approximation emulation, since according to [9] it demonstrates an increase in speed of up to 2.8x compared to other mixed-precision libraries. The library provides functions for casting from full-precision types to reduced-precision types.

The reduction rounding mode R' is dynamic and follows the same rounding mode R of the instruction being approximated. The RISC-V ISA[23] 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.

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 use of the simulator when dealing with search algorithms to explore several FPU configurations.

5. Evaluation

In this section, we demonstrate the effectiveness of our methodology through three case studies. The first one is a direct application of the AxQEMU tool introduced in Section 4 on a 3D gaming application. The second demonstrates the use of AxQEMU along with a VTO tool from the literature called PROMISE[15], whereas the last one is a hardware-level trade-off study of the energy savings vs. QoR loss.

5.1. Case Study 1: Direct application of AxQEMU

With this first use case, we demonstrate the direct use of AxQEMU alone to examine how the FP approximation affects a computationally heavy application, Jmeint[24], in the 3D gaming domain. This application detects the intersection of two triangles in space and takes as an input 100,000 random pairs of 3D triangle coordinates. For this application, all variables are declared as doubles, and the QoR metric is the intersection detection miss rate. The source code was instrumented with SA macros as explained in Section 4.3. Figure 2 shows the QoR variation as a function of the exponent E_d and mantissa M_d bit-widths.

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 [6] which has a 5-bit exponent and a 10-bit mantissa, a miss rate of just 6.4% can be achieved alongside significant energy savings and reductions in memory footprint. Using ARP, it is possible to trade accuracy against hardware overheads in a more fine-grained way.

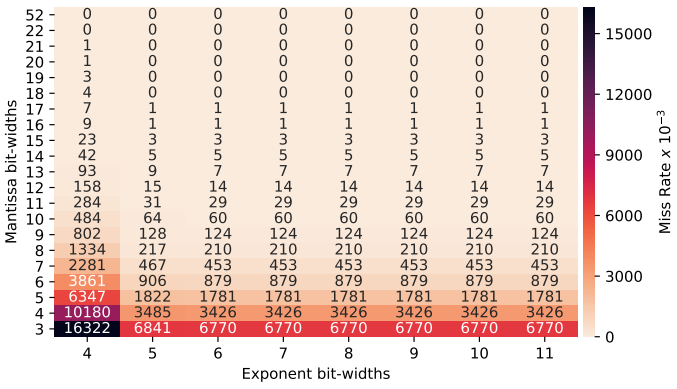


Figure 2: QoR variation as a function of the exponent E_d and mantissa M_d bit-widths.

5.2. Case Study 2: AxQEMU applied to Mixed-Precision applications

In this case study, we demonstrate how AxQEMU can be used in conjunction with other VTO tools. For this experiment, we selected the tool described in [15] which is briefly introduced in Section 5.2.1. The software type configurations described in [15] were instrumented with AxQEMU to select the best hardware configurations according to the specified QoR constraint (Phase ②).

5.2.1. The PROMISE tool

PROMISE[15][4] was selected as the primary tool for VTO (Phase ①). Given an application and an input dataset, it applies a delta debugging algorithm [25] 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 [26], which implements Discrete Stochastic Arithmetic (DSA)[27], a technique that estimates round-off error propagation and detects numerical instabilities in a program.

5.2.2. Benchmark QoR Metric

In the remainder of this section, we use the same QoR metric as in [15] *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 $(8, 23, 11, 52)$ configuration. The numerical results for these applications are non-zero.

5.2.3. Evaluation benchmarks

The approach was evaluated using three applications from [15]: `arclength`, `rectangle`, and `squareroot`. For each application, an optimized type configuration is generated by PROMISE for various QoR thresholds (Phase ①, Section 2.3).

Table 1 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 1: Benchmark summary

5.2.4. Design space exploration (DSE) with AxQEMU

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:

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.,* 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.

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 leads 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

[4] `http://promise.lip6.fr`

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 $M_{fj} \geq M_{fi}$.

5.2.5. Phase (2) results

Phase ② (from Section 2.3) consists in performing a series of simulations using AxQEMU for each benchmark, and for each type configuration (V10, V6, V4). Figure 3 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.

5.2.6. Phase (2) analysis

Figure 3 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. Moreover, 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.

Figure 3 could be used by design engineers 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 a more fine-grained control of the accuracy vs. energy comparison.

The last column of Table 1 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 was performed in Case Study 3 to quantitatively determine which bit-width minimized the overall energy consumption.

5.3. Case Study 3: Hardware-level evaluation

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 on 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.

5.3.1. HW evaluation methodology

To perform the HW-level evaluation, an approximate-aware hardware FPU was implemented in SystemVerilog. This FPU is based on an open-source parametrized FPU[20] available online[5]. 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. For the sake of concision, we have omitted the HW implementation details.

The implementation was synthesized as an ASIC, on a 28-nm FD-SOI technology node, in the typical corner (Regular V_t, 1.00V, 25C, 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. Post-synthesis simulations were performed using Synopsys VCS®, and power consumption was estimated by considering both the circuit's static power as well as its dynamic power associated with the studied application using Synopsys PrimeTime®.

5.3.2. Phase (3) results

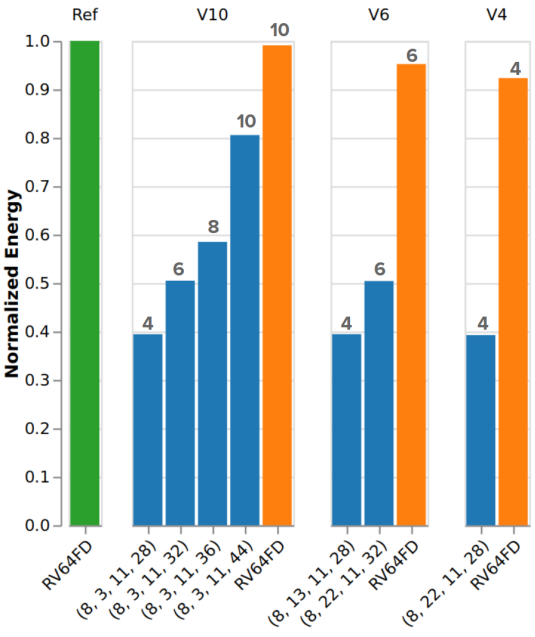


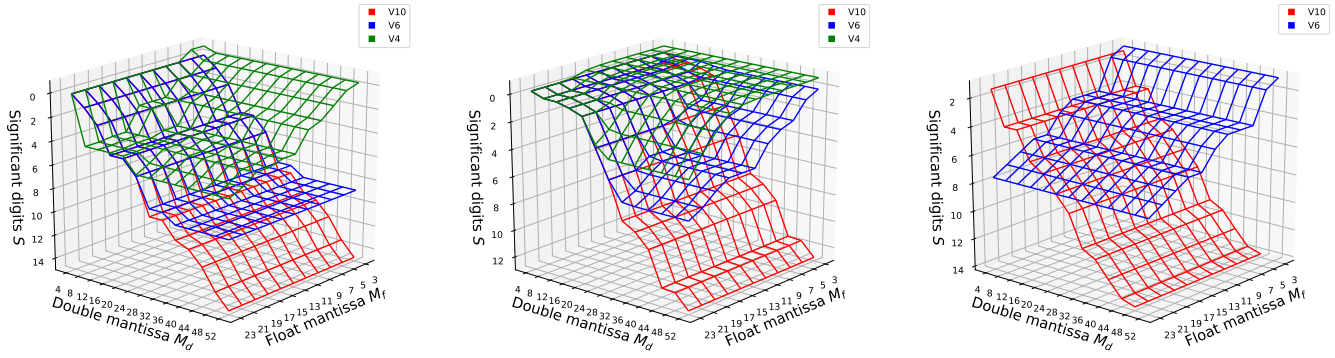
Figure 4: Energy vs. QoR trade-offs

Figure 4 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 RV64FD[6] 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. The results presented only relate to the computational part *i.e.*,

[5] `https://github.com/pulp-platform/fpnew`

[6] RISC-V architecture with D and F extensions.



(a) rectangle (b) arclength (c) squareRoot

Figure 3: QoR results for `arclength`, `rectangle`, and `squareroot`.

the amount of energy spent on floating-point 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 paper.

5.3.3. Phase (3) analysis

Figure 4 shows that the type configurations generated by PROMISE only provided 0.9%, 4.8%, and 7.7% savings on FP computation 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. No optimized type configuration can satisfy an 8-digit QoR constraint, and the designer will therefore have to select a configuration producing 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 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 RV64FD architecture.

6. Related Works

Our methodology takes advantage of ARP to improve energy efficiency. In this section we present some related works from the literature.

FP Variable type optimization. Tools such as [15–17] are designed to find, given an {application, input dataset, QoR constraint}, a variable type configuration of the original application that minimizes the number of high-precision variables and maximizes the number of low-precision variables. For some of these tools [16, 17], the objective is to optimize speed, whereas for others [15], the goal is to maximize the number of single-precision variables. As a consequence, results produced by [15] are more reproducible. All these tools are based on delta-debugging heuristics [25], which constitutes a scalability bottleneck when dealing with programs with numerous FP variables. Although [17] is relatively more scalable, the search space still depends on the number of variables in the application, and the transformations are specific to the x86 architecture. As a result, [16, 17] are architecture- and compiler-dependant.

Non-standard/arbitrary format support. The previous tools only support standard IEEE 754 [6] formats, although [17] also supports extended precision formats (*i.e.* Intel's 80-bit format implemented as `long double` in C). As explained in Section 2.2, our approach is complementary with these and other works [12–14], since it provides further computation power/energy/execution time optimization thanks to ARP and it can reuse their results to determine a first coarse-grained memory footprint and computation optimization.

Simulation of FP approximation impact. [18] proposes an arbitrary precision impact simulation methodology and introduces an automatic source code transformation tool. However, since it exploits the MPFR C library [7], it only allows variation in the mantissa bit-width. Other libraries such as FlexFloat [9] (written in C) and FloatX [10] (its C++ successor) were developed to make it possible to express variable precision behavior in software, to simulate

the effects of arbitrary FP bit-width reduction on the QoR provided by applications. Using these libraries, developers are expected to modify all or some of their FP variable types using arbitrary-precision types as substitutes for the standard ones. The code transformation effort required can be very significant, especially for low-level specialized computational kernels in C/Assembly for example. Moreover, some of the classic languages do not support advanced features such as operator overloading. Adapting existing system stacks (hardware, compilers, standard math, scientific libraries, operating systems, etc.) to FP resizing consequently becomes a tremendous development effort, especially at the production stage.

Non-intrusive impact simulation. One of the most closely related works to the one described here is VPREC[28], a back-end system that enables non-intrusive variable precision simulation in the Verificarlo[29] software toolchain. This tool was introduced to simulate in-time variable precision, specifically for iterative algorithms which benefit from adaptive gradually-increasing precision rather than fixed precision. Although the idea is very similar to ours, there are some key differences: 1) Verificarlo requires a specific instrumentation toolchain, making recompiling mandatory, and as a consequence the instrumented application is not "production-ready", 2) Instrumentation is performed on Valgrind, which means that the application was studied in a high-level operating system context. In contrast, here, we support virtually all contexts (including bare-metal). In addition, and most importantly 3) VPREC is only designed for simulation, and its conclusions cannot be readily transferred to hardware due to the lack of the necessary hardware support.

In contrast with all these approaches, ours operates on the final executable binary after all user-specific and compiler optimizations and can benefit from existing VTO tools. It can also be directly implemented in hardware to take advantage of the fast software design space exploration. To the best of our knowledge, our methodology is the first **non-intrusive hardware-oriented** approach to simulate the impact of FP approximation.

7. Discussion

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.

Limitations. The approach described in this paper 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 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.

Future work. Future studies will focus on introducing variable precision in time in AxQEMU and propose an appropriate hardware architecture that benefits from operator-level precision tuning. A more complete hardware-level evaluation taking system parameters into consideration will also be established.

Conclusion

This work introduces a hardware-friendly and code non-intrusive approach to optimize FPU bit-width sizing for a given embedded application, dataset, and quality of result level (QoR). A software implementation (AxQEMU) based on the well-known QEMU dynamic binary translator made it possible to perform a rapid evaluation of the impact of arbitrary reduced precision (ARP) on an application's QoR, without modifying the software stack. Compared to other approaches, our methodology allows fine-grained bit-width tuning to obtain the best QoR vs. energy consumption trade-off. Our approach can be used either as a standalone process or in conjunction with variable type optimization tools for fine-grained tuning. The approach was tested on several benchmarks from the literature through case studies. Several hardware FPUs were designed on an ASIC 28-nm FD-SOI technology node to demonstrate the proposed methodology's efficiency. Experimental results exhibited computational energy savings of between 19.4% and 60.7% when using ARP, compared to 7.7% when using the best current alternative tools.

Acknowledgement

The authors would like to thank Stefan Mach (ETH Zurich) and Rodrigo Iga Jodue (TIMA Laboratory) for their valuable technical advice on the HW implementation.

CRediT authorship contribution statement

N. Ait Said: Conceptualization, Methodology, Formal analysis, Software, Validation, Investigation, Data curation, Visualization, Writing - Original Draft, Writing -

Review & Editing. **M. Benabdenbi:** Conceptualization, Writing - Review & Editing, Supervision, Project administration, Funding acquisition. **K. Morin-Allory:** Conceptualization, Formal analysis, Writing - Original Draft, Writing - Review & Editing, Supervision, Project administration.

Declaration of competing interest

The authors declare that they have no known competing financial interests or personal relationships that could have appeared to influence the work reported in this paper.

References

[1] Q. Xu, T. Mytkowicz, N. S. Kim, Approximate computing: A survey, IEEE Design Test 33 (1) (2016) 8–22. doi:10.1109/MDAT.2015.2505723.

[2] S. Mittal, A survey of techniques for approximate computing, ACM Comput. Surv. 48 (4). doi:10.1145/2893356.
URL https://doi.org/10.1145/2893356

[3] T. Moreau, J. S. Miguel, M. Wyse, J. Bornholt, A. Alaghi, L. Ceze, N. E. Jerger, A. Sampson, A Taxonomy of General Purpose Approximate Computing Techniques, IEEE Embedded Systems Letters 10 (1) (2018) 2–5. doi:10.1109/LES.2017.2758679.

[4] A. C. I. Malossi, M. Schaffner, A. Molnos, L. Gammaitoni, G. Tagliavini, A. Emerson, A. Tomás, D. S. Nikolopoulos, E. Flamand, N. Wehn, The transprecision computing paradigm: Concept, design, and applications, in: 2018 Design, Automation Test in Europe Conference Exhibition (DATE), 2018, pp. 1105–1110. doi:10.23919/DATE.2018.8342176.

[5] O. Matoussi, Y. Durand, O. Sentieys, A. Molnos, 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, IEEE, New York, United States, 2019.
URL https://hal.inria.fr/hal-02183945

[6] IEEE Standard for Floating-Point Arithmetic, IEEE Std 754-2008 (2008) 1–70doi:10.1109/IEEESTD.2008.4610935.

[7] L. Fousse, G. Hanrot, V. Lefèvre, P. Pélissier, P. Zimmermann, MPFR: A multiple-precision binary floating-point library with correct rounding, ACM Trans. Math. Softw. 33 (2). doi:10.1145/1236463.1236468.
URL http://doi.acm.org/10.1145/1236463.1236468

[8] V. Lefèvre, Sipe: a Mini-Library for Very Low Precision Computations with Correct Rounding, working paper or preprint (Sep. 2013).
URL https://hal.inria.fr/hal-00864580

[9] G. Tagliavini, A. Marongiu, L. Benini, FlexFloat: A Software Library for Transprecision Computing, IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems (2018) 1–1doi:10.1109/TCAD.2018.2883902.

[10] G. Flegar, F. Scheidegger, V. Novaković, G. Mariani, A. E. Tomás, A. C. I. Malossi, E. S. Quintana-Ortí, FloatX: A C++ Library for Customized Floating-Point Arithmetic, ACM Trans. Math. Softw. 45 (4) (2019) 40:1–40:23. doi:10.1145/3368086.
URL http://doi.acm.org/10.1145/3368086

[11] N. Ait Said, M. Benabdenbi, 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.

[12] P. Panchekha, A. Sanchez-Stern, J. R. Wilcox, Z. Tatlock, Automatically improving accuracy for floating point expressions, SIGPLAN Not. 50 (6) (2015) 1–11. doi:10.1145/2813885.2737959.
URL https://doi.org/10.1145/2813885.2737959

[13] A. Sanchez-Stern, P. Panchekha, S. Lerner, Z. Tatlock, Finding Root Causes of Floating Point Error with Herbgrind, arXiv:1705.10416 [cs]ArXiv: 1705.10416.
URL http://arxiv.org/abs/1705.10416

[14] F. Benz, A. Hildebrandt, S. Hack, A Dynamic Program Analysis to find Floating-Point Accuracy Problems 10.

[15] S. Graillat, F. Jézéquel, R. Picot, F. Févotte, B. Lathuilière, Auto-tuning for floating-point precision with Discrete Stochastic Arithmetic, Journal of computational sciencedoi:10.1016/j.jocs.2019.07.004.
URL https://hal.archives-ouvertes.fr/hal-01331917

[16] M. O. Lam, J. K. Hollingsworth, B. R. de Supinski, M. P. Legendre, Automatically adapting programs for mixed-precision floating-point computation, in: Proceedings of the 27th International ACM Conference on International Conference on Supercomputing, ICS '13, ACM, New York, NY, USA, 2013, pp. 369–378. doi:10.1145/2464996.2465018.
URL http://doi.acm.org/10.1145/2464996.2465018

[17] C. Rubio-González, Cuong Nguyen, Hong Diep Nguyen, J. Demmel, W. Kahan, K. Sen, D. H. Bailey, C. Iancu, D. Hough, Precimonious: Tuning assistant for floating-point precision, in: SC '13: Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis, 2013, pp. 1–12. doi:10.1145/2503210.2503296.

[18] N. Ho, E. Manogaran, W. Wong, A. Anoosheh, Efficient floating point precision tuning for approximate computing, in: 2017 22nd Asia and South Pacific Design Automation Conference (ASP-DAC), 2017, pp. 63–68. doi:10.1109/ASPDAC.2017.7858297.

[19] Yamin Li, 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.

[20] S. Mach, D. Rossi, G. Tagliavini, A. Marongiu, 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.

[21] F. Bellard, Qemu, a fast and portable dynamic translator, in: Proceedings of the Annual Conference on USENIX Annual Technical Conference, ATEC '05, USENIX Association, USA, 2005, p. 41.

[22] J. R. Hauser, SoftFloat release 3, https://github.com/ucb-bar/berkeley-softfloat-3 (2019).

[23] A. Waterman, K. Asanović, The RISC-V Instruction Set Manual, Volume I: User-Level ISA, Document Version 2019121.
URL https://riscv.org/specifications

[24] A. Yazdanbakhsh, D. Mahajan, H. Esmaeilzadeh, P. Lotfi-Kamran, AxBench: A Multiplatform Benchmark Suite for Approximate Computing, IEEE Design Test 34 (2) (2017) 60–68. doi:10.1109/MDAT.2016.2630270.

[25] A. Zeller, R. Hildebrandt, Simplifying and isolating failure-inducing input, IEEE Transactions on Software Engineering 28 (2) (2002) 183–200.

[26] F. Jézéquel, J. M. Chesneaux, CADNA: a library for estimating round-off error propagation, Comput. Phys. Commun. 178 (12) (2008) 933–955. doi:10.1016/j.cpc.2008.02.003.
URL https://doi.org/10.1016/j.cpc.2008.02.003

[27] J. Vignes, Discrete stochastic arithmetic for validating results of numerical software 37 (1) 377–390. doi:10.1023/B:NUMA.0000049483.75679.ce.
URL https://doi.org/10.1023/B:NUMA.0000049483.75679.ce

[28] Y. Chatelain, E. Petit, P. de Oliveira Castro, G. Lartigue, D. Defour, Automatic exploration of reduced floating-point representations in iterative methods, in: R. Yahyapour (Ed.), Euro-Par 2019: Parallel Processing, Springer International Publishing, Cham, 2019, pp. 481–494.

[29] C. Denis, P. de Oliveira Castro, E. 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. doi:10.1109/ARITH.2016.31.
URL http://dx.doi.org/10.1109/ARITH.2016.31