Journal of Systems Architecture 155 (2024) 103257

Contents lists available at ScienceDirect

Journal of Systems Architecture

journal homepage: www.elsevier.com/locate/sysarc

Design-time methodology for optimizing mixed-precision CPU architectures on FPGA

Lev Denisov *, Andrea Galimberti, Daniele Cattaneo, Giovanni Agosta, Davide Zoni

Department of Electronics, Information and Bioengineering, Politecnico di Milano, Milan, Italy

ARTICLE INFO

Keywords:
Mixed-precision
Design-time analysis
Transprecision computing
FPGA
Softcore CPU
FPU
Co-design

ABSTRACT

Approximate computing can significantly reduce the energy consumption of computing systems. Mixed-precision hardware architectures and precision-tuning tools for software provide the ability to introduce approximations, but when applied separately, they do not give complete control over the accuracy-energy trade-off. The co-optimization of approximations in hardware and software is a complex task, but it promises considerable benefits. We present a methodology for the fast design-time selection of mixed-precision hardware-software combinations that minimize the energy consumption and the area of the target FPGA-based softcore CPUs with configurable support for floating-point and fixed-point arithmetic. Our approach can evaluate configurations more than 2000 times faster than the alternative approach of using gate-level simulation. On benchmarks from the PolyBench suite the identified hardware-software configurations showed improvement of the energy-to-solution metric ranging from 20% to 95%.

1. Introduction

Current trends in computing are pushing for more computational capabilities to address the needs of emerging applications such as machine learning and artificial intelligence, while conversely the energy and power efficiency of the computing platforms becomes an ever more important aspect to optimize.

On the one hand, moving computing tasks to the edge makes it paramount to minimize the energy consumption of embedded systems that are often either battery-powered or relying on energy harvesting. On the other hand, managing the energy and power consumed by cloud computing data centers is critical for the profitability of service providers and their end users, as well as for the overall environmental sustainability of computing.

In both scenarios, FPGA technology has surpassed its role of being just a tool for ASIC prototyping becoming a viable production-grade computing platform. In the edge computing scenario, employing FPGAs helps satisfy the tight time-to-market deadlines, as well as deliver heterogeneous system-on-chip (SoC) computing platforms that combine hard central processing units (CPUs) and programmable FPGA logic. Conversely, in the data center scenario, the ability to divide large FPGAs into independently-managed partitions that can be assigned to different users and reconfigured dynamically gave rise to the multi-tenant on-demand programmable logic offering.

At the same time, a vast amount of research has been devoted to the effective design and deployment of CPU softcores on FPGA targets [1–7]. The reconfigurability of FPGAs coupled with the generality of softcore CPUs makes it the perfect opportunity to explore their co-design at a low economic cost.

RISC-V has emerged in the last decade as the de facto standard architecture for both academic and industry research due to its intrinsic extensibility, which has provided the ideal basis for the development of novel approaches to designing CPU cores with state-of-the-art energy efficiency and performance [8,9]. However, most of the time, even for RISC-V cores, the CPU is designed first and then software is developed to take advantage of it. This is due to the disconnect between the development processes of hardware and software, which happen independently with minimal information exchange between them. Such an approach, however, is intrinsically limited, as it takes time for software developers to optimize applications for the hardware, a task typically performed by trial and error. Therefore, providing parametric and easily configurable at design time architectures is crucial for effective integration between the hardware and software domains.

At the same time, for software and algorithm design, it is often difficult to produce an energy-efficient but semantically equivalent version of a given program. This is due at least in part to the need for two different skills: expertise in the application domain needed to preserve the expected functional behavior of the application, and expertise in

* Corresponding author.
E-mail address: lev.denisov@polimi.it (L. Denisov).

https://doi.org/10.1016/j.sysarc.2024.103257
Received 23 February 2024; Received in revised form 23 July 2024; Accepted 4 August 2024
Available online 7 August 2024

low-power computing, needed to produce energy-efficient code. Since these skills usually derive from different educational and professional paths, there are only a few specialists able to master both, leading to a sharp increase in the development and maintenance costs of such code bases. This cost becomes even higher when custom hardware accelerators or specialized instruction set extensions are involved. Thus, there is a need for tools to support the automation of co-design of hardware and software for energy-efficient embedded systems.

Precision tuning is one technique that can provide significant benefits in terms of energy efficiency, hardware area, and performance. Precision tuning is a part of the family of *approximate computing* techniques, and it deals with trading off the precision of individual operations with their performance and energy efficiency, as well as with the complexity of the hardware units needed to perform them [10]. Precision tuning must be performed at the fine grain to be effective, balancing the advantages of employing the optimal data type size for each operation with the cost of performing data type casts. Automated tools aid significantly in performing optimizations of this complexity. Unfortunately, only a few such tools exist that are suitable for industrial use [10].

Main contributions. This work proposes a methodology for the automated hardware-software (HW/SW) co-design of mixed-precision CPUs meant for FPGA deployment. Differently from the existing methodologies discussed in the open literature, our proposal considers the hardware-software combination as a whole, and it can, therefore, deliver a CPU hardware architecture and optimized software that are custom-tailored to the target workload to maximize the energy efficiency of the computing platform while guaranteeing the required accuracy.

Our manuscript provides four main contributions:

1. we introduce two error estimation approaches, namely an approach based on a machine-learning-based model and a simulation-based one, and compare them with a state-of-the-art method; the proposed error estimation techniques are combined with performance and area metrics to drive the multi-objective exploration of the design space;
2. we propose a multi-stage design space exploration (DSE) process meant for mixed-precision, FPGA-based softcore CPUs that minimizes the time needed for the exploration while maximizing the quality of the resulting solutions, with a reduction of more than 2000× in the time required by the DSE compared to the gate-level simulation-based approach;
3. we evaluate the effectiveness of our proposal through a comprehensive experimental campaign that targets, as the mixed-precision hardware architecture, a softcore FPGA-based 32-bit RISC-V CPU with configurable support for both floating-point and fixed-point arithmetic and, as the software application workload, a set of compute-intensive kernels from a state-of-the-art benchmark suite;
4. the full experimentation suite, including the simulation data and optimization framework described in this article, is available as open-source on GitHub.[1]

Organization of the paper. The rest of this paper is organized as follows. In Section 2 we review the state of the art in mixed-precision computing, while in Section 3 we provide an overview of the tools adopted as the main components for our co-design methodology. In Section 4 we introduce the proposed co-design methodology, for which we then provide validation through the experimental campaign discussed in Section 5. Finally, in Section 6 we draw conclusions and highlight future research directions.

[1] https://github.com/TAFFO-org/jsa_co_design.

2. Related work

When engineering computing systems and programs, the choice of the number representation to use is often taken for granted. However, that choice has a measurable impact on the quality of service provided and the energy efficiency of the system [11], and the latter property is of great interest for embedded systems in general, and for other fields such as cloud computing, in order to enable more sustainable data center operation [12]. In particular, adopting lower-precision representations has been shown to significantly impact on energy consumption in low-power embedded platforms [13–16]. This approximate computing practice, which focuses on choosing the best data representation to use in a computation, is called *precision tuning* [10,17]. Precision tuning allows automatically transforming a program into a reduced precision form, saving time and effort for the programmer.

The most important aspect of precision tuning is the choice of the data types, which is typically performed to enable the automatic transformation of a program into a reduced-precision form. Most precision tuning research targeted to high-performance-computing focuses on the use of differently-sized floating point representations. Instead, in tools targeted to embedded systems applications, the typical goal is to move from floating-point representations to fixed point ones. Indeed, fixed point representations have been exploited in hardware-software co-design techniques since the early years of ASIC design, particularly for DSP applications [18]. In this context, research has developed several approaches for automating the conversion of a program from floating point to fixed point representations, such as *FRIDGE* [19], *Autoscaler for C* [20], *fixify* [21] and *Xfp* [22]. The typical limitation of these tools is that they are only able to perform the conversion to the fixed point, without considering the option of adopting small floating-point types such as IEEE 754 binary16 or bfloat16, which have nevertheless been shown to be effective for embedded applications [22]. Only some tools in this category, such as TAFFO [23], are able to simultaneously handle fixed point and floating point representations together. Another limitation is that the value added by these tools in a hardware-software co-design methodology is limited. Indeed, a tool that solely focuses on the transformation of the software has a limited ability to inform the design process of the hardware. One could perform multiple manual conversions, changing the tool parameters each time, to determine which data types and sizes are the most suitable for the application. However, this approach reintroduces the same time-consuming iterative processes that these tools intend to remove in the first place.

Indeed, current research on hardware-software co-design in approximate computing systems mostly considers design flows that instantiate optimal hardware to fit the software, without the ability to consider the hardware/software system as a whole [24]. Even where a system-level approach is followed, the methodologies being considered are often specific to a restricted application area, such as neural networks [25] or image processing [24].

3. Background

The hardware-software co-design methodology proposed in this manuscript is implemented by integrating existing mixed-precision approaches by the authors. Ideally, the mixed-precision tool we employ must support a wide range of hardware architectures, and it must also be able to handle both floating-point and fixed-point data types to consider as many relevant points in the design space as possible. This tool must also be extensible and modular to enable its modification. Among the various tools discussed in Section 2, TAFFO meets all these requirements. On the hardware side, a reconfigurable CPU core design is required to allow the instantiation of hardware that meets the recommendations made by the design methodology. In our work, we employ a highly parameterized RISC-V core implementation featuring multiple possible ALU and FPU setups, which provides a wide range of synthesizable cores for several application scenarios.

```
⟨c_ann⟩   →  __attribute((annotate(" ⟨top⟩ ")))
  ⟨top⟩   →  ⟨target⟩ ⟨datat⟩
⟨target⟩  →  target(' ⟨id⟩ ') | ε
⟨datat⟩   →  ⟨scalar⟩
⟨scalar⟩  →  scalar( ⟨datav⟩ )
⟨datav⟩   →  ⟨range⟩ ⟨final⟩
⟨range⟩   →  range( ⟨num⟩ , ⟨num⟩ ) | ε
⟨final⟩   →  final | ε
```

Fig. 1. BNF grammar of user annotations in the TAFFO precision tuning tool.

3.1. The TAFFO precision tuning framework

TAFFO is an automated precision tuning solution based on the LLVM compiler framework. TAFFO is composed of five independent passes, namely: *Initializer*, *Value Range Analysis* (VRA), *Data Type Allocation* (DTA), *Conversion*, and *Feedback Estimator* (FE). This modular architecture enables its extensibility, and the use of the production-grade LLVM framework allows TAFFO to support all state-of-the-art computer architectures and instruction sets. The INIT (Initializer) pass parses the annotations created by the developer and prepares metadata for future passes. The VRA pass calculates numerical intervals for all variables that have been annotated, as well as any other variable that is dependent on them. The intervals are propagated through the program with Interval Arithmetic [23,26,27]. This approach is efficient for programs that do not contain exceedingly long chains of arithmetic operations or combinations of operations that lead to the explosive growth of the value intervals (e.g. exponentiation to high degrees or division by quantities close to zero). In cases where every loop iteration modifies a unique array element and does not have a shared state, the value range can be calculated for the whole array by evaluating only one loop iteration. This is possible because every element of the array would undergo the exact same transformation in terms of interval arithmetic. In other cases, the standard LLVM loop trip-count estimation is used coupled with loop unrolling. The upcoming step is the DTA, which selects a suitable data type for each intermediate value and variable used. Two algorithms are available for this process: a greedy algorithm that chooses the fixed point type with the highest correct fractional point position, or a linear-optimizer-based algorithm that produces mixed precision results [28]. Conversion modifies the LLVM-IR based on the information provided by DTA. Finally, FE statically analyzes the error using state-of-the-art estimation methods [29].

TAFFO static analysis requires the programmer to provide the annotations specifying the initial value range of the variables and the precision tuning scope. The annotations are inserted in the source code and follow the formal grammar shown in Fig. 1. On scalar variables or arrays that need to be converted, the annotation contains a `scalar` declaration that marks the variables as explicitly being a part of the set of variables that require tuning. Within this declaration, additional optional attributes provide the initial range of the annotated variable (the `range` attribute), and if that range shall be assumed immutable (the `final` keyword). In addition, the `target` declaration may be used to create a new VRA and FE analysis entry point.

3.2. Mixed-precision hardware architecture

The mixed-precision CPU is characterized by a highly configurable architecture which exposes a variety of configurations to be selected at design time. The single-core, in-order CPU, based on the RISC-V ISA, can provide dedicated mixed-precision hardware support for both floating-point and fixed-point arithmetic.

Its baseline configuration, without both floating-point and fixed-point hardware support, implements the standard RISC-V I and M extensions [30].

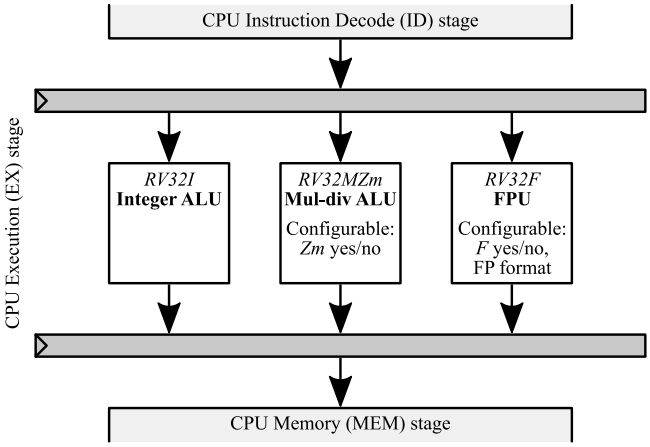

Fig. 2. Configurable architecture of the mixed-precision CPU.

The mixed-precision floating-point hardware support can be optionally provided by instantiating a floating-point unit (FPU) within the execute stage of the CPU [31]. The FPU implements the single-precision floating-point instructions defined within the RISC-V F extension, which targets the standard 32-bit IEEE 754 *float32* floating-point format, but the instantiated hardware can be configured at design time to target smaller data types. In particular, the mixed-precision FPU supports any floating-point format which is encoded by a 1-bit sign and an 8-bit exponent, as for the IEEE 754 *float32* format, while the mantissa can be encoded by any value ranging from 1 to 23 bits. The FPU can also instantiate functional units for various operations that target different floating-point formats, i.e., have different precision, in this work, we limit its configuration to a single floating-point format for the whole FPU, for the sake of simplicity.

The mixed-precision fixed-point hardware support can be optionally provided by instantiating a modified multiplication-division functional unit within the execute stage of the CPU [32]. The fixed-point hardware implements custom fixed-point instructions which are the fixed-point-equivalent of the integer multiplication and division instructions defined in the standard RISC-V M extension. In particular, the fixed-point format is the same for both the operands and for the result, the fixed-point values are 32-bit ones and the number of fractional bits is defined at run time by an immediate encoded within the opcode of the single fixed-point instruction to be executed.

Fig. 2 summarizes the configurable options that can be selected by the system designer when instantiating the mixed-precision CPU. The integer ALU, providing hardware support for the RISC-V I extension, is always implemented. The multiplication-division ALU can be instantiated either in its baseline configuration, supporting the sole RISC-V M extension, or in its fixed-point configuration, supporting both the standard RISC-V M and custom RISC-V Zm extensions, with the latter implementing fixed-point multiplication-division instructions. Finally, the CPU can optionally implement the FPU to provide support for the standard RISC-V F extension, and, when the FPU is instantiated in the CPU, the floating-point format employed internally can be selected at design time.

4. Methodology

This section describes the proposed hardware-software co-design methodology, introducing a novel multi-stage DSE process for mixed-precision, FPGA-based softcore CPUs and two error estimation approaches based on a machine-learning-based model and on a simplified simulation, respectively.

The proposed DSE process, depicted in Fig. 3, can be split into eight main steps, which are detailed in their separate subsections, namely,

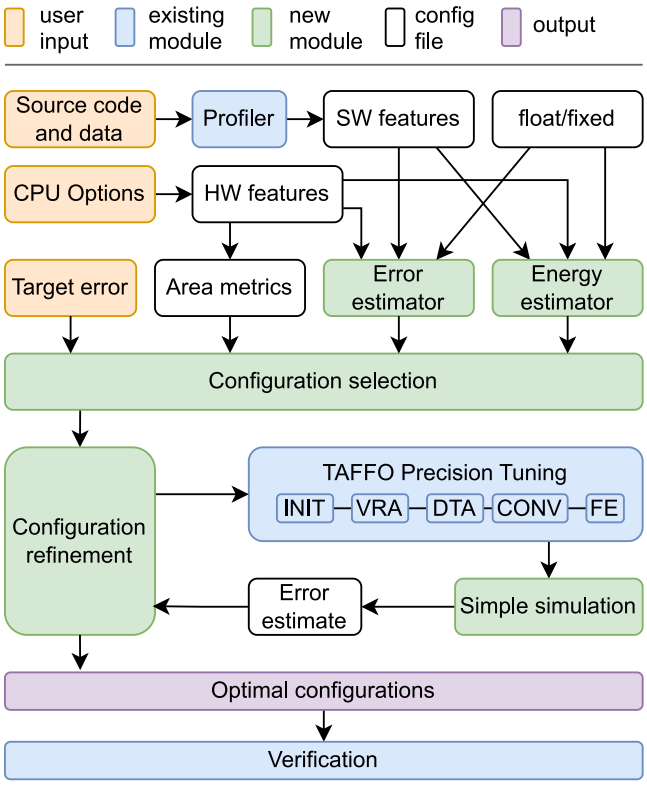

Fig. 3. Block diagram of the proposed hardware-software co-design methodology.

hardware characterization (see Section 4.1), software requirement collection (Section 4.2), error estimation (Section 4.3), energy estimation (Section 4.4), area metrics (Section 4.5), configuration selection (Section 4.6), dynamic configuration selection refinement (Section 4.7), and configuration verification (Section 4.8).

A running example aims to provide the reader with a better understanding of the proposed hardware-software co-design methodology, demonstrating its usage in a simple use case scenario. The various steps of the running example are detailed at the end of each part.

We remark that the proposed approach is independent of the tools chosen for its evaluation, i.e. the mixed-precision compiler and configurable softcore CPU, therefore we first present it in a general way in this section and later show its specific implementation in Section 5 for experimental evaluation purposes.

4.1. Hardware characterization

Before the design phase can begin, it is necessary to collect information about the reconfigurable hardware available. This step, depicted as the *HW Features* block in Fig. 3, would typically be done by the hardware vendor as this information is independent of the software running on the system, and it can be collected once and then reused. Required data concerns the set of possible hardware configurations, such as the presence of an FPU, the available floating-point formats in terms of exponent and mantissa sizes in bits, and the size of integers in bits. Apart from these parameters that can be obtained from the system definition, we also require information about the energy consumption of the floating-point arithmetic operations for all the given configurations. This information is collected by running simple micro-benchmarks with the number and the mix of floating-point operations pre-defined. The configurations are formed by the combinations of FPU implementation (FPUImp) and HW fixed-point support (HWFixed): $CONF = FPUImp \times HWFixed$. For each CPU configuration, the hardware characterization outputs (1) the availability of a HW FPU, (2) the availability of HW support for fixed-point arithmetic, (3) the area, i.e., FPGA resource utilization, (4) the average energy consumption per floating-point operation, (5) the bit length of integers, and (6) the bit length of the floating-point mantissa and exponent.

Example 1. The example scenario considers two instances of a mixed-precision FPGA-based CPU, *IMFbfloat16* and *IMZm*. The hardware characterization of the two CPU instances shows they are configured as follows:

1. *IMFbfloat16* - FPU available, no hardware fixed-point support, 10053 LUT, 7256 FF, 4 DSP, 15.6 nJ per fixed-point operation, 23.2 nJ per floating-point operation, 32-bit integer, 7-bit floating-point mantissa, 8-bit floating-point exponent.
2. *IMZm* - FPU unavailable, hardware fixed-point support, 8342 LUT, 5761 FF, 4 DSP, 14.1 nJ per fixed-point operation, 187.7 nJ per floating-point operation, 32-bit integer, 23-bit floating-point mantissa, 8-bit floating-point exponent.

4.2. Software requirement collection

This step is represented by the *Profiler* and *SW Features* blocks in Fig. 3. The design process itself starts with the user providing one or more programs that are going to be running on the system. Each program should have at least one representative input dataset. As a first step, we collect information about the program: the number of floating-point operations, dynamic ranges of the floating-point input variables, output dimensions, and acceptable relative error threshold. The developer can provide some of these properties: the output dimensions are specified explicitly, the error threshold depends on the acceptable output quality, and the dynamic ranges of inputs sometimes have natural limits (e.g. the maximum speed of a vehicle or the maximum heart rate of a patient), can be estimated, or can be specified in the operating limits of a device or an application. Others are not easily derivable: for these, we employ an automatic collection step. The automatic collection consists of instrumenting the programs with profiling instructions that log statistics about the floating-point operations performed at run time, to derive their number and run-time dynamic ranges. This instrumentation is added at LLVM-IR level. We only collect value data for variables whose range cannot be determined statically, i.e. the input variables, to reduce the amount of required logging. Fully profiling all variables can potentially improve the quality of the estimated ranges. In our previous work [33], we observed that using profiling in precision tuning can lead to up to 4 orders of magnitude lower resulting relative error than the static analysis is able to achieve in cases when representative training datasets are available, mostly due to more accurate value range estimation. However, we opt out for a simpler version of range estimation in the interest of making the configurations search faster and more robust to input data variation. In case the whole program is too big for instrumentation, only the most intensive kernels can be instrumented, or statistical approximations can be used to reduce the number of calls to the logging function. These approaches do not need to be very precise as the overall design model is not very sensitive to small variations in the number of floating-point operations. An example of how our implementation adds instrumentation to the code can be found in Listings 1 (original code) and 2 (instrumented code).

Listing 1: Requirement collection: original LLVM-IR code

```
1  ...
2  %var1 = fmul float %9, %12
3  %var2 = fadd float %10, %14
4  ...
```

Listing 2: Requirement collection: instrumented LLVM-IR code

```
@var1_name = constant "uniq1"
@var2_name = constant "uniq2"
define void @log_value(%name, %value) {...}
...
%uniq1  = fmul float %9, %12
call @log_value(@var1_name, %uniq1)
%uniq2  = fadd float %10, %14
call @log_value(@var2_name, %uniq2)
...
```

For each program, the software requirement collection outputs (1) the number of floating-point operations, (2) the minimum and maximum values for the input variables, and (3) the output size in terms of the number of elements, i.e. floating point values.

Example 2. The software workload of the running example scenario is the *3mm* kernel from the PolyBench/C benchmark suite, with an input scaling factor of 4 and a target relative error threshold of 10^{-3}. Software profiling characterizes the considered kernel as having 5220 floating-point operations, 72 output elements, and an input value range from 0 to 13.9.

4.3. Error estimation

We evaluate 3 methods for estimating the relative error for every combination of hardware and software configuration: static formula, linear model, and simplified simulation. The static formula and the linear model correspond to two possible implementations of the *Error estimator* block, while the simplified simulation corresponds to the *Simple simulation* block in Fig. 3.

4.3.1. Static formula error estimation

For floating-point configurations, we use Formula (1), that is derived from the one proposed by Rump et al. [34].

$$ERR_{fl} = 2^{log_2(N_{op})-M-log_2(D_{out})} \tag{1}$$

N_{op} is the number of floating-point operations in the program, M is the mantissa size in bits for the floating-point format chosen, and D_{out} is the number of elements in the output. This formula represents the intuitive understanding of the error in floating-point computations: the mantissa size directly defines the precision, the accuracy decreases with the number of arithmetic operations, but the more elements the output has, the less error is accumulated per element.

Similarly to floating-point configurations (see Formula (1)), we employ instead Formula (2) for fixed-point ones.

$$ERR_{fix} = 2^{max((log_2(varMax)-I),-M)} \tag{2}$$

Here, I is the integer size in bits, and M is the mantissa size in bits of the *floating-point* format when mixed-precision with both floating-point and fixed point variables is used. $varMax$ corresponds to the maximum value among the input variables: $varMax = max_{i=0..nvar}(abs(VAR_i))$, where $nvar$ is the number of input variables in the program and VAR_i is the maximum value of the variable with the index i. In this equation, we assume that, at the end of the computation, the fixed-point result will be converted into a floating-point format, as floating-point is likely to be used at the interface level between different parts of the system. In case the configuration has no FPU, we assume that the standard *float32* floating-point software implementation is used. The intuitive understanding of this formula is that the error is limited by the size of the fractional part which uses the rest of the bits after the integer part is allocated. The bit size of the integer type in the system will define the maximum precision and $varMax$ defines the number of bits taken by the integer part of the variable.

Example 3.1. The error for floating-point and fixed point configurations of *3mm* benchmark running on the *IMFbfloat16* and *IMZm* hardware can be estimated with the static Formulas (1)–(2) from the previous examples' data, resulting in 4 configurations:

- *IMFbfloat16×float*
  $ERR_{fl} = 2^{log_2(5220)-7-log_2(72)} = 5.66 \cdot 10^{-1}$
- *IMFbfloat16×fixed*
  $ERR_{fix} = 2^{max((log_2(13.9)-32),-7)} = 7.81 \cdot 10^{-3}$
- *IMZm×float*
  $ERR_{fl} = 2^{log_2(5220)-23-log_2(72)} = 8.64 \cdot 10^{-6}$
- *IMZm×fixed*
  $ERR_{fix} = 2^{max((log_2(13.9)-32),-23)} = 1.19 \cdot 10^{-7}$

4.3.2. Linear model error estimation

The machine-learning approach extends the static formula approach by trying to learn the underlying dependencies between the features of the program and the relative error. This has the advantage of possibly recovering less obvious relationships and more accurately quantifying the impact of every feature on the resulting error. The disadvantage of this method is that it highly depends on the quality and variety of the training data, and it might over-fit resulting in poor prediction power on the unseen data. Given the considerations above we choose a simple linear model that is unlikely to over-fit due to its limited capacity to remember the input. We also follow the standard machine-learning protocol by splitting the available data into training and test subsets.

We employ a linear *Ridge* regression on the gate-level simulation data, trained on the following features: *mantissa bit size (M), number of floating-point operations (N), absolute maximum initial variable value (V), number of output elements (O)*. We pre-process the *N, V, O* features by first computing their log_2 and then standardizing the latter by scaling to unit variance and removing the mean. The prediction target for the model is the error obtained from the gate-level simulation of the benchmarks. We train two separate models, for floating-point and for fixed-point configurations, respectively, in order to simplify them and improve accuracy. The hyper-parameters of the models are optimized with n-fold cross-validation on the train set.

Example 3.2. The errors estimated by using the linear model with weights given in Table 5 are:

- *IMFbfloat16×float* - $ERR = 7.70 \cdot 10^{-4}$
- *IMFbfloat16×fixed* - $ERR = 7.10 \cdot 10^{-3}$
- *IMZm×float* - $ERR = 2.27 \cdot 10^{-7}$
- *IMZm×fixed* - $ERR = 2.09 \cdot 10^{-6}$

4.3.3. Simplified simulation error estimation

This step simulates the reduced-bit hardware floating-point units in software by automatically modifying the program according to the size of the floating-point type. We propose a faster way of simulating the different FPU sizes on the general-purpose architecture. Our idea is to truncate the least significant bits of floating-point variables' mantissa in software by setting those bits to 0 after every floating-point operation. We do this by inserting bitwise *AND* operations with a pre-computed mask corresponding to the size of the simulated float type in LLVM-IR code for every floating-point register. Our implementation of this approach is available on GitHub.[2]

Listing 3: Mantissa truncation: original LLVM-IR code

```
...
%7 = load float, ptr %x
%8 = load float, ptr %x
%mul = fmul float %7, %8
...
```

[2] https://github.com/TAFFO-org/LAMPSimulator.

Listing 4: Mantissa truncation: modified LLVM-IR code

```
1  ...
2  %7 = load float, ptr %x
3  %8 = load float, ptr %x
4  %res = fmul float %7, %8
5  %sim5 = bitcast float %res to i32
6  %sim6 = and i32 %sim5, -65536
7  %sim7 = bitcast i32 %sim6 to float
8  ...
```

An example of such modification is shown in Listings 3 and 4. Fixed-point formats are handled by the same float-to-fixed transformation pass used by the precision tuning framework (i.e. the *Conversion* pass in TAFFO: *TAFFO Precision Tuning* block in Fig. 3). Such simulation is a good approximation to running the programs on the hardware implementation, providing a tradeoff between the accuracy and the complexity of configuration. Libraries such as MPFR [35] that can simulate floating-point formats exactly were not used for several reasons. First, the rounding errors were found not large enough to justify the slowdown of the simulation. Second, the implementation of the library assumes modification of the source code of the program being tuned, which would introduce a manual step from the programmer, significantly reducing the value of the approach. Third, using this library would prevent the optimizations done at the LLVM-IR level, interfering with the precision-tuning process. Our method allows us to completely avoid these problems. In this approach we are taking advantage of the exponent size being unchanged between different floating-point formats we are considering. In case if there are more floating-point formats with different exponent sizes present, more sophisticated simulation approaches need to be used, such as MPFR [35] or FlexFloat [36], complicating the implementation but leaving the general approach essentially the same.

Example 3.3. The errors obtained by running the simplified simulation are:

- *IMFbfloat16×float* - $ERR = 5.56 \cdot 10^{-2}$
- *IMFbfloat16×fixed* - $ERR = 3.11 \cdot 10^{-3}$
- *IMZm×float* - $ERR = 7.52 \cdot 10^{-8}$
- *IMZm×fixed* - $ERR = 5.38 \cdot 10^{-7}$

4.4. Energy estimation

This step, corresponding to the *Energy estimator* block in Fig. 3, estimates the energy consumption of every configuration by means of Formula (3).

$$Energy = ENOP_{conf} \cdot N_{op} \tag{3}$$

$ENOP_{conf}$ is the measured average energy per floating-point operation on the $conf$ HW/SW configuration, and N_{op} is the number of floating-point operations in the program. The criteria that define a configuration ($conf$) are described in Section 4.1. The main point of this step is not to get an accurate estimate of the energy a configuration would consume, but rather establish the relative order between the configurations in terms of energy efficiency.

We collect the average energy consumption of floating-point configurations on a known mix of arithmetic operations. The advantage of this simple approach is that it is fast and it requires less data to be collected in advance, while providing accurate enough results for modeling the energy consumption of FPGA-based CPU softcores. It is possible to build a more accurate model by considering different costs per operation. This information can be collected by running a micro-benchmark for each type of arithmetic operation, and then the software characterization shall be updated with separate accounting for each instruction type (see Section 4.2). We leave the exploration of the effect of more accurate energy models for future study.

Example 4. The energy estimates obtained by applying Formula (3) are:

- *IMFbfloat16×float*
  $Energy = 23.2 \cdot 5220 = 121104$
- *IMFbfloat16×fixed*
  $Energy = 15.6 \cdot 5220 = 81432$
- *IMZm×float*
  $Energy = 187.7 \cdot 5220 = 979794$
- *IMZm×fixed*
  $Energy = 14.1 \cdot 5220 = 73602$

4.5. Area metrics

The area metrics information includes the number of flip-flops (FF), lookup tables (LUT), and digital signal processing (DSP) elements required for the FPGA implementation of a certain configuration. We optimize the area by minimizing the share of the resources the design takes on the reference FPGA architecture. This step, corresponding to the *Area metrics* block in Fig. 3, employs Formula (4) to compute the optimization target for the area.

$$AreaShare = \max_{i \in \{LUT, FF, DSP\}} \frac{param_i}{arch_i} \tag{4}$$

The separate metrics by themselves could have been used in the optimization, but we chose to reduce them to one metric to reduce the number of dimensions in the Pareto-set optimization. Indeed, all area attributes are highly correlated, known, and fixed for every HW configuration. Therefore, optimizing them individually is unlikely to yield a significantly better result.

Example 5. The area share for configurations targeting the Artix-7 100 FPGA, that features 63400 LUTs, 126800 FFs, and 240 DSP, is computed according to Formula (4) as:

- *IMFbfloat16*

$$AreaShare = \max\{\frac{10053}{63400}, \frac{7256}{126800}, \frac{4}{240}\} = 0.16$$

- *IMZm*

$$AreaShare = \max\{\frac{8342}{63400}, \frac{5761}{126800}, \frac{4}{240}\} = 0.13$$

4.6. Configuration selection

The estimates for error, energy, and area described above are used to compute the Pareto set of optimal configurations. For this step, depicted as the *Configuration selection* block in Fig. 3, we use the Linear model approach for error estimation, rather than the Static formula, due to its better accuracy, as is evidenced by the experiment described in Section 5.3. Since the configuration selection framework is agnostic of the methods used for the estimation, if more accurate estimators are available they can be plugged in without modifying the rest of the framework. This step operates as follows. First, a set of configurations are pre-selected, based on which ones satisfy the error threshold provided by the user. We then calculate the Pareto-optimal set out of the pre-selected configurations. This set represents the configurations for which any of the dimensions cannot be improved without making other dimensions worse. The same method can be applied in cases when the design selection is done for multiple applications or kernels that need to be running on the same hardware by computing the Pareto set over the features of all applications combined.

Example 6. Filtering the four hardware-software configurations by a target relative error threshold of 10^{-3} using the linear model estimate from Example 3.2 excludes the *IMFbfloat16×fixed*, which exceeds the threshold with a value of 7.10×10^{-3}, leaving the following 3 configurations:

- *IMFbfloat16×float*
  $ERR = 7.70 \cdot 10^{-4}$
  $Energy = 121104$
  $AreaShare = 0.15$
- *IMZm×float*
  $ERR = 2.27 \cdot 10^{-7}$
  $Energy = 979794$
  $AreaShare = 0.13$
- *IMZm×fixed*
  $ERR = 2.09 \cdot 10^{-6}$
  $Energy = 73602$
  $AreaShare = 0.13$

IMZm×float and *IMZm×fixed* constitute the Pareto set, since they have a combination of attributes that cannot be improved by any other configuration.

4.7. Dynamic configuration selection refinement

The configurations selected with the static rules may be suboptimal in terms of error due to the impossibility of modeling the complex interactions in the programs precisely with simple heuristics. We employ therefore the simplified simulation described in Section 4.3.3 to get a better error estimate. This step, depicted as the *Configuration refinement* block in Fig. 3, introduces the time overhead due to the compilation and running of every configuration, but since the previous step would significantly limit the design space to be explored, it can still be faster than the exhaustive search. We assume that the Pareto set from the previous step is close to the ground truth Pareto set, so we also sample some of the points in the vicinity of the statically discovered Pareto set and test them directly to see if they yield better results. We have found that we can just test the fixed-point configurations, leaving out the floating-point ones, since the floating-point accuracy is predicted well enough by the linear model estimation. The number of configurations tested in this refinement step can be adjusted according to the available time budget. The gate-level simulation, however, would still be too slow for this step. Various strategies are possible for selecting among the configurations to run. We propose taking configurations with the error estimate exceeding the target relative error and picking from them the ones having the best energy estimate. The reasoning behind this is they give the biggest benefit in case their error turns out to be acceptable, significantly improving the quality of the Pareto set. It is possible to go for less risky exploration strategies but with less expected benefit.

Example 7. Running the simple simulation on the Pareto-optimal configurations from Example 6 returns the errors previously shown in Example 3.3:

- *IMZm×float*
  $ERR = 7.52 \cdot 10^{-8}$
  $Energy = 979794$
  $AreaShare = 0.13$
- *IMZm×fixed*
  $ERR = 5.38 \cdot 10^{-7}$
  $Energy = 73602$
  $AreaShare = 0.13$

Filtering by a 10^{-3} error threshold results in the same Pareto set.

4.8. Configuration verification

This step is represented by the *Verification* block in Fig. 3. At this stage, we can verify that the selected configuration does indeed satisfy our requirements by running it on the gate-level simulator. This, however, is highly computationally- and time-intensive. This step would be evaluated only with the final selected configuration before producing the physical devices to make sure that the design performs as expected.

Example 8. A gate-level simulation of the configurations in Example 7 allows verifying they meet the specified requirements:

- *IMZm×float*
  $ERR = 7.35 \cdot 10^{-8}$
  $Energy = 922535940$
  $AreaShare = 0.13$
- *IMZm×fixed*
  $ERR = 5.38 \cdot 10^{-7}$
  $Energy = 72029000$
  $AreaShare = 0.13$

The two configurations notably perform as expected relatively to each other, i.e., the first one produces a smaller error and the second one consumes less energy.

5. Experimental evaluation

We evaluate our methodology on instances of the mixed-precision, FPGA-based softcore CPU described in Section 3.2 that can optionally support one of four different floating-point formats as well as instantiate dedicated fixed-point hardware, for a total of 10 CPU configurations. We select a set of 14 kernels from a state-of-the-art benchmark suite as their target workload, scale their inputs by 11 different scaling factors in order to stress the dynamic range, and consider 2, i.e., floating- and fixed-point, arithmetic modes, for a total of 3080 ($10 \cdot 14 \cdot 11 \cdot 2$) hardware-software configurations.

In this section, we apply our hardware-software co-design methodology to such scenario and demonstrate its effectiveness in exploring the trade-off between energy efficiency, accuracy, and resource utilization and its speedup in examining the design space by orders of magnitude compared to state-of-the-art simulation-based approaches.

5.1. Software setup

In our experimental evaluation, we made use of the Polybench/C 4.2 benchmark suite [37], which features programs written in *ANSI C* that are characterized by a large share of floating-point computations. In particular, we considered a representative set of 18 applications, encompassing BLAS routines (*gemm*, *gemver*, *gesummv*, *symm*, *syr2k*, *syrk*, and *trmm*), linear algebra kernels (*2mm*, *3mm*, *atax*, *bicg*, and *mvt*), stencils (*jacobi-1d*, *jacobi-2d*, *seidel-2d*, and *heat −3d*), and linear algebra solvers (*lu* and *ludcmp*). The applications were compiled with TAFFO[3] and LLVM 15.0 to obtain their fixed-point and floating-point versions. The whole methodology described in Section 4, also including the hardware design and verification flow, was executed on a server that features a 10-core *Intel Xeon E5-2650 v3* CPU and a 64 GB memory and that runs the Ubuntu 22.04.3 LTS operating system.

Because of the long time required to measure the energy consumption of larger applications we did not evaluate data type propagation beyond the kernel boundaries and co-optimization of multiple kernels. Although data type propagation beyond the kernel boundary has been shown beneficial [38], it can be seen as a separate technique that does not significantly affect the kernel-hardware co-optimization and it can be applied after. We refer the reader to Section 4.6 for details of how

Table 1
Software features of the PolyBench benchmark applications considered in the experimental evaluation.

Benchmark	# Annot.	Output size	#float ops
2mm	9	112	5 700
3mm	9	72	5 220
atax	7	22	2 244
bicg	7	42	2 242
gemm	7	120	6 352
gemver	14	440	4 640
gesummv	9	16	1 600
lu	6	400	21 550
ludcmp	9	20	22 390
mvt	7	40	2 080
symm	7	160	4 935
syr2k	7	256	8 872
syrk	6	256	4 632
trmm	5	160	1 805
jacobi-1d	3	15	840
jacobi-2d	4	225	18 250
seidel-2d	7	400	30 360
heat-3d	5	125	8 350

our method can be applied in case of optimization of multiple kernels sharing the same hardware.

We ran benchmarks with different scales of input to characterize how the dynamic range affects the selection of the hardware platform and the precision-tuning settings. Dynamic range is the biggest difference between floating-point and fixed point [39], and it directly affects the accuracy of number representations both in terms of available fractional bits in fixed-point formats and the unit in the last place (ulp) in floating-point formats [40]. Fixed point formats have uniform precision throughout the dynamic range and perturbations exclusively in the mantissa of floating-point formats do not change their ulp, making small perturbations to inputs inefficient at stressing the system. As such, scaling provides the worst case for the system evaluation as it is likely to introduce conditions where one or more formats are not adequate and force the system to choose a more optimal format. In our experimental evaluation, we scaled the default inputs of the applications by multiplying them by scaling factors from the set $Scales = \{1, 2, 4, 8, 16, 32, 64, 128, 256, 512, 1024\}$ and tested the selection of the Pareto-optimal set of configurations targeting relative error thresholds from the set $TargetErrThr = \{0.01, 0.001, 0.0001, 0.00001\}$.

The dynamic-range scale-independent software features of the benchmarks collected with the instrumentation described in Section 4.2 are shown in Table 1, which also lists the number of TAFFO annotations required for every benchmark.

5.2. Hardware setup

Functional validation and experimental evaluation were performed on a configurable state-of-the-art system-on-a-chip (SoC) meant for FPGA targets [30]. In particular, we considered an instance of the reference SoC that featured a 32-bit in-order RISC-V CPU, a 32-bit Wishbone bus, a 64 KB BRAM-based main memory, a user-space UART for application input and output, and the debug infrastructure to allow the communication between the host and either the prototyping board or the simulation environment.

The complete SoC was implemented employing *AMD Vivado ML 2022.2* targeting a 50 MHz clock frequency on the *Digilent Nexys 4 DDR* prototyping board, which is equipped with an *AMD Artix-7 100* (*xc7a100tcsg324-1*) FPGA, a mid-range cost-effective chip packing 63 400 look-up tables (LUT), 126 800 flip-flops (FF), 240 digital signal processing (DSP) elements, and 135 blocks of 36 kb block RAM (BRAM). To provide a fair evaluation, we implemented each one of the design

[3] https://github.com/TAFFO-org/TAFFO, commit 8d6daef

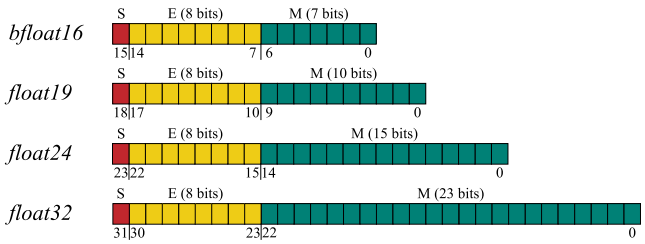

Fig. 4. Floating-point formats supported by the mixed-precision CPU instances considered in the experimental evaluation.

variants within the same SoC, employing the *Vivado* default synthesis and implementation strategies. The resource utilization was extracted from the post-implementation netlist for each SoC variant, and the resources available on an *Artix-7 100* FPGA were used as the basis to calculate the resource share of each of the solutions, as described in Section 4.5. We employed *xsim*, included in *AMD Vivado ML 2022.2*, as the gate-level simulator. Power consumption results were collected for each SoC and benchmark application as the dynamic power obtained by *Vivado Report Power* from the corresponding post-implementation simulations, and the energy consumption for an application executing on an SoC was accordingly computed as the product of the corresponding power consumption and execution time. We collected the power consumption for the whole SoC; however, in our experimental analysis, we remarkably only report values for the energy and power consumption limited to the CPU since the latter is the actual target of the optimizations brought by the proposed methodology.

Table 2 lists the configurations of the mixed-precision CPU considered in the experimental evaluation of this work, providing details on the corresponding hardware support provided for fixed-point and floating-point arithmetic operations and listing the latency of arithmetic operations. The leftmost column lists the labels assigned to each CPU throughout the rest of our discussion, while the next two columns on its right highlight whether the corresponding CPU supports (✓) or not (–) fixed-point arithmetic, and whether it supports floating-point arithmetic (✓) or does not implement an FPU (–), indicating the specific format (*bfloat16*, *float19*, *float24*, or *float32*) in the former case.

All the floating-point formats considered in this experimental evaluation, as depicted in Fig. 4, share the same 8-bit exponent width and only differ in the mantissa bit width. *bfloat16* and *float32* are widely used representations, while *float19* and *float24* are formats with an intermediate precision. We remark that the proposed methodology does not take into account any aspect related to memory organization, e.g., using smaller or larger representations for floating-point values to minimize the memory footprint or accesses.

Finally, the rightmost part of the latency of Table 2 lists the latency, expressed as the number of clock cycles, for the addition-subtraction (Add), multiplication (Mul), division (Div), and comparison (Cmp) operations in their integer, fixed-point, and floating-point arithmetic variants.

Fixed-point addition and subtraction take the same time with and without specific fixed-point hardware support, i.e., across all rows of Table 2, since both operations are executed as the corresponding integer operations. Fixed-point multiplication and division require 14 and more than 350 clock cycles when the custom Zm extension is not supported, i.e., in the top five rows of Table 2, due to the need to execute a sequence of instructions from the I and M extensions, whereas integer multiplication and division take instead 5 and 14 clock cycles, respectively. In particular, without Zm hardware support, the fixed-point multiplication requires the execution of two multiplications, two shifts, and an OR instruction, while the fixed-point division includes invoking a software-implemented routine that takes hundreds of clock cycles, severely hindering its latency. On the contrary, fixed-point multiplications and divisions executed with dedicated hardware support,

Table 2
Latency, in clock cycles, of the integer, fixed-, and floating-point operations of the considered configurations of the mixed-precision CPU.

CPU configuration	Hardware support		Operations latency											
	Fixed-point	Floating-point	Integer				Fixed-point				Floating-point			
			Add	Mul	Div	Cmp	Add	Mul	Div	Cmp	Add	Mul	Div	Cmp
IM	–	–	1	5	14	1	1	14	>350	1	Soft-float execution			
IMFbfloat16	–	✓ *bfloat16*	1	5	14	1	1	14	>350	1	4	4	8	4
IMFfloat19	–	✓ *float19*	1	5	14	1	1	14	>350	1	4	4	10	4
IMFfloat24	–	✓ *float24*	1	5	14	1	1	14	>350	1	5	5	12	4
IMFfloat32	–	✓ *float32*	1	5	14	1	1	14	>350	1	5	5	12	4
IMZm	✓	–	1	5	14	1	1	5	14	1	Soft-float execution			
IMZmFbfloat16	✓	✓ *bfloat16*	1	5	14	1	1	5	14	1	4	4	8	4
IMZmFfloat19	✓	✓ *float19*	1	5	14	1	1	5	14	1	4	4	10	4
IMZmFfloat24	✓	✓ *float24*	1	5	14	1	1	5	14	1	5	5	12	4
IMZmFfloat32	✓	✓ *float32*	1	5	14	1	1	5	14	1	5	5	12	4

Table 3
FPGA resource utilization of the considered configurations of the mixed-precision CPU.

CPU configuration	LUT	FF	DSP	BRAM
IM	7 936	5725	4	0
IMFbfloat16	10 053	7256	4	0
IMFfloat19	9 874	7281	6	0
IMFfloat24	10 245	7356	6	0
IMFfloat32	10 719	7507	10	0
IMZm	8 342	5761	4	0
IMZmFbfloat16	10 431	7288	4	0
IMZmFfloat19	10 351	7317	6	0
IMZmFfloat24	10 699	7388	6	0
IMZmFfloat32	11 192	7543	10	0

Table 4
Mean energy per floating-point operation for every CPU configuration, referring to the execution of a benchmark with a 59% mul, 30% add, 10% sub, and 1% div distribution.

CPU configuration	Fixed (nJ)	Float (nJ)
IM	18.0	187.7
IMFbfloat16	15.6	23.2
IMFfloat19	16.2	23.8
IMFfloat24	16.8	25.2
IMFfloat32	17.5	26.6
IMZm	14.1	187.7
IMZmFbfloat16	12.5	23.2
IMZmFfloat19	13.1	23.8
IMZmFfloat24	13.4	25.2
IMZmFfloat32	14.0	26.6

i.e., in the bottom five rows, take as long as the corresponding integer instructions. Comparing hardware-supported fixed- and floating-point arithmetic operations highlights faster fixed-point addition-subtractions and comparisons in the SoC instances considered in our experimental evaluation, while FPUs can execute faster or equally fast multiplications and divisions.

Table 3 lists, for each considered configuration of the mixed-precision CPU previously detailed in Table 2, the FPGA resources occupied by their implementation on an *AMD Artix-7 100* chip. The resource utilization is reported in terms of LUTs, FFs, DSPs, and BRAM blocks.

Starting from the baseline CPU, implementing the sole I and M standard RISC-V extensions, the instantiation of additional functional units correspondingly increases resource utilization. Instantiating an FPU requires up to around 2800 LUTs, 1800 FFs, and 6 DSPs, and the additional resource utilization increases as more mantissa bits are used to represent floating-point values internally to the FPU. Similarly, adding hardware fixed-point support has a smaller impact on resource utilization, with an increase of around 500 LUTs and 40 FFs. Overall, the largest configuration of the mixed-precision CPU, supporting all the I, M, Zm, and F extensions and implementing a *float32* FPU, occupies 41% LUTs, 32% FFs, and 150% DSPs more than the smallest baseline configuration, supporting the sole I and M extensions.

Table 5
Coefficients of the linear model.

Feature	Float	Fixed
Intercept	−15.651015	−12.408356
Mantissa bits (M)	−6.822522	−3.022593
Float op. count (N)	2.256770	1.667797
Variable max. value (V)	0.104946	2.085219
Output size (O)	1.803398	−1.754469

The energy per floating-point operation was collected by running benchmarks on every hardware configuration. The resulting measurements can be seen in Table 4. This data is derived by dividing the total energy consumption measured during each benchmark run by the statically-known amount of floating-point operations performed. The same applies to fixed-point configurations. Note that the number of fixed-point operations is equal to the number of floating-point ones.

The CPU configurations without an FPU running floating-point code show the worst energy efficiency (187.7 nJ) due to the software emulation of the floating-point arithmetic. The floating-point code consumes more energy per operation than the fixed-point code by about 8 nJ, regardless of the FPU configuration: 23.2–26.6 nJ per floating-point operation, saving 0.6–1.4 nJ per every level of FPU size. The fixed-point code benefits from the fixed-point hardware extension (Zm), saving 3–4 nJ in comparison to the analogous version without it. Since the fixed-point code still has its output converted to float, the availability of the FPU and its size has a small effect on the energy consumption: about 0.5 nJ per every level of FPU size. The choice to have the output converted to float is explained by the need to interface with other parts of the program or other applications altogether that may be running on the same CPU but not necessarily using the fixed point.

5.3. Experiment: Error estimation

Out of three models for error estimation described in Section 4.3 the static formula and the simple simulation can be used without modification, but the linear model needs to be trained on the training dataset. To train and verify the linear model, we divide the benchmarks into two sets: training and testing, with 7 benchmarks in the training set and 11 in the testing set. Every benchmark has various configurations with different scales, precision-tuning options, and hardware options. The test and training split was chosen randomly; *syrk, ludcmp, 2 mm, mvt, syr2k, atax, gemver* were used for training, while *gemm, bicg, symm, 3 mm, gesummv, lu, trmm, jacobi-1d, jacobi-2d, seidel-2d, heat −3d* were used for testing. The final coefficients of the linear models are given in Table 5. This shows the relative impact of the features on the relative error of floating-point and fixed-point configurations. The regularization hyper-parameter *alpha* of the *Ridge* regression was set to 0.5 for the floating-point model and 48 for the fixed-point model.

We then evaluate all methods for relative error estimation on the test set of benchmarks. The goal of all three models is to accurately predict the output relative error for benchmarks running on the various

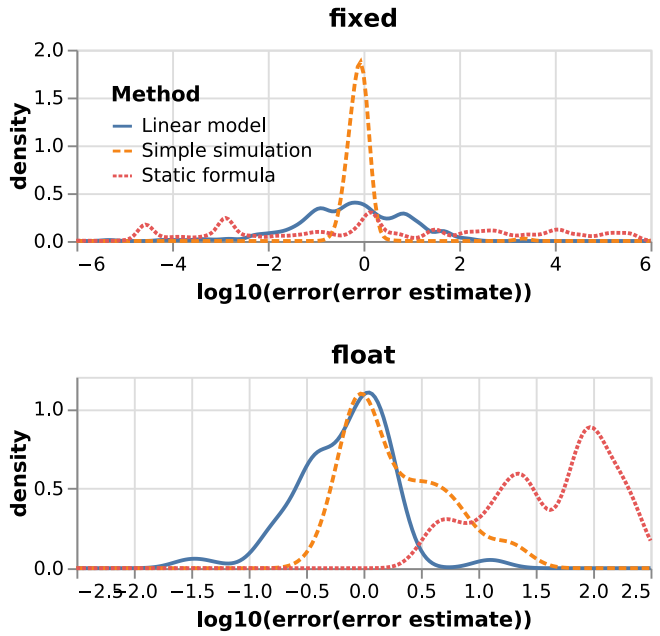

Fig. 5. The error distribution of error estimation methods. This chart demonstrates the distribution of $log_{10}(Predicted\,Error/Actual\,Error)$ for $Predicted\,Error$ obtained with three different methods: Static formula (discussed in 4.3.1), Linear model (discussed in 4.3.2) and Simple simulation (discussed in 4.3.3). The top chart demonstrates the fixed-point software configuration, while the bottom one shows the floating-point. The prediction is computed on the test dataset. For the fixed-point configuration the methods rank Simple simulation > Linear model > Static formula. The Simple simulation method dominates in accuracy, with the metric tightly distributed around the 0 mark. The Linear model gives slightly more accurate results than the Static formula. For the floating-point, the ranking is Linear model > Simple simulation > Static formula, with the Linear model giving a slightly tighter distribution around the 0 mark than the Simple simulation. The Static formula distribution is skewed significantly to the right.

Table 6
Prediction quality metrics.

Metric	Before refine	After refine (+2 conf)
True positive	77.36%	79.58%
False positive	33.19%	12.99%
False negative	22.64%	20.42%
Recall	77.36%	79.58%
Precision	69.98%	85.97%
F1 score	73.48%	82.65%
Error > Target	0.00%	1.44%

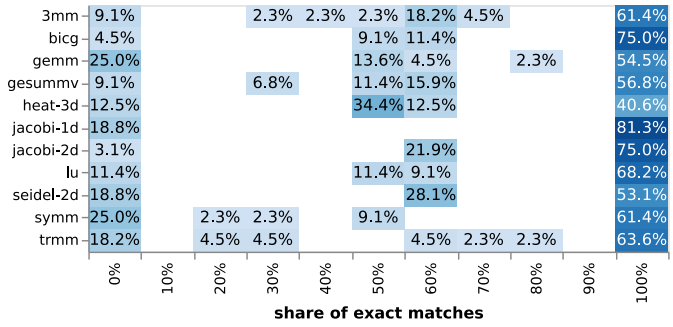

Fig. 6. Share of the exact matches in Pareto sets between prediction and the ground truth after the refinement step. The horizontal axis shows the (binned) share of the exact matches between the predicted and the actual Pareto sets. The heatmap color and numbers represent the percent of the benchmark configurations that fall in every bin. The numbers in every row add up to 100% within the rounding error. (For interpretation of the references to color in this figure legend, the reader is referred to the web version of this article.)

CPU configurations. As such, the ground truth for this evaluation is the error observed from running all configurations on the gate-level simulator described in Section 5.2. Fig. 5 shows the distribution of the error for every error estimation method for floating-point and fixed-point configurations. For fixed-point, the simplified simulation works best. This is explained by the fact that fixed-point operations are always prevalent in the instruction mix of fixed-point configurations, hence the error introduced by the floating-point-only simulation is minimal. As this estimator employs the precision tuning framework itself to handle fixed-point representations, in that respect, the profiled code behaves identically to the one effectively executed by the completed design. For floating-point, the linear model and the simple simulation provide similar accuracy, with the linear model being slightly more conservative. The static formula approach is too inaccurate for both cases. Based on these results, we decided to use the linear model and the simple simulation approaches in our HW/SW co-design methodology.

5.4. Experiment: HW/SW co-design

As the basis of our comparison, we computed the ground-truth Pareto sets for every software configuration and target error threshold discussed in Section 5.1 across all of the CPU configurations. For that, we use the target error thresholds to filter the configurations that have relative errors exceeding these thresholds. Among the filtered records we then find the Pareto set with the values of energy and area metrics that cannot be improved without worsening other metrics. Thus, for every software configuration and error requirement we obtain a set of 1 or more CPU configurations that are empirically optimal. We call that the *Ground Truth Sets*.

We made estimations for relative error and energy for every combination of software and CPU configurations as described in Sections 4.3.2 and 4.4. The error estimates were then used to filter by the error threshold, and the energy estimates used in the Pareto selection procedure together with the area metrics. We call these sets *Prediction Sets*. In addition, we used the refinement technique discussed in Section 4.7 with the addition of 2 configurations outside the *Prediction Sets* to obtain the *Refined Prediction Sets*.

We compare the *Ground Truth Sets* with the *Prediction Sets* and the *Refined Prediction Sets* in terms of the exact matches across all benchmarks using several metrics such as *true positive*, *false positive*, *false negative* rates. In addition, we calculate *Recall* (another name for true positive), *Precision*, and *F1 score* metrics. The *F1 score* in particular shows the overall balance of the model, as it penalizes selecting *too many* and *too few* configurations equally. We also track the percentage of configurations that exceed the target relative error and report it as *Error > Target*. The result is shown in Table 6.

Before the refinement step the *Prediction Sets* have about 77% *Recall*, but also have a relatively high false positive rate of 33%. The *Refined Prediction Sets* improve the *Recall* and the false negative rate by about 2%, but more importantly reduce the false positive rate by 20% leading to much improved *F1 score* at the cost of about 1.5% of configurations exceeding the target error threshold. The low false positive and the high false negative rates together with the low *Error > Target* metric evidence that the model is erring on the safer, conservative side. For an automatic HW/SW co-design tool this is a desired quality as finding slightly suboptimal solutions is usually less of a problem than not satisfying the accuracy requirements. From this point forward we only analyze the *Refined Prediction Sets* as they show better overall performance.

Then we analyze the distribution of share of the *Refined Prediction Sets* that match the P_i% of the *Ground Truth Sets* with P_i representing intervals $P_i = [0\%, 10\%), [10\%, 20\%), \ldots, [90\%, 100\%), [100\%, +\inf)$. This distribution is shown in Fig. 6. We can see that for all benchmarks more than 40% of configurations match the full *Ground Truth Set* exactly.

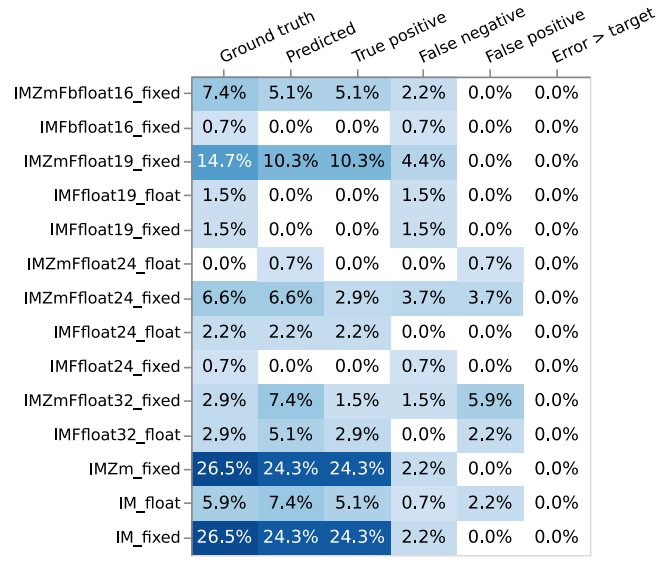

Fig. 7. *3mm* benchmark: distribution of HW/SW configurations in the predicted and ground truth Pareto sets after the refinement step. Other columns are true positive, false negative, false positive, and Error bigger than the target error threshold. The values are normalized to the sum of Ground truth.

Moreover, 70%–95% of configurations match their *Ground Truth Set* by more than 50%. Only some benchmarks have a relevant share of configurations (18%–25%) that match the *Ground Truth Set* less than 10%.

We then drill down on the distribution of the *Ground Truth Sets* and the *Refined Prediction Sets* by CPU configuration and look at the distribution of the CPU configurations across all Pareto sets for all configurations for the *3mm* benchmark, as depicted in Fig. 7. The latter reports some of the same metrics as Table 6, with the percentage values normalized to the total sum size of the *Ground Truth Sets*. We can see that the prediction closely matches the *Ground Truth Sets*, especially on the *fixed* configurations. With the *float* configurations, it tends to prefer the bigger FPU sizes. It correctly identifies the biggest group — the two CPUs without an FPU, one with and one without *fixed* SW configuration. It has a good match rate at the lower FPU sizes as well for the *fixed* configurations. The false negative rate shows that some CPU configurations are not identified in prediction, for example, *IMFbfloat16_fixed*, *IMFfloat19_fixed*, *IMFfloat19_float*, *IMFfloat24_fixed*. Those configurations though do not make up a big percentage of the *Ground Truth Sets*.

The exact matches between Pareto sets of the prediction and the ground truth do not fully characterize the quality of the solution as some sub-optimal configurations may be close enough to the configurations in the ground truth Pareto set to be considered usable. To quantify this we computed the *Euclidean distance* from every configuration in the *Refined Prediction Set* to every configuration in the *Ground Truth Set*. Since every dimension (relative error, energy, LUT, FF, DSP) has different units we scale them by subtracting the mean and dividing by the standard deviation before calculating the Euclidean Distance to exclude the scale effect on the distance. Then for every configuration P_i in the *Refined Prediction Set* we find the configuration from the *Ground Truth Set* G_j with the minimal distance to it and calculate the ratio $R_d = \frac{P_{i,d}}{G_{j,d}}$, where $d \in \{error, energy, LUT, FF, DSP\}$. We use the actual measurements of the predicted configurations instead of estimated because we are evaluating the real performance of the selected configuration. The resulting distribution of ratios of every dimension is shown in Fig. 8. Overall, for every dimension the absolute majority of the configurations of the *Refined Prediction Set* lies very close to the corresponding configuration in the *Ground Truth Set*: $R_{error} \in [10^0, 10^{0.02})$ for 83.3–98.2% of configurations, $R_{energy} \in [100\%, 105\%)$ for 89.8–99.2%, $R_{LUT} \in [100\%, 105\%)$ for 78.9–97.8%, $R_{FF} \in [100\%, 105\%)$ for 92.4%–100%, $R_{DSP} \in [100\%, 105\%)$ for 90.8–98.5%.

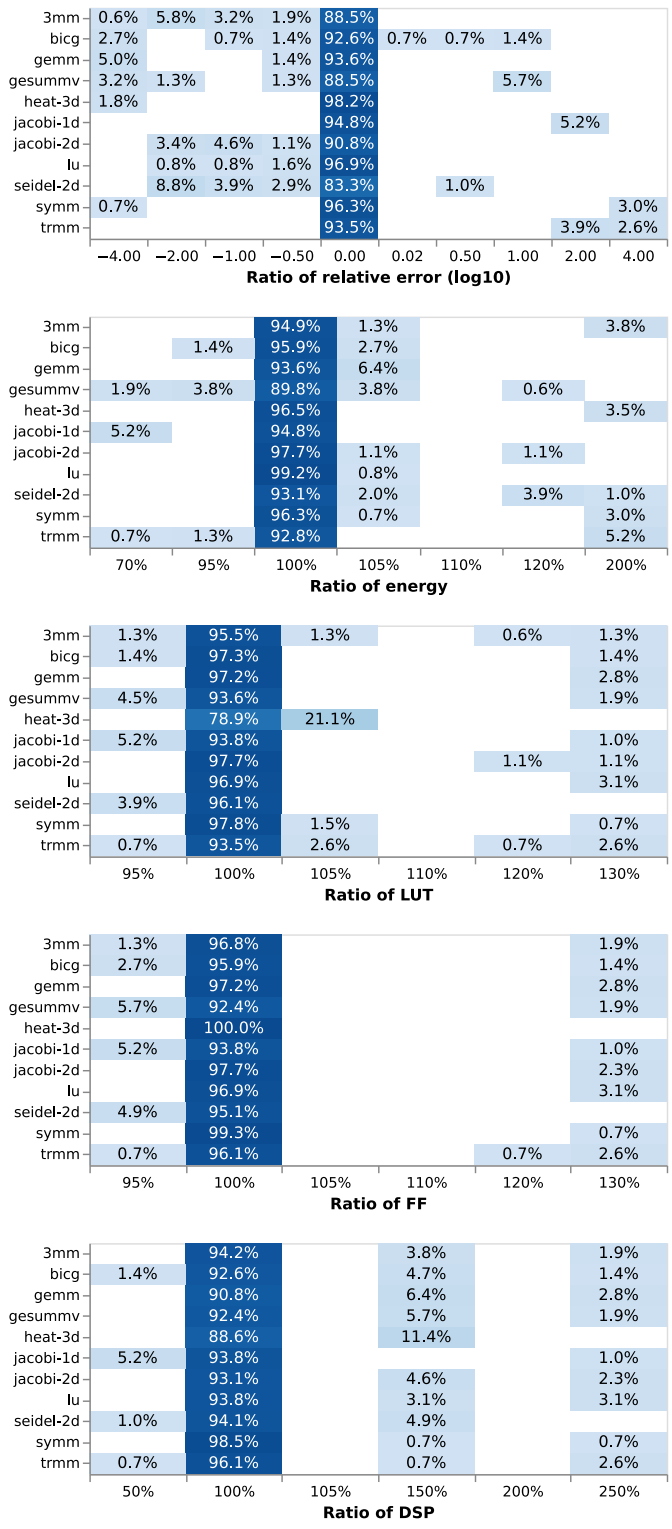

Fig. 8. Distribution of ratios between the target dimensions of configurations from *Refined Prediction Set* to the closest (z-normalized Euclidean distance) configuration from the *Ground Truth Set*. Rows are benchmarks and columns are bins of ratios for the respective attribute. Column labels signify the lower bound of the bin (included); the next label to the right signifies the end of the bin (not included). The last column label signifies the bin from its value to infinity.

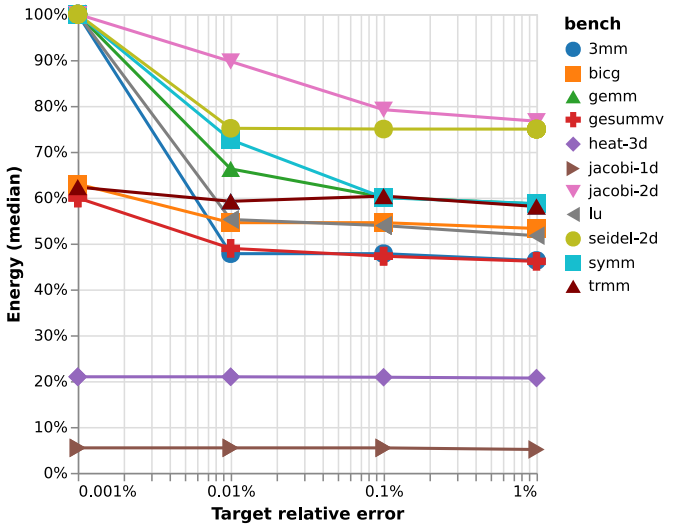

Fig. 9. Energy/error tradeoff. This chart shows the median energy consumption among all the configurations in the predicted Pareto set by benchmark, normalized to baseline HW/SW configuration (*IMFfloat32_float*) and shown in %. Irrespective of the error threshold, energy consumption for *jacobi-1d* is around 5% and for *heat −3d* is around 20%. At the 0.001% target relative error *3mm*, *lu*, *gemm*, *symm*, *seidel-2d*, *jacobi-2d* are not consuming less energy, but with increasing the error threshold the energy consumption drops below 80%. For *bicg*, *trmm*, and *gesummv* even at 0.001% target relative error the energy consumption is less than 65% with further 5%–15% energy decrease as the error threshold increases. At the 1% target relative error, the energy consumption is less than 50% for *3mm* and *gesummv*, less than 55% for *bicg* and *lu*, and less than 60% for *trmm*, *gemm* and *symm*.

For the error chart, we can see that there are some configurations in the *Refined Prediction Set* with lower error than the corresponding configuration from the *Ground Truth Set*. This shows that the selection approach is conservative in terms of error and does not always fully utilize the error budget. On the energy, LUT, FF, and DSP charts we can see the opposite: some configurations with higher values than 100%, which is the result of selecting non-optimal configurations. The small amount of values lower than 100% corresponds to the cases when the overall non-optimal selected configuration is slightly better in one or more dimensions than the optimal.

5.5. Optimization outcomes

The end goal of the HW/SW co-design approach we describe is the optimization of the energy consumption under the relaxed requirements for the accuracy of the output. As such, we compare the energy consumption of the selected configurations with regard to the default choice (*IMFfloat32_float*). Fig. 9 shows the median energy reduction depending on the target error threshold. The two outlier benchmarks, *heat −3d* and *jacobi-1d*, are particularly well-suited for precision tuning and show significant energy savings of around 80% and 95% respectively. This is explained by both applications using a small number of fractional bits allowing them to run correctly at the smallest and most energy-efficient hardware configurations. For the majority of the applications tested the benefit is not so exceptional but is still significant. As expected, the bigger the error allowance the more energy can be saved. Allowing for 0.01% relative error saves more than 50% energy for some benchmarks. The selected configurations introduce at most 4% overhead in terms of the area metrics (LUT, FF, DSP) in comparison to the default choice.

5.6. Optimization time

Traditional state-of-the-art approaches require executing RTL simulations, which are expensive in terms of time and computational resources. In particular, given a new application to be executed on the target computing platform, it is necessary to simulate its execution for each of the HW/SW configurations through a post-implementation RTL simulation in order to collect accurate information concerning the energy and power consumption. For instance, identifying an optimal configuration for the execution of a new application, in our experimental scenario with 10 different CPU instances and 2 software versions, i.e., *fixed* and float, for a total of 20 HW/SW configurations, requires therefore the execution of 20 gate-level simulations that, for the considered target applications, take on average 45 min per each.

Conversely, the proposed methodology avoids the need for such lengthy RTL simulations. Instead, it leverages simplified models to estimate the error and the energy consumption and delivers the architecture that is expected to minimize them and the area metrics. For instance, identifying an optimal configuration for the execution of a new application, in our experimental scenario with 20 HW/SW configurations, requires therefore the execution of 20 simplified model error estimations, each of whom takes 1.1 s per one configuration, including instrumentation, compilation, and running. The proposed methodology provides therefore a speedup in the optimization time of around 2500× on average compared to state-of-the-art approaches based on RTL simulation.

Moreover, with larger programs, gate-level simulation quickly becomes infeasible, making estimation methods the only viable option. Consequently, the proposed solution is an attractive alternative to direct gate-level simulation, saving time and programmer effort in designing energy-efficient approximate applications by optimizing both software and hardware.

We also note that, while gate-level simulations would be required to provide accurate energy and power consumption metrics for a target hardware architecture without applying the methodology in this manuscript, faster behavioral simulations cannot provide power-related metrics. Additionally, cycle-level architectural simulators, such as gem5, would need a power model for the target CPU, which is complex to obtain for a new architecture.

6. Conclusions

In this paper, we introduced a hardware-software co-design methodology, and its supporting design automation tools, to explore the design space of mixed-precision systems leveraging a configurable, softcore mixed-precision CPU meant for FPGA targets and a precision tuning compiler framework.

We proposed two novel methods to estimate the error induced by precision tuning choices that provided an improvement over the static methods currently employed in precision tuning frameworks, and we introduced a multi-objective co-design methodology that employs successive refinements of the estimated Pareto set to reduce the exploration time, by avoiding the need to perform costly gate-level simulations, while retaining a very high accuracy (recall over 79% and precision over 85%).

The proposed methodology was validated by employing gate-level simulations as the ground truth. Considering all benchmarks, over 54% of exact matches were found for the Pareto set, and less than 30% of the cases had less than 40% exact matches. The chosen configurations that were not the exact matches for the Pareto set still lie close to it, with more than 75% of configurations having the error, energy, and area metrics matching the corresponding metrics in the Pareto set within a 5% range. The implemented tool takes only an average of slightly over 1 s to perform the design space exploration, while the same exploration performed using the gate-level simulation takes an average of 45 min, corresponding to a speedup of around 2500×. Moreover, the solutions identified by applying the proposed methodology with 1% allowed error threshold reduced the energy-to-solution by up to 20% in the worst case and up to 95% in the best case on the tested benchmarks.

Future developments involve the application of our framework to more complex design space exploration techniques and extending it to support a broader set of precision tuning options.

CRediT authorship contribution statement

Lev Denisov: Writing – review & editing, Writing – original draft, Visualization, Software, Methodology, Investigation, Data curation, Conceptualization. **Andrea Galimberti:** Writing – review & editing, Writing – original draft, Validation, Software, Methodology, Investigation, Conceptualization. **Daniele Cattaneo:** Writing – review & editing, Writing – original draft, Software, Conceptualization. **Giovanni Agosta:** Writing – review & editing, Supervision, Project administration, Funding acquisition, Conceptualization. **Davide Zoni:** Writing – review & editing, Supervision, Project administration, Funding acquisition, Conceptualization.

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.

Data availability

Data will be made available on request.

Acknowledgments

This work is supported by the MSC-ITN grant agreement No. 956090 (APROPOS: Approximate Computing for Power and Energy Optimization, http://www.apropos-itn.eu/) and by the European Union's Chips Joint Undertaking program under grant agreement No. 101112274 (ISOLDE: High Performance, Safe, Secure, Open-Source Leveraged RISC-V Domain-Specific Ecosystems, https://www.isolde-project.eu/).

References

[1] M. Makni, M. Baklouti, S. Niar, M.W. Jmal, M. Abid, A comparison and performance evaluation of FPGA soft-cores for embedded multi-core systems, in: 2016 11th International Design & Test Symposium, IDT, IEEE, 2016, http://dx.doi.org/10.1109/idt.2016.7843032.

[2] J.-M. Gorius, S. Rokicki, S. Derrien, Design exploration of RISC-V soft-cores through speculative high-level synthesis, in: 2022 International Conference on Field-Programmable Technology, ICFPT, IEEE, 2022, http://dx.doi.org/10.1109/icfpt56656.2022.9974478.

[3] T. Kuwahara, S. Fujita, Y. Sato, Y. Sibuya, A. Pala, H. Tomio, Y. Murata, Y. Sakamoto, On-board computers for micro-satellites, Trans. Jpn Soc. Aeronaut. Space Sci. Aerosp. Technol. Jpn 19 (4) (2021) 485–492, http://dx.doi.org/10.2322/tastj.19.485.

[4] M. Vousden, J. Morris, G. McLachlan Bragg, J. Beaumont, A. Rafiev, W. Luk, D. Thomas, A. Brown, Event-based high throughput computing: A series of case studies on a massively parallel softcore machine, IET Comput. Digit. Tech. 17 (1) (2022) 29–42, http://dx.doi.org/10.1049/cdt2.12051.

[5] E. Taka, G. Lentaris, D. Soudris, Improving the performance of RISC-V softcores on FPGA by exploiting PVT variability and DVFS, in: 2022 IEEE International Symposium on Circuits and Systems, ISCAS, IEEE, 2022, http://dx.doi.org/10.1109/iscas48785.2022.9937320.

[6] N. Brown, M. Jamieson, J.K.L. Lee, Experiences of running an HPC RISC-V testbed, 2023, http://dx.doi.org/10.48550/ARXIV.2305.00512, arXiv:2305.00512.

[7] A. Dörflinger, M. Albers, B. Kleinbeck, Y. Guan, H. Michalik, R. Klink, C. Blochwitz, A. Nechi, M. Berekovic, A comparative survey of open-source application-class RISC-V processor implementations, in: Proceedings of the 18th ACM International Conference on Computing Frontiers, CF '21, ACM, 2021, http://dx.doi.org/10.1145/3457388.3458657.

[8] E. Cui, T. Li, Q. Wei, RISC-V instruction set architecture extensions: A survey, IEEE Access 11 (2023) 24696–24711, http://dx.doi.org/10.1109/ACCESS.2023.3246491.

[9] S. Kalapothas, M. Galetakis, G. Flamis, F. Plessas, P. Kitsos, A survey on RISC-V-based machine learning ecosystem, Information 14 (2) (2023) http://dx.doi.org/10.3390/info14020064, URL https://www.mdpi.com/2078-2489/14/2/64.

[10] S. Cherubin, G. Agosta, Tools for reduced precision computation: a survey, ACM Comput. Surv. 53 (2) (2020) http://dx.doi.org/10.1145/3381039.

[11] P. Klavík, A.C.I. Malossi, C. Bekas, A. Curioni, Changing computing paradigms towards power efficiency, Phil. Trans. R. Soc. A 372 (2018) (2014) 20130278.

[12] S. Kumar, R. Buyya, Green cloud computing and environmental sustainability, in: Harnessing Green It, John Wiley & Sons, Ltd, 2012, pp. 315–339, http://dx.doi.org/10.1002/9781118305393.ch16, Ch. 16, arXiv:https://onlinelibrary.wiley.com/doi/pdf/10.1002/9781118305393.ch16, URL https://onlinelibrary.wiley.com/doi/abs/10.1002/9781118305393.ch16.

[13] G. Tagliavini, S. Mach, D. Rossi, A. Marongiu, L. Benini, A transprecision floating-point platform for ultra-low power computing, in: DATE 2018, 2018, pp. 1051–1056, http://dx.doi.org/10.23919/DATE.2018.8342167.

[14] A. Linhares, et al., A SystemC profiling framework to improve fixed-point hardware utilization, in: 2020 33rd Symposium on Integrated Circuits and Systems Design, SBCCI, 2020, pp. 1–6, http://dx.doi.org/10.1109/SBCCI50935.2020.9189919.

[15] D. Cattaneo, et al., FixM: Code generation of fixed point mathematical functions, Sustain. Comput.: Inform. Syst. 29 (2021) http://dx.doi.org/10.1016/j.suscom.2020.100478, URL http://www.sciencedirect.com/science/article/pii/S2210537920302018.

[16] C. Inacio, D. Ombres, The DSP decision: fixed point or floating? IEEE Spectr. 33 (9) (1996) 72–74, http://dx.doi.org/10.1109/6.535397.

[17] P. Stanley-Marbell, et al., Exploiting errors for efficiency: A survey from circuits to algorithms, 2018, CoRR abs/1809.05859, arXiv:1809.05859, URL http://arxiv.org/abs/1809.05859.

[18] R. Cmar, et al., A methodology and design environment for DSP ASIC fixed-point refinement, in: DATE 1999, 1999, http://dx.doi.org/10.1109/DATE.1999.761133, URL https://doi.ieeecomputersociety.org/10.1109/DATE.1999.761133.

[19] H. Keding, et al., FRIDGE: A fixed-point design and simulation environment, in: Proceedings of the Conference on Design, Automation and Test in Europe, DATE '98, 1998, pp. 429–435.

[20] K.-I. Kum, et al., AUTOSCALER for C: an optimizing floating-point to integer C program converter for fixed-point digital signal processors, IEEE Trans. Circuits Syst. II 47 (9) (2000) 840–848, http://dx.doi.org/10.1109/82.868453.

[21] P. Belanovic, M. Rupp, Automated floating-point to fixed-point conversion with the fixify environment, in: 16th IEEE International Workshop on Rapid System Prototyping, RSP'05, 2005, pp. 172–178, http://dx.doi.org/10.1109/RSP.2005.15.

[22] E. Darulova, et al., Synthesis of fixed-point programs, in: Proceedings of the 11th ACM International Conference on Embedded Software, EMSOFT '13, 2013, pp. 22:1–22:10.

[23] D. Cattaneo, M. Chiari, G. Agosta, S. Cherubin, TAFFO: The compiler-based precision tuner, SoftwareX 20 (2022) 101238, http://dx.doi.org/10.1016/j.softx.2022.101238, URL https://www.sciencedirect.com/science/article/pii/S235271102200156X.

[24] A. Sampson, J. Bornholt, L. Ceze, Hardware-software co-design: Not just a cliché, in: SNAPL 2015, in: Leibniz International Proceedings in Informatics (LIPIcs), vol. 32, 2015, pp. 262–273, http://dx.doi.org/10.4230/LIPIcs.SNAPL.2015.262.

[25] P. Huang, et al., A hardware/software co-design methodology for adaptive approximate computing in clustering and ANN learning, IEEE Open J. Comput. Soc. 2 (2021) 38–52, http://dx.doi.org/10.1109/OJCS.2021.3051643.

[26] R.E. Moore, et al., Introduction to Interval Analysis, Siam, 2009.

[27] D. Cattaneo, et al., Embedded operating system optimization through floating to fixed point compiler transformation, in: 21st Euromicro Conference on Digital System Design, DSD, Vol. 00, 2018, pp. 172–176, http://dx.doi.org/10.1109/DSD.2018.00042.

[28] D. Cattaneo, et al., Architecture-aware precision tuning with multiple number representation systems, in: 2021 58th ACM/IEEE Design Automation Conference, DAC, 2021, pp. 673–678, http://dx.doi.org/10.1109/DAC18074.2021.9586303.

[29] S. Cherubin, et al., Dynamic precision autotuning with TAFFO, ACM Trans. Archit. Code Optim. 17 (2) (2020) http://dx.doi.org/10.1145/3388785.

[30] G. Scotti, D. Zoni, A fresh view on the microarchitectural design of FPGA-based RISC CPUs in the IoT Era, J. Low Power Electron. Appl. 9 (2019) 19, http://dx.doi.org/10.3390/jlpea9010009.

[31] D. Zoni, A. Galimberti, W. Fornaciari, An FPU design template to optimize the accuracy-efficiency-area trade-off, Sustaina. Comput.: Inform. Syst. 29 (2021) 100450, http://dx.doi.org/10.1016/j.suscom.2020.100450, URL https://www.sciencedirect.com/science/article/pii/S2210537920301761.

[32] D. Zoni, A. Galimberti, Cost-effective fixed-point hardware support for RISC-V embedded systems, J. Syst. Archit. 126 (2022) 102476, http://dx.doi.org/10.1016/j.sysarc.2022.102476, URL https://www.sciencedirect.com/science/article/pii/S1383762122000595.

[33] L. Denisov, G. Magnani, D. Cattaneo, G. Agosta, S. Cherubin, The impact of profiling versus static analysis in precision tuning, IEEE Access 12 (2024) 69475–69487, http://dx.doi.org/10.1109/access.2024.3401831.

[34] S.M. Rump, Error estimation of floating-point summation and dot product, BIT Numer. Math. 52 (2012) 201–220, http://dx.doi.org/10.1007/s10543-011-0342-4.

[35] L. Fousse, et al., MPFR: A multiple-precision binary floating-point library with correct rounding, ACM Trans. Math. Software 33 (2) (2007) http://dx.doi.org/10.1145/1236463.1236468.

[36] G. Tagliavini, A. Marongiu, L. Benini, FlexFloat: A software library for transprecision computing, IEEE Trans. Comput.-Aided Des. Integr. Circuits Syst. 39 (1) (2020) 145–156, http://dx.doi.org/10.1109/tcad.2018.2883902.

[37] L.-N. Pouchet, et al., Polybench: The polyhedral benchmark suite, 437 (2012) 1, URL: http://www.cs.ucla.edu/pouchet/software/polybench.

[38] D. Cattaneo, A. Maggioli, G. Magnani, L. Denisov, S. Yang, G. Agosta, S. Cherubin, Mixed precision in heterogeneous parallel computing platforms via delayed code analysis, in: Embedded Computer Systems: Architectures, Modeling, and Simulation, Springer Nature Switzerland, 2023, pp. 469–477, http://dx.doi.org/10.1007/978-3-031-46077-7_33.

[39] J. Eldon, C. Robertson, A floating point format for signal processing, in: ICASSP '82. IEEE International Conference on Acoustics, Speech, and Signal Processing, Institute of Electrical and Electronics Engineers, 1982, http://dx.doi.org/10.1109/icassp.1982.1171532.

[40] E. Goubault, Static Analyses of the Precision of Floating-Point Operations, Springer Berlin Heidelberg, 2001, pp. 234–259, http://dx.doi.org/10.1007/3-540-47764-0_14.