Universidade Estadual de Campinas
Instituto de Computação

Isaías Bittencourt Felzmann

Architectural Support for Approximate Computing

Suporte Arquitetural para Computação Aproximada

CAMPINAS
2023

Isaías Bittencourt Felzmann

Architectural Support for Approximate Computing

Suporte Arquitetural para Computação Aproximada

Tese apresentada ao Instituto de Computação da Universidade Estadual de Campinas como parte dos requisitos para a obtenção do título de Doutor em Ciência da Computação.

Thesis presented to the Institute of Computing of the University of Campinas in partial fulfillment of the requirements for the degree of Doctor in Computer Science.

Supervisor/Orientador: Prof. Dr. Lucas Francisco Wanner

Este exemplar corresponde à versão final da Tese defendida por Isaías Bittencourt Felzmann e orientada pelo Prof. Dr. Lucas Francisco Wanner.

CAMPINAS
2023

Ficha catalográfica
Universidade Estadual de Campinas
Biblioteca do Instituto de Matemática, Estatística e Computação Científica
Ana Regina Machado - CRB 8/5467

F349a

Felzmann, Isaías Bittencourt, 1992-
Architectural support for approximate computing / Isaías Bittencourt Felzmann. – Campinas, SP : [s.n.], 2023.

Orientador: Lucas Francisco Wanner.
Tese (doutorado) – Universidade Estadual de Campinas, Instituto de Computação.

1. Computação aproximada. 2. Computação consciente de energia. 3. Arquitetura de computador. I. Wanner, Lucas Francisco, 1981-. II. Universidade Estadual de Campinas. Instituto de Computação. III. Título.

Informações Complementares

Título em outro idioma: Suporte arquitetural para computação aproximada
Palavras-chave em inglês:
Approximate computing
Energy-aware computing
Computer architecture
Área de concentração: Ciência da Computação
Titulação: Doutor em Ciência da Computação
Banca examinadora:
Lucas Francisco Wanner [Orientador]
Jorge Castro-Godínez
Alfredo Goldman Vel Lejbman
Guido Costa Souza de Araújo
Sandro Rigo
Data de defesa: 04-08-2023
Programa de Pós-Graduação: Ciência da Computação

Identificação e informações acadêmicas do(a) aluno(a)
- ORCID do autor: https://orcid.org/0000-0003-3048-8310
- Currículo Lattes do autor: https://lattes.cnpq.br/3499341475094394

Universidade Estadual de Campinas
Instituto de Computação

Isaías Bittencourt Felzmann

Architectural Support for Approximate Computing

Suporte Arquitetural para Computação Aproximada

Banca Examinadora:

- Prof. Dr. Lucas Francisco Wanner
  IC/UNICAMP
- Prof. Dr. Jorge Castro-Godínez
  TEC/Costa Rica
- Prof. Dr. Alfredo Goldman vel Lejbman
  IME/USP
- Prof. Dr. Guido Costa Souza de Araújo
  IC/UNICAMP
- Prof. Dr. Sandro Rigo
  IC/UNICAMP

A ata da defesa, assinada pelos membros da Comissão Examinadora, consta no SIGA/Sistema de Fluxo de Dissertação/Tese e na Secretaria do Programa da Unidade.

Campinas, 04 de agosto de 2023

To Andressa.

Acknowledgements

This work was directly supported by the São Paulo Research Foundation (FAPESP) grant #2018/24177-0. Any opinions, findings and conclusions or recommendations expressed in this material are those of the author and do not necessarily reflect the views of the Foundation. This work received additional support from the National Council for Scientific and Technological Development – Brazil (CNPq), under grants 404261/2016-7, 438445/2018-0, and 402467/2021-3, and the Coordination for the Improvement of Higher Education Personnel – Brazil (CAPES), Finance Code 001.

This text contains significant text excerpts extracted, with permission, from Isaías Felzmann, João Fabrício Filho, and Lucas Wanner, *Risk-5: Controlled approximations for RISC-V*, IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems, November 2020 [46]. ©2020 IEEE.

This text contains significant text excerpts extracted, with permission, from Isaías Felzmann, João Fabrício Filho, Juliane Regina de Oliveira, and Lucas Wanner, *Special Session: How much quality is enough quality? A case for acceptability in approximate designs*, IEEE 39th International Conference on Computer Design (ICCD), October 2021 [45]. ©2021 IEEE.

In reference to IEEE copyrighted material which is used with permission in this thesis, the IEEE does not endorse any of University of Campinas's products or services. Internal or personal use of this material is permitted. If interested in reprinting/republishing IEEE copyrighted material for advertising or promotional purposes or for creating new collective works for resale or redistribution, please go to `http://www.ieee.org/publications_standards/publications/rights/rights_link.html` to learn how to obtain a License from RightsLink.

This text contains significant text excerpts extracted from Isaías Felzmann, João Fabrício Filho, Lucas Wanner, *AxPIKE: Instruction-level Injection and Evaluation of Approximate Computing*, Design, Automation & Test in Europe Conference & Exhibition, February 2021 [47]. ©2021 EDAA.

Resumo

A Computação Aproximada é uma metodologia que proporciona ganhos em eficiência energética ao relaxar requisitos de qualidade em aplicações resilientes. Várias técnicas de *hardware*, desenvolvidas sem vínculo com uma aplicação, têm potencial de proporcionar grandes benefícios em cenários favoráveis, mas a integração delas em uma arquitetura de propósito geral traz novos desafios para seu controle, tais como determinar, em tempo de execução, que regiões de aplicação se beneficiam de aproximações, que tipos de aproximações são essas, e até que ponto elas são vantajosas.

Nesta tese, apresentamos extensões para a arquitetura RISC-V que implementam mecanismos de controle para coordenar múltiplas técnicas de aproximação coexistentes no mesmo sistema. Através dessas extensões, as habilidades de um *hardware* de aproximação são expostas ao *software* por meio de registradores para identificação, estruturas de dados e *drivers* que descrevem a natureza e parâmetros de configuração para cada elemento do sistema aproximado. Isso permite que a pilha de *software* controle o que e quanto é aproximado em uma aplicação. As aproximações podem ser dinamicamente configuradas e combinadas em tempo de execução, ampliando os horizontes de exploração.

Para expor ao *software* os mecanismos de controle, nós também construímos uma interface em nível de *software* supervisório contendo uma camada de abstração e permitindo a coexistência de diferentes configurações de aproximação dentre as aplicações que compartilham o processador. Os elementos necessários para esse nível de controle foram implementados em dois níveis: um simulador em software e um protótipo sintetizado para FPGA, que possibilitaram uma demonstração da funcionalidade do sistema e estimativas de custo energético.

Nos nossos resultados, selecionamos aproximações para avaliação tanto no simulador como no protótipo em FPGA. Esses resultados destacam a necessidade de integração em nível de arquitetura de aproximações em *hardware* para melhor avaliação de como aplicações se comportam quando expostas a aproximação. Nesse sentido, esta tese propõe uma nova ferramenta que preenche a lacuna entre o *software* e *hardware* de aproximação, permitindo que desenvolvedores avaliem os benefícios e custos de técnicas de aproximação em um ambiente controlado e configurável.

Abstract

Approximate Computing offers enhanced energy efficiency by exploring quality relaxation on resilient applications. Application-agnostic hardware-level techniques can provide high benefits under certain scenarios, but their integration on a general-purpose architecture presents novel control challenges, such as determining, at runtime, what application regions benefit from approximation, which approximations are favorable, and by how much.

In this thesis, we present extensions to the RISC-V architecture that implement control mechanisms to orchestrate multiple coexisting approximation techniques within an architecture. Through these extensions, approximate hardware capabilities are exposed to software through identification registers, data structures, and drivers that describe the nature and configuration parameters for each approximate design. Approximations may be configured and combined at runtime, allowing for simplified design space exploration.

To allow high-level software to take control of the approximate state of the system, we also built a supervisor-level software interface that provides the hardware abstraction layer of the approximation and supports the coexistence of different levels of reliability within applications that share the processor. The underlying hardware needed to support the control of approximations was implemented in two levels, a software simulator and a prototype synthesized for FPGA, which allowed the functional demonstration of the system and energy estimation. In our experiments, we selected approximations for evaluation both in the simulator and in the FPGA prototype.

Our results highlight the need for architectural integration of hardware approximations in order to provide an accurate evaluation of how applications behave when subjected to approximations. To this end, this thesis proposes a novel hardware-software framework that bridges the gap between software and hardware approximations, allowing designers to easily evaluate energy-quality trade-offs of approximation techniques in a controlled and configurable environment.

List of Figures

List of Tables

Glossary

This is a non-exhaustive list of terms and acronyms used in this thesis. The provided definitions are not intended to be broad, but to reflect the meaning in the scope of this thesis, as a reference for the reader.

6T-SRAM	6T Static Random Access Memory. *The typical SRAM cell made of 6 transistors.*
ALU	Arithmetic and Logic Unit. *The part of the processor that computes arithmetic and logic functions.*
AMBA AXI4	Advanced Microcontroller Bus Architecture (AMBA) Advanced eXtensible Interface (AXI). *The fourth generation of a royalty-free on-chip communication bus protocol developed by ARM.*
ARCS	International Conference on Architecture of Computing Systems. *An international scientific event.*
ARM	Advanced RISC Machine. *Both a family of ISAs and the company that develops it.*
Arty A7-100T	*A development board based on a Xilinx Artix-7 FPGA.*
AXI	*See AMBA AXI4.*
AxPIKE	*Our in-house functional simulator for Approximate Computing.*
AxRAM	*Our in-house data access interface for approximate memories.*
CPU	Central Processing Unit. *The main processor in a computing system.*
CSR	Control and Status Registers. *The registers that control and store information about the CPU in a RISC-V architecture.*
DATE	Design, Automation and Test in Europe. *An international scientific event.*
DDR4	Double Data Rate SDRAM. *The fourth generation of the most common class of DRAM.*
DMA	Direct Memory Access. *A method that allows a peripheral device to communicate data directly with the main memory.*

DRAM	Dynamic Random Access Memory. *A type of random access memory typically used as main memory.*
DSP	Digital Signal Processing (slice). *An internal structure in an FPGA.*
DVFS	Dynamic Voltage-Frequency Scaling. *A mainstream energy-saving technique that adjusts voltage and frequency on demand.*
EDAA	European Design and Automation Association. *A scientific association.*
ELF	Executable and Linkable Format. *A common standard for executable files in Linux.*
ERAD-SP	Escola Regional de Alto Desempenho de São Paulo. *A regional academic event held in São Paulo.*
EvoApprox8b	*A library of approximate integer multipliers and adders.*
FFT	Fast Fourier Transform. *An algorithm that calculates the discrete form of the Fourier Transform.*
FGCS	Future Generation Computer Systems. *A scientific journal published by Elsevier.*
FPGA	Field Programmable Gate Array. *A hardware device designed to be reconfigured after manufacturing, representing different logic circuits.*
FPU	Floating-Point Unit. *The part of a processor that computes arithmetic with real numbers (Floating-Point representation).*
GPIO	General-Purpose Input/Output. *An I/O interface for general purposes.*
I/O	Input/Output. *One of the main functions of a computer.*
ICCD	International Conference on Computer Design. *An international scientific event.*
IEEE	Institute of Electrical and Electronics Engineers. *A professional association.*
IoT	Internet of Things. *A network of small connected devices and sensors.*
IP	Intelectual Property. *The proprietary designs by a given company.*
ISA	Instruction Set Architecture. *The interface between hardware and software implemented by a processor.*
JEDEC	Joint Electron Device Engineering Council. *An independent standardization body.*
JPEG	*An algorithm for lossy image compression.*
JTAG	*The industry standard interface for design verification.*
L1, L2	Level 1/2 Cache. *The first/second level of cache memory.*

Linux	*A reference to a given Operating System based on the Linux kernel.*
LLC	Last Level of Cache. *The last level of cache memory, from which misses are then fetched from the main memory.*
LUT	Lookup Table. *An internal structure in an FPGA.*
MMIO	Memory-mapped Input/Output. *A technique that maps communication with peripherals as memory-like accesses.*
OS	Operating System. *A collection of software that manages computer hardware resources and provides common services for other programs.*
RAM	Random Access Memory. *A memory that can be accessed at any order.*
RISC-V	*An open-standard ISA.*
Risk-5	*Our ISA extension to support approximation.*
Rocket Chip	*A processor implementation based on the RISC-V architecture.*
RTL	Register-transfer level. *A design abstraction that models digital circuits in terms of the flow of data.*
RV64g	*A subset of the RISC-V ISA.*
SBAC-PAD	International Symposium on Computer Architecture and High Performance Computer. *An international scientific event.*
SBESC	Brazilian Symposium on Computing Systems Engineering. *A national scientific event held in Brazil.*
SDRAM	Synchronous Dynamic Random Access Memory. *See DRAM.*
Sobel	*An algorithm to identify borders in an image.*
SSIM	Structural Similarity Index Metric. *A metric that computes the similarity between two images.*
SUSCOM	Sustainable Computing: Informatics and Systems. *A scientific journal published by Elsevier.*
TCAD	Transactions on Computer-Aided Design of Integrated Circuits and Circuits. *A journal published by IEEE.*
TLB	Translation Lookaside Buffer. *A part of the virtual memory translation system that caches page table entries.*
UART	Universal Asynchronous Receiver/Transmitter. *A protocol for asynchronous serial communication.*
USB	Universal Serial Bus. *An industry standard for serial communication.*
WSCAD	Simpósio em Sistemas Computacionais de Alto Desempenho. *A national scientific event held in Brazil.*
ZCU102	*A development board based on a Xilinx FPGA.*

Contents

Chapter 1

Introduction

An architecture for Approximate Computing

In modern computing systems, power dissipation and energy efficiency are important factors to take into consideration in the design process. It is not uncommon to find these limiting scalability, as performance requirements may not be sustainable in the long run [68,140]. Approximate Computing has emerged as a design methodology to answer the computing systems' ever-increasing need for energy efficiency. This methodology explores the exposition of intermediary processing steps to minor deviations that do not affect the final result in a significant way for many computing domains. When the applications are resilient to some accuracy degradation, allowing approximate results can potentially lead to significant energy savings. These led to multiple software- and hardware-level approximation techniques to explore this energy-quality trade-off [6, 83, 91].

These approximation techniques, by themselves, are just a different way to perform a given operation. They need to be applied in solving a computing problem, on a target scenario, to achieve their expected energy-quality trade-off. This requires that the approximations are integrated into an application. Since software-level approximations are designed to be part of an application, this integration is resolved at design time. Loop perforation [125], for example, modifies the implementation of loops to skip iterations to save computation time and energy and, therefore, is integrated within an application at design time. Hardware-level techniques, on the other hand, require an architectural interface to support the execution of a full application stack. An approximate arithmetic unit [96] would require a coexistent accurate counterpart to support the execution of non-resilient code segments, imposing overheads and adding control requirements. Existing hardware techniques, however, typically lack this level of architectural integration.

Approximation-hardware units are usually built as standalone units that introduce some level of imprecision at design time, or designed to offer some level of controllability, either directly or by adjusting some external parameter [6, 91]. In this thesis, these are refered as nonconfigurable, directly-configurable, and indirectly-configurable approximation units, respectively. Any of the approaches cause overhead at the architecture level, such as the need for replicated units to support non-resilient application segments, additional hardware to provide controllability or error recovery, or circuitry to allow operating parameters to be adjusted and to coexist. Approximation-hardware designs are, however,

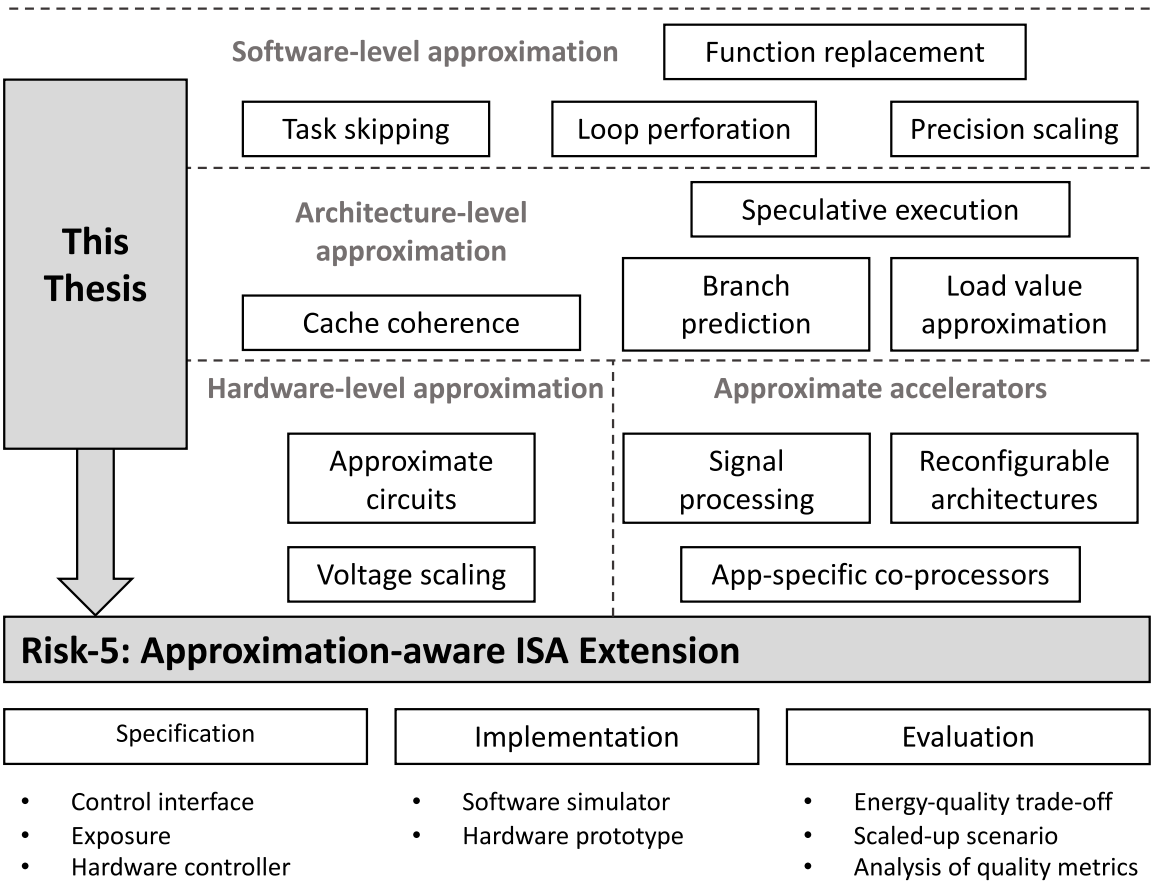


Figure 1.1: The role and contributions of this thesis. *We propose an approximation-aware ISA extension to integrate approximate hardware in the general-purpose scenario.*

usually demonstrated and evaluated in isolation, applied to specific application-level constructions, with limited or no discussion of the architecture-level integration with other hardware components in an actual processor.

Individual approximate multiplication units, for example, can report energy savings in the range of tens and up to 80%, in comparison to their accurate counterparts, with limited impact on expected error per operation [8, 77, 96]. These results, however, are oftentimes obtained in very limited scenarios based on mathematical modeling and software simulation of the multiplier alone, applied over synthetic workloads or case studies that profit from the multipliers at their best. Multipliers do not live alone in a general-purpose processor. They are just a part of the design, even if a significantly costly one [144]. Also, they may not be sized and optimized as demonstrated for their approximate designs and represent just a fraction of the instructions executed in a typical application. By taking these factors into consideration, we show that energy savings can be reduced to modest single-digit values at a significantly higher quality impact compared to other classes of approximation [46, 49]. Thus, the lack of architecture-level considerations when analyzing and demonstrating approximation hardware results can potentially leave energy savings figures overestimated.

Figure 1.1 shows where this thesis is inserted in the Approximate Computing stack and summarizes its contributions. We study approximation techniques and approximate

architectures to answer whether and how approximation-hardware modules can be applied to general-purpose computing. The architecture-level integration is determinant in this scenario, and this requires an adequate hardware-software interface to recognize, expose, and control the approximation capabilities: recognize what kind of approximation is available and how to use it, expose it to the software application or supervisor so that they can evaluate whether they can profit from it, and control where and when the approximation should be employed as determined by the software.

Previous work has shown the potential of integrating approximations into architecture design [25–27, 53, 63, 93, 99] and proposed generic interfaces to control approximations and account for execution resilience [28, 31, 48, 49, 81, 114, 134]. These integration alternatives are, however, usually applied only to a limited number of specific approximation scenarios, and the generic control is mostly focused on the software controller, not reflecting a wide range of available approximation techniques. The differences in the behavior of applications when subjected to approximations [25, 44] require a control interface to be designed using a more approximation-centered approach, while still leaving enough links with the software for reliable execution. Existing integrated hardware-software alternative interfaces, however, are not focused on a general-purpose scenario, have limited compatibility with existing approximation techniques, underestimate control overhead, or are applied directly to the application, in a scenario not compliant with modern multiprocess and multi-application computing controlled by a supervisor system [6, 29]. Thus, how to orchestrate multiple coexisting hardware approximation techniques in a general-purpose processor is a question that has not yet been fully answered [83]. Since different applications have different levels of compatibility with different approximations, managing coexisting techniques is essential to apply Approximate Computing into compatible applications in a general-purpose scenario.

This thesis advances into the ability to provide approximation control by proposing and demonstrating Risk-5 [46], an ISA extension that provides a hardware-software interface, based on RISC-V, to coordinate multiple hardware-level approximation techniques within an architecture. In Risk-5, each approximation technique is integrated within the processor as a dedicated controllable module that can be activated or deactivated at execution time as instructed by a set of standardized control registers. The main contributions of this thesis are:

- the architecture-level specification of the Risk-5 extension and interface, discussing how it manages and integrates approximation techniques,
- the definition of a hardware controller that allows applications and operating systems to manage different degrees of approximation,
- the definition of a software interface for the applications to communicate with the operating system and underlying hardware to recognize and configure the approximation capabilities of the system, and
- a scaled-up exploration and demonstration of approximation techniques and target applications, identifying computing scenarios that can benefit from hardware approximation and how they behave when scaled to larger requirements.

Furthermore, to demonstrate the capabilities of the proposed control interface and the applicability of hardware approximation in a general-purpose processor, this thesis incorporates two different levels of implementation as technical contributions. At the simulation level, we developed AxPIKE, an instruction-level simulator that introduces versatility in representing approximate systems allowing designers to quickly model and evaluate their approximation concepts. Based on the demonstration performed on AxPIKE, we show how our control interface can allow the reliable and energy-efficient execution of applications on a general-purpose system employing approximation in two architectural resources: the memory hierarchy [37, 38] and the multiplication operation [8, 77, 96].

At the hardware level, we implemented the approximation controller and modules in Hardware Description Language, integrated with the RISC-V Rocket Chip Generator. The design was synthesized as a standalone multi-core system in an FPGA and demonstrates how the control operations behave in a multi-application scenario. The prototype runs the approximations in a bare-metal multithreaded framework, limited only by the FPGA size and connectivity, that communicates data with an auxiliary computer using a standard serial interface and controls each test application. It also exposes a standard debugging interface that allows the designer to observe and interfere with the execution, which includes loading different applications using off-the-shelf debugging tools.

Based on both the software-level simulator and the hardware-level prototype, experiments were conducted to demonstrate and validate the designs and their applicability, considering scenarios where approximations may be employed to improve the efficiency of application execution. In the more versatile simulation environment, we selected a subset of six test applications to be evaluated in the data approximation scenario, where errors are introduced in the access to a DRAM memory, considering three different reliability scenarios. The test scenarios were then augmented to include the approximate multiplication operator in the processor. Our results show how applications can potentially save 22.4% energy for a 90% quality requirement, on average, and up to 22.8% energy in memory and 1.79% in CPU execution, while still maintaining visual quality in a more specific Sobel Edge Detection scenario. More prominently, the results highlight the impact of approximation control on the energy benefits provided by approximations, reinforcing the need for architectural integration to properly evaluate approximation-hardware units.

The simulation environment is more flexible in representing different applications and scenarios, including support for Linux-based libraries and redirection of environment calls to the host system. However, the simulation time and memory usage scale up as more execution statistics are needed, especially for detailed energy accounting and simulation of multiple levels of cache. Moreover, the instruction-level functional simulation cannot fully represent the behavior of executing the approximations in a real scenario, as the simulator lacks architectural characteristics such as a real cache and branch prediction. The hardware prototype, on the other hand, is faster, more scalable, and represents precisely the behavior of the approximation when implemented at the hardware level, but requires significantly more effort to adapt the applications. Thus, to validate the simulation results against the hardware prototype, we designed experiments that meet the requirements of both environments, in a single Image-Processing scenario executing over a larger amount of data, showing how the DRAM and multiplication approximations be-

have at JPEG compression of approximately ten thousand images from the Imagenette dataset [56]. The results show that quality evaluation is compatible between the two environments, as evidence that the approximations can be applicable at these scenarios, and that this applicability of the approximation techniques can scale to larger scenarios. Moreover, the results highlight the need to design the approximation themselves for architecture integration from the beginning, since the integration depends on decisions made early in the design process.

Throughout every experiment conducted, quality evaluation was chosen as the primary method of deciding what results are "good enough" approximations of the expected result [76] and, therefore, would be an acceptable application execution instance. Quality metrics were chosen per application, considering the computing domain and expected results, in alignment with the evaluation found in the relevant literature. However, the process of analyzing the quality results of the application led to the observation that the value of the quality metric would not always correlate with quality as perceived by a human observer. We put idea to test by reinterpreting the meaning of what is an acceptable approximation as not how high is the quality of the results, but as how useful the results are for a second processing step. The results show that designing approximate systems that rely on this acceptability concept can produce up to 20% more valid results than the conservative quality thresholds commonly adopted in the literature, allowing for higher error rates and, consequently, lower energy costs. This points to a direction where quality evaluation may be replaced by a more useful acceptability criteria, as we further discuss in Chapter 7, and represents a complementary contribution of this thesis.

The main contributions in this thesis, the technical implementation, and the analysis of quality and acceptability metrics represent alternatives to the architecture-level integration of Approximate Computing in general-purpose systems. The control interface was essential in determining the impact of approximating hardware modules both in terms of quality and energy, and architecture-level design decisions also influence the applicability of approximations. General-purpose systems need to be built for interoperability, and the support proposed in this thesis enables a level of cooperation between the approximations and the system as a whole. Only after enabling this cooperation, the approximate computing can be reliably employed in general scenarios, as we show in this thesis. The selected approximations scale to larger scenarios and show potential to provide significant energy savings, even though the effects are amortized by the architectural integration.

The results obtained by the work in this thesis resulted in publications as listed in Table 1.1. The main thesis publications specify the ISA extension [46], present and demonstrate the software-level simulator [47], and analyze the correlation between quality metrics and perceived acceptability [45]. These were based on other previously or concurrently published results. In ADeLe [48, 49], we present a descriptive language that describes the behavior of approximations, at the architecture level, as software models that can replace existing operators in a functional CPU simulator. The language enabled further studies on the impact of injected approximations in memory and internal registers as seen from the architecture level [39, 40, 44]. Understanding the error profiles resulted in the definition of an implicit memory access interface that divides a memory array into approximate and accurate regions, protecting non-resilient data from errors [37, 38]. Aided

Table 1.1: List of authored publications related to this thesis

Target	Title/Co-authors	Ref.
Main thesis publications		
TCAD 2020	Risk-5: Controlled approximations for RISC-V Isaías Felzmann, João Fabrício Filho, Lucas Wanner	[46]
DATE 2021	AxPIKE: Instruction-level Injection and Evaluation of Approximate Computing Isaías Felzmann, João Fabrício Filho, Lucas Wanner	[47]
ICCD 2021	How much quality is enough quality? A case for acceptability in approximate designs Isaías Felzmann, João Fabrício Filho, Juliane Regina de Oliveira, Lucas Wanner	[45]
Supporting and background publications		
SBAC-PAD 2018	ADeLe: Rapid Architectural Simulation for Approximate Hardware Isaías Felzmann, Matheus Susin, Liana Duenha, Rodolfo Azevedo, Lucas Wanner	[48]
WSCAD 2018	(In Portuguese) Impact of Memory Approximation on Energy Efficiency Isaías Felzmann, João Fabrício Filho, Rodolfo Azevedo, Lucas Wanner	[44]
ERAD-SP 2019	(In Portuguese) *Tratamento de Ponteiros Incorretos armazenados em Memórias Aproximadas* João Fabrício Filho, Isaías Felzmann, Lucas Wanner	[39]
SBESC 2019	A Resilient Interface for Approximate Data Access João Fabrício Filho, Isaías Felzmann, Rodolfo Azevedo, Lucas Wanner	[37]
FGCS 2020	ADeLe: A description language for approximate hardware Isaías Felzmann, Matheus Susin, Liana Duenha, Rodolfo Azevedo, Lucas Wanner	[49]
ERAD-SP 2020	(In Portuguese) *Sensibilidade a erros em aplicações na arquitetura RISC-V* João Fabrício Filho, Isaías Felzmann, Lucas Wanner	[40]
FGCS 2020	AxRAM: A lightweight implicit interface for approximate data access João Fabrício Filho, Isaías Felzmann, Rodolfo Azevedo, Lucas Wanner	[38]
Derived publications		
ARCS 2021	Transparent Resilience for Approximate DRAM João Fabrício Filho, Isaías Felzmann, Lucas Wanner	[41]
SUSCOM 2022	SmartApprox: Learning-based Configuration of Approximate Memories for Energy-efficient Execution João Fabrício Filho, Isaías Felzmann, Lucas Wanner	[43]
SBAC-PAD 2022	Approximate Memory with Protected Static Allocation João Fabrício Filho, Isaías Felzmann, Lucas Wanner	[42]

by the architecture-level integration developed in this thesis, the approximate memory models were later enhanced with accounting for the cache impact [41], considering the application memory layout [42], and automatic inference of approximate level parameters [43]. These show the potential of architectural integration to show the impact of approximations in general-purpose scenarios.

This thesis is organized as follows. Chapters 2 and 3 present the relevant literature in the field, containing the main background aspects for understanding Approximate Computing and hardware modules, as well as a review of the related work. Chapter 4 contains the core proposal of this work, discussing the main contributions and how they relate to the specification for architecture integration. Chapter 5 describes the implementation of the two technical contributions in this project, the software simulator and the FPGA prototype. Chapter 6 details the experiments conducted to evaluate the proposed designs and discusses the results. Chapter 7 makes a case for the use of acceptability metrics disjoint from quality-accuracy evaluation in the design of Approximate Computing. The thesis is concluded with the main conclusions and future directions in Chapter 8.

Chapter 2

Background

Approximate Computing and approximation techniques

The evolution of Computing, as we know it, has been made upon a research effort focused on improving the performance of hardware and software within an energy budget. The improvements in both performance and energy have been powered by miniaturization [14], leading to an exponential growth in the number of transistors within a single chip and little impact on power and energy, which has been known even outside of the computing world as Moore's law. This level of profitable and low-overhead miniaturization, however, has ended with the end of Dennard scaling [35, 68].

For years since scaling alone ceased to be enough to sustain the rapid evolution of Computing, research effort has been mitigating the challenges associated with post-Dennard, deep-submicron miniaturization. For example, increased leakage power caused by down-scaled voltages forbids chips to be used at full capacity for extended periods of time, leaving them partially dimmed or even off, creating the phenomenon called "Dark Silicon" [35]. Despite being a waste of resources, leaving silicon dark does not necessarily improve efficiency, as the increased distance the data needs to travel within the chip affects both performance and energy [62, 140].

The challenge to design high-performance and energy-efficient computers led to advances at various levels in the design stack, such as longer-lasting batteries, application-specific processor designs, hardware accelerators, cloud offloading, more efficient algorithms, and lower overhead communication [29]. In common, and as with any other design choice in engineering, these trade some aspect for improved performance or energy efficiency, such as weight, size, portability, programmability, or cost. In particular, one aspect that can be included in this trade is *quality*, defined as the accuracy of the final application results [29, 76]. In many computing domains, fully accurate results are not needed or even not expected, and thus resources invested in producing results at higher-than-needed quality could be redirected to other useful computation. In Edge devices, for example, users expect low-latency, *good-enough* quality, and *long enough* battery life [108].

Despite the uncertainty in defining what is *"good enough"*, Approximate Computing has emerged as a promising solution to explore how to achieve less-than-perfect quality and trade it for energy. Approximate Computing techniques can achieve energy savings and improve efficiency both at the software and hardware levels [6, 83, 91]. Software-level

approximation techniques are changes performed in the software code that reduce the quality of results by exploring some inherent resilience of the target application. They include loop perforation, in which some iterations of a long loop are skipped [87, 89, 90, 125], task skipping, in which parallel computation units are selectively dropped [60, 117], and precision scaling, such as lowering the precision of floating-point computations or converting values to fixed-point representations [1, 55, 78, 88, 120]. Even though some of these transformations may be automatically performed by dedicated compilation tools [15, 32, 119, 121], the final result is still a dedicated implementation of the given application for a specific approximation. Thus, these techniques are, by definition, integrated within the application at design time and can run and communicate with the underlying hardware using the usual means that any other not approximated software would use.

Hardware-level approximation techniques, on the other hand, should not be intimately linked to a single application or execution scenario to be applicable to general-purpose computing. Different approximation techniques cause different quality and energy impacts on an application, and different applications behave differently under the same approximation [25–27, 44]. Thus, there is no generic approach to offer hardware approximation in a general-purpose computing scenario. Approximations need to be tweaked to allow reliable execution, such as isolating critical or nonresilient application regions, scaling data precision levels, or even selecting individual arithmetic operations that allow imprecision in their results. In other words, approximations need to be configured for each scenario, preferably configured at runtime by the application itself, and this raises the need for a hardware-software interface to support approximations.

2.1 Configurability of hardware-level approximations

Hardware-level approximation techniques require different levels of configurability according to their design and the source of approximation. If the logic is fully optimized to offer approximated results, it cannot produce error-free results to reliably execute nonresilient software. Alternatively, the design may offer a configuration knob that allows the approximation hardware to be defined in a reliable execution status. These different options are translated into different requirements from the point-of-view of the controller.

2.1.1 Non-configurable approximation techniques

One approach to developing approximation hardware is to design and fully optimize a logic circuit to offer a result that shows some deviation from an accurate result for the same operation deterministically. This is the case of, for example, slicing the carry chain of a ripple-carry adder [16, 50, 106], replacing an adder's basic building blocks with a different dedicated logic function [16, 141], simplifying the truth table of low-order multipliers [77], replacing multiplication with less costly alternatives [8, 106], implementing neural-network accelerators to predict the outputs of complex software functions [36, 54, 95], or other general methods to evolve approximate logic circuits [79, 96, 123, 133, 145]. The circuits that follow this design approach share two characteristics that are especially relevant for a control interface. First, since they are optimized to provide an approximate result, they

are typically smaller, less complex, faster, and more energy-efficient than the accurate version of a circuit to perform the same operation. Second, once implemented and deployed, they can only produce the same results for the same inputs, which can be known in advance whether by design modeling or experimentally sampling the operations. In other words, what this second aspect means is that they cannot be configured to produce a more (or less) approximated result and, by the same means, cannot be configured to reliably produce accurate results for critical computation.

If a hardware operator cannot produce reliable results, it also cannot be employed for general-purpose computation without additional supporting hardware. Thus, any architecture that implements such a non-configurable operator needs to also provide additional means for accurate computation. In the best-case scenario, error-free execution can be achieved enhancing the non-configurable unit with an error-correction circuit [19, 77]. In the worst-case scenario, a fully optimized accurate design to perform the same operation needs to coexist in the system. Replicating or adding hardware modules in the system increase the area cost, and this overhead, if not considered, can void any energy benefits achieved by the approximate design. Thus, the additional hardware needs to be dynamically deactivated using gating techniques. Traditional clock- and power-gating techniques, however, introduce themselves a trade-off between latency and residual energy cost, which is one of the main challenges in controlling non-configurable units.

2.1.2 Directly-configurable approximation techniques

Instead of fully optimizing a hardware design to offer approximate operation, a module can be generalized to expose a configuration knob that directly scales the precision of the operation. These are still deterministic designs since the output of the operation can be modeled in advance taken as input the selected precision. This is the case of adders in which the size of the carry chain is selectable [61, 70, 71], multipliers that can use multiple-precision adders to aggregate partial products [106], precision-scalable Floating-Point Units [17, 72, 86, 102], or scaling the precision of data values by truncating lower magnitude bits [94, 115, 142]. Generalizing the components and introducing the configuration knob, however, tends to create components that are less energy efficient in the error-free mode of operation when compared to fully optimized error-free counterparts [49]. Thus, configurable approximation units may not eliminate the need for replicated and optimized units as in non-configurable designs.

Furthermore, configurable approximation units introduce another design challenge on the controller, which is the configuration knob itself. While non-configurable units support at most two states (off/accurate and on/approximate), configurable ones support many, and this needs to be taken into account by hardware and software controllers. More states increase the amount of information to be transferred from the controller to the approximation unit on state changes, requiring wider communication buses, imposing additional hardware overhead, and increasing the complexity of software drivers. Moreover, the different levels of configurability increase the number of degrees of freedom for design space exploration, complicating the process of finding optimal operating points of applications with different quality requirements and resilience characteristics.

2.1.3 Indirectly-configurable approximation techniques

Another approach for introducing approximate behavior into hardware modules is to scale operating parameters below the nominal guardbands. Nominal operating parameters are usually overestimated to ensure error-free reliable execution and to account for hardware variability. Further scaling of parameters may trigger non-deterministic execution errors. In exchange, they offer high potential energy benefits. The most popular parameter of choice is the supply voltage, in a technique commonly referred to as Voltage Overscaling [24,30,37,38,92,98,103,112]. Lower voltages directly lead to circuit timing errors when the frequency is not adjusted accordingly, and even lower (Near-Threshold) voltages may lead to transistor switching failures [25, 30]. Other similar approaches are extending the refresh rate or artificially decreasing the latency of DRAMs [23, 75, 82, 84, 113] or scaling the retention time of STT-MRAMs [111, 115].

Because of the extended level of configurability, indirectly-configurable approximation techniques also share the challenges that lead to overhead in the hardware and software controllers. In indirectly-configurable hardware, the scaling knob is outside the hardware unit, and thus the error-free point of operation should not be less efficient than a non-scalable version of the same unit. However, high-performance, fully optimized hardware may not be compatible with this type of configurability. For example, a Ripple-Carry Adder is slower and less efficient than a Kogge-Stone Adder for error-free execution, but it fails gracefully at each step of voltage overscaling instead of abruptly [99]. Thus, the Kogge-Stone Adder is more appropriate for error-free execution, while the Ripple-Carry Adder performs better under approximation, and using both in the same design could lead to the best energy efficiency overall. However, joining both modules adds an aspect of non-configurability to the design, increasing the complexity of the controller.

Furthermore, scaling operating parameters introduces circuit-level area overhead to the design. To scale voltage, for example, additional voltage domains need to be present in the project, including additional layers of insulation and level shifters. The scaling itself produces noise that may affect nearby circuits. Also, scaling operating parameters is not instantaneous, and this delay needs to be accounted for in the latency of hardware and software controllers. Finally, these non-deterministic approximations require more complex design space exploration to search for the proper operating points of the applications, in addition to recovery or resilience mechanisms to account for the probabilistic outcome of the execution [38].

Chapter 3

Related Work

Interfacing hardware approximations and software

This project proposes the definition of an architecture-level interface and underlying hardware support to allow general-purpose applications to take advantage of the Approximate Computing benefits, towards a general-purpose approximation-capable CPU. The concept of scaling precision, at the architecture level, to introduce some kind of hardware-level approximation in exchange for efficiency has been introduced by many architectures. The most straightforward example is the standardized adoption of multiple floating-point instructions that control different precision levels in a hardware Floating-Point Unit. The choice between at least single- and double-precision IEEE 754 floats is available in most modern architectures. The translation of this concept to Approximate Computing scenarios, however, brings additional challenges. Approximation techniques increase imprecision across common guardbands, possibly triggering unrestrained errors and unreliable execution if not properly controlled. This more complicated control requirement imposes overhead, and thus the ratio between specific approximation capabilities and their control needs to be well tweaked to amortize the overhead and to allow efficiency improvement. Moreover, the diversity of possible approximation techniques and their applicability to distinct and disjoint execution scenarios requires the software – or the programmer – to have deep knowledge of the hardware approximations. Thus, a general Hardware-Software interface for approximation control is the subject of existing research efforts. In this Chapter, we describe existing alternatives in face of the mentioned design challenges in comparison with our proposed interface.

Approximation techniques have been integrated into *ad hoc*, specific designs. Dedicated hardware for video post-processing [33], biosignal analysis [13, 110], and more general signal processing [64, 118, 139] demonstrate the potential of Approximate Computing for these domains, expanding results obtained for single hardware operators to full application flows. Some of these specific architectural alternatives allow reconfigurability [3–5, 124, 129], potentially increasing applicability to different domains. Although dedicated architectures can be redesigned as accelerators, acting in conjunction with software-level approximation interfaces [18, 36], they are accessories of a full general-purpose programmable processor. One of the main goals of our project is to allow approximate execution in a general-purpose processor.

Table 3.1: Qualitative classification of related work and comparison with this thesis

Year	Work	Purpose	Approximation				Hardware-Software interface	Software controllability		Impl.	
			Coverage	NC[1]	DC[2]	IC[3]		Granularity	Level	SW[4]	HW[5]
2000	**Example:** Multiple-precision FPU	General	Single		✓		Dedicated instructions	Instruction	Application	✓	✓
Approximation-only related projects											
2009	Significance Driven Computation [93]	Single	Single			✓	—	—	—		✓
2010	HERQULES [63]	Single	Single			✓	—	—	—		✓
2010	Scalable Stochastic Processors [99]	Specific	Single	✓		✓	—	—	—		✓
2010	Scalable Effort Hardware Design [25–27]	Specific	Multiple		✓	✓	—	—	—		✓
2017	ACR [53]	General	Single		✓					✓	[6]
Interface-only related projects											
2007	ERSA [28, 81]	General	—	—	—	—	Scheduling	Thread	Supervisor		✓
2010	Relax [31]	General	—	—	—	—	Transition instructions	Block	Application	✓	
2013	Rahimi *et al.* environment [114]	General	—	—	—	—	MMIO registers	Statement	Application	✓	✓
2018	ADeLe [48, 49]	General	—	—	—	—	Transition instructions	State	Both[7]	✓	
2019	Crash Skipping [134]	General	—	—	—	—	Scheduling	Process	Supervisor	✓	[6]
Approximation and interface related projects											
2011	EnerJ [122]	General	Generic		✓	✓	Dedicated instructions	Instruction	Application	✓	
2012	Truffle [34]	General	Single			✓	Dedicated instructions	Instruction	Application	✓	
2013	Quora [132]	Specific	Multiple		✓	✓	Dedicated instructions	Instruction	Application		✓
2013	TABSH [2]	Single	Single	✓			Dedicated instructions	Instruction	Application	✓	
2014	Accordion [65, 69]	Specific	Single		✓	✓	Scheduling	Thread	Application	✓	
2015	Tagliavini *et al.* memory [126, 127]	General	Single			✓	HAL (unspecified)	Process	Application	✓	
2017	Parasyris *et al.* Execution on Unreliable Hardware [107]	General	Single			✓	DVFS (unspecified)	Task	Application	✓	
2017	ProACt [22]	General	Single	✓	✓		Transition instructions	Block	Both[7]		✓
2018	Ndour *et al.* approximate units [100, 101]	General	Multiple	✓	✓		Dedicated instructions	Instruction	Application	✓	
2019	AxRAM [37, 38]	General	Single			✓	MMIO registers	Process	Both[7]	✓	
2019	Nongpoh *et al.* approximate cache coherence [104]	General	Single		✓		Dedicated instructions	Instruction	Application	✓	
2019	Nongpoh *et al.* approximate speculative execution [105]	General	Multiple		✓		Dedicated instructions	Instruction	Application	✓	
2020	Taştan *et al.* approximate CPU for IoT [128]	Specific	Multiple	✓	✓		Dedicated instructions	Instruction	Application		✓
2021	HIART-MCS [58, 59]	General	Single		✓		MMIO registers	Task	Supervisor		✓
2021	VSX [74]	General	Single		✓		Dedicated instructions	Block	Application	✓	[6]
2021	SIA [97]	Specific	Single		✓		Dedicated instructions	Instruction	Application	✓	
2022	AxE [12]	General	Single	✓			Dedicated instructions	Instruction	Application		✓
2023	**This thesis**	**General**	**Generic**	✓	✓	✓	**Control registers**	**State**	**Both**	✓	✓

[1] Non-configurable [2] Directly-configurable [3] Indirectly-configurable [4] Software-level simulation [5] Hardware-level prototype
[6] Hardware implementation without behavioral results [7] No explicit privilege-level protection

Table 3.1 establishes a qualitative classification and compares related projects that are compatible with this programmable model. For each project, we evaluate the purpose of the design, its compatibility with approximation techniques, the proposed hardware-software interface for control, the behavior of the control mechanism itself, and how its demonstration was implemented.

- The **purpose** of a design may be (a) *general*, in which it applies to a wide range of applications or application domains, including automatic adaptation mechanisms to different applications or a general methodology for it, (b) *single*, when a single application was manually tweaked for the demonstration, or (c) *specific*, in which the design is demonstrated for multiple applications but without enough evidence of its generality.

- The *coverage* of **approximation techniques** may (a) include a *single ad hoc* model, (b) be demonstrated to *multiple* approximations, or (c) provide evidence of being *generic*. Also, we classify the levels of configurability of the compatible techniques according to the classification in Section 2.1: (a) *non-configurable*, in which accurate counterparts of the modules are required to support critical computation, (b) *directly-configurable*, in which a quality scaling knob is designed and included in the implementation, and (c) *indirectly-configurable*, in which an existing operating parameter is used to cause side effects on quality.

- The proposals of **hardware-software interfaces** include (a) *dedicated instructions* for accurate and approximate operations (e.g., an accurate `mul` and an approximate `mul.approx`), (b) *transition instructions* that activate or deactivate the approximate mode of operation for one or more modules in the system, (c) *memory-mapped input/output (MMIO) registers* or *control registers* that store the current mode of operation, (d) *scheduling* mechanisms that define the status of approximations for given processes or threads, and (e) other unspecified mechanisms based on existing general structures.

- About the **software-level controllability**, we detail the *granularity* at which changes in the mode of operation are assumed – from finer to coarser, (a) a single *instruction*, (b) a software-level *statement*, (c) a *block* of instructions, (d) a software-level *task*, (e) a *thread* within a process, (f) a *process*, or (g) a system-level *state*. Also, we identify the software *entity* responsible for control as the user-level *application*, a *supervisor* system, or *both*.

- The level of **implementation** that the design was demonstrated for, if (a) some software-level simulation was used, such as directly modifying an application, leveraging an architecture-level or a full-system simulator, or building a Matlab model, or (b) a hardware-level prototype, whether in an FPGA or synthesized for ASIC. Some designs report area and power results after hardware synthesis, implying the existence of a hardware-level implementation, but base their behavioral demonstration solely on software simulation, without any results from the actual hardware – these are marked with a footnote.

As an example, we include in the proposed classification the interface for a multiple-precision Floating-Point Unit. It has a general purpose since it is compatible with any software that uses floating-point values and is seamlessly introduced by compilers from higher-level software data types. It introduces precision scaling of a particular data value representation. Thus, it handles a single directly-configurable approximation technique. Hardware and software communicate using dedicated instructions for each operation and precision level. The hardware operations that use the determined precision are only the ones triggered by the executed instruction, so it acts at an instruction granularity. The dedicated instructions are injected directly into the application binary and require recompilation to change the precision level, which means the application itself is in control of the approximations. Finally, implementations exist in both hardware and software.

Also, we split the related projects into three major categories. The first (Section 3.1) holds experimental results that demonstrate the impact of approximation techniques on the software. These are baseline projects that do not propose nor evaluate a full approximate architecture. Although they do not share the general controllability objectives we have in our project, they present results on how applications react after approximation injection and help to understand the requirements of an architecture-level interface between hardware approximations and software. Since they do not define the interface itself, the classification of the interface and its controllability characteristics are out-of-scope. Understanding the requirements leads to the second category (Section 3.2), which contains projects that detail only the control interface. These propose to be agnostic of approximation technique but their demonstration uses software instrumentation or error injection that does not necessarily represent the outcome of generic approximations. This approach also tends to ignore much of the overhead caused by requirements of different approximation techniques, such as the need for replicated units and power gating. Then, details about approximation techniques are out-of-scope. The third category (Section 3.3), in which our project is inserted, contains projects that both propose the control interface and demonstrate its interaction with models of approximation techniques. Finally, Section 3.4 presents and justifies the concepts and design choices behind our proposal, pointing out how they compare with the alternative solutions.

3.1 Approximation-only related projects

Understanding the impact on applications of introducing approximations into hardware components is a fundamental step towards adopting Approximate Computing in a general-purpose scenario. Also, introducing a feedback loop from the software design flow to the architecture definition demonstrates the specificity of each application when it comes to approximation and, consequently, the need for control in a programmable scenario. Mohapatra *et al.* [93] proposed *Significance Driven Computation*, a methodology to identify less significant computations at the algorithmic level to voltage-overscale processing elements in the underlying architecture. The method was demonstrated for a motion estimator in video encoders, from which they evaluated the energy-quality trade-off after injecting approximations. In HERQULES [63], voltage overscaling in a digital camera and a

wireless transceiver was also employed to increase the design space for image acquisition software. The design was optimized to ensure low energy cost for a given quality requirement. He *et al.* [53] exploited locality to introduce approximation in computation reuse for error-tolerant applications. Their Approximate Computation Reuse (ACR) method relaxes the similarity criteria in the input of computation tasks when deciding whether previous results for the same computation pattern can be extrapolated to the current operation. If results not exactly for the same scenario are reused, due to the relaxed criteria, then approximation was introduced. These projects use *ad hoc* evaluation and do not expose a control interface to the applications, but provide evidence that applications may support execution under approximation. In this thesis, we expand the evaluation of how applications behave under approximation techniques to general-purpose scenarios.

Narayanan *et al.* [99] introduced the concept of *Scalable Stochastic Processors*, a system architecture that includes "voltage scaling-friendly" blocks. These are functional unit designs that present predictable and controllable error patterns when subjected to voltage scaling. The authors proposed three alternative organizations. First, to replace the existing functional units with friendly units, introducing error scalability at the cost of compromising the best power-performance trade-off. Second, to include friendly units in addition to the original ones and switching them on and off, adding a non-configurable aspect to the design. Third, to design different cores with distinct structures. The concept was demonstrated by evaluating the impact of scaling a non-friendly, but highly optimized, Kogge-Stone adder in comparison with a friendly, but much slower, ripple-carry adder in a mobile video encoding application. This conceptual evaluation, however, considered energy and quality only specific to the adders, not detailing their architectural integration. These results highlight that a configurable approximation unit, when operating in the higher accuracy mode, is not necessarily equivalent to an optimized accurate unit. Thus, besides the overhead caused by any configuration delay, a configuration interface needs to take into account that the module itself may reduce the efficiency and that additional levels of controllability may be needed to overcome this limitation. Our proposal for a control interface is designed to mitigate the overheads of power and clock gating in such scenarios.

Chippa *et al.* [25–27] presented *Scalable Effort Hardware Design*, one of the first projects to introduce the term Approximate Computing at the architectural level. The project implements a specific-purpose Recognition and Mining processor that aggregates approximation scalability at algorithm, architecture, and circuit levels. At the algorithm level, the design flow skips computation entities. At the architecture level, it scales the precision of less significant variables identified in the algorithm. At the circuit level, it overscales the voltage of processing elements that compute less significant software structures. Demonstration results based on *ad hoc* software instrumentation showed significant energy savings for Support Vector Machine (SVM) Classification and K-Means clustering. These results show that integrating different techniques within the same architectural implementation increases the potential benefits. Also, the distinct patterns of the energy-quality trade-off for different applications and input datasets highlight that runtime scalability, as we propose in this thesis, is required to tweak the execution to the expected quality requirement.

3.2 Interface-only related projects

Studying the impact of hardware-level approximations on applications suggest that their behavior varies according to the source of approximation, the application itself, and the operands [25, 44]. The distribution of critical and non-critical data and the nature of computation determine how deviations from accurate results in specific operations affect the final quality of results. Thus, the ability to control the approximations at execution time is necessary to favor energy efficiency while still maintaining enough quality.

In ERSA [28, 81], a scheduling mechanism was proposed to allow this control in a many-core architecture. The idea is that applications that are compatible with massive parallelism, such as Recognition, Mining, and Synthesis (RMS), are distributed into a set of reliable and unreliable computing cores. The reliable cores execute the main threads, responsible for the control and critical execution, while the unreliable cores deal with highly-resilient code in worker threads. To represent unreliable computation, ERSA's evaluation injects probabilistic errors into architecturally visible registers. Before execution under ERSA, applications are subjected to an analysis and "enhancement" methodology at design and compilation time. At this stage, resilient and non-resilient threads are chosen and application-level modifications, such as sanity checks, convergence control, and function inline expansion, are employed to improve error resilience. Under ERSA, although applications support substantially high error rates while still providing bounded quality degradation, this comes at the cost of time overhead that may amortize the energy benefits of approximation. Also, the error injection methodology used in the evaluation is artificial and not based on any real approximation models. Thus, the evaluation does not allow energy estimations, which would require the error injection to be completely redesigned. Our proposal is built around a subset of representative approximation techniques, allowing for an actual implementation. The design was demonstrated using a dedicated software simulator and evaluated for energy off-the-shelf specialized tools, and a hardware-level prototype accounts for scaling-up the results.

Relax [31] presented a framework to allow software-level recovery of hardware errors. A software representation and an ISA extension mimic the behavior of exception control try/catch blocks, in which error-prone code (try) may trigger software recovery (catch) if hardware-level errors are detected during execution. For error-resilient code, the recovery routine may be "relaxed", or skipped, introducing approximation control. The approach, however, initially assumes that hardware error patterns can be bounded to specific high-level "try" blocks, which itself requires a preliminary level of controllability of approximations. Rahimi *et al.* [114] presented an OpenMP-based extension that guides computation offloading to processing units subjected to variability. Software-level pragmas are translated into special instructions sent to memory-mapped registers that trigger the approximate execution of software statements. In this thesis, an ISA extension includes control registers that store the current configuration of approximations in the system. The software can handle these registers using the instructions defined for this purpose in the RISC-V specification. Thus no special compiler is required. However, this concept is compatible with the try/catch-like blocks and the OpenMP-based extension, allowing a customized compiler to translate similar high-level constructs to our extensions.

In our previous work, ADeLe [48, 49], an ISA extension introduces instructions that activate or deactivate a predetermined approximate operation for the whole core. This defines an architectural-level state at which all instructions that are affected by a given approximation hold the approximate behavior. Crash Skipping [134] proposed resilience and approximation control based on events of software crashes. When exceptions, such as memory access violations or illegal instructions, occur, a treatment mechanism chooses between skipping the faulty instruction, skipping the whole function, or re-executing the entire application. The Operating System stores application-specific choices of behavior and exception counters for each process to base the decision. In this thesis, we refrain from defining approximation- or scenario-specific recovery routines, leaving these to the underlying approximation units, if they support any. Instead, we inherit the controlling mechanism based on states from our previous project [48, 49] to isolate the areas that are affected by approximation and scale quality at execution time.

3.3 Approximation and interface related projects

Analyzing the behavior of applications when subjected to approximation highlights the need for runtime control [25, 44], and approximation-agnostic control interfaces miss specificities of hardware approximations and the impact of their overhead on energy efficiency [28, 31, 81]. Thus, integrating approximation and application in the evaluation is essential to accurately demonstrate the impact of specific execution scenarios on underlying approximation hardware. The EnerJ interface [122] and its Truffle [34] hardware support are an outstanding attempt towards this objective. EnerJ introduced the concept of high-level software type specifiers that indicate data values that allow deviations from the exact value or not. Upon compilation, software operations that handle approximable values are replaced with approximate versions of instructions that guide fine-grained scaling of approximations at multiple levels, such as logic and memory voltage scaling, FPU precision scaling, and DRAM refresh rate. Truffle defined the microarchitectural structure to support the language, as well as the actual ISA semantics. The architecture implements scalable dual-voltage storage structures, register bank and SRAM, and shadow approximate functional units as sources of approximations. The fine-grained control, however, does not consider the control delay to transition voltage domains or to adjust parameters that may amortize the energy benefits of approximation. Our design, instead, is based on a coarser control mechanism in the granularity of states to mitigate such delay.

Quora [132] introduced a domain-specific vector processor that can scale the precision of computation elements. An ISA extension includes fields that encode a quality requirement within each instruction, which Quora uses to scale precision of operands, choose between fully-accurate or approximate processing elements, or guide voltage scaling. In TABSH [2], a conceptual processor containing approximate and precise functional units was proposed. Instructions were manually tagged, from the assembly code, to run on the approximation units. The evaluation included synthetic tests, not applied to applications, but indicates potential energy savings on instruction execution. Accordion [65, 69] presented a conceptual many-core processor at which individual cores may run at a nom-

inal or an overscaled near-threshold voltage. The authors demonstrated, for Recognition, Mining, and Synthesis (RMS) applications, how scaling the problem size and correctly distributing the application into the accurate and approximate cores jointly affects the level of parallelism and the application output quality. The ISA extension designed in this thesis encodes the scaling of quality within architectural states in the target processor, instead of directly embedded within instructions fields such as in Quora [132] and TABSH [2]. Thus, compatibility with non-approximated binaries is maintained and modifications in the compilation tools are not required.

Some designs exploit existing speculation and reuse techniques, created to improve performance, in order to introduce approximation. The Value Similarity Extensions (VSX) [74] extended the reuse concepts discussed by He *et al.* [53] in an ISA extension. After an analysis step on what code regions may be targets of relaxed similarity for code reuse, the compiler tags groups of instructions that can be skipped and generates custom code to populate hardware-level tables. Upon execution, when the skippable instructions are fetched and the input similarity criteria with previous results are met, the VSX microarchitecture uses the previously computed results as an approximation of the outputs for the code block. Nongpoh *et al.* exploited the cache coherence protocols [104] and speculative execution [105] to introduce approximation. In shared-memory multithreaded environments, a preliminary sensitivity analysis tests which variables can read non-coherent values between different processing cores [104]. These memory accesses are marked with specialized load instructions that skip steps in the MESI cache coherence protocol implemented in the processor and may return correct, although not coherent, values. A similar compiler-time analysis marks branch and load instructions that have a lower influence in the final application results [105]. These are replaced with specialized approximate instructions that are not rolled back in case of a branch misprediction or failed load value speculation, proceeding with the processor execution as if the values were correctly predicted. These designs are based on existing architecture-level features and are naturally integrated within the architecture. However, a true general-purpose approximate architecture requires integrating traditional hardware-level approximation techniques, which is the purpose of this thesis.

Tagliavini *et al.* [126, 127] proposed an architecture in which a multi-banked tightly coupled data memory (TCDM) is shared between processing elements as a first-level memory in the hierarchy. Each TCDM bank is organized in two memory regions: a smaller, lower-voltage, but lower-density region, constructed using standard cells (SCM), and a larger, higher-voltage, but higher-density region, constructed using 6T-SRAM cells. Although the 6T-SRAM cells require higher voltage for reliable operation, the voltage can be overscaled at the cost of read and write errors. Thus, the design trades higher memory capacity within the core for reliability using voltage scaling. A reliability management unit (RMU) is responsible for placing critical data entirely in the SCMs, fully approximable values in the overscaled 6T-SRAMs, and tolerant data split between both, allowing errors only in the least significant bits. For control, the authors proposed an OpenMP-based programming model that tags variables or computation blocks as approximation tolerant. Then, the compiler places the associated data within the correct memory regions and inserts voltage switch points that activate the approximate operation in the 6T-SRAMs.

Nakamura *et al.* [97] exploited compression in the cache level to introduce approximate memory accesses using a specialized load instruction. The authors discussed a programming model to define the level of approximation using dedicated probabilistic branch instructions, manually annotated by the programmer. Thus, error-resilient data can be accessed through the compressed cache to introduce approximation. The idea of resilient and nonresilient data to be distributed into distinct memory regions was also demonstrated in our previous work, AxRAM [37, 38]. Instead of being based on annotations, AxRAM attempts to automatically identify structures that typically store critical data and maps them in the reliable region. The resilient data are stored in voltage-overscaled regions within the same memory structure. Although such a generic approach does not guarantee every execution instance to be reliable, it allows for supervisor-level controlled execution of unchanged applications. If the applications allow changes, they can tweak their configurability for improved resilience.

Parasyris *et al.* [107] presented another OpenMP-based programming model that classifies individual tasks in the application according to their significance to the final result. Less significant tasks may be scheduled to run in entirely voltage-overscaled cores in the target system employing Dynamic Voltage-Frequency Scaling (DVFS) to select a fast and low-energy, but unreliable, operating point (low voltage, high frequency). The programming model includes the definition of recovery routines, that trigger a fast transition to a slow, but reliable, configuration at the same core (low voltage, low frequency). HIART-MCS [58, 59] introduced a processor for mixed-critically systems that embeds an accuracy-configurable floating-point unit to reduce the overhead of computation. Their main objective was to improve the survivability of lower-priority tasks in such systems by varying the accuracy of computation. AxE [12] proposed another task-based programming model in which tasks are mapped according to their resilience in accurate and approximate cores within the same chip. In their demonstration, the authors introduced approximation capturing a specialized multiplication instruction in a co-processor in the approximate core. In this thesis, we discuss an architectural extension compatible with multicore implementations that can integrate multiple levels of approximation within different cores, allowing a supervisor-level software arbitrator to distribute tasks according to their resilience to error and criticality.

Finally, few projects attempt to cover aspects of nonconfigurability into their designs, even though many are the dedicated logic alternatives for approximate operations available in the literature. ProACt [22] proposed an *Approximate Floating Point Unit* that employs memoization to predict the results of floating-point computations. The memoization look-up table is activated and has its precision configured using a dedicated instruction. This transition instruction activates an approximate mode of operation, at which the results of floating-point operations are preferably fetched from the table. ProACt was built to provide performance improvement, thus the energy overhead of maintaining both the look-up table and the Floating-Point Unit in the system is not treated. Ndour *et al.* [101] presented an architecture that included approximate arithmetic and memory management units that coexist with their accurate counterparts. Accurate and approximate versions of instructions were implemented to select which unit to use for execution. Taştan *et al.* [128] implemented approximate arithmetic units and precision scaling to of-

fer approximate behavior to specific instructions. The design was optimized for machine learning applications in IoT environments. Our architectural extension was specifically built to accommodate the overhead of non-configurable units, allowing it to be amortized between coarser-grained state changes, instead of rapid transitions as in ProACt [22] and the proposal by Ndour *et al.* [101].

3.4 The design choices that led to this thesis

Our fundamental design requirement is a generic hardware-software interface that allows general-purpose hardware to employ and control hardware approximations and run resilient applications reliably based on application-level instructions or supervisor-level knowledge of the behavior of approximations. To offer such a generality, first, we build the design as an extension to a prominent open ISA Specification, RISC-V. Next, the architecture needs to take into account the different levels of configurability of approximation modules. Non-configurable modules require that their accurate counterparts are available in the system, which imposes area and energy overhead, and control mechanisms that account for gating the unused modules. Indirectly-configurable modules need to be linked to external configuration knobs to scale operating parameters, such as DVFS mechanisms, and expose this interface to the software controller. Directly-configurable modules need to expose their own dedicated configuration knobs. Moreover, these configuration knobs need to be consistent and coherent between different applications running on the same system.

To amortize the overhead of approximation control, we increase the control granularity to create approximation states. A state exposes the execution of all affected instructions to the approximate operation. This allows for lengthy power-gating transitions of non-configurable modules, as well as selectable voltage scaling of indirectly-configurable ones. The transition between states is handled by control registers integrated within the ISA, creating a straightforward interface for the software to query the current state and define the next one by reading and writing the registers. Differently from employing dedicated control instructions, writing to and reading from dedicated control registers can be done using the existing RISC-V CSR instructions without any change to the existing software toolchain, reducing the design overhead. The register-based control may slightly increase the complexity of analyzing code that may be approximated or not. The designer would need to identify whether a code section is enclosed or not by the operations to turn approximations on or off to identify the approximation state, instead of just focusing on the code section itself. This increased complexity, however, is easily mitigated by the ability to actually query the system for the current approximation level, by reading the registers, and delegating control to an external entity, such as a supervisor system. Also, the control registers are tightly integrated within the core, where approximation modules such as multipliers and adders are, without the need to go through the data bus, as for memory-mapped registers.

Separated registers for different privilege levels allow explicit supervisor-level control of the approximate state in which the user-level application runs. Thus, the approximation

state is also seamlessly saved and restored during context switches. Finally, this thesis defines the means for the applications to recognize the approximation environment and adapt to it. This exposition of approximation capabilities is handled by a dedicated data structure stored in memory that holds enough information for the software to know, at execution time, what are the available approximations and how they should be used. This allows future software implementations to query their approximation environment and then explore the approximations under their requirements, instead of requiring *ad hoc* approximation-specific implementations for each target scenario.

Chapter 4

Integrating Approximate Computing

Architecture-level specification

The main goal of this thesis is to design a general-purpose programmable processor that allows the integration of coexistent hardware-level approximation techniques. To achieve this, it is fundamental to provide a generic hardware-software interface that allows the software to recognize the hardware approximation capabilities and control the approximations at execution time. Our design is the Risk-5 ISA extension (Section 4.1), an interface between the approximations and the running software offering control registers for different privilege levels to establish their requirements. To actually support this extension, we also describe the concepts of a hardware controller and the underlying approximation units that communicate with it (Section 4.2). Finally, to allow efficient and reliable execution in multi-application scenarios, we propose a supervisor extension defining high-level control operations (Section 4.3).

4.1 Risk-5: Approximation-aware ISA Extension

One of the main requirements of approximation control is to provide a low-overhead software interface. Energy savings achieved by approximations are limited to the amount of time they are active and how much of the total application execution time this represents. Thus the time dedicated to control should be kept to a minimum. Moreover, different software, including the Operating System, have different levels of quality requirements, and their execution needs to be protected from each other. Different approximation techniques also offer different levels of controllability. The characteristics of a modern system, in which multiple applications coexist and communicate with a higher privilege-level supervisor, need to be accounted for to design a hardware-software interface.

A common approach to offer a software interface for approximation units is to define alternative approximation-specific instructions that indicate that the operation should be executed in the approximate functional unit [2, 34, 100, 101, 122, 128, 132]. Although this incurs the lowest possible software overhead, it requires every application to be recompiled to the new architecture to support approximations. Moreover, such control in an instruction granularity would require functional units to be duplicated and left always active to support both accurate and approximate execution, imposing an energy overhead.

Table 4.1: CSRs defined in Risk-5

Register	Description
`mrkgroup`	Current approximation group
`mrkaddr`	Approximation Description Table base address

`mrkav`
`srkav`
`urkav`
	Available approximations at machine/supervisor/user privilege mode

`mrkst`
`srkst`
`urkst`
	Control of approximations at machine/supervisor/user privilege mode

`mrkacbhv` `mrkdcbhv`	Activation/Deactivation behavior of approximations

Risk-5, instead, exposes to the software stack RISC-V Control and Status Registers (CSRs) that define an *approximate state* of the target RISC-V implementation. After the state is defined, the execution is affected by the selected approximations without any changes in the instruction-level software interface. This allows privileged-level code in a supervisor system to launch an application in approximate or unreliable hardware, regardless of any support from the application side. Moreover, the coarse-grained control allows the architecture implementation to fully disable hardware modules that are not in use in the given approximate state, increasing the potential energy savings.

The CSRs, defined in Table 4.1, are registers that encode a selection of approximation groups and define the base address of a data structure containing information about each approximation unit, the availability of each approximation unit within the current group, whether the control of specific approximation units is delegated from machine-level to less privileged code, the current status of every available approximation, and the behavior of their activation and deactivation operations. Moreover, individual approximation units that support additional controllability are exposed as peripherals, and the software interacts with them using approximation-specific device drivers.

4.1.1 Approximation groups

Approximation units in an implementation are distributed within *approximation groups*. A group is an arbitrary selection of approximation units where individual units may be part of one or more groups concurrently. Approximation groups are particularly necessary for implementations that integrate a large number of approximation units in the same architecture so that single registers cannot store status information for all of them at the same time.

The selection of an approximation group is the primary control mechanism in the ISA extension. The register `mrkgroup` is visible only at the machine level and holds the value of the current selection. If the implementation integrates a few approximation units and does not use any group, `mrkgroup` may be hardwired to zero. Otherwise, its value is monitored and any changes trigger a *group change operation*.

When a group change operation is started, the implementation ensures consistency of the execution state by suspending the issue of any new instructions. All pending operations that should take place before the write to `mrkgroup` are committed and the remaining ones are discarded, leaving all execution units idle. Any active approximation units are deactivated, and only the ones belonging to the new group are seen as available by the software. Any delegation settings from machine-level to less privileged code are discarded. The program counter is set to the instruction just after the write to the control register, and only after every functional unit is ready the execution is resumed.

Changing the current approximation group may be a lengthy operation, and software should see it as an initialization step or an adjustment to a different execution scenario. Thus, implementations should place in a group approximation units that are likely to be used together by the software stack, even if the same unit appears in multiple groups, thus allowing an approximation group setting to be kept for an extended time.

4.1.2 Approximation Description Table

The *Approximation Description Table* (Figure 4.1) is a data structure stored in a read-only protected memory region that contains additional information about the approximation units and the approximation groups in the implementation. It should be read upon initialization to identify the appropriate approximation group for the scenario and load the respective device drivers for fine control of configurable approximations. The register `mrkaddr` is a read-only register, visible only for machine-level code, containing the base memory address of the table. This base address contains a value that encodes the specification version of the implementation, and it should be used by software to gain knowledge on how to read the table and its respective fields.

Below the specification version, the current specification of the Approximation Description Table is divided into two major sections: the list of groups and the list of approximation units. The first row in the list of groups contains the number of available groups and is followed by a description of each group. Each group has an ID (gid_g), which is the value that the software should set to `mkrgroup` to select it, and a value representing the number of approximation units belonging to the particular group. Finally, the description of the group contains a list of approximations in the group, formed by the approximation ID within the group and a pointer to the location of the approximation unit description. Software can find the description of the approximation unit in the second section of the Approximation Description Table by fetching the memory address $\texttt{mrkaddr} + Papprox_{group_{approx}}$. Group IDs and approximation unit IDs within each group are completely independent and not necessarily incremental or contiguous.

In the list of approximation units, the second section of the Approximation Description Table, each unit is represented by five attributes: a unique identifier of this unit in the implementation, a taxonomy index, the JEDEC Manufacturer ID encoded according to the `mvendorid` RISC-V CSR [137], a vendor-specific identifier, and the hardware identifier for the software to find a driver. The taxonomy locates the approximation unit in a group of attributes that characterizes it, such as the type of controllability it offers, the execution unit or hardware module it affects, the technique it employs, and the expected outcome.

Specification Version	
G: Number of groups	
gid_1: Group ID	
NA_1: Number of approximations in group gid_1	
$ApproxID_{1_1}$	$Papprox_{1_1}$: pointer
$ApproxID_{1_2}$	$Papprox_{1_2}$: pointer
...	...
$ApproxID_{1_{NA_1}}$	$Papprox_{1_{NA_1}}$: pointer
...	
gid_G: Group ID	
NA_G: Number of approximations in group gid_G	
$ApproxID_{G_1}$	$Papprox_{G_1}$: pointer
$ApproxID_{G_2}$	$Papprox_{G_2}$: pointer
...	...
$ApproxID_{G_{NA_G}}$	$Papprox_{G_{NA_G}}$: pointer
aid_1: Implementation-specific approximation ID tid_1: Taxonomy index vid_1: JEDEC Manufacturer ID $vaid_1$: Vendor-specific approximation ID $hwid_1$: Hardware ID	
...	
aid_n: Implementation-specific approximation ID tid_n: Taxonomy index vid_n: JEDEC Manufacturer ID $vaid_n$: Vendor-specific approximation ID $hwid_n$: Hardware ID	

Figure 4.1: Structure of the Approximation Description Table. *This data structure stores information about all approximation units available in the system.*

(Reused, with permission, from [46] ©2020 IEEE)

4.1.3 Approximation availability and delegation

The registers `mrkav`, `srkav`, and `urkav` expose and control the availability of approximation units at machine, supervisor, and user privilege levels, respectively. Each of its bits represents the availability of one of the approximation units belonging to the current approximation group – a value of 1 means the unit is present. The register `mrkav` is seen as read-only by software. It can be written only by the hardware implementation during an approximation group change operation.

Registers `srkav` and `urkav` can be written by more privileged code to delegate the control of a given approximation unit to less privileged code. The control of approximation units that are not delegated falls into more-privileged code and is completely transparent to less-privileged code. This allows a higher-ranked application to force execution under approximation hardware even though the target application does not request nor is aware of it. When the control of an approximation unit is delegated to less-privileged code, the execution of any operation that changes the privilege level may result in the activation or

deactivation of given approximation units. By default, no approximation unit is delegated to less privileged code, leaving all control to machine mode only. This default configuration translates into a default value of 0 for all bits of both `srkav` and `urkav`.

4.1.4 Approximation status and control

The registers `mrkst`, `srkst`, and `urkst` define the status of each approximation unit in the current group at the machine, supervisor, and user privilege levels, respectively. Each of its bits represents the status of one unit in the group, and a value of 1 means that the approximation is active. Deactivating an approximation unit may mean turning it off and directing the datapath to an accurate counterpart, in case of non-controllable units, or resetting the specific control knobs to an accurate level, in case of controllable ones.

Control and status registers can be read and written by code in the same and higher privilege levels, but only writes of valid values are held. The invalid are values that attempt to activate approximations that are not in the current group, have not been delegated to the current privilege level, or would lead to a conflicting approximate state. Reads on the same privilege level of non-delegated approximations should show a deactivated state.

A conflicting state is a state in which multiple approximation units that serve the same purpose are concomitantly activated in an implementation that does not support it. For example, a scalar architecture in which one single integer multiplier may be replaced by an approximate non-configurable multiplier. To offer multiple approximation levels, the architecture contains three approximate multipliers connected to the datapath, each yielding a different error pattern. Since they all replace the multiplication operation, they cannot be activated at the same time. In that case, writes to control registers that would lead to more than one multiplier active should be filtered out by the implementation.

4.1.5 Activation/Deactivation behavior

The status and control registers expose to the application information on the status that the approximation units were requested to be. However, this is not necessarily their actual status. The activation or deactivation of approximation units may impose some delay before the requested execution unit is available to be used. It is the responsibility of the machine-level code to determine whether it is safe to execute code while activation or deactivation operations are in process.

The expected behavior, when activating or deactivating approximation units, is defined in registers `mrkacbhv` and `mrkdcbhv`, respectively. Each of their bits contains the behavior for one of the approximation units in the currently selected approximation group. A value 0 means that, while the activation/deactivation operation is in progress, the implementation should proceed with software execution as usual. This behavior assumes that (a) the operation has no side effects in the overall execution behavior, (b) it is safe to run the section of code that follows the operation beginning because the particular approximation unit in question is not used, or (c) the activation/deactivation delay of the unit in question is negligible in the current execution scenario. These assumptions imply a design choice in the system and have their own implementation- and application-specific

considerations. For example, if changing the status of an approximate data memory sets it in a very unreliable state for a delayed time, it is likely not safe to proceed. On the contrary, if dealing with an approximate multiplier, it is probably safe to proceed as long as the application does not use multiplication in the status change delay timeframe. If the assumptions do not hold true for the current execution scenario, the behavior can be set to 1, which instructs the implementation to wait until the operation is complete, similar to the *group change operation* described in Section 4.1.1.

4.1.6 Approximation-specific controllability

Apart from the generic activate/deactivate control offered by the status and control registers, controllable approximation units may expose to software additional control knobs. In that case, the approximation unit itself defines the appropriate software interface, connecting the control knob to the architecture as a peripheral and providing a driver. The Approximation Description Table includes information, such as the Vendor ID and Hardware ID, for the software to find and load the appropriate driver. Regardless of approximation-specific controllability, however, approximation units should still offer the default activate/deactivate control for architecture-level and application-agnostic support.

4.1.7 Interaction in multicore architectures

In a multicore architecture, some approximation units, such as memory or storage, may reside outside of the cores and be shared by them. Attempting to implement independent control for these units would likely result in a conflict if the software running on different cores requests different levels of approximation to the same shared resource, such as cache or main memory. Thus, approximation units that are shared between multiple cores have their control interface attached only to a single master core. Software running in other cores should communicate their approximation requirements to the master core, which configures a shared approximate resource to the lowest value required. This conservative configuration exposes all applications that share the same approximation unit to the lowest value supported, restricting energy savings but allowing for safer resource sharing. Since different cores are likely running under different privilege levels at a particular time, the delegation of control to lower-privilege code is not practical for shared resources and should be restricted by machine-level software at the master core.

4.2 Hardware support

The architectural specification describes the architecture-level interface to offer software the ability to control approximation-hardware units without any implementation details. In this section, we discuss a sample implementation of the hardware features needed to support the ISA extension. Figure 4.2 shows a high-level diagram of the additional supporting hardware, including the approximation units and the main connections to the relevant existing RISC-V core. This sample implementation includes, within the core, as approximation units, two integer multipliers [96]. Since the integer multipliers are

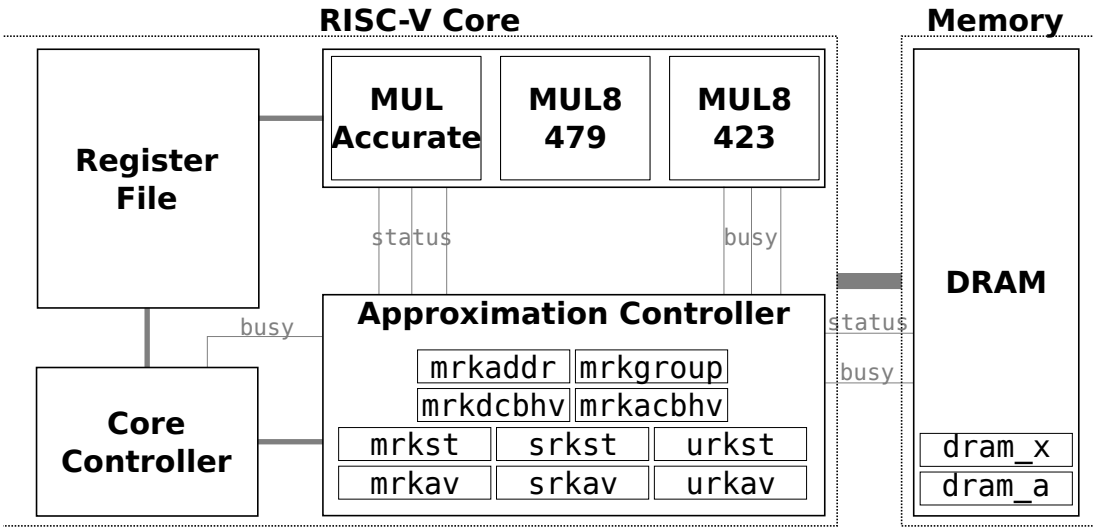


Figure 4.2: Supporting hardware for the ISA extension. *An approximation controller decodes the control and status registers and distributes internal circuitry to activate or deactivate each approximate module.*

(Reused, with permission, from [46] ©2020 IEEE)

non-configurable, an accurate multiplier is also included. Outside the core, an external voltage overscaled DRAM module communicates with the core through a configurable approximation-aware data access interface [37, 38]. To control the execution, the core includes the approximation controller and the special registers described in Section 4.1. This controller determines the status of each approximation unit and sends to the core a single aggregated `busy` signal, which indicates whether the execution needs to be held to accommodate an activation/deactivation operation (see Section 4.1.5).

4.2.1 Non-configurable approximate multipliers

Non-configurable approximation units are designed to replace their accurate counterparts during approximate execution. Naturally, this requires that both accurate and approximation units coexist in the design, which produces area and energy overheads. To minimize overhead and maximize energy savings, units that are not in use need to be deactivated, preferably for extended periods of time to mitigate the delay of power gating [66]. Our coarse-grained control is designed to allow the mitigation of this delay.

Integer multiplication is a computationally intensive operation with high energy cost [144], which makes it a popular target of research effort in designing approximation units. In this implementation, we used two multipliers from the EvoApprox8b library [96], *mul8_479* and *mul8_423*. These were composed from 8-bit partials and selected to produce approximately 2% and 5% mean relative error, at an energy cost of 70% and 50% compared to an accurate multiplier [49].

Figure 4.3 shows the hardware implementation. Each of the multipliers – two approximate and one accurate – is placed in its own power domain with independent power-gating controllers. Each controller takes an input $\texttt{status}_{mult_i}$, to indicate whether the unit is active or not, and outputs $\texttt{busy}_{mult_i}$ while a status change is in progress. The input

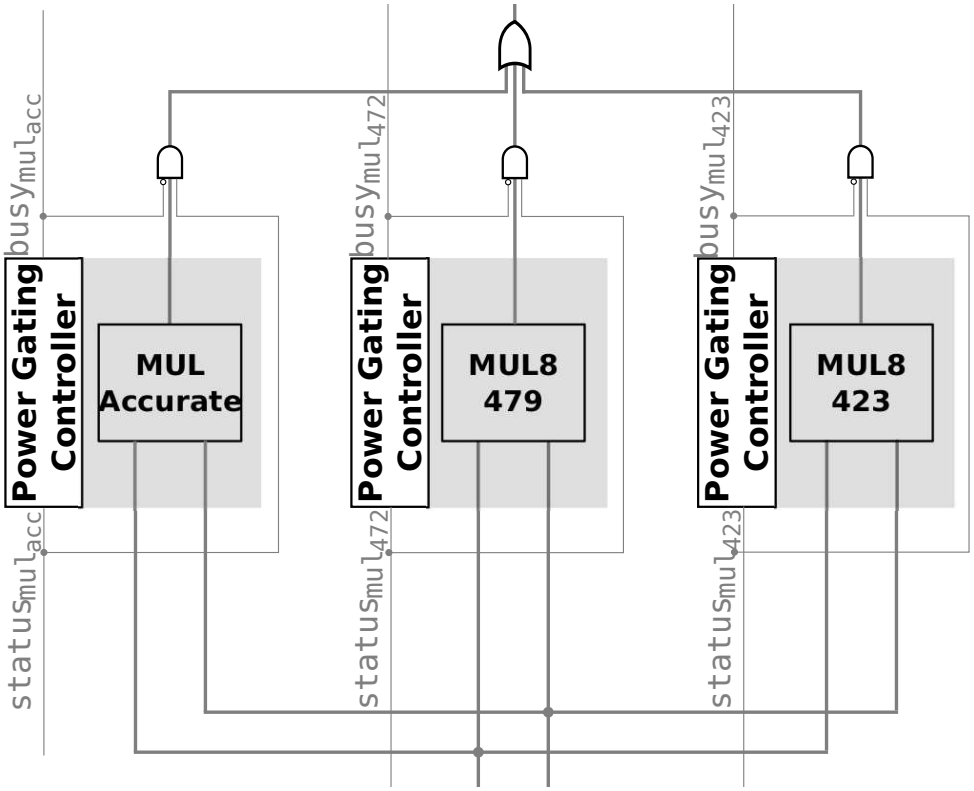


Figure 4.3: Implementation of non-configurable multiplication hardware. *Multipliers that are not in use in the current state are power gated for further energy savings.*
(Reused, with permission, from [46] ©2020 IEEE)

operands of all multipliers are directly connected to the same source. Each individual output product is clamped low in isolation cells when $\texttt{status}_{mult_i}$ is 0 or $\texttt{busy}_{mult_i}$ is 1 to ensure signal integrity [66], and the final multiplication product is the logical OR of each output. Taking a reasonable assumption that the critical path delay of each multiplier is significantly higher than the delay to build the $\texttt{busy}_{mult_i}$ signal (implemented as a counter, for example), this adds a delay of two gates to the overall multiplication critical path. This power-gating approach is compatible with existing high-efficiency power-gating controllers available in the literature [85, 109, 143, 147].

An alternative gating for non-configurable approximation units is clock gating. This would eliminate the power-gating controllers and the isolation cells, adding a two-way multiplexer in the input operands that selects a value 0 if the individual multiplier is deactivated (the $\texttt{status}_{mult_i}$ signal is 0). Since such a two-way multiplexer can be built as just a logical AND with $\texttt{status}_{mult_i}$, this clock-gated approach adds the same two gates delay to the overall multiplication critical path, reduces hardware overhead, and allows a finer-grained control for eliminating the power-gating controller delay. However, clock gating reduces only dynamic power dissipation, leaving static power untouched, and leakage has become a significant source of energy overhead [68], especially if multiple replicas of the same hardware module need to be included in the design. The control mechanism favors coarse-grained control to accommodate the power-gating control delay, and the implementation employs isolation cells to allow the execution to proceed, provided that the specific hardware unit is not required.

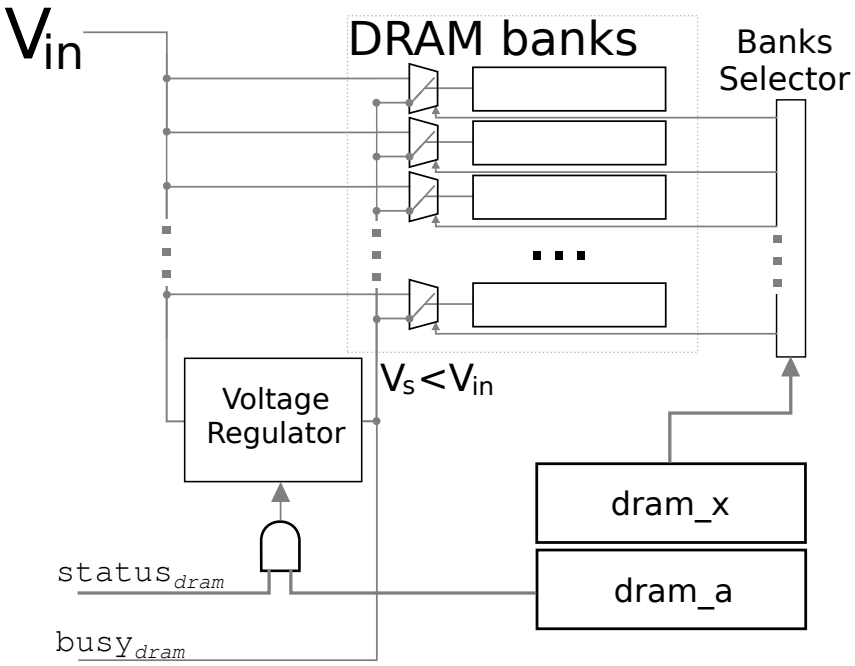


Figure 4.4: Implementation of indirectly-configurable DRAM. *The memory is divided into nominal reliable banks and error-prone energy-efficient banks.*

(Reused, with permission, from [46] ©2020 IEEE)

4.2.2 Configurable approximate DRAM

Configurable approximation units expose an architectural control knob that allows an external controller to define the amount of approximation for computation. It would be desirable that the knob allowed the unit to be configured in accurate mode. However, this is not always necessarily true, in which case it should coexist with an accurate counterpart as such a non-configurable unit. The configuration may affect the approximation directly, such as the selection of an approximated Floating-Point format [86], or indirectly, by adjusting an external parameter, such as the supply voltage of the memory module [37]. From a controller point of view, the interface for both approaches is the same. Both expose an interface where some value needs adjustment. Despite that, controlling external parameters imposes additional hardware overhead and execution delay, while directly-configurable units are usually self-contained.

DRAM represents a large fraction of the power dissipation of a RISC-V system implementation [67]. Adjusting common DRAM parameters, such as supply voltage, refresh rate, or latency, can significantly reduce energy cost, while exposing the stored data to some degree of error [23, 24, 75, 82, 113]. However, some values stored in RAM, such as code regions, pointers, and control variables, are critical data and cannot be safely subjected to errors. Since such data is indistinguishable at the memory level, an approximate DRAM implementation requires additional information from software and a dedicated control interface. In this implementation, we use voltage scaling as a source of approximation [24, 75], controlled through a memory access protection interface [37, 38].

Figure 4.4 shows the approximate DRAM controller and its interface. It implements a power network that can independently supply each memory bank in the DRAM mod-

ule [75]. Each voltage sink connects to a switch that selects one of two voltage sources: an external and higher supply and an internal and lower, provided by a voltage regulator. The selection is made according to the value of a `dram_x` memory-mapped register and a $\texttt{status}_{dram}$ input signal. Each bit of `dram_x` corresponds to a memory bank and takes a value of 1 to select the lower voltage from the regulator, which is then logically AND-ed with $\texttt{status}_{dram}$ to feed the selector.

The error characteristic of a DRAM module suffers from strong variability when subjected to lower voltage supplies [24]. To account for this variability, the interface allows the voltage supply to be adjusted through a `dram_a` register. The approximate DRAM controller also outputs a $\texttt{busy}_{dram}$ signal, which is generated from a fixed-value counter that is triggered at every change on any of the input signals or registers. The counter value reflects the delay imposed by changing the voltage levels in the DRAM banks.

4.2.3 Approximation controller

The approximation units communicate with the main control unit of the processing core through an approximation controller. The controller stores the Control and Status Registers, outputs the $\texttt{status}_i$ signals for each approximation unit, and builds the core `busy` signal composing all $\texttt{busy}_i$. It also receives a CSR selector, write enable, and data values, acting as a portion of the CSRs register file, and a two-bit representation of the current privilege level the core is running [137]. In this implementation, we demonstrate the integration of three approximation units – two multipliers and a DRAM controller – that are selected using the three least significant bits of control registers, leaving the remaining unused and hardwired to zero. Considering the reduced number of approximation units, no additional approximation groups are necessary, and `mrkgroup` is hardwired to zero.

Figure 4.5 shows the partial structure of the controller that builds status and busy signals in the user privilege level. Gray lines represent connections, and their thickness relates to the width of the bus. For the sake of simplicity, we omit circuitry to select, read, and write registers, as well as protection and trap generation. The former is no different from a regular implementation of a register file, and the latter traps the processor whenever the software attempts to read or write registers that it is not allowed to in the current privilege level, to maintain and mask virtualization support [137]. The implementations for the supervisor and machine levels are equivalent, except that the machine-level `mrkav` is a constant, not a regular register. In the complete implementation, the outputs $\texttt{status}_i$ are multiplexed according to the privilege level.

The status registers `mrkst`, `srkst`, and `urkst` are multiplexed according to the current privilege level to produce the status signals for the approximate multipliers ($\texttt{status}_{mult_{472}}$ and $\texttt{status}_{mult_{423}}$) and the DRAM controller ($\texttt{status}_{dram}$). The accurate multiplier ($\texttt{status}_{mult_{acc}}$) is active whenever neither of the approximate ones is. Attempted reads and writes to `srkst` and `urkst` from the same privilege level as the register are logically AND-ed with `srkav` and `urkav` to mask not delegated approximation control. The implementation also restricts that both approximate multipliers are activated at the same time. The group change operation, identified by an attempted write to `mrkgroup`, forces the default initial values to be written to all registers.

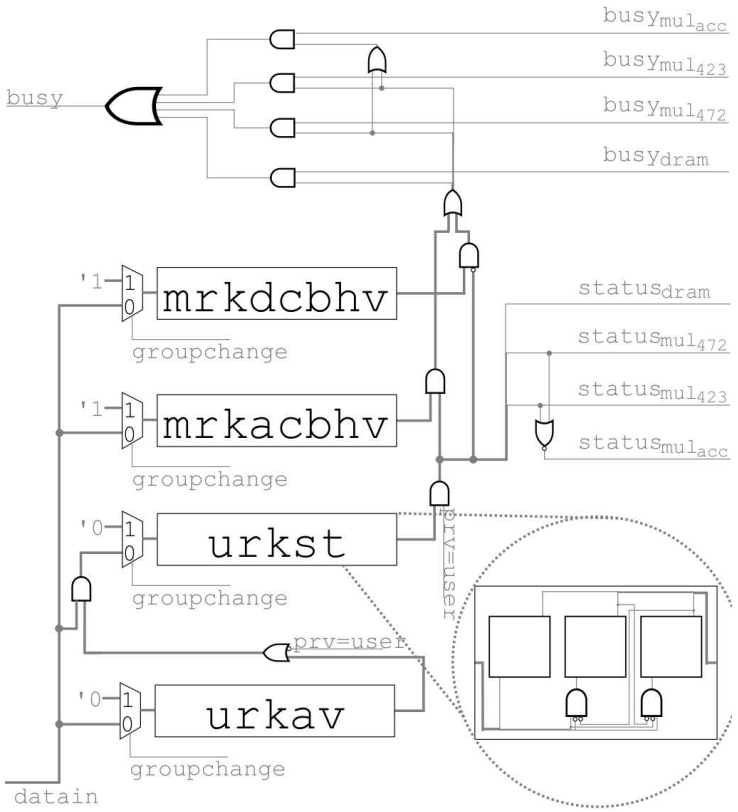


Figure 4.5: Implementation of the approximation controller. *Control signals are generated from the state indicated by Risk-5 CSRs.*

(Reused, with permission, from [46] ©2020 IEEE)

The global `busy` signal is a combination (logical OR) of $\texttt{busy}_i$ masked with its respective behavior and current state. Each individual $\texttt{busy}_i$ is considered if its current state is deactivated and the deactivation behavior is to hold execution, or if the state is activated and the activation behavior is to hold. Since the accurate multiplier is a complement of the approximate ones, its busy signal considers the state of any of the approximate counterparts. During a group change operation, the default values written to all registers force the processor execution to be held while any of the $\texttt{busy}_i$ signals are active, ensuring execution consistency, and only then any configuration is applied.

Finally, in a scenario in which multiple approximation groups are necessary, the implementation requires some modifications. The approximation availability register `mrkav` is implemented as a read-only memory and the status registers `mrkst`, `srkst`, and `urkst` as inner register files, all addressed by `mrkgroup`. Since all registers are reset on a group change, only one of the status registers for each privilege level can contain a value different from zero at any time, and the output status signal can be inferred by the logical OR of their individual positions in each status register they appear.

4.3 Software Interface

The ISA extension specified in this thesis defines that any software interacts with the architecture-level control structure by reading and writing the specific Control and Status Registers. RISC-V provides dedicated instructions to read and write CSRs from/to integer registers, immediate values, or by setting or resetting specific bits in the CSR while leaving the remaining unaltered. Thus, no additional instructions are required. However, some operations are privileged, such as modifying the execution behavior while controlling approximations, or require interaction with device drivers, in the case of controllable approximation units. For this reason, in a multi-privileged system, these should be implemented as environment calls. In this Section, we discuss a software interface to support the approximation control capabilities, initially focusing on a minimalist operating system that offers approximate execution behavior to a mixed-resilience application. We then discuss support for more complex scenarios with shared resources.

4.3.1 Minimalist support

To offer minimalist support to approximation control during execution, we describe an Operating System (OS) that implements a Hardware Abstraction Layer, including device drivers to communicate with the approximate DRAM controller from Section 4.2.2, and a virtual memory manager. The target application reads four matrices from memory and performs two matrix multiplications, where the first multiplication result is used in a scenario that admits some error and the second does not. For illustration purposes, we consider that the OS runs entirely in machine level, addressing all memory physically, and the application at the user level. Before the deployment of the application, a profiling step is executed in the target execution hardware. This step extracts the error characteristics of the installed DRAM module, mitigating variability aspects [24], and establishing a safe, although error-prone, voltage level for execution.

Upon initialization, the OS reads the Approximation Description Table and finds that it has available a single approximation group. It infers from the taxonomy index that the approximation units are two integer multipliers and a DRAM controller and loads the driver to configure the DRAM. Then, the memory-mapped register `dram_a` is configured to provide the voltage level established during the profiling step, and `dram_x` sets in the accurate state only the memory banks that contain addresses within the bounds of the OS own code and data. These configurations protect the OS operations from unreliable execution while preparing the system to transparently offer to applications an approximate environment. Then, the approximate DRAM is activated for both machine and user levels by setting the third bit of `mrkst` and `urkst` to 1. The approximate multipliers are made available to the application by setting the two right-most bits of `urkav` to 11 and, to minimize the delays of power gating, execution holding is disabled for the multipliers by setting the two right-most bits of `mrkacbhv` and `mrkdcbhv` to 00. In the worst-case scenario, if the application attempts to perform multiplication while the operators are not yet available, the operation will return zero, which is itself a degree of approximation. The system is now ready to build the application address space for execution.

The Page Table is built, adding an extra field for each Page Table Entry that categorizes the page as accurate or approximate. When storing pages in the RAM, pages tagged as accurate are always physically placed in memory banks marked as accurate in `dram_x`, increasing the number of accurate banks if necessary. Approximate pages may be stored in either type of memory bank, but preferably in the approximate ones. Then, the system allocates accurate pages to store the application code and the program stack. Fetching instructions reliably is critical for any application since a single bit flip in an instruction would likely result in the unsuccessful execution of the application. The program stack stores other critical data, such as statically allocated local variables, loop indices, memory pointers, and return addresses, thus protecting it can avoid execution crashes [37, 38].

In the end, the OS launches the application. Upon initialization, the application uses a system call to inquire about the availability of any approximations and receives back a modified copy of the Approximation Description Table containing only the multipliers. It selects *mul8_472*, based on its taxonomy tag, for having the higher potential for energy saving, and activates it by setting the second bit of `urkst` to 1 before loading the first two matrices. It performs the multiplication, stores the matrices, and deactivates the approximate multiplier before loading the remaining matrices. Fetching and loading the data from storage is a lengthy operation, so it allows enough time for the approximate multiplier to be activated and deactivated safely before the multiplication. Since all matrices are dynamically allocated, they are transparently stored within approximate memory pages, exposing even the data of the second set of matrices to a low level of errors without any intervention or knowledge from the application side.

Finally, both memory allocation and I/O handling use environment calls in their implementation. These implicitly trigger approximation status changes to whatever is set in `mrkst` when elevating the privilege level. To ensure that the application gets the level of approximation it requested, the OS checks the value of `urkst` when entering the environment call, and resets its own setting accordingly before returning the execution to the user level, allowing enough time for the setting to be applied.

4.3.2 Approximation coherence of shared resources

The minimalist support discussed in Section 4.3.1 coordinates approximation control for a single application scenario. However, more complex scenarios may introduce multiple applications running under multiple processing cores, accessing shared computing resources, including the approximation units, in a multi-user virtualized execution. All these variables create additional challenges to coherently control the approximate execution level of the target architecture. The architectural interface offers direct control over the underlying hardware, thus allowing these matters to be dealt with at the software level.

In scenarios of shared approximate resources, the software controlling approximations should guarantee that the configuration is compliant with the requirements of every application. A configurable approximation unit should avoid coherence issues so that multiple applications, cores, operating systems, and users can concurrently access the approximate resources. In essence, the interfaces of the shared configurable approximation hardware should be centralized on a single controller that decides which configuration is applied.

If multiple applications are allowed, each application informs the OS about its specific configuration through system calls that adjust the availability, the status, and any configurable parameters of the shared approximation units. If more than one running application shares the same resource, conservative configurations should be kept by the OS to avoid any application being subjected to a higher level of approximation than supported and to reduce configuration overhead. Furthermore, status control of approximations comes naturally from the interface by saving and restoring the user-level status register (`urkst`) on context switches.

In multicore environments, the control interfaces of the approximate resources that are shared between cores are attached to one single master core. For example, in the configurable approximate DRAM introduced in Section 4.2.2, all cores would be aware of the memory approximation, but only the master core would have the ability to enable, disable, and configure it. The software running on other cores sends their approximation requirements to the master core that configures each approximation unit with a conservative configuration, the lowest level received. Thus, although the higher error-tolerant software had decreased energy benefits, the less error-tolerant one avoided execution faults due to higher-than-supported error. This level of coordination between cores can be achieved either explicitly, when the higher-privileged OS is the only responsible for the configuration, such as in the shared DRAM example, or implicitly, by trapping accesses to control registers for the OS to handle the request in the master core. This implicit approach also allows the OS to coordinate accesses from concurrent processes to a single shared approximate resource, such as a neural accelerator [36], by trapping activation requests and only re-scheduling the process for execution once the resource is available.

Finally, if supervisor execution level is supported, it would implement a second control layer, with its own access to status and delegation registers, communicating with machine level using environment calls. Additional privilege levels are possible by replicating the status and delegation registers. In a virtualized scenario, our extension supports trapping CSR accesses for higher-privileged software to handle the request, as defined in the RISC-V specification. Thus, when a virtualized OS, running in user-level, attempts to access higher privileged control registers, these accesses are trapped and redirected to the host OS. The host OS deals with the request and provides the virtualization interface. Therefore, Risk-5 supports the specific control registers for each privilege level to protect and configure the approximate resource through the management of the host OS.

Chapter 5

Implementation

Simulated behavior and FPGA prototype

This project proposes the architectural integration of approximation-hardware modules in a general-purpose processor. This is achieved by setting up an ISA extension that exposes the approximation capabilities of the underlying hardware to the software and allows for the configuration and orchestration of these capabilities. To support these features, the organization needs hardware controllers that store the status and interface with the approximation-specific modules. Chapter 4 illustrated these concepts for approximations that can allow configuration on-the-fly, discussing some of the implications of the hardware controllers. To better understand these implications and demonstrate the designs, we offer two levels of implementation for these concepts: an architecture-level functional ISA simulator and a full-system FPGA-based hardware prototype.

The ISA simulator, AxPIKE, was designed as a generalizable tool that allows designers to inject models of hardware approximation at the instruction level and evaluate their impact on the quality of results. AxPIKE is a high-level representation of a RISC-V system and implements the control registers that allow the simulated software to manage the approximate behavior of compatible execution scenarios. The environment also provides detailed execution statistics that are forwarded to dedicated tools for energy accounting. In the simulation, the framework can forward environment ("system") calls to the host system, aiding in the flexibility of the implementation by allowing applications written for Linux to be executed with minimal modification. These features allow approximation designers to rapidly evaluate the impact on quality and estimate the impact on energy of their design while accounting for controllability and possible architectural overheads.

The flexibility of the simulation platform, however, comes with a cost, which is mostly related to performance, in terms of scaling the experiments to larger scenarios, and to simulation fidelity and accuracy, in terms of how representative the results are. Accounting for multiple levels of cache and multicore organizations, for example, can cause simulation delays higher than desirable. Moreover, even though the simulator does represent the architectural integration between components, it does not consider that an architectural operation may be implemented in different ways in the microarchitectural level. This is not a problem when they are all accurately predictable, but it may be when their implementation is tampered with – i.e., when they are approximated. To validate these factors,

our second level of implementation is a hardware prototype. The prototype is powered by a real standalone quad-core RISC-V processor connected to cache and external memories, UART communication, and JTAG debugging. We augmented the RISC-V processor with the approximation controller, approximate multipliers, and an error injector on the memory accesses to represent the system proposed in Chapter 4. The communication and debugging ports, all implemented within an FPGA, allow designers to interact with the system using standard tools. Both the simulated and FPGA implementations were used to base the demonstration of the architecture proposed in this project and are described in further detail throughout this Chapter.

5.1 The AxPIKE ISA Simulator

Common hardware techniques to achieve some level of approximation are the design of dedicated logic that outputs error-prone results, the scaling of external parameters that have side effects on quality, and the inclusion of configuration knobs that introduce deviations [91]. These hardware modules are often evaluated in isolation from the rest of the system or in applications by replacing some high-level software operation with a specific software model. This approach tends to disregard the interaction and integration between the approximation units and other components in the system, limiting essential co-design possibilities [83]. An alternative for evaluation is to inject the approximations at the instruction level. Thus, instead of, for example, evaluating a hardware multiplier using a subset of random operands or overloading the high-level multiplication operator with a specialized function [8], the designer would replace the behavior of multiplication instructions in an ISA simulator. This places the approximation where it would be noticed in the application, considering the interaction with instructions and reducing the software overhead of higher-level modeling.

Setting up a simulator for this purpose, however, involves several challenges. CPU simulators are designed to produce the application output and some simplified aggregated execution statistics, such as a global instruction counter, number of memory accesses, and cache misses. Even though these statistics can be used to estimate performance, more data is needed to account for energy. Moreover, although some simulators disclose the modeling of instructions, allowing for straightforward changes, control mechanisms such as instruction issue and decode, memory and register accesses, and the production of statistics are often embedded in the software structure, requiring extensive effort to understand and modify the simulator.

To overcome these difficulties in representing and evaluating hardware for Approximate Computing in off-the-shelf instruction-level simulators, we built our own RISC-V-based simulation environment. AxPIKE uses a CPU simulator to inject approximation into instructions or data accesses. In addition, an ISA-level interface and a software library control the approximate behavior, protecting critical regions to tweak the energy-quality trade-off. The approximate behavior is represented in *ad hoc* instruction-level models interpreted through a configuration file. The CPU simulator produces the application output and detailed execution statistics to account for energy, such as memory access

traces, instruction-specific counters, and specific operand traces. These statistics are forwarded to dedicated tools and models that evaluate energy and quality.

This Section presents the AxPIKE environment development and implementation and demonstrates its usefulness in a set of summarized experimentation results. For a set of applications, we inject a high-level approximate multiplication model [8] and a DRAM error model based on voltage overscaling [38]. We configure the simulator to produce statistics about executed multiplications and DRAM accesses and then apply specialized tools to estimate energy consumption.

5.1.1 Comparison with other simulators

Approximate Computing designers have proposed different solutions to evaluate approximations at the architecture level. VarEMU [136] extends instructions by replacing or augmenting them with custom software models, estimating energy based on internal power models. It does, however, limit the representation of microarchitecture details to improve performance. This reduces the accuracy of execution statistics to the modeled details and restricts the evaluation of low-level approximations. Also embedding its own power model, React [138] instruments applications at design time for later simulation. Apart from requiring changes in the source code and recompilation, it limits the evaluation to pre-defined approximation techniques and statistics, not allowing further configuration.

Approxilyzer [131] evaluates the impact of perturbations in the execution caused by Approximate Computing using single soft-error models. The approach allows designers to estimate the application resiliency but limits a more accurate representation of many approximation techniques. ADeLe [49] introduces customizable error and power models to be applied to a custom CPU model. Although stacking levels of customization enlarges the frontiers of design space exploration, it tends to increase the effort in setting up the evaluation. Moreover, the simulation is only as accurate as the architecture is detailed in the base CPU model, creating a direct dependency on the representativeness of results.

Our simulator inherits the customization of error models from VarEMU [136] and ADeLe [49] while linking it with the rapid setup time of React [138] and Approxilyzer [131]. Based on the Spike RISC-V ISA simulator, AxPIKE can run an application off-the-shelf, representing a multi-privilege multicore CPU and including ISA-level microarchitectural details. Still, it allows the injection of errors and the collection of statistics. Also, we refrain from embedding power models in favor of producing data for external energy estimation. Thus, we leave to the CPU simulator only the aspects that are related to the behavioral execution of instructions itself while still allowing flexible extension and collecting enough information for the evaluation of Approximate Computing designs.

5.1.2 The Simulation Environment

AxPIKE core simulator is forked from the Spike RISC-V ISA simulator. Figure 5.1 shows the workflow to evaluate an application. The core simulator is compliant with the RISC-V ISA and runs binaries compiled using standard tools. A high-level descriptive file configures the simulation and associates instructions and data accesses with custom

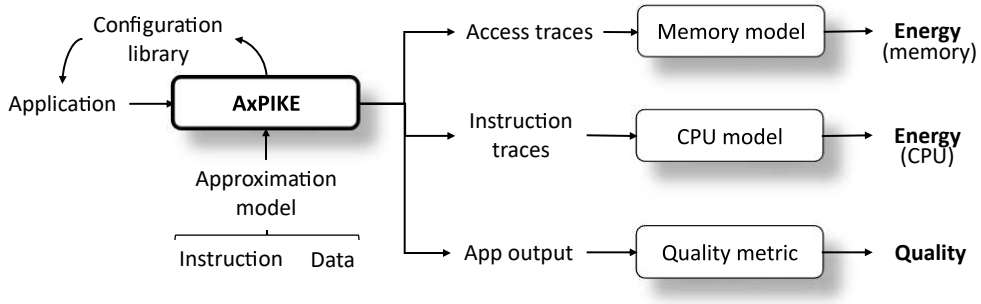


Figure 5.1: Workflow of the simulation environment. *It produces detailed execution statistics to estimate quality and energy of application execution.*

(Reused from [47] ©2021 EDAA)

approximation models [49]. A priori, the simulator does not require any changes in the application to inject approximations. However, in a common scenario in which certain structures or code regions need to be protected, we provide a low overhead configuration interface based on dedicated control registers [46]. For convenience, AxPIKE generates a software library for the current configuration of approximations, translating comprehensive function calls into the lower-level register-based control.

The execution produces the application output and simulation statistics. The output is evaluated using a scenario-specific quality metric to determine the quality of the results. By default, the statistics include memory accesses and instruction counters, such as the number of accesses and misses for each level of cache and the number of executions of each instruction. These are detailed specifying privilege level, approximation configuration, and region in the application code. Additionally, we provide a configurable statistics generator to produce custom traces, as well as other aggregated performance metrics.

5.1.2.1 Approximation Modeling and Injection

Approximation hardware is modeled as functions that can be injected into any instruction in the ISA. Also, the models can modify the data read from or written to any register or memory location. These models can receive user-defined parameters and access the full state of the simulated CPU. Moreover, the data models provide metadata about the operation, such as the source, the type of operation (read, write, instruction fetch), virtual and physical addresses, and whether there was a TLB entry for it. Thus, the models allow for flexible and extensible instruction-level instrumentation of the approximated scenario.

Figure 5.2 exemplifies the approximation models to implement an approximate logarithm multiplier (*ILM_EA* [8]) and to inject errors into data read from the DRAM main memory (*AxRAM* [38]). The *word* datatype holds references to custom parameters that assume the size of a data word in the simulated CPU. In this multiplier scenario, they will refer to the respective registers in the register bank. The DRAM approximation emulates a scenario in which the supply voltage of the DRAM is scaled below common guardbands, subjecting the data to errors modeled as uniformly distributed bitflips. Supervisor and machine privilege code (e.g., the Operating System) run in protected memory regions and are thus exempt from these errors.

```
IM ILM_EA(word a, word b, word r, processor_t* p) {
  // r ≈ a * b
  Reinterpret(a = m1 + q1, where m1 = 2^k1);
  Reinterpret(b = m2 + q2, where m2 = 2^k2);
  r = 2^(k1+k2) + q2*2^k1 + q1*2^k2;
}

DM AxRAM(processor_t* p, source_t* source, void* data) {
  if (source->hierarchy == DRAM &&
      p->get_state()->prv == USER) {
      UniformBitFlip(1.4E-5, data);
  }
}
```

Figure 5.2: Sample approximation models. *They represent instruction behavior (IM) or changes in data accesses (DM).*

(Reused from [47] ©2021 EDAA)

```
IM ILM_EA(word a, word b, word r);
DM AxRAM();

approximation Approx_ILM_EA {
  instruction mul {
    alt_behavior = ILM_EA(FETCH_RS1, FETCH_RS2, FETCH_RD);
  }
}

approximation Approx_AxRAM {
  mem_read = AxRAM();
}
```

Figure 5.3: Sample approximation configuration. *Approximation models are distributed into approximation states that can be switched off and on dynamically.*

(Reused from [47] ©2021 EDAA)

A configuration file associates the models with instructions and data accesses [49]. Figure 5.3 illustrates the configuration to inject the multiplier and the DRAM models. The configuration starts with declaring the models, in which the metadata (`processor_t* p`, `source_t* source`, and `void* data`) are implicitly assumed and referenced when processing the configuration. The multiplier model uses the attribute `alt_behavior` to replace the behavior of the multiplication (`mul`) instruction and receives its operands. Similarly, a model could be injected before (`pre_behavior`) or after (`post_behavior`) the instruction execution. The DRAM model is injected when reading data from memory (`mem_read`), regardless of the instruction. The error could also affect write accesses (`mem_write`) and other storage structures such as the register bank (`regbank_[read/write]`).

axpike_iface.h :

```
#define Approx_ILM_EA 1
#define Approx_AxRAM  2

#define axpike_activate(approx) \
  asm volatile ("csrrs %0, urkst, %1" :: "rK"(approx));

#define axpike_deactivate(approx) \
  asm volatile ("csrrc %0, urkst, %1" :: "rK"(approx));
```

matrix_multiply.c :

```
#include "axpike_iface.h"
#define SIZE 100

int main(int argc, char* argv[]) {
  int A[SIZE][SIZE], B[SIZE][SIZE], C[SIZE][SIZE];
  int i, j, k;

  read_inputs(A, B);
  axpike_activate(Approx_ILM_EA);

  for (i = 0; i < SIZE; i++) {
    for (j = 0; j < SIZE; j++) {
      C[i][j] = 0;
      for (k = 0; k < SIZE; k++) {
        C[i][j] += A[i][k] * B[k][j];
      }
    }
  }

  axpike_deactivate(Approx_ILM_EA);
  write_output(C);
  return 0;
}
```

Figure 5.4: AxPIKE control interface. *Software communicates with the simulator using control registers via a dedicated software library.*

(Reused from [47] ©2021 EDAA)

The models are grouped together into higher-level *approximations*. An approximation is an entity that defines a state of the simulated system, whose behavior can be described by multiple models injected into multiple instructions. For example, an approximate multiplier could also affect multiply-and-accumulate instructions, in which case the software model differs but would be grouped in the same *approximation*. Approximations can be enabled or disabled at execution time, defining when the models should be injected.

5.1.2.2 Software Control Interface

The simulator offers a control interface that allows the simulated software to take control over the approximations using a control interface based on the CSRs proposed in Chapter 4. Figure 5.4 shows an example in which the main computation kernel of matrix multiplication uses the *ILM_EA* approximate multiplier [8]. The interface library

Table 5.1: Statistics collected for Matrix Multiply

Instruction statistics			
Instruction	**Prv.**[1]	**Approximation**	**Counter**
– All –	M/S	None	52,100,132
		Approx_ILM_EA	21,383
	U	None	21,036,963
		Approx_ILM_EA	7,082,321
Multiplication[2]	M/S	None	0
		Approx_ILM_EA	0
	U	None	93,143
		Approx_ILM_EA	1,000,000
Memory statistics			
Access level	**Prv.**[1]	**Approximation**	**Counter**
L1 cache	M/S	Approx_AxRAM	1,451,651
	U	Approx_AxRAM	8,373,444
L2 cache	M/S	Approx_AxRAM	52,308
	U	Approx_AxRAM	244,016
DRAM	M/S	Approx_AxRAM	51,540
	U	Approx_AxRAM	195,925

[1] Privilege: M = machine, S = supervisor, U = user

[2] Includes the whole integer multiplication family

(`axpike_iface.h`) is automatically generated by the simulator and defines the functions to control the approximations. Approximations are activated by setting the corresponding bits in the `urkst` control register, which defines their status for any code running in user privilege level. Similarly, the interface allows approximations to be controlled by machine and supervisor code, and the state can be controlled by a supervisor system.

5.1.2.3 Statistics generator

To support performance and energy evaluation, AxPIKE provides detailed data about the execution. A configurable statistics generator allows designers the creation of custom performance counters and traces. For example, it can create detailed memory traces to feed external tools for energy estimation and log operands and results of every instruction to produce activity factors for RTL simulation. Collecting statistics represents a significant overhead in simulation execution. For example, for the simple Matrix Multiply case, tracing every memory access almost doubles the execution time and produces a log file of more than 40X the size, compared to tracing only accesses in the main DRAM memory.

Even when the customized statistics class is not used, some aggregated instruction and memory access counters are produced. Table 5.1 exemplifies the default counters for the Matrix Multiplication application. The counters are detailed for each instruction in the ISA and level in the memory hierarchy. They also provide information on which was the privilege level of the execution and the active approximations at the time. In the example, some of the multiplication instructions are executed in the approximate multiplier [8] and only unprotected memory accesses are subjected to the DRAM error model [38].

Table 5.2: Quality and Energy evaluation of selected applications subjected to approximation

	ILM_EA		**AxRAM**	
Application	Quality	Energy	Quality	Energy
Matrix multiply	79.85%	99.62%	95.17%	90.46%
Sobel	79.55%	98.30%	95.12%	89.58%
JPEG	97.68%	99.62%	85.83%	88.53%

(Reused from [47] ©2021 EDAA)

Table 5.3: Comparison of simulation performance under different configurations

Application	**Spike**	**AxPIKE**	**ILM_EA**	**AxRAM**
Matrix multiply	22 s	61 s	61 s	63 s
Sobel	8 s	23 s	23 s	23 s
JPEG	50 s	158 s	160 s	158 s

(Reused from [47] ©2021 EDAA)

5.1.3 A sample usage case

To illustrate how the simulator operates, we selected the applications Matrix multiplication, Sobel, and JPEG to evaluate the approximate logarithm multiplier [8] and approximate DRAM access [38] scenarios. Both scenarios are exclusive. Matrix multiply operates on two 100x100 integer matrices, Sobel detects edges in a 256x256 image, and JPEG compresses a 512x512 bitmap. For each application, we compute the quality of results and estimate the energy relative to an execution that does not employ an approximation. In our experiments, the applications are compiled as Linux ELF files, and the OS behavior is emulated by the RISC-V Proxy Kernel, running on a single-core RV64g CPU, configured with independent 32 KB instruction and data L1 caches and a single 128 KB L2 cache.

Table 5.2 summarizes quality and energy results. The quality metrics are the Mean Relative Error for Matrix multiply and the Structural Similarity Index for Sobel and JPEG. To calculate the DRAM energy, we configured the simulator to produce traces of every DRAM access, which were then forwarded to Ramulator [73] and DRAMPower [21] for estimation. The DRAM energy and error models were derived from experimental data from Chang *et al.* [24]. We consider that both the CPU and the DRAM contribute equally to the overall system energy cost, and that multiplication accounts for 8.5% of the energy cost of the CPU [144]. In the multiplication scenario, only the main computation kernel can be exposed to the approximation, which restricts energy savings. In the DRAM scenario, on the other hand, the entire application can be subjected to approximation, except for some small protected memory regions that store mostly control structures. Thus, approximating the memory accesses shows a wider margin to improve energy efficiency.

Producing simulation statistics and injecting the approximations imposes overhead in the base Spike simulator. Table 5.3 shows the execution time for each of the applications in the original Spike simulator, without injecting any approximation nor producing additional statistics, and each of the approximation scenarios. Although the approximated scenarios required three times longer to compute results, they represent a richer evaluation allowing approximation control and producing extended statistics.

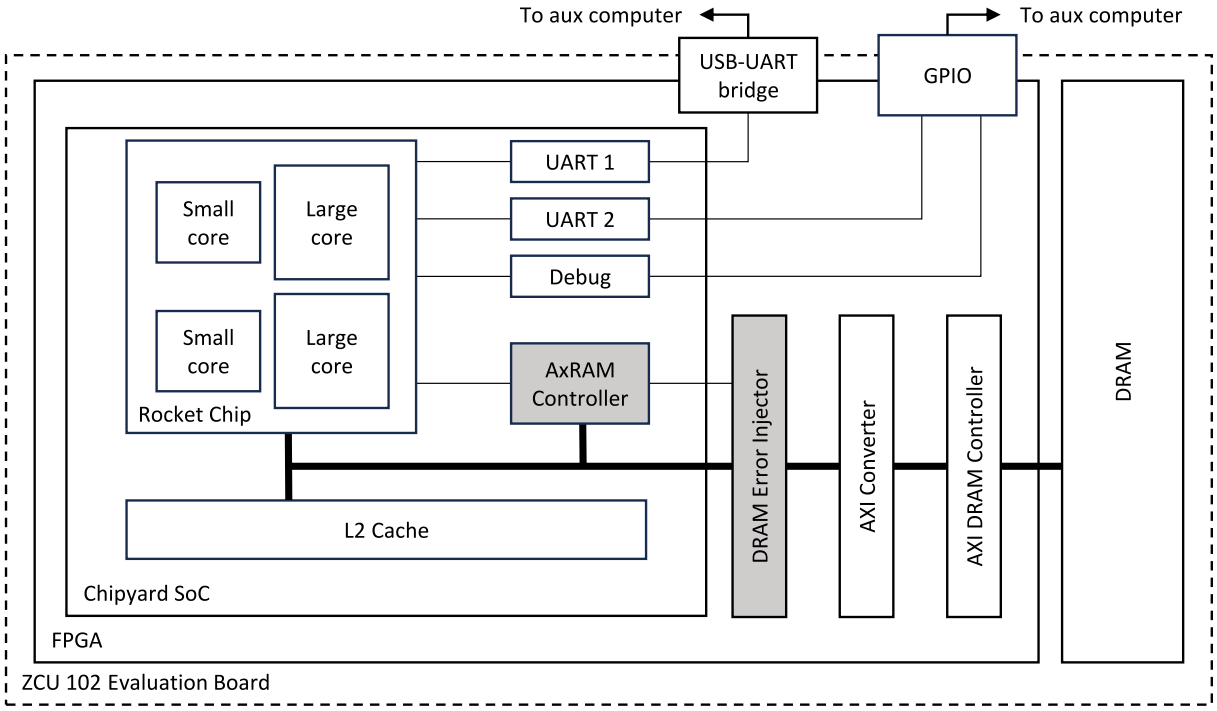
5.2 FPGA-based Full Approximate System Prototype

The aforementioned AxPIKE simulator is very versatile in representing applications when executed in an Approximate Computing scenario. It allows for rapid prototyping since the approximation modules are translated as straightforward C-language models that replace the original behavior of instructions. It requires little-to-no modification in the original application source code, depending on the scenario to be represented, since it supports loading Linux applications and forwarding environment calls to the host system. However, accounting for cache behavior and generating simulation statistics significantly affects simulation performance, jeopardizing the ability to represent more complex systems. Moreover, the functional simulator abstracts and may not correctly account for organization-level design choices, even when they affect the overall application behavior, such as the pipelined organization, exception handling, branch prediction, and cache organization. To introduce a new perspective on how a general-purpose approximate processor behaves, we built a complete system prototype synthesized for an FPGA.

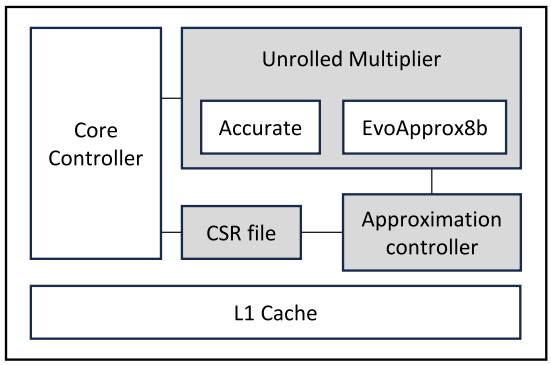
The system is based on the Chipyard framework [7] and is synthesized for the Xilinx ZCU102 Evaluation Board. The board is powered by a Zynq UltraScale+ SoC, which embeds an ARM Processor with the FPGA in the same chip. The ARM Processor was not used in this project, which significantly limits the available peripherals and connectivity in the evaluation board. However, by using only the FPGA side of the SoC, we demonstrate the approximate processor as a standalone system, instead of as an accelerator dependent on another main processor.

Figure 5.5 shows the general architecture of the FPGA prototype. The Chipyard system features a Quad-Core Rocket Processor, where each core received the internal approximation controller and five approximate multipliers derived from the EvoApprox8B library [96]. In the classification introduced in Chapter 2.1, the multipliers are non-configurable units and, as such, the execution control relies on selecting which of the multiplication modules receives the operands and provides the product in the particular operation. Outside the core, another source of approximation was introduced to emulate a voltage-overscaled DRAM module. An error injector sits between the main memory port and the external DRAM controller and causes bit-flips in data read from the DRAM according to a configurable probability in a uniform distribution [24]. The error injector exposes a configuration interface, as memory-mapped I/O, that allows protecting certain memory regions and defining the probability of error [38]. The system communicates with the external world via two UART interfaces and a debugging JTAG interface, which can be used to load an application, control execution, and send and receive data.

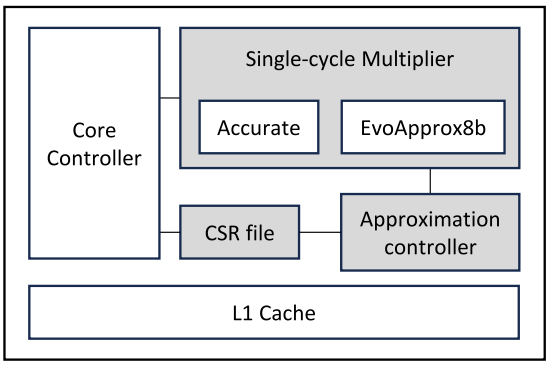
To allow application execution, the prototype uses SiFive's Freedom E Software Development Toolkit (SDK). The SDK includes Freedom Metal, a Hardware Abstraction Layer built to interface SiFive's hardware, including hardware with Chipyard, with standard software tools. Our application code is a multithreaded software that receives data to be processed, along with information about the target application, from an auxiliary computer using UART communication lines, runs the target application, and returns the application result back via the UART lines. In this environment, we show how the proposed architecture behaves in a multi-application multicore scenario.



(a) Overall architecture



(b) Small core



(c) Large core

Figure 5.5: Architecture of the hardware prototype. *The prototype is a multi-core SoC implemented in a FPGA. Gray modules were created or modified in this thesis.*

5.2.1 Hardware Workflow

The Chipyard framework encapsulates a series of RTL generators and auxiliary synthesis and simulation tools that can help designers in the hardware development flow of RISC-V-based systems. It includes code for processing cores, intra-chip communication modules, and peripherals that can be joined together as described in configuration files. The customization options, however, are primarily designed towards reusing the available modules as they are, with little guidance on how to customize the internals. Our design requires changes in the Register File and internal communication buses to allow the register-based control described in Chapter 4. Moreover, Chipyard's support to bench FPGA prototyping is rudimentary and largely undocumented, as the targets moved to relying on cloud-based providers. However, Chipyard groups the reference implementation, by the creators of the RISC-V Open Standard themselves, of various aspects of the archi-

Table 5.4: Resource utilization of FPGA Baseline implementation.

Component	**LUTs**		**Registers**		**Block RAM**		**DSPs**	
available	274,080		548,160		912		2,520	
CPU	142,864	52.1%	68,755	12.5%	210	23.0%	60	2.4%
Core (Each of 4)	31,537	11.5%	15,501	2.8%	43.5	4.8%	15	0.6%
FPU	12,383	4.5%	3,824	0.7%	–	–	11	0.4%
Multiplier	1,721	0.6%	214	0%	–	–	4	0.2%
CSR File	2,365	0.9%	1,081	0.2%	–	–	–	–
L1 I-Cache	1,929	0.7%	1,235	0.2%	10	1.1%	–	–
L1 D-Cache	3,565	1.3%	2,279	0.4%	32	3.5%	–	–
L2 Cache	9,331	3.4%	3,399	0.6%	36	3.9%	–	–
DRAM Controller	8,492	3.1%	9,472	1.7%	25.5	2.8%	3	0.1%
AXI Clock Converter	446	0.2%	1,232	0.2%	–	–	–	–
Total	**152,291**	**55.6%**	**80,467**	**14.7%**	**235.5**	**25.5%**	**63**	**2.5%**

tecture, such as the Rocket Chip Generator [9], the Berkeley Out-of-Order Machine [146], the Hwacha vector extension [80], the Gemmini accelerator [51], and others. Thus, building our prototype in this framework, despite the design and development effort, prepares the environment for future compatibility with these standard and third-party tools.

5.2.1.1 Base ZCU102 Support

The FPGA prototype was developed as a standalone system. That is, even though it may communicate with an external system using standard communication interfaces, it should not be dependent on another CPU for its base functionality. The base support for the ZCU102 board was derived from the existing Chipyard support to the Diligent Arty A7-100T, a significantly smaller and less resourceful board but similar in features and FPGA-connected peripherals to the larger ZCU102. The base support comprises the main CPU, two levels of cache, an external DRAM controller, two UART communication ports, and JTAG debugging. The main CPU is a Quad-Core RV64 Rocket CPU, at which each core consists of a single hardware thread in a 5-stage in-order pipeline, augmented with hardware multiplication and division, atomic extensions, and floating-point support. Each core is connected to split instruction and data L1 caches of 32 KB each. All cores share a single 128 KB L2 cache. These settings are derived from the Rocket "Big Core" configuration and were adapted for compatibility with both the Arty FPGA support and experimental settings with the AxPIKE simulator.

The ZCU102 exposes to the FPGA a 512 MB DDR4 memory compatible with the Xilinx DDR4 Controller IP. The DDR4 Controller can communicate with user logic (in this case, the Chipyard implementation) using a standard AMBA AXI4 interface, which is also supported as the external memory port interface in the Rocket Chip configuration. Both sides of the channel, however, operate at different clock domains, and thus an AXI Clock Converter was needed. In terms of communication, the board exposes a single channel of a USB-UART bridge to user logic, out of the four existing in the communication module. Also, the USB-JTAG port is not readily available. Thus, we took advantage of the GPIO pins to enable a second UART port and the JTAG port. Both are connected to external devices to allow communication with an auxiliary computer. The auxiliary computer transfers the needed application and data for experimentation.

Table 5.4 shows the area of the baseline implementation, expressed in terms of resource utilization in the FPGA. The percentages are relative to the total number of each resource instance available in the FPGA. In particular, we highlight the internal utilization of the internal CPU Control and Status Registers Files and hardware multipliers, as these are the components that need modification to include the approximations. The CSRs include the configuration registers defined in our ISA extension, as described in Chapter 4, and the multiplication hardware is replicated to allow the inclusion of non-configurable approximate multipliers [96].

5.2.1.2 Approximation controller

The Approximation Controller is the hardware module that enables the architecture to recognize and configure the status of different approximation sources. The configurations are derived from the Control and Status Registers defined in Chapter 4, which are decoded in the controller to determine which approximations are active at any given point in the execution. The CSR Register File in each core was modified to include references to the registers specified in the ISA extension, where any read or write operation within these registers is forwarded to the Approximation Controller. The controller itself sits next to the Register File and is based on the concept described in Figure 4.5, Chapter 4 to store the control registers and translate their data into the necessary configuration signals.

The ISA extension proposed in this thesis considers that approximations may require some time to be activated or deactivated, such as to allow power gating of unused components. Because of that, the extension introduces configuration registers that indicate what is the general behavior of the processor execution during this time. In our behavioral FPGA implementation, however, power gating is not achievable and, in practice, the activation and deactivation delay is irrelevant in comparison to a CPU cycle time. Therefore, the behavior registers (`mrkacbhv` and `mrkdcbhv` in Table 4.1) are irrelevant in the implementation and were ignored.

5.2.1.3 Approximate Multipliers

The FPGA prototype was configured to include partial Approximate Multipliers selected from the EvoApprox8b library [96]. The selection of multipliers was derived from our previous work on the ADeLe language [49]. The design considers how the individual multiplication modules behave when extended from their original 8-bit length to wider 32-bit multipliers, in a Wallace Tree organization, in terms of energy and error, as shown in Figure 5.6. Energy was accounted for a 45 nm fabrication process in Cadence Design and Synthesis tools, and quality is the mean relative error, compared to an accurate multiplier, for an extensive uniform multiplication dataset. From these, we selected five multipliers on the Pareto frontier that show the lowest energy cost at different error levels. Specific error and energy metrics for each selected multiplier are detailed in Table 5.5.

The Rocket Chip implements a multicycle “unrolled” multiplier, where a full-length multiplication operation is divided into a series of multiplications of narrower data words. Each partial multiplication is calculated in a hardware multiplier and accumulated to build the final result. Our approximate implementation replicates the partial hardware

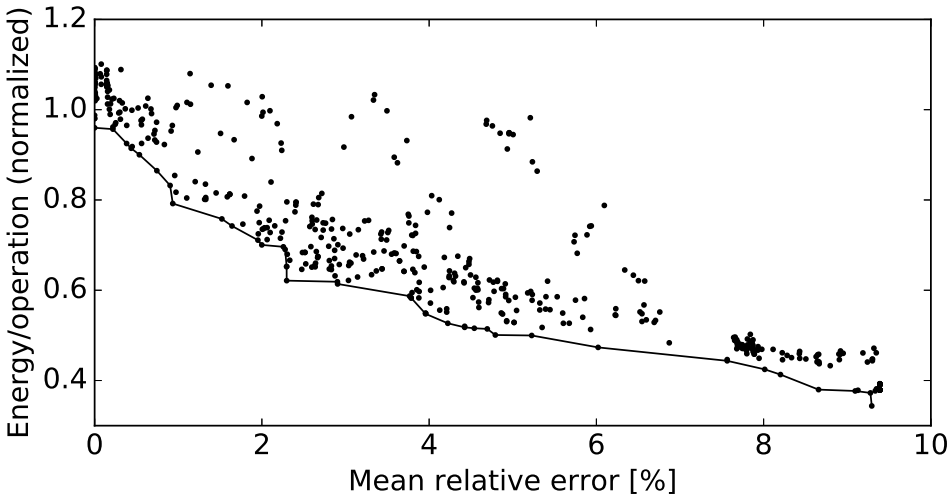


Figure 5.6: Energy-quality trade-off individual EvoApprox8b multipliers. *Widening operand and product data words affects both energy and error, and this analysis can be used to select the most applicable multiplication instances.*

(Reused, with permission, from [49] ©2020 Elsevier)

Table 5.5: Selected EvoApprox8b multipliers.

Multiplier	**Mean relative error**	**Relative energy**
Accurate	0.00%	100.0%
mul8_303	0.22%	95.68%
mul8_469	0.93%	79.19%
mul8_479	2.00%	70.05%
mul8_423	5.23%	49.97%
mul8_279	9.39%	39.30%

(Reused, with permission, from [49] ©2020 Elsevier)

multiplier including each of the five EvoApprox8b options side by side. The unit that performs each partial operation is selected according to the configuration indicated by the Approximation Controller. This implementation introduces an area and static power overhead, due to replicating hardware units and power gating being impractical. The impact on dynamic power was mitigated by keeping the operand inputs of each individual partial multiplier constant when not in use, which reduces the activity factor and translates to what would be expected from clock gating.

The multi-cycle unrolled method taken by Rocket Chip to perform wider multiplication operations is only coherent for accurately computed partials. This means that when each partial multiplication is taken from an approximation unit, the accumulated result may not be, and most likely is not, the same as expected if the partial were taken from the wider operands in a larger single-cycle multiplication unit. This situation adds a second level of approximation in the operation and may severely affect the overall error rate and quality of the results. This illustrates precisely the reason for considering the integration of individual approximation units into a larger system at design time. The unrolled multiplier is efficient in terms of area and energy for accurate multiplication, but when considering approximate partials the expected results are only achievable in a much

larger and more energy costly single-cycle multiplier. In our FPGA implementation, two of the four cores were configured to use the original unrolled multiplier with replicated approximated partials, and the other two implement single-cycle multipliers derived from each narrower EvoApprox8b unit. This difference in implementation has a significant impact in the final resource utilization in the FPGA and efficient mapping of the design in LUTs and DSP slices, as discussed later in Section 5.2.1.5.

5.2.1.4 DRAM Error Injector

The second source of approximation in the system is the main memory. The prototype simulates the behavior of a voltage-scaled DRAM device and implements a protection interface to control which memory regions are exposed to error and at what level [38]. However, since scaling the supply voltage in the single-chip DDR4 module in the ZCU102 board is not practical, we introduce an artificial error injector that sits between the DRAM controller and the CPU external memory port.

The error injector acts as a peripheral and communicates with the CPU via two memory-mapped registers, `dram_a` and `dram_x`, as exemplified in Figure 4.4, Chapter 4. Register `dram_a` configures the error rate introduced in data accesses, which is translated from the supply voltage level [24], while register `dram_x` selects which memory regions are affected. Each memory region is a fixed and predetermined set of contiguous physical addresses in the memory array that can be controlled independently. Errors are injected in the form of a single bitflip at a random position of the data read from memory into an L2 cache block. A pseudorandom number extracted from a Fibonacci Linear Feedback Shift Register (LFSR) is used to both determine whether a memory access is affected, according to the probability determined by the configuration register, and which bit in the data word should be flipped. The LFSR was imported from a package of common hardware components package available in the Chisel Hardware Description Language.

5.2.1.5 Area overhead

The prototype implementation required additional hardware modules and modifications to existing ones. In particular, the extension is exposed to the application software via Control and Status Registers, which were included among the existing CSRs. The Approximation Controller coordinates the activation and deactivation of the approximation modules and was implemented as a new component in each core. Including the approximate multiplication operator required changes in the existing multiplier organization, besides replication of partial multiplication units. Finally, simulating the voltage-scaled DRAM module included an additional peripheral and, similarly, another component would be required to control an actual approximate memory. All these components impact the area of the final system.

Table 5.6 details the area overhead of the final system compared to the baseline implementation (Table 5.4), in terms of FPGA resources utilization. For each hardware component, we show the absolute value (left column) and the impact on the overall resource utilization in percentage points over the baseline (right column). The “Small Core” uses the original multicycle unrolled multiplier, while the “Large Core” had the hardware

Table 5.6: Overhead introduced in the FPGA implementation.

Component	**LUTs**		**Registers**		**Block RAM**		**DSPs**	
available	274,080		548,160		912		2,520	
CPU	227,170	+30.8	70,162	+0.3	210	–	44	-0.6
Small Core (Each of 2)	38,891	+2.7	15,722	–	43.5	–	11	-0.2
Approximation controller	266	+0.1	222	–	0	–	0	–
Multiplier	8,377	+2.4	233	–	0	–	0	-0.2
CSR File	2,728	+1.0	1,081	–	0	–	0	–
Large Core (Each of 2)	66,184	+12.6	15,902	+0.1	43.5	–	11	-0.2
Approximation controller	1,019	+0.4	222	–	0	–	0	–
Multiplier	35,222	12.2	413	–	0	–	0	-0.2
CSR File	2,460	+0.9	1,081	–	0	–	0	–
Common (All 4 cores)								
FPU	12,393	–	3,824	+0.7	0	–	11	–
L1 I-Cache	765	-0.4	1,235	–	10	–	0	–
L1 D-Cache	3,567	–	2,277	–	32	–	0	–
L2 Cache	9,408	–	3,399	–	36	–	0	–
DRAM Controller	8,480	–	9,472	–	25.5	–	3	–
DRAM Error Injector	360	+0.1	582	+0.1	0	–	0	–
Fibonacci LFSR	182	+0.1	64	–	0	–	0	–
AXI Clock Converter	447		1,232		0	–	0	–
Total	**238,926**	**+31.6**	**82,328**	**+0.3**	**235.5**	**–**	**47**	**-0.6**

multiplier replaced with a single-cycle unit within the pipeline. The distribution of resources during the Place & Route phase of synthesis can cause more or fewer elements in the FPGA to be allocated to the same component, which explains the small variations perceived in the instruction cache and the floating-point unit. This also happens with different instances of the same component, such as the CSR files and the approximation controller between the small and large cores. On average, the most significant impact of introducing the extension itself, disregarding the specific multiplication and DRAM approximation units and considering only the additional control registers and the approximation controller, adds 1.2 p.p. area overhead per core. In absolute terms, this is comparable with the area of the original CSR File.

The more considerable impact comes from the approximation sources themselves, in particular the multiplication units. The modified multiplier organization, which replaces a wider abstract multiplier with an explicit Wallace tree organization of narrower ones, forces the synthesis tool to build the logic using look-up tables instead of allocating the existing DSP slices in the FPGA, which itself impacts resource utilization. The more reliable multiplication unit in the large core is substantially more costly than the original multiplier because the previous multi-cycle organization was not compatible with the partial approximate multipliers. However, the multiplication unit in the small core uses less than 5 times the number of LUTs in comparison with the original one, and yet encapsulates 6 different multipliers – the accurate and five approximate ones – an impact lower than the inclusion of a floating-point unit, for example. This highlights that, had the approximate multiplier units been designed for architecture-level integration, considering the error propagation within the design [19, 20], they would have caused a more favorable energy-quality trade-off, as translated from area and error rates.

Table 5.7: Software tools in the prototype framework.

Tool	Version
SiFive Freedom E SDK	20.05.01.00 dev
SiFive RISC-V64 Bare Metal Toolchain	10.2.0-2020.12.8
SiFive Freedom Binutils	
SiFive Freedom GCC	
SiFive Freedom GDB	
SiFive Freedom OpenOCD	0.10.0-2020.12.1

5.2.2 Software Framework

Chipyard's Rocket Chip implementation contains a small boot ROM with code to wake the processor up from a halt state. This boot procedure, however, does little more than looping forever waiting for an external interrupt. Since there is no access to a boot device in the implemented prototype, any application software needs to be sideloaded from external sources. In our prototype, we use the JTAG port and an external hardware debugger to load software. Rocket Chip implements the RISC-V ISA Open Standard, and thus it is compatible with *de facto* standards to build and debug applications. Table 5.7 details the software tools and respective versions, available from SiFive.

To create application software, we rely on the Hardware Abstraction Layer and compatibility libraries provided by SiFive in the Freedom E Software Development Kit. The SDK is intended to allow the development of software targeting SiFive's embedded RISC-V platforms, which included[1] a ported development board based on the Arty FPGA Evaluation Kit, which is an earlier generation of the Arty A7-100T board on which we based our hardware prototype implementation. The Freedom Metal compatibility library, included in the SDK, provides routines and underlying drivers to support features such as UART interfaces, serial terminals, software locks, and general I/O, as well as a base implementation of common Linux system calls. The proper drivers and libraries are recognized and configured for the FPGA target according to the description imported from the Device-tree, which is automatically generated from the Chipyard configuration. Thus, following the same principles as in the available Arty target, we created a new ZCU102 target that matches our prototype implementation.

The target software is a multi-threaded data loader and processor, distributed through the four cores, based on a producer-consumer fashion. The scheme in Figure 5.7 shows this organization. Two of the four threads manage data acquisition, data delivery, and control. The other two are worker threads that execute the target application, selectively controlling how they are exposed to the multiplier approximation. The data acquisition and delivery threads communicate with an auxiliary computer through the two UART interfaces. The more reliable port, connected to the USB-UART bridge in the board, was configured at a faster 1 MBd rate[2] and transfers the main application data. The other port, connected through the GPIO pins and external devices, serves the serial terminal for message exchange and control flow at the standard 115.2 KBd.

[1] SiFive discontinued the Freedom project repository and FPGA targets as of March 2021.

[2] 1 MBd = 1 megabaud, roughly equivalent to 1 megabit/s in this scenario.

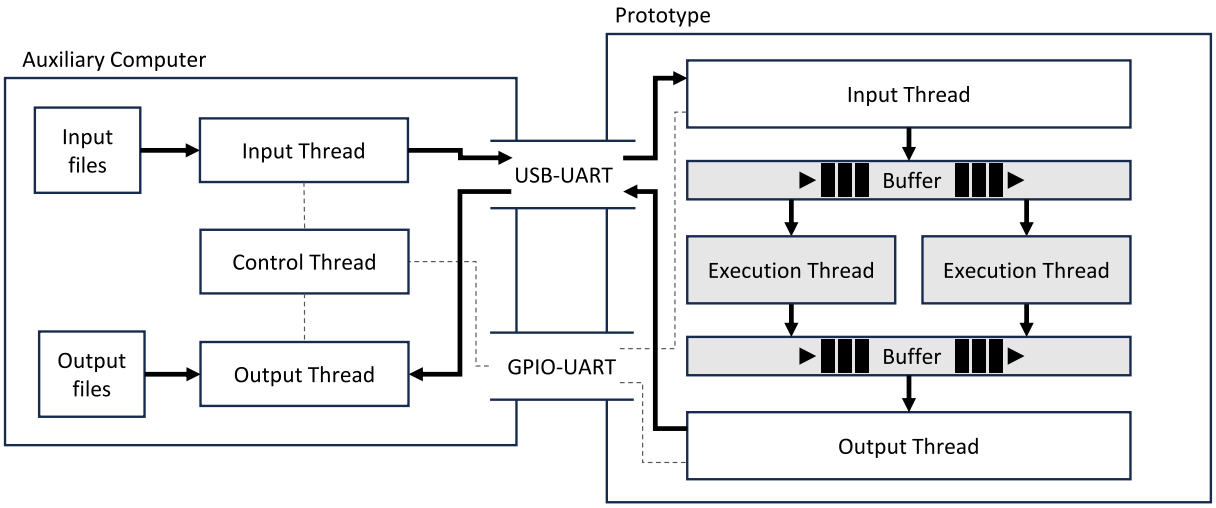


Figure 5.7: Software framework to support the hardware prototype. *A multi-threaded framework consumes input data and executes the target applications. Gray blocks are subjected to approximations.*

The memory array is divided into two contiguous regions: one small accurate region and a large region subjected to the DRAM approximation, which is injected by the hardware error injector. The two main data buffers are allocated in a "data blob" in approximable memory and are used to store raw application data. Thus, the dynamically allocated application data is stored separately from other control structures and local variables, which are usually less resilient to the noise of the approximation [42]. Space in the data blob is requested by the data acquisition thread, to store input data, and by the worker threads, to use as working memory and store output data, using specialized memory allocation functions. Requested memory regions are reserved noncontiguously in the approximable memory region to simulate the behavior of an Operating System mapping memory for multiple concurrent applications and to mitigate the impact of the cache by breaking data locality.

Unlike the AxPIKE Simulator implementation (Chapter 5.1), which could run Linux-compiled applications virtually unchanged due to redirecting system calls to the host system, target applications in the hardware prototype require some more extensive adaptation. First, support to system calls and the C library is limited, thus file and serial terminal I/O need to be replaced with direct UART handling, with secondary software running in the auxiliary computer to translate the raw data. Additionally, memory allocation must be analyzed and directed to the approximable memory when applicable, effectively selecting what data can be exposed to errors. Finally, every shared-memory and global variables need to be accounted for to allow the multi-threaded execution of the execution cores. These changes are needed to make the application compatible with the bare-metal execution model in the software framework, in its current implementation.

Despite this need for further adaptation, the prototype is a real approximate system and provides higher performance in the experimentation. In Chapter 6, we demonstrate the proposed ISA extension and its two levels of implementation using the simulator to produce more detailed energy results and the prototype to represent a scaled-up scenario.

Chapter 6

Experimentation

Evaluating the Approximate Computing integration

We demonstrate the proposed control interface in operation and how different approximation techniques interact with each other in both the functional AxPIKE simulator and the hardware prototype. Both demonstration scenarios were constructed to show our ISA interface managing non-configurable replicated hardware multipliers and an externally-configurable approximate memory, considering the particularities of each environment in designing the experiments. The environments were configured to target an RV64g architecture, accessing the main DRAM approximate memory through independent 32 KB instruction (4-way 256 sets) and data (8-way 128 sets) L1 caches and a shared 128 KB L2 cache (4-way 1024 sets). The simulation environment can execute a wider range of applications and produces more data about each execution, and thus can base a more extensive analysis on the energy-quality trade-off of different design choices. The hardware prototype involves a higher design effort on integrating the approximation modules and adapting applications for execution and produces a limited amount of data of the execution. However, it is more representative, for being an actual implementation, and it is much faster to show how the approximation behaves when applied to a variety of different input data. For these reasons, we took advantage of the simulation results to evaluate the energy and quality of results for different applications, while the prototype results show the real integration of approximation for a single application consuming a larger amount of data, in a more limited approximation scenario.

6.1 Experiments on the simulation environment

The AxPIKE simulation environment is more versatile in representing different applications and approximations. Also, it produces richer information about each execution using various statistical counters and logs. It allows direct communication with the host computer Operating System, reducing the required effort to introduce new applications. Thus, we designed the experiments in the simulation scenario to cover more applications and a more diverse approximation scenario. The demonstration was divided into two phases, as shown in the scheme in Figure 6.1. First, we modeled a voltage-overscaled DRAM main memory to show how setting the level of approximation affects energy and quality for

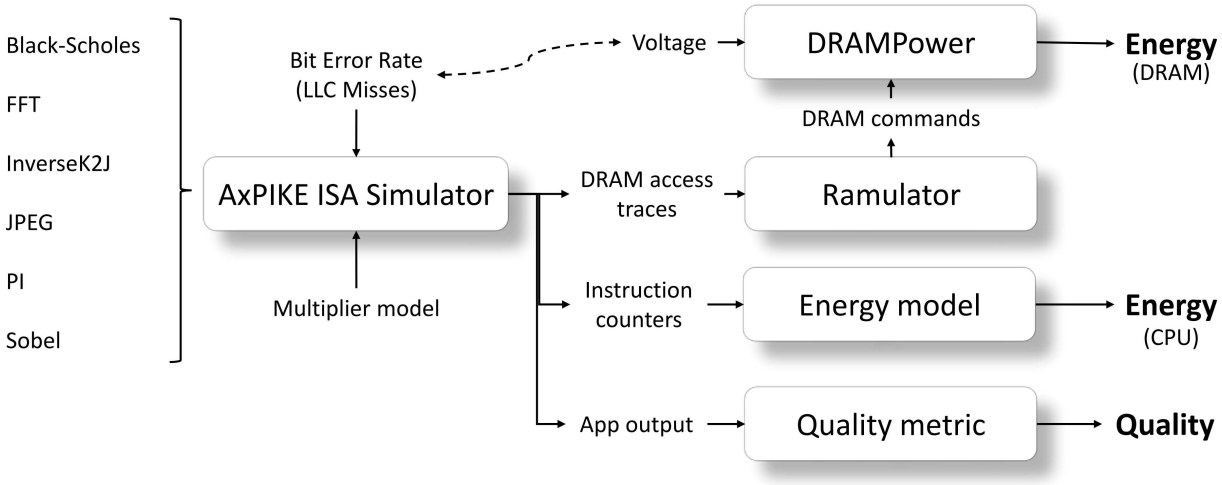


Figure 6.1: Demonstration experiments in the simulation environment. *Applications were evaluated for energy and quality under approximate multipliers and DRAM.*

different applications, also accounting for variability. This analysis highlights scenarios that can profit the most from the approximations by efficiently computing higher-quality results. Then, we selected one of these favorable DRAM approximation scenarios to introduce a second level of approximation in the multiplication operation.

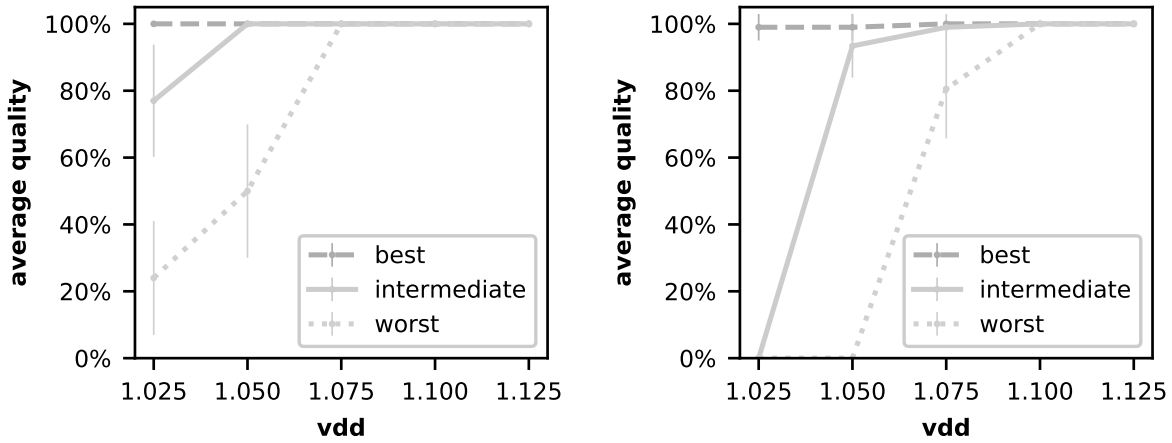
6.1.1 Configurable approximate DRAM

In our proposed interface implementation, the DRAM controller exposes two knobs that determine the bit error rate (i.e., the approximation level) and the memory regions that are free from errors [37, 38]. These knobs were set to protect machine- and supervisor-level accesses, instruction fetch, and the program stack, as discussed in Section 4.3. These controls are sufficient to manage the approximation without changes in the application. Thus, after configuration by the OS, the application is launched to run entirely and transparently in DRAM approximation mode. The simulator allows direct access to the internal registers that store information on the privilege level of execution and stack pointers. It is also possible to precisely trace which instruction triggered a given memory access. Thus, we rely on such detailed information about processor execution to implement data protection in the simulated environment, not requiring a redesign of the application or memory allocation scheme.

The probability of data being incorrectly fetched from DRAM was derived from experiments reported by Chang *et al.* [24]. The authors studied the reliability of DRAM chips by three different vendors when subjected to adjustments in supply voltage in the range from lower-than-nominal 1.025 V to nominal 1.35 V at a fixed access latency. They report the fraction of cache lines within the DRAM chip that exhibit erroneous data after stress, which can be reinterpreted as the probability of error when fetching a cache line at the given voltage level. These data exhibit high variability due to the characteristics of the memory modules. Of the three experimented vendors, two behave not friendly to voltage scaling as the errors start happening too soon, at high voltage levels, and scale up

Table 6.1: Regression of error rates in the DRAM approximation scenario.

Scenario	1.025 V	1.050 V	1.075 V	1.100 V	1.125 V
best	3.728×10^{-5}	3.016×10^{-5}	2.980×10^{-8}	0	0
intermediate	1.096×10^{-1}	6.461×10^{-4}	5.181×10^{-5}	5.811×10^{-7}	0
worst	5.638×10^{-1}	3.196×10^{-1}	1.660×10^{-3}	1.016×10^{-5}	0



(a) PI

(b) JPEG

Figure 6.2: Output quality of applications using the approximate DRAM. *All tested applications show graceful quality degradation in the intermediate scenarios of error injection.* (Reused, with permission, from [46] ©2020 IEEE)

too quickly. Thus, to account for such variability, we extracted the data from the third vendor, which exhibits the most intermediary error values across a wider voltage range between 1.025 V and 1.125 V. Then, we regressed, for each 0.025 V voltage step, the error rates that represent the *worst*-case (maximum error), the *best*-case (minimum error), and an *intermediate* (median error) scenarios. Table 6.1 shows the considered error rates for each voltage level.

For each voltage level, we execute 100 instances of applications Black-Scholes, FFT, InverseK2J, JPEG, PI, and Sobel. The applications were chosen to represent different computing domains, such as Image Processing, mathematical applications (number crunching), signal processing, and mechanical simulation, besides different access patterns to memory. The Image Processing applications JPEG and Sobel are expected to be particularly applicable in the approximation scenarios due to their perceived resilience to error, higher use of memory to store the raw image data, and reliance on multiplication during processing.

To evaluate each application behavior, we compute the quality of results in comparison to an accurately-computed output. The quality metrics essentially represent the accuracy of the application output. For the Image Processing applications, quality is the Structural Similarity Index Measure (SSIM) [10, 135] of the output images. The SSIM compares an original and a noisy image attempting to predict the perceived similarity from a human observer, where a maximum value of 1.0 (or 100%) means images are identical. For the other applications, we compute the mean relative error comparing accurately-computed

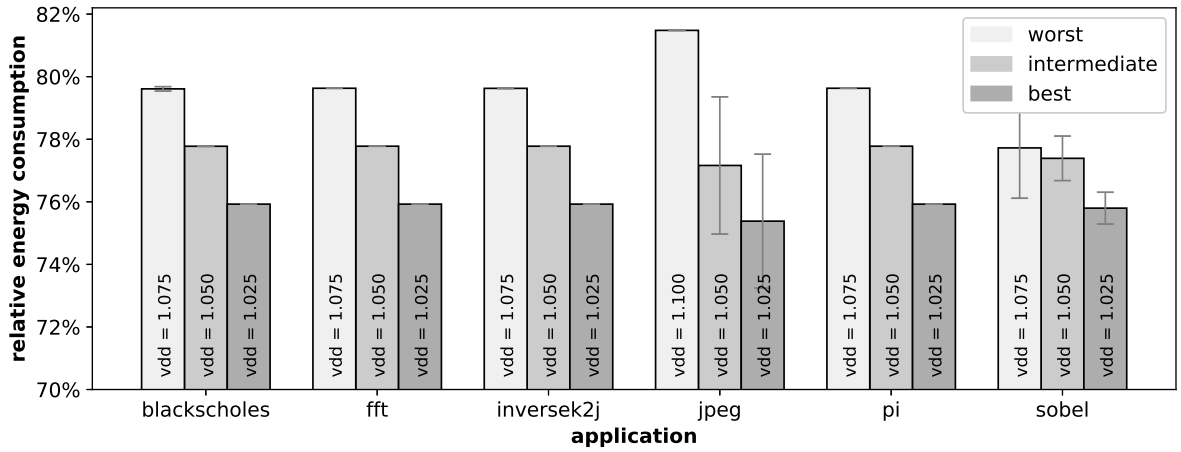


Figure 6.3: Energy to achieve 90% quality on approximate DRAM. *The DRAM approximation can provide average energy savings of 22.4% considering the intermediate scenarios.*

(Reused, with permission, from [46] ©2020 IEEE)

and approximation-affected outputs. All metrics are normalized and reported as percentages from 0% to 100%. In the event of failed execution that produces no valid output, the quality metric is considered as 0%.

Figure 6.2 shows the average quality and the 95% confidence interval of results for PI and JPEG, which are the two applications that presented the best and the worst resilience to the injected errors, respectively. From the evaluated applications, PI is the less memory-intensive one, so its high resilience to errors injected in DRAM memory is the expected behavior. As for JPEG, even though Image Processing applications are usually considered more resilient due to human perception of results, in the case of compression the injected noise can affect greater regions of the image during encoding. All other applications presented a similar pattern of degradation as voltage levels scale down. Even so, in the intermediate case, all applications achieve quality of more than 90% in the 1.05 V level. This means that considering the evaluated scenario, selectively and controllably varying the voltage levels from the nominal 1.35 V down to 1.05 V can potentially lead to energy savings maintaining an expected level of quality of results, indicating a reasonably wide margin of controllability of approximations.

To understand these savings, we evaluate the energy cost of each execution instance based on traces of DRAM accesses after each L2 cache miss in the simulation. For every execution of each application at each voltage level, these traces were filtered to include only DRAM user-level approximate accesses. Then, they were fed to Ramulator [73] to generate DRAM commands, based on a DDR3 module compatible with the error rate data (1600 Mhz, 64 bit, 8 banks, 2 ranks, 1024 columns, 8 bytes burst length, 16384 rows, 1.35 V Vdd, 13.75 ns tRCD, 13.75 ns tRP) [24]. These commands are used as input of DRAMPower [21] to get the energy cost according to supply voltage changes.

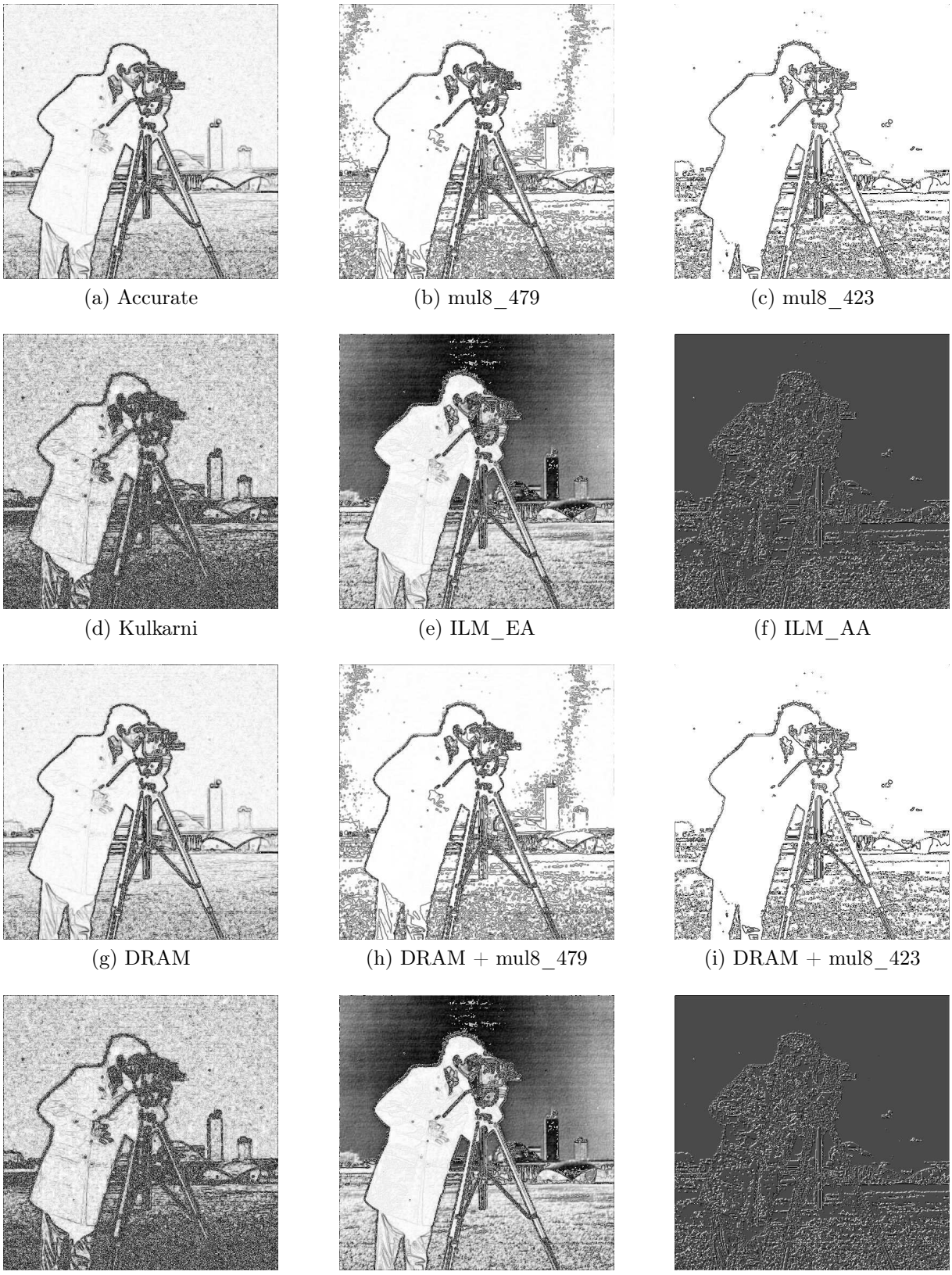
Figure 6.3 shows the memory energy cost of executing each application at the lowest voltage level that exhibits average quality higher than 90% for the best-case, worst-case, and intermediate scenarios, normalized to an execution at the nominal 1.35 V voltage.

Energy savings range from 18.6% (JPEG, worst-case scenario) to 24.6% (JPEG, best-case scenario). All applications are degraded by the injected approximation, and all applications can profit in the form of energy savings. The most efficient voltage level in the intermediate scenario is 1.05 V, expected to provide 22.4% energy savings, on average, considering all applications. In the best- and worst-case scenarios, the optimal voltage scales one step down or up, respectively, for all applications except JPEG. This is consistent with the raw error rates considered (Table 6.1), indicating that the applications can tolerate errors up to the order of 10^{-4} in these scenarios, which is equivalent to one access affected by noise for each circa 10 thousand LLC misses. In the other variability scenarios, however, these error rates are characteristic of different voltage levels. Also, they are only sustainable considering data access protection to avoid errors in critical data and application crashes [38,42]. Account for variability and protection highlights the need for configurability of the approximation at the architecture level.

6.1.2 Non-configurable approximate multipliers

The quality and energy characterization points to operating conditions at which an application, although error-prone, is expected to produce high-quality results. In the intermediate scenario, this operating point is 1.05 V, at which the Sobel application, for example, shows a memory energy cost of 77.2% of an accurate execution. Thus, to evaluate the impact of integrating the multipliers, we elected one arbitrary execution instance at this operating point and executed Sobel with approximate multipliers varying the selection of an approximate region. To emulate the multiplication behavior, our extended simulator was configured to replace multiplication instructions by the software model that describes two EvoApprox8b multipliers, *mul8_479* and *mul8_423*, [96] using a Wallace-tree architecture to produce higher bit length multiplications.

For the approximate multiplication unit, the code region affected by approximation was manually selected by analyzing the application execution pattern. The approximate region is delimited by the application using the user-level status registers defined in Section 4.1.4. Firstly, attempting to expose the whole application to errors in multiplication results in failed execution. Application initialization, finalization, I/O operations, and memory allocation rely on multiplications of critical information, causing the application to crash. Then, we narrowed the possible approximate region down to the main computation kernel, composed of the Sobel operator gradients computation and final combination. From these, using the approximate multiplier to compute the squaring of each element on the gradient matrices amplifies all errors injected by the multipliers, resulting in a much noisier output image. Thus, we apply the approximate multiplication only in the convolutions to compute the gradients themselves. According to the detailed instruction counters produced by our extended simulator, these convolutions represent 78.3% of all multiplication instructions executed by the application, and the approximate region covers 42.1% of the total number of instructions. Alternatively, automated tools such as ACCEPT [121], ASAC [119], DECAF [15], ApproxSymate [32], and SmartApprox [43] can be extended using our architectural specifications to allow designers to find optimal selections of approximate regions and configurations.



(a) Accurate (b) mul8_479 (c) mul8_423

(d) Kulkarni (e) ILM_EA (f) ILM_AA

(g) DRAM (h) DRAM + mul8_479 (i) DRAM + mul8_423

(j) DRAM + Kulkarni (k) DRAM + ILM_EA (l) DRAM + ILM_AA

Figure 6.4: Sobel execution using approximate DRAM and multipliers. *Approximate multiplication lead to increased quality degradation in comparison to the DRAM approximation.*

Figure 6.4 shows a quality comparison between an accurate execution (6.4.a) of the Sobel application and scenarios in which the DRAM (6.4.g) and multiplication approximation units are used in isolation (6.4.b and 6.4.c) and integrated (6.4.h and 6.4.i). In this example, the quality degradation imposed by the DRAM approximation (6.4.g) is not visually noticeable, yielding very high quality metrics (image 6.4.g SSIM = 99.9%; average SSIM = 98% for the Sobel application). Thus, the impact on quality in the integrated scenario is predominantly derived from the multipliers only, even though all pixel matrices are stored in the approximate region and the image does not fit our L2 cache. Both multipliers fail to capture minor contrast differences in the source image, which results in highlighting less prominent edges. This effect is more perceptive in *mul8_479* (6.4.b and 6.4.h) than *mul8_423* (6.4.c and 6.4.i), that fails to detect some edges and pushes down quality to 58.4% and 47.2%, respectively.

The impact imposed by the multipliers on energy depends on various factors, such as the energy savings provided by the multiplication operator, the relative energy cost of multiplication in the target hardware, and the coverage of the approximate region in the whole application. Considering that *mul8_479* and *mul8_423* cost 70% and 50% of an accurate multiplication, respectively [48, 49], that a multiplier represents 8.5% of the execution energy cost per instruction of a generalized RISC-V workload [144], and that we achieve 42.1% coverage, this represents saving 1.07% and 1.79% on execution energy. Even for applications that tolerate a larger fraction of approximate multiplications, the upper bound on energy savings is 4.25%, maintaining a 5% mean relative error (*mul8_423*). In the integrated scenario, the accumulated energy savings depend on how the memory energy cost compares to the execution energy. The DRAM energy cost varies significantly according to the CPU and cache configuration and the application access pattern, for example [67]. Taking a reasonable conservative assumption that both DRAM and the CPU contribute equally to the total energy cost of the system, energy savings add up to 11.9% for *DRAM + mul8_479* and 12.8% for *DRAM + mul8_423* in the sample Sobel application, and 14% for an application that accepts all multiplications to be approximated to 5% mean relative error.

The evaluation of the Sobel application in this scenario suggests that approximating multiplications imposes higher quality degradation and lower energy savings than the DRAM approximation. Any multiplication energy gains are limited both by the coverage this operation achieves in the scenario and the relative energy cost of multipliers, an upper bound of 3.58% of the overall execution energy cost. To verify that the higher quality degradation characteristic is not particular to the selected multiplier library [96], we expanded our demonstration to other classes of approximate multipliers: Kulkarni's dedicated logic multiplier [77] and two logarithm multipliers – *ILM_EA* and *ILM_AA* [8]. All three architectures exhibit higher quality degradation, even though the image is still distinguishable. *Kulkarni* (SSIM = 39.4%, 6.4.d and 6.4.j) and *ILM_EA* (SSIM = 39.1%, 6.4.e and 6.4.k) tend to overly highlight lighter edges, such as in the image background. The multiplier *ILM_AA* includes an approximate adder to accumulate the logarithmic multiplication partials that truncates most of the pixels to a mid-range gray, highlighting only the main edges of the image (SSIM = 12.2%, 6.4.f and 6.4.l).

6.2 Experiments on the hardware prototype

The experiments in the simulation environment were designed to demonstrate how the approximation control behaves when operating with different configurations of approximation and for different application domains. The simulator allows easy access to the internal structures of the processor, producing detailed statistics and traces of execution that can be used for a more representative energy estimation of the different execution scenarios. The higher level of abstraction in the implementation also helps to design special cases of approximation injection, such as protecting the execution of a supervisor system and specific memory regions. The main drawback of experimenting with the simulation environment is scalability. One single execution instance of the JPEG application for a reasonably small input image (512x512 pixels), for example, takes a simulation time in the order of minutes, due mainly to a performance bottleneck on accounting for cache accesses and producing the memory access traces for later offline energy estimation. Even though single execution instances of the simulator can be run in parallel, that only goes up to a point and does not help with scaling up the input data size. Moreover, the simulator is a functional representation of the system and abstracts microarchitectural details of the implementation that can affect the overall execution, the design choices for approximation, and the final quality results.

The FPGA prototype, on the other hand, performs much better than the simulator and is an actual processor implementation, thus representing the system without the level of abstraction in the simulator. The same previous example of a JPEG execution instance takes less than 4 seconds to complete in the prototype while representing a real implementation of the approximate architecture. However, the design effort to adapt applications to run in the FPGA prototype is greater and the flexibility to represent different approximations is limited. Also, the prototype produces less detailed information about execution, jeopardizing the possibility of energy estimation without the detailed instruction counters and access traces produced by the simulator.

Taking into account these differences between the simulation and prototype environments, we designed the hardware prototype experiments to scale the quality results for one pair of applications and operating points obtained in the simulator. We selected the JPEG application and the 1.05 V intermediate operating point, subjected to a 6.461×10^{-4} error rate, combined with the five approximate multipliers in the prototype (Table 5.5) [96]. Each configuration was iterated over the complete Imagenette Full-size train dataset [56]. The dataset comprises 9,855 images ranging in raw image data size from 11.3 KB (80x24 pixels) to 78.8 MB (2,912x4,368 pixels), totalizing 13 GB of data for each of the 12 different configurations.

The JPEG application was modified to work with the raw image data as input and produce the final compressed image file as output. Both input and output data are received from and transmitted to an auxiliary computer using the faster 1 MBd UART port. The auxiliary computer application reads the image files and extracts the raw image data, preparing the input for the JPEG application, and after processing, it receives the compressed image and stores them for quality evaluation. Within the JPEG application, input and output data are stored in the approximable memory region and thus

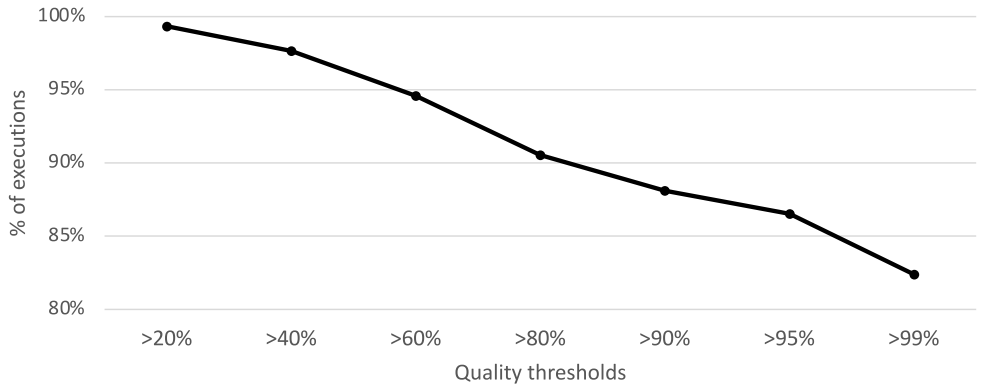


Figure 6.5: Quality distribution on approximate DRAM. *More than 85% of the JPEG executions meet common quality thresholds of 90%, 95% and 99%.*

are subjected to the configured error rate if missed from the cache into DRAM. Due to limitations in the ported JPEG application, the images are converted to single-channel grayscale before compression. The approximate multiplication operators were activated for all multiplication instructions within the main encoding algorithm, which includes quantization, downsampling, computing the discrete cosine transformation, and writing the output data. To account for quality of results, we use the Structural Similarity metric (SSIM), comparing the accurate results with the ones subjected to approximation.

6.2.1 Evaluating the Approximate DRAM operating point

The previous experiments in the simulation environment tested the behavior of applications subjected to five different levels of approximation from a voltage-overscaled DRAM memory, considering three scenarios of variability. The results concluded that, for the applications tested and considering the intermediate variability scenario, a voltage level of 1.050 V can provide results at an average quality level higher than 90% and provide significant energy savings. For the JPEG application, in particular, these savings are expected to range from 21% to 25% at this operating point. These expectations, however, are based on a more limited scenario where the applications were executed on a single rather small input. In the prototype environment, we analyze whether the expectation of producing results with good enough quality at 1.05 V is maintained as the applications grow in size and in the number of inputs.

Figure 6.5 shows the percentage of JPEG executions that meet different criteria regarding quality of results. On average, the 1.05 V operating point provided results of 94.60% SSIM ($\sigma = 14.91$ p.p.), and only 11.9% of the execution instances would not meet a 90% quality threshold. This indicates that the perceived resilience of the application scales up with the number and size of inputs. Moreover, by visually analyzing the image outputs, the JPEG application is particularly well-behaved under DRAM approximation.

Figure 6.6 exemplifies output images that represent each level of reported quality of results. For each level, we selected an image with Structural Similarity close to the average in the class, lower-bounded by the quality threshold and upper-bounded by the first higher level. At higher quality levels, the noise in the outputs is composed mostly of small groups

(a) Accurate (b) 20% (c) Accurate (d) 40%

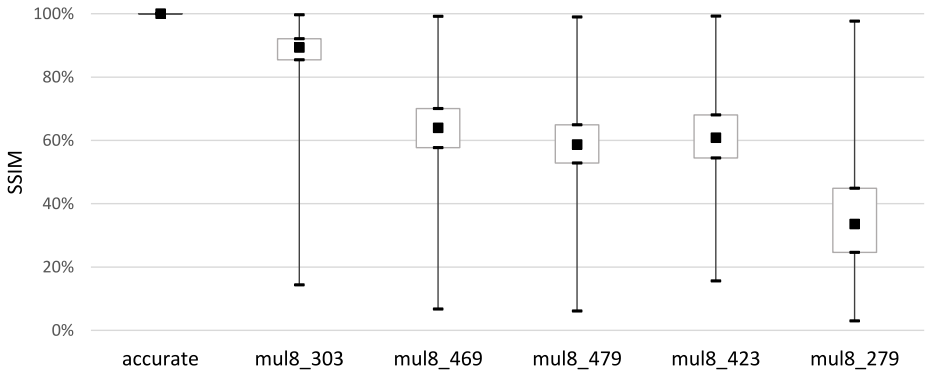
(e) Accurate (f) 60% (g) Accurate (h) 80%

(i) Accurate (j) 90% (k) Accurate (l) 95%

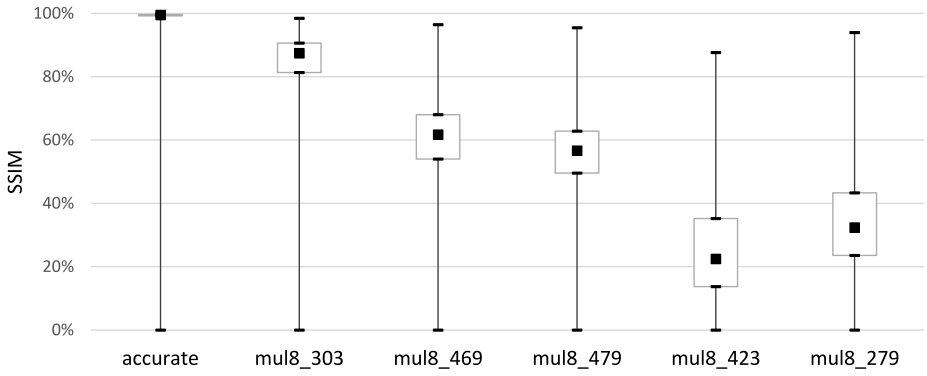
(m) Accurate (n) 99%

Figure 6.6: Sample images at given quality thresholds. *The JPEG application behaves resiliently under DRAM approximation, maintaining visual quality.*

of distorted pixels distributed within the image, as shown in Figure 6.6.n. This is consistent with output data being affected by noise after computation. As quality decreases, the images exhibit displaced stripes of pixels (Figures 6.6.b-h) or inconsistent levels of brightness (Figures 6.6.f-l). These indicate that the processing step was further affected by noise propagated throughout cascaded Huffman coefficients in the JPEG implementation, causing a more significant impact. Despite the quality measurement, the images appear distinguishable to the human eye. Depending on the scenario the application is introduced, all images could be acceptable for their endpoint use case. This observation may indicate that the usual methods to evaluate the quality of results, such as the Structural Similarity Index for Image Processing applications, may not necessarily correlate with the actual outcome of the approximation, as we further analyze in Chapter 7.



(a) Accurate DRAM



(b) Approximate DRAM

Figure 6.7: Quality of results after integrating the approximate multipliers. *On average, the multipliers have a higher impact on the Structural Similarity.*

6.2.2 Integrating the Approximate Multipliers

The results of the DRAM approximation show that setting the DRAM voltage level to the most efficient operating point in an intermediate scenario, 1.05 V, introduces a significant error probability of 6.461×10^{-4} in memory accesses but can still result in high-quality results, on average, and particularly well distinguishable images for the JPEG application. They also indicate that scaling input sizes maintains the expected average output quality at similar levels. After studying the DRAM operating point, we expand the analysis by integrating the selected approximate multipliers from the EvoApprox8b library [96]. The previous results in the simulation environment showed that the approximate multiplication highly impacts the quality of results and has an energy cost jeopardized by the integration of the multipliers within the architecture. These factors indicate that approximating the multiplication operator may not be profitable in terms of energy-quality trade-off. The integrated experiments on the FPGA prototype analyze how these quality findings translate when the experiment is scaled further.

Figure 6.7 shows the distribution of the quality of the results generated with each approximate multiplier, with and without the approximation injected also in the DRAM. Introducing approximate multiplication significantly degrades quality compared to the previous DRAM approximation. Less than 10% of the executions would meet the 90% quality criteria, considering all multipliers combined. Even the higher-quality *mul8_ 303* achieves

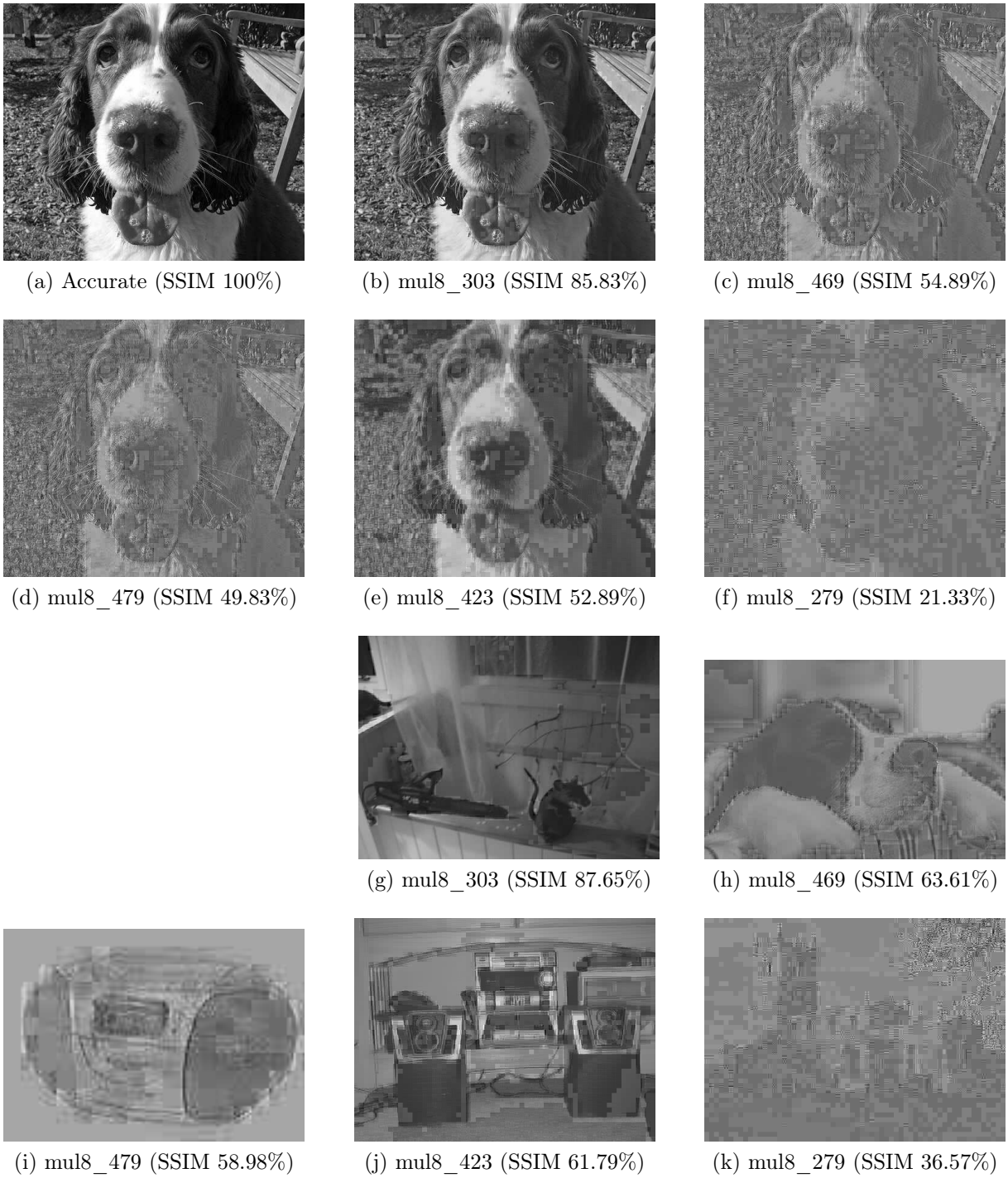
90% at less than 45% of the executions, which is 90% of the total results that meet the criteria. The best multiplier (*mul8_303*) has average quality of 87.65% ($\sigma = 7.85$ p.p.), the worst (*mul8_279*) averages at 36.57% ($\sigma = 16.12$ p.p.), and the three intermediate ones overlap ranging from 58.98% to 63.61% averages ($\sigma = 11.47 - 11.75$ p.p.).

The introduction of DRAM approximation together with the multipliers (Figure 6.7.b) has little impact on quality of results, except for *mul8_423*. The similarity is expected, as the DRAM approximation introduces little degradation in the design and, being the individual multiplier degradation higher, it is expected to dominate. However, the EvoApprox8b multipliers, although all members of the same library, are actually different multiplication hardware designs that were conceived using the same methodological approach [96]. Thus, even though they are expected to have similar behavior, this is uncertain, especially when integrated with other sources of approximation.

Figure 6.8 selects individual JPEG execution instances to visually demonstrate how the approximate operation affects the image output. Figures 6.8.a-f show the six levels of approximation injection in the same input image, while Figures 6.8.g-k select a particular input that represents the average quality for each of the approximate multipliers. Introducing approximate multiplication noticeably and strongly degrades the visual quality of output images. Differently from the DRAM approximation, which is non-deterministic and results in less predictable distortions on the output image, the approximate multipliers are deterministic and produce results that follow a clear pattern of creating less detailed images as quality degrades.

The multiplier *mul8_423*, which introduces the second-highest mean relative error on the order of 5% [49], shows another discrepant behavior and has better visual quality than the lower error *mul8_469* and *mul8_479*. Despite any differences in the design of the multipliers, as discussed above, the specific mean relative error metric is computed over a uniform distribution of multiplication operands and does not capture any architectural level integration of the multiplication hardware or particularities of the target application, JPEG in this case. At an energy cost of approximately 50% of an accurate multiplication [49], this could indicate that *mul8_423* is particularly suitable for the JPEG application and a good energy-error trade-off. However, any per-operation energy savings are rapidly amortized when considering the architecture-level integration, as discussed in the simulation-environment results, making the particular approximate multiplication seldom worth it. Similarly, *mul8_303*, although producing higher-quality results in comparison to the other multipliers, has an energy cost per operation that comes near 95% of an accurate operation and would not particularly impact the overall energy efficiency in the architecturally-integrated scenario.

Differently from the DRAM-only scenario in Section 6.2.1, the approximate multiplication SSIM metric captures more linearly the perceived degradation of visual quality. However, the similarity results of the multiplication and DRAM scenario seem not comparable, even though the same metric, with the same interpretation, has been used. Figure 6.8.c, for example, has a higher similarity metric than Figure 6.6.d, both generated from the same input, but appears significantly more degraded. Both Figures 6.8.b and 6.8.g have fairly good quality visually, at over 85% SSIM, but would hardly be considered better than Figures 6.6.b, 6.6.d, and 6.6.f by a human observer, all bellow 80% SSIM. This



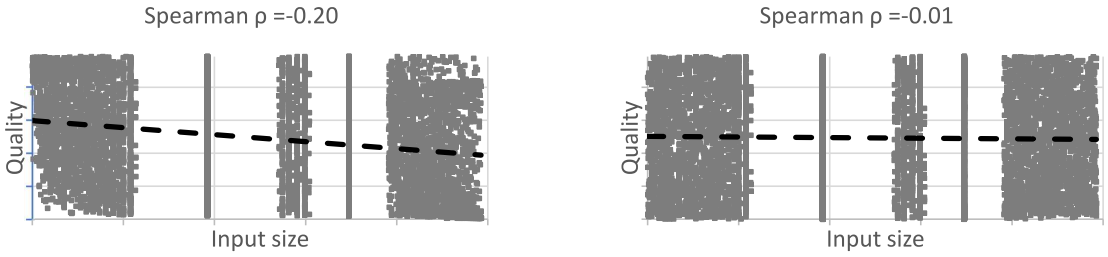
(a) Accurate (SSIM 100%) (b) mul8_303 (SSIM 85.83%) (c) mul8_469 (SSIM 54.89%)

(d) mul8_479 (SSIM 49.83%) (e) mul8_423 (SSIM 52.89%) (f) mul8_279 (SSIM 21.33%)

(g) mul8_303 (SSIM 87.65%) (h) mul8_469 (SSIM 63.61%)

(i) mul8_479 (SSIM 58.98%) (j) mul8_423 (SSIM 61.79%) (k) mul8_279 (SSIM 36.57%)

Figure 6.8: Sample images illustrating outputs of approximate multiplication. *The approximate multiplication operators impact heavily on the visually perceived quality of results.*



(a) Approximate DRAM (b) Approximate Multiplier

Figure 6.9: Correlation test between input size and quality of results. *The Spearman test shows no correlation, which indicates both techniques scale gracefully with input size. Both axes X and Y are ranked and do not represent the actual input size or quality values.*

flawed correlation between the quality metric and the perceived visual quality indicates that the SSIM metric may not be the most suitable evaluation in these approximate JPEG scenarios, especially for the DRAM case, even though it is a traditional and extensively used metric. In Chapter 7, we extended this empirical observation to other metrics and applications, using a methodological approach to evaluate the correlation between quality metrics and the perceived acceptability of results.

6.2.3 The impact of scaling input size

In the simulation environment, the experimentation results were limited by the number and size of the inputs. Mainly because of simulator performance limitations, experimenting with larger input sizes was impractical. The performance improvement achieved by the FPGA prototype allows these results to be scaled to a greater number of use cases and larger input sizes. The quality results demonstrated that the experiments are comparable and, in the scaled-up scenario, the selected approximate DRAM operating point can also lead to results that meet the proposed quality criteria. These indicate that the approximations can deal with larger scenarios, but do not provide any evidence on to what extent the results are representative.

To better understand how the input size impacts the quality of results, we analyzed the correlation between these two aspects. We compared how the input size influences the overall quality of results for the DRAM approximation and one of the approximate multipliers. In the multiplication scenario, we elected only *mul8_303*, which is the one that showed higher average output quality. The approximate multiplication operators are deterministic and affect the operands in a similar and predictable way throughout the execution, and thus were expected to correlate poorly with input size. The approximate DRAM accesses, on the other hand, are probabilistic and distributed through the execution. The longer an application lives, the more access errors are expected to happen. Since larger input sizes would typically lead to longer execution times, more access errors happen for larger inputs. Moreover, being the approximation source the DRAM accesses, larger inputs also are less likely to be efficiently stored in the cache, increasing the likelihood of errors. It would be natural to assume that more errors lead to lower overall quality, thus inferring some correlation between input size and quality.

Figure 6.9 shows the Spearman correlation between input size and quality of results for both the DRAM and multiplication scenarios. The Spearman's rank correlation coefficient ρ measures how related are two variables in the same population. The coefficient is computed over ranked values, and thus it assesses the relationship capturing non-linear relations. In our scenario, we want to evaluate whether quality actually decreases as the input size increases, regardless of the nature of the relation, and thus this monotonic description in Spearman's test is applicable. The coefficient ρ increases to ± 1 the stronger the correlation, and equals 0 when there is no correlation.

If the quality of results indeed decreased with larger input sizes, the Spearman coefficient ρ should be close to -1. However, both scenarios have values close to 0, showing that the correlation is very weak, if any. This indicates that the size of inputs is unlikely to significantly affect the overall quality of results in these scenarios. Also, it validates that the results reported both in the more limited simulation environment and in the hardware prototype are representative of what should be expected in a real-world scenario, regarding quality of results.

Chapter 7

A case for acceptability

Is quality enough?

Approximate systems offer improved efficiency in computing at a cost of accuracy in the final result. The main observation behind such systems is that there are applications, within certain scenarios, that do not require accurate computation in order to produce useful results for their purpose. Typically, less accuracy means less effort on computation, lower energy cost, and higher performance, which may highly benefit applications that tolerate inaccuracy during computation [91]. The level of inaccuracy can be measured using application-specific quality metrics that compare the results with golden standards and quantify how far away they are [11]. With such an quantifiable method, approximate systems tend to rely on quality metrics to determine when the approximation results are acceptable. Automatic configuration of approximate systems is designed to provide results above a predetermined quality threshold, and approximate models are evaluated based on similar quality requirements [91]. However, the correlation between quality and acceptability of results is not always implied, as an empiric inspection of approximate data shows that low-quality results may be acceptable within certain scenarios, while high-quality results, as measured by a certain metric, may not imply acceptability.

In this Chapter, we propose an application-oriented approach to determine whether results from approximate models are acceptable. In this method, the approximate application is inserted within a computation scenario, at which the results of the approximation are the inputs of later transformations in the pipeline. The later step can unequivocally determine whether it had affected the result in a way that the final interpretation was changed. This translates the objective quality threshold into a subjective acceptability evaluation, as it takes into account the purpose of the application in the evaluation workflow, as the answer to whether results are useful cannot be fully generic.

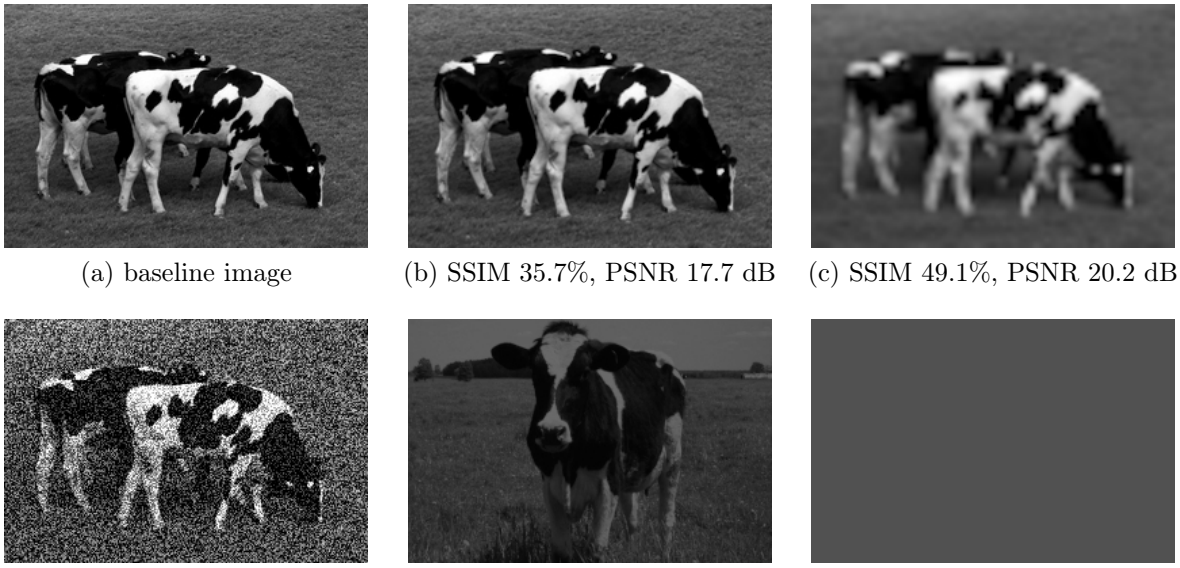
We demonstrate the method in applications of Image Classification, License Plate Recognition, and Pitch Recognition when subjected to approximation. The scenarios were solved at increasing levels of approximation according to each application resilience. Our results provide statistical evidence on the empiric observation that quality does not translate into acceptability, quantify the fraction of results that are misclassified as useful or not useful by arbitrary quality thresholds, and show to what extent acceptable results are discarded and the unacceptable ones are not on a quality evaluation.

7.1 Quality evaluation of Approximate Systems

The configuration of approximate systems typically hinges on finding an error balance that respects the limit of acceptable depreciation of the results. This allowed depreciation depends on a decision problem that evaluates whether a result is useful to the context of its application. A practical solution often involves quality metrics that rely on the type of data, where multiple metrics can be used for evaluating quality loss for several applications, although they are not mutually exclusive [91]. Common quality metrics adopted for approximate systems are Structural Similarity Index Measure (SSIM) and Peak Signal-to-Noise Ratio (PSNR) for image processing applications, PSNR for signal processing, variants of Mean Absolute Error (MAE) for numerical results, and the number of mismatches from positional data [11]. In common, these metrics measure how far from an "accurate" result the approximation was.

The utilization of quality metrics transforms the initial decision problem into calculating or quantifying the metric, which means defining a quality threshold. Approximate systems are often designed to work with conservative quality thresholds, and approximate applications are demonstrated as more effective when they are able to meet these requirements [91]. Although these thresholds are an objective way to determine useful results, they can be ineffective by not considering the context of the application. For instance, noise may spoil a minor but determinant portion of an image like a character from a license plate, leading to a high-quality but useless result in a recognition application. Therefore, quality metrics do not determine effectively the acceptability of a result and may mislead the configuration of approximate components. This misguided configuration may lead to excessive production of useless results or avoidance of useful ones, which decreases approximation benefits by wasting computing resources.

Figure 7.1 illustrates how quality measures can mislead the observation with examples of noise injection and morphological changes and their respective values of SSIM and PSNR. Figures 7.1b and 7.1c emulate the behavior of lossy compression algorithms executing on approximate systems, where the image was slightly cropped and has blurred results by restoring a resized pixel array, respectively. In Figure 7.1b, the image has no subject visible differences with baseline but has lower values of PSNR and SSIM than Figure 7.1c that has visible distortions from the baseline image. Figure 7.1d inserts noise in the baseline image as the data had passed through an unreliable channel (e.g., an approximate memory). Despite Figures 7.1b, 7.1c, and 7.1d being acceptable in an application of image classification, for example, they would seldom meet the quality requirements of commonly adopted thresholds. Figures 7.1e and 7.1f are completely different images, while still evaluated as better than Figure 7.1d, which is visually similar to the baseline. Considering that Figure 7.1b is the most useful one, followed by 7.1c and 7.1d if more error is accepted, there is no quality threshold that can solve the acceptability decision problem, based on SSIM and PSNR metrics alone, in this particular set. Deciding what is acceptable is highly dependent on the application and the overall computing scenario, and, therefore, an application-oriented approach is required.



(a) baseline image (b) SSIM 35.7%, PSNR 17.7 dB (c) SSIM 49.1%, PSNR 20.2 dB

(d) SSIM 18.5%, PSNR 14.0 dB (e) SSIM 19.3%, PSNR 12.4 dB (f) SSIM 26.8%, PSNR 14.1 dB

Figure 7.1: Effect of morphological changes and noise injection in the quality metric. *All images are below conservative quality thresholds, visually better images are evaluated worse, and clearly different images are evaluated better than similar ones.*

7.2 Analyzing Acceptability of Application Output

Approximate systems use a threshold on common quality metrics to define whether the application output is acceptable or not. However, this approach tends to produce misguided results, as discussed in Section 7.1. One factor that contributes to the sub-optimal evaluation of quality thresholds is that they do not consider *what for* a result is acceptable. In our analysis, the approximate app is embedded as a portion of a more complete computation scenario. Thus, instead of evaluating the quality of the "midstream" data that outputs from the approximate portion, we evaluate how useful it is for the later processing steps. The output of the approximation is considered useful, and therefore acceptable, if and only if the output of the whole computation scenario is interpreted as the same as what would result from accurately-computed data. This translates an objective, but unreliable, quality threshold into a subjective, but more meaningful, acceptability evaluation.

Figure 7.2 summarizes the three computation scenarios analyzed in this work. For each scenario, we employed architectural simulation [47] to model the execution of the approximate portion using voltage-overscaled memory devices that reduce power with an associated increased probability of errors in memory accesses [38]. Each application was subjected to at least four incremental error rates, according to its inherent resilience to error, iterating over the full input dataset for each error rate. The approximate outputs are evaluated for quality against accurately-computed ones using common quality metrics for the given applications.

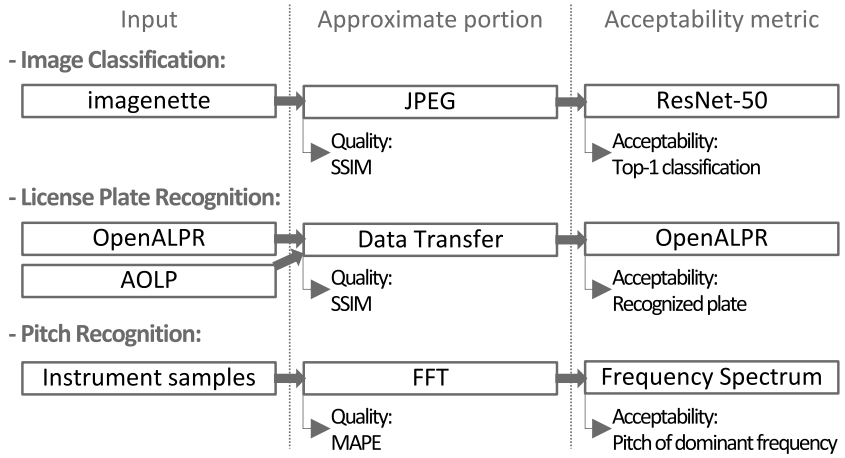


Figure 7.2: Our proposed approach to evaluate the acceptability of results. *Approximate applications are inserted as part of a computing scenario, and acceptable results are the ones that do not change the general outcome.*

(Reused, with permission, from [45] ©2021 IEEE)

The computing scenarios are:

- **Image Classification:** We use ResNet-50 [52] under TensorFlow to label the Imagenette [56] dataset, adopting the network pre-trained with Imagenet weights. The approximation was injected in JPEG preprocessing, and the quality metric is SSIM. The acceptable outputs are those whose top-1 classification label matches the error-free JPEG. In this JPEG application, the approximation in the output images is visually perceived as losses in resolution, similar to Figure 7.1c.

- **License Plate Recognition:** We use OpenALPR [116] to read the license plates from the software's benchmark dataset (400+ plates from Brazil, Europe, and The United States) and the AOLP [57] complete dataset (2000+ Taiwanese plates). The approximation emulates the data transfer of the images, from a camera to memory, for example, through an unreliable channel, and the quality metric is SSIM. The acceptable outputs are the ones in which the recognition result is the same, even if no plate is detected. This scenario represents a more classical feature extraction and character recognition approach for Computer Vision. Also, the approximate data transfer inserts random noise in the data, such as depicted in Figure 7.1d.

- **Pitch Recognition:** We analyze the frequency spectrum of 2.5K+ single-note instrument samples [130]. The frequency spectrum was derived from the Fast Fourier Transform (FFT) of the samples, and the final pitch is the dominant (highest amplitude) frequency in the signal. Even though the dominant frequency is not the actual definition of pitch, it is a good estimation for the single-note samples that was accurate 68% of the time. FFT outputs are compared for quality using the complement of Mean Absolute Percentage Error (100% - MAPE), and the result is considered acceptable if both accurate and approximate frequencies translate to the same note – for example, 430 Hz and 450 Hz would be a match since they both represent the note A4.

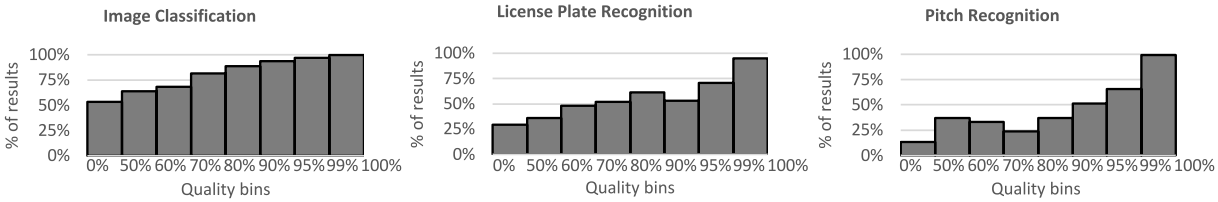


Figure 7.3: Percentage of results that are acceptable within bins delimited by quality thresholds. *Higher quality generally correlates with higher acceptability, but low-quality bins still contain acceptable results and high-quality bins, unacceptable ones.*

(Reused, with permission, from [45] ©2021 IEEE)

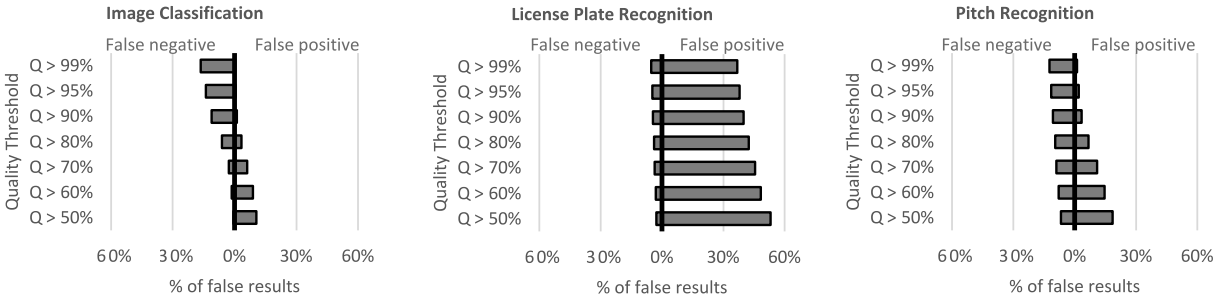


Figure 7.4: False positives or negatives: percentage of unacceptable or acceptable results that are wrongfully classified using quality thresholds only. *Since they are inversely proportional, quality thresholds cannot sufficiently minimize them both.*

(Reused, with permission, from [45] ©2021 IEEE)

7.3 Results: Quality vs Acceptability

The relation between the quality of an application output and how acceptable it is for a given computing scenario is not clear, and, thus, relying on a quality threshold may lead to unreliable results. Thus, we propose to replace the quality evaluation with an acceptability analysis that inserts the approximate application within a larger scenario. This Section shows how the Image Classification, License Plate Recognition, and Pitch Recognition scenarios behave when subjected to approximation, and how quality evaluation correlates with but does not translate to acceptability of outputs.

Figure 7.3 shows the fraction of results that are rendered acceptable within bins delimited by quality thresholds. In general, higher-quality bins contain a higher fraction of acceptable results, but not exclusively. The average quality of acceptable results is higher than 93% and lower than 43% for unacceptable ones. However, these averages present standard deviations in the order of 8–20 p.p. for acceptable results and 23–42 p.p. for unacceptable ones, which highlights how unreliable the quality evaluation can be to determine whether results are useful. The quality of acceptable results can be as low as 0.6% for Pitch Recognition, 20.4% for Image Classification, and 51.93% for License Plate Recognition, while unacceptable results can have quality up to 99.9% for all three scenarios. This means that, whichever is a chosen quality threshold, a fraction of the unacceptable results would be falsely evaluated as useful, and acceptable ones would be falsely discarded.

The fraction of these incorrectly classified results in relation to the overall number of acceptable or unacceptable results is quantified in Figure 7.4. False positives are unac-

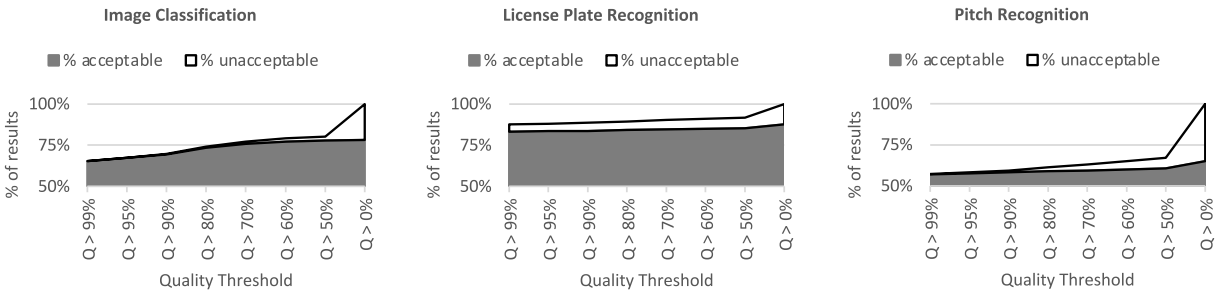


Figure 7.5: Percentage of acceptable and unacceptable results with quality above predetermined thresholds. *Conservative quality thresholds limit the number of acceptable results and still allow unacceptable ones.*

(Reused, with permission, from [45] ©2021 IEEE)

ceptable results that are incorrectly classified as useful by a quality threshold, and false negatives are acceptable ones left behind. Naturally, the false negatives increase and the false positives decrease the higher is the quality threshold. This inverse proportion makes it difficult to design a system that minimizes false results below an acceptable noise margin based on quality thresholds only. For example, the quality threshold that most efficiently minimizes false results is Q > 70% for Image Classification, even though it includes 6.1% of the unacceptable results and misses 2.9% of the acceptable ones. In approximate systems, both false positives and false negatives represent losses in the overall efficiency of the system, since false positives mean computation time that could be used to produce actually useful results, and false negatives imply missed opportunities to take profit from more aggressive approximations.

Figure 7.5 shows the cumulative distribution of both acceptable and unacceptable results where the quality metric is evaluated above given thresholds. This aggregates the previous analysis and highlights how many of the individual instances that were to be accepted by a quality threshold are actually acceptable, as well as how many are left behind and incorrectly evaluated. In all cases, decreasing the quality thresholds increases the number of acceptable results produced, especially for more resilient applications such as Image Classification. Compared with a system trained to deliver 90% quality, an acceptability analysis can deliver up to 12.6% more useful results in this scenario. Even for less-resilient applications, such as License Plate Recognition, the number of useful results can increase 4.7%, while still avoiding 4.9% of results that were not acceptable but surpassed the quality threshold. Considering the extra-conservative threshold 99%, up to 19.7% more results could be rendered useful. These results show that any choice of quality threshold leads to sub-optimal results regarding acceptability.

Our evaluation shows that arbitrary thresholds can misclassify the results, as acceptable results are perceived within a wide range of quality bins. Our method of evaluation, on the other hand, can produce up to 20% more valid results, while still avoiding a large fraction of unacceptable ones. These results suggest that a quality-only approach to analyze approximate systems may leave their efficiency evaluation over- or under-estimated, while its evaluation against a more realistic acceptability scenario can help to understand the approximation gains.

Chapter 8

Conclusions

Architectural support and future directions

Hardware-level approximation techniques present a significant opportunity to offer increased energy efficiency in error-tolerant applications. Their design either focuses on the development of the approximated modules themselves in a dedicated and fully optimized way or is built to provide a scaling knob that configures a level of quality in the results of the hardware unit. The dedicated hardware alternatives, however, cannot handle full accuracy computation and, therefore, are not compatible with computation in a general-purpose scenario. The alternatives that support configuration, on the other hand, provide an evaluation of how each quality level impact the final quality, with little insight into how a target application or system could control it. Either way, an adequate hardware-software interface is needed to allow applications to control what and how the approximation affects their behavior.

Previous attempts to integrate approximations and control within an architecture present limitations to be applicable in a general-purpose scenario, and do not represent the full picture on how approximation hardware affects the quality of results and energy. To overcome these limitations, we propose a generic architecture-level interface that allows software to control approximation hardware. Our ISA extension was built to offer control capabilities in a coarse granularity, thus allowing for the mitigation of the time overhead imposed by the need of turning approximation-hardware modules on or off at runtime. In the software side, the mechanism to allow control is based on rapidly-available control and status registers that can be written and read using existing instructions in the ISA specification, which reduces the software overhead and derives compatibility with existing compiled software. This control mechanism also applies to multi-application and multi-user environments, in which a supervisor system can analyze and control the approximation capabilities of the system on behalf of application software. This ISA extension represents a significant contribution of this thesis that allows approximation-hardware designers and approximation-aware software developers to test approximations integrated within a general-purpose computing scenario.

Our demonstration of how integrating hardware-level approximation affects general-purpose application scenarios covers two levels of implementation. First, in a functional simulator, we show how both quality and energy figures diverge greatly from the ones

reported for approximation units in isolation when architectural requirements and the need for allowing reliable execution levels are considered. Also, the results demonstrated that architectural integration and a proper control interface are required to properly understand the impact of Approximate Computing on applications. Based on functional simulation, however, these results cannot capture all hardware- and architectural-level considerations that apply to an actual processor. The simulator does not cover aspects such as cache implementation, branch prediction, and the actual organization of some approximation units.

The limitations of the simulator create a level of uncertainty on whether these results actually scale to real-world, larger, general-purpose scenarios. Thus, we introduce a second level of implementation for the same architecture-level design concepts. A full-system multi-core hardware prototype was developed and demonstrated for an execution scenario involving a substantially larger input dataset. This scenario showed a similarity of results with the simulated scenario, indicating that the approximation benefits can be extended. Moreover, an statistical analysis of the results over a varied-sized input dataset showed that the benefits scale gracefully in larger scenarios. This highlights that an appropriate level of architectural control can enable approximate computing in a general-purpose scenario. Also, the results with the hardware prototype validated that the energy-quality trade-offs obtained with the software simulator are representative of real use cases in which such approximations could be inserted in a general-purpose computer.

Further analysis of the results obtained in both the simulation and prototype environments raised concerns about how representative the usual quality metrics are when facing the actual perceived quality by a potential human observer. This motivated an auxiliary study on quality and acceptability metrics. We show that, even though there is some correlation, both high- and low-quality metrics do not necessarily mean results are acceptable or not in a significant number of cases. This analysis can serve as motivation for further studies about the actual meaning of quality metrics, the development of new quality and acceptability metrics, as well as the extension of previous quality results to the subjective scenario. With the design tools presented in this thesis, researchers can pursue opportunities to improve approximate systems to offer guarantees of acceptable results. Previous work has shown an inference-based configuration of approximations based on existing hardware-level counters by correlating their values with known applications [43]. However, the evaluation of results was based on a fixed quality threshold, which we showed may not mean an acceptable result. By introducing acceptability metrics in the learning mechanism, researchers can obtain a system that is trained to provide acceptable results and introduce scenarios in which a fully-featured quality metric to analyze results is superseded.

Besides the opportunities with the acceptability analysis, the studies conducted in this thesis identified other different scenarios that require further understanding and research questions that can also motivate future work. The experimental analysis can be extended to include more application and approximation scenarios. There is increasing interest in creating new approximation sources and identifying different target applications that can be resilient to them. In particular, applications that rely on heavy floating-point computation can be good candidates to approximate Floating Point Units [17, 72, 86, 102].

Floating-point values are likely not in the control flow of applications, or at least not directly compared for equality in the control flow of applications. Our previous studies of resilience of applications indicated that a separation of the control flow is a dominant characteristic to avoid application crashes, and avoiding application crashes is determinant towards increasing energy efficiency in approximation scenarios [37,40]. Also, the floating-point representation is, by the very definition, an approximated concept to represent the infinite set of real numbers. Thus, these applications and supporting hardware are good candidates to be resilient to approximation techniques.

Another computing domain that is considered to be resilient to input noise, which extends to being resilient to approximations, is the computation of deep neural networks. Popular machine learning frameworks can be implemented within our software simulator and hardware prototype to analyze the energy efficiency of introducing approximations towards acceptable results. These, however, are likely to depend more strongly on external libraries and tools that rely on functions by an Operating System. In this case, even though our experimental environment is Linux-capable, running an actual Operating System requires additional kernel extensions and device drivers to accommodate approximation control at the supervisor level. Similarly, in a multi-core environment, different cores configured with different approximation environments create a scenario similar, in essence, to what is expected in a heterogeneous multi-core architecture, raising an additional need for supervisor-level orchestration. These aspects are discussed in this thesis, but the implementation of an actual generalizable OS extension is a prominent research opportunity.

Finally, superscalar characteristics in a CPU pipeline and out-of-order execution represent additional challenges in the integration of approximation units. In this case, the problem of orchestrating replicated approximation units in the pipeline may be a natural extension of the utilization problem of replicated execution units in the superscalar pipeline with additional design choices and considerations. Still, all of those require some level of cooperation at the architectural level and a better understanding of quality requirements and their correlation with acceptability, which are part of the main contributions of this thesis.

Bibliography

[1] Tor M. Aamodt and Paul Chow. Compile-time and instruction-set methods for improving floating- to fixed-point conversion accuracy. *ACM Trans. Embed. Comput. Syst.*, 7(3), May 2008.

[2] S Abdallah, A Chehab, A Kayssi, and I H Elhajj. TABSH: Tag-based stochastic hardware. In *2013 4th Annual International Conference on Energy Aware Computing Systems and Applications (ICEAC)*, pages 115–120, 2013.

[3] O. Akbari, M. Kamal, A. Afzali-Kusha, M. Pedram, and M. Shafique. PX-CGRA: Polymorphic approximate coarse-grained reconfigurable architecture. In *2018 Design, Automation Test in Europe Conference Exhibition (DATE)*, pages 413–418, 2018.

[4] O. Akbari, M. Kamal, A. Afzali-Kusha, M. Pedram, and M. Shafique. Toward approximate computing for coarse-grained reconfigurable architectures. *IEEE Micro*, 38(6):63–72, 2018.

[5] O Akbari, M Kamal, A Afzali-Kusha, M Pedram, and M Shafique. X-CGRA: An Energy-Efficient Approximate Coarse-Grained Reconfigurable Architecture. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, page 1, 2019.

[6] M Alioto, V De, and A Marongiu. Energy-Quality Scalable Integrated Circuits and Systems: Continuing Energy Scaling in the Twilight of Moore's Law. *IEEE Journal on Emerging and Selected Topics in Circuits and Systems*, 8(4):653–678, 2018.

[7] Alon Amid, David Biancolin, Abraham Gonzalez, Daniel Grubb, Sagar Karandikar, Harrison Liew, Albert Magyar, Howard Mao, Albert Ou, Nathan Pemberton, Paul Rigge, Colin Schmidt, John Wright, Jerry Zhao, Yakun Sophia Shao, Krste Asanović, and Borivoje Nikolić. Chipyard: Integrated Design, Simulation, and Implementation Framework for Custom SoCs. *IEEE Micro*, 40(4):10–21, 2020.

[8] M S Ansari, B F Cockburn, and J Han. A Hardware-Efficient Logarithmic Multiplier with Improved Accuracy. In *2019 Design, Automation Test in Europe Conference Exhibition (DATE)*, pages 928–931, 2019.

[9] Krste Asanović, Rimas Avižienis, Jonathan Bachrach, Scott Beamer, David Biancolin, Christopher Celio, Henry Cook, Palmer Dabbelt, John Hauser, Adam Izraelevitz, Sagar Karandikar, Benjamin Keller, Donggyu Kim, John Koenig, Yunsup Lee,

Eric Love, Martin Maas, Albert Magyar, Howard Mao, Miquel Moreto, Albert Ou, David Patterson, Brian Richards, Colin Schmidt, Stephen Twigg, Huy Vo, and Andrew Waterman. The Rocket Chip Generator. Technical Report UCB/EECS-2016-17, EECS Department, University of California, April 2016.

[10] Alireza Nasiri Avanaki. Exact global histogram specification optimized for structural similarity. *Optical Review*, 16(6):613–621, nov 2009.

[11] R. Iris Bahar, Ulya Karpuzcu, and Sasa Misailovic. Special session: Does approximation make testing harder (or easier)? In *2019 IEEE 37th VLSI Test Symposium (VTS)*, pages 1–9, 2019.

[12] A S Baroughi, S Huemer, H S Shahhoseini, and N TaheriNejad. AxE: An Approximate-Exact Multi-Processor System-on-Chip Platform. In *2022 25th Euromicro Conference on Digital System Design (DSD)*, pages 60–66, 2022.

[13] Soumya Basu, Loris Duch, RubÉn Braojos, Giovanni Ansaloni, Laura Pozzi, and David Atienza. An Inexact Ultra-Low Power Bio-Signal Processing Architecture With Lightweight Error Recovery. *ACM Trans. Embed. Comput. Syst.*, 16(5s), 2017.

[14] Shekhar Borkar and Andrew A Chien. The future of microprocessors. *Communications of the ACM*, 54(5), may 2011.

[15] Brett Boston, Adrian Sampson, Dan Grossman, and Luis Ceze. Probability type inference for flexible approximate programming. In *Proceedings of the 2015 ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications*, OOPSLA 2015, page 470–487, New York, NY, USA, 2015. Association for Computing Machinery.

[16] Martin Bruestel and Akash Kumar. Accounting for systematic errors in approximate computing. In *Design, Automation & Test in Europe Conference & Exhibition (DATE), 2017*, pages 298–301. IEEE, mar 2017.

[17] A. Carvalho and R. Azevedo. Towards a transprecision polymorphic floating-point unit for mixed-precision computing. In *2019 31st International Symposium on Computer Architecture and High Performance Computing (SBAC-PAD)*, pages 56–63, 2019.

[18] Jorge Castro-Godínez, Julián Mateus-Vargas, Muhammad Shafique, and Jörg Henkel. AxHLS: Design space exploration and high-level synthesis of approximate accelerators using approximate functional units and analytical models. In *Proceedings of the 39th International Conference on Computer-Aided Design*, ICCAD '20, New York, NY, USA, 2020. Association for Computing Machinery.

[19] Jorge Castro-Godínez, Muhammad Shafique, and Jörg Henkel. ECAx: Balancing error correction costs in approximate accelerators. *ACM Trans. Embed. Comput. Syst.*, 18(5s), oct 2019.

[20] Jorge Castro-Godínez, Sven Esser, Muhammad Shafique, Santiago Pagani, and Jörg Henkel. Compiler-driven error analysis for designing approximate accelerators. In *2018 Design, Automation & Test in Europe Conference & Exhibition (DATE)*, pages 1027–1032, 2018.

[21] Karthik Chandrasekar, Christian Weis, Yonghui Li, Sven Goossens, Matthias Jung, Omar Naji, Benny Akesson, Norbert Wehn, and Kees Goossens. DRAMPower: Open-source DRAM Power & Energy Estimation Tool. `http://www.drampower.info`.

[22] Arun Chandrasekharan, Daniel Groundefinede, and Rolf Drechsler. ProACt: A Processor for High Performance On-Demand Approximate Computing. In *Proceedings of the on Great Lakes Symposium on VLSI 2017*, GLSVLSI '17, pages 463–466, New York, NY, USA, 2017. ACM.

[23] Kevin K. Chang, Abhijith Kashyap, Hasan Hassan, Saugata Ghose, Kevin Hsieh, Donghyuk Lee, Tianshi Li, Gennady Pekhimenko, Samira Khan, and Onur Mutlu. Understanding latency variation in modern DRAM chips: Experimental characterization, analysis, and optimization. *SIGMETRICS Perform. Eval. Rev.*, 44(1):323–336, June 2016.

[24] Kevin K. Chang, A. Giray Yağlıkçı, Saugata Ghose, Aditya Agrawal, Niladrish Chatterjee, Abhijith Kashyap, Donghyuk Lee, Mike O'Connor, Hasan Hassan, and Onur Mutlu. Understanding reduced-voltage operation in modern DRAM devices: Experimental characterization, analysis, and mechanisms. *Proc. ACM Meas. Anal. Comput. Syst.*, 1(1), June 2017.

[25] V. K. Chippa, D. Mohapatra, K. Roy, S. T. Chakradhar, and A. Raghunathan. Scalable effort hardware design. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, 22(9):2004–2016, Sept 2014.

[26] V K Chippa, S Venkataramani, S T Chakradhar, K Roy, and A Raghunathan. Approximate computing: An integrated hardware approach. In *2013 Asilomar Conference on Signals, Systems and Computers*, pages 111–117, nov 2013.

[27] Vinay K. Chippa, Debabrata Mohapatra, Anand Raghunathan, Kaushik Roy, and Srimat T. Chakradhar. Scalable effort hardware design: Exploiting algorithmic resilience for energy efficiency. In *Proceedings of the 47th Design Automation Conference*, DAC '10, page 555–560, New York, NY, USA, 2010. Association for Computing Machinery.

[28] H Cho, L Leem, and S Mitra. ERSA: Error Resilient System Architecture for Probabilistic Applications. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 31(4):546–558, 2012.

[29] Hans Jakob Damsgaard, Aleksandr Ometov, and Jari Nurmi. Approximation opportunities in edge computing hardware: A systematic literature review. *ACM Comput. Surv.*, 55(12), mar 2023.

[30] V. De, S. Vangal, and R. Krishnamurthy. Near threshold voltage (NTV) computing: Computing in the dark silicon era. *IEEE Design Test*, 34(2):24–30, 2017.

[31] Marc de Kruijf, Shuou Nomura, and Karthikeyan Sankaralingam. Relax: An Architectural Framework for Software Recovery of Hardware Faults. *SIGARCH Comput. Archit. News*, 38(3):497–508, 2010.

[32] Himeshi De Silva, Andrew E. Santosa, Nhut-Minh Ho, and Weng-Fai Wong. ApproxSymate: path sensitive program approximation using symbolic execution. In *Proceedings of the 20th ACM SIGPLAN/SIGBED International Conference on Languages, Compilers, and Tools for Embedded Systems - LCTES 2019*, pages 148–162, New York, New York, USA, 2019. ACM Press.

[33] W El-Harouni, S Rehman, B S Prabakaran, A Kumar, R Hafiz, and M Shafique. Embracing approximate computing for energy-efficient motion estimation in high efficiency video coding. In *Design, Automation Test in Europe Conference Exhibition (DATE), 2017*, pages 1384–1389, 2017.

[34] H Esmaeilzadeh, A Sampson, L Ceze, and D Burger. Architecture support for disciplined approximate programming. In *International Conference on Architectural Support for Programming Languages and Operating Systems - ASPLOS*, pages 301–312, 2012.

[35] Hadi Esmaeilzadeh, Emily Blem, Renee St. Amant, Karthikeyan Sankaralingam, and Doug Burger. Dark silicon and the end of multicore scaling. *SIGARCH Comput. Archit. News*, 39(3):365–376, June 2011.

[36] Hadi Esmaeilzadeh, Adrian Sampson, Luis Ceze, and Doug Burger. Neural acceleration for general-purpose approximate programs. In *Proceedings of the 2012 45th Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO-45, page 449–460, USA, 2012. IEEE Computer Society.

[37] João Fabrício Filho, Isaías Felzmann, Rodolfo Azevedo, and Lucas Wanner. A Resilient Interface for Approximate Data Access. In *2019 IX Brazilian Symposium on Computing Systems Engineering (SBESC)*, pages 1–8, nov 2019.

[38] João Fabrício Filho, Isaías Felzmann, Rodolfo Azevedo, and Lucas Wanner. AxRAM: A lightweight implicit interface for approximate data access. *Future Generation Computer Systems*, 113:556–570, 2020.

[39] João Fabrício Filho, Isaías Felzmann, and Lucas Wanner. Tratamento de Ponteiros Incorretos armazenados em Memórias Aproximadas. In *10ª Escola Regional de Alto Desempenho de São Paulo*, ERAD-SP'2019, Campinas, 2019.

[40] João Fabrício Filho, Isaías Felzmann, and Lucas Wanner. Sensibilidade a erros em aplicações na arquitetura RISC-V. In *11ª Escola Regional de Alto Desempenho de São Paulo*, ERAD-SP'2020, 2020.

[41] João Fabrício Filho, Isaías Felzmann, and Lucas Wanner. Transparent resilience for approximate DRAM. In Christian Hochberger, Lars Bauer, and Thilo Pionteck, editors, *Architecture of Computing Systems*, pages 35–50, Cham, 2021. Springer International Publishing.

[42] João Fabrício Filho, Isaías Felzmann, and Lucas Wanner. Approximate memory with protected static allocation. In *2022 IEEE 34th International Symposium on Computer Architecture and High Performance Computing (SBAC-PAD)*, pages 51–59, 2022.

[43] João Fabrício Filho, Isaías Felzmann, and Lucas Wanner. Smartapprox: Learning-based configuration of approximate memories for energy-efficient execution. *Sustainable Computing: Informatics and Systems*, 34:100701, 2022.

[44] Isaías Felzmann, João Fabrício Filho, Rodolfo Azevedo, and Lucas Wanner. Impact of Memory Approximation on Energy Efficiency. In *2018 Symposium on High Performance Computing Systems (WSCAD)*, pages 53–60, 2018.

[45] Isaías Felzmann, João Fabrício Filho, Juliane Regina de Oliveira, and Lucas Wanner. Special Session: How much quality is enough quality? A case for acceptability in approximate designs. In *2021 IEEE 39th International Conference on Computer Design (ICCD)*, pages 5–8, 2021.

[46] Isaías Felzmann, João Fabrício Filho, and Lucas Wanner. Risk-5: Controlled approximations for RISC-V. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 39(11), 2020.

[47] Isaías Felzmann, João Fabrício Filho, and Lucas Wanner. AxPIKE: Instruction-level Injection and Evaluation of Approximate Computing. In *2021 Design, Automation & Test in Europe Conference & Exhibition (DATE)*, pages 491–494, 2021.

[48] Isaías Felzmann, Matheus M. Susin, Liana Duenha, Rodolfo Azevedo, and Lucas Wanner. ADeLe: Rapid Architectural Simulation for Approximate Hardware. In *2018 30th International Symposium on Computer Architecture and High Performance Computing (SBAC-PAD)*, pages 9–16, 2018.

[49] Isaías Felzmann, Matheus M. Susin, Liana Duenha, Rodolfo Azevedo, and Lucas Wanner. ADeLe: A description language for approximate hardware. *Future Generation Computer Systems*, 102:245 – 258, 2020.

[50] S. Geetha and P. Amritvalli. High Speed Error Tolerant Adder for Multimedia Applications. *Journal of Electronic Testing*, 33(5):675–688, oct 2017.

[51] Hasan Genc, Seah Kim, Alon Amid, Ameer Haj-Ali, Vighnesh Iyer, Pranav Prakash, Jerry Zhao, Daniel Grubb, Harrison Liew, Howard Mao, Albert Ou, Colin Schmidt, Samuel Steffl, John Wright, Ion Stoica, Jonathan Ragan-Kelley, Krste Asanovic, Borivoje Nikolic, and Yakun Sophia Shao. Gemmini: Enabling systematic deep-learning architecture evaluation via full-stack integration. In *Proceedings of the 58th Annual Design Automation Conference (DAC)*, 2021.

[52] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Deep residual learning for image recognition, 2015.

[53] Xin He, Shuhao Jiang, Wenyan Lu, Guihai Yan, Yinhe Han, and Xiaowei Li. Exploiting the potential of computation reuse through approximate computing. *IEEE Transactions on Multi-Scale Computing Systems*, 3(3):152–165, 2017.

[54] Xin He, Liu Ke, Wenyan Lu, Guihai Yan, and Xuan Zhang. AxTrain: Hardware-Oriented Neural Network Training for Approximate Inference. In *Proceedings of the International Symposium on Low Power Electronics and Design*, page 20. ACM, 2018.

[55] N. Ho, E. Manogaran, W. Wong, and A. Anoosheh. Efficient floating point precision tuning for approximate computing. In *2017 22nd Asia and South Pacific Design Automation Conference (ASP-DAC)*, pages 63–68, 2017.

[56] Jeremy Howard. imagenette. `https://github.com/fastai/imagenette/`.

[57] Gee-Sern Hsu, Jiun-Chang Chen, and Yu-Zu Chung. Application-oriented license plate recognition. *IEEE Trans. Veh. Technol.*, 2012.

[58] Zhe Jiang, Xiaotian Dai, and Neil Audsley. HIART-MCS: High resilience and approximated computing architecture for imprecise mixed-criticality systems. In *2021 IEEE Real-Time Systems Symposium (RTSS)*, pages 290–303, 2021.

[59] Zhe Jiang, Xiaotian Dai, Alan Burns, Neil Audsley, Zonghua Gu, and Ian Gray. A high-resilience imprecise computing architecture for mixed-criticality systems. *IEEE Transactions on Computers*, 72(1):29–42, 2023.

[60] Jiayuan Meng, S. Chakradhar, and A. Raghunathan. Best-effort parallel execution framework for recognition and mining applications. In *2009 IEEE International Symposium on Parallel Distributed Processing*, pages 1–12, May 2009.

[61] Andrew B. Kahng and Seokhyeong Kang. Accuracy-configurable adder for approximate arithmetic designs. In *Proceedings of the 49th Annual Design Automation Conference*, DAC '12, page 820–825, New York, NY, USA, 2012. ACM.

[62] N Kapadia and S Pasricha. A runtime framework for robust application scheduling with adaptive parallelism in the dark-silicon era. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, 25(2):534–546, 2017.

[63] G Karakonstantis, G Panagopoulos, and K Roy. HERQULES: System level cross-layer design exploration for efficient energy-quality trade-offs. In *2010 ACM/IEEE International Symposium on Low-Power Electronics and Design (ISLPED)*, pages 117–122, 2010.

[64] Georgios Karakonstantis, Debabrata Mohapatra, and Kaushik Roy. Logic and Memory Design Based on Unequal Error Protection for Voltage-scalable, Robust and Adaptive DSP Systems. *Journal of Signal Processing Systems*, 68(3):415–431, 2012.

[65] U R Karpuzcu, I Akturk, and N S Kim. Accordion: Toward soft Near-Threshold Voltage Computing. In *2014 IEEE 20th International Symposium on High Performance Computer Architecture (HPCA)*, pages 72–83, 2014.

[66] Michael Keating, David Flynn, Robert Aitken, Alan Gibbons, and Kaijian Shi. Designing power gating. In *Low Power Methodology Manual: For System-on-Chip Design*, pages 41–73. Springer US, Boston, MA, 2007.

[67] D. Kim, A. Izraelevitz, C. Celio, H. Kim, B. Zimmer, Y. Lee, J. Bachrach, and K. Asanovicc. Strober: Fast and accurate sample-based energy simulation for arbitrary RTL. In *2016 ACM/IEEE 43rd Annual International Symposium on Computer Architecture (ISCA)*, pages 128–139, 2016.

[68] N S Kim, T Austin, D Baauw, T Mudge, K Flautner, J S Hu, M J Irwin, M Kandemir, and V Narayanan. Leakage current: Moore's law meets static power. *Computer*, 36(12), 2003.

[69] Nam Sung Kim and Ulya R Karpuzcu. Approximate Ultra-Low Voltage Many-Core Processor Design. In Sherief Reda and Muhammad Shafique, editors, *Approximate Circuits: Methodologies and CAD*, pages 371–382. Springer, Cham, 2019.

[70] Sunghyun Kim and Youngmin Kim. Adaptive approximate adder (A^3) to reduce error distance for image processor. In *2016 International SoC Design Conference (ISOCC)*, pages 295–296. IEEE, oct 2016.

[71] Sunghyun Kim and Youngmin Kim. Energy-efficient hybrid adder design by using inexact lower bits adder. In *2016 IEEE Asia Pacific Conference on Circuits and Systems (APCCAS)*, pages 355–357. IEEE, oct 2016.

[72] Sunwoong Kim and Rob A. Rutenbar. An Area-Efficient Iterative Single-Precision Floating-Point Multiplier Architecture for FPGA. In *Proceedings of the 2019 on Great Lakes Symposium on VLSI - GLSVLSI '19*, pages 87–92, New York, New York, USA, 2019. ACM.

[73] Y. Kim, W. Yang, and O. Mutlu. Ramulator: A fast and extensible dram simulator. *IEEE Computer Architecture Letters*, 15(1):45–49, 2016.

[74] Younghoon Kim, Swagath Venkataramani, Sanchari Sen, and Anand Raghunathan. Value similarity extensions for approximate computing in general-purpose processors. In *2021 Design, Automation & Test in Europe Conference & Exhibition (DATE)*, pages 481–486, 2021.

[75] Skanda Koppula, Lois Orosa, A. Giray Yağlıkçı, Roknoddin Azizi, Taha Shahroodi, Konstantinos Kanellopoulos, and Onur Mutlu. EDEN: Enabling energy-efficient, high-performance deep neural network inference using approximate dram. In *Proceedings of the 52nd Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO '52, page 166–181, New York, NY, USA, 2019. Association for Computing Machinery.

[76] Logan Kugler. Is "good enough" computing good enough? *Commun. ACM*, 58(5):12–14, apr 2015.

[77] Parag Kulkarni, Puneet Gupta, and Milos Ercegovac. Trading Accuracy for Power with an Underdesigned Multiplier Architecture. In *2011 24th Internatioal Conference on VLSI Design*, pages 346–351. IEEE, jan 2011.

[78] Michael O. Lam, Jeffrey K. Hollingsworth, Bronis R. de Supinski, and Matthew P. Legendre. Automatically adapting programs for mixed-precision floating-point computation. In *Proceedings of the 27th International ACM Conference on International Conference on Supercomputing*, ICS '13, page 369–378, New York, NY, USA, 2013. Association for Computing Machinery.

[79] Seogoo Lee, Lizy K. John, and Andreas Gerstlauer. High-level Synthesis of Approximate Hardware Under Joint Precision and Voltage Scaling. In *Conference on Design, Automation & Test in Europe*, pages 187–192, 3001 Leuven, Belgium, Belgium, 2017. European Design and Automation Association.

[80] Yunsup Lee. *Decoupled Vector-Fetch Architecture with a Scalarizing Compiler*. PhD thesis, EECS Department, University of California, Berkeley, May 2016.

[81] L Leem, H Cho, J Bau, Q A Jacobson, and S Mitra. ERSA: Error Resilient System Architecture for probabilistic applications. In *2010 Design, Automation Test in Europe Conference Exhibition (DATE 2010)*, pages 1560–1565, 2010.

[82] Jamie Liu, Ben Jaiyen, Richard Veras, and Onur Mutlu. RAIDR: Retention-aware intelligent DRAM refresh. In *Proceedings of the 39th Annual International Symposium on Computer Architecture*, ISCA '12, page 1–12, USA, 2012. IEEE Computer Society.

[83] W Liu, F Lombardi, and M Shulte. A Retrospective and Prospective View of Approximate Computing [Point of View]. *Proceedings of the IEEE*, 108(3):394–399, 2020.

[84] Jan Lucas, Mauricio Alvarez-Mesa, Michael Andersch, and Ben Juurlink. Sparkk: Quality-Scalable Approximate Storage in DRAM. In *The Memory Forum*, pages 1–6, 2014.

[85] Anita Lungu, Pradip Bose, Alper Buyuktosunoglu, and Daniel J. Sorin. Dynamic power gating with quality guarantees. In *Proceedings of the 2009 ACM/IEEE International Symposium on Low Power Electronics and Design*, ISLPED '09, page 377–382, New York, NY, USA, 2009. Association for Computing Machinery.

[86] S. Mach, D. Rossi, G. Tagliavini, A. Marongiu, and L. Benini. A transprecision floating-point architecture for energy-efficient embedded computing. In *2018 IEEE International Symposium on Circuits and Systems (ISCAS)*, pages 1–5, 2018.

[87] Daniel Maier, Biagio Cosenza, and Ben Juurlink. Local memory-aware kernel perforation. In *Proceedings of the 2018 International Symposium on Code Generation and Optimization*, CGO 2018, page 278–287, New York, NY, USA, 2018. Association for Computing Machinery.

[88] Ramy Medhat, Michael O. Lam, Barry L. Rountree, Borzoo Bonakdarpour, and Sebastian Fischmeister. Managing the performance/error tradeoff of floating-point intensive applications. *ACM Trans. Embed. Comput. Syst.*, 16(5s), October 2017.

[89] Sasa Misailovic, Daniel M. Roy, and Martin C. Rinard. Probabilistically accurate program transformations. In Eran Yahav, editor, *Static Analysis*, pages 316–333, Berlin, Heidelberg, 2011. Springer Berlin Heidelberg.

[90] Sasa Misailovic, Stelios Sidiroglou, Henry Hoffmann, and Martin Rinard. Quality of service profiling. In *Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering - Volume 1*, page 25–34, New York, NY, USA, 2010. Association for Computing Machinery.

[91] Sparsh Mittal. A survey of techniques for approximate computing. *Computing Surveys*, 48(4), March 2016.

[92] D. Mohapatra, V. K. Chippa, A. Raghunathan, and K. Roy. Design of voltage-scalable meta-functions for approximate computing. In *2011 Design, Automation Test in Europe*, pages 1–6, March 2011.

[93] Debabrata Mohapatra, Georgios Karakonstantis, and Kaushik Roy. Significance Driven Computation: A Voltage-Scalable, Variation-Aware, Quality-Tuning Motion Estimator. In *Proceedings of the 2009 ACM/IEEE International Symposium on Low Power Electronics and Design*, ISLPED '09, pages 195–200, New York, NY, USA, 2009. Association for Computing Machinery.

[94] Bert Moons and Marian Verhelst. DVAS: Dynamic Voltage Accuracy Scaling for increased energy-efficiency in approximate computing. In *International Symposium on Low Power Electronics and Design*, volume 2015-Septe, pages 237–242. IEEE, jul 2015.

[95] Thierry Moreau, Mark Wyse, Jacob Nelson, Adrian Sampson, Hadi Esmaeilzadeh, Luis Ceze, and Mark Oskin. SNNAP: Approximate computing on programmable SoCs via neural acceleration. In *International Symposium on High Performance Computer Architecture*, pages 603–614. IEEE, feb 2015.

[96] V. Mrazek, R. Hrbacek, Z. Vasicek, and L. Sekanina. EvoApprox8b: Library of approximate adders and multipliers for circuit design and benchmarking of approximation methods. In *Design, Automation Test in Europe Conference Exhibition (DATE), 2017*, pages 258–261, March 2017.

[97] Tomoki Nakamura, Kazutaka Tomida, Shouta Kouno, Hidetsugu Irie, and Shuichi Sakai. Stochastic iterative approximation: Software/hardware techniques for adjusting aggressiveness of approximation. In *2021 IEEE 39th International Conference on Computer Design (ICCD)*, pages 74–82, 2021.

[98] Farzaneh Nakhaee, Mehdi Kamal, Ali Afzali-Kusha, Massoud Pedram, Sied Mehdi Fakhraie, and Hamed Dorosti. Lifetime improvement by exploiting aggressive voltage scaling during runtime of error-resilient applications. *Integration*, 61:29 – 38, 2018.

[99] Sriram Narayanan, John Sartori, Rakesh Kumar, and Douglas L Jones. Scalable Stochastic Processors. In *Proceedings of the Conference on Design, Automation and Test in Europe*, DATE '10, pages 335–338, Leuven, BEL, 2010. European Design and Automation Association.

[100] Geneviève Ndour, Tiago Trevisan Jost, Anca Molnos, Yves Durand, and Arnaud Tisserand. 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*, SAMOS '18, pages 146–149, New York, NY, USA, 2018. Association for Computing Machinery.

[101] Geneviève Ndour, Tiago Trevisan Jost, Anca Molnos, Yves Durand, and Arnaud Tisserand. Evaluation of Variable Bit-Width Units in a RISC-V Processor for Approximate Computing. In *Proceedings of the 16th ACM International Conference on Computing Frontiers*, CF '19, pages 344–349, New York, NY, USA, 2019. ACM.

[102] J. Jean Jenifer Nesam and S. Sivanantham. An area-efficient 32-bit floating point multiplier using hybrid GPPs addition. In *2017 International conference on Microelectronic Devices, Circuits and Systems (ICMDCS)*, pages 1–4. IEEE, aug 2017.

[103] Mohammad Taghi Teimoori Nodeh, Mostafa Bazzaz, and Alireza Ejlali. Exploiting approximate MLC-PCM in low-power embedded systems. *ACM Trans. Embed. Comput. Syst.*, 17(1), December 2017.

[104] Bernard Nongpoh, Rajarshi Ray, and Ansuman Banerjee. Approximate computing for multithreaded programs in shared memory architectures. In *Proceedings of the 17th ACM-IEEE International Conference on Formal Methods and Models for System Design*, MEMOCODE '19, New York, NY, USA, 2019. Association for Computing Machinery.

[105] Bernard Nongpoh, Rajarshi Ray, Moumita Das, and Ansuman Banerjee. Enhancing speculative execution with selective approximate computing. *ACM Trans. Des. Autom. Electron. Syst.*, 24(2), feb 2019.

[106] Mario Osta, Ali Ibrahim, Hussein Chible, and Maurizio Valle. Inexact Arithmetic Circuits for Energy Efficient IoT Sensors Data Processing. In *2018 IEEE International Symposium on Circuits and Systems (ISCAS)*, pages 1–4. IEEE, may 2018.

[107] Konstantinos Parasyris, Vassilis Vassiliadis, Christos D Antonopoulos, Spyros Lalis, and Nikolaos Bellas. Significance-Aware Program Execution on Unreliable Hardware. *ACM Trans. Archit. Code Optim.*, 14(2), 2017.

[108] Jongse Park, Emmanuel Amaro, Divya Mahajan, Bradley Thwaites, and Hadi Esmaeilzadeh. Axgames: Towards crowdsourcing quality target determination in approximate computing. In *Proceedings of the Twenty-First International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '16, page 623–636, New York, NY, USA, 2016. Association for Computing Machinery.

[109] M. Powell, Se-Hyun Yang, B. Falsafi, K. Roy, and T. N. Vijaykumar. Gated-v/sub dd/: a circuit technique to reduce leakage in deep-submicron cache memories. In *ISLPED'00: Proceedings of the 2000 International Symposium on Low Power Electronics and Design (Cat. No.00TH8514)*, pages 90–95, July 2000.

[110] Bharath Srinivas Prabakaran, Semeen Rehman, and Muhammad Shafique. XBioSiP: A Methodology for Approximate Bio-Signal Processing at the Edge. In *Proceedings of the 56th Annual Design Automation Conference 2019*, DAC '19, New York, NY, USA, 2019. Association for Computing Machinery.

[111] K. Qiu, J. Luo, Z. Gong, W. Zhang, J. Wang, Y. Xu, T. Li, and C. J. Xue. Refresh-aware loop scheduling for high performance low power volatile stt-ram. In *2016 IEEE 34th International Conference on Computer Design (ICCD)*, pages 209–216, 2016.

[112] Rengarajan Ragavan, Benjamin Barrois, Cedric Killian, and Olivier Sentieys. Pushing the limits of voltage over-scaling for error-resilient applications. In *Design, Automation and Test in Europe*, pages 476–481. IEEE, mar 2017.

[113] A. Raha, S. Sutar, H. Jayakumar, and V. Raghunathan. Quality configurable approximate DRAM. *IEEE Transactions on Computers*, 66(7):1172–1187, July 2017.

[114] A Rahimi, A Marongiu, R K Gupta, and L Benini. A variability-aware OpenMP environment for efficient execution of accuracy-configurable computation on shared-FPU processor clusters. In *2013 International Conference on Hardware/Software Codesign and System Synthesis (CODES+ISSS)*, pages 1–10, 2013.

[115] Ashish Ranjan, Swagath Venkataramani, Zoha Pajouhi, Rangharajan Venkatesan, Kaushik Roy, and Anand Raghunathan. STAxCache: an approximate, energy efficient STT-MRAM cache. In *Proceedings of the Conference on Design, Automation & Test in Europe*, pages 356–361. European Design and Automation Association, 2017.

[116] Rekor Systems. OpenALPR - Automatic License Plate Recognition. `https://github.com/openalpr/openalpr`.

[117] Martin Rinard. Probabilistic accuracy bounds for fault-tolerant computations that discard tasks. In *Proceedings of the 20th Annual International Conference on Supercomputing*, ICS '06, page 324–334, New York, NY, USA, 2006. Association for Computing Machinery.

[118] Gennaro S Rodrigues, Juan Fonseca, Fabio Benevenuti, Fernanda Kastensmidt, and Alberto Bosio. Exploiting Approximate Computing for Low-Cost Fault Tolerant Architectures. In *Proceedings of the 32nd Symposium on Integrated Circuits and Systems Design*, SBCCI '19, New York, NY, USA, 2019. Association for Computing Machinery.

[119] Pooja Roy, Rajarshi Ray, Chundong Wang, and Weng Fai Wong. ASAC: Automatic sensitivity analysis for approximate computing. In *Proceedings of the 2014 SIGPLAN/SIGBED Conference on Languages, Compilers and Tools for Embedded Systems*, LCTES '14, page 95–104, New York, NY, USA, 2014. Association for Computing Machinery.

[120] Cindy Rubio-González, Cuong Nguyen, Hong Diep Nguyen, James Demmel, William Kahan, Koushik Sen, David H. Bailey, Costin Iancu, and David Hough. Precimonious: Tuning assistant for floating-point precision. In *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*, SC '13, New York, NY, USA, 2013. Association for Computing Machinery.

[121] A. Sampson, A. Baixo, B. Ransford, T. Moreau, J. Yip, L. Ceze, and M. Oskin. ACCEPT: A Programmer-Guided Compiler Framework for Practical Approximate Computing. Technical report, University of Washington, UW-CSE, 01 2015.

[122] Adrian Sampson, Werner Dietl, Emily Fortuna, Danushen Gnanapragasam, Luis Ceze, and Dan Grossman. EnerJ: Approximate Data Types for Safe and General Low-power Computation. *SIGPLAN Not.*, 46(6), 2011.

[123] Doochul Shin and Sandeep K Gupta. Approximate logic synthesis for error tolerant applications. In *2010 Design, Automation & Test in Europe Conference & Exhibition (DATE 2010)*, pages 957–960. IEEE, mar 2010.

[124] Qilin Si, Prattay Chowdhury, Rohit Sreekumar, and Benjamin Carrion Schafer. Application specific approximate behavioral processor. *IEEE Transactions on Sustainable Computing*, 8(2):165–179, 2023.

[125] Stelios Sidiroglou-Douskos, Sasa Misailovic, Henry Hoffmann, and Martin Rinard. 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*, ESEC/FSE '11, page 124–134, New York, NY, USA, 2011. ACM.

[126] G Tagliavini, D Rossi, L Benini, and A Marongiu. Synergistic Architecture and Programming Model Support for Approximate Micropower Computing. In *2015 IEEE Computer Society Annual Symposium on VLSI*, pages 280–285, 2015.

[127] G Tagliavini, D Rossi, A Marongiu, and L Benini. Synergistic HW/SW Approximation Techniques for Ultra-Low-Power Parallel Computing. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 37(5):982–995, 2018.

[128] Ibrahim Taştan, Mahmut Karaca, and Arda Yurdakul. Approximate CPU Design for IoT End-Devices with Learning Capabilities. *Electronics*, 9(1):125, jan 2020.

[129] F Tu, S Yin, P Ouyang, L Liu, and S Wei. Reconfigurable Architecture for Neural Approximation in Multimedia Computing. *IEEE Transactions on Circuits and Systems for Video Technology*, 29(3):892–906, 2019.

[130] University of Iowa Electronic Music Studios. Musical instrumental samples. `http://theremin.music.uiowa.edu/MIS.html`.

[131] R Venkatagiri, A Mahmoud, S K S Hari, and S V Adve. Approxilyzer: Towards a systematic framework for instruction-level approximate computing and its application to hardware resiliency. In *MICRO*, pages 1–14, 2016.

[132] S Venkataramani, V K Chippa, S T Chakradhar, K Roy, and A Raghunathan. Quality programmable vector processors for approximate computing. In *2013 46th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pages 1–12, 2013.

[133] S. Venkataramani, A. Sabne, V. Kozhikkottu, K. Roy, and A. Raghunathan. SALSA: Systematic logic synthesis of approximate circuits. In *DAC Design Automation Conference 2012*, pages 796–801, June 2012.

[134] Yan Verdeja Herms and Yanjing Li. Crash Skipping: A Minimal-Cost Framework for Efficient Error Recovery in Approximate Computing Environments. In *Proceedings of the 2019 on Great Lakes Symposium on VLSI*, GLSVLSI '19, pages 129–134, New York, NY, USA, 2019. Association for Computing Machinery.

[135] Zhou Wang, A.C. Bovik, H.R. Sheikh, and E.P. Simoncelli. Image quality assessment: from error visibility to structural similarity. *IEEE Transactions on Image Processing*, 13(4):600–612, 2004.

[136] L Wanner, S Elmalaki, L Lai, P Gupta, and M Srivastava. VarEMU: An Emulation Testbed for Variability-aware Software. In *CODES+ISSS*, 2013.

[137] Andrew Waterman and Krste Asanovic. The RISC-V Instruction Set Manual - Volume II: Privileged Architecture, 2019.

[138] Mark Wyse, Andre Baixo, Thierry Moreau, Bill Zorn, James Bornholt, Adrian Sampson, Luis Ceze, and Mark Oskin. REACT: A Framework for Rapid Exploration of Approximate Computing Techniques. In *Workshop on Approximate Computing Across the Stack*, 2015.

[139] S Xu and B C Schafer. Toward Self-Tunable Approximate Computing. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, 27(4):778–789, 2019.

[140] Lei Yang, Weichen Liu, Weiwen Jiang, Chao Chen, Mengquan Li, Peng Chen, and Edwin H M Sha. Hardware-software collaboration for dark silicon heterogeneous many-core systems. *Future Generation Computer Systems*, 68:234–247, 2017.

[141] Zhixi Yang, Ajaypat Jain, Jinghang Liang, Jie Han, and Fabrizio Lombardi. Approximate XOR/XNOR-based adders for inexact computing. In *2013 13th IEEE International Conference on Nanotechnology (IEEE-NANO 2013)*, pages 690–693. IEEE, aug 2013.

[142] Hasan Erdem Yantir, Ahmed M. Eltawil, and Fadi J. Kurdahi. Approximate Memristive In-memory Computing. *ACM Transactions on Embedded Computing Systems*, 16(5s):1–18, sep 2017.

[143] Yi-Ping You, Chung-Wen Huang, and Jenq Kuen Lee. Compilation for compact power-gating controls. *ACM Trans. Des. Autom. Electron. Syst.*, 12(4):51–es, September 2007.

[144] F. Zaruba and L. Benini. The cost of application-class processing: Energy and performance analysis of a Linux-ready 1.7-GHz 64-bit RISC-V core in 22-nm FDSOI technology. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, 27(11):2629–2640, 2019.

[145] Ziji Zhang, Yajuan He, Jin He, Xilin Yi, Qiang Li, and Bo Zhang. Optimal slope ranking: an approximate computing approach for circuit pruning. In *IEEE International Symposium on Circuits & Systems (ISCAS)*, pages 1–4. IEEE, may 2018.

[146] Jerry Zhao, Ben Korpan, Abraham Gonzalez, and Krste Asanovic. SonicBOOM: The 3rd generation Berkeley Out-of-Order Machine. In *Fourth Workshop on Computer Architecture Research with RISC-V*, May 2020.

[147] Zhigang Hu, A. Buyuktosunoglu, V. Srinivasan, V. Zyuban, H. Jacobson, and P. Bose. Microarchitectural techniques for power gating of execution units. In *Proceedings of the 2004 International Symposium on Low Power Electronics and Design (IEEE Cat. No.04TH8758)*, pages 32–37, Aug 2004.