Approximate computing for high energy-efficiency in IoT applications

Geneviève Ndour

▶ To cite this version:

Geneviève Ndour. Approximate computing for high energy-efficiency in IoT applications. Other [cs.OH]. Université de Rennes, 2019. English. NNT : 2019REN1S033 . tel-02292988v2

HAL Id: tel-02292988
https://theses.hal.science/tel-02292988v2

Submitted on 22 Nov 2019

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

UNIVERSITE BRETAGNE LOIRE / MATHSTIC

UNIVERSITÉ DE RENNES 1

THESE DE DOCTORAT DE

L'UNIVERSITE DE RENNES 1
COMUE UNIVERSITE BRETAGNE LOIRE

ECOLE DOCTORALE N° 601
Mathématiques et Sciences et Technologies
de l'Information et de la Communication
Spécialité : *Informatique*

Par

« Geneviève NDOUR »

« Approximate computing for high energy-efficiency in internet-of-things applications »

Thèse présentée et soutenue à Grenoble, le 17 juillet 2019
Unités de recherche : CEA LETI Grenoble, IRISA INRIA Lannion

Rapporteurs avant soutenance :

Florent DE DINECHIN	Professeur à l'INSA Lyon, CITI
Lionel TORRES	Professeur à l'Université de Monpellier, LIRMM

Composition du Jury :

Président:	Olivier SENTIEYS	Professeur à l'Université Rennes 1, IRISA INRIA
Examinateurs:	Alberto BOSIO	Professeur à l'Ecole Centrale de Lyon, INL
	Olivier SENTIEYS	Professeur à l'Université Rennes 1, IRISA INRIA
	Florent DE DINECHIN	Professeur à l'INSA Lyon, CITI
	Lionel TORRES	Professeur à l'Université de Monpellier, LIRMM
Dir. de thèse:	Arnaud TISSERAND	Directeur de Recherche CNRS, Lab-STICC Lorient
Co-dir. de thèse:	Anca MOLNOS	Ingénieur de Recherche, CEA LETI Grenoble

ACKNOWLEDGEMENT

J'adresse mes remerciements à mon directeur de thèse Arnaud TISSERAND et à mon encadrante Anca MOLNOS pour leurs conseils sur toutes les décisions prises pour la réalisation des travaux de recherche et pour mes démarches personnelles. Merci également à Yves DURAND et à Edith BEIGNE pour leur soutien dans l'encadrement de la thèse.

Mes remerciements vont aussi aux rapporteurs Lionel TORRES et Florent DE DINECHIN pour la relecture de la thèse et à tous les membres du jury qui ont accepté d'évaluer mon travail.

Je remercie également Tiago TREVISAN-JOST pour l'implémentation du simulateur RISC-V et les collègues du DACLE/LISAN pour l'estimation du modèle d'énergie. Ces outils m'ont permis de valider les idées développées durant la thèse par une série d'expériences.

J'adresse toute ma reconnaissance à mes collègues du LIALP pour l'ambiance de travail qui constituait une source de motivation tout au long de mon stage de fin d'études et de ma thèse. Merci au chef de labo Vincent OLIVE pour son accueil.

Un grand merci aux personnes que j'apprécie beaucoup et qui ont constitué ma seconde famille : Adja (my bestie), Roxana, Maha, Sanaa, Oumayma et les membres de la chorale Sainte Cécile de Grenoble, pour leur amitié et leur aide (e.g. relecture d'articles, pot de thèse, conseils, moments de détente entre la rédaction de deux chapitres de thèse).

Je ne pourrai finir sans exprimer toute ma gratitude à ma famille. Merci à mes parents Palé TOURÉ et Robert NDOUR pour leur soutien moral et matériel ainsi que leur confiance indéfectible dans mes choix. Merci à mes frères : Djibril, Jean et à ma soeur Coumba pour leur soutien.

Au terme de ce parcours, je remercie enfin mon mari Justino LOPES pour son soutien et son attention qui m'ont accompagnée tout au long de ces années, dans les bons et les mauvais moments. Merci d'avoir fait la thèse avec moi et d'avoir accepté de différer beaucoup de nos projets personnels pour achever cette thèse dans les meilleures conditions.

TABLE OF CONTENTS

1 Introduction **9**

1.1 Context . . . 9
1.2 Problem statement . . . 12
1.3 Contributions . . . 13
1.4 Thesis organization . . . 15

2 State of the art **17**

2.1 Numerical data representation . . . 19
2.1.1 Floating-point . . . 19
2.1.2 Fixed-point . . . 20
2.2 Algorithmic approximations . . . 23
2.3 Hardware blocks for approximate computing . . . 25
2.3.1 Fixed width adders and multipliers . . . 26
2.3.2 Variable width adders and multipliers . . . 27
2.4 Software for approximate computing: programming, compiler and runtime support . . . 28
2.5 Benchmark applications and quality metrics . . . 31
2.6 RISC-V processor . . . 37
2.6.1 RV32I instruction formats in our work . . . 37
2.6.2 RISC-V base opcodes map . . . 38
2.6.3 RISC-V architecture . . . 39
2.7 Energy models . . . 40
2.8 Chapter summary . . . 42

3 RISC-V processor extended with reduced width units **45**

3.1 Extended RISC-V ISA with reduced width instructions . . . 46
3.2 Architecture of the extended RISC-V . . . 50
3.3 Energy model per instruction class . . . 51
3.3.1 Energy model for full width instructions . . . 51

3.3.2 Energy model for reduced width instructions 51
3.4 Experimental environment 55
3.4.1 Programming support for our extended RISC-V 55
3.4.2 Instrumentation tool 59
3.5 Chapter summary 62

4 Evaluation of reduced width units on applications 63
4.1 Methodology 64
4.2 Impact of the fixed-point conversion on the applications output quality 65
4.2.1 Jmeint application 65
4.2.2 Sobel filter application 66
4.2.3 Forwardk2j application 69
4.3 Impact of reduced width units for addition and multiplication on applications energy consumption 70
4.3.1 Instruction breakdown 71
4.3.2 Energy evaluation 75
4.4 Impact of the reduced width units for both computations and memory accesses on energy consumption and output quality of applications 80
4.4.1 Instruction breakdown 80
4.4.2 Output quality evaluation 82
4.4.3 Output quality *vs* energy trade-off study 85
4.5 Chapter summary 89

5 Global energy model with software and architecture parameters 91
5.1 Global energy model 92
5.1.1 Notations 92
5.1.2 Energy reduction 94
5.2 Case study 1: impact of software parameters 95
5.2.1 Width estimation for a given energy reduction 95
5.2.2 Energy reduction estimation for a given width 97
5.3 Case study 2: impact of hardware units 99
5.4 Chapter summary 102

6 Conclusion and future work 103

Publications **120**

Bibliography **129**

CHAPTER 1

INTRODUCTION

1.1 Context

Reducing *energy consumption* is crucial for embedded computing and to deploy new applications as highlighted, for instance, by the power challenge stated in the International Roadmap for Devices and Systems [37]. This reduction is particularly important for the devices of the *Internet of Things* (IoT) since they are often battery powered and may harvest energy in their immediate environment. Such devices have to be designed for being powered-up during many years. Hence every opportunity for *reducing* their energy consumption should be taken into account.

The IoT is a *global network of numerous devices* including *sensors* (*e.g.* smoke detectors, microphones, antennas, light detectors), *actuators* (*e.g.* motors, speakers, lights) and *computers* (*e.g.* microcontrollers, data loggers, small multicore processors). These devices are dispersed into the environment or embedded in everyday objects (*e.g.* smartphones, tablets, connected watches). They exchange various types of informations from a few bits to larger data with possible connections to the cloud. See [43] for more details. IoT devices are used in many applications from various domains such as building and home automation, emergency notification systems, transportation systems, bio-medical systems, gaming and entertainment applications. Computations in IoT devices have to comply with strong resource constraints such as power consumption (due to battery limits), silicon area (for reducing fabrication costs) and also timings (to ensure service quality). Reducing the energy consumption during the global design of IoT services is a major challenge to increase the battery life of the devices.

Recently many applications and systems based on various *approximation* solutions have emerged. Some approximations can be applied due to the natural robustness of the applications or the algorithms to small errors. IoT devices collect data acquired by sensors in their environment, process and aggregate those

data into compact messages sent to a higher-level system. These sensors data are subject to inherent variations and noises (*e.g.* measurement noise(s)). For instance, small differences for a few pixels over an image do not change its informational content. Processing all of those data with the maximal precision (*i.e.* width of the operands in operators) or with the maximal intended accuracy (*e.g.* algorithms with the higher quality) allowed by the circuit can be wasteful. For instance, temperature data for home automation may not require a large dynamic and accuracy.

Approximate computing is a field that explores various methods to reduce some computation costs, such as power consumption, execution time or silicon area, by allowing, ideally small, degradations during intermediate computations without compromising the quality of the final result [92, 12, 53]. Examples of error-tolerant applications amenable to approximate computing are: signal and image recognition, mining, fuzzy search, lossy compression, multimedia, data analytics, *etc*. Approximate computing evolved along three main directions: hardware, applications, and analysis.

First, on the hardware side, the literature advanced from switches and gates [62, 64], towards arithmetic operators [35, 61], and finally to dedicated accelerators [55, 33]. Many types of approximate hardware units are proposed, see [18] for a complete study. Whereas some computation kernels can benefit from hardware accelerators, another direction is to evaluate the interest of *integrating approximate hardware units into a general purpose embedded processor*. Using this type of flexible solution, multiple applications and kernels could potentially benefit from these approximate units, without the need to build a specific accelerator for only one kernel. Therefore one of the main question in this thesis is: *would IoT applications benefit from a small general purpose processor containing functional units with a reduced width?*

Second, on the application side, several types of work are proposed. For instance, algorithmic modifications are proposed to reduce the number of operations required to solve a problem by skipping loop iterations or instructions [17, 76]. These modifications lead to good initial energy reductions. In this work, we consider a reference implementation of an algorithm and we investigate how to reduce the energy consumption further using an embedded processor equipped with reduced width units. Furthermore, new programming models and data-types

are proposed to express approximations in a high-level language [74, 73]. Compiler support for approximate computing targets either code generation for specific programming models [74, 68], or introduce static approximations in a conventional source code [67, 14]. Methods for runtime control are also implemented to manage the output quality and the energy consumption during the execution of a program [39, 94].

Some programming and compiler methods can be beneficial to an embedded processor with reduced width instructions. For this purpose, we build a set of tools around the RISC-V environment [10] for program annotation, compilation and simulation. We aim to enable fast exploration of applications and kernels, without necessarily claiming novelty at this level. Future work will explore which programming support and compiler level optimizations are more suitable for this task.

Third, to evaluate the impact of approximations in complete applications, a growing number of works deal with methods and tools for analyzing errors in some computation kernels [72, 65]. This type of approaches may also involve algorithmic changes. Some formal error-analysis methods are proposed, however intensive simulations are still required for large workloads and realistic applications [21]. As this is not the main topic of this thesis, we use simulations to evaluate the output quality for various applications executed on our modified processor.

Energy consumption estimations and models are important to evaluate some trade-offs between the output quality and the actual energy reduction, although this topic is not central to approximate computing. Complete and accurate energy models for processor cores and memories are important but they are hard to obtain from the literature. The thesis was carried out in CEA Leti where we have an internal, private, implementation of a RISC (reduced instruction set computer) 32-bit processor in a test chip. We used our internal data from the test chip measurements integrated into some power models from the literature [61] to construct our energy model for our processor at the instruction level.

Numerous low-power embedded processors do not include floating-point units because the hardware implementation of a floating-point support requires a higher silicon area and power consumption than fixed-point or integer ones. A recent work by Barrois *et al.* [18] show that, for some particular workloads such as K-means, floating-point operations on a reduced width (*e.g.* 8-bit) are more efficient than

fixed-point solutions and with larger width (*e.g.* 16-bit) the fixed-point operations are more efficient than the floating-point operations. As we aim to study applications running on a general embedded processor, we stick to conventional cores with integer or fixed-point units.

1.2 Problem statement

As introduced above, the main goal of this thesis is to determine whether IoT applications would benefit from a general purpose processor core equipped with reduced width units for approximate computing. This general question implies to answer several more detailed questions stated below.

In the current literature, the evaluation of approximate arithmetic operators in general, and for reduced width ones in particular, is only performed for stand-alone units (*i.e.* not embedded in a processor running a complete application). In this stand-alone context, approximate operators can lead to important energy reductions, for instance up to 58% in [35]. But, in complete applications running in a general purpose processor not all operations can be approximated. Subsequently, one question in our more complete context is: *how much global energy reduction can be obtained on complete applications with such approximate units embedded in a processor?*

In approximate computing, most of the functional units studied in the state of the art are adders and multipliers. However the energy evaluation of full width (*e.g.* 32 bits) arithmetic and data-memory operations in [54] and [82] indicates that data-memory operations may consume more than 2 times the energy of arithmetic ones in current circuits technologies. Hence we investigate the extension of the reduced width principle to data-memory units. Here another question is: *to which extent both approximate arithmetic and data-memory units impact the energy consumption and the result quality of a given application executed in our processor?*

Finally, various optimizations can be performed on the hardware or/and on the software parts of a complete system leading to very different impacts on the global energy reduction. Not all application parts are amenable to approximations. Currently, there is a lack of general models providing an early insight into the global energy reduction offered by approximate computing methods (in software or/and in hardware). It would be interesting to have simple and relevant models such as

the Amdahl's law for speedup evaluation in parallel computers. For a given a set of applications, another question is: *when and where implementing reduced width units is worthwhile?* More precisely, *which processor units impact the most the energy consumption, and can they benefit from reduced width approximations in hardware?* Similarly on the software side, another question is: *what software characteristics impact the most the global power consumption?* For example, the ratio between memory and computation operations may be a key factor to estimate the power gain when using reduced width units.

1.3 Contributions

Our first contribution is the evaluation of some trade-offs between the application output quality and the energy consumed by an embedded general purpose processor extended with approximate units. We target a platform composed of a RISC-V processor core [10] coupled with a data and an instruction memory. The RISC-V is an open source processor which allows us to easily extend compilation and simulation tools needed for our exploration. The RISC-V supports instruction set architecture (ISA) extensions. Then, we extend the RISC-V with approximate units where the width is reduced. In these reduced width units, the operations, for both computations and data-memory accesses, are performed on a given number of most significant bits configurable at runtime.

We propose a set of annotations and an instrumentation tool to ease our experimental investigations. During an internship, Tiago Trevisan Jost implemented a compiler support to handle *pragmas* added in the source code and he extends the RISC-V simulator [11] with profiling tools that return statistics including the number of executed instructions of a given type [81]. This work was put in perspective and integrated with an energy model proposed in this thesis. We construct an energy model for each reduced width instruction and we integrated it in the tool-chain to perform application-level energy consumption estimations.

After testing and validating the extended RISC-V platform, we evaluate the impact of the reduced width integer units on some fixed-point applications. Our objective is to study the trade-offs between the estimated energy reduction and the application quality reported during intensive simulations, with various reduced widths for the units. In this evaluation, we first study the impact of common reduced

arithmetic operators (integer addition and multiplication in our processor). Then we extend the investigation to both computations and data memory accesses with various reduced widths. Our evaluation is performed on a selection of applications from the Axbench benchmark suite [93]: jmeint, Sobel filter and forwardk2j.

We also deal with the impact of the conversion from the floating-point representation into the fixed-point one on the output quality for the tested applications. Most of the benchmarks proposed in the state of the art are implemented in floating-point. Then, we have to convert them into fixed-point for our processor. Using several errors metrics, we evaluate the quality degradation due to this conversion for the selected benchmarks. Our experiments suggest that these applications are suitable for fixed-point computations, the error compared to the initial floating-point solution is acceptable (less than 0.1%) for configured widths from 16 bits and above.

Our results show an energy reduction, for the tested applications executed with reduced width units for both computations and data-memory accesses, improved up to 14%, compared to the reduction obtained using only reduced width units for addition and multiplication. This improvement is due to the fact that all tested applications contain other types of instructions than addition and multiplication, *e.g.* numerous memory accesses, `load` and `store`. Extending the reduced width principle to the data-memory unit decreases a lot the global energy consumption. We can conclude that a general purpose processor extended with reduced width units must integrate both computations and data-memory approximate units. Using approximate units only for computations lead a very small power gain.

Our second contribution is a global energy model for both hardware and software designers to have an early insight into the application-level energy consumption. Our model includes both software parameters and hardware architecture ones. The software parameters are the percentages of reduced width operations for both computations and data-memory accesses and the width required for some target output quality. The hardware parameters are the widths of the units and the energy reduction obtained for each type of stand-alone approximate unit. Our global energy model is inspired from Amdahl's law. In parallel computing, the Amdahl's law evaluates the speedup of an application executed in a parallel computer. It clearly shows that both the actual number of processors and the ratio of the parallel and the sequential parts in the program are keys elements. Similarly,

in approximate computing for complete applications (*i.e.* not only small kernels), our model as well as our experimental results show that the energy consumption depends on both the degree of approximation (*i.e.* the configured width in the units in our study) and on the proportion of approximate instructions in a complete program.

1.4 Thesis organization

The manuscript is organized in five main chapters as follows.

The **Chapter 2** presents the state of the art. We first discuss the closest related work to the thesis contributions. Second we present several quality metrics and applications in various domains to evaluate the tested approximate computing methods. Then we describe the RISC-V processor targeted in our study. Finally a summary that leads to the thesis contributions is presented.

The **Chapter 3** describes the architecture of the RISC-V processor extended with reduced width units. We first describe the extensions for reduced width computations and data-memory units. Then we present our energy model used to estimate the energy consumption of both full width and reduced width instructions.

The **Chapter 4** investigates the energy *vs* output quality trade-offs of applications executed on our RISC-V extended with reduced width units. At first the tested applications are converted from the floating-point to the fixed-point representation to study the impact of this conversion on the output quality. Then we study the impact of reduced width computations and data-memory accesses on the output quality and on the global energy consumption.

The **Chapter 5** presents a global energy model that includes both software parameters and hardware architecture ones. At first we present a generic energy model. Then we use our global energy model on several applications and we show how a software and a hardware designer can estimate the impact of some optimizations on the global energy reduction.

The **Chapter 6** concludes the thesis. At first the main contributions are summarized, then some future work is proposed.

CHAPTER 2

STATE OF THE ART

This chapter reviews the literature in the domain of approximate computing. Approximate computing is a field that explores methods to trade computation cost (in terms of execution time, power consumption, or chip area) with degradations in the quality of the computation result. Such degradations should be minimal or acceptable for the application concerned. The means to realize this trade-off are multiple: changing the algorithm, skipping operations, reducing the arithmetic precision, embedding memory blocks that have a non-negligible probability to lose their state, etc. As approximate computing is not yet a well established field, the terminology is not yet widely adopted. The idea of accepting errors in operations is present in many domains, and it may be referred to as:

- *Inexact computing*: “designing unreliable hardware and computing systems that are useful for unreliable computing elements while garnering resource savings in return” [62]. Palem, Enz and their collaborators explore the energy saving limits of what they call inexact circuits [63], [64], [45]. The studies are performed at a fundamental physical level, *e.g.* network of switches including *AND*, *OR* and *NOT* gates, and further build inexact arithmetic operators, *e.g.* additions, and kernels, *e.g.* FFT.

- *Probabilistic computing*: "energy spent by the processing units is lowered, resulting in an increase of the probability that some operations might go wrong" [30]. George et al. from Georgia Institute of Technology, USA, explore the potential, in terms of energy reduction, of the probabilistic arithmetic units such as adders and multipliers. In these units, the circuits supply voltage is lowered proportionally to the output errors. The probabilistic arithmetic units are used to implement an FFT reducing the energy by a factor of 5.

- *Fault-tolerant computing*: “the ability of computing in a presence of faults to reduce the resource costs” [71]. This concept is already investigated for

a long time [71, 60], and it is applied by different communities in hardware and software. Building fault-tolerant systems requires to analyze the types of errors tolerated and the components reliable for fault-tolerant computing [16], [51], which is rather close to what approximate computing requires.

- *Stochastic computing*: “The ability to exploit the statistical nature of application-level performance metrics and to match it to the statistical attributes of the underlying device and circuit fabrics” [75]. Solutions for stochastic computing are proposed by various communities. For example Shanbhag and this team, from Illinois University, Urbana, USA, describe the potential in terms of performance and energy reduction in the design of non-ideal circuits, *i.e.* circuits designed for error-tolerant operations [75]. The authors raise also the challenges related to the design of computer aided design (CAD) for stochastic computing, *e.g.* develop techniques for mapping software programs onto programmable processors. Xiu et al. from Purdue University, USA, explore numerical methods for stochastic computations, *e.g.* Monte Carlo [91].
- *Imprecise computing*: “The approach which enables programs to produce results that are not correct using less time or resources” [31]. Imprecise computing [46] is oftenly employed interchangeably with approximate computing [46, 31, 90]. This term seems to be preferred in the real-time domain.
- *Significance-driven computing*: “the ability to maximize quality while meeting user-specified energy constraints” [83]. The term is coined by with a team from Thessaly university, Greece and Queen’s university Belfast, United Kingdom [83, 66, 84]. The authors propose a programming framework for the investigation of the energy reduction and the output quality of software programs. The framework includes a programming model with pragmas directives to annotate the source code, a compiler support that handles the pragmas directives and a runtime controller that makes decisions at runtime. The pragmas directives indicate the relative importance of the tasks and allow the runtime controller to execute the approximated version of a given source code, provided by the programmers.

Our work investigates the impact of reduced width arithmetic operations on applications. As such, it may fall under the name of *significance-driven computing* or *imprecise computing*.

This chapter is organized as follows. Since our study focuses on the reduced width, we first describe the numerical data representation formats used in the approximate computing methods in Section 2.1. Second we present algorithmic approximations applied in both hardware and software solutions in Section 2.2. Third we present the hardware blocks for approximate computing in Section 2.3, and the software-level solutions for approximate computing including programming and compiler support and runtime control in Section 2.4. Then we describe the benchmark applications and the quality metrics for evaluation of our approximation strategy in Section 2.5. Then we present the target processor to execute these applications in Section 2.6. Finally we present the energy model estimated to evaluate the energy consumption of various instructions in Section 2.7. Section 2.8 summarizes the chapter.

2.1 Numerical data representation

In a computer, real numbers (*i.e.* reals) are typically encoded on a finite number of bits. The two common number formats are: the floating-point representation and the fixed-point one. Because the width of a given format is finite, not all reals can be represented, thus, for some numbers, rounding is required. Several rounding modes exist. For an in-depth presentation of real numbers and the associated arithmetic we refer the reader to [56]. In what follows we introduce the basic concepts necessary to understand the rest of the thesis.

2.1.1 Floating-point

The number x is represented in floating-point by three bit fields, as presented in Figure 2.1:

- *sign* s; $s = 0$ if $x > 0$ and $s = 1$ if $x < 0$;
- *exponent* e; e is an integer such that $e_{min} < e < e_{max}$; e_{min} is the smallest possible *exponent* and e_{max} is the largest possible *exponent*;
- *significant* (or *mantissa*) m; m has one bit before the radix point and at most $p - 1$ bits after; p is the precision: the number of bits of the *mantissa*.

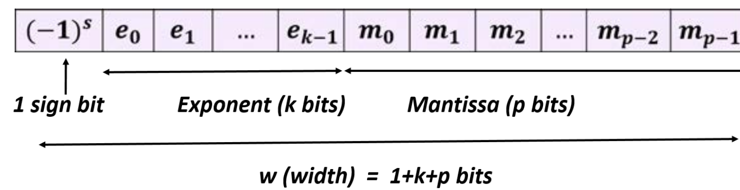


Figure 2.1: Floating-point representation

$$x = (-1)^s \times m_0.m_1m_2...m_{p-1} \times 2^e \tag{2.1}$$

Rounding modes

The rounding modes to represent a number (x) in a given format are:

- rounding towards $-\infty$ ($R_-(x)$): returns the largest machine number that is less than or equal to x;
- rounding towards $+\infty$ ($R_+(x)$): returns the smallest machine number that is greater than or equal to x;
- rounding towards 0 ($R_0(x)$): is equivalent to $R_-(x)$ if $x \geq 0$ and to $R_+(x)$ if $x \leq 0$;
- rounding to nearest: returns the closest machine number to x; if x is equidistant to two consecutive machine numbers, the result could be: the number that is away from 0, the even number, or the odd one.

2.1.2 Fixed-point

The number x is represented in fixed-point by a fixed number of bits before and after the binary point, as presented in Figure 2.2. The fixed-point format consists of:

- an integer part represented on "$i+1$" bits including the signed bit that is the most significant bit (x_i)
- a fractional part represented on "f" bits

- a width, that is the total number of bits: $w = 1 + i + f$

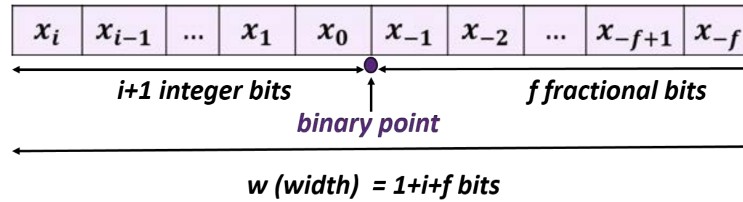


Figure 2.2: Fixed-point representation

$$x = \sum_{k=-f}^{i+1} (x_k \times 2^k) \tag{2.2}$$

The same rounding modes as ones presented above in Sub-section 2.1.1 can be used in the fixed-point representation.

The conversion from one format to another is possible. The direction of the conversion in embedded systems is mostly from the floating-point representation to the fixed-point one. For example, [50] proposes a methodology for the floating-to-fixed-point conversion for software implementations for DSP architectures. The aim is to determine the fixed-point specification minimizing a program source code execution time for a given accuracy constraint. Fixed-point libraries, proposed to help programmers implementing their applications, are compared in what follows.

Fixed-point libraries

`Libfixmath`, a C/C++ library [1], implements all mathematical functions such as trigonometric functions, logarithm and exponential functions, root square function and arithmetic operators. However the library handles only the format $Q32.16$ for data representation, *i.e.* w = 32 and f = 16. The smallest and the largest width in `Libfixmath` is 32 bits and the rounding mode is the rounding towards $+\infty$.

The `Fixed Point Class` C++ [2] implements the cosines and sinus functions, the exponential and the root square functions and the arithmetic operators. The format $Qw.f$ is represented with $w \in \{8, 16, 32\}$ and each of the three w values includes a number of fractional bits f, with $f \in \{1, 2, \ldots, 8\}, \{1, 2, \ldots 16\}\, or\, \{1, 2, \ldots 32\}$,

respectively. In `Fixed Point Class`, the smallest width is equal to 8 bits and the largest width is 32 bits and the rounding mode is the rounding towards $+\infty$.

The `MFixedPoint` C/C++ library [3] implements only arithmetic operators. The width of the format $Qw.f$ is represented with $w \in \{32, 64\}$ and $f \in \{1, 2, ...32\}$ or $\{1, 2, ...64\}$, respectively. In `MFixedPoint`, the smallest width is equal to 32 bits and the largest width is 64 bits and the rounding mode is the rounding towards $+\infty$.

`Libfi`, a C/C++ library [4] implements only arithmetic operators. The represented formats in `Libfi` are more flexible than the above ones: the width does not have to be a power of two: $w \in \{2, 3, ..., 32\}$ and $f \in \{1, 2, ...32\}$.

In `Libfi`, the overflow behavior and the rounding mode are customizable. `Libfi` proposes a data-type format that allows to handle five parameters:

- *TOTAL_WIDTH*: total number of bits in binary representation, including the sign for signed numbers; it corresponds to w in the $Qw.f$ format;
- *FRACTION_WIDTH*: number of fractional bits; it corresponds to f in the $Qw.f$ format;
- *SIGNEDNESS: Fi::SIGNED* for signed numbers; *SIGNEDNESS:Fi::UNSIGNED* for unsigned numbers;
- *OVERFLOW*: behavior when a number overflows the range representable with the selected quantization parameters; the valid parameters are:
  - *Fi::Saturate*: saturate the number to the maximum or to the minimum value allowed with the selected quantization parameters;
  - *Fi::Wrap*: wrap the value around when overflow occurs; if we try to increase the largest possible value, the smallest possible value is returned; as opposite, if we try to decrease the smallest possible value, the largest possible value is returned;
  - *Fi::Throw*: when overflow occurs, throw a *Fi::PositiveOverflow* or a *Fi::NegativeOverflow* exception, depending on the direction of the overflow;
  - *Fi::Undefined*: the behavior of overflow is undefined; this overflow option is selected when execution speed is more important than the output results;

- *ROUNDING*: behavior when a number is not representable with the selected quantization parameters. The rounding modes are the same as ones presented in Subsection 2.1.1;

Table 2.1 summarizes the features of each fixed-point library presented above. For each of them, we indicate the types of implemented functions: arithmetic operations and mathematical functions such as trigonometric functions, exponential, square root. Although `Libfi` implements less functions than others, it allows to make computation in low precision, *e.g.* less than 8 bits and it allows to handle overflow behavior during simulations. In our investigation, we select `Libfi` which is more flexible than the others libraries. To handle the functions not implemented in the `Libfi` library an approximation of such functions is performed with `Sollya` [25] designed for implementation of numerical functions, including the estimation of the mathematical functions in their polynomial versions.

Libraries	Arith. operations	Math. functions	Low precision ($w \leq 8$)	Scalability (w not necessarily a power of 2)	Overflow handled	All rounding modes
`Libfixmath`	✓	✓	✗	✗	✗	✗
`Fixed Point Class`	✓	✓	✓	✗	✗	✗
`MFixedPoint`	✓	✗	✗	✗	✗	✗
`Libfi`	✓	✗	✓	✓	✓	✓

Table 2.1: Fixed-point libraries

2.2 Algorithmic approximations

A given problem may be solved by multiple algorithms. Approximations can be done at this level by choosing an algorithm with less steps or without mathematical functions that have a costly implementation.

One of the algorithmic approximation approaches applied on both hardware and software solutions is the approximations of the costly mathematical functions with CORDIC [87] or polynoms [42]. For example, [49] estimates the $cos(x)$ and $sin(x)$ functions with a CORDIC-based approach using shifts and additions operations. Other methods such as polynomial approximation are proposed to reduce the computation time due to iterations in CORDIC-based methods.

[42] proposes a polynomial approximation solution. The authors estimate, with additions and multiplications operations, the mathematical functions includ-

ing $-(\frac{x}{2})log_2(x)$, $cos^{-1}(x)$, $\sqrt{-ln(x)}$, $ln(1+x)$, $\frac{1}{(1+x)}$. The proposed solutions aim to reduce the size of the interval of input data by proposing a segmentation method other than the traditional uniform segmentation, as presented on Fig.2 of the third page of the paper. The experimental results indicate that for more than 8 bits of precision, the area is reduced by a factor of 2 with their segmentation method compared to the uniform one.

One common limitation that we note in the existing methods for mathematical polynomial approximation is the fact that the energy consumption of the studied functions is not evaluated. The metrics to evaluate the effects of the function approximations are for example, the output accuracy [80], the memory size [22].

One of the most implemented methods applied on software solutions is task skipping [17], [77], [76]. The method consists in reducing the number of executed operations for a given program source code. For example, [17] proposes an automatic framework that allows programmers to reduce the energy consumption of the expensive loops. The programmer provides the source code with loop perforation (*i.e.* with less iterations of the loops than the actual ones) and the function for evaluation of the output quality. For each execution, the output is compared against the expected output to evaluate the quality loss by running the application with less iterations in the loops. The reduction of the number of instructions is implemented with an early termination of the loops when the required quality of result is reached. The experiments are performed on commercial applications and a web-search engine executed on an Intel core 2 Duo (3GHz) processor and an Intel 64-bit Xeon Quad core (2.33 GHz) processor. The results indicate that the energy consumption can be reduced by up to 21% for an error rate equal to 0.27%. We note that the proposed approach allows users to be less involved in the quality of results evaluation and the method improves energy reduction for an acceptable quality of output. The task skipping allows also to reduce the computation time of a given program as indicated in [77]. [77] evaluates the loop perforation technique on a set of applications, *e.g.* Monte Carlo simulation, search space enumeration, in order to generate source code that produces results with less computations. The experiments are performed on the PARSEC suite [20] that includes image processing, multimedia and scientific applications executed on an Intel Xeon X5460 Quad core processor. The results indicate that for an error less than 5%, the execution

time is reduced by a factor of 2.

Another approach that can be applied on loops and other operations are proposed in [76]. The approach consists in the implementation of a mechanism called *Hardware Redundant Execution (HaRE)*. The *HaRE* allows to reduce the errors on computations by returning in some previous states that are not costly and that have an impact on the program output quality. The *HaRE* allows to activate a task with the pragma *HaRE on* or to avoid a task to be executed with the pragma *HaRE off*. The experiments are performed on machine learning applications executed on Graphite, a simulator for multi-core systems [52]. The results indicate that with an error rate equal to 0.1%, the computation time can be improved by up to 54%.

The above solutions produce good costs reductions, *e.g.* energy consumption, computation time for an acceptable quality of output of the evaluated applications. In this work, we keep the number of program operations and investigate how the energy consumption can be reduced with an acceptable output quality, by executing them with reduced width units embedded in a processor.

2.3 Hardware blocks for approximate computing

Decreasing circuit complexity or clocking up less area are ways to reduce the energy consumption in hardware design. Several solutions have been proposed, for example: the approximate accelerators [55], [33] and the approximate operators [35], [61].

The design of arithmetic operators for approximate computing, *e.g.* adders and multipliers, is one of the most common work in the approximate computing community at hardware level. The energy consumption of these operators is reduced by applying methods such as the voltage scaling and the width reduction.

The voltage scaling aims to reduce the circuit's energy consumption. The principle is to increase or decrease the supply voltage depending on the required energy reduction and the target output quality. Several solutions have been proposed *e.g.* [26], [70]. The width reduction at hardware level consists in reducing the number of bits in the design of hardware blocks to reduce their complexity [95],[35].

The proposed arithmetic operators for approximate computing can be grouped into two categories: the fixed width ones and the variable width ones. The difference between the two categories is the fact that the variable width operators can

be reconfigured at runtime and the fixed width operators cannot be changed at runtime.

2.3.1 Fixed width adders and multipliers

The fixed width operators are operators in which the precision is not configurable at runtime, *i.e.* the precision does not change with the new parameters, *e.g.* input data, energy budget. Different solutions have been proposed for adders and multipliers.

[95] proposes an adder that consists of two parts: an accurate part and an inaccurate one. For the accurate part, the adder follows the standard addition principles from the least significant bit (LSB) to the most significant bit (MSB). For the inaccurate part another mechanism is applied from MSB to LSB. The mechanism consists in performing the standard one-bit addition if the two bits of the operands are equal to "0" or if the two bits are different from one to another; but when the two bits are equal to "1", all bits in the right part, from this bit are set to "1". The adder is implemented using a library for $0.18\mu m$ CMOS technology and simulated with a frequency set to 100 MHz. The results indicate 60% of power reduction when compared to the conventional adders. We note that the mechanism applied on the fixed width adders improves the energy reduction and the delay. The limitation of such operators is the fact that it is optimized for a given width configuration.

The design of multipliers is also one of the topic of interest in approximate computing at circuit level. In [40] an approximate multiplier is proposed. It is composed of two parts: a multiplication part in which the standard accurate multiplication is applied and a non-multiplication part in which a special mechanism is applied. The mechanism consists in checking the bit position of each operand. If the two bits are equal to "0", the corresponding bit result is set to "0" and if one or both of the two bits are equal to "1", the process finishes and all result bits that follow are set to "1". The multiplier is implemented using a library for $0.18\mu m$ CMOS technology and simulated with a frequency set to 100 MHz. The results indicate that, in comparison with the standard multiplier, for a 12-bit multiplier, the power dissipation decreases from 50% to 96% and a reduction of area by a factor of 2.1 for more than 90% of output quality. The results are interesting but we note the same limitations as in the fixed width adders, *i.e.* the complexity in the configurability of the circuits

to fit with parameters such as accuracy, energy savings requirements.

The above operators (adders and multipliers) are energy-efficient and can be configured for a given application. We target a general purpose embedded processor that can execute multiple applications. The fixed width operators are not convenient to handle different widths demands.

2.3.2 Variable width adders and multipliers

Variable-width operators are operators in which the precision can be configured at runtime. Several operators are proposed, *e.g.* [86], [38], [35], [61].

[86] presents an adder with an accurate and an inaccurate part. The inaccurate part performs computations with reduced bits in the carry chain. In this adder the output is evaluated at runtime. A signal indicates if the result is equal to the expected one. If the result is not correct, the errors induced by the approximations are corrected by performing the addition on the accurate part. The adder is implemented using a library for $0.18\mu m$ CMOS technology. The results indicate that the proposed adder can be $1.5\times-2.5\times$ faster than an accurate adder.

[38] proposes an accuracy-configurable approximate adder (ACA) that includes an error-correction functionality and aims to reduce the number of bits in the carry chain. ACA supports both accurate and inaccurate computations; the accuracy of computation is reconfigurable. The adder is divided into three sub-adders and the middle one corrects the potential errors, as presented on Figure 2 on the second page of the paper. The error detector is implemented with several AND gates. The ACA design is synthesized to a TSMC 65nm cell library with Synopsys Design Compiler. For simulations at gate-level, Cadence NC-Sim is used. The results indicate that the energy can be reduced by up to 30% when compared with the conventional pipelined adder.

[35] proposes a Dynamic Range Unbiased Multiplier for Approximate Applications (DRUM). The parameterizable multiplier is implemented to dynamically tune the precision of computations depending on the accuracy and the power consumption target. The proposed method limits the number of bits by selecting dynamically a range of bits on the two operands. For each operand, the $k-1$ bits from the most significant bits (MSBs) are selected; the kth bit is set to 1. The design is synthesized to a TSMC 65nm cell library with Synopsys Design Compiler. The evaluation

of DRUM is performed on a hardware Gaussian filter. The results indicates that for an acceptable accuracy, *i.e.* SNR of 91 dB, the whole Gaussian filter design with DRUM achieves power savings of 58% for $k = 6$ bits.

[61] implements a 16-bit multiplier in which the width is scaled with the voltage scaling method. A given number of LSBs of the operator inputs are set to zero. The method is evaluated on booth multiplier implemented on a 28nm FDSOI standard-cell library from STMicroelectronics. The results indicate that the energy consumption of a 16-bit fixed-point multiplier, at 10 bits is reduced by up to 32%.

The variable width operators solved the issues raised in the fixed width ones, *i.e.* the configurability at runtime. Although the proposed operators have large energy savings, up to 58%, when evaluated separately, most of them are not evaluated on a complete application. The evaluation of these operators on real applications, are required to have an insight into the global energy reduction for a given program.

2.4 Software for approximate computing: programming, compiler and runtime support

The application of the approximate computing techniques at software level requires programming models to express specific data-types for the *approximable* variables. Note that in this thesis, the *approximable* variables refer to the variables in which the approximation leads to acceptable quality degradation. The authors in [74], [68], [73] propose programming and compiler support for approximate computing.

[74] proposes EnerJ, an extension of the Java programming language. EnerJ includes type qualifiers ($@Approx$ and $@Precise$) to declare the *approximable* variables. The type qualifiers allow to mark *approximable* variables for low-power memory storage operations and approximate operations for computation. The strategy applied in operations for computation is the floating-point width reduction. For the low-power memory storage operations, the applied strategies are the reduction of the supply voltage and the reduction of the refresh rate of the DRAM. EnerJ is evaluated on a set of applications in various domains: SciMark2 suite [5] including scientific kernels, ZXing [6], a bar code reader for mobile devices, jMon-

keyEngine [7], a 2D and 3D game engine that implements triangle intersection problems, ImageJ [8], a program for image processing, Raytracer [9], a 3D renderer that generates images by tracing the path of light as pixel. The evaluated applications are executed on both desktop and mobile environments. The results indicate that with at most 34% of annotated variables, the energy reduction is from 10% to 50% with an acceptable output quality. We note that the proposed extensions for EnerJ are simple to implement by a programmer.

[68] proposes FLEXJAVA, a framework for Java applications that allows programmers to make annotations on the source code to mark the *approximable* parts, as in [74]. Moreover FLEXJAVA handles object-oriented programming concepts, *e.g.* inheritance, polymorphism. The same approximation strategies as [74] are applied both in the computation and memory storage operations (*i.e* floating-point width reduction, reduction of the supply voltage and reduction of the refresh rate). The evaluation is performed on the same applications and platforms as EnerJ. Compared to the EnerJ, FLEXJAVA claims to reduce the programmer efforts in the annotation process: from $6\times$ to $12\times$, because for the same energy reduction, FLEXJAVA reduces the number of required annotations from $2\times$ to $17\times$. We note that FLEXJAVA proposes a safety analysis to identify the sensitive operations.

[73] proposes ACCEPT, a framework that includes C/C++ type qualifiers derived from EnerJ ($@Approx$ and $@Precise$). The pointer types are not yet handled in the approximation scheme. The approximation strategies are the loop perforation [77], the neural acceleration [28]. ACCEPT is evaluated with the PARSEC parallel benchmark suite [19] in 3 platforms: a standard x86 server, a mobile SoC with an FPGA for neural acceleration, and a low-power embedded sensing device. The results indicate that the speed up is improved by up to $2.3\times$ on the x86 server, up to $4.8\times$ on the mobile SoC, and up to $1.5\times$ on the embedded device for errors less than 10%, acceptable for the evaluated applications.

The above solutions claim a high energy reduction, *i.e.* up to $2\times$. To improve the energy reduction (by up to $2.8\times$), other work proposes runtime control solutions. However most of them implement the task skipping techniques [69] or handle the cores for execution (*i.e.* host processor, accelerators) [48].

[69] develops a runtime framework that automatically skips operations, on a given source code during the program execution. The authors target applications

with high volume of data including a *reduced-and-rank (RnR)* kernel, used in, *e.g.* video processing, recognition, search and data mining. A RnR kernel allows to perform a reduction operation (*e.g.* L1-norm, distance computation, dot product) between a given input data vector and a set reference vectors to return a set of reduction outputs. From these outputs, the user deducts the degree of approximation of a given source code. An other approach deducts the degree of approximation of a given source code by studying the correlation between the current input data and a set of previous input data. The experiments are performed on eye detection and clustering applications executed in a hardware extended with additional registers, counters, and control logic. The additional components, synthesized with Synopsys Design Compiler and mapped to a 45nm Open Cell Library, allow to automatically tune the quality knobs. The results indicate that the energy is improved up to $2.38\times$ and $2.5\times$ when the quality constraints are relaxed to 2.5% and 5% respectively, acceptable for the evaluated applications. The energy overhead induced by the search of the degree of approximation and by the additional components is around 10% compared to the energy consumption of the original program. We note that the overhead energy costs are acceptable in comparison with the global energy reduction obtained with the method.

[48] implements a predictor that estimates the quality of output degradation of a given program and indicates which source code version (original or approximated) has to be executed. The selected version of the source code to be executed on a host processor or on an approximate accelerator depends on the required quality of output. The methods for the predictor design are a table-based approach and a neural approach. The table-based approach, with a training phase, maps the input data to the corresponding prediction, *e.g.* errors on the quality of output. The neural approach estimates models for quality of output control, with a set of training data. The experiments are performed on a set of applications in various domains: image processing, clustering, 3D gaming, executed on a processor that includes an accurate core and a neural processing unit. The results indicate that for the table-based predictor the energy consumption is reduced by up to $2.8\times$ for an error of 5% and the neural predictor achieves 17% larger energy reduction than the table-based predictor. We note large energy reduction, however we have no idea about the values of energy overhead costs induced by the runtime process.

The proposed solutions for runtime control mostly investigate the task skipping

technique. To exploit the configurability of the units that we study in our contributions, the runtime control of the data representation could be studied for more energy reduction. Moreover the solutions are mostly for floating-point architectures. Since numerous low-power embedded processors do not include floating-point units because the hardware implementation of a floating-point support requires a higher silicon area and power consumption than fixed-point or integer ones. Barrois *et al.* [18] highlights the advantages of fixed-point operators compared to the floating-point ones, *e.g.* in terms of energy reduction, silicon area. On a real application such as K-means, the floating-point representation with reduced widths (*e.g.* 8 bits) provides more energy reduction. However with larger widths (*e.g.* 16 bits) the fixed-point representation is still less costly in terms of energy consumption compared to the floating-point one. As we aim to study applications running on a general embedded processor, we stick to conventional cores with integer or fixed-point units.

2.5 Benchmark applications and quality metrics

The benchmark applications potentially of interest for approximate computing area, for example Mediabench [41], ALPbench [44] which aim at the evaluation of multimedia and communication systems. These suites include applications such as data compression, human machine interface. The San Diego vision benchmark [85] includes applications for features tracking, image segmentation, robot localization. MEVBench [27] includes applications for mobile computer vision. Minebench [57] is a data mining benchmark suite for clustering, classification applications. Axbench [93], a benchmark suite proposed by the approximate computing community, it includes a series of applications from various domains.

We select benchmark applications with available input data and quality metrics, and that tolerate approximations. Thus in this thesis we perform experiments with Axbench applications that provide a large set of input data and quality metrics, and that are specifically implemented to evaluate the impact of an approximate computing method.

Axbench

Axbench is a suite of CPU and GPU applications proposed to evaluate the impact of the neural network method [28] on the application output quality, the energy consumption and/or the computation time. The CPU applications are more suitable for our investigation. Axbench includes as CPU applications: blackscholes, FFT, inversek2j, forwardk2j, jmeint, jpeg, K-means, Sobel filter.

All the above applications are implemented in floating-point format. We target integer architectures to reduce the computation costs of the floating-point operations on the embedded systems. For each of the evaluated applications, we convert them in fixed-point representation with `Libfi` presented in Subsection 2.1.2. The errors introduced by the fixed-point conversion are evaluated with metrics presented below.

Most of these applications include complex mathematical functions such as logarithm, exponential, square root, and trigonometric functions, as presented on Table 2.2. Most of the embedded systems do not include floating-point arithmetic unit (FPU). We estimate these functions with polynomial approximation using `Sollya` [25].

Applications	Arith. op	Cos/Sin	Tan	Acos/Asin	Atan	Log	Exp	Sqrt
Blackscholes	✓	✗	✗	✗	✗	✓	✓	✓
FFT	✓	✓	✗	✗	✓	✓	✗	✓
Inversek2j	✓	✓	✗	✓	✗	✗	✗	✗
Forwardk2j	✓	✓	✗	✗	✗	✗	✗	✗
Jmeint	✓	✗	✗	✗	✗	✗	✗	✗
Jpeg	✓	✗	✗	✗	✗	✗	✗	✓
K-means	✓	✗	✗	✗	✗	✗	✗	✓
Sobel filter	✓	✗	✗	✗	✗	✗	✗	✓

Table 2.2: CPU-Axbench applications

We note that jmeint includes only arithmetic operations. Sobel filter, K-means, jpeg, the image processing applications, include arithmetic operations and the square root function. Blackscholes includes arithmetic operations, logarithm, exponential and square root functions. Other applications, *i.e.* inversek2j, forwardk2j and FFT include arithmetic operations, trigonometric functions. FFT includes the square root function in addition to trigonometric and arithmetic operations.

We choose to study the jmeint application because it includes only arithmetic functions and 2 other applications: the Sobel filter, that includes the square root

function in addition to arithmetic operations, and the forward2j including arithmetic and trigonometric functions. The square root and the trigonometric functions are estimated in polynomial approximations, using the `fpminimax` function of `Sollya`.

Jmeint application

Jmeint is an algorithm used in many 3D applications, *e.g.* gaming. Jmeint, with geometric computations, verifies if two 3D-triangles intersect. The input is a pair of triangles' coordinates in 3D-dimensional space and the output is a boolean value which indicates whether the two triangles intersect: 1 if the two triangles intersect and 0 if they do not intersect. The steps of the algorithm are presented on Figure 2.3. To evaluate the quality of output, we compute the error rate that is the ratio between the number of correct outputs and the total number of outputs.

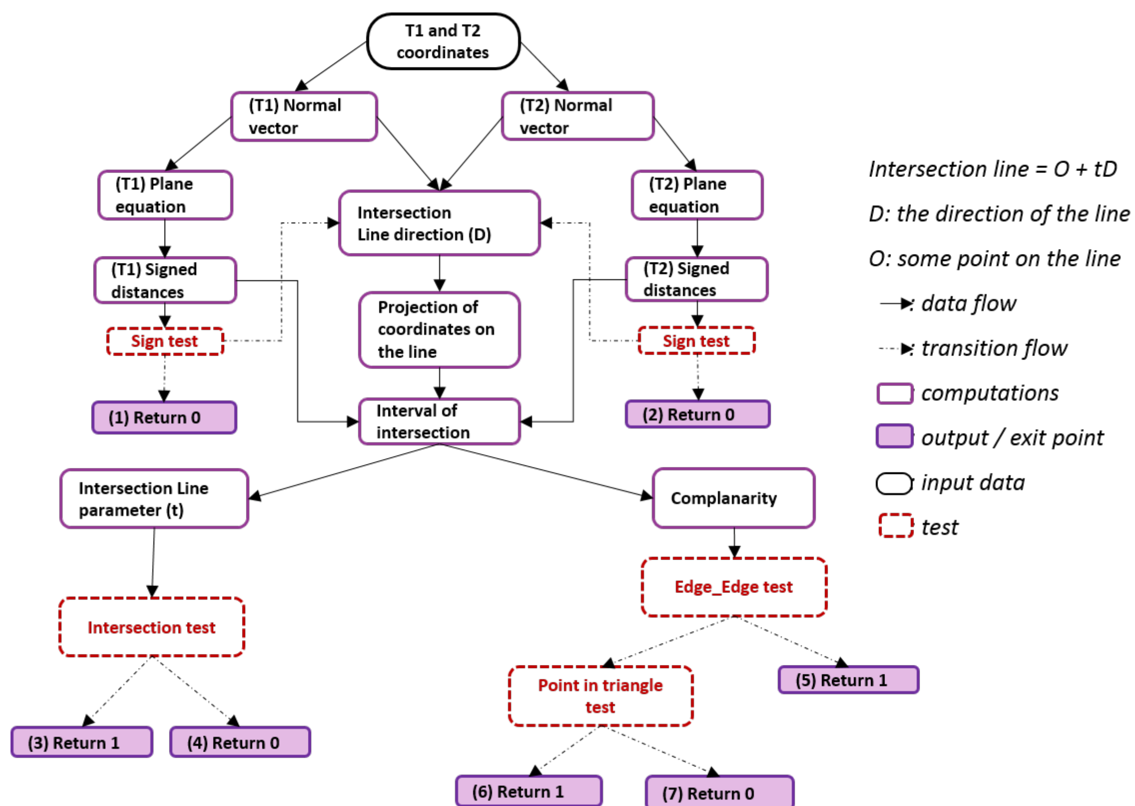


Figure 2.3: Jmeint algorithm steps

Jmeint quality metrics

To evaluate the output quality of the jmeint application, we compute the error rate. The error rate is an error metric that indicates the frequency of errors in a method evaluation. For jmeint, we compute the error rate between the vector of outputs returned by the approximated program and the vector of outputs returned by the original program. The formula of the error rate is:

$$\epsilon = \frac{N_{inc}}{N_t} \tag{2.3}$$

where N_{inc} is the number of incorrect outputs and N_t is the number total of outputs. N_{inc} is estimated with the *hamming distance*. The *hamming distance* between two words (of the same length) is the number of places where the digits are different, *i.e.* the number of positions where one is equal to 0 and the other equal to 1 and vice versa [79]. The two words in jmeint correspond to the two compared vectors of outputs.

Example: let's consider *a = 010110*, *b = 111101*, the hamming distance *d = 1+0+1+0+1+1 = 4*.

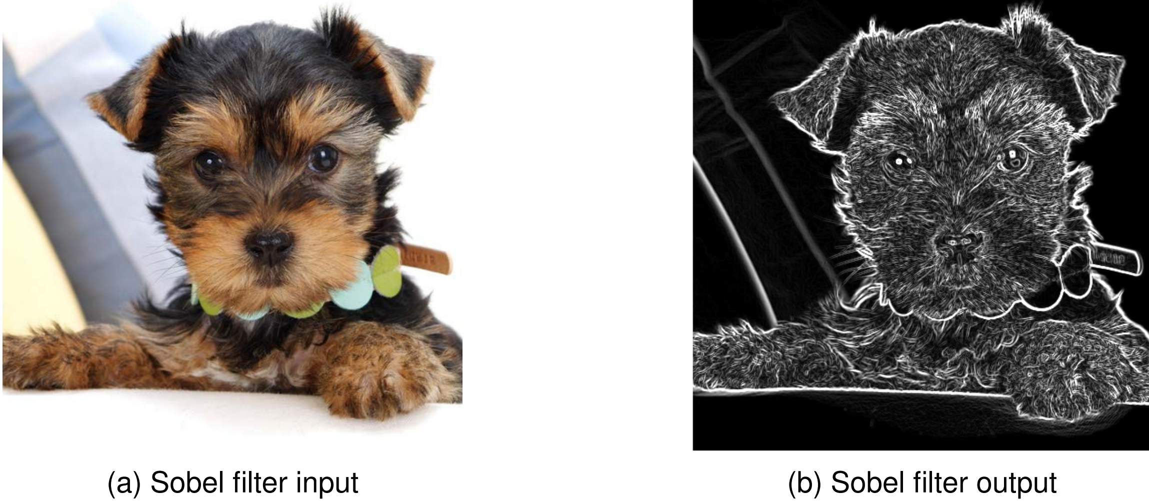
Sobel filter application

The Sobel filter is a kernel for image processing and computer vision applications, particularly for edge detection algorithms. Edge detection is an image processing technique to discover the boundaries between regions in an image.

The input is a RGB image and the output is a gray-scale image (PNG format) in which the edges are emphasized, as presented on Figure 2.4. The image gradient of each pixel is calculated by convolving the image with a pair of filters (horizontal and vertical filters), that are 3×3 matrix.

Sobel filter quality metrics

The image quality evaluation is an issue in image processing applications, for that, several metrics are proposed in this area, *e.g.* the RMSE, the PSNR and the SSIM. The metrics values are computed with the parameters of the two compared images, *i.e.* the image returned by the reference program and the image returned by the approximated one.



(a) Sobel filter input (b) Sobel filter output

Figure 2.4: Sobel filter input/output

The RMSE [78] and the PSNR [78] estimate the absolute errors between pixels of two compared images but are less correlated to human perception of image quality compared to the SSIM metric [88].

Let x and y be the matrix of the 2 compared images.

$$\mathrm{RMSE} = \sqrt{\frac{1}{N}\sum_{i=1}^{N}(x_i - y_i)} \tag{2.4}$$

where:

- $N = 3 \times height \times weight$
- 3 is the number of pixel components: r, g, b;
- $height$ and $width$ are the dimensions of the images;
- i is the position of the pixel component on the images;
- x_i is the reference pixel component value, y_i is the approximated pixel component value;

$$\mathrm{PSNR} = 20 \times \log_{10}\left(\frac{L}{RMSE}\right) \tag{2.5}$$

where L is the dynamic range of the pixel values (*e.g.* 255 for 8-bit images).

Let x and y the 2 compared images:

$$\mathrm{SSIM}(x,y) = \frac{(2\mu_x\mu_y + C_1)(2\sigma_{xy} + C_2)}{(\mu_x^2 + \mu_y^2 + C_1)(\sigma_x^2 + \sigma_y^2 + C_2)} \tag{2.6}$$

where:

- μ_x and μ_y are the mean pixel values of respectively the image x and the image y;
- σ_x and σ_y are the standard deviations between pixel values of respectively the image x and the image y;
- σ_{xy} is the covariance of the two images x and y;
- $C_1 = (K_1L)^2$ and $C_2 = (K_2L)^2$, included to avoid instability when $\mu_x^2 + \mu_y^2$ and $\sigma_x^2 + \sigma_y^2$ are very close to zero;
- $K_1 = 0.01$, $K_2 = 0.03$ and L is the dynamic range of the pixel values (255 for 8-bit images).

Forwardk2j application

Forwardk2j is a kernel for robotic applications. It aims to compute the positions of a robot's end-effector with the angles of the 2-joint robotic arm.

Forwardk2j takes as input the angles of the 2-joint robotic arm (Θ_1 and Θ_2) and computes the position of the end-effector of the 2-joint robotic arm (x and y).

Forwardk2j quality metrics

To evaluate the quality of output in the forwardk2j application, we compute the *mean relative error (MRE)* for each coordinate between the outputs returned by the approximated program and the outputs returned by the original program.

$$\mathrm{MRE} = \frac{1}{N}\sum_{i=1}^{N}\left|\frac{x_i - y_i}{x_i}\right| \tag{2.7}$$

where N is the total number of data, x the set of original reference values, y the set of approximated values.

Note that computing the *mean relative error (MRE)* with a series of very small or null expected values (that are the denominators) is an issue in mean relative error computations. To overcome these issues, there are some other measures

proposed in the literature, *e.g. Mean Absolute Scaled Error (MASE)* [29], *Symmetric Mean Absolute Percentage Error (SMAPE)* [32].

The *standard deviation* is computed on the values obtained with the above metrics (*i.e. RMSE*, *PSNR*, *SSIM*, *MRE*) for the purpose of evaluating the dispersion of a set data around the mean value. The lower the standard deviation, the most the data are closed to the mean, *i.e.* low differences between the elements of the studied data set.

$$\sigma = \sqrt{\frac{\sum_{i=1}^{N}(x_i - \bar{x})^2}{N-1}} \tag{2.8}$$

where $\bar{x}$ is the mean value of the data set.

2.6 RISC-V processor

This section presents the processor core baseline that we consider in our experiments. We utilize an open source RISC-V core [10]. The RISC-V community provides an open Instruction Set Architecture (ISA) specification and a large set of tools to simulate and synthesize RISC-V processor cores, as well as a large software tools base to program, debug, and test applications for RISC-V.

One can customize a RISC-V processor core to include, for example, integer multiply/divide, single and double precision floating-point arithmetic. The processor can work on 32, 64, or 128 bits. In the context of this thesis, we are interested in a 32-bit integer processor, with code-name ”RV32I base” in the RISC-V terms. From the proposed basic instruction formats [89], we use R-type, I-type and, S-type, described below.

2.6.1 RV32I instruction formats in our work

The R-type format is for instructions with two source registers and one destination register, *e.g.* addition (add), subtraction (sub). The I-type is for instructions with one source and one destination register, *e.g.* load word (lw), addition with an immediate value (addi). The S-type format has two source registers and an immediate destination, *e.g.* conditional branch instructions.

For the R-type format, the $bits[31:25]$ (*i.e.* the field `funct`) indicate the type of

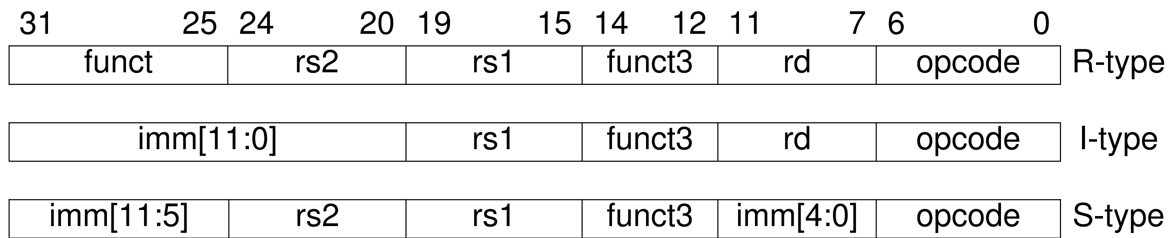


Figure 2.5: 32-bit RISC-V base instruction formats in our work [89]

operations to be executed, the $bits[24:20]$ (*i.e.* the field `rs2`) correspond to one source register and the $bits[19:15]$ (*i.e.* the field `rs1`) to the other source register, the $bits[14:12]$ (*i.e.* the field `funct3`) inform to the processor which registers from the registers source and destination communicate with an external accelerator, the $bits[11:7]$ (*i.e.* the field `rd`) correspond to the destination register, the $bits[6:0]$ (*i.e.* the field `opcode`) are for the code of the executed operation.

The I-type format has a `imm[11:0]` field in the $bits[31:19]$ for the immediate values to compute with the value of the register `rs1`. The format includes only one source register file that corresponds to the $bits[19:15]$. From 19 bits and below the I-type and the R-type have the same formats.

The S-type has the same formats as the R-type, except the fields `funct` and `rd` in R-type, that correspond to `imm[11:5]` and `imm[4:0]` for the immediate values in S-type.

2.6.2 RISC-V base opcodes map

An opcode is mapped with each of the instructions as presented on Table 2.3.

The opcodes of the operations are different from one operation to another. The opcode targeted *reserved* are only for the standard extensions of the RISC-V. Hence these opcodes cannot be used for external extensions added in the standard RISC-V ISA. Inversely, the opcodes *custom-0* and *custom-1* are recommended for the external extensions and are avoided for the standard extensions of the RISC-V. The opcodes *custom-2/rv128* and *custom-3/rv128* are reserved for the RV128 format and can utilized for the custom instruction-set extensions in RV32 and RV64.

In this thesis, we use the opcodes *custom-0* and *custom-1* to extend the standard RISC-V with reduced width units.

Table 2.3: RISC-V base opcode map [89]

inst[4:2] / inst[6:5]	000	001	010	011	100	101	110	111
00	LOAD	LOAD-FP	*custom-0*	MISC-MEM	OP-IMM	AUIPC	OP-IMM-32	48b
01	STORE	STORE-FP	*custom-1*	AMO	OP	LUI	OP-32	64b
10	MADD	MSUB	NMSUB	NMADD	OP-FP	*reserved*	*custom-2/ rv128*	48b
11	BRANCH	JALR	*reserved*	JAL	SYSTEM	*reserved*	*custom-3/ rv128*	$\geq$ 80b

2.6.3 RISC-V architecture

The RV32I base can be implemented in several ways. Figure 2.6 presents the RISC-V architecture. We consider a pipeline with the following classic stages: instruction fetch (IF) to bring the instruction from the instruction memory at the address supplied by the program counter (PC); instruction decode (ID) to decode the instruction, *i.e.* interpret the type of instruction and specify the register operands; execution (EXE) to compute the arithmetic and logic instructions results or to compute addresses for memory instructions, *i.e.* load and store; full width load/store unit (LSU) to load/store the data from/in the data-memory. Note that in our investigation, the memory does not refer to both the *dynamic random access memory (DRAM)* and the *static random access memory (SRAM)*, we only consider the SRAM.

The standard instructions in the original RISC-V processor could be grouped into categories:

- arithmetic and logic instructions: addition (add), subtraction (sub), multiplication (mul), division (div), negate value (neg), shift left logical (sll), shift right logical (srl),...
- branch instructions: branch on equal (beq), branch on less than (blt), branch

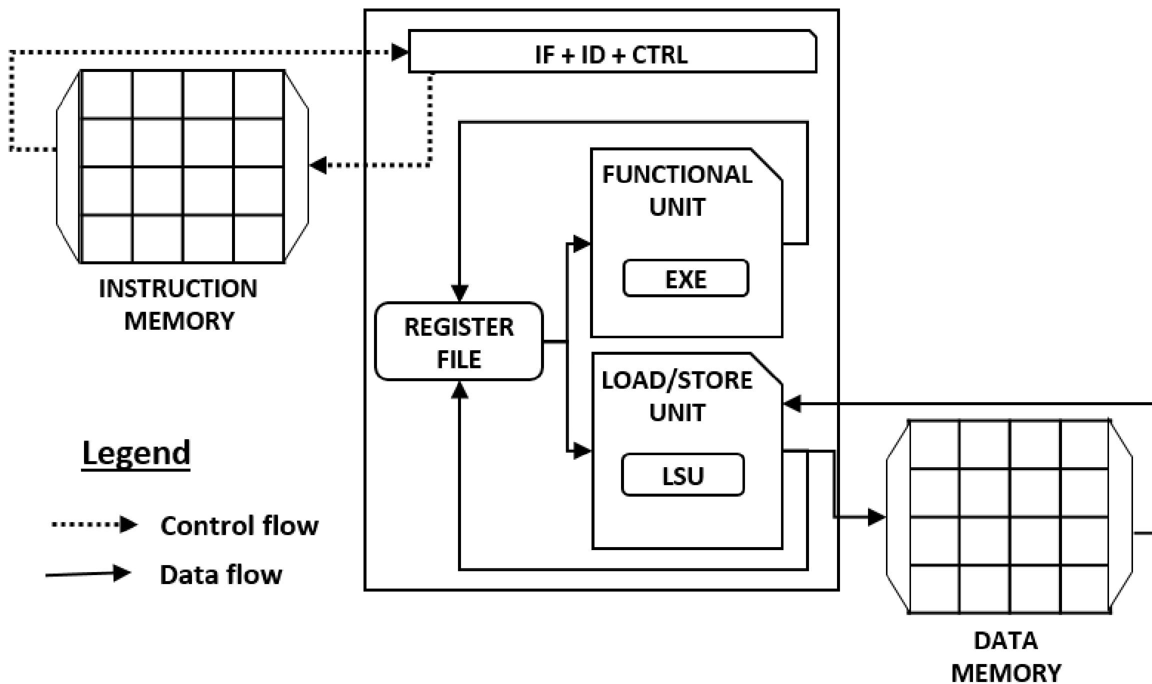


Figure 2.6: RISC-V Architecture

on greater than (bgt), jump (j),...

- data transfer instructions: load immediate (li), load word (lw), store word (sw), move (mv),
- comparison instructions: set equal (seq), set not equal (sne), set less than (slt), greater than (sgt),...

2.7 Energy models

For estimation of the global energy consumption of a given application, several energy models have been applied on various architectures, *e.g.* [54] and [82] estimate the accurate values of energy consumption of the arithmetic and memory instructions.

[54] measures the contribution of the data transfer and arithmetic instructions on the total power consumption of the AMD and Intel systems. The energy consumption of each instruction is measured: for double and single floating-point instructions and for access cache and RAM. The results indicates that the data

transfer power consumption is higher in the AMD architecture, *i.e.* $2.8\times$, compared to the Intel power consumption of the data transfer. For Intel, we note that a byte transfer on the RAM consumes $2.9\times$ more than a double precision addition and $2.6\times$ more than a double precision multiplication.

[82] proposes an energy model for ARM instructions, based on the energy measurements on two different ARM processors: Cortex-A7 and Cortex-A15. The measurements are obtained with the voltage/current sensors provided by the ODROID XU+E big.LITTLE Platform [34]. The measurements are performed on several types of instructions: integer, float and, double. We note that for the integer instructions, on the smallest memory (4KB), the load and store instructions consume $2\times$ more than the additions and multiplications and $2.5\times$ for a higher memory (*i.e.* 256KB). For the double instructions, on the smallest memory (4KB), the load and store instructions consume $2\times$ more than the additions and have approximately the same values as the multiplications energy consumption.

We note that in the above architectures, the costs for memory accesses is larger than the costs of the arithmetic instructions (*i.e.* from $2\times$ to $2.9\times$). We can deduce that in addition to the approximation of the common arithmetic units, the approximation of the memory units could be beneficial for energy efficiency.

A set of operations have been proposed for approximate computing in order to exploit the potential of the proposed approximate arithmetic and memory units, when added in a complete processor. The related energy models for each of the approximate arithmetic and memory operations are estimated, *e.g.* [67]. [67] proposes an energy model to evaluate the impact of the approximations performed on both arithmetic and memory operations. The approximate arithmetic operations are width scaling floating-point arithmetic ones. For the approximate storage, the methods consists of reducing the supply voltage and the refresh rate. For each approximate operation, an energy model that consists of a pair of parameters *(e_a;e_s)* is proposed, where e_a is the energy cost of an instruction when executed accurately and e_s is the factor of energy saved when an instruction is executed approximately. The results indicate an energy reduction for each operation, *e.g.* $e_s = 12\%$ for integer and logical operations, $e_s = 32\%$ for floating-point operations with 16 mantissa bits, $e_s = 17\%$ for the DRAM and $e_s = 70\%$ for the SRAM. We note that one limitation of the proposed model is the fact that overheads energy of implementing or switching to approximate hardware are omitted in the model. The methods are

evaluated on a set of applications including ZXing [6] a bar code reader for mobile devices, jMonkeyEngine [7] a 2D and 3D game engine that implements triangle intersection problems, Raytracer [9] a 3D renderer that generates images by tracing the path of light as pixel. The results indicate that the energy can be reduced by up to 35%. We note the energy reduction induced by the method is large. The large energy reduction is with not only the approximate arithmetic operations but with both approximate arithmetic operations and approximate memory ones. We can conclude that the approximate memory units have potential in terms of global energy reduction of a given application.

We can see that to integrate the energy estimation of approximate units within the energy consumption of an entire processor we would need to separate between: (1) the energy consumed by the arithmetic units, (2) the energy consumed by the SRAM, and (3) the energy consumed by the DRAM, in case the system has one. For our needs we consider the approximate arithmetic operations and the energy consumed by the SRAM to construct an energy model per instruction, as presented in Subsection 3.3.2. The width scaling method is applied in both operations categories.

2.8 Chapter summary

In this chapter we have presented an overview of the approximate computing approaches. Since we aim to put in perspective the reduced width units within an embedded processor, at first we present the numerical representation format of data, that are the floating-point format and the fixed-point one. Second we present the most used algorithmic approximations techniques, that are the task skipping and the methods for approximation of mathematical functions. Third we have reviewed the hardware blocks including approximate operators and we have discussed the programming, compiler support and runtime support for approximate computing on application source code. Then we have presented the benchmarks application and quality metrics for evaluation and we have described our target processor in which the reduced width units will be embedded. Finally we have presented the energy models that estimate different categories of instructions.

We note that most of the solutions proposed at software level, *e.g.* programming and compiler support, focus on floating-point architectures. Several embed-

ded systems do not include floating-point units and the implementation of floating-point support requires a high silicon area and power consumption. We target integer or fixed-point architectures to reduce the floating-point operations costs in embedded systems.

We note also that most of the hardware units designed for approximate computing focus on arithmetic units. The studied energy models indicate that the energy consumption of the memory accesses are largely higher than the energy consumed by the arithmetic operations, up to $2.9\times$. Hence the optimization of memory units has to be investigated to improve the global energy consumption of applications.

Moreover, the proposed approximate integer arithmetic units are only evaluated in a separate block, which does not provide an insight into their impact on a given application in terms of output quality and energy reduction. Hence these units need to be embedded in a processor for a global evaluation of their impact on real applications for more conclusive results.

CHAPTER 3

RISC-V PROCESSOR EXTENDED WITH REDUCED WIDTH UNITS

This chapter presents the extended RISC-V used for our approximate computing approach. We focus on integer operations since they are more appropriate for embedded systems than floating-point ones. The RISC-V processor is a popular platform for experimentations because it is open source, flexible and supports the implementation of several customized extensions.

Several approximate integer operators are proposed in the state of the art. Most of them are implemented as stand-alone units, *i.e.* not integrated in a processor. We extend the RISC-V processor with approximate integer units. Our approach uses reduced width units for both computation and data-memory access. In reduced width units, the computations are performed only on the b most significant bits (MSBs); $b \in \{1, ..., B\}$; B is the maximum possible width. For each class of the reduced width operation, an energy model is proposed for evaluating the global energy consumption of several benchmark applications.

To easily experiment with the RISC-V processor, during an internship Tiago Trevisan Jost implemented compiler support to handle pragmas added in the source code and extended the RISC-V simulator [11] with the profiling capabilities to return statistics including the number of instructions of a given type class. This work was put in perspective and integrated with an energy model by this thesis. The joint contribution was presented in a RISC-V workshop [81]. The energy model for our processor at the instruction level is a combination of internal data from CEA Leti test chip measurements with some power models from the literature [61].

This chapter is organized as follows. Section 3.1 presents the ISA of the extended RISC-V including both the standard full width instructions and the reduced width instructions added in the RISC-V. Section 3.2 presents the architecture of the RISC-V core extended with reduced width units. Section 3.3 presents our energy

model for applications evaluation. Section 3.4 presents our experimental environment including a set of annotations to ease programming and an instrumentation tool which allows to evaluate application output quality *vs* energy trade-offs. Section 3.5 summarizes the chapter.

3.1 Extended RISC-V ISA with reduced width instructions

The ISA of the extended RISC-V consists of its standard instructions and our reduced width instructions. We consider the RV32I base of the RISC-V processor, in which the format of the instructions are fixed to 32 bits as presented in Figure 2.5 in Chapter 2.

The categories of operations available with reduced width units are presented on Table 3.1. These operations are arithmetic, logic and data transfer operations. The basic RV32I formats of RISC-V are presented in Figure 2.5. The fields `funct` (i.e., $bits[31:25]$), `funct3` (i.e., $bits[14:12]$) and `opcode` (i.e., $bits[6:0]$) indicate the instruction category.

Table 3.1: RV32I base for full width instructions

31 25	24 20	19 15	14 12	11 7	6 0	
0000000	src2	src1	000	dst	0110011	add
0100000	src2	src1	000	dst	0110011	sub
0000001	src2	src1	000	dst	0110011	mul
0000001	src2	src1	101	dst	0110011	udiv
0000001	src2	src1	100	dst	0110011	sdiv
imm[11:0]		src1	000	dst	0010011	addi
imm[11:0]		src1	101	dst	0010011	udivi
imm[11:0]		src1	100	dst	0010011	sdivi
imm[11:0]		src1	010	dst	0000011	ld
imm[11:5]	src2	src1	010	imm[4:0]	0100011	st

For each of the above operations presented on Table 3.1, to reduce the global energy consumption of applications we implement the corresponding reduced width operation, as presented below.

The implemented reduced width operations are described on Table 3.2. The reduced width operations for computation, the programming model and the instrumentation tooling are published in [81]. We adopt the following convention: we prefix with `a.` the standard full width instructions, to designate the corresponding reduced width versions. The instructions `a.set.b`, `a.get.b` are implemented to set or to get the actual width of the operator. The instruction `a.set.b` sets the width into a source register and `a.get.b` gets the width from the destination. Furthermore, we extend the work in [81] by adding a reduced width unit for data-memory accesses. Note that the instructions for reduced width memory accesses are not implemented in the initial extended RISC-V. The field `funct3` informs the processor which registers from the two registers source and the destination register are activated in the communication with an external accelerator. In the context of this work, we do not implement an external accelerator, we have just added units in the standard pipeline. Nevertheless we use the same principle to inform which registers are activated. If the bit 14 is activated, the value returned by the extended units is stored in the destination register (*dst*). If the bits 13 and 12 are activated, the source registers *src1* and *src2* are sent to the extended units. Table 3.2 indicates that:

- only one source register is activated for `a.set.b`,
- only the destination register is activated for `a.get.b`,
- all the three bits are activated for `a.add`, `a.sub` , `a.mul` , `a.udiv`, `a.sdiv`,
- one source register and the destination register are activated for the reduced width immediate instructions `a.addi`, `a.subi` , `a.muli` , `a.udivi`, `a.sdivi`,
- the destination register is activated for `a.ld` and the source register is activated for `a.st`.

The opcodes correspond to the *custom-0* and the *custom-1* values on Table 2.3: $bits[6:2]$ are equal to 00010 or 01010 and the $bits[1:0]$ are set to 11 to avoid overlaps with other standard extensions of the RISC-V.

Table 3.2: RV32I base for reduced width instructions

31 — 25	24 — 20	19 — 15	14 — 12	11 — 7	6 — 0	
0011111	00000	src1	010	00000	0001011	a.set.b
0011110	00000	00000	100	dst	0001011	a.get.b
0000000	src2	src1	111	dst	0001011	a.add
0000001	src2	src1	111	dst	0001011	a.sub
0000010	src2	src1	111	dst	0001011	a.mul
0000011	src2	src1	111	dst	0001011	a.udiv
0000100	src2	src1	111	dst	0001011	a.sdiv
immediate		src1	110	dst	0101011	a.addi
immediate		src1	110	dst	0101011	a.subi
immediate		src1	110	dst	0101011	a.muli
immediate		src1	110	dst	0101011	a.udivi
immediate		src1	110	dst	0101011	a.sdivi
imm[11:0]		src1	100	dst	0101011	a.ld
imm[11:5]	src2	src1	010	imm[4:0]	0001011	a.st

Let us consider the registers a1, a2, a3, a4, and a5. For each reduced width operation an example is presented below.

Example: `a.set.b`, `a.get.b`

The instructions `a.set.b` and `a.get.b` are implemented for the configuration of the units width.

```
li a5, 2                  # Moves 2 to a5
a.set.b a5                # Sets the width to 2
a.get.b a4                # Stores the current width in a4
```

Listing 3.1: Set/Get the width

Example: addition with reduced width (`a.add`)

The instruction `a.add` computes the reduced width addition. The addition is performed on b bits determined by the `a.set.b` instruction. An example of `a.add` usage is indicated in Listing 3.2.

```
li   a5, 2            # Moves 2 to a5
a.set.b  a5           # Sets the width to 2
a.add a3, a2, a1      # adds a2 to a1 and stores the result in a3,
                      # based on the actual width
```

Listing 3.2: Addition with reduced width

Example: immediate addition with reduced width (`a.addi`)

It performs a reduced width addition between an immediate and a register value, based on the configured width.

```
li   a5, 2            # Moves 2 to a5
a.set.b a5            # Sets the width to 2
a.addi a3, a2, 4   # adds 4 to a2 and stores the result in a3,
                      # based on the actual width
```

Listing 3.3: Immediate addition with reduced width

Example: load with reduced width (`a.ld`)

The instruction `a.ld` performs a reduced width load by copying a value from the data-memory to a register, based on the configured width.

```
li   a5, 2            # Moves 2 to a5
a.set.b a5            # Sets the width to 2
a.ld  a3, [a2]        # loads the value at address found in a2 to a3
```

Listing 3.4: Load with reduced width

Example: store with reduced width (`a.st`)

The instruction `a.st` performs a reduced width store by copying a value of a register into the data-memory, based on the configured width.

```
li   a5, 2            # Moves 2 to a5
a.set.b a5            # Sets the width to 2
a.st  [a3], a2        # stores the value of a2 in a3
```

Listing 3.5: Store with reduced width

3.2 Architecture of the extended RISC-V

We consider a conventional pipeline with the following stages: instruction fetch (IF) to bring the instruction from the instruction memory at the address supplied by the program counter (PC); instruction decode (ID) to decode the instruction, *i.e.* interpret the type of instruction and specify the register operands; full width execution (EXE) to compute the result of full width multiplications, arithmetic and logic instructions or to compute address for memory instructions, *i.e.* load and store.

We extend the RISC-V processor with both reduced width computation unit, denoted `a.EXE` and load and store unit denoted `a.LSU`. The unit `a.EXE` computes the result of reduced width multiplications, arithmetic and logic instructions; the unit `a.LSU` handles the reduced width load and store instructions. Figure 3.1 presents our architecture of the complete processor including full width units and both approximate functional unit (`a.EXE`) and approximate data-memory unit (`a.LSU`).

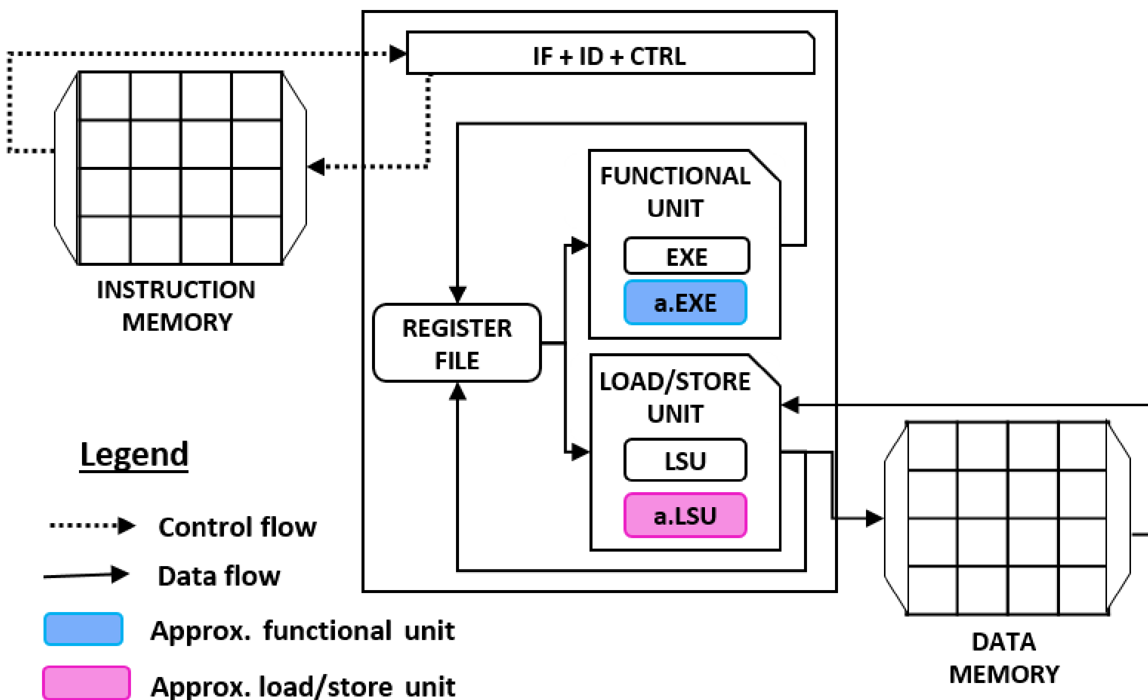


Figure 3.1: Our extended RISC-V Architecture

The proposed reduced width units support multiple formats which are charac-

terized by a number of MSBs b; $b \in \{1, ..., B\}$; B is the maximum possible width. The width of the unit is configured with `a.set.b` to set the width of the unit and `a.get.b` to get the current width of the unit.

In our investigation, we aim to evaluate the global energy reduction of applications executed in a complete processor with reduced width units. For energy evaluation, we propose an energy model for each class of instruction.

3.3 Energy model per instruction class

The energy model per instruction is constructed using measurements on a 28nm FD-SOI test-chip. This chip includes a processor core that follows the architecture described on Figure 3.1, connected to a 256KB memory, split into 64KB of instruction memory and 192KB of data memory. We consider two types of instructions: the full width instructions and the reduced width instructions. The energy values for the full width instructions are obtained from internal measurements on the this test-chip. We construct the energy values of the reduced width instructions with power models from the literature [61] that use the same technology.

3.3.1 Energy model for full width instructions

The energy values of full width instructions of the RISC-V ISA, relative to multiplication, are presented on Table 3.3. For each instruction class, the energy is measured for a set of random operands. The energy consumption variations due to the input data are under 15%, and hence in what follows we use an average value for each instruction class. The memory is implemented in a low voltage technology, which explains why the `ld` and `st` instructions consume less energy than a multiplication instruction.

3.3.2 Energy model for reduced width instructions

The full width instructions energy values are presented on Table 3.3. For the reduced width multiplication and arithmetic and logic instructions, the energy values are obtained by considering that the energy consumption of the a.EXE part varies with the width as specified in [61], whereas the energy consumption of the other

Table 3.3: Relative energy values of the full width instructions

Classes of instructions	relative energy values
`add`	0.69
`sub`	0.85
`mul`	1.00
`lgc`	0.67
`br`	1.56
`st`	0.78
`ld`	0.71

core parts are the same as in the full width case. For the reduced width data-memory instructions, we consider that 40% of the full width energy is not scalable, and the 60% varies linearly with the width. Note that this percentage is dependent on the design of the memory. Setting it to a value representative for another memory implementation does not invalidate our investigation method. The implementation of the reduced width operators introduces an energy cost overhead. This overhead is caused by the extra elements needed to partition the operators into several threshold voltage domains, and it is taken into account in our model.

Let e_{A_c} be the energy consumption of the reduced width computations instructions, *i.e.* the energy consumed in the `a.EXE` part and the other full width core parts, and e_{A_m} the energy consumed by the reduced width data-memory instructions, *i.e.* in the `a.LSU` part and the other core parts. With $b = B$, $e_{A_c}(B)$ and $e_{A_m}(B)$ are calculated with the following formulas:

$$e_{A_c}(B) = e_{\bar{A}_c} + o_c \tag{3.1}$$

$$e_{A_m}(B) = e_{\bar{A}_m} + o_m \tag{3.2}$$

where $o_c > 0$ and $o_m > 0$ are the energy overheads when implementing the reduced width computation and data-memory instructions, respectively.

The energy values e_{A_c} for each b are constructed as follows. We had access to: (1) measurements of $e_{\bar{A}_c}$ (which corresponds to full width) and (2) numbers in simulation for `a.EXE` for all values of b, and o_c from CEA Leti's previous work [61]. Starting with these numbers, the values of e_{A_c} for each b are tabulated.

Furthermore, we estimate the energy value $e_{A_m}(b)$ with the following formula:

$$e_{A_m}(b) = e_{A_m}(B) \times (r + (1 - r) \times b/B), \tag{3.3}$$

where $r \in [0, 1]$ is a non-scalable ratio from the energy consumed by a data-memory instruction. The ratio r is dependent on the design of the memory. In this work we assume that 40% of the energy consumption of the a data-memory instruction is non-scalable with the width, *i.e.* $r = 0.4$.

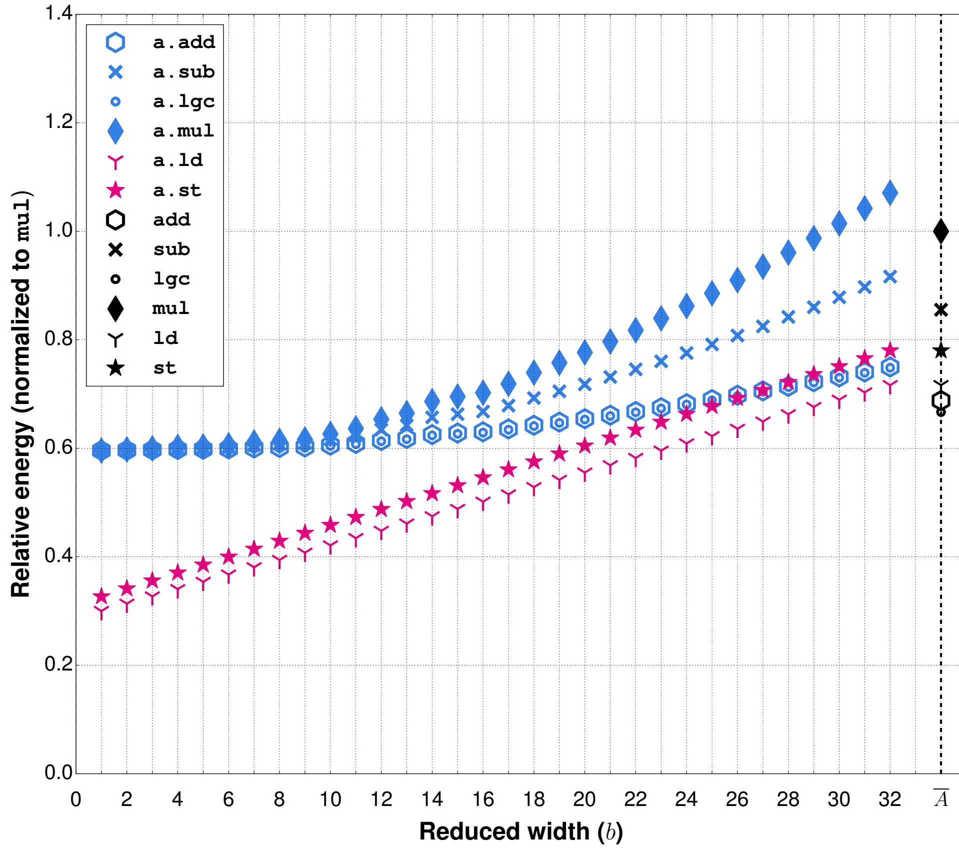


Figure 3.2: Relative energy values of reduced width and full width instructions

Figure 3.2 summarizes the energy values for reduced width instructions classes and full width ones. We note that the energy overhead is easily observable from 26 bits and above, *i.e.* the energy consumption of the reduced width units is higher than the energy consumption of the full width instruction. We can deduce that in

our extended RISC-V, computing with more than 26 bits is costly in terms of energy consumption, *i.e.* more than the full width computations, due to the width hardware management.

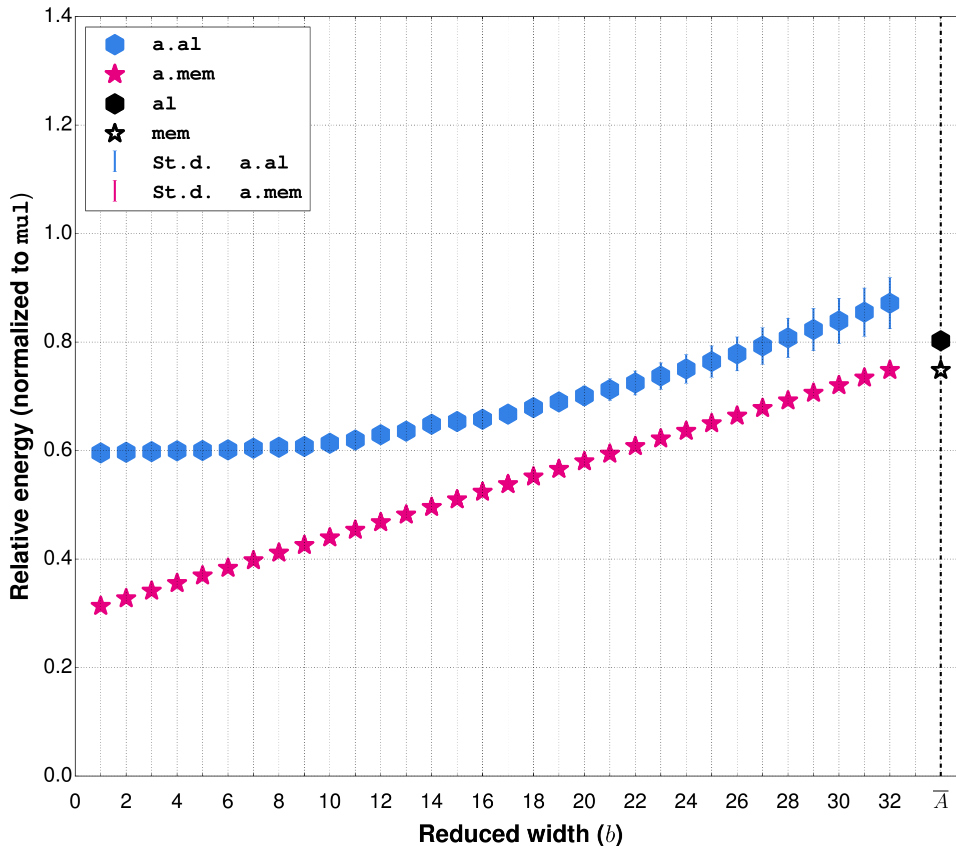


Figure 3.3: Average relative energy values of reduced width and full width instructions

Figure 3.2 indicates that for the reduced width operations for computations, *i.e.* `a.add`, `a.lgc`, `a.sub`, `a.mul` the energy values are close for widths in the range 1 to 8 bits and the energy values of the reduced width operations for memory access, *i.e.* `a.ld`, `a.st`, have not very large differences (absolute difference less than 0.09). Figure 3.3 presents, for both reduced width operations for computations and for data-memory accesses, the average energy values and the standard deviations for each width.

The relative standard deviation of the average energy values of the `a.add`,

`a.lgc`, `a.sub`, `a.mul` operations is less than 5%. For widths in the range 1 to 8 bits, the reduced width operations for computations have more or less the same energy consumption; the relative standard deviations of the energy values are less than 2% for widths in the range 9 to 18 bits, and between 2% and 5% in the range 18 to 32 bits.

The relative standard deviation of the average energy values of the `a.ld`, `a.st` operations is less than 1%. Note that the implementation of reduced width data-memory would probably not be efficient for width values that are not a power of two. However, for the sake of the investigation, we consider all width values from 1 to 32 bits.

We note that in our RISC-V architecture, the estimation of the energy reduction on a given application can be performed with average energy values, grouped into categories of instructions, *i.e.* `comp`, `mem`, `a.comp`, and `a.mem`. We will exploit the results of the Figure 3.2 in Chapter 4 and the results of the Figure 3.3 in Chapter 5.

3.4 Experimental environment

We propose a set of annotations and an instrumentation tool to ease the experimental investigations. We implement a compiler support to handle pragmas added in the source code and we extend the spike simulator [11] with the capabilities to return statistics including the number of instructions for each class and the global energy consumption of an application using our energy model. The statistics are input of scripts implemented to plot the graphs presented in Chapter 4 and Chapter 5.

3.4.1 Programming support for our extended RISC-V

The proposed reduced width operations may be executed with various width values. Two pragmas directives are implemented to handle the format of data representation in the source code and to delimit the part of the source code to execute with reduced width units.

#pragma `fixed`

The pragma `fixed` is proposed for fixed-point computations on applications. It manages automatically the adjustments of the scale factor in fixed-point multiplication and division operations. The pragma includes the following parameters:

- *ListOfVariables*: the list of the variables to be handled in the fixed-point conversion; the variables have to be integers and they may be local or global.
- *(Width, Frac)*: the total width and the fractional width for the conversion from floating-point to fixed-point types.

```
#include <stdio.h>
#define Width 30
#define Frac 10
#include <math.h>

typedef int fixed_point;

int float_to_fixed(float a)
{
   return (int)(a*pow(2, Frac));
}
float fixed_to_float(fixed_point a)
{
   return (float)(a/pow(2, Frac));
}

int main ( )
{
   float a = 0.44;
   float b = 0.23;
   float c = 3.51;
   float d = 0.0;

   fixed_point a_fp = float_to_fixed(a);
   fixed_point b_fp = float_to_fixed(b);
   fixed_point c_fp = float_to_fixed(c);
   fixed_point d_fp = float_to_fixed(d);
```

```
    #pragma fixed (a_fp, b_fp, c_fp, d_fp) (Width, Frac)
    {
       d_fp = a_fp*b_fp+c_fp;
    }
    printf("%f \n", fixed_to_float(d_fp)); //the result is
        equal to 3.610352
    return 0;
}
```

Listing 3.6: Pragma `fixed` example

#pragma `reducedWidth`

The pragma `reducedWidth` delimits the part of the source code to be executed with the reduced width units. The compiler replaces all the operators with the corresponding reduced width ones presented on Table 3.2. The parameters of this pragma are:

- *ListOfVariables*: the list of the variables involved in the reduced width operations; they may be local, global variables or function arguments in integer type.
- *Reduced_width*: allows to obtain the number of the MSBs for computation, the LSBs (equal to $Width - Reduced_width$) are set to 0 on the operands.
- RwTypePropagation (True/False): indicates what the compiler should do when the operation is between a variable declared as reduced width and a full width one. The option set to *True* indicates that the reduced width operation should be used; otherwise the full width operation is applied.

```
#include <stdio.h>
#define Width 30
#define Frac 10
#define Reduced_width 23 // 7 LSBs set to 0
#include <math.h>

typedef int fixed_point;
```

```
int float_to_fixed(float a)
{
   return (int)(a*pow(2, Frac));
}
float fixed_to_float(fixed_point a)
{
   return (float)(a/pow(2, Frac));
}

int main ( )
{
   float a = 0.44;
   float b = 0.23;
   float c = 3.51;
   float d = 0.0;

   fixed_point a_fp = float_to_fixed(a);
   fixed_point b_fp = float_to_fixed(b);
   fixed_point c_fp = float_to_fixed(c);
   fixed_point d_fp = float_to_fixed(d);

   #pragma fixed (a_fp, b_fp, c_fp, d_fp) (Width, Frac)
   {
      #pragma reducedWidth(a_fp, b_fp, c_fp, d_fp)
          Reduced_width True
      {
            d_fp = a_fp*b_fp+c_fp;
      }
   }
   printf("%f \n", fixed_to_float(d_fp)); //the result is
       equal to 3.606445
   return 0;
}
```

Listing 3.7: Pragma `reducedWidth` example

In Listing 3.7, an example explains how to use the pragmas in a source code.

The first step is to initialize the pragma parameters, e.g., *Width, Frac, Reduced_width*. Two functions are added in the source code to convert all floating-point variables into integer one before applying the pragmas. The pragma `fixed` allows to handle the adjustment in fixed-point multiplications and divisions. The pragma *reduced-Width* replaces all full width operations (**, +, /* in this example) by the reduced width ones.

3.4.2 Instrumentation tool

An instrumentation tool is implemented to investigate the instructions breakdown in a program, *i.e.* to compute the number of executed instructions for each class. Scripts are implemented to parse the file that contains the assembly code generated by the compiler. The number of instructions and the energy model per instruction class estimated in Chapter 3 allow to have an insight into the global energy consumption for a given application.

The energy consumption can be evaluated on a region of the application code. The evaluated region is delimited by the functions *stats_begin(id)* and *stats_end(id)* (Cf. example in listing 3.8). Several regions can be evaluated at a time, they are characterized by an `id` that is a parameter of *stats_begin()* and *stats_end()* functions.

```
#include <stdio.h>
#define Width 30
#define Frac 10
#define Reduced_width 23 // 7 LSBs set to 0
#include <math.h>
#include "statistics.h" // header added to know the statistics
    information on regions of a given application

typedef int fixed_point;

int float_to_fixed(float a)
{
  return (int)(a*pow(2, Frac));
}
```

```
float fixed_to_float(fixed_point a)
{
   return (float)(a/pow(2, Frac));
}

int main ( )
{
   float a = 0.44;
   float b = 0.23;
   float c = 3.51;
   float d = 0.0;

   fixed_point a_fp = float_to_fixed(a);
   fixed_point b_fp = float_to_fixed(b);
   fixed_point c_fp = float_to_fixed(c);
   fixed_point d_fp = float_to_fixed(d);

   stats_begin(1); // Delimit the regions to investigate
      #pragma fixed (a_fp, b_fp, c_fp, d_fp) (Width, Frac)
      {
         #pragma reducedWidth(a_fp, b_fp, c_fp, d_fp)
            Reduced_width True
         {
            d_fp = a_fp*b_fp+c_fp;
         }
      }
   stats_end(1);
   printf("%f \n", fixed_to_float(d_fp));
   return 0;
}
```

Listing 3.8: Simulator input source code

Listing 3.8 presents a source code example for the simulator to get profiling information. The profiler information includes the number of executed instructions per class, the energy consumption of the delimited region(s) by the *stats_begin()* and

stats_end() functions and the global energy consumption of the complete application. The statistics of the example in Listing 3.8 are in Listing 3.9:

```
Reg  Application
Energy: 4079.31366528334
add 61
a.add 1
a.mul 1
branch 4
load 37
logical 1
move 44
notmodelled 3
store 57

Reg 1
energy: 213.5803319502
a.add 1
a.mul 1
load 3
store 1
```

Listing 3.9: Example of the simulator output

In Listing 3.9, the region *Application* includes the evaluation of system calls, e.g. *printf*, *scanfs* and the evaluation of the user specified regions (*Reg 1* in this example).

Note that in the energy model proposed in Section 3.3, we do not estimate all instructions categories. We group these categories into a class named `notmodelled`, *e.g.* `la` (load address), `ecall` (used to make a request to the supporting execution environment), `auipc` (add a 20-bit upper immediate to pc). In our investigation, we evaluate only the operations included in the program algorithm, *i.e.* we do not include the system call functions in the application energy consumption evaluation. Given a source code, we delimit the regions excluding the system call functions. Finally, the percentage of `notmodelled` instructions in the profiling information is very small, less than 1%.

3.5 Chapter summary

In this chapter we have presented our platform for evaluation of the approximate computing technique proposed in this thesis. It includes an extended RISC-V processor and a simulation experimental environment. We have first presented the RISC-V ISA for both the standard full width instructions and the reduced width instructions. Second we describe our extended RISC-V architecture including reduced width units for computations and data-memory accesses. Third we present the energy model of each instruction class for both the full width ones and the reduced width instructions. Finally we present our experimental environment including a set of annotations to ease programming and an instrumentation tool. The above tools allow the evaluation of the reduced width units on applications in terms of output quality and energy reduction.

CHAPTER 4

EVALUATION OF REDUCED WIDTH UNITS ON APPLICATIONS

In this chapter we investigate the potential of reduced width units in terms of energy reduction on applications executed in a complete processor. The studied reduced width units are reconfigurable at runtime. The evaluation is performed on benchmark applications proposed in the state of the art. These applications are originally implemented in floating-point format. In the context of this thesis we target integer architectures to overcome embedded systems constraints in terms of energy, silicon area and/or computation time.

In our investigation, we first evaluate the impact of the conversion from the floating-point to fixed-point representation in the applications output quality before starting the energy estimation. The errors are computed with respect to the reference floating-point outputs. To estimate these errors, we perform simulation on a Intel core using the `Libfi` library [4].

Second we evaluate the potential in terms of energy reduction of some common approximate adders and multipliers on the fixed-point applications. The evaluated approximate adders and multipliers are reduced width units that can be reconfigurable at runtime. In a first study, all the memory accesses are full width. Furthermore, for more energy efficiency, we extend the study with width configuration in the data-memory accesses. We evaluate the output quality *vs* energy reduction trade-off. In our reduced width units, the computations are performed on a number of most significant bits (MSBs) and for data-memory access, only a number of MSBs is loaded/stored from/in the memory. For the energy evaluation we count the number of executed instructions and with the energy model proposed in Chapter 3, we estimate the global energy consumption of the applications.

This chapter is organized as follows. Section 4.1 describes the methodology. Section 4.2 evaluates the impact of the conversion from the floating-point to fixed-

point representation on the applications output quality. Section 4.3 presents the evaluation results on applications executed with only reduced width units for addition and multiplication. Section 4.4 investigates the impact of reduced width units for both computations and memory accesses on the output quality and on the energy reduction of the applications executed on the extended RISC-V processor. Section 4.5 summarizes the chapter.

4.1 Methodology

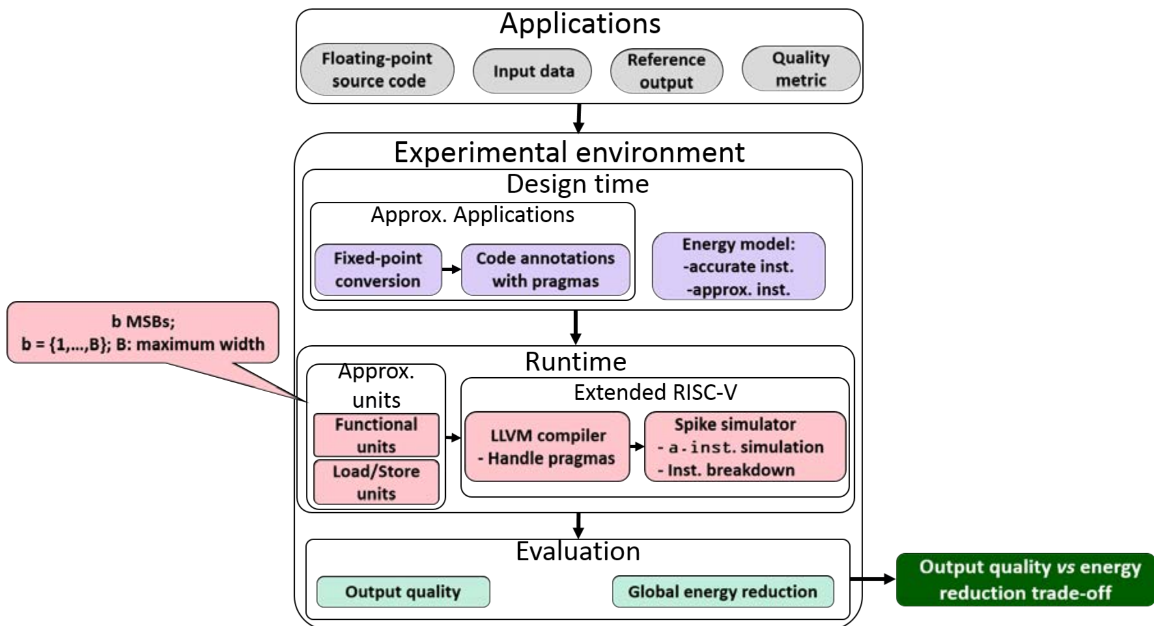


Figure 4.1: Methodology of output quality *vs* energy reduction trade-off evaluation.

The impact of reduced width units on applications output quality and energy consumption is evaluated using the flow proposed on Figure 4.1. We first select benchmark applications that provide a large set of input data and relevant quality metrics for evaluation. Second we implement the experimental environment that aims to make easier the output quality *vs* energy reduction trade-off evaluation of the applications. We convert the floating-point applications in fixed-point ones for the purpose of ensuring that the applications are suitable for computations in an integer processor. With pragmas, we annotate the source code to delimit the operations that can be executed with reduced width units without high

quality degradation. For energy evaluation, we estimate the energy consumption for each instruction (both accurate and approximate), as presented in Chapter 3. Third we extend the RISC-V processor [89] with reduced width units for both computations and memory accesses. The pragmas added in the fixed-point source code are handled by new passes implemented in the LLVM compiler. The RISC-V spike simulator [11] is augmented with the capabilities to execute the reduced width instructions and to estimate for a given application source code, the number of instructions executed for each class. For each instruction class, this number is multiply with the energy value of the instruction class. We propose to use the sum of the energy values for each instruction class to estimate the global energy consumed by the application.

4.2 Impact of the fixed-point conversion on the applications output quality

The fixed-point conversion is required on the evaluated applications to investigate how the output quality is impacted by the rounding modes and the representation format, *i.e.* the number of fractional widths. To convert the benchmark applications from floating-point to fixed-point, we use the fixed point library `libfi` that includes several fixed-point formats and several rounding modes. The `libfi` rounding modes are presented in the subsection 2.1.1, *i.e.* rounding towards 0 (Fix), rounding towards $+\infty$ (Ceil), rounding towards $-\infty$ (Floor), rounding to nearest. For the rounding to nearest, 3 possible cases are proposed: the number can be away from 0 (Classic), an even number (NearEven), or an odd number (NearOdd).

4.2.1 Jmeint application

Figure 4.2 presents the evaluation results of the errors introduced by the conversion from floating-point to fixed-point for the jmeint application from Axbench [93], in various rounding modes and various fractional widths.

The evaluation of jmeint in fixed-point representation is performed on 1 000 000 couples of triangles in the fixed-point format $Qw.f$ with w = 32 bits, f in the range $\{1, 2, \ldots, 30\}$ and the integer part i = 2 bits. We note that from 14 bits and

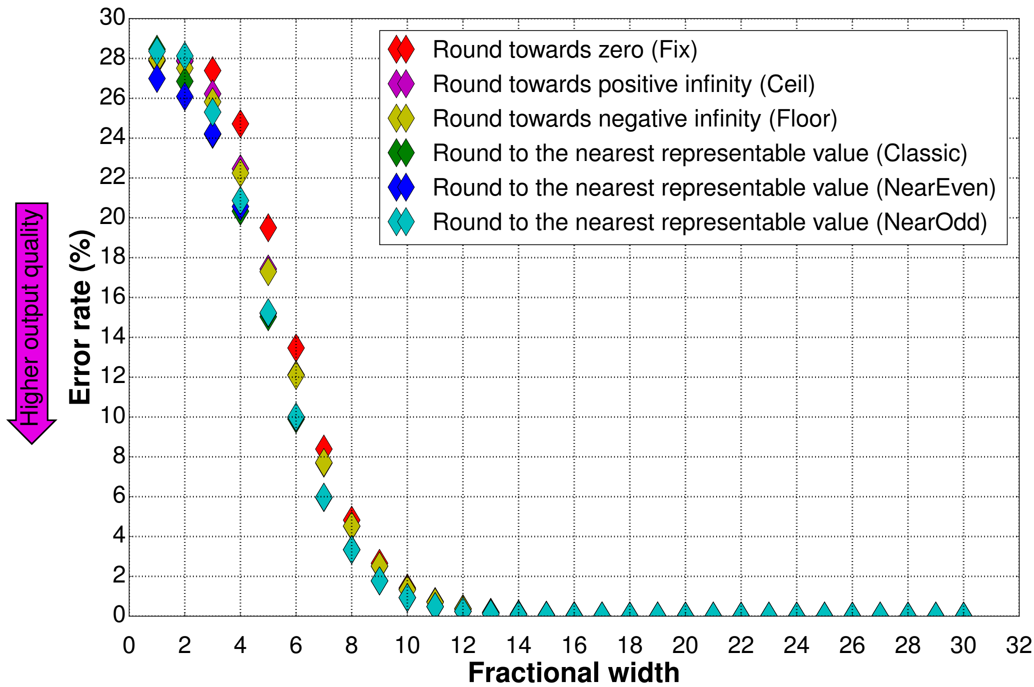


Figure 4.2: Errors evaluation on fixed-point jmeint.

above in the fractional part, the error rate is close to 0, *i.e.* for all the 1 000 000 couples of triangles, the output computed with the original floating-point version and with the fixed-point one are almost the same, with an error rate $\leq 0.03\%$. The error rate is deduced from the Hamming distance by comparing one by one the elements of the boolean vectors of size 1 000 000, returned by the floating-point and the fixed-point applications. For fractional widths higher than 5 bits, we note that the *round to the nearest representable value (NearOdd)* is the best rounding mode for jmeint in terms of output quality. From this experiment, we can deduce that jmeint requires to be executed with at least width = 16 bits for an acceptable output quality.

4.2.2 Sobel filter application

Figures 4.3, 4.4, and 4.5 report the errors evaluation results of the Sobel filter application executed in fixed-point representation with various rounding modes and various fractional widths. The errors are estimated with three common quality met-

rics used in the image processing applications. These metrics computed for output quality evaluation are the root mean square error (RMSE), the peak signal to noise ratio (PSNR) and the structural similarity (SSIM). Each metric value is computed with parameters of the image returned by the reference floating-point program and the image returned with computations in fixed-point in various rounding modes and various fractional widths.

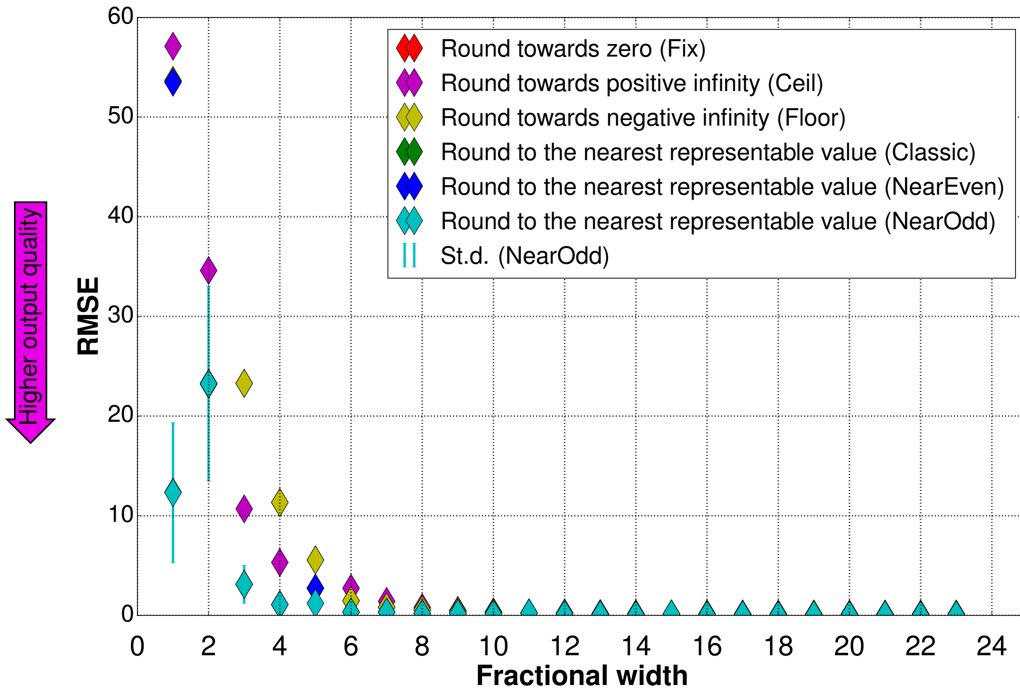


Figure 4.3: RMSE evaluation on fixed-point Sobel filter.

The experiments are performed on 100 input images selected randomly, in the fixed-point format $Qw.f$ with w = 32 bits, f in the range $\{1, 2, \ldots, 23\}$ and the integer part i = 9 bits. We estimate the mean values and the standard deviation for each quality metric: the RMSE and the PSNR. The standard deviation (std) indicates that, for each fractional width, the metric value varies depending on the input image.

Figure 4.3 presents, for each fractional width, the average value and the standard deviation of the RMSE computed with 100 images. For the RMSE error metric, the lower the RMSE value, the higher the output quality is. The results indicate that from 10 bits and above in the fractional part, we have an acceptable output

image, *i.e.* RMSE ≤ 0.01.

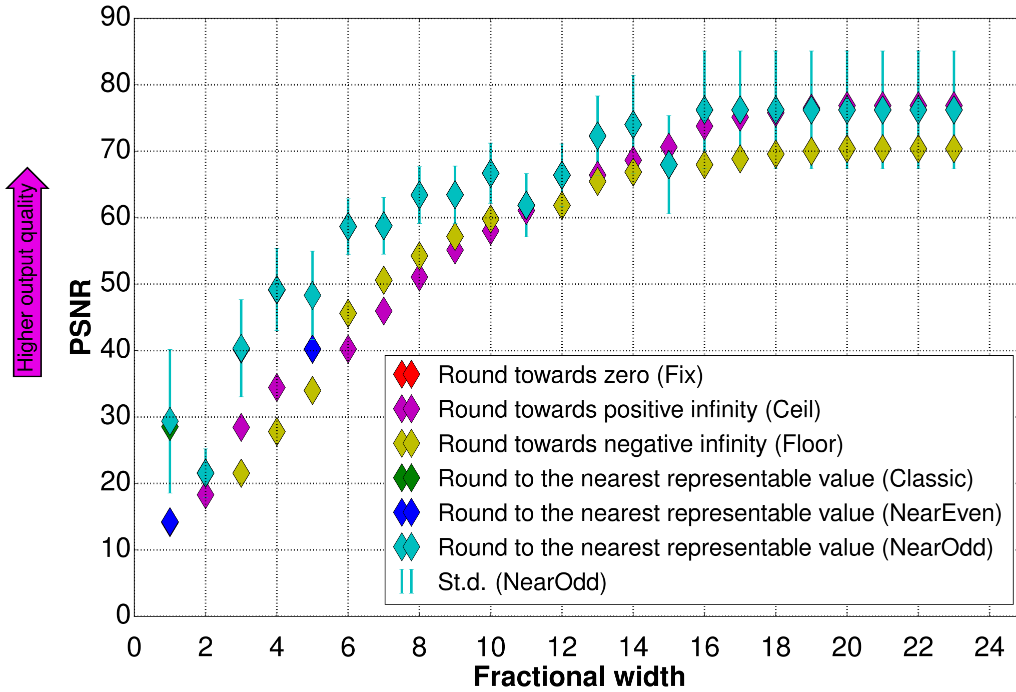


Figure 4.4: PSNR evaluation on fixed-point Sobel filter.

Figure 4.4 presents, for each fractional width, the average value and the standard deviation of the PSNR computed with 100 images. For the PSNR error metric, the higher the PSNR, the higher the output quality is. The results indicate that from 10 bits and above in the fractional part, we have a PSNR higher than 50dB, which is considered as a good quality.

The RMSE and the PSNR estimate the absolute errors between pixels of two compared images, *i.e.* the image computed by the floating-point Sobel filter program and the image returned by the fixed-point one. However these two metrics are less correlated to human perception of image quality compared to the SSIM [88].

The SSIM evaluates the similarity between the reference image, *i.e.* the image computed with the floating-point program and the image returned by the fixed-point one. For the SSIM metric in image processing, the values are in the range $[0, 1]$. If the SSIM is equal to 1, the two compared images are identical. On Figure 4.5, we note that from 8 bits and above in the fractional part, the two images are quite

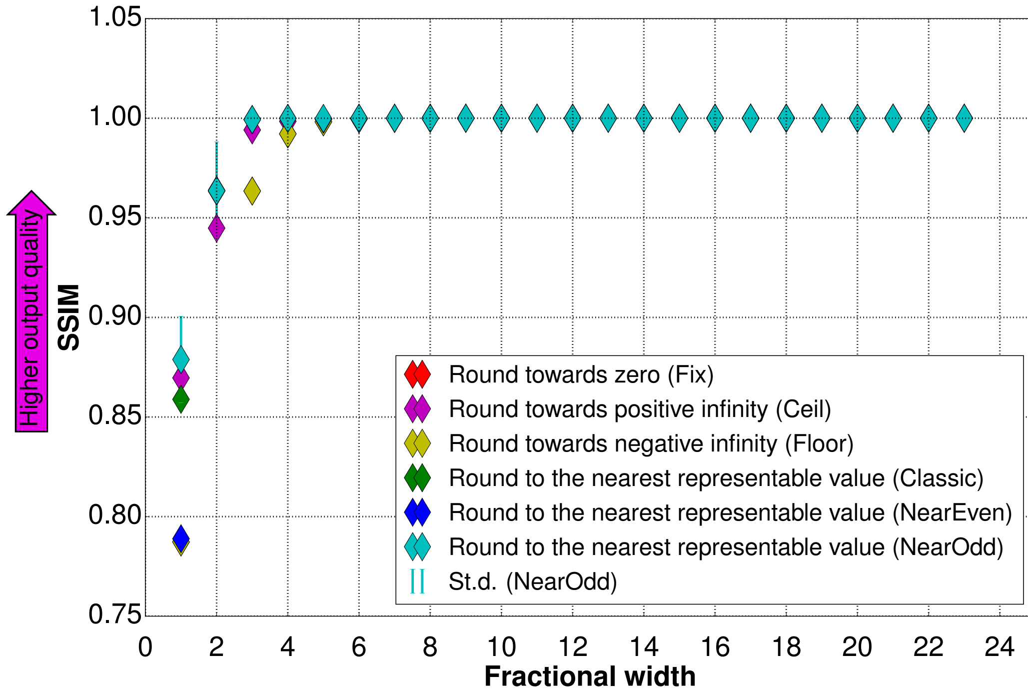


Figure 4.5: SSIM evaluation on fixed-point Sobel filter.

similar, *i.e.* SSIM ≥ 0.999.

4.2.3 Forwardk2j application

The forwardk2j takes as input the angles of a 2-joint robotic arm and computes the position of its end effector. To evaluate the errors induced by the fixed-point computations with various rounding modes and various fractional widths, we use the relative error metric.

The experiment is performed on 10 000 random couples of angles. For each couple of the computed coordinates, the maximum value between the errors of the two coordinates is returned. The mean values and the standard deviation of the 10 000 maximum errors are computed. Figure 4.6 indicates that from 14 bits and above in the fractional part, the relative error is close to 0, *i.e.* $\leq 0.01\%$. We note that for low precisions, *i.e.* fractional widths in the range $\{1, 2, \ldots, 10\}$, with the same number of fractional widths the relative error varies depending on the rounding mode. From 7 bits and above, the *round to the nearest representable value*

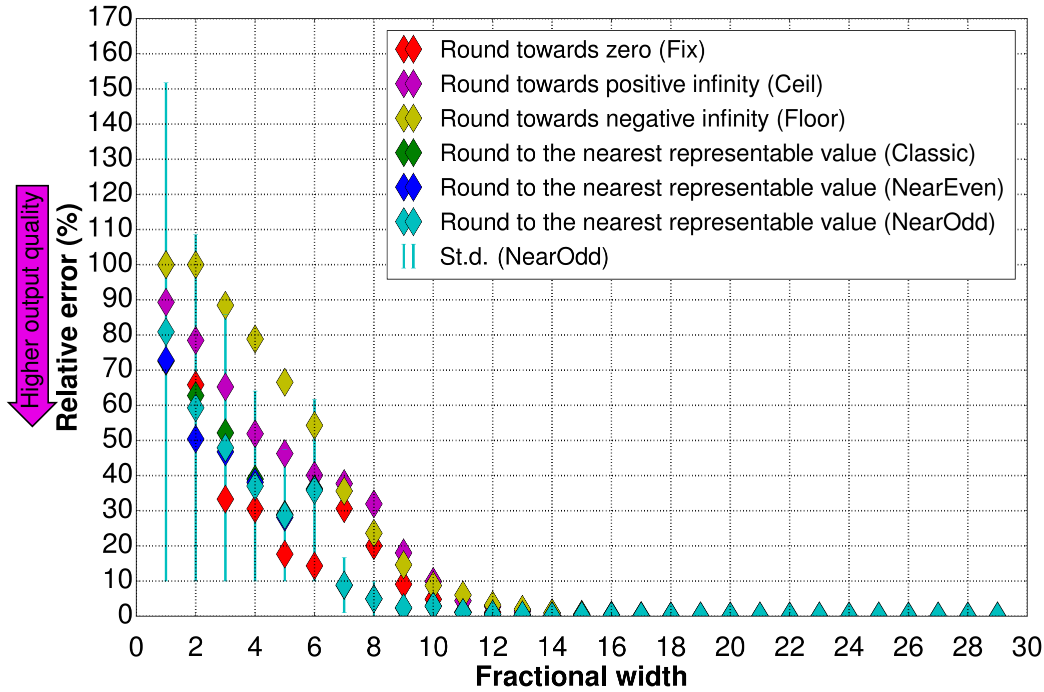


Figure 4.6: Errors evaluation on fixed-point Forwardk2j.

(NearOdd) is the best rounding mode for Forwardk2j in terms of output quality.

4.3 Impact of reduced width units for addition and multiplication on applications energy consumption

Our first experiments on the energy evaluation are performed with the reduced width units for addition and multiplication, which are the most investigated in the state of the art. The global evaluation is performed on applications executed on the RISC-V processor extended with the evaluated approximate operators (*i.e.* `a.add` and `a.mul`). We performed the experiments on three applications that are jmeint, Sobel filter and forwardk2j. For the energy consumption evaluation of an application, we first study the instruction breakdown to compute the number of executed instructions for each class.

4.3.1 Instruction breakdown

We study the instruction breakdown to compute the number of executed instructions for both reduced width instructions and full width instructions. In this section we investigate the most common studied approximate operators in the literature, *i.e.* adders (`a.add`) and multipliers (`a.mul`).The notations `a.add` and `a.mul` refer to reduced width units for addition and for multiplication, respectively.

Jmeint application

The experiments are performed with 10 000 couples of triangles provided in the Axbench suite. In jmeint the output is a boolean value which indicates whether the two triangles intersect or not (1 or 0). Jmeint has several possible exit points, as presented on Figure 2.3. The number of executed instructions could be different depending on the input data value and on the width value, as shown on Figure 4.7.

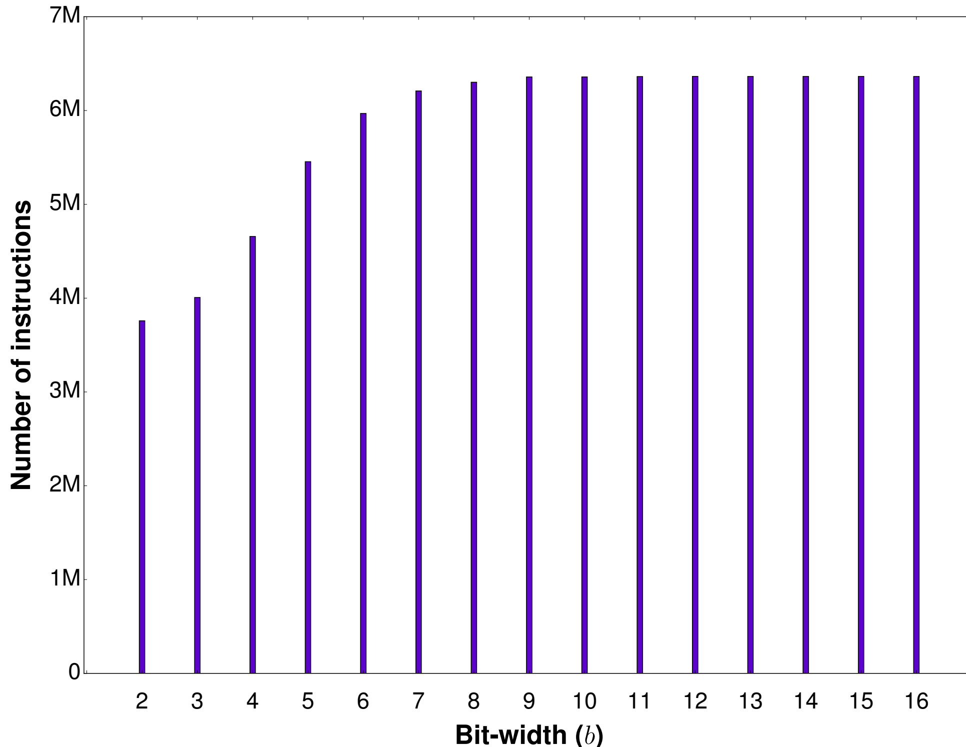


Figure 4.7: Number of executed instructions

In very low precision, *i.e.* width in the range $\{1, 2, \ldots, 8\}$, the number of executed instructions varies from one input to another depending on the width. Figure 4.7 indicates that from 9 bits and above, the execution of all 10 000 input data seems to respect the actual program exit points, *i.e.* the floating-point and the fixed-point programs have the same exit point for a given input data. The instruction breakdown is performed by computing the mean value of the executed instructions and the standard deviation, as indicated on Figure 4.8. Jmeint includes 11.82% of reduced width additions and multiplications.

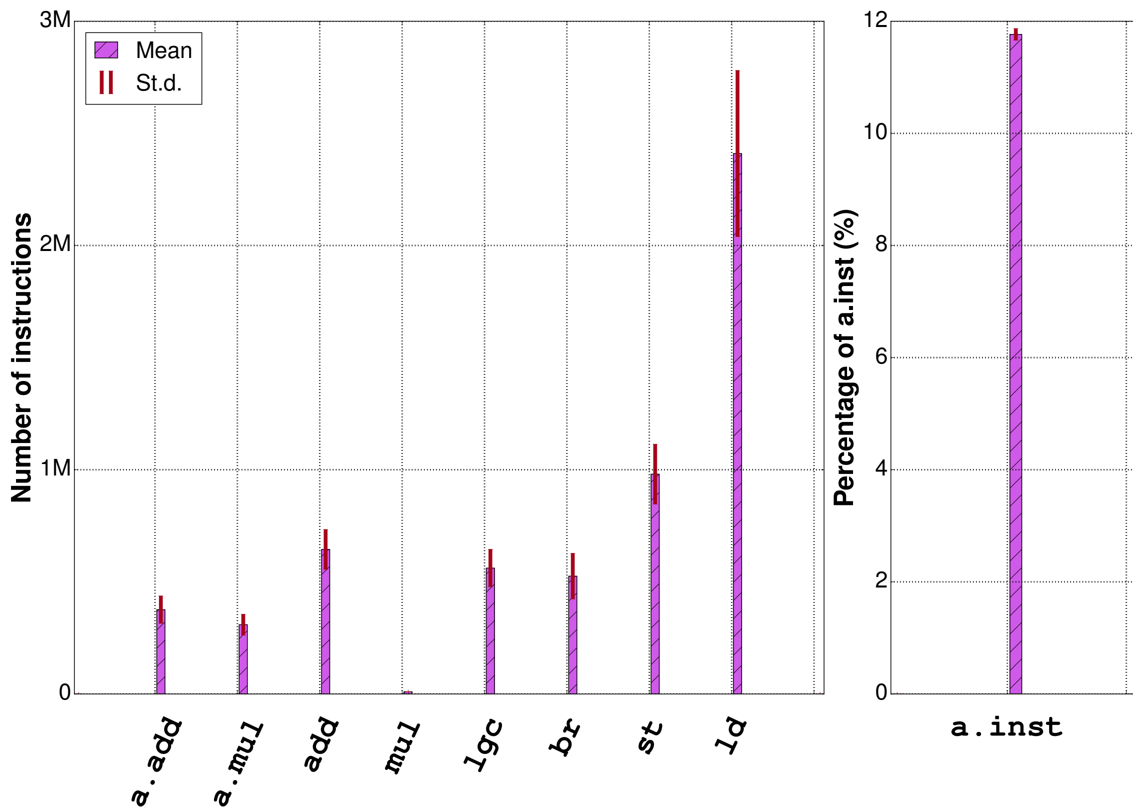


Figure 4.8: Jmeint instruction breakdown.

Sobel filter application

We applied several optimizations typical for the low power domain. Note that these optimizations do not impact the output quality. The optimizations are stacked-up one after the other, as follows:

- V1.: 2D vector for pixel storage is replaced by 1D vector to reduce the performance costs due to address computations.
- V2.: V1 + the redundant code lines are removed. The 3 components of a pixel have the same value in gray-scale image, the same value is set to the 3 components.
- V3.: V2 + the loops for convolution computations are unrolled to reduce the instructions introduced by the loop indexing.
- V4.: V3 + filtering with null values are removed. To apply the filters on the pixels that are at the border of the image, in the original version, the image is padded with zeros. In this optimization we rewrite the filtering loops to remove this padding.

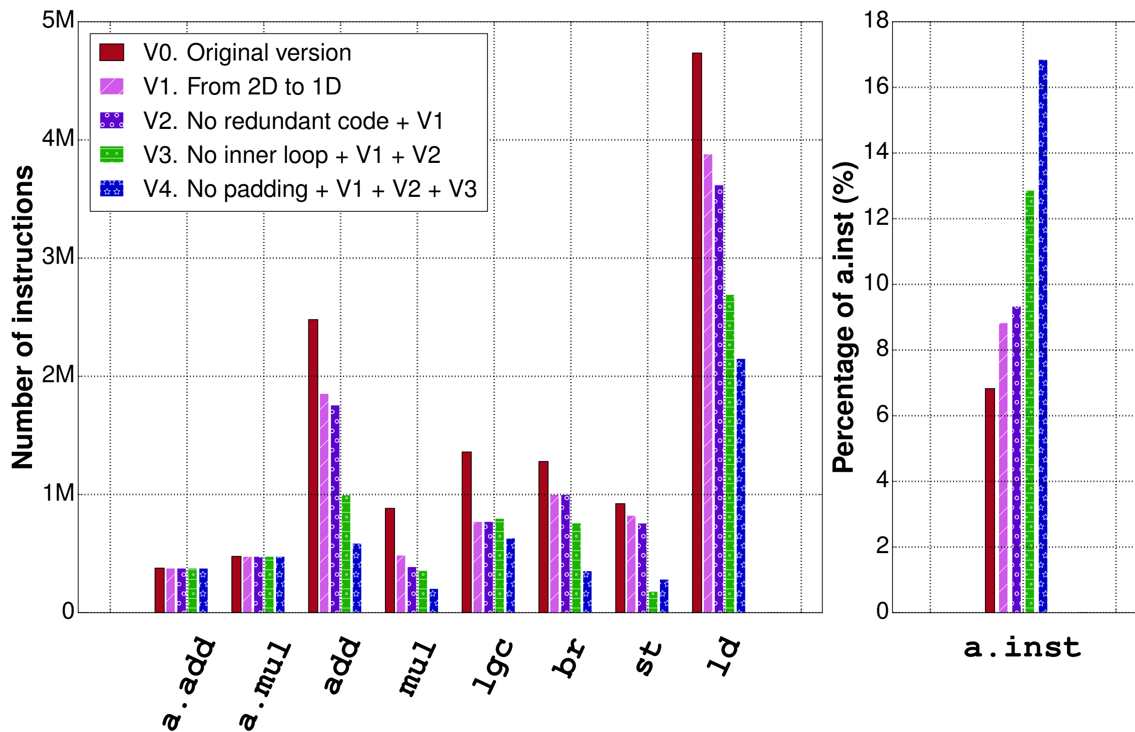


Figure 4.9: Sobel instruction breakdown.

Figure 4.9 presents the effect of these optimizations in terms of instruction breakdown. The total number of executed instructions is reduced by up to $2.6\times$ in the most optimized program, compared to the original one. The percentage of reduced width instructions (additions and multiplications) is equal to 6.8% in the original version of the program and 17% the most optimized program, V4. Despite

optimizations, most instructions are still full width (83% in the most optimized one). These instructions consist of full width instructions such as arithmetic instructions for address computation, data-memory access and branch instructions. The high percentage of the load instructions is due to the fact that the pixels of the image are stored into the data-memory and to compute the gradient of one pixel, the 3×3 matrix elements of the two filters and ones of the region of the pixel are loaded several times from the data-memory.

Forwardk2j application

Forwardk2j computes the position of a robotic arm. The following experiments are performed with 10 000 couples of 2-joint robotic arm. Figure 4.10 presents the instruction breakdown of forwardk2j. The forwardk2j application includes 46% of reduced width additions and multiplications. We note that the forwardk2j application includes more reduced width additions and multiplications, than the Sobel filter and jmeint applications. Hence it might be have more energy reduction than these two application.

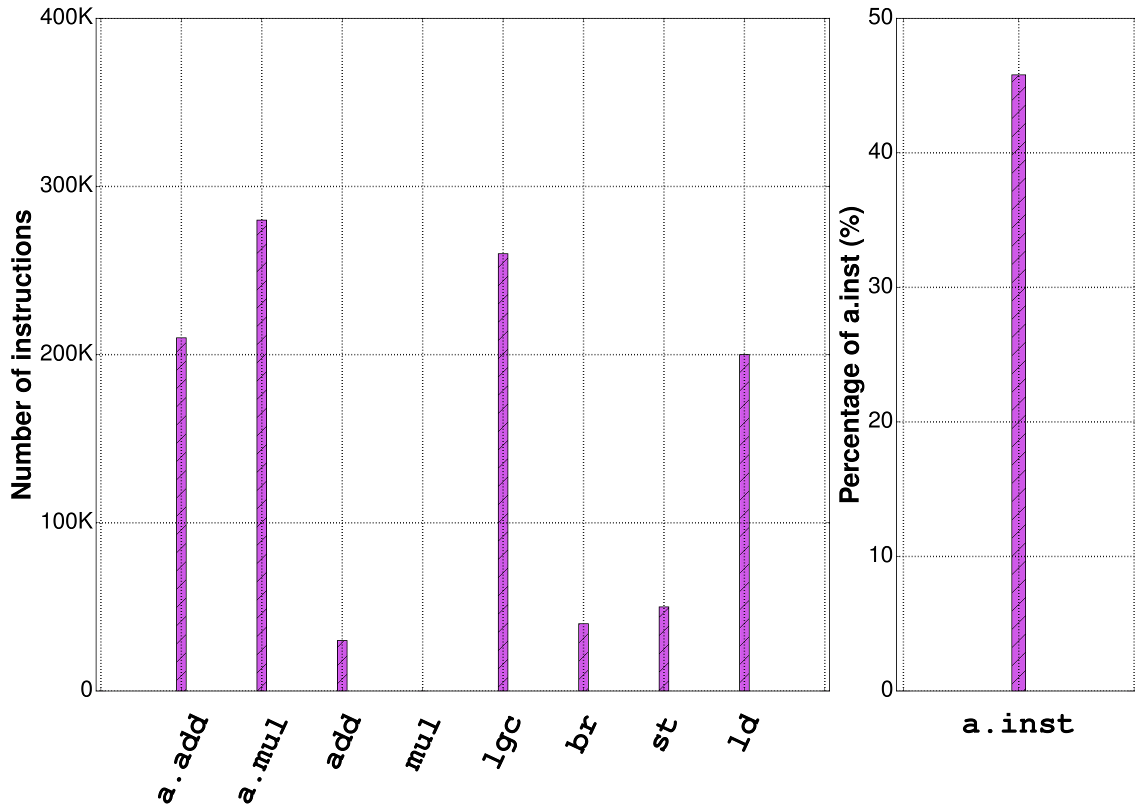


Figure 4.10: Forwardk2j instruction breakdown.

4.3.2 Energy evaluation

In the state of the art, the energy reduction obtained with the approximated adders and multipliers are evaluated separately, *i.e.* not on a complete application executed in a processor. The evaluation of these individual approximate operators returns high energy reduction, *e.g.* up to 58% in [35]. We aim to evaluate the reduced width adders and multipliers on complete applications executed in a processor in order to have an insight into the global energy reduction. Sobel filter and jmeint include less than 20% of reduced width additions and multiplications (*i.e.* `a.add` and `a.mul`); and forwardk2j includes more `a.add` and `a.mul` (*i.e.* up to 46%) than Sobel filter and jmeint applications.

Using the energy model per instruction class on Chapter 3 and the instruction breakdown studied on Subsection 4.3.1, we evaluate the energy consumption of these applications.

Jmeint application

The energy consumption can be reduced by up to 44% for width = 1 bit. The high energy reduction does not depend only on the reduced width computation but also on the reduced number of executed instructions on early rejection tests. As indicated on Figure 2.3, jmeint program performs several tests to indicate if two 3D-triangles intersect or not. Two early rejection tests are performed to avoid computations on triangles that can never intersect. For example, if one triangle lies on one side of the other triangle plan, these two triangles can never intersect (Cf. (1) and (2) exit points on Figure 2.3). The number of executed instructions in the program path related to (1) and (2) exit points on Figure 2.3 is less than the number of executed instructions in the other paths, *i.e.* (3), (4), (5), (6), and (7). For several couples of triangles from the 10 000 couples, the exit points are (1) or (2) in the range $\{1, 2, \ldots, 8\}$. Hence, the energy consumption of the whole application when a couple of triangle follows the (1) and (2) paths is less than its energy consumption with (3), (4), (5), (6), and (7) paths. If we consider the actual exit points of the jmeint application, (*i.e.* from 9 bits and above) and evaluate only the energy reduction due to the reduced width, the global energy reduction is less than 5%, as indicated on Figure 4.11.

Sobel filter application

Figure 4.12 presents the result of the energy consumption evaluation of the Sobel filter application with only `a.add` and `a.mul`. The results indicate that the energy can be reduced no more than 7%. The low energy reduction compared to the evaluation of the operators in a stand-alone context, is due to the fact that the application includes more than 80% of full width instructions and each instruction (for both reduced width and full width instructions) has an energy value that is not scalable with the width, as indicated on Figure 3.2.

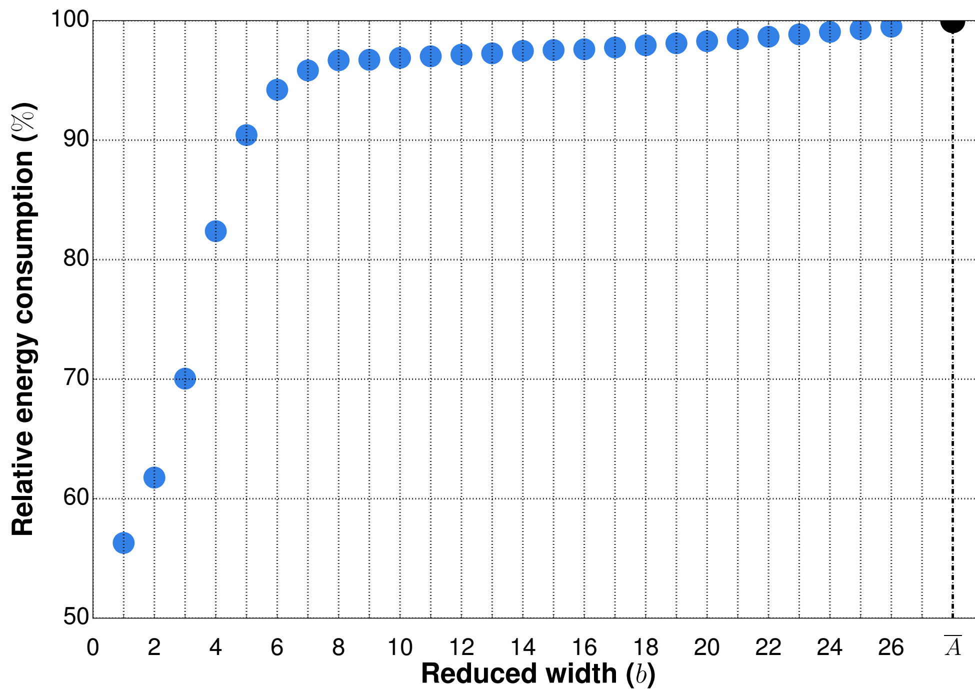


Figure 4.11: Jmeint energy consumption with `a.add` and `a.mul`.

Forwardk2j application

Forwardk2j computes the coordinates of an end-effector of a robotic arm. The output is a couple of coordinates (x;y). Forwardk2j is more computation intensive than Sobel filter and jmeint. Figure 4.10 indicates that the forwardk2j program includes more additions and multiplications (46%) that can be executed with reduced width than Sobel filter $(17\%$, Cf. Figure 4.9) and jmeint $(11.82\%$, Cf. Figure 4.8) programs. Hence its energy reduction $(19\%$ as indicated on Figure 4.13) is higher than the energy reduction of the Sobel filter (7%) and jmeint $(4\%$, if we consider the actual exit points) applications.

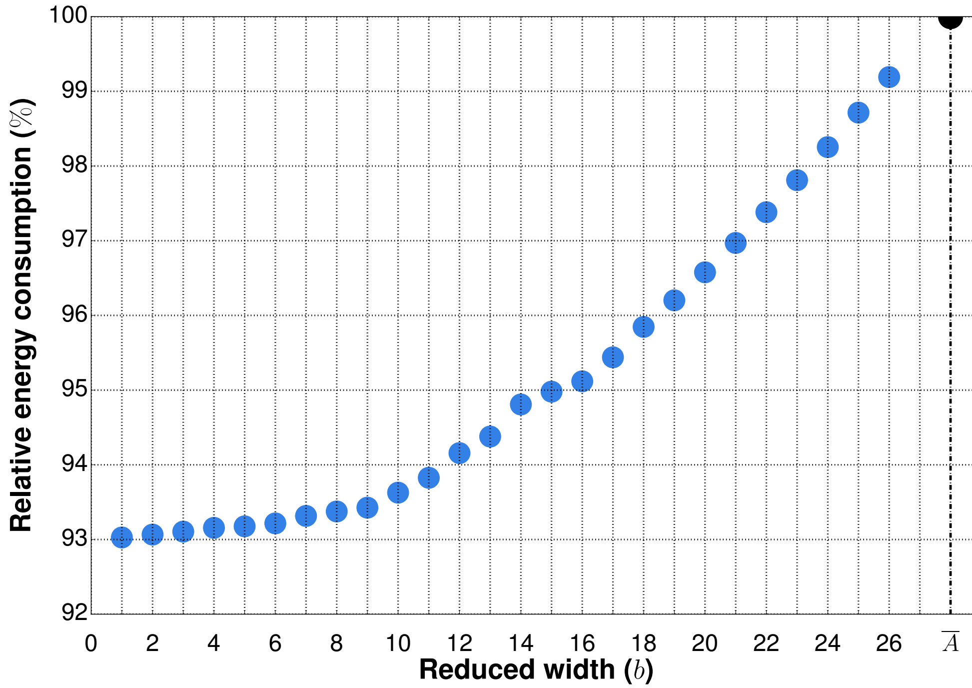


Figure 4.12: Sobel filter energy consumption with `a.add` and `a.mul`.

In summary, in the above experiments, we consider only the reduced width units for addition and multiplication that are the most investigated operators in the literature. Table 4.1 presents an overview on the energy reduction of these operators when evaluated separately and the global energy reduction when evaluated on an application executed on a complete processor.

Table 4.1: Energy evaluation of stand-alone reduced width operators and their use in a RISC-V processor

Solution	Operators / Programs	Energy reduction	Accurate bits
Stand-alone operators	Adder 32-bit [38]	30%	4
	Multiplier 16-bit [35]	58%	6
	Multiplier 16-bit [61]	39%	10
Operators embedded in a RISC-V processor	Sobel filter	6%	10
	Jmeint	4%	10
	Forwardk2j	16%	10

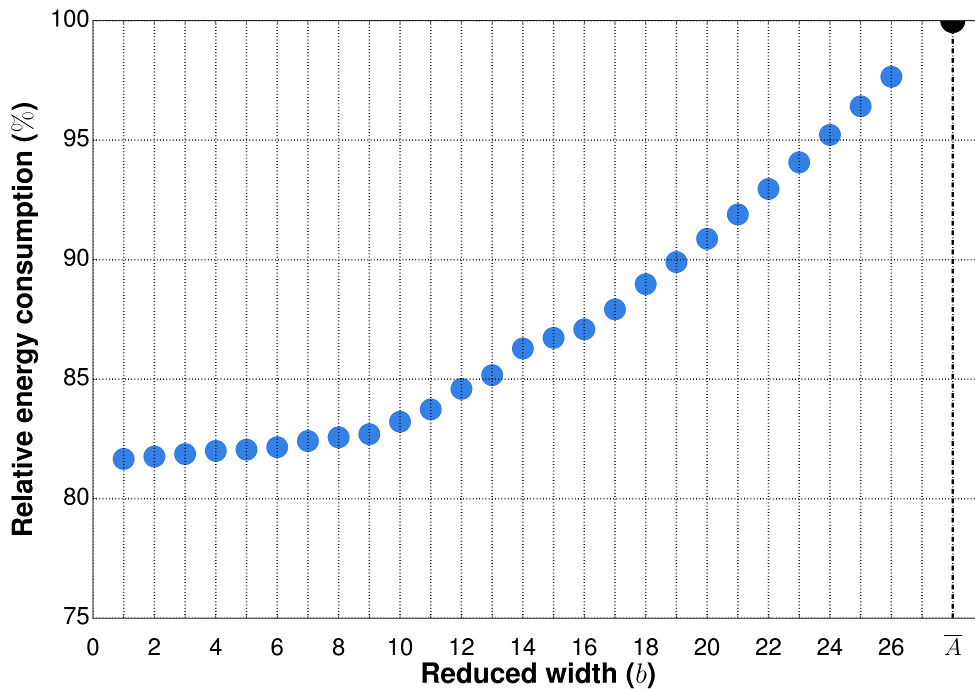


Figure 4.13: Forwardk2j energy consumption with `a.add` and `a.mul`

These results indicate that, even after code optimization, the energy reductions are not as large as the ones obtained evaluating the individual operators separately, as we can see in Table 4.1. The low energy reduction is due to the fact that the applications include several categories of instructions, not only additions and multiplications which can be approximated. Sobel filter and jmeint include more than 80% of full width instructions including all logic, branch and memory instructions, and full width arithmetic instructions for *e.g.* loop indexing, address computations. The forwardk2j application includes more approximated additions and multiplications: 46%, hence the energy reduction is higher (*i.e.* 19%) than one of the Sobel filter (*i.e.* 7%) and jmeint (*i.e.* 4%, if we consider the actual exit points). To improve the energy reduction achieved by using configurable width units, the width reduction principle is extended to the memory accesses.

4.4 Impact of the reduced width units for both computations and memory accesses on energy consumption and output quality of applications

The approximation of only `add` and `mul` computation units seems not sufficient to reach a high energy reduction. In the instruction breakdown investigation, we note that each application includes several instructions for logic and data-memory access that are not executed with reduced width. The data-memory instructions could be executed with a reduced width, *i.e.* load/store a number of MSBs from/in the memory. We extend the reduced width approach on logic and memory units and we evaluate them in what follows.

4.4.1 Instruction breakdown

We investigate the category of the executed instructions on jmeint, Sobel filter, and forwardk2j applications. The instructions are divided in two types: the full width ones and the reduced width ones. The first type of instructions are the full width ones corresponding to the full width arithmetic and logical instructions (`al`) for *e.g.* address computation, loop indexing, control statement; the full width multiplications (`mul`); the full width memory instructions (`mem`) to read instructions and to read/store data related to full width instructions; and branch instructions (`br`). The second type is the reduced width instructions that are only the reduced width arithmetic and logical instructions (`a.al`), the reduced width multiplications (`a.mul`) and the reduced width data-memory instructions (`a.mem`) that load/store data related to the reduced width computation instructions from/in the data-memory.

Jmeint application

As Sobel filter, in the first version of the energy evaluation with only width reduction on the additions and multiplications, we note low energy reductions. When the reduced width is extended to logic and memory instructions, the number of reduced width instructions increases (from 11.82% to 77.4%), hence the energy reduction could be improved. Due to the difference in the number of executed instructions that depends on the width and on the triangle coordinates, for each instruction

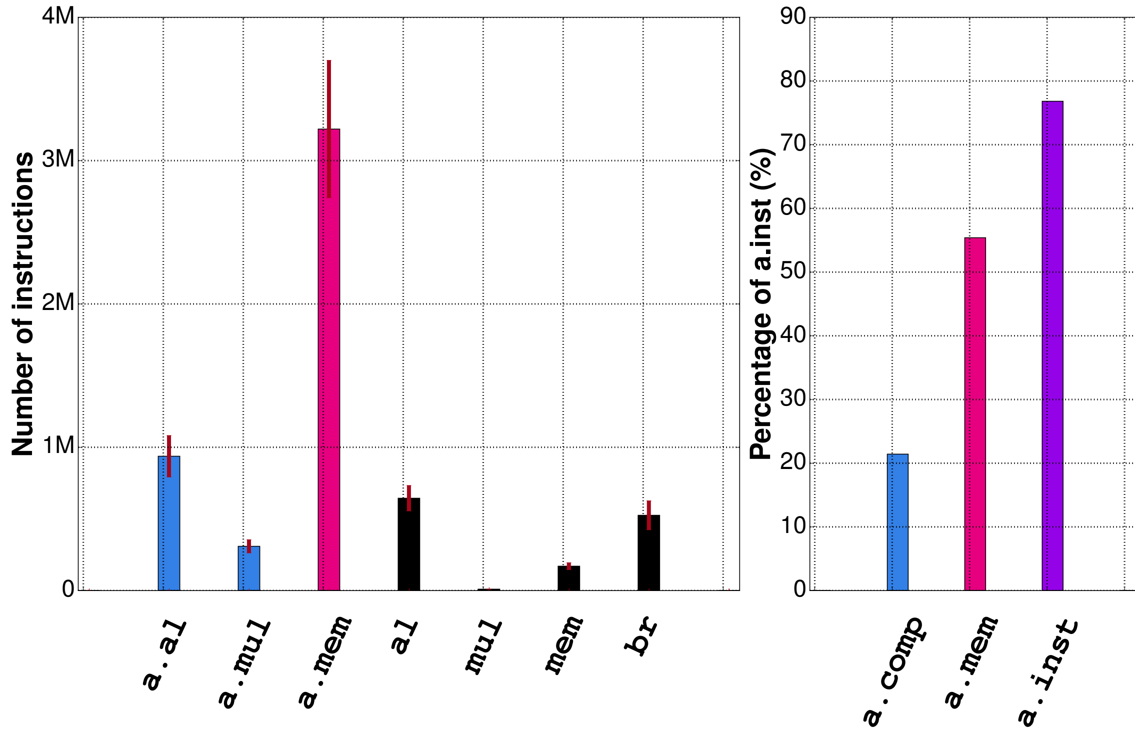


Figure 4.14: Jmeint instruction breakdown.

category we estimate the mean value and the standard deviation of the number of executed instructions.

Figure 4.14 indicates that jmeint includes 77.4% of instructions that can be executed with the reduced width units with 21.1% of reduced width computation instructions and 56.3% of reduced width data-memory instructions.

Sobel filter application

The energy evaluation on the most optimized Sobel filter program with only `a.add` and `a.mul` leads energy reduction less than 10%. When reduced width is extended to logic and data-memory instructions, the percentage of reduced width instructions is higher than the first version (*i.e.* from 17% to 73.7%), hence the energy reduction could be improved. Figure 4.15 indicates that Sobel filter includes 44.4% of reduced width data-memory instructions (`a.mem`) and 29.3% of reduced width computation instructions including arithmetic and logic instructions (`a.al`) and multiplication instructions (`a.mul`).

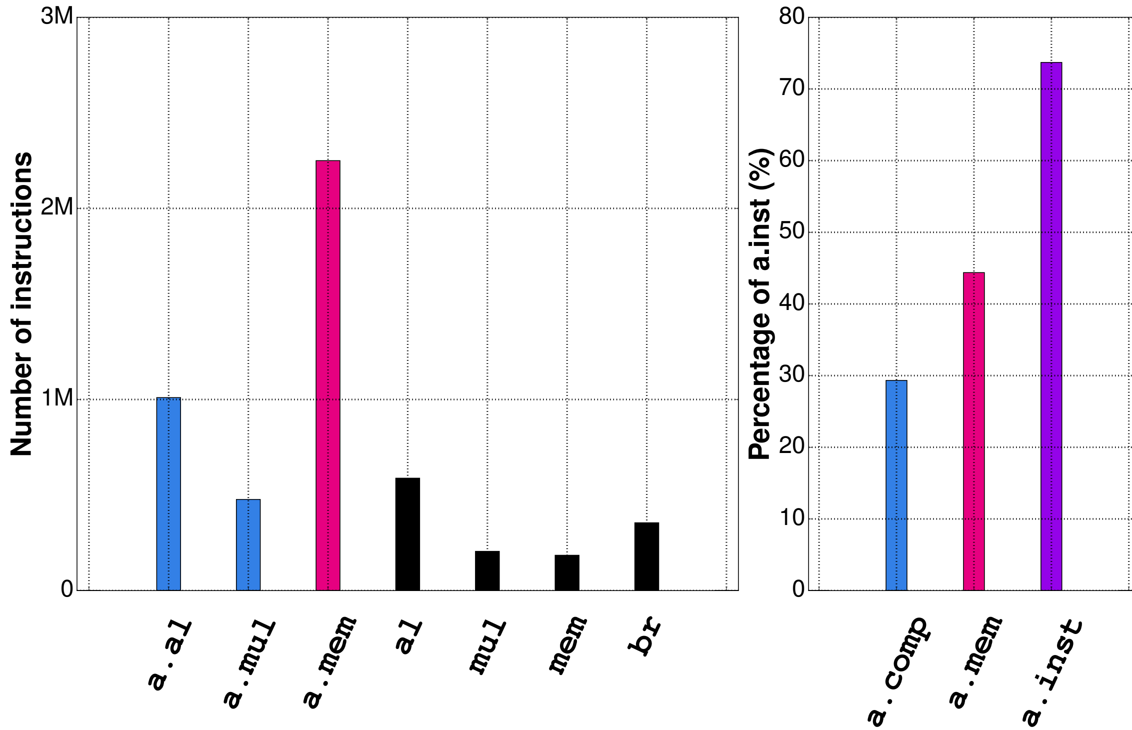


Figure 4.15: Sobel filter instruction breakdown.

Forwardk2j application

As jmeint application, no specific optimization is performed on the forwardk2j program. The proposed program in the state of the art is already optimized. Unlike Sobel filter and jmeint, forwardk2j includes more than 20% of `a.add` and `a.mul` as indicated on Figure 4.10; Hence the energy reduction with only `a.add` and `a.mul` is higher than the first two applications: Sobel filter and jmeint. When the reduced width is extended to logic and memory instructions, the number of reduced width instructions increases (from 46% to 81.8%), hence the energy reduction is improved. Figure 4.16 indicates that forwardk2j includes 70.1% of reduced width computation instructions and 11.7% of reduced width data-memory instructions.

4.4.2 Output quality evaluation

In this section we aim to evaluate the impact of the reduced width units on the output quality of the fixed-point applications. As limit of width in our investigation, we select 26 bits in which the overflow energy costs are low. Above 26 bits, computing

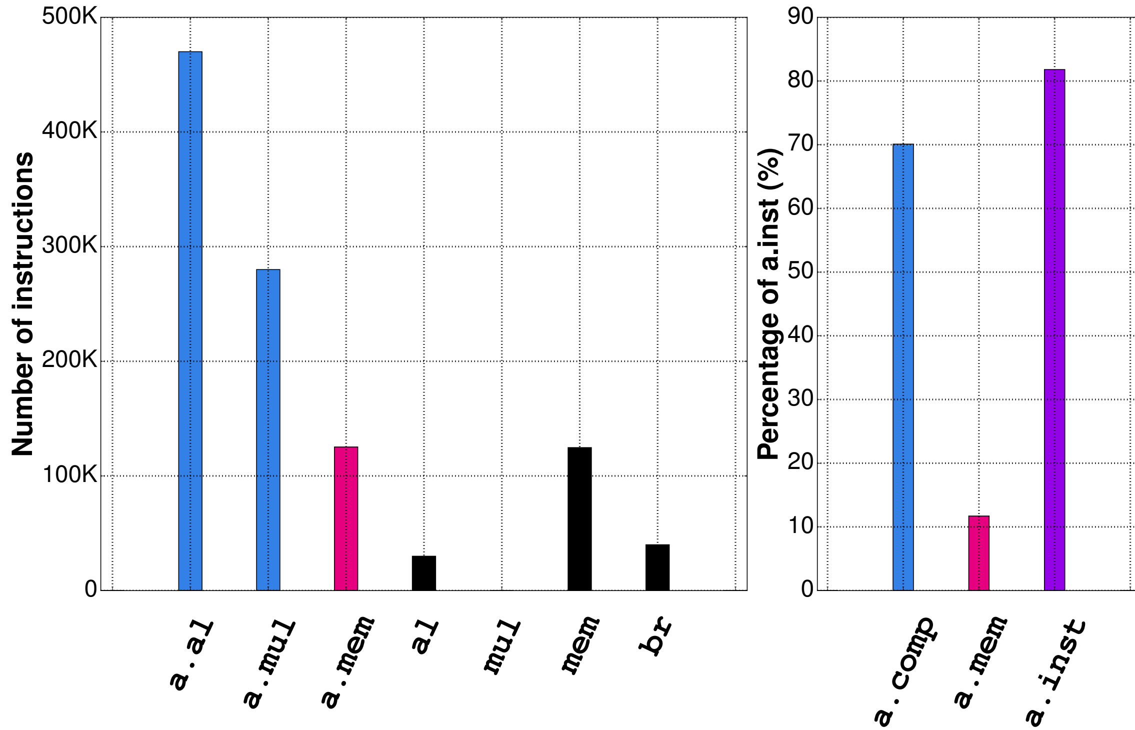


Figure 4.16: Forwardk2j instruction breakdown.

with the accurate units is less costly than computing with reduced width units as indicated on Figure 3.2. The results from the errors evaluation of the fixed-point applications indicate the number of integer and fractional bits required for each of them for an acceptable error. The selected 26 bits are more than sufficient for the evaluated applications to get an acceptable output quality, as indicated on Figures 4.2, 4.3, 4.4, 4.5 and 4.6. We set to 0 a number of LSBs to get the reduced width value, *i.e. reduced width = width - LSBs*.

Jmeint application

Jmeint output is a boolean value that indicates if the two triangles intersect or not. The Figure 4.17 presents the errors introduced by the reduced width units. The quality metric for evaluation is the error rate.

We compare the two boolean vectors returned respectively by the reference floating-point jmeint program and the fixed-point jmeint program executed with reduced width units on the fixed-point program. To estimate the error rate, we com-

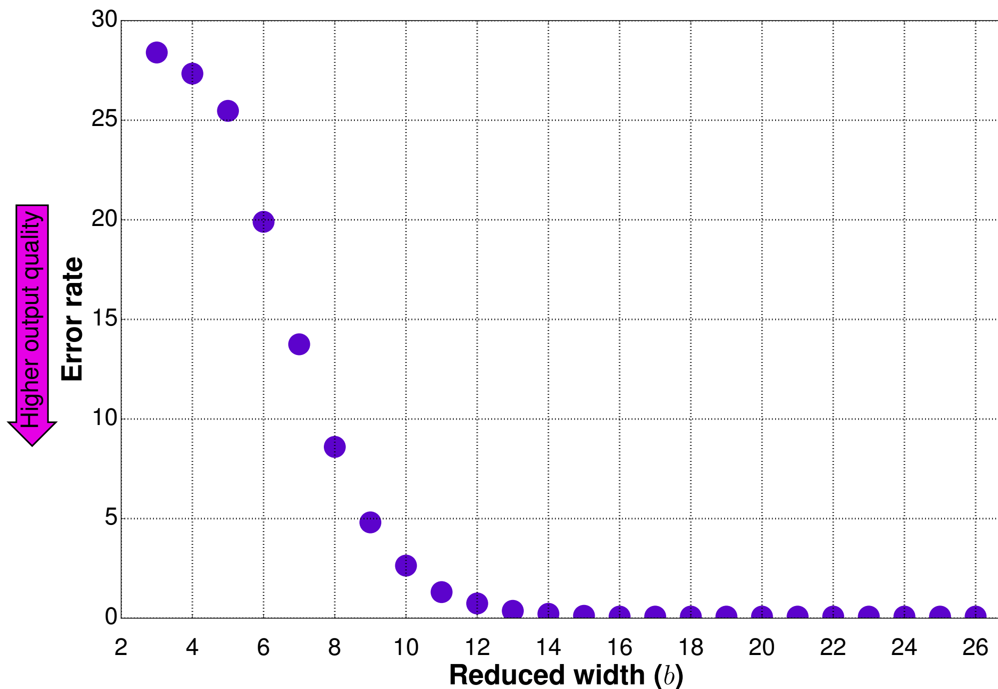


Figure 4.17: Jmeint output quality evaluation.

pute the percentage of output data for which reference and approximate values are similar (*i.e.* have the same boolean value: 1 or 0). Figure 4.17 indicates that from 16 bits, the error rate $\leq 0.1\%$.

Sobel filter application

We evaluate the impact of the reduced width units on the Sobel filter output quality with the SSIM metric. The Sobel filter requires at least 9 bits in the integer part, hence for width = 26, the maximum number of fractional bits is equal to 17 bits.

Sobel filter application, as all image processing applications, can be evaluated using various quality metrics. The most commonly used are the RMSE, the PSNR and the SSIM. The SSIM is the most correlated metric with human perception, it is able to quantify the human visual perceptual quality. We perform the output quality *vs* energy trade-off study with the SSIM metric. Figure 4.18 presents the mean SSIM value and the standard deviation of the 100 random input images. We note that from 16 bits and above, Sobel filter has an acceptable quality of results, *i.e.*

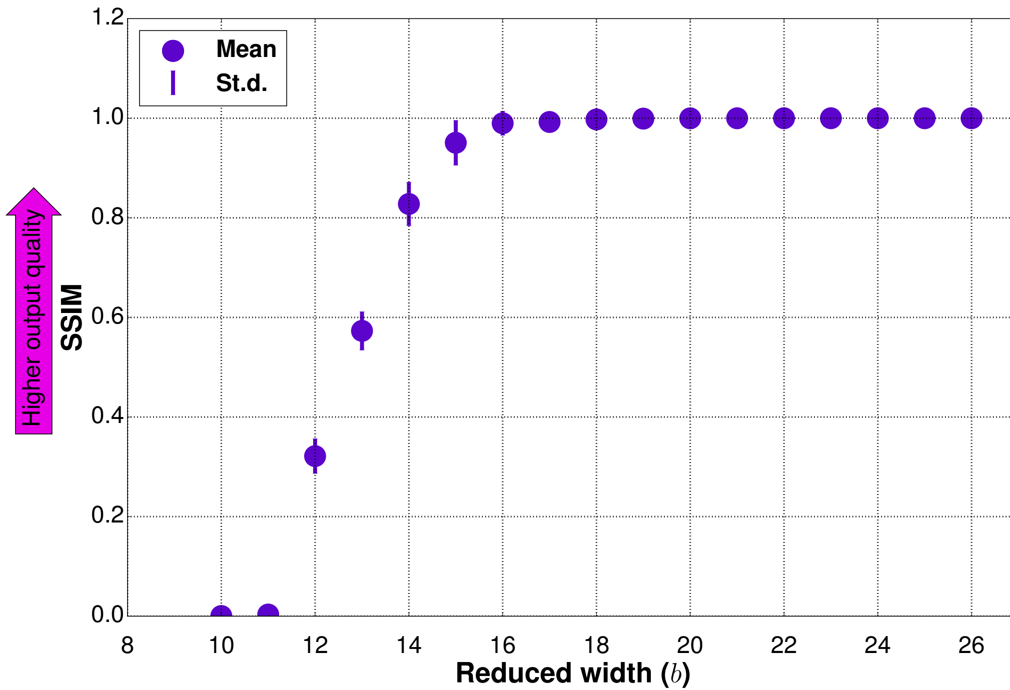


Figure 4.18: Sobel filter output quality evaluation.

SSIM ≥ 0.99.

Forwardk2j application

Figure 4.19 presents the relative errors due to reduced width computations on forwardk2j. For each couple of coordinates, the maximum value between the errors of the two coordinates is returned, and the mean value and the standard deviation of the 10 000 maximum errors are computed. We note that from 16 bits and above, the relative error $\leq 0.1\%$.

4.4.3 Output quality *vs* energy trade-off study

We compare the potential in terms of energy reduction on our three applications when executed with only reduced width computation units and when executed with reduced width units for both computations and data-memory accesses. Figures 4.14, 4.15, and 4.16 indicate that the applications include more approximated

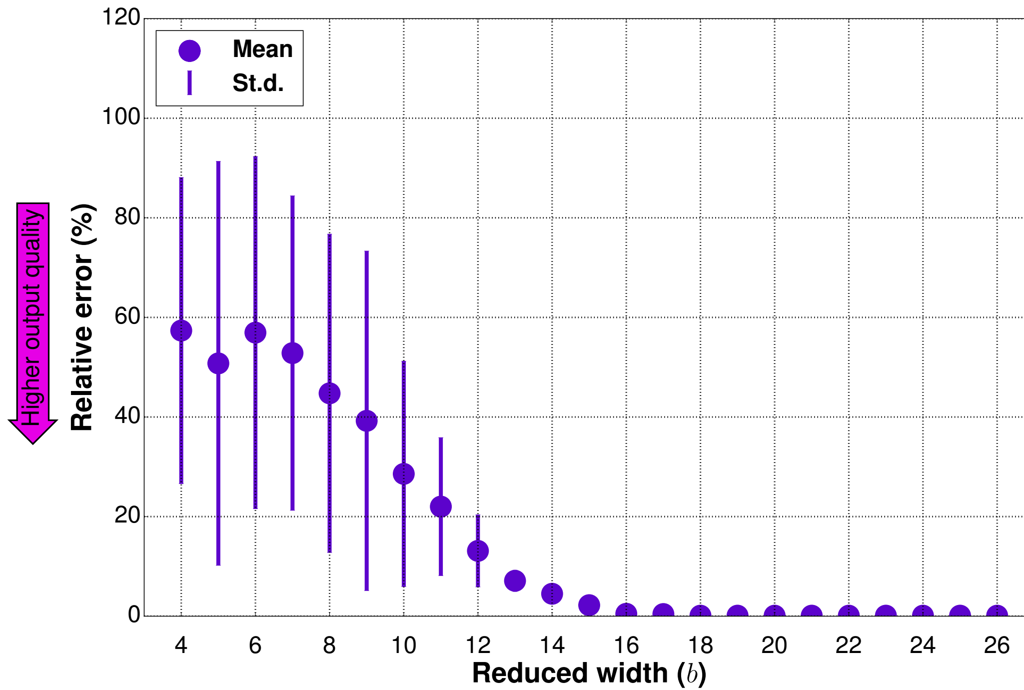


Figure 4.19: Forwardk2j output quality evaluation.

instructions than full width ones (more than 75%). Hence the extension of reduced width units on logic and memory instructions could improve the energy reduction. In the following experiments we investigate the output quality *vs* energy trade-off on our three applications.

Jmeint application

Figure 4.20 indicates that, for a high output quality, *i.e.* error rate $\leq 0.1\%$, the energy reduction when executing the jmeint application with only `a.comp` is up to 2% for b = 16 bits. When accessing the data-memory with reduced width (`a.mem`), the energy can be reduced by up to 14% for b = 16 bits. We note that the reduced width data-memory units improve the energy reduction obtained with only reduced width computation units by up to 12% for an acceptable output quality, *i.e.* error rate $\leq 0.1\%$ (for width equal to 16 bits and above).

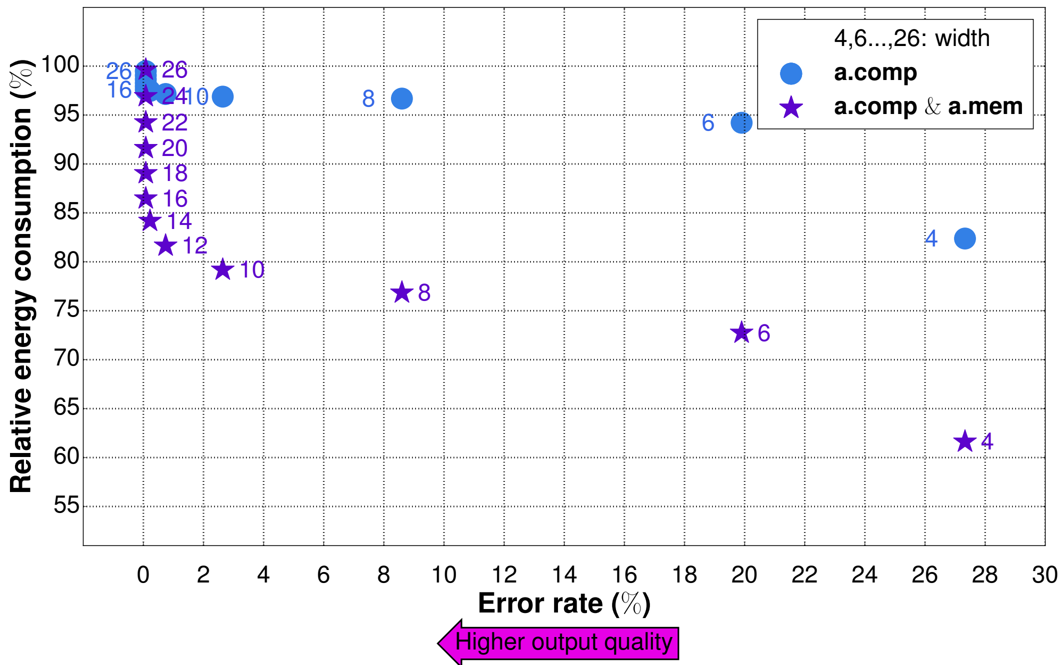
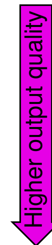


Figure 4.20: Jmeint output quality *vs* energy trade-off

Sobel filter application



The energy reduction of Sobel filter is evaluated with reduced width for both computations and memory accesses. Figure 4.21 indicates that the energy reduction when executing the Sobel filter application with only reduced width computation instructions (`a.comp`) is up to 6% for b = 10 bits. With width reduction extension when accessing the data-memory (`a.mem`), the energy can be reduced by up to 23% for b = 10 bits. For higher output quality, *e.g.* width equal to 16 bits, the energy reduction decreases: equal to 5% with `a.comp`) and equal to 16% with both `a.comp` and `a.mem`. We can deduce that for the Sobel filter application, the energy reduction with only `a.comp` can be improved by up to 11% with both `a.comp` and `a.mem`, for an acceptable output quality, *i.e.* $SSIM \geq 0.99$ (for width equal to 16 bits and above).

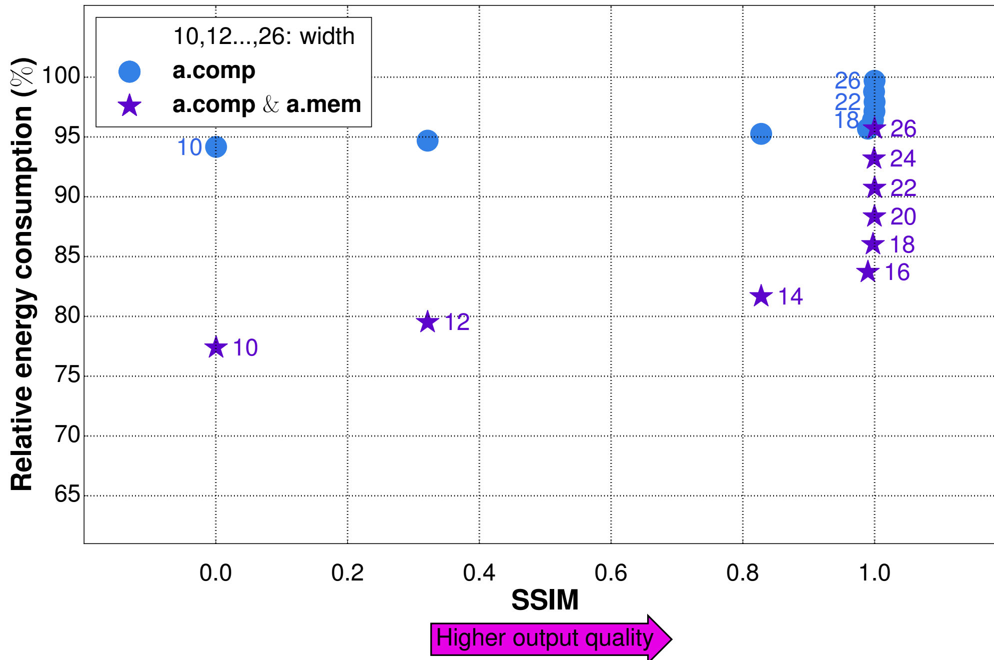


Figure 4.21: Sobel filter output quality *vs* energy trade-off

Forwardk2j application

Figure 4.22 indicates that, for a high output quality, *i.e.* relative rate $\leq 0.1\%$ the energy reduction when executing the forwardk2j application with only reduced width computation instructions (`a.comp`) is up to 12% and up to 15% with both `a.comp` and `a.mem`. The energy reduction with only a.comp is higher in forwardk2j than in Sobel filter and jmeint because the fraction of reduced width computation instructions is higher in forwardk2j application than in these two applications. The integration of reduced width memory access instructions (`a.mem`) improves the energy reduction obtained with `a.comp` by only up to 3% for an acceptable output quality, *i.e.* relative rate $\leq 0.1\%$ (for width equal to 16 bits and above).

In summary, our experiments indicate that the extension of the width reduction to the memory improves the energy reduction on the evaluated applications. When the evaluation is performed with only reduced width for arithmetic units, for an acceptable output quality, *i.e.* width equal to 16 bits, the energy is reduced at most by up to 2% for jmeint, up to 5% for Sobel filter, up to 12% for forwardk2j.

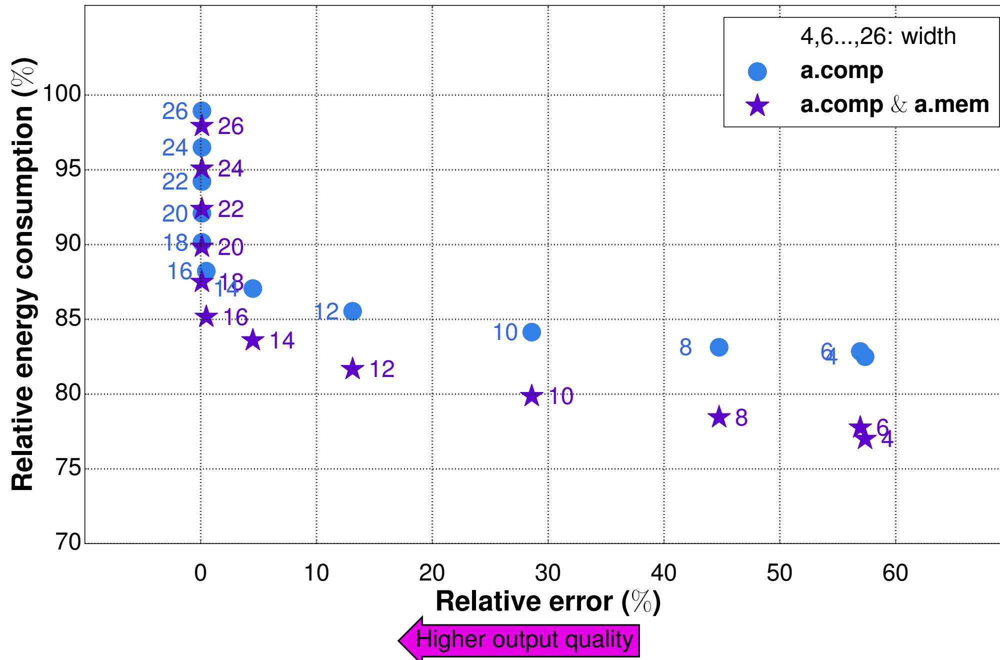


Figure 4.22: Forwardk2j output quality *vs* energy trade-off

When the evaluation is performed with reduced width in both arithmetic and data-memory units, the energy can be reduced at most by up to 14% for jmeint, up to 16% for Sobel filter,and up to 15% for forwardk2j. For the evaluated applications, the average energy reduction with only reduced computation units is improved, by up 11.5% for the memory intensive applications, *i.e.* jmeint and Sobel filter, and up to 3% forwardk2j which is computation intensive. We can conclude that for a high quality requirements, the energy improvements are significant in memory intensive applications, such as image processing ones which are widely used in approximate computing.

4.5 Chapter summary

In this chapter we have evaluated the impact of the reduced width units on applications output quality and energy consumption when executed in a RISC-V processor. We have estimated the global energy reductions on the three applications: jmeint, Sobel filter and forwardk2j with both reduced width computations and mem-

ory units.

At first the evaluation is performed with only approximate units including the approximated adders and multipliers, which are the typical approximate units proposed in the literature. The results indicate that, even after low-power optimizations on the applications source code, the global energy reduction at the level of the entire systems (processor core + memory) are rather low. This is because: (1) the applications include many other instructions (*e.g.* memory accesses, branch instructions, full width arithmetic instructions for computing addresses) and not only the additions and multiplications and (2) although the energy reduction at the operator level is high, when integrating the operators in a processor, the gains of reducing the width diminish because this reduction is not applied on some parts of the processor, *e.g.* instruction fetch/decode.

More precisely, we note that when only width of computations is reduced, for an acceptable output quality *i.e.* error $\leq 0.1\%$, the energy can be decreased up to 2% for jmeint, up to 5% for Sobel filter, up to 12% for forwardk2j. When the reduced width is also applied in memory access the energy can be reduced by up to 14% for jmeint, up to 16% for Sobel filter, up to 15% for forwardk2j. Note that the impact of the reduced width data-memory units is highlighted in memory intensive applications such as image processing ones widely used in approximate computing. We can conclude that if one would like to benefit from scaling the width on the memory intensive applications, the design of reduced width memory units must be included.

CHAPTER 5

GLOBAL ENERGY MODEL WITH SOFTWARE AND ARCHITECTURE PARAMETERS

In the previous chapter we have evaluated the impact of reduced width units on three applications. In this chapter we go one step forward and we attempt to generalize a model for the RISC-V processor extended with reduced width units for computation and data-memory access. The model combines both software and hardware architecture parameters to estimate the global energy consumption of any given application. Both software and hardware designers can make use of the proposed model to have an early insight into the impact of optimizations on the global energy reduction of a given application.

Software designers can use our model to have an idea on the parts of the source code (*e.g.* computations or data-memory accesses) in which optimizations may lead to important energy reduction. Moreover the model can be used to have an idea into the global energy reduction reachable knowing the fraction of approximate instructions and the required width for a target output quality.

Hardware designers can use our model to find, for a given hardware architecture, the types of units that have potential in terms of energy reduction. For example, one can decide to optimize a complex computation unit and/or the data-memory unit.

The energy reduction in approximate computing depends both on the degree of approximations and on the fraction of *approximable* operations on a given application, similarly to Amdahl's law in parallel computing. In parallel computing [13], Amdahl indicates that for a given algorithm to be executed on a multi-core platform, the number of cores is not sufficient to reduce the computation time. The fraction of *parallelizable* operations is also involved in the speedup.

Our model can be applied on various hardware architectures. In this thesis, it is evaluated on applications executed in a RISC-V processor extended with reduced width units for arithmetic and logic instructions, multiplications, and data-memory accesses.

This chapter is organized as follows. Section 5.1 describes the proposed global energy model. Section 5.2 presents a case study that indicates how a software designer can use the global energy model on a given application. Section 5.3 presents a case study that shows how a hardware designer can use the model to evaluate the impact of the circuit-level optimizations on an application energy reduction. Section 5.4 summarizes the chapter.

5.1 Global energy model

In this section, we first define the notations included in the global energy model, then we describe the global energy reduction formula.

5.1.1 Notations

Table 5.1 defines the notations used in our global energy model. Note that in our work, *approximate* refers to *reduced width*.

SW parameters	Description
N_{A_m}	Number of approx. data-memory instructions
N_{A_c}	Number of approx. computation instructions
$N_A = N_{A_m} + N_{A_c}$	Number of approx. instructions
$N_{\bar{A}}$	Number of non-approx. instructions
$N = N_A + N_{\bar{A}}$	Total number of instructions
$f_{A_m} = \frac{N_{A_m}}{N_A}$	Fraction of approx. data-memory instructions from N_A
$f_{A_c} = \frac{N_{A_c}}{N_A}$	Fraction of approx. computation instructions from N_A
$f_A = f_{A_c} + f_{A_m}$	Fraction of approx. instructions
$f_{\bar{A}} = 1 - f_A$	Fraction of non-approx. instructions
Archi. parameters	**Description**
o_m	Energy overhead for data-memory instructions
o_c	Energy overhead for computation instructions
r	Ratio of non-scalable energy
e_{A_m}	Average energy of an approx. data-memory instruction
e_{A_c}	Average energy of an approx. computation instruction
$e_{\bar{A}_m}$	Average energy of a non-approx. data-memory instruction
$e_{\bar{A}_m}$	Average energy of a non-approx. computation instruction
$e_{\bar{A}}$	Average energy of a non-approx. instruction
Global parameters	**Description**
B	Maximum width
b	Active width The width is configurable: $1 \leq b \leq B$
E_A	Energy consumed by all approx. instructions of an application
$E_{\bar{A}}$	Energy consumed by all non-approx. instructions of an application
$E_{T_{\bar{A}}}$	Energy total of a non-approx. application
E_T	Energy total of an approx. application
α	Energy reduction of an application

Table 5.1: Software and hardware architecture parameters

5.1.2 Energy reduction

Let us consider a program as a set of reduced width and full width instructions. The reduced width instructions can be grouped into two categories. The first category is the reduced width computations instructions (*i.e.* `a.comp`) including the reduced width arithmetic and logic instructions (*e.g.* `a.add`, `a.sub`, `a.lgc`) and the reduced width multiplication (*i.e.* `a.mul`). The second category is the reduced width data-memory accesses (*i.e.* `a.mem`) including the reduced width load/store instructions (*i.e.* `a.ld`, `a.st`).

Let $E_T(b)$ be the total energy consumed by an application executed with b bits (with $1 \leq b \leq B$). We assume that the total energy consumed by an application is the sum of the energy consumed by the reduced width instructions (i.e. E_A) and the energy consumed by the full width instructions (i.e. $E_{\bar{A}}$), as indicated in Equation 5.1.

$$E_T(b) = E_A(b) + E_{\bar{A}} \tag{5.1}$$

The energy consumed by all the reduced width instructions executed with b bits, *i.e.* $E_A(b)$ is equal to the sum of the energy consumed by the reduced width computations instructions and the energy consumed by the reduced width data-memory accesses, as indicated in Equation 5.2.

$$E_A(b) = N_{A_m} \times e_{A_m}(b) + N_{A_c} \times e_{A_c}(b) \tag{5.2}$$

In Equation 5.2, we use the mean values of energy consumption for each category of instructions: e_{A_c} is the mean value of the energy consumption of a typical reduced width computation instruction (`a.comp`) and e_{A_m} is the mean value of the energy consumption of a typical reduced width data-memory access (`a.mem`). Figure 3.3 indicates low variations of the energy consumed by instructions of the same category. For the reduced width computation instructions, the energy consumed by each instruction class is almost the same for widths in the range 1 to 12 bits; the relative standard deviations of the energy values are less than 2% for widths in the range 13 to 18 bits and between 2% and 5% in the range 18 to 32 bits. For the reduced width data-memory instructions, the relative standard deviation of the energy values is less than 1%. We assume that for a given instruction category (`a.comp` or `a.mem`) the small variations between the energy consumption of the in-

struction classes allow us to use the mean values of the energy to estimate the global energy consumption of an application.

For the full width instructions, we also use the mean energy value to deduce the total energy consumed by all the full width instructions of a given program, as presented in Equation 5.3.

$$E_{\bar{A}} = N_{\bar{A}} \times e_{\bar{A}} = (N - N_A) \times e_{\bar{A}} \tag{5.3}$$

When an application is executed with reduced width (i.e., $b < B$), the energy consumption with full width execution (*i.e.* $E_{T_{\bar{A}}}$) is reduced. The energy reduction α is estimated with Equation 5.4:

$$\alpha(b) = \left(1 - \frac{E_T(b)}{E_{T_{\bar{A}}}}\right) \tag{5.4}$$

$$\frac{E_T(b)}{E_{T_{\bar{A}}}} = \frac{N_{A_m} \times e_{A_m}(b) + N_{A_c} \times e_{A_c}(b) + (N - N_A) \times e_{\bar{A}}}{N_{A_m} \times e_{\bar{A}_m} + N_{A_c} \times e_{\bar{A}_c} + (N - N_A) \times e_{\bar{A}}} \tag{5.5}$$

Equation 5.5 expressed with fractions of reduced width instructions gives:

$$\frac{E_T(b)}{E_{T_{\bar{A}}}} = \frac{f_{A_m} \times e_{A_m}(b) + f_{A_c} \times e_{A_c}(b) + \left(\frac{1}{f_A} - 1\right) \times e_{\bar{A}}}{f_{A_m} \times e_{\bar{A}_m} + f_{A_c} \times e_{\bar{A}_c} + \left(\frac{1}{f_A} - 1\right) \times e_{\bar{A}}} \tag{5.6}$$

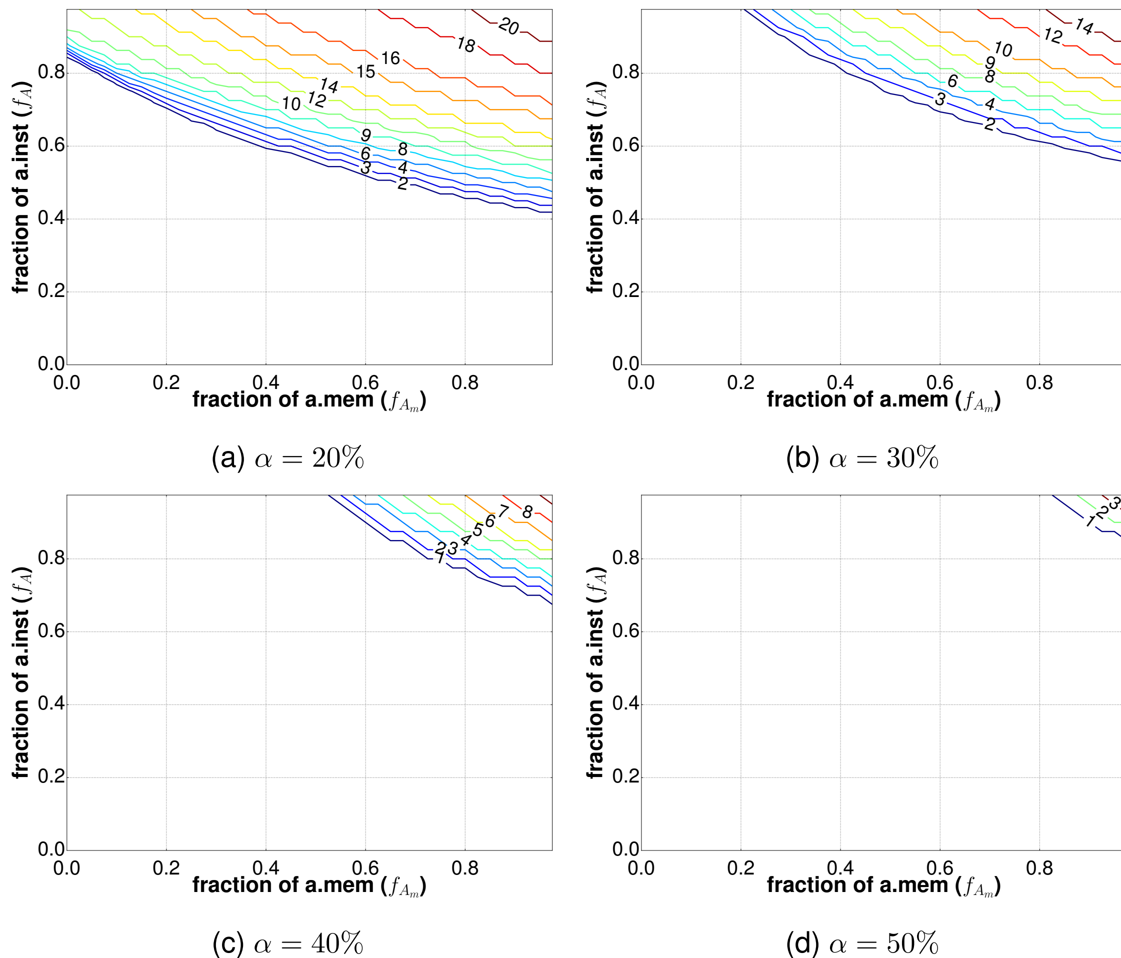
5.2 Case study 1: impact of software parameters

The figures presented in this section provide to a software designer an insight into the energy reduction, knowing the fraction of reduced width instructions. In an application, for a given energy reduction, the figures determine the adequate widths for computations and data-memory storage. Inversely, knowing the required width for a given output quality, one can get an estimation of the energy reduction based on the fraction of approximate instructions.

5.2.1 Width estimation for a given energy reduction

Figure 5.1 indicates that for a given application, the software designer can have an early insight into the width required to execute the application for various energy reduction values. The energy reduction depends both on the width and the fraction

of approximate instructions, as presented on Figures 5.1a, 5.1b, 5.1c and 5.1d. For a given energy reduction value (α), these figures indicate the width range for which an application can be executed depending on the fraction of reduced width data-memory instructions. For example, Figure 5.1a indicates that if one requires an energy reduction equal to 20%, an application can be executed with width in the range 1 to 20 bits in our extended RISC-V. For example for a given application including 80% of approximate instructions from the number total of instructions, with 60% of approximate data-memory instructions from the number of approximate instructions (*i.e.* 60% of `a.inst.`), if the software designer, targets 20% of energy reduction, this application can be executed with 15 bits.



(a) $\alpha = 20\%$

(b) $\alpha = 30\%$

(c) $\alpha = 40\%$

(d) $\alpha = 50\%$

Figure 5.1: widths (b) for a given value of energy reduction

When we apply our model on Sobel filter and forwardk2j applications, we ob-

tain approximatively the same width for a given energy reduction, as one in Figures 4.21 and 4.22 obtained by simulations. In fact, in simulations, for 20% of energy reduction, the Sobel filter (with f_A = 0.737, f_{A_m} = 0.443) should be executed with 12 bits and forwardk2j (with f_A = 0.818, f_{A_m} = 0.117) should be executed with 10 bits. Our model indicates that Sobel filter and forwardk2j can be executed with 12 bits and 9 bits, respectively. We can conclude that our model gives an insight into the required width for a given energy reduction.

We note that the width range decreases if the required energy reduction increases as visible on Figures 5.1a, 5.1b, 5.1c and 5.1d. On Figure 5.1b if the required energy reduction is equal to 30%, the width range is 1 to 14 bits with a f_{A_m} of at least 0.2. If the required energy reduction is equal 40%, the width range is 1 to 8 bits with a f_{A_m} of at least 0.5, as indicated on Figure 5.1c. Figure 5.1d indicates that, if the target energy reduction is equal to 50%, the width range is 1 to 3 bits with a f_A of at least 0.85. From the experiments we can conclude that an energy reduction (α) equal or more than 50% can be never reached in our extended RISC-V, for practical values of the width, because computing with less than 3 bits can rarely provide high output quality.

5.2.2 Energy reduction estimation for a given width

Figure 5.2 presents the energy reduction, for various widths, based on various fractions of reduced width instructions f_{A_c} and f_{A_m} in $[0, 1]$. Figure 5.2a indicates that the energy can be reduced by up to 45% with a width equal to 4 bits. Figure 5.2b indicates that the energy can be reduced by up to 38% with a width equal to 8 bits. Figure 5.2c indicates that the energy can be reduced by up to 32% with a width equal to 12 bits. Figure 5.2d indicates that the energy can be reduced by up to 26% with a width equal to 16 bits.

We note that the energy reduction decreases if the fraction of reduced width instruction decreases. For example Figure 5.2d indicates that with $f_A = 0.8$ and $f_{A_m} = 0.4$, the energy can be reduced by up to 16% for $b = 16$ bits. For the same number of bits (*i.e.* $b = 16$ bits), the energy reduction decreases if the fraction of approximate instructions is reduced, for example with $f_A = 0.4$ and $f_{A_m} = 0.2$, the energy can only be reduced by up to 4%, as indicated in Figure 5.2d.

When we apply our model on Sobel filter and forwardk2j applications, we ob-

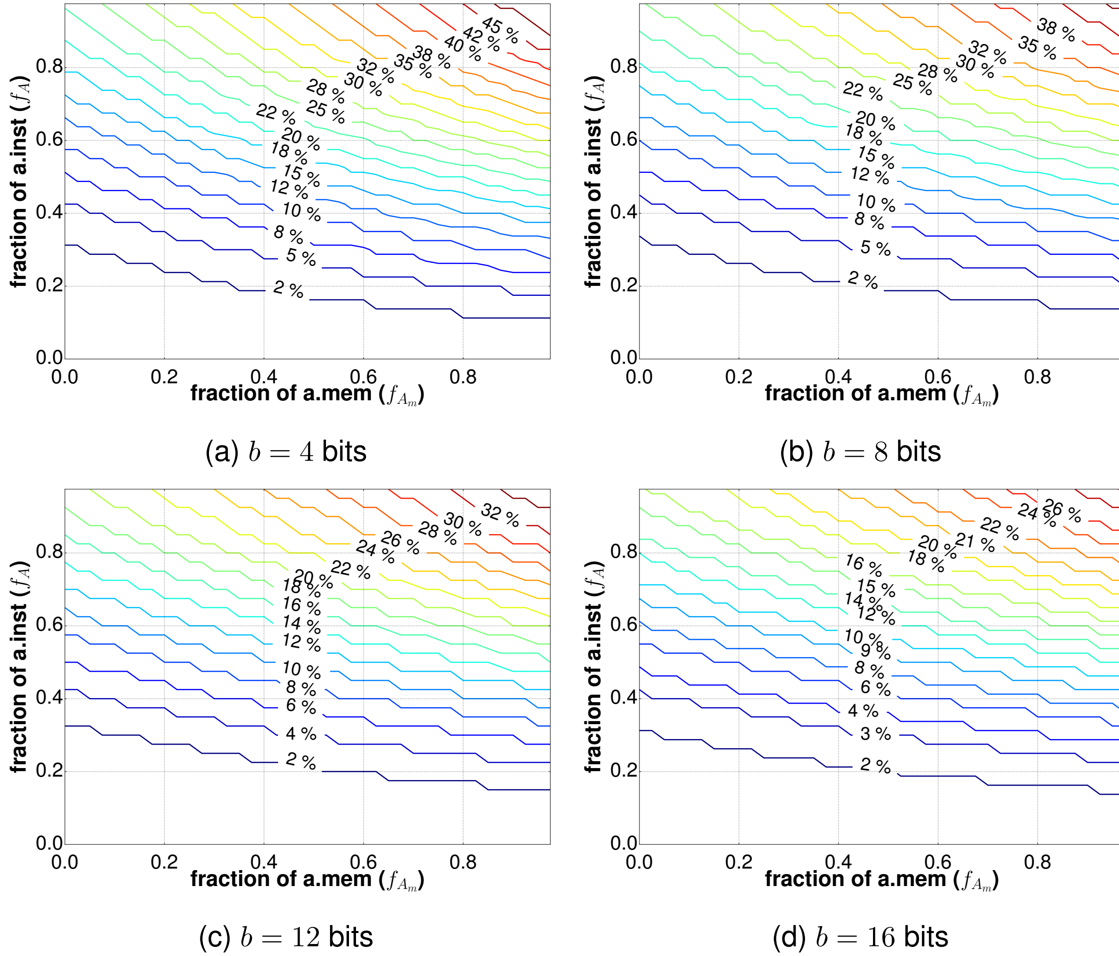


Figure 5.2: Energy reduction (α) for a given width

tain approximatively the same energy reduction for a given width, as one in Figures 4.21 and 4.22 obtained by simulations. In fact, with simulations, for $b = 16$ bits the energy reduction of Sobel filter is equal to 16% (with f_A = 0.737, f_{A_m} = 0.443, as indicated in Figure 4.15) and the energy reduction of forwardk2j is equal to 15% (with f_A = 0.818, f_{A_m} = 0.117, as indicated in Figure 4.15. Our model indicates that when Sobel filter and forwardk2j are executed with 16 bits, the energy reductions are equal to 16% and 14%, respectively. We can conclude that our model can be used to get an insight into the energy reduction obtained with a given width. For energy efficiency, the aim is to be on the right upper part of Figure 5.2.

5.3 Case study 2: impact of hardware units

The reduced width units for computations and data-memory accesses embedded in our RISC-V processor, could be optimized by hardware designers to improve the energy reduction. Our proposed global energy model allows to estimate the energy reduction for a given application when hardware units are optimized. For example, when the energy consumed by a unit, *e.g.* a multiplier, is reduced in half by some optimizations, the proposed energy model allows to estimate the global energy reduction on a complete application execution. We estimate at various widths the global energy reduction when one unit is optimized at a time. The global energy model in Equation 5.6 is instantiated with the energy values per instruction presented in Chapter 3 and the fraction of reduced width instructions (for both computations and data-memory accesses) of the Sobel filter and forwardk2j applications, deduced from the instructions breakdown study in Chapter 4.

Sobel filter application

The aim is to evaluate the potential in terms of energy reduction of the optimized hardware units for several widths on the Sobel filter application with the fraction of reduced width instructions deduced Figure 4.15. Figure 5.3 presents the global energy reduction of the Sobel filter application for a given percentage of energy reduction obtained in the optimization of one reduced width unit for several widths. For example, Figures 5.3a, 5.3b, 5.3c indicate that for $b = 8$ bits, when the energy of an approximate instruction is reduced by up to 60%, the global energy reduction at application level is equal to 30% for computation units (both `a.al` and `a.mul`) and equal to 40% for data-memory units (`a.mem`). For $b = 16$ bits the energy reduction decreases: 23% with `a.al`, 24% with `a.mul` and 35% with `a.mem`.

Forwardk2j application

The same evaluation is performed on the forwardk2j application with the fraction of the reduced width instructions deduced from Figure 4.16. Figure 5.4 presents the global energy reduction of the forwardk2j application for a given percentage of the energy reduction obtained when optimizing one reduced width unit at various widths. For example, Figures 5.4a, 5.4b, 5.4c indicate that for $b = 8$ bits, when the

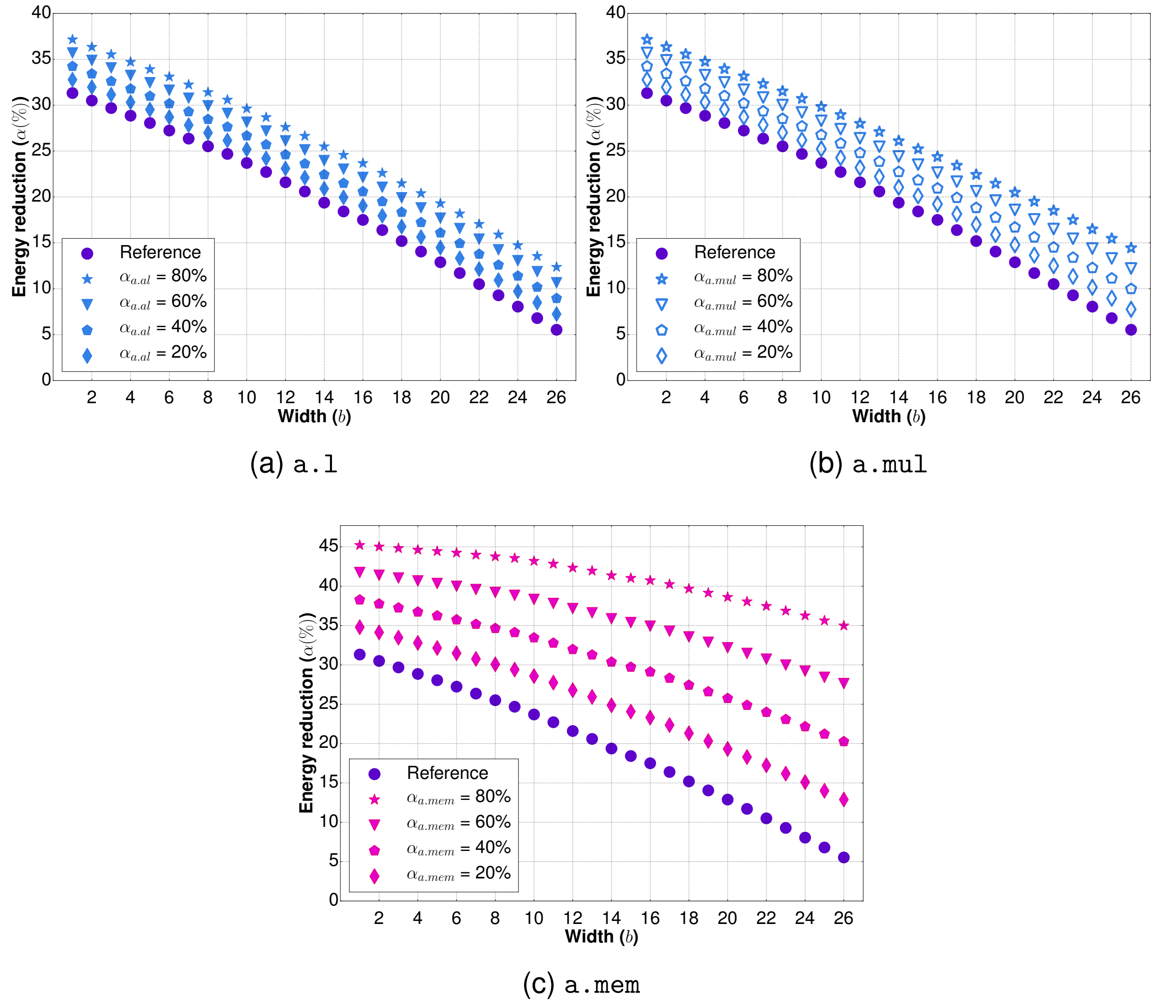


Figure 5.3: Sobel filter energy evaluation with optimized units

energy of an approximate instruction is reduced by up to 60%, the global energy reduction at application level is equal to 32% for computation units (both `a.al` and `a.mul`) and equal to 25% for data-memory units (`a.mem`). For $b = 16$ bits the energy reduction decreases: 26% with `a.al`, 28% with `a.mul` and 20% with `a.mem`.

Summary

These results indicate the units in which the optimizations may lead to high energy reduction. For our extended RISC-V processor, we can see on Figures 5.3 and 5.4

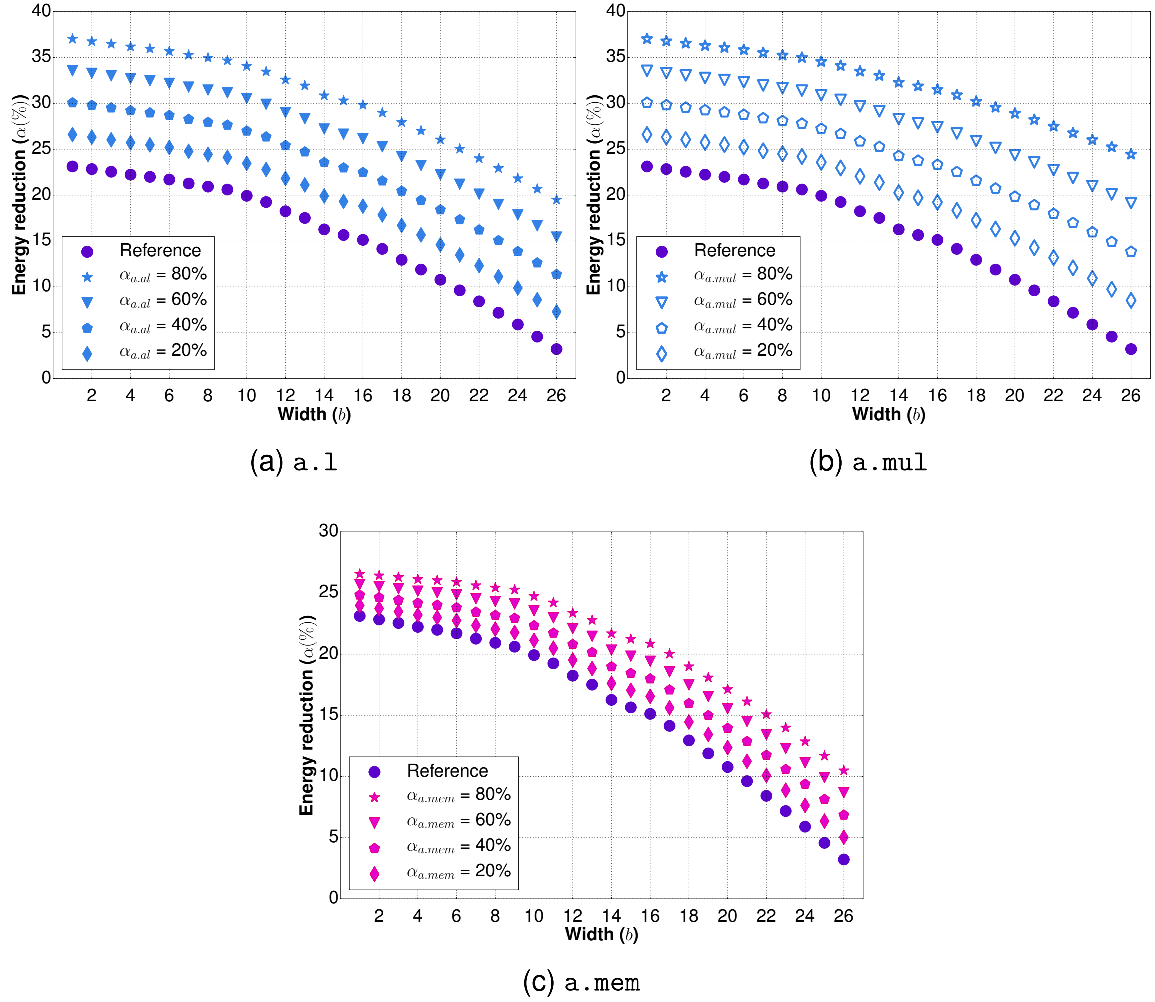


Figure 5.4: Forwardk2j energy evaluation with optimized units

that the data-memory units have more potential in terms of energy reduction than the computation units. As example, let us consider the optimization of the units (for both computations and data-memory accesses) such that the energy reduction of the stand-alone optimized units equal to $\alpha_{a.\texttt{unit}} = 60\%$, $\texttt{unit}$ in $\{\texttt{al}, \texttt{mul}, \texttt{mem}\}$. When the forwardk2j application, including only 11% of reduced width data-memory instructions, is executed with the optimized data-memory units (with $\alpha_{a.mem} = 60\%$), its global energy reduction is equal to 25% at 8 bits, as presented on Figure 5.4c. To obtain these 25% of global energy reduction with the reduced width computation units, forwardk2j needs to be executed with more than 50% of `a.al` or `a.mul`, *i.e.* $5\times$

of `a.mem`. For example in Figures 5.4a and 5.4b with only 32% of energy reduction forwark2j have to be executed with 70% of `a.al` or `a.mul`, *i.e.* $f_c = 0.7$).

5.4 Chapter summary

This chapter has presented a global energy model with software and hardware architecture parameters. The proposed model allows designers to have an early insight into the energy reduction reachable when optimizations are performed on software and/or hardware. Knowing the percentages of reduced width operations for both computations and data-memory accesses, one can have an early estimation of the width required to execute an application for an energy reduction target. Or, inversely, one can estimate the energy reduction of a given application for a required number of bits to reach a given output quality. furthermore one can estimate the global energy reduction of a given application when a unit is optimized or in which units of the architecture further optimizations can benefit given applications.

We evaluate our model on two different applications: the Sobel filter application that is memory intensive and the forwardk2j application that is computation intensive. These two applications are executed in an extended RISC-V including reduced width units for both computations and data-memory accesses.

From this model we reach the same conclusion as in Chapter 4, namely that the reduced width units for data-memory accesses have more potential in terms of energy reduction than the reduced width units for computations. The energy reduction of a reduced width data-memory instruction is $1.2\times$ to $1.8\times$ higher than one of a reduced width computation instruction, as visible on Figure 3.3.

CHAPTER 6

CONCLUSION AND FUTURE WORK

IoT devices have to comply with strong resource constraints, such as power consumption (due to battery limits), silicon area (to reduce fabrication costs) and timing (to ensure service quality). Approximate computing is a field that studies the trade-offs between power, area, and/or performance and the quality of an application result. One method is to reduce the width of the operands in operators, in applications in which maximal precision is not necessary to ensure a given quality in order to decrease energy consumption. The main goal of this thesis is to determine whether IoT applications would benefit from a small general purpose processor core equipped with reduced width units.

To this end, we extend a RISC-V processor core [10] with functional units where the width is reduced and configurable at runtime. This work was published in [81]. Our evaluation is performed on a selection of applications from the Axbench benchmark suite [93]: jmeint, Sobel filter and forwardk2j. For these benchmarks we first had to deal with the impact of the conversion from the floating-point representation into the fixed-point one, as most of the benchmarks proposed in the state of the art are implemented in floating-point. We evaluate the quality degradation due to this conversion for the selected benchmarks using several errors metrics. Our experiments suggest that these applications are suitable for fixed-point computations, the error compared to the initial floating-point solution is acceptable (less than 0.1%) for configured widths from 16 bits and above.

Second, we investigated both computations and data-memory accesses with various reduced widths. For the investigated applications, the energy consumption decreases when reducing the width from 32 (*i.e.* full width) to 16 bits. The maximum average of the energy reduction is equal to 6% when considering only the computation units, and equal to 14% when reducing also the width for data-memory accesses, for an error $\leq 0.1\%$. This work was published in [58, 59]. Using reduced width units only for computations lead to a small power gain, even though the lit-

erature reports energy reductions of 1.5x to 2x at arithmetic operator level. We conclude that it is worth extending the reduced width principle to the data-memory to decrease the global energy consumption. As we target a general purpose embedded processor, it is worth having a width that can be configured at runtime because different applications demand different widths to deliver a given quality of results.

Third we have proposed a global energy model for both hardware and software designers to have an early insight into the application-level energy consumption. Our model includes both software parameters and hardware architecture ones. The software parameters are the percentages of reduced width operations for both computations and data-memory accesses and the width required for some target output quality. The hardware parameters are the widths of the units and the energy reduction obtained for each type of stand-alone approximate unit. This model bares a similarity with Amdahl's law. We observe that in approximate computing, for complete applications (*i.e.* not only small kernels), our model as well as our experimental results show that the energy consumption depends on both the degree of approximation (*i.e.* the configured width in the units) and on the proportion of approximate instructions in a complete program. This work was part of our publication in [59]. This model can be used for other approximate computing techniques, beyond reduced width.

This thesis opens up several directions of research.

First, the practical realization of reduced or variable width memory unit is still to be done. The details of memory buses, address decoders, and physical memory organization should be explored to find what is an efficient implementation of variable width memory unit.

Another aspect is programming and compiler support required for automatic code generation for a processor with variable width. Domain-specific languages are increasingly studied in the literature, however tuning the width is not common practice.

Moreover, applications include elementary mathematical functions that need to be implemented. The functions can be approximated by a polynomial approximation [23], for instance. One limitation of the existing methods for polynomial approximation is that the energy consumption is not taken into account when deciding the degree or the intervals for the polynomials.

Finally, when we have a processor in which the width can be reduced/set at runtime, control techniques can be employed to exploit this feature and further minimize the energy consumption. Variations due to, *e.g.* changing energy budget, input data dependencies, can represent new optimization potential. Note that the dynamic width control is often constrained by the additional costs associated with the techniques used, and hence achieving true benefits is not straightforward.

Résumé

Titre: Calcul approximatif pour l'efficacité énergétique des applications de l'Internet des objets

Introduction

L'internet des objets (IoT, pour *Internet of Things* en anglais) peut se voir comme un ensemble de petits ordinateurs, de capteurs, d'actionneurs et d'autres dispositifs embarqués qui interagissent entre eux par des échanges de données sur Internet ou sur des réseaux locaux connectés à internet (voir p.~ex. [43]). Les dispositifs de l'IoT sont implantés de plus en plus dans de nombreuses applications de divers domaines comme la domotique, les systèmes de notification d'urgence, les systèmes de transport, les applications de jeux, les systèmes bio-médicaux, *etc*. Les calculs effectués dans ces applications sont soumis à de fortes contraintes en consommation d'énergie, surface de silicium et temps de calcul.

Le *calcul approximatif* est l'une des nombreuses solutions proposées pour réduire la consommation d'énergie et la surface de silicium dans l'implémentation des services de l'IoT. Le calcul approximatif est une approche qui consiste à réduire les coûts de calcul, comme p.~ex. la consommation d'énergie, la surface de silicium ou le temps de calcul en réduisant la précision ou la qualité des calculs [92, 12, 53], [62], [31], [71], [83], [75]. Le calcul approximatif est appliqué dans plusieurs domaines où les applications sont naturellement tolérantes aux erreurs, c.-à-d. dans lesquelles l'utilisateur n'a pas besoin d'une grande précision pour obtenir une qualité de sortie acceptable. Des exemples d'applications tolérantes aux erreurs dans le calcul approximatif sont: les applications de reconnaissance, de recherche, multimédia, d'analyse de données, *etc*.

Dans le calcul approximatif, plusieurs stratégies ont été proposées à différents niveaux: de la couche applicative au niveau circuit.

Par exemple au niveau applicatif, des extensions de langages de program-

mation ont été proposées [74], [73], [68], des compilateurs approximatifs ont été implémentés pour interpréter les annotations ajoutées sur les parties identifiées comme *approximables* par l'utilisateur (ici le programmeur) dans un code source [81], [24] ou pour introduire des approximations dans le code source approximé [67, 14]. D'autres solutions ont été proposées afin d'ajuster des paramètres de calcul (p.~ex. relatifs à la consommation d'énergie, et aux erreurs de calcul) lors de l'exécution [39], [15], [36].

Au niveau circuit, des unités approximatives ont été conçues pour des calculs et des accès mémoire à basse consommation d'énergie [96], [47], [40], [61].

La plupart des solutions proposées en calcul approximatif au niveau matériel ne sont pas évaluées dans un cadre applicatif complet qui permettrait d'étudier l'impact global des techniques d'approximation proposées sur la consommation d'énergie et la qualité des résultats. L'évaluation des solutions matérielles est souvent effectuée de manière locale et isolée (p.~ex. sur les unités approximatives seules), ce qui ne permet pas d'évaluer leur impact global si elles étaient appliquées dans des applications complètes. Différentes évaluations de solutions matérielles au niveau applicatif sont proposées dans l'état de l'art. Cependant, elles se limitent aux architectures à virgule flottante qui sont trop complexes pour la plupart des systèmes embarqués de l'IoT (les microcontrôleurs et petits processeurs employés dans ce domaine ont très rarement des unités flottantes, mais seulement des unités entières ou virgule fixe).

Dans cette thèse, nous évaluons d'abord l'impact de la *conversion* depuis la représentation en virgule flottante vers celle en virgule fixe sur différentes applications de test (c.-à-d. une sélection de *benchmarks*). En effet, la plupart des applications proposées dans l'état de l'art utilisent des données en virgule flottante mais notre étude cible des architectures entières. La conversion en virgule fixe induit des erreurs dues aux arrondis et la modification de la dynamique de certaines données. À l'aide de plusieurs *métriques d'erreur*, nous évaluons certaines erreurs dues à la conversion de virgule flottante vers la virgule fixe. Ensuite, nous étendons le processeur RISC-V avec des unités entières approximatives afin d'effectuer une évaluation globale de la réduction d'énergie et de la qualité de sortie sur les applications de test sélectionnées.

La plupart des modèles énergétiques proposés dans la littérature pour l'évaluation de la consommation d'énergie des applications n'incluent pas à la fois des paramètres

logiciels et des paramètres d'architecture matériels; seuls des paramètres architecturaux sont proposés dans ces modèles. Ces modèles ne permettent pas aux concepteurs de logiciels ou de matériels d'avoir une estimation, sans simulations, de la réduction globale d'énergie induite par des approximations réalisées sur une application donnée. Nous proposons un modèle énergétique global incluant à la fois des paramètres logiciels et architecturaux pour estimer l'impact, sur la réduction d'énergie des applications, des optimisations réalisées par des concepteurs logiciels sur le code source de l'application et/ou par des concepteurs matériels sur le circuit.

Processeur RISC-V avec des unités à taille réduite

Plusieurs opérateurs entiers (ou en virgule fixe) approximatifs ont été proposés dans la littérature pour réduire la consommation des ressources. Ces opérateurs ne sont pas évalués une fois intégrés dans un processeur complet, avec sa mémoire, exécutant des applications. Pourtant, l'évaluation de ces opérateurs sur des applications nécessiterait une exécution sur un processeur complet pour avoir une vraie idée des gains réalisables globalement. Ainsi, nous étendons l'architecture du processeur RISC-V avec une unité de calcul et une unité de chargement/stockage configurable en nombre de bits, dénommées respectivement `a.EXE` et `a.LSU` (Cf. Figure 3.1). Une instruction spécifique a été ajoutée au jeu d'instructions pour configurer la taille active des unités `a.EXE` et `a.LSU` par l'utilisateur. Pour ces unités à taille réduite, les opérations (de calcul ou de chargement/écriture en mémoire de données) sont effectuées uniquement sur les b bits les plus significatifs (MSBs); $b \in \{1, ..., B\}$ où B est la largeur maximale des données (la taille du chemin de données totale).

Un modèle d'énergie est proposé pour chaque catégorie d'instructions classiques et approximatives. Les valeurs d'énergie des instructions classiques du jeu d'instructions RISC-V, normalisées par rapport au coût d'une multiplication, sont présentées dans le tableau 3.3. Dans notre circuit, la mémoire est implantée dans une technologie à basse tension, ainsi les instructions `ld` et `st` consomment moins d'énergie qu'une instruction de multiplication pour notre processeur.

Pour la multiplication et les instructions arithmétiques et logiques à taille réduite, les valeurs d'énergie sont estimées en considérant que la consommation

d'énergie de la partie `a.EXE` varie avec le nombre de bits comme spécifié dans la méthodologie [61]. D'après notre étude sur la mémoire de données, nous estimons que 40% de l'énergie est indépendante de la taille et que 60% varie linéairement avec la taille. Notons que ce pourcentage dépend de la conception de la mémoire et notre méthode reste valide pour une autre implémentation de la mémoire. L'implantation des opérateurs à taille réduite s'accompagne d'un surcoût. Ce surcoût pris en compte dans notre modèle, est causé par les éléments supplémentaires nécessaires pour partitionner les opérateurs en plusieurs domaines de tension de seuil. Ainsi une opération avec la taille maximale activée ($b = B$ bits) consomme un peu plus que la version à taille fixe de l'unité originale.

La figure 3.2 présente les valeurs d'énergie des classes d'instructions des unités à taille réduite et des unités à taille fixe (originales). Nous notons que les instructions de calcul à taille réduite, c.-à-d. `a.add`, `a.sub`, `a.lgc` ont à peu près la même consommation d'énergie lorsque l'exécution est effectuée dans la plage de 1 à 8 bits. À partir de 9 bits, la consommation d'énergie de ces instructions est différente de l'une à l'autre. Le surcoût d'énergie est très important à partir de 26 bits, c.-à-d. que la consommation d'énergie des unités à taille réduite est supérieure à la consommation d'énergie de l'instruction classique. On peut déduire que pour notre architecture du RISC-V, le calcul avec plus de 26 bits est plus coûteux en terme de consommation d'énergie que le calcul sur des unités à taille fixe 32 bits originales.

Un compilateur et des outils d'aide à la simulation sont proposés pour réduire les efforts des programmeurs. Le compilateur est implémenté pour gérer les *pragmas* ajoutées dans le code source par le programmeur et le simulateur `spike` [11] est étendu pour compter le nombre d'instructions de chaque catégorie et d'évaluer la consommation d'énergie globale d'une application grâce à notre modèle énergétique.

Évaluation des unités à taille réduite sur des applications

Dans notre étude, nous évaluons d'abord l'impact de la conversion de la virgule flottante vers la virgule fixe sur la qualité des résultats des applications avant

d'estimer leur consommation d'énergie. Ensuite, nous évaluons le potentiel en terme de réduction d'énergie des opérateurs les plus étudiés dans l'état de l'art, notamment les additionneurs et les multiplieurs où la taille réduite est configurable lors de l'exécution. Enfin, nous étudions des compromis entre la précision de calcul et la réduction d'énergie sur des applications, exécutées avec des unités à taille réduite (de calcul et d'accés mémoire). Pour l'évaluation de la consommation d'énergie, nous comptons le nombre d'instructions exécutées et nous utilisons notre modèle d'énergie présenté au chapitre 3, enfin nous estimons la consommation d'énergie globale de trois applications de Axbench [93]: jmeint, filtre de Sobel et forwardk2j.

Conversion de virgule flottante en virgule fixe

Les applications de référence sont converties en virgule fixe, avec les fonctions de la bibliothèque `libfi` [4]. Les modes d'arrondi de `libfi` sont les mêmes que ceux présentés dans la sous-section 2.1.1.

Jmeint est un algorithme qui détermine si deux triangles 3D ont une intersection ou pas. Les données d'entrée sont les coordonnées (une paire de valeurs par sommet) des deux triangles et la sortie est une valeur booléenne qui indique s'il y a intersection entre les deux triangles (1 codant vrai) ou pas (0 codant faux). La métrique d'erreur pour jmeint est le taux d'erreur sur l'ensemble des sorties. Pour évaluer la qualité en sortie, nous calculons le taux d'erreur, par mesure de la distance de Hamming en comparant un par un les éléments des vecteurs booléens pour 1 000 000 de paires de triangles données dans la suite Axbench, retournés par les applications en virgule flottante et en virgule fixe (en supposant ici la taille maximale activée). La figure 4.2 indique qu'à partir de 14 bits dans la partie fractionnaire, le taux d'erreur est $\leq 0.03\%$, c.-à-d. que pour presque tous les couples de triangles, la sortie calculée avec le programme en virgule flottante et celle retournée par celui en virgule fixe sont identiques. Avec plus de 5 bits fractionnaires, on note que le *round to the nearest representable value (NearOdd)* est le meilleur mode d'arrondi pour jmeint en terme de précision.

Le filtre de Sobel est utilisé dans des applications de traitement d'images et de vision par ordinateur, en particulier pour des algorithmes de détection de contour. La détection de contour est une technique de traitement d'image utilisée pour

délimiter les différentes régions d'une image. La donnée d'entrée est une image RGB et la sortie est une image en niveau de gris (format PNG) dans laquelle les contours des régions sont mis en évidence. Le gradient de chaque pixel est calculé par une convolution avec deux filtres: horizontal et vertical, qui sont des matrices de taille 3×3. Les métriques pour l'évaluation des erreurs pour le filtre de Sobel sont l'erreur moyenne quadratique (RMSE), le rapport signal sur bruit (PSNR) et l'indice de mesure de similarité structurelle (SSIM) entre des images. Chacune des métriques est calculée avec les paramètres de l'image retournée par le programme à virgule flottante et l'image retournée avec le programme en virgule fixe, pour plusieurs modes d'arrondi. Les expériences sont effectuées sur 100 images choisies de manière aléatoire. La figure 4.3 présente, pour chaque taille de la partie fractionnaire, la valeur moyenne de la RMSE calculée avec 100 images et son écart type. Pour cette métrique, plus la valeur est faible, plus la qualité de l'image retournée par le programme en virgule fixe est bonne. Les résultats indiquent qu'à partir de 10 bits dans la partie fractionnaire, nous avons une image de sortie acceptable, c.-à-d. le RMSE est ≤ 0.01. La figure 4.4 présente, pour chaque taille de la partie fractionnaire, la valeur moyenne du PSNR calculée sur 100 images et son écart type. Pour le PSNR, plus il est élevé, plus la qualité de l'image retournée par le programme en virgule fixe est bonne. Les résultats indiquent qu'à partir de 10 bits dans la partie fractionnaire, nous avons un PSNR supérieur à 50dB (seuil classique dans le domaine). Le RMSE et le PSNR estiment les erreurs absolues entre les pixels des deux images comparées. Cependant, ces deux métriques sont moins corrélées à la perception humaine de la qualité de l'image que le SSIM. Le SSIM évalue la similarité entre l'image de référence, c.-à-d. l'image calculée avec le programme en virgule flottante et l'image renvoyée par le programme en virgule fixe. Les valeurs du SSIM sont dans l'intervalle $[0, 1]$. Si le SSIM est égal à 1, les deux images comparées sont identiques. La figure 4.5 montre qu'à partir de 8 bits dans la partie fractionnaire, les deux images sont similaires, avec un SSIM ≥ 0.999.

Le forwardk2j prend en entrée les angles d'un bras d'un robot à 2 articulations et calcule la position de son actionneur. Pour évaluer les erreurs induites par le calcul en virgule fixe, nous utilisons la métrique d'erreur relative, calculée sur 10 000 couples de coordonnées donnés dans la suite Axbench. Pour chaque couple de coordonnées, la valeur maximale des erreurs des deux coordonnées

est retournée, et la valeur moyenne et son écart type sont calculés sur les 10 000 valeurs maximales. La figure 4.6 indique qu'à partir de 14 bits dans la partie fractionnaire, l'erreur relative est $\leq 0.01\%$. Nous notons que pour un nombre de bits fractionnaires dans l'intervalle $[1, 10]$ varie selon le mode d'arrondi. À partir de 7 bits et plus, le *round to the nearest representable value (NearOdd)* est le meilleur mode d'arrondi pour le forwardk2j en terme de précision.

Évaluation des additions et des multiplications à taille réduite

Nos premières expériences sur l'évaluation de la consommation d'énergie sont effectuées avec seulement des additionneurs et des multiplicateurs approximatifs, à taille variable, parmi les plus étudiés de l'état de l'art. L'objectif est d'estimer la réduction globale d'énergie apportée par ces unités approximatives qui sont seulement évaluées de façon isolée dans la littérature. L'évaluation globale est effectuée sur des applications exécutées sur le processeur RISC-V étendu avec les opérateurs à taille réduite `a.add` et `a.mul`. Nous avons effectué les expériences sur les trois applications: jmeint, filtre de Sobel et forwardk2j. La première étape est l'étude de la décomposition en différentes catégories d'instructions exécutées. La deuxième étape consiste à évaluer le potentiel en terme d'économie d'énergie de ces applications exécutées avec `a.add` et `a.mul`.

L'algorithme jmeint a plusieurs points de sortie. Le nombre d'instructions exécutées peut donc varier en fonction des valeurs en entrée et de la taille courante configurée pour les calculs comme indiqué sur la Figure 4.7. Les expériences ont été réalisées avec 10 000 couples de triangles. Pour une taille active des unités de calcul dans $\{1, 2, ...8\}$ bits, le nombre d'instructions exécutées varie d'une entrée à l'autre. Par conséquent, l'étude des catégories d'instructions exécutées est effectuée en calculant la valeur moyenne des instructions et l'écart type comme indiqué sur la Figure 4.8. L'application jmeint comporte 11.82% d'instructions approximatives `a.add` et `a.mul`.

Pour plusieurs couples de triangles parmi les 10 000 utilisés, les points de sortie sont (1) ou (2) sur la Figure 2.3. Ainsi, la consommation d'énergie peut être réduite jusqu'à 44%. La forte réduction d'énergie ne dépend pas seulement des calculs avec une taille réduite, mais aussi d'un nombre plus faible d'instructions pour effectuer les tests de rejet précoce des points de sortie (1) ou (2) de la Fig-

ure 2.3.

Pour le filtre de Sobel, le code source proposé dans la suite Axbench est en virgule flottante et n'est pas optimisé pour l'exécution sur un processeur embarqué avec des unités entières ou virgule fixe. Après conversion du code source flottant en virgule fixe, nous avons appliqué plusieurs optimisations usuelles pour réduire la consommation d'énergie. Ces optimisations sont:

- V1: le stockage matriciel (2D) des pixels est remplacé par un stockage vectoriel (1D) pour réduire le coût de calcul des adresses;
- V2: V1 + les lignes redondantes dans le code initial sont supprimées pour éviter de calculer plusieurs fois la même valeur car les 3 composantes RGB d'un pixel ont la même valeur en niveau de gris;
- V3: V2 + les boucles pour les calculs de convolution sont déroulées pour réduire le coût des instructions introduites par le calcul des indices de boucles;
- V4: V3 + éviter l'application du filtre sur des valeurs systématiquement nulles. Pour appliquer simplement les filtres sur les pixels qui sont au bord de l'image, dans la version originale du code, des zéros artificiels sont ajoutés juste autour de la matrice originale de l'image (*padding*). Dans cette optimisation, nous avons réécrit les boucles pour enlever ces calculs avec des zéros.

La figure 4.9 présente l'impact des optimisations en terme de nombre d'instructions exécutées. Le pourcentage d'instructions à taille réduite (c.-à-d. `a.add` et `a.mul`) est égal à 6.8% dans la version originale du programme et à 17% dans le programme le plus optimisé V4. Malgré les optimisations, la plupart des instructions exécutées restent sur la taille originale (83% dans la version la plus optimisée). Ces instructions correspondent à des calculs exacts (non approchables) destinés p.~ex. aux calculs d'adresses, aux accès à la mémoire de données et aux instructions de branchement. Le pourcentage élevé des instructions de chargement de la mémoire est dû au fait que les pixels de l'image sont stockés dans la mémoire de données et que pour calculer le gradient d'un pixel, des matrices 3×3 des deux filtres et celle de la fenêtre autour du pixel courant, sont aussi chargés de la mémoire de données.

La figure 4.12 présente le résultat de l'évaluation de la consommation d'énergie de l'application du filtre Sobel avec des unités approximatives pour les instructions `a.add` et `a.mul`. Les résultats indiquent que l'énergie peut être réduite seulement jusqu'à 7% pour l'application complète (alors que le gain pour les mêmes unités

isolées est beaucoup plus important).

La figure 4.10 présente les catégories d'instructions exécutées dans l'application test forwardk2j. Celle-ci comprend plus d'instructions `a.add` et `a.mul`, ici 46%, que les applications filtre de Sobel et jmeint. Ainsi, sa réduction d'énergie pour une erreur $\leq 0.1\%$ (12% comme indiqué sur la Figure 4.13) est bien supérieure à la réduction d'énergie des applications du filtre de Sobel (5%) et jmeint (2%, si on considère les points de sortie réels) pour un même taux d'erreur.

Ces résultats indiquent que même après optimisation des algorithmes, la réduction d'énergie n'est pas aussi importante que celle obtenue en évaluant seulement les opérateurs isolés comme on peut le voir dans le tableau 4.1. Les petites réductions d'énergie sont dues au fait que les applications en calcul approximatif incluent toujours plusieurs catégories d'instructions autres que les `a.add` et `a.mul`. Pour réduire encore plus la consommation d'énergie, nous étendons le principe réduction de la taille des unités de calcul aux unités logiques et de mémoire de données.

Évaluation des unités de calcul et des unités mémoire à taille réduite

Notre première étude portait seulement sur l'ajout d'instructions, et des unités à taille réduite correspondantes aux additions et aux multiplications, parmi les plus étudiées dans l'état de l'art. Toutefois, ces unités ne sont pas suffisantes pour assurer une forte réduction de la consommation d'énergie. Lors de l'étude des catégories d'instructions exécutées, nous remarquons que chaque application comprend de nombreuses instructions logiques et d'accès à la mémoire de données qui ne sont pas exécutées avec une taille réduite. Certaines instructions d'accès à la mémoire de données pourraient être exécutées avec une taille réduite, c.-à-d. charger/stocker seulement un certain nombre de MSB depuis/dans la mémoire. Nous étendons donc le principe d'approximation utilisé aux unités logiques et de mémoire de données (celle d'instructions ne peut pas être approchée dans notre architecture). Notre objectif est d'évaluer l'impact des instructions arithmétiques et logiques à taille réduite (`a.al`, et `a.mul`) et des unités mémoire à taille réduite (`a.mem`) sur la consommation d'énergie et la qualité des résultats des applications.

Lorsque les instructions mémoire à taille réduite sont ajoutées au processeur

RISC-V, le pourcentage d'instructions approximatives est plus élevé que dans la version avec seulement les unités de calcul approximatives, ce qui réduit l'énergie plus encore. Pour jmeint les instructions à taille réduite passent de 11.82% à 77.4%, de 17% à 73.7% pour le filtre de Sobel et de 46% à 81.8% pour forwardk2j. Pour un taux d'erreur acceptable, p. ex. $\leq 0.1\%$, la réduction d'énergie avec les instructions à taille réduite passe de 2% à 14% pour jmeint, de 5% à 16% pour le filtre de Sobel, et de 12% à 15% pour forwardk2j. Forwardk2j inclut plus d'instructions de calcul que d'instructions mémoire, ce qui justifie les faibles améliorations de la réduction d'énergie avec l'unité de mémoire à taille réduite. Nous pouvons conclure que les unités mémoires à taille réduite sont plus importantes pour les types d'applications qui font plus d'accés mémoire que de calcul, comme p.~ex. les applications de traitement d'images qui sont très utilisées dans le domaine du calcul approximatif.

Modèle d'énergie global

Le modèle d'énergie proposé inclut des paramètres logiciels et matériels. Il permet à un concepteur logiciel et matériel d'avoir une estimation rapide de l'impact des optimisations effectuées au niveau algorithmique et/ou au niveau circuit ou architecture, sur la consommation d'énergie d'une application donnée.

Un concepteur logiciel peut utiliser notre modèle pour avoir un aperçu sur les parties de l'application qui auront le plus d'impact sur la réduction de l'énergie. Il permet d'avoir une idée sur les limites des gains d'énergie possibles, pour une application donnée, connaissant les catégories et le nombre d'instructions exécutées. Connaissant la fraction des instructions à taille réduite et un budget d'énergie, on peut estimer la taille adéquate pour la configuration des unités approximatives. Inversement, le concepteur, connaissant la taille minimale requise pour une qualité de résultat donnée, peut avoir une estimation de la réduction d'énergie en se basant sur la fraction d'instructions approximatives exécutées. Par exemple, la figure 5.1a indique que pour une réduction d'énergie jusqu'à 20%, un programme peut être exécuté avec une taille active de 1 à 20 bits. Par exemple si un concepteur logiciel désire une réduction d'énergie de 20% et que l'application inclut 80% d'instructions approximatives dont 60% d'instructions mémoire approximatives, c.-à-d. f_{A_m} = 0.6, l'application pourrait être exécutée avec 15 bits; comme indiqué sur la figure 5.1a.

En plus des extensions sur RISC-V avec des unités de calcul et d'accès à la mémoire de données à taille réduite, ces unités pourraient être optimisées par les concepteurs matériels pour améliorer la réduction d'énergie (p.~ex. avec de nouveaux algorithmes ou de meilleures architectures). Un concepteur matériel peut utiliser notre modèle pour trouver les paramètres architecturaux qui ont du potentiel. Le concepteur matériel peut décider d'optimiser les unités de calcul et/ou de mémoire, en fonction des gains d'énergie espérés. Le modèle énergétique global proposé permet d'estimer la réduction d'énergie sur une application donnée lorsque les unités matérielles sont optimisées. Ainsi lorsque l'énergie consommée par une unité, p.~ex. un multiplieur, est réduite de moitié par des optimisations, le modèle énergétique proposé permet d'estimer la réduction énergétique globale sur une application donnée. Par exemple, les figures 5.3a, 5.3b, 5.3c indiquent que pour $b = 8$ bits, lorsque l'énergie d'une instruction approximative est réduite jusqu'à 60%, la réduction globale sur le filtre de Sobel est égale à 30% pour les unités de calcul (`a.al` et `a.mul`) et égal à 40% pour les unités de mémoire (`a.mem`). Pour $b = 16$ bits, la réduction d'énergie diminue: 23% avec `a.al`, 24% avec `a.mul` et 35% avec `a.mem`.

L'étude effectuée sur notre modèle d'énergie nous suggère une analogie avec la loi d'Amdhal. En calcul parallèle [13], la loi d'Amdahl stipule que pour qu'un algorithme donné s'exécute rapidement sur une plateforme à plusieurs processeurs, augmenter le nombre de processeurs n'est pas suffisant pour réduire le temps de calcul. La fraction de la partie parallélisable est également impliquée et clé. Avec notre modèle, nous montrons clairement qu'en calcul approximatif, la stratégie d'approximation n'est pas suffisante pour obtenir une réduction importante des coûts de calcul. Un autre paramètre important est impliqué dans la réduction d'énergie. Il s'agit de la fraction de la partie où la stratégie d'approximation peut être appliqué sans un large impact sur la qualité de sortie, comme indiqué sur les figures 5.1 et 5.2.

Conclusion

La première partie de la thèse présente un aperçu des solutions proposées en calcul approximatif et étudie les travaux les plus proches de notre thèse. Plusieurs solutions sont présentées dans l'état de l'art. Cependant, la plupart d'entre elles

sont destinées à des architectures à virgule flottante qui sont trop complexes pour les systèmes embarqués.

Nos travaux ciblent des architectures à unités entières (ou en virgule fixe) mais la plupart des applications sont décrites en représentation à virgule flottante. Nous avons d'abord converti les applications flottantes en représentation à virgule fixe. Les erreurs dues à la conversion en virgule fixe sont évaluées afin de voir si les applications sont appropriées à la représentation en virgule fixe. Ensuite, nous avons étudié le compromis entre la qualité de sortie et la réduction d'énergie de ces applications une fois converties en virgule fixe lorsqu'elles sont exécutées avec des unités de calcul approximatives. Parmi les unités approximatives proposées, nous avons utilisé des unités approximatives pour l'addition (`a.add`) et la multiplication (`a.mul`) parmi les plus étudiées dans l'état de l'art. Cependant, l'évaluation de la plupart de ces unités est effectuée de façon isolée, c.-à-d. elles ne sont pas intégrées dans un processeur exécutant une application complète. Nous avons étudié l'impact de ces unités lorsqu'elles sont intégrées dans un processeur RISC-V et exécutées sur trois applications: jmeint, filtre de Sobel, forwardk2j. Les résultats indiquent que lorsque ces unités sont évaluées de façon isolée, l'énergie des opérations de calcul seules peut être réduite jusqu'à 46% alors que lorsqu'elles sont évaluées pour des applications complètes exécutées sur un processeur dans lequel sont embarquées ces unités, la réduction d'énergie est seulement au mieux de 2% pour jmeint, 5% pour le filtre Sobel et 12% pour forwardk2j. La faible réduction de la consommation d'énergie avec seulement les unités pour l'addition et la multiplication est due au fait que ces applications comportent d'autres types d'opérations qui sont nécessairement exécutées avec une taille maximale (les instructions classiques ou non-approchables). Parmi ces opérations classiques, nous avons, p.~ex. les instructions de sauts, celles pour les calculs d'adresses, *etc*. Nous avons d'abord étendu le RISC-V avec `a.add` et `a.mul` à taille réduite. Puis, nous avons ajouté d'autres unités approximatives avec les instructions logiques `a.lgc`, et les opérations mémoire `a.ld` et `a.st`. Les résultats indiquent que pour une erreur inférieur à 0.1%, la réduction d'énergie est améliorée : de 2% à 14% pour jmeint, de 5% à 16% pour le filtre Sobel, de 12% à 15% pour forwardk2j qui inclue plus de calcul que d'accés mémoire.

Nous avons également proposé un modèle d'énergie global qui inclut à la fois des paramètres logiciels et matériels. Il permet aux concepteurs de logiciels et

de matériels, pour une application donnée, d'avoir un aperçu global de l'impact d'optimisations d'un code source ou de circuits sur la consommation d'énergie. L'étude avec le modèle nous a permis d'établir une analogie avec la loi d'Amdahl habituelle en calcul parallèle.

Les unités évaluées sont configurables au moment de l'exécution. Nous souhaitons pouvoir aller plus loin dans nos expériences en explorant la *configurabilité* des unités. Pour cela, des paramètres dynamiques tels que la qualité de la sortie, le budget d'énergie peuvent être pris en compte dans les décisions dynamiques lors de la configuration. De plus, les fonctions mathématiques qui sont coûteuses pour les systèmes embarquée pourraient être approximées pour améliorer la réduction d'énergie tout en maintenant une qualité de sortie acceptable.

PUBLICATIONS

[58] Geneviève Ndour et al., "Evaluation of approximate operators case study: sobel filter application executed on an approximate RISC-V platform", *in*: *Proceedings of the 18th International Conference on Embedded Computer Systems: Architectures, Modeling, and Simulation*, ACM, 2018, pp. 146–149.

[59] Geneviève Ndour et al., "Evaluation of Variable Bit-width Units in a RISC-V Processor for Approximate Computing", *in*: *Proceedings of the 5th Workshop on design of Low Power EMbedded Systems (LP-EMS) Co-located with International Conference on Computing Frontiers*, ACM, 2019.

[81] Tiago Trevisan et al., "ApproxRISC: An Approximate Computing Infrastructure For RISC-V", *in*: *RISC-V Workshop, Barcelona, Spain*, 2018.

BIBLIOGRAPHY

[1] https://github.com/mhfan/libfixmath/tree/master/libfixmath.

[2] http://www.codeproject.com/Articles/37636/Fixed-Point-Class.

[3] https://github.com/mbedded-ninja/MFixedPoint.

[4] https://github.com/gsarkis/libfi.

[5] http://math.nist.gov/scimark2/.

[6] http://code.google.com/p/zxing/.

[7] http://www.jmonkeyengine.com/.

[8] http://rsbweb.nih.gov/ij/.

[9] http://www.planet-source-code.com/vb/scripts/ShowCode.asp?txtCodeId=5590&lngWId=2

[10] https://riscv.org/, 2018.

[11] https://github.com/riscv/riscv-isa-sim, 2018.

[12] Ankur Agrawal et al., "Approximate computing: Challenges and opportunities", *in*: *2016 IEEE International Conference on Rebooting Computing (ICRC)*, IEEE, 2016, pp. 1–8.

[13] Gene M Amdahl, "Validity of the single processor approach to achieving large scale computing capabilities", *in*: *Proceedings of the April 18-20, 1967, spring joint computer conference*, ACM, 1967, pp. 483–485.

[14] Jason Ansel et al., "Language and compiler support for auto-tuning variable-accuracy algorithms", *in*: *International Symposium on Code Generation and Optimization (CGO 2011)*, IEEE, 2011, pp. 85–96.

[15] Jason Ansel et al., "Siblingrivalry: online autotuning through local competitions", *in*: *Proceedings of the 2012 international conference on Compilers, architectures and synthesis for embedded systems*, ACM, 2012, pp. 91–100.

[16] Algirdas Avizienis, "Design of fault-tolerant computers.", *in*: *AFIPS Fall Joint Computing Conference*, 1967, pp. 733–743.

[17] Woongki Baek and Trishul M Chilimbi, “Green: a framework for supporting energy-conscious programming using controlled approximation”, *in*: *ACM Sigplan Notices*, vol. 45, 6, ACM, 2010, pp. 198–209.

[18] Benjamin Barrois, “Methods to evaluate accuracy-energy trade-off in operator-level approximate computing”, PhD thesis, Rennes 1, 2017.

[19] Christian Bienia and Kai Li, *Benchmarking modern multiprocessors*, Princeton University Princeton, 2011.

[20] Christian Bienia et al., “The PARSEC benchmark suite: Characterization and architectural implications”, *in*: *Proceedings of the 17th international conference on Parallel architectures and compilation techniques*, ACM, 2008, pp. 72–81.

[21] Justine Bonnot, Karol Desnos, and Daniel Menard, “Fast Simulation-Based Fixed-point Refinement with Inferential Statistics”, *in*: *2019 Design Automation Conference*, ACM, 2019.

[22] Justine Bonnot, Erwan Nogues, and Daniel Menard, “New non-uniform segmentation technique for software function evaluation”, *in*: *Application-specific Systems, Architectures and Processors (ASAP), 2016 IEEE 27th International Conference on*, IEEE, 2016, pp. 131–138.

[23] Nicolas Brisebarre, Jean-Michel Muller, and Arnaud Tisserand, “Computing machine-efficient polynomial approximations”, *in*: *ACM Transactions on Mathematical Software (TOMS)* 32.*2* (2006), pp. 236–256.

[24] Arun Chandrasekharan, Daniel Große, and Rolf Drechsler, “ProACt: A Processor for High Performance On-demand Approximate Computing”, *in*: *Proceedings of the on Great Lakes Symposium on VLSI 2017*, ACM, 2017, pp. 463–466.

[25] Sylvain Chevillard, Mioara Joldeş, and Christoph Lauter, “Sollya: An environment for the development of numerical codes”, *in*: *International Congress on Mathematical Software*, Springer, 2010, pp. 28–31.

[26] Vinay K Chippa et al., “Scalable effort hardware design: Exploiting algorithmic resilience for energy efficiency”, *in*: *Proceedings of the 47th Design Automation Conference*, ACM, 2010, pp. 555–560.

[27] Jason Clemons et al., “MEVBench: A mobile computer vision benchmarking suite”, *in*: *Workload Characterization (IISWC), 2011 IEEE International Symposium on*, IEEE, 2011, pp. 91–102.

[28] Hadi Esmaeilzadeh et al., “Neural acceleration for general-purpose approximate programs”, *in*: *Proceedings of the 2012 45th Annual IEEE/ACM International Symposium on Microarchitecture*, IEEE Computer Society, 2012, pp. 449–460.

[29] Philip Hans Franses, “A note on the mean absolute scaled error”, *in*: *International Journal of Forecasting* 32.*1* (2016), pp. 20–22.

[30] Jason George et al., “Probabilistic arithmetic and energy efficient embedded signal processing”, *in*: *Proceedings of the 2006 international conference on Compilers, architecture and synthesis for embedded systems*, ACM, 2006, pp. 158–168.

[31] Marin Golub, Domagoj Jakobović, and Leo Budin, “Genetic algorithms in real-time imprecise computing”, *in*: *Journal of computing and information technology* 8.*3* (2000), pp. 249–257.

[32] Paul Goodwin and Richard Lawton, “On the asymmetry of the symmetric MAPE”, *in*: *International journal of forecasting* 15.*4* (1999), pp. 405–408.

[33] Beayna Grigorian and Glenn Reinman, “Accelerating divergent applications on simd architectures using neural networks”, *in*: *ACM Transactions on Architecture and Code Optimization (TACO)* 12.*1* (2015), p. 2.

[34] Marcus Hähnel and Hermann Härtig, “Heterogeneity by the Numbers: A Study of the {ODROID} XU+ E big. LITTLE Platform”, *in*: *6th Workshop on Power-Aware Computing and Systems (HotPower 14)*, 2014.

[35] Soheil Hashemi, R Bahar, and Sherief Reda, “DRUM: A dynamic range unbiased multiplier for approximate applications”, *in*: *Proceedings of the IEEE/ACM International Conference on Computer-Aided Design*, IEEE Press, 2015, pp. 418–425.

[36] Henry Hoffmann, “JouleGuard: energy guarantees for approximate applications”, *in*: *Proceedings of the 25th Symposium on Operating Systems Principles*, ACM, 2015, pp. 198–214.

[37] *International Roadmap for Devices and Systems, Executive Summary*, tech. rep., IEEE, 2017.

[38] Andrew B Kahng and Seokhyeong Kang, "Accuracy-configurable adder for approximate arithmetic designs", *in*: *Design Automation Conference (DAC), 2012 49th ACM/EDAC/IEEE*, IEEE, 2012, pp. 820–825.

[39] Daya S Khudia et al., "Rumba: An online quality management system for approximate computing", *in*: *Computer Architecture (ISCA), 2015 ACM/IEEE 42nd Annual International Symposium on*, IEEE, 2015, pp. 554–566.

[40] Khaing Yin Kyaw, Wang Ling Goh, and Kiat Seng Yeo, "Low-power high-speed multiplier for error-tolerant application", *in*: *Electron Devices and Solid-State Circuits (EDSSC), 2010 IEEE International Conference of*, IEEE, 2010, pp. 1–4.

[41] Chunho Lee, Miodrag Potkonjak, and William H Mangione-Smith, "Media-Bench: a tool for evaluating and synthesizing multimedia and communicatons systems", *in*: *Proceedings of the 30th annual ACM/IEEE international symposium on Microarchitecture*, IEEE Computer Society, 1997, pp. 330–335.

[42] Dong-U Lee et al., "Hierarchical segmentation for hardware function evaluation", *in*: *IEEE Transactions on Very Large Scale Integration (VLSI) Systems* 17.*1* (2009), pp. 103–116.

[43] In Lee and Kyoochun Lee, "The Internet of Things (IoT): Applications, investments, and challenges for enterprises", *in*: *Business Horizons* 58.*4* (2015), pp. 431–440.

[44] Man-Lap Li et al., "The ALPBench benchmark suite for complex multimedia applications", *in*: *Workload Characterization Symposium, 2005. Proceedings of the IEEE International*, IEEE, 2005, pp. 34–45.

[45] Avinash Lingamneni et al., "Energy parsimonious circuit design through probabilistic pruning", *in*: *2011 Design, Automation & Test in Europe*, IEEE, 2011, pp. 1–6.

[46] Jane WS Liu et al., "Imprecise computations", *in*: *Proceedings of the IEEE* 82.*1* (1994), pp. 83–94.

[47] Shih-Lien Lu, “Speeding up processing with approximation circuits”, *in*: *Computer* 37.*3* (2004), pp. 67–73.

[48] Divya Mahajan et al., “Prediction-based quality control for approximate accelerators”, *in*: *Workshop on Approximate Computing Across the System Stack*, 2015.

[49] Christophe Mazenc, Xavier Merrheim, and J-M Muller, “Computing Functions cos/sup-1/and sin/sup-1/Using CORDIC”, *in*: *IEEE Transactions on Computers* 42.*1* (1993), pp. 118–122.

[50] Daniel Menard et al., “Automatic floating-point to fixed-point conversion for DSP code generation”, *in*: *Proceedings of the 2002 international conference on Compilers, architecture, and synthesis for embedded systems*, ACM, 2002, pp. 270–276.

[51] John F. Meyer, “On evaluating the performability of degradable computing systems”, *in*: *IEEE Transactions on computers 8* (1980), pp. 720–731.

[52] Jason E Miller et al., “Graphite: A distributed parallel simulator for multicores”, *in*: *HPCA-16 2010 The Sixteenth International Symposium on High-Performance Computer Architecture*, IEEE, 2010, pp. 1–12.

[53] Sparsh Mittal, “A survey of techniques for approximate computing”, *in*: *ACM Computing Surveys (CSUR)* 48.*4* (2016), p. 62.

[54] Daniel Molka et al., “Characterizing the energy consumption of data transfers and arithmetic operations on x86- 64 processors”, *in*: *Green Computing Conference, 2010 International*, IEEE, 2010, pp. 123–133.

[55] Thierry Moreau et al., *Approximate computing on programmable socs via neural acceleration*, tech. rep., Technical report, 2014.

[56] J.-M. Muller et al., *Handbook of Floating-Point Arithmetic*, Birkhauser, 2010, ISBN: 978-0-8176-4704-9.

[57] Ramanathan Narayanan et al., “Minebench: A benchmark suite for data mining workloads”, *in*: *Workload Characterization, 2006 IEEE International Symposium on*, IEEE, 2006, pp. 182–188.

[58] Geneviève Ndour et al., “Evaluation of approximate operators case study: sobel filter application executed on an approximate RISC-V platform”, *in*: *Proceedings of the 18th International Conference on Embedded Computer Systems: Architectures, Modeling, and Simulation*, ACM, 2018, pp. 146–149.

[59] Geneviève Ndour et al., “Evaluation of Variable Bit-width Units in a RISC-V Processor for Approximate Computing”, *in*: *Proceedings of the 5th Workshop on design of Low Power EMbedded Systems (LP-EMS) Co-located with International Conference on Computing Frontiers*, ACM, 2019.

[60] J Oblonsky, “A self-correcting computer”, *in*: *Digital Information Processors*, Interscience, 1962, pp. 533–542.

[61] Daniele Jahier Pagliari et al., “A methodology for the design of dynamic accuracy operators by runtime back bias”, *in*: *2017 Design, Automation & Test in Europe Conference & Exhibition (DATE)*, IEEE, 2017, pp. 1165–1170.

[62] Krishna Palem and Avinash Lingamneni, “Ten years of building broken chips: The physics and engineering of inexact computing”, *in*: *ACM Transactions on Embedded Computing Systems (TECS)* 12.*2s* (2013), p. 87.

[63] Krishna V Palem, “Energy aware algorithm design via probabilistic computing: from algorithms and models to Moore’s law and novel (semiconductor) devices”, *in*: *CASES*, Citeseer, 2003, pp. 113–116.

[64] Krishna V Palem, “Energy aware computing through probabilistic switching: A study of limits”, *in*: *IEEE Transactions on Computers* 54.*9* (2005), pp. 1123–1137.

[65] Karthick Nagaraj Parashar, Daniel Menard, and Olivier Sentieys, “Accelerated performance evaluation of fixed-point systems with un-smooth operations”, *in*: *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 33.*4* (2014), pp. 599–612.

[66] Konstantinos Parasyris et al., “A Significance-Aware Software Stack for Computing on Unreliable Hardware”, *in*: *Significance* 26.*27* (), p. 28.

[67] Jongse Park et al., *Expax: A framework for automating approximate programming*, tech. rep., Georgia Institute of Technology, 2014.

[68] Jongse Park et al., “Flexjava: Language support for safe and modular approximate programming”, *in*: *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering*, ACM, 2015, pp. 745–757.

[69] Arnab Raha et al., “Quality configurable reduce-and-rank for energy efficient approximate computing”, *in*: *Proceedings of the 2015 Design, Automation & Test in Europe Conference & Exhibition*, EDA Consortium, 2015, pp. 665–670.

[70] Abbas Rahimi et al., “Approximate associative memristive memory for energy-efficient GPUs”, *in*: *Proceedings of the 2015 Design, Automation & Test in Europe Conference & Exhibition*, EDA Consortium, 2015, pp. 1497–1502.

[71] David A. Rennels, “Fault-Tolerant Computing? Concepts and Examples”, *in*: *IEEE Transactions on computers 12* (1984), pp. 1116–1129.

[72] Pooja Roy et al., “Asac: Automatic sensitivity analysis for approximate computing”, *in*: *ACM SIGPLAN Notices*, vol. 49, 5, ACM, 2014, pp. 95–104.

[73] Adrian Sampson et al., “Accept: A programmer-guided compiler framework for practical approximate computing”, *in*: *University of Washington Technical Report UW-CSE-15-01* 1 (2015).

[74] Adrian Sampson et al., “EnerJ: Approximate data types for safe and general low-power computation”, *in*: *ACM SIGPLAN Notices*, vol. 46, 6, ACM, 2011, pp. 164–174.

[75] Naresh R Shanbhag et al., “Stochastic computation”, *in*: *Design Automation Conference (DAC), 2010 47th ACM/IEEE*, IEEE, 2010, pp. 859–864.

[76] Qingchuan Shi, Hamza Omar, and Omer Khan, “Exploiting the tradeoff between program accuracy and soft-error resiliency overhead for machine learning workloads”, *in*: *arXiv preprint arXiv:1707.02589* (2017).

[77] Stelios Sidiroglou-Douskos et al., “Managing performance vs. accuracy trade-offs with loop perforation”, *in*: *Proceedings of the 19th ACM SIGSOFT symposium and the 13th European conference on Foundations of software engineering*, ACM, 2011, pp. 124–134.

[78] R Sivakumar and D Nedumaran, “Comparative study of speckle noise reduction of ultrasound b-scan images in matrix laboratory environment”, *in*: *International Journal of Computer Applications* 10.*9* (2010), pp. 46–50.

[79] Andrew M Steane, “Error correcting codes in quantum theory”, *in*: *Physical Review Letters* 77.*5* (1996), p. 793.

[80] David B Thomas, “A general-purpose method for faithfully rounded floating-point function approximation in FPGAs”, *in*: *Computer Arithmetic (ARITH), 2015 IEEE 22nd Symposium on*, IEEE, 2015, pp. 42–49.

[81] Tiago Trevisan et al., “ApproxRISC: An Approximate Computing Infrastructure For RISC-V”, *in*: *RISC-V Workshop, Barcelona, Spain*, 2018.

[82] Evangelos Vasilakis, “An instruction level energy characterization of arm processors”, *in*: *Foundation of Research and Technology Hellas, Inst. of Computer Science, Tech. Rep. FORTH-ICS/TR-450* (2015).

[83] Vassilis Vassiliadis et al., “Exploiting significance of computations for energy-constrained approximate computing”, *in*: *International Journal of Parallel Programming* 44.*5* (2016), pp. 1078–1098.

[84] Vassilis Vassiliadis et al., “Towards automatic significance analysis for approximate computing”, *in*: *2016 IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*, IEEE, 2016, pp. 182–193.

[85] Sravanthi Kota Venkata et al., “SD-VBS: The San Diego vision benchmark suite”, *in*: *Workload Characterization, 2009. IISWC 2009. IEEE International Symposium on*, IEEE, 2009, pp. 55–64.

[86] Ajay K Verma, Philip Brisk, and Paolo Ienne, “Variable latency speculative addition: A new paradigm for arithmetic circuit design”, *in*: *Proceedings of the conference on Design, automation and test in Europe*, ACM, 2008, pp. 1250–1255.

[87] Jack E Volder, “The CORDIC trigonometric computing technique”, *in*: *IRE Transactions on electronic computers 3* (1959), pp. 330–334.

[88] Zhou Wang, Alan C Bovik, and Hamid R Sheikh, “Structural similarity based image quality assessment”, *in*: *Digital Video image quality and perceptual coding* (2005), pp. 225–241.

[89] Andrew Waterman et al., “The risc-v instruction set manual”, *in*: *volume I: User-level ISA, version 2.0, EECS Department, University of California, Berkeley, Tech. Rep. UCB/EECS-2014-54* (2014).

[90] Darrell Whitley, “A genetic algorithm tutorial”, *in*: *Statistics and computing* 4.*2* (1994), pp. 65–85.

[91] Dongbin Xiu, “Fast numerical methods for stochastic computations: a review”, *in*: *Communications in computational physics* 5.*2-4* (2009), pp. 242–272.

[92] Qiang Xu, Todd Mytkowicz, and Nam Sung Kim, “Approximate computing: A survey”, *in*: *IEEE Design & Test* 33.*1* (2016), pp. 8–22.

[93] Amir Yazdanbakhsh et al., *AxBench: A Benchmark Suite for Approximate Computing Across the System Stack*, tech. rep., Georgia Institute of Technology, 2016.

[94] Qian Zhang et al., “Approxit: An approximate computing framework for iterative methods”, *in*: *2014 51st ACM/EDAC/IEEE Design Automation Conference (DAC)*, IEEE, 2014, pp. 1–6.

[95] Ning Zhu, Wang Ling Goh, and Kiat Seng Yeo, “An enhanced low-power high-speed adder for error”, *in*: (2009).

[96] Ning Zhu et al., “Enhanced low-power high-speed adder for error-tolerant application”, *in*: *SoC Design Conference (ISOCC), 2010 International*, IEEE, 2010, pp. 323–327.

LIST OF FIGURES

2.1 Floating-point representation 20
2.2 Fixed-point representation 21
2.3 Jmeint algorithm steps 33
2.4 Sobel filter input/output 35
2.5 32-bit RISC-V base instruction formats in our work [89] 38
2.6 RISC-V Architecture 40

3.1 Our extended RISC-V Architecture 50
3.2 Relative energy values of reduced width and full width instructions . 53
3.3 Average relative energy values of reduced width and full width instructions 54

4.1 Methodology of output quality *vs* energy reduction trade-off evaluation. 64
4.2 Errors evaluation on fixed-point jmeint. 66
4.3 RMSE evaluation on fixed-point Sobel filter. 67
4.4 PSNR evaluation on fixed-point Sobel filter. 68
4.5 SSIM evaluation on fixed-point Sobel filter. 69
4.6 Errors evaluation on fixed-point Forwardk2j. 70
4.7 Number of executed instructions 71
4.8 Jmeint instruction breakdown. 72
4.9 Sobel instruction breakdown. 73
4.10 Forwardk2j instruction breakdown. 75
4.11 Jmeint energy consumption with `a.add` and `a.mul`. 77
4.12 Sobel filter energy consumption with `a.add` and `a.mul`. 78
4.13 Forwardk2j energy consumption with `a.add` and `a.mul` 79
4.14 Jmeint instruction breakdown. 81
4.15 Sobel filter instruction breakdown. 82
4.16 Forwardk2j instruction breakdown. 83
4.17 Jmeint output quality evaluation. 84

4.18 Sobel filter output quality evaluation. 85
4.19 Forwardk2j output quality evaluation. 86
4.20 Jmeint output quality *vs* energy trade-off 87
4.21 Sobel filter output quality *vs* energy trade-off 88
4.22 Forwardk2j output quality *vs* energy trade-off 89

5.1 widths (b) for a given value of energy reduction 96
5.2 Energy reduction (α) for a given width 98
5.3 Sobel filter energy evaluation with optimized units 100
5.4 Forwardk2j energy evaluation with optimized units 101

LIST OF TABLES

2.1 Fixed-point libraries 23
2.2 CPU-Axbench applications 32
2.3 RISC-V base opcode map [89] 39

3.1 RV32I base for full width instructions 46
3.2 RV32I base for reduced width instructions 48
3.3 Relative energy values of the full width instructions 52

4.1 Energy evaluation of stand-alone reduced width operators and their use in a RISC-V processor 78

5.1 Software and hardware architecture parameters 93

UNIVERSITE BRETAGNE LOIRE / MATHSTIC

UNIVERSITÉ DE RENNES 1

Title: Approximate computing for high energy-efficiency in IoT applications

Keywords : Reduced width units; energy reduction.

Abstract : Approximate computing explores methods to trade-off the quality of result and the computation costs, e.g. energy consumption. One of the proposed methods is to reduce the width of the computation units. To date, such units have been mostly evaluated separately, i.e. not evaluated in a complete application. In this thesis, we evaluate the global energy reduction *vs* quality of output trade-offs of applications. These applications are executed on a RISC-V processor extended with reduced width computation and memory units. In these units, only a number of most significant bits, configurable at runtime, is active. The results indicate in average that the energy can be reduced by up to 14% for an error $\leq 0.1\%$. Moreover we propose a generic energy model that indicates that both software parameters (e.g. fraction of *approximable* code) and hardware architecture ones (e.g. degree of approximation) impact the applications energy reduction.

Titre: Calcul approximatif à haute efficacité énergétique pour des applications de l'IoT

Mot clés : Unités à taille réduite; réduction d'énergie.

Resumé : Le calcul approximatif est l'une des solutions proposées pour trouver un compromis entre la qualité de résultat et les coûts de calcul, p.~ex. l'énergie consommée. L'une des méthodes proposées est la réduction de la taille des unités de calcul. Cependant, la plupart de ces unités sont évaluées séparément, c.-à-d. elles ne sont pas évaluées sur une application complète. Dans cette thèse, nous avons étudié le compromis entre la réduction d'énergie globale et la qualité de sortie des applications. Ces applications sont exécutées sur un processeur RISC-V étendu avec des unités à taille réduite pour le calcul et pour l'accès à la mémoire de données. Ces unités sont configurables au moment de l'exécution. Les résultats indiquent qu'en moyenne la consommation d'énergie peut être réduite jusqu'à 14% pour une erreur $\leq 0.1\%$. De plus, nous avons proposé un modèle d'énergie générique qui indique qu'à la fois les paramètres logiciels (p.~ex. la fraction de code *approximable*) et architecturaux (p.~ex. le degré d'approximation) ont un impact sur la réduction globale d'énergie des applications.