João Fabrício Filho

Software and Hardware Interfaces for Approximate Memories

Interfaces Software e Hardware para Memórias Aproximadas

CAMPINAS
2022

João Fabrício Filho

Software and Hardware Interfaces for Approximate Memories

Interfaces Software e Hardware para Memórias Aproximadas

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

CAMPINAS
2022

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

F114s

Fabrício Filho, João, 1990-
Software and hardware interfaces for approximate memories / João Fabrício Filho. – Campinas, SP : [s.n.], 2022.

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

1. Computação aproximada. 2. Computação consciente de energia. 3. Sistemas de memória de computadores. 4. Arquitetura de computador. I. Wanner, Lucas Francisco, 1981-. II. Universidade Estadual de Campinas. Instituto de Computação. III. Título.

Informações para Biblioteca Digital

Título em outro idioma: Interfaces software e hardware para memórias aproximadas
Palavras-chave em inglês:
Approximate computing
Energy-aware computing
Computer storage devices
Computer architecture
Área de concentração: Ciência da Computação
Titulação: Doutor em Ciência da Computação
Banca examinadora:
Lucas Francisco Wanner [Orientador]
Antonio Carlos Schneider Beck Filho
Marco Antonio Zanata Alves
Edson Borin
Sandro Rigo
Data de defesa: 10-02-2022
Programa de Pós-Graduação: Ciência da Computação

Identificação e informações acadêmicas do(a) aluno(a)
- ORCID do autor: https://orcid.org/0000-0001-6036-4031
- Currículo Lattes do autor: http://lattes.cnpq.br/3874284354315074

Universidade Estadual de Campinas
Instituto de Computação

João Fabrício Filho

Software and Hardware Interfaces for Approximate Memories

Interfaces Software e Hardware para Memórias Aproximadas

Campinas, 10 de fevereiro de 2022

Acknowledgements

I would like to thank the many people that give me support during this Ph.D.

To my wife, Aline Maria Gonçalves Fabrício, for walking alongside me on this journey giving me support, and making me happy every day.

To my advisor Lucas Wanner, for all the mentoring and guidance that were fundamental in formulating research questions and in my professional development.

To Isaías Bittencourt Felzmann, for the feedback, ideas, collaborations on manuscripts, and the several discussions about the topics of this research.

To Rodolfo Azevedo, for the support that improves the contributions of this work.

To the committee members Edson Borin, Sandro Rigo, Marco Antonio Zanata Alves, and Antonio Carlos Beck Filho for their time and interest.

To Diego Bertolini Gonçalves, for helping to clear up some doubts.

To Jonathas Silveira and Juliane Oliveira, for the collaborations.

Last but not least, I want to thank my parents João Fabrício (*in memoriam*) and Helena de França Fabrício, for their life example and encouragement.

Copyright and re-use of published material

This dissertation contains significant material that has been published or is intended to be published. Section 3.2 is mostly based on the content published in Future Generation Computer Systems 113 [26], but also has content based on the papers from the Brazilian Symposium on Computing Systems Engineering (SBESC) 2019 (© IEEE 2019 reprinted, with permission from [25]) and the Regional School of High-Performance Computing of São Paulo (ERAD-SP) 2019 [27]. The exploration of transparent resilience mechanisms from Section 3.3 is based on the content of the work presented on the International Conference on Architecture of Computing Systems (ARCS) 2021 [29] and the ERAD-SP 2020 [28], while Section 3.4 contains part of an ongoing paper, planned to be submitted soon. Chapter 4 has most of its content based on the work presented at the International Green and Sustainable Computing Conference (IGSC) 2021 that is published in Sustainable Computing: Informatics and Systems, in a special edition integrated with the proceedings of IGSC [30]. The titles of each chapter and section have been changed somewhat to differentiate them from the published version when applicable.

Resumo

Componentes de memória são sensíveis a variações de processo, tensão e temperatura e, para assegurar confiabilidade nos dados armazenados, seus fabricantes especificam os parâmetros de operação considerando o pior caso de projeto com uma margem de proteção. Memórias aproximadas ajustam os parâmetros para fora da margem de proteção, o que permite economia de energia ao custo de erros probabilísticos nos dados armazenados. Enquanto diversas aplicações toleram alguma imprecisão em seus resultados, não são todos os seus dados que toleram erros, e até dados resilientes os toleram até um limite. Erros não controlados podem produzir resultados com mais imprecisão do que o aceitável, o que os torna inúteis, ou até quebrar uma execução da aplicação inesperadamente por causa de erros em dados críticos. Interfaces para acesso a dados controlam quais dados são expostos a erros, por meio da proteção a dados críticos, e quanto erro pode ser inserido para respeitar o limite de imprecisão dos resultados, por meio da configuração da memória aproximada. Esse controle depende da aplicação e do impacto do erro nos resultados da computação. Interfaces tipicamente fiam-se em anotações do programador e métricas de domínio específico para identificar os dados suscetíveis a aproximações ou para configurar a quantidade de erros permitida. Contudo, anotações prejudicam a portabilidade e manutenção do código e métricas de domínio específico demandam algum conhecimento sobre o impacto do erro na aplicação. Além disso, o erro é um elemento dinâmico dependente de um cenário composto de variáveis do ambiente, como temperatura, localidade e processo de fabricação. Mecanismos de interfaces transparentes procuram, de forma automática, proteger dados críticos e controlar o erro no limite aceitável da aplicação, alterando a configuração de erro de acordo com as variáveis do ambiente. Este trabalho propõe interfaces transparentes para o controle de aproximação de memória que melhoram a resiliência da execução e aumentam a eficiência energética. Analisamos a execução das aplicações quando elementos de dados são expostos a erros probabilísticos e encontramos dados críticos que causam quebras de execução e que são comuns a várias aplicações. Então, propusemos mecanismos de *hardware* e *software* para tratar esses dados e evitar quebras no intuito de gerar resultados úteis. Apresentamos proteções para endereçamento físico e virtual e investigamos o impacto do erro de diferentes posições na hierarquia de memória, além da exploração de alternativas para um sistema supervisor tratar execuções inválidas para recuperar os dados da aplicação. Finalmente, este trabalho relaciona o comportamento da execução das aplicações com sua tolerância a erros para configurar a memória aproximada de forma transparente. O comportamento é abstraído de estatísticas de execução mensuráveis que são correlacionadas com a configuração utilizando uma base de conhecimento de execuções de treinamento prévias. Nossos resultados evidenciam que as interfaces propostas melhoram a resiliência da execução reduzindo uma fração significativa das quebras e configuram memórias aproximadas com economia de energia e qualidade média próximas às alcançadas por uma busca exaustiva.

Abstract

Memory components are sensitive to process, voltage, and temperature variability and, to ensure reliability in the stored data, vendors specify operating parameters considering the worst case in the process design with a guard-band margin. Approximate memories adjust parameters out of the guard-band range, which allows for energy savings at the cost of probabilistic errors in the stored data. While several applications tolerate some imprecision in their results, not all data are resilient to errors, and even resilient data have limits on their tolerance to errors. Uncontrolled errors may produce results with more imprecision than acceptable, which render them useless or even crash an application execution unexpectedly because of errors in critical data. Data access interfaces control what data are exposed to errors, through critical data protection, and how much error can be inserted to respect the imprecision limit of the results, through the configuration of the approximate memory. This control depends on the application and the error impact in the results of the computation. Interfaces typically rely on programmer annotations that change the application and domain-specific metrics to identify the data amenable to approximation or to configure the allowed error amount. Nevertheless, annotations hinder the portability and maintainability of the code, and domain-specific metrics demand some knowledge about the error impact on the application. Moreover, the error is a dynamic element dependent on a scenario that is composed of environment variables, such as temperature and fabrication process. Transparent interface mechanisms attempt to automatically protect critical data and control the error into the threshold that the application tolerates changing the error configuration according to the environment variables. This work proposes transparent interfaces for the control of memory approximation to improve execution resilience and increase energy efficiency. We analyze the execution of applications when data elements are exposed to probabilistic errors and find common critical data that cause execution crashes. Then, we propose hardware and software mechanisms to treat these data and avoid crashes aiming to generate useful results. We present protections for physical and virtual addressing and investigate the impact of error from different memory hierarchy levels, besides the exploration of alternatives for a supervisor system to treat incorrect executions for recovering application data. Lastly, this work relates the execution behavior of applications with their error tolerance to transparently configure the approximate memory. The behavior is abstracted from measurable execution statistics that are correlated with a configuration using a knowledge base of previous training executions. Our results show that these interfaces improve execution resilience by reducing a significant part of the crashes and configure approximate memories with energy savings and average quality close to the achieved by an exhaustive search.

List of Figures

List of Tables

List of Abbreviations

AC:	Approximate Computing
AMA:	Approximate Memory Accesses
BFGS:	Broyden–Fletcher–Goldfarb–Shanno algorithm
bss:	Block Starting Symbol
CE:	Cache Efficiency
CPU:	Central Processing Unit
CS:	Crash Skipping
DIMM:	Dual Inline Memory Module
DNN:	Deep Neural Networks
DRAM:	Dynamic Random Access Memory
DS:	Data Size
DVFS:	Dynamic Voltage and Frequency Scaling
e.g.:	*Exempli Gratia*
EI:	Executed Instructions
ELF:	Executable and Linkable Format
FEE:	Fraction of Equal Elements
FEM:	Fraction of Elements within a Margin
GA:	Genetic Algorithm
IoT:	Internet of Things
ISA:	Instruction Set Architecture
LSB:	Least Significant Bits
MA:	Memory Accesses
MAPE:	Mean Absolute Percentage Error
MCC:	Memory Controller Commands
MEB:	Memory Energy Breakdown
MI:	Memory Instructions
ML:	Machine Learning
MLC:	Multi-Level Cells
MLP:	Multi-Layer Perceptron
MSB:	Most Significant Bits
MTJ:	Magnetic Tunneling Junction
NNA:	Nearest Neighbor Algorithm

NN:	Neural Network
OS:	Operating System
PCM:	Phase-Change Memory
Ph.D.:	Philosophiæ Doctor
pp:	Percentage Points
PSNR:	Peak Signal-to-Noise Ratio
PTE:	Page Table Entry
QEE:	Quality-Energy Efficiency
RBF:	Radial Basis Function
RF:	Random Forest
SDC:	Silent Data Corruption
SEU:	Single Event Upset
SRAM:	Static Random Access Memory
SSI:	Structural Similarity Index
STT-MRAM:	Spin-Transfer Torque Magnetoresistive RAM
SVM:	Support Vector Machine
TLB:	Translation Lookaside Buffer
VPN:	Virtual Page Number

Contents

Chapter 1

Introduction

Computation may not, and frequently cannot, be exact. Applications that involve signal processing, pattern recognition, and data analysis require results that are acceptable rather than accurate [62, 72]. The acceptance of these results involves limitations on human perception and statistical behavior of data patterns. Figure 1.1 shows examples of grayscale images affected by controlled errors in a given percentage of the pixels from the same baseline image. In these images, each pixel is an integer value that represents a scale from 0 (blank) to 255 (fully dark), and the errors are manifested through flipping random bits in the pixels. These errors may depreciate the perceived quality of the images without corrupting the perception of their content until a certain limit. Thus, image pixels are examples of data that tolerate some degree of approximation. Applications that tolerate approximation in their results have data resilient to errors. An opportunity emerges when computer systems have to deal with scaling challenges that increase the exposition to faults of the hardware components [22]. As the occurrence of faults increases, mechanisms to suppress their influence on computation have more impact on performance and energy efficiency [4, 43]. Thus, the relaxation of these mechanisms can provide improvements in performance and energy on error resilient applications.

Approximate Computing (AC) comprises techniques that deliberately allow errors on computational elements, exploring the inaccuracy toleration in the results, in exchange for energy benefits [79, 128]. Approximation techniques provide opportunities to improve

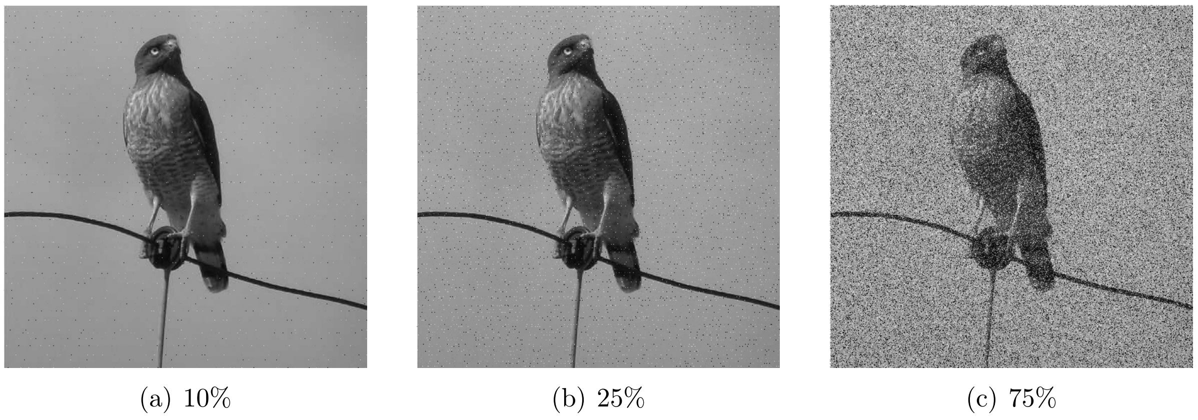

(a) 10% (b) 25% (c) 75%

Figure 1.1: Result of bit flipping a given percentage of the pixels from the same baseline image. *The results may not corrupt the entire content, we just notice a quality loss.*

the area, performance, and energy efficiency by relaxing accuracy constraints across the system layers, from applications to circuits [109]. However, uncontrolled errors lead to unexpected behaviors in the application, causing perturbations that may impact different aspects of the execution such as control flow, memory protection, and input and output integrity [115].

Approximations in memory components affect data storage in the application, which includes code, data, and by extension control flow. Effective use of approximate memories therefore requires protecting critical data such as code and function pointers, while allowing for energy efficient (and error-prone) operation for non-sensitive data such as pixels or time series. Storage and data access represent a significant fraction of overall energy usage in contemporary systems [12, 107]. For many applications, error-resilient data comprise the majority of memory usage, and therefore approximating these data can lead to significant improvements in energy efficiency.

1.1 Approximate Memories

Memories represent a key role in computational systems, especially with the applications executing and data workloads being produced by billions of low-power devices nowadays [1, 107]. Approximately 59 zettabytes were created and processed in 2020, and this number is expected to increase in each one of the next years [107]. Furthermore, memories represent a significant part of the total energy consumption of computational systems, achieving more than half on memory-intensive workloads [12]. Memory components are sensitive to circuit variation and, to improve reliability, vendors specify their nominal operating parameters with a guard-band margin for the worst case in the process design [13, 71, 94]. However, this margin brings overheads to the computing systems. For example, access latency is increased by almost 40% and the supply voltage of the data array is increased by 20% in the error-free operation for usual conditions, which impacts the energy consumption or performance [12, 13]. Thus, it is possible to explore a wide range of operating points with insignificant or even no errors.

Approximate memories allow for adjusting parameters out of the guard-band margin while exposing stored data to errors through a redesign of the memory data interface to reduce energy consumption [37]. These errors are nondeterministic and may occur at any point on the exposed data according to a probability related to the configuration parameters, which provides control of the approximation [130]. Figure 1.2 shows the relative energy consumption collected from the execution of diverse applications with an instance of an approximate DRAM face to the error probability per access in this memory, based on an error characterization of the DRAM voltage [12] executing on a controlled environment with a median instance of error rate [30]. The energy consumption decreases as the vdd is adjusted below the guard-band margin, however, the error probability grows exponentially at each step of the supply voltage. The errors must be controlled to ensure that their impact does not trespass the resilience of the applications.

Exposing all application data to errors may compromise the results, decreasing their accuracy or crashing the execution. Execution crashes are premature terminations of the

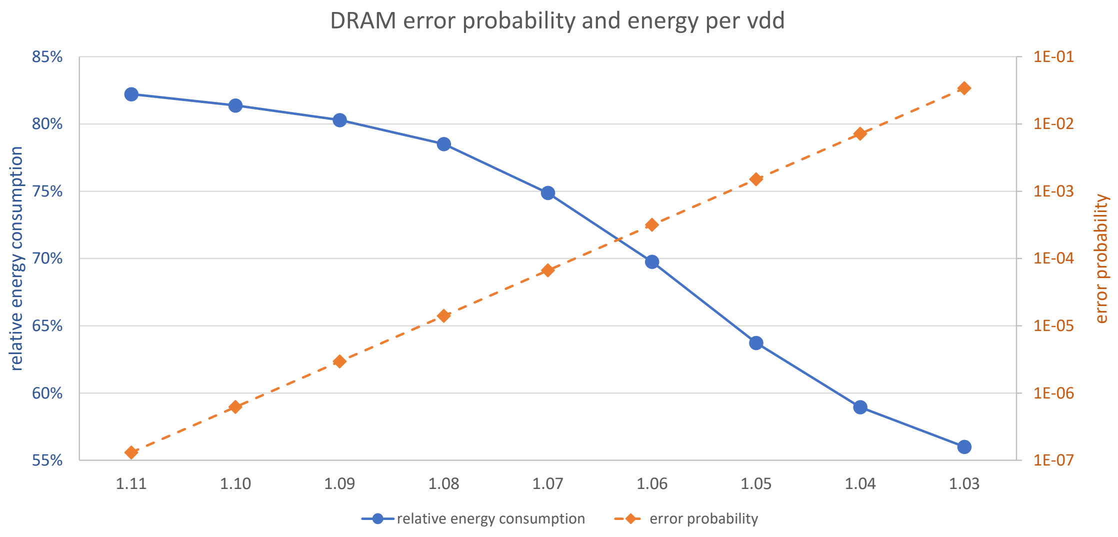


Figure 1.2: Relative energy consumption of several applications and error probability per access on an approximate DRAM with adjusted vdd. *As the energy consumption decreases, the error probability grows exponentially.*

execution flow before producing results [118]. Without an output, computational efforts and energy resources are wasted, resulting in decreased benefits. Interfaces that control approximations through the protection of data level usually separate application data into error-resilient and accurate [23, 103]. This separation is necessary to avoid errors in application data that are critical and do not tolerate errors, such as file headers and memory references. Errors on these data may cause crashes or nullify execution results, decreasing the average inaccuracy of the outputs and increasing the energy consumed by recovery mechanisms, such as re-executions. However, identifying all critical data depends on the data structures, inputs, approximation technique, and application context [8, 93]. Several interfaces [8, 9, 17, 20, 103] rely on annotations in the application to choose what data are approximate or protected from errors. While effective in protecting against crashes, these techniques bring additional complexity, since programmers need to worry about the approximation control and must have expert knowledge about the application data. Moreover, an approximation-specific layer of code must be supported over the lifetime of the application, jeopardizing the portability and maintainability of the code.

Applications tolerate inaccurate results until a certain limit. If errors accumulate and affect the output more than the tolerable limit, useless results are generated, wasting computational efforts and reducing potential benefits [112]. Configuration interfaces act to prevent such waste of effort by controlling the probability of error insertion in the computation or recalibrating the approximation at runtime [56, 129]. The configuration of an approximate memory is application-specific and depends on memory access patterns of the application [126]. Several configuration interfaces [56, 77, 112, 129, 130] also depend on annotations or on application-specific metrics to evaluate the acceptance of the results and adjust the parameters of the approximation as knobs. The additional complexity of the configuration is to tackle the error fluctuation that changes according to the hardware instance and environment, comprising an error scenario dependent on some variables, such as temperature, access delay, and fabrication process [13, 65].

1.2 Contributions

This work proposes transparent interfaces for approximate memories that improve execution resilience and configure the approximation knob without requiring domain-specific annotations and metrics from the application. Moreover, we present mechanisms for a supervisor system to avoid crashes and recover lost data resulting from the execution with approximate memories. To this end, we (1) present interfaces for the protection of critical data with recovery mechanisms that treat data that cause crashes to avoid data loss, (2) introduce a configuration interface that correlates execution statistics with the error tolerance of the applications, and (3) propose an architectural model that allows for the isolation of data regions from errors to store critical data.

In this work, we introduce transparent mechanisms that perform without domain-specific interventions in the application. In this context, transparent interfaces act without annotations in the source code to protect, recover, and configure the approximate environment based on general application behavior. Protection mechanisms identify critical data in common for many types of applications. Recovery considers features from a supervisor system that triggers re-executions when necessary and maps memory addresses to the application. Likewise, the configuration of an approximate memory determines the adjustment of the knob without domain-specific metrics or annotations that indicate the error tolerance of the application.

While the mechanisms introduced in this work act without requiring changes in the applications, such as modifications in the source code or recompilation, adaptations in the runtime and application support system may be required. These mechanisms require a supervisor system that controls the execution and manages the protected and approximate data. Environments without an Operating System (OS) need a runtime system to detect execution crashes and control the approximation. Moreover, transparent configuration demands some analysis of application behavior through architecture counters or execution statistics that depend on a representative input. Furthermore, some approximation control techniques of this work may be not fully transparent by interpreting input or output data of the application. A technique that requires data interpretation (e.g., timeout values and validation of the outputs) can be calibrated at design-time, alleviating the burden on the application programmer.

For protecting critical application data transparently, we classify execution crashes by the type of data that cause the deviation and verify that invalid memory references are the main cause of these lost executions for many applications running on systems with approximate memories. To validate incorrect memory references without annotations, we present an addressing scheme that recovers from execution crashes and transforms the incorrect reference into an access into allowed memory boundaries. Furthermore, we investigate hardware and software mechanisms that avoid and recover from a large fraction of critical errors, besides an evaluation of the impact of errors from different memory hierarchical levels.

To configure the approximate memory, we propose an analysis technique that describes the approximate behavior of applications as a function of common and easily extractable execution statistics, without requiring changes in the source code or domain-specific metrics. In this technique, prior knowledge is built upon the execution of training applications, where we extract the statistics as features of the applications and correlate them with the specified limit of inaccuracy to establish a configuration of the approximation knob. At runtime, features of new applications are sampled and knobs are adjusted to correspond to the predicted error tolerance, according to existing knowledge and the current error scenario, in consonance with previous hardware characterization. Thus, this interface acts transparently to the application and mitigates the dynamic fluctuations of the error.

Our architectural model allows the implementation of our interfaces based on an approximation technique that changes the supply voltage of the memory data array and provides two global values of voltage. Thus, two reliability modes are provided, which guarantees the execution of memory operations at reliable and approximate levels. The approximate level controls the error probability of the unreliable memory array through a register that defines the approximation knob.

We evaluate our interfaces through simulations with software models that replace memory accesses to expose the data array to a given error probability. The error probability is given by characterizations of errors from SRAM or DRAM from the literature [12, 119]. In our evaluation, the protection interface eliminates half of the execution crashes, while transparent mechanisms of a supervisor system achieve energy savings from 14% to 31%, depending on the application. Our configuration interface obtains 36% of average energy savings with acceptable output degradation and configures the approximate memory 97% closer to an ideal configuration with significantly lower effort and without application-specific metrics to analyze the quality of the results.

The main contributions of this work are:

1. An addressing scheme that recovers from execution crashes and improves execution resilience of applications that store data into approximate memories;

2. An analysis of the relation between error tolerance and application features that determines the configuration of the approximate memory;

3. A memory architecture model that allows the coexistence of accurate and approximate data of variable sizes in the same system;

4. A transparent runtime system that configures approximate memories according to different error scenarios that the application can be exposed with negligible overhead on the reconfiguration;

5. A study of the impact of error amongst levels of the memory hierarchy;

6. An exploration of transparent mechanisms for a supervisor system to detect and recover from invalid results generated on executions with approximate memories;

7. A study of classes of application features and their impact on the execution with approximate memories;

8. An evaluation featuring applications from several computing domains and an analysis of how they behave under the approximation of data elements.

During the development of the doctoral course, 12 papers were produced among collaborations and researches directly related to this work. Table 1.1 shows the list of these papers relating them with the listed contributions. The publications directly related to the Ph.D. research are featured in the body of the dissertation, differently from the papers that are contributions to other research efforts, which are not the focus of this work.

Table 1.1: List of papers produced during the Ph.D. *Papers that have a related contribution are partially on the body of this dissertation.*

Rel. contr.[2]	**Target**	**Title**	**Authors**	**Ref.**
1, 8	ERAD-SP 2019*	*Tratamento de Ponteiros Incorretos armazenados em Memórias Aproximadas*	**J. Fabrício Filho**, I. Felzmann, and L. Wanner	[27]
1, 3, 8	SBESC 2019	*A Resilient Interface for Approximate Data Access*	**J. Fabrício Filho**, I. Felzmann, R. Azevedo, and L. Wanner	[25]
1, 3, 6, 8	FGCS Dec/2020	*AxRAM: A lightweight implicit interface for approximate data access*	**J. Fabrício Filho**, I. Felzmann, R. Azevedo, and L. Wanner	[26]
8	ERAD-SP 2020	*Sensibilidade a erros em aplicações na arquitetura RISC-V*	**J. Fabrício Filho**, I. Felzmann, and L. Wanner	[28]
5, 6, 8	ARCS 2021	*Transparent Resilience for Approximate DRAM*	**J. Fabrício Filho**, I. Felzmann, and L. Wanner	[29]
2, 4, 7, 8	SUSCOM 2022	*SmartApprox: Learning-based Configuration of Approximate Memories for Energy-efficient Execution*	**J. Fabrício Filho**, I. Felzmann, and L. Wanner	[30]
6, 8	planned to 2022	*Transparent Approximate Heap*	**J. Fabrício Filho**, I. Felzmann, and L. Wanner	–
-	WSCAD 2018[1]	*Impact of Memory Approximation on Energy Efficiency*	I. Felzmann, **J. Fabrício Filho**, R. Azevedo, and L. Wanner	[32]
-	TCAD Nov/2020[1]	*Risk-5: Controlled approximations for RISC-V*	I. Felzmann, **J. Fabrício Filho**, and L. Wanner	[34]
-	WSCAD 2020[1]	*RV-Across: An Associative Processing Simulator*	J. E. Silveira, I. Felzmann, **J. Fabrício Filho**, and L. Wanner	[110]
-	DATE 2021*[1]	*AxPIKE: Instruction-level Injection and Evaluation of Approximate Computing*	I. Felzmann, **J. Fabrício Filho**, and L. Wanner	[35]
-	ICCD 2021[1]	*How Much Quality is Enough Quality? A Case for Acceptability in Approximate Designs*	I. Felzmann, **J. Fabrício Filho**, J. R. Oliveira, and L. Wanner	[33]

*best paper award
[1]collaboration
[2] related contributions of this dissertation

Chapter 2

Background and Related Work

Susceptibility to faults on modern hardware has increased with the technological scaling challenges in the dark silicon era [22]. Circuits are less reliable due to higher exposure to transient faults, and mechanisms to suppress and correct these faults are increasingly costly with a negative impact on performance and energy efficiency [4, 43, 68]. At the same time, applications of data mining, classification, and synthesis have emerged as a significant portion of global computational resources and energy consumption, from mobile devices to large-scale data centers [1, 14, 62]. For many of these applications that rely on massive volumes of data, exactness is not required or even possible.

Approximate Computing (AC) provides a spectrum of techniques to achieve performance or energy efficiency in applications that allow relaxing their accuracy requirements through controlled errors in computational elements [79, 80, 128]. The approximate elements can be at any layer of the computational system from hardware to application [109]. The approximation benefits depend on the accuracy requirements of the application results that usually rely on the acceptance of a well-defined statistical behavior of the computational outcome [94]. Approximation techniques have been applied to several contexts, including wireless sensor networks [3], Internet of Things (IoT) for health monitoring [41], Deep Neural Networks (DNN) [61], and image processing [54].

In this chapter, we present a review of AC techniques across system layers focusing on memory approximation techniques and characteristics that represent their behavior. Furthermore, we present interfaces that mitigate the approximation impact on the application, comparing their features with the proposal of this work.

2.1 Approximate Computing across System Layers

Approximation techniques explore system layers to obtain energy savings at the cost of inaccuracy in the computational results. AC techniques are classified into hardware and software approximations. In the following sections, we discuss these types of techniques, the error impact on the results, and how to measure this impact on the execution output.

2.1.1 Hardware Approximation

Hardware approximation techniques modify variables or structures to relax architectural accuracy. Two fundamental types of approximations have been proposed in the literature: replacement of a component by its approximate version, and changes on parameters of the architecture to ranges that expose the circuit to errors.

Techniques that replace hardware components with the approximate version can target basic circuits, such as adders [31, 39] and multipliers [47, 63, 87], complex architectural components, such as accelerators [24, 81], or even the entire circuit synthesis [117, 133]. Basic circuits components are widely used through architecture, such as on arithmetic logic units, address calculators, and increment operators. Hence, an approximate version of a component has the potential to achieve benefits by on several operations of the computer system. However, operations may be critical for the system and demand an accurate version of the circuit, reducing area gains. Accelerators are specialized components that improve performance by executing part or the entire computational job in dedicated hardware, and approximate accelerators replace this dedicated hardware by a mimic of its behavior, such as with Neural Network (NN) [24, 81]. Techniques that approximate circuit synthesis have the potential to achieve power and area savings on complex arithmetic circuits, blocks, or entire data paths [117]. This process can involve improvements in the resulting circuit by pruning or transformation algorithms [133].

Process variability may affect hardware components bringing a higher susceptibility to faults, and, thus, their vendors specify a large margin on the parameters to ensure reliable operation even in the worst-case design [94]. Approximation techniques that change parameters of the hardware make components operate below this guard-band, which perturbs some operation, exposing components to a noise that may cause circuit switching or timing failures [16, 19]. Unlike replacement techniques, parameters changes usually offer control of how much error is inserted into the approximate elements and, thus, are configurable techniques. The parameters are adjusted to unreliable ranges working as approximation knobs that control the error. Examples of such parameters are supply voltage [19, 100], frequency [123], and memory refresh rate [18, 71, 90]. The manifestation of errors depends on the approximate component, but these techniques usually are nondeterministic and the error probability has a defined relation with the approximation knob.

2.1.2 Software Approximation

Software approximation techniques do not depend on hardware to perform unreliable computation with application data. These techniques modify or even skip instructions or data through changes in the algorithm, code, compiler, or execution flow. Examples of such techniques are precision scaling [49], loop perforation [50], data reconstruction [58] and memoization [102]. Approximation techniques that explore numeric precision act on programming language types, analyzing and evaluating precision configurations [15, 49]. Loop perforation consists of ignoring and skipping a subset of the computational work inside a loop iteration [50, 67]. Less work means less effort, higher performance, and power savings. Data reconstruction consists of reducing a set of data to a sample to infer the missing data when it is necessary [58, 74]. Memoization samples patterns of computations to identify and return possible similar outputs previously computed and stored [102]. Further exclusive software and hardware techniques, there are co-approximations between hardware and software that enable control by software to hardware components used only on the approximation, like on quantized lookup tables controlled by software procedures [52, 91].

2.1.3 Effect of Errors in the Results

An application executing in an approximate environment exposes some computational elements to errors. The consequence of such errors may be on the execution behavior or in the data. In the execution behavior, the error is more easily detectable because of deviations of the execution flow that may cause execution crashes [118]. Errors manifested in the data may lead to Silent Data Corruption (SDC) [43, 53], which may degrade the produced output without deviating the execution flow. While execution crashes induce the loss of the partial execution not producing results, SDCs may compromise the entire output or trespass a limit of acceptable depreciation in the results.

The perturbation outcome problem consists of guaranteeing a statistical limit of error impact in the output quality with improvements in energy or performance automatically for any general-purpose application and perturbation model [115]. The impact of nondeterministic errors is not easily predicted when different data is exposed to these errors. Figure 2.1 shows examples of images generated by a lossy compression algorithm [131] with the same input (the baseline image) and exposed to the same error probability (10^{-6}) on the same approximate DRAM main memory but with evident differences in quality. Figure 2.1(b) has no clear deformations with the baseline, while Figure 2.1(c) has visible distortions on the pixels in the bottom, evidencing a possible loss in the control loop for the reference of the pixels, and Figure 2.1(d) contains perceptive noise where more than half of the pixel references are missed by the application. Therefore, the error effect in the results depends on other variables than its probability of occurrence and may impact distinct aspects of the computation.

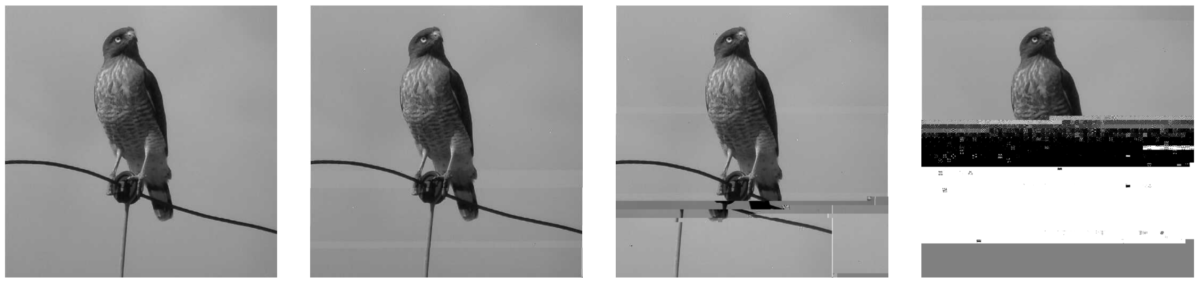

(a) baseline (b) (c) (d)

Figure 2.1: Resulting of the execution of a lossy compression algorithm with the same baseline image exposed to the same error probability. *Even with the same probability, the error may have perceptible different impacts on the results.*

2.1.4 Quality Metrics

Approximate systems produce results with some deviation from the accurate value. The appropriate limit of acceptable depreciation depends on the application and the type of manipulated data. This depreciation is quantifiable by quality metrics that are more general and conventional to several data types [79]. A defined quality threshold has the minimum requirements of an execution output for a useful result of an application. Quality metrics quantify how different is an approximate output compared to the accurate output. Thus, a quality metric results from a comparison between the accurate and approximate outputs of the same application and input. To standardize the values of quality metrics and to improve readability and understanding of the result analysis, we adopt in this work normalized metrics that return values between 0 (no quality on output) and 1 (output identical to the accurate). Thus, the quality metrics that have been applied in the literature and are adopted in this work include the following:

- **Mean Absolute Percentage Error (MAPE)**: The relative error of the approximate points in the results, defined as

$$MAPE(A, F) = \frac{1}{n}\sum_{t=1}^{n}|\frac{A_t - F_t}{A_t}| \tag{2.1}$$

  where

  - n: the number of elements in the data array;
  - A_i: is the i-th element in the accurate array;
  - F_i: the i-th element in the approximate array.

- **Fraction of Equal Elements (FEE)**: The fraction of elements (single datum, lines, or words) in both output data, defined as

$$FEE(A, F) = \frac{1}{n}\sum_{t=1}^{n}\{^{1,\ \textbf{if}\ A_t=F_t}_{0,\ \textbf{otherwise}} \tag{2.2}$$

  where

  - n: the number of elements in the data array;
  - A_i: is the i-th element in the accurate array;
  - F_i: the i-th element in the approximate array.

- **Fraction of Elements within a Margin (FEM)**: The fraction of elements out of an acceptable margin of error, defined as

$$FEM(A, F) = \frac{1}{n}\sum_{t=1}^{n}\{^{1,\ \textbf{if}\ |A_t-F_t|\leq(A_t\times M)}_{0,\ \textbf{otherwise}} \tag{2.3}$$

  where

- n: the number of elements in the data array;
- A_i: is the i-th element in the accurate array;
- F_i: the i-th element in the approximate array;
- $0 < M \leq 1$: the relative acceptable margin in each element.

- **Structural Similarity Index (SSI)** [6]: The difference between an accurate (A) and an approximate image (F), defined as

$$SSI(A,F) = \frac{(2\mu_A\mu_F + c_1)(2\delta_{AF} + c_2)}{(\mu_A^2 + \mu_F^2 + c_1)(\delta_A^2 + \delta_F^2 + c_2)} \tag{2.4}$$

where

  - μ_x: the average of the array x;
  - δ_x^2: the variance of the array x;
  - δ_{xy}: the covariance of x and y;
  - $c_1 = (k_1 L)^2, c_2 = (k_2 L)^2$: two variables to stabilize the division with weak denominator;
  - L: the dynamic range of the pixel values (typically $2^{\#bitsperpixel} - 1$);
  - $k_1 = 0.01$ and $k_2 = 0.03$.

All of these metrics return a percentage indicating how similar the application output is compared to a non-approximate execution, where 100% means they are identical. While different metrics are not numerically comparable, the normalized range enables us to define a unique threshold for all applications for a better understanding of the results. Furthermore, different quality metrics can be used to evaluate the same kind of data, such as SSI and Peak Signal-to-Noise Ratio (PSNR) measuring the difference between images. Some metrics, however, are restricted to specific data types, as SSI to images.

Figure 2.2 exemplifies how the occurrence of memory errors affects the output of the application jpeg from AxBench [131] and how the quality metric SSI captures the quality depreciation. The first image shows an error-free output, in which the computation result is identical to the accurately computed baseline. Images 2.2(b)-2.2(d) show two types of consequences of errors, depending on the code region they affect: incorrectly storing parts

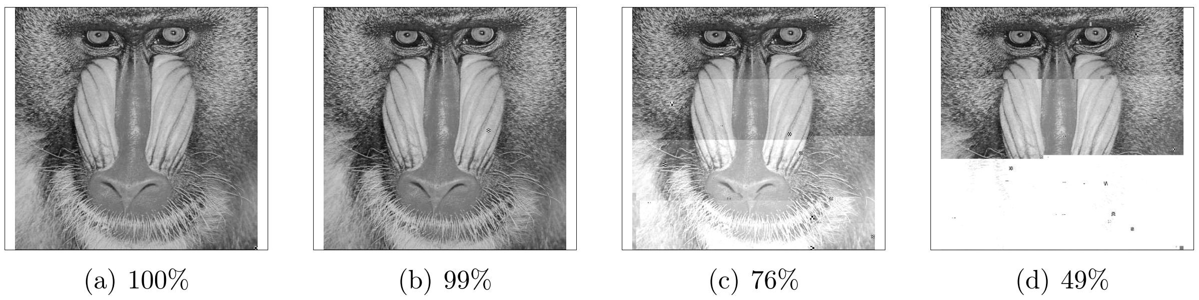

(a) 100% (b) 99% (c) 76% (d) 49%

Figure 2.2: Examples of SSI quality measure of images. *Quality metrics show how far an approximate image is from the accurate one.*

of the Huffman code into memory cause errors isolated to single 8x8 pixels blocks, causing a lower impact on similarity; and incorrectly retrieving the Huffman coefficients from the last computed block, where the coefficients are cascaded, causes the image to exhibit stripes of different brightness, leading to a larger impact on similarity. Since errors are injected randomly, they can happen at any point in the execution, and the final impact on quality is not directly a function of the error rate. For example, both images 2.2(c) and 2.2(d) were subjected to the same error rate of 1.27×10^{-5}, but in the second case a high brightness causes the image to appear white at the bottom, thus the difference from the original image is more evident and the quality metric is lower.

2.2 Techniques for Memory Approximation

Memory cells are intrinsically sensitive to circuit variability due to the size of components and dense layouts [42]. External factors, such as process variation and temperature, may affect the data and cause errors, thus vendors define the nominal values of memory parameters according to the worst-case operating conditions to guard-band a secure margin to ensure reliability [13, 94]. These parameters add overheads to the performance and energy consumption of the memory systems [43]. Thus, a typical operation of the memory components allows for lower overheads at an error-free margin [94]. Approximate memories adjust parameters out of this margin to achieve benefits on energy or performance but at the cost of occasional errors in stored data [37]. In the remainder of this section, we discuss memory technologies and techniques that explore changes in working parameters to achieve energy or performance benefits.

2.2.1 DRAM

Dynamic Random Access Memory (DRAM) has been used as the main memory system due to offering relatively high-density storage and low cost [113, 114]. A DRAM device is accessed by the CPU through a memory controller and is organized hierarchically into modules, ranks, chips, banks, and cells [37]. A bank is an array of DRAM cells organized into rows and columns accessed through bitlines shared by each column of cells and sense amplifiers that convert the received charge to digital binary data [12]. A DRAM cell consists of a transistor that acts as an access switch to the bitline and a capacitor that stores binary data in the form of electrical charge. The transistor is activated when the signal of the wordline is activated, releasing the state of the capacitor to the bitline. The storage of a DRAM is based on the charge of capacitors. This charge, however, is slowly decreased because of leakage, and, to prevent the stored data to be lost, a periodic refresh operation needs to be performed at DRAM cells [69]. A refresh operation causes the row buffer to read the cell charge and restore it to the full value, being an identical operation as opening a row, from a circuit perspective [57]. The period between each necessary refresh depends on the retention time of each cell, defined by the leakage behavior of the components that varies within cells of the same device [92]. A DRAM device works through commands by the memory controller. Initially, all DRAM banks are in the precharge state, ready for activation. The process of accessing data pass through activation and read or write commands to access the corresponding column in the row buffer [12]. New access on another row occurs after a precharge command to prepare the DRAM bank and restore the previously accessed data. DRAM has timing parameters that define the period between the change of these states or the refresh operations [65, 69].

Approximate DRAM changes parameters to achieve energy savings or performance. These parameters can be retention time [83], latency [132], and supply voltage of the data array [61]. The retention time regards the period between refreshes, a higher period means less energy spent in these operations but may expose the stored data to retention errors during the hold operation of the stored data. Latency parameters account for the time that the controller must wait before another change of state (e.g., access or restoration) and lower latency allows better performance but an exponential increase in the error probability on accessing data. The supply voltage has quadratic relation with the dissipated power but also an exponential relation with the error probability.

2.2.2 SRAM

Static Random Access Memory (SRAM) is a volatile type of memory that is the main choice for caches due to its relatively lower latency but higher cost-per-bit. A typical SRAM cell has 6 transistors to ensure the stability of the stored data, which causes this type of memory to occupy more area and be more complex than a DRAM per stored bit [37]. The organization of SRAM is in blocks that store the data words. The activation of a block is performed through bitlines that can have sense amplifiers attached in low-power environments. The leakage power of the SRAM is a significant fraction of the total power dissipation on a chip due to mechanisms to ensure reliability in the inherent higher exposure to hardware failures in these memories [64].

The adjustment of the supply voltage of the SRAM array is an effective technique for power reduction, however, with a limit that defines the reliability of the memory operation [5]. The static and dynamic noise margins define the minimum supply voltage to ensure that the data is not exposed to errors [119, 121]. Out of these margins, the error probability grows exponentially according to the decreasing of the supply voltage but the energy consumption decreases quadratically [46]. The transistor count and sizes are other parameters that can be adjusted at design time to increase or decrease the robustness of the SRAM cells [2]. Decreasing the sizing or the number of the transistors may increase the occurrence of read and write failures to gain energy savings.

2.2.3 STT-MRAM

Spin-Transfer Torque Magnetoresistive RAM (STT-MRAM) is a potential candidate for caches or scratchpad memories due to low leakage and high-density storage [106, 134]. An STT-MRAM cell has one transistor and one Magnetic Tunneling Junction (MTJ). The MTJ has two independent ferromagnetic layers separated by an oxide barrier layer. One of the ferromagnetic layers is fixed (the reference layer) and the other (the free layer) can have magnetization in parallel or anti-parallel direction to the reference, which defines the resistance state of the logical value of the stored bit. The read operation in the STT-

MRAM inserts a low current through the MTJ to sense the resistance state of the cell. The write operation inserts a large current to switch the magnetization direction of the free layer, which adds performance and energy issues due to the large current and the long switching time [134]. Furthermore, thermal factors may affect the data retention capability and disturb the reading operations of STT-MRAMs, thus, the parameter thermal stability factor is used to define the current and latency of memory operations [85].

Approximation techniques in STT-MRAM include memory operation with lower thermal stability factor, lower current, or shorter pulse [85, 106]. The thermal stability factor defines the operation of the memory and a change in this parameter increases the probability of retention failures and read disturbs but decreases both energy consumption and performance overheads. Shorter pulse duration in read or write operations leads to less access delay and higher performance at the cost of write errors, which affect data during their lifetime (until be overwritten), or read decision failures, causing transient errors. Lower read or write current can reduce the energy spent on these operations but also increase the probability of these dynamic errors.

2.2.4 PCM

Phase-Change Memory (PCM) is a non-volatile solid-state memory that can be used as a storage device or main memory as an alternative to DRAM [104]. This type of memory is composed of chalcogenide, a material that offers several levels of resistance, which enables a single cell to store multiple bits [97]. Thus, the Multi-Level Cells (MLC) storage is essential to reduce the cost-per-bit of the storage on PCMs, however, exposes the data to the drift phenomenon that may lead data to errors [89]. Elevated levels of density on multi-level PCM demand more costly validations to ensure data integrity. These validations are mitigated through set/reset iterations during each write operation. The drift phenomenon also decreases the lifetime of data in the memory, which is the period while the data remains in the cell without being overwritten [84].

Approximation of PCM is performed through relaxed set/reset iterations to improve the write latency but merging intermediate resistance levels and causing persisted errors in the data. Lowering the number of iterations, the occurrence of the drift phenomenon continuously increases, worsens the error rate in the approximate PCM [112]. To mitigate the energy spent on these operations, the voltage pulse of the set/reset iterations may be decreased but also expose the memory cells to a higher probability of the drift phenomenon [2]. Furthermore, the density of PCM can be changed to achieve performance or energy improvements, where increasing the number of levels per cell requires more time and energy per access [104].

2.3 Properties and Models of Memory Approximations

Reliability concerns are the focus of modern memory devices [42]. Volatile and non-volatile memories face problems in aspects of scaling, performance, energy, or aging that may affect the reliability of the stored data [12, 64, 84]. As the technology advances are restricted by these issues, the stored data may be affected by errors that can occur in different manners. In this section, we discuss aspects of memories that provide support to represent the behavior of their errors and components when changing parameters to unreliable ranges to achieve energy savings. First, we classify errors from memories and present the characteristics and effects of them in the stored data. Then, we present tools for simulation and modeling of memory energy and performance.

2.3.1 Types of Errors

Error manifestation can be classified by perdurability, determinism, time, persistency, and distribution. Perdurability accounts for the physical constancy of the failures into the components, which may be hard when errors result from permanent physical failures caused by damages or flaws in the circuit [108], or soft when errors corrupt the data but the device is not permanently damaged [48]. Determinism considers whether the error impacts equally the same data with the same values. Adjusting operational parameters of memories usually exposes data to nondeterministic errors that have a probability of occurrence according to the approximation knob [37]. Examples of deterministic approximations are types of memory compression [93].

Time relates to the moment of the error manifestation and, consequently, to the variable that its occurrence probability depends on. The manifestation can occur at hold, read, and write operations on the memory data array. Failures in hold operation (or static error) depend on how much time the data need to be stored before a reading inserts them into the computation [60]. These errors may occur, for example, when the retention time of a DRAM is not enough to ensure the reliability of the stored data. Failures in read or write operations (or dynamic error) depend on the access operation into the data array. The characterization of dynamic errors usually is performed through an occurrence probability per access. An example of errors in read and write operations is on changing the set/reset iterations in PCM.

Error persistency considers whether the incorrect value is persisted into the data array or is temporary, only present in the computation without affecting the memory storage. Persistent errors affect other accesses in the same data until the data is overwritten [12]. Non-persistent errors are multiple Single Event Upset (SEU) that are not propagated in the data array. Errors from hold and write operations usually have a direct impact on the stored data, while errors from read operations may be transient (e.g., a voltage glitch in the circuit) or persisted (e.g., an error in the row buffer that affects the data array on a precharge).

The error manifestation is specific by approximation and is obtained through the characterization of the memory devices. A characterization performs several iterations exposing real or simulated hardware conditions of possible error fluctuations by adjusting parameters and verifying how the reliability of the stored data is affected [12, 13, 65, 106, 114, 120]. Table 2.1 exhibits some of the major characteristics of techniques for memory approximation, where techniques that adjust parameters are nondeterministic and depend on the value of the changed parameter. However, the error rate may fluctuate due to process variation and other environment variables, such as temperature. The process variation may affect some cells or memory regions that experience more errors in the

Table 2.1: Memory approximation and their corresponding principal characteristic of errors. *Memory approximation by adjusting parameters allows for energy and/or performance improvements. Despite compression being a generic technique, its errors usually are deterministic.*

memory	parameter	time	persisted	energy[1]	perf.[2]	dyn.[3]
DRAM	voltage	access	✓	✓		✓
	access delay		✓		✓	✓
	retention time	hold	✓	✓		✓
SRAM	transistor count or size	both	✓	✓		
	voltage	access		✓		✓
PCM	density	both	✓	✓	✓	
	set/reset iterations	access	✓		✓	✓
	voltage pulse		✓	✓		✓
STT-MRAM	read pulse				✓	✓
	write pulse		✓		✓	✓
	read current			✓		✓
	write current		✓	✓		✓
	thermal stability factor	hold	✓	✓	✓	✓
generic	compression	access	✓	✓	✓	✓

[1] energy savings
[2] performance improvement
[3] allows for dynamic adjustment

stored data than others, called weak cells [13]. The temperature may change according to the power dissipation of the memory components and may affect the stored data depending on the approximation technique [65].

2.3.2 Tools for Memory Simulation

Changes in memory architecture are easily performed in platforms for design exploration. These tools allow simulation according to changes in hardware components or working parameters of the device. Moreover, these tools support a wide range of standard and emerging memory subsystems with extensive models.

Ramulator [59] is a cycle-accurate DRAM simulator that provides models for a variety of standards. This simulator is designed for enabling users to extend models with additional standards and implementations. The input of Ramulator can be memory traces or instructions, and the simulator offers options to generate command traces to feed a controller simulator or other tool for power estimation.

DRAMPower [11] is a tool for energy estimation for several DRAM models based on JEDEC standards. DRAMPower performs energy analysis based on traces of DRAM commands or transactions. The model of power estimations is based on the effects of process variations obtained by MonteCarlo simulations on DRAM cross-sections [10]. The input for the simulation is a trace of memory commands compatible with the file generated by Ramulator.

VAMPIRE [40] is a tool for energy estimation of DRAM power and energy consumption based on experimental characterization that accounts for variations among characteristics of data and hardware structure. The authors state that the energy consumption of real DRAM modules varies significantly from the specification of their vendors. Moreover, the power dissipation depends on the data value and there is significant structural variation across banks and rows of multiple DRAM modules from the same model.

CACTI [82] is a set of tools for the simulation of memory design. The user may change the parameters and conditions of running the simulations among memory hierarchy to obtain the results in access time, power, cycle time, and area. CACTI offers an analytical memory simulation supporting models for SRAM caches and DRAM main memories.

In this work, we perform simulations with tools that change the parameters of memory devices. In the early experiments, we use an energy model based on a characterization from the literature [119] and account for all memory accesses on an SRAM with lower supply voltage with a direct calculation of energy savings. In the remaining evaluations, we consider an energy model from a characterization of the DRAM main memory also with underdesigned supply voltage on the data array [13]. We generate memory access traces of the memory hierarchy to feed Ramulator. We choose Ramulator due to its capacity of converting them to cycle-accurate memory commands. These commands are input for the DRAMPower that accounts for energy estimation of the approximate memory.

2.4 Related Work

The impact of the error on the application results depends on how the approximate data are manipulated during the computation. The benefits of the approximation depend on the inaccuracy tolerance of the application but also on the impact that the error may cause. If errors affect application data that is more sensitive, this impact may invalidate the computational results. Furthermore, if the errors affect the output more than the tolerable limit of the application, computational efforts are wasted, decreasing energy benefits. Interfaces for approximate data explore techniques for controlling the error impact on the execution results through the protection of critical data, recovery at runtime, or configuration of the approximation knobs. We present interfaces for approximate memories in two major proposals: `AxRAM` and `SmartApprox`. `AxRAM` contains the major features for the protection and runtime mechanisms, while `SmartApprox` is a configuration interface that controls the approximation knobs of an approximate memory. In the remainder of this section, we list and compare interfaces related to our proposal in these three categories.

2.4.1 Data Protection Interfaces

Interfaces that control approximations at the data level usually separate application data into error-resilient and accurate. This separation is necessary to avoid errors in application data that are critical and do not tolerate errors. Errors on these data may nullify execution results, decreasing the average quality of outputs and increasing the energy consumed by recovery mechanisms.

Relax [20] is an architectural framework that offers runtime control to software recovery of hardware faults. In this work, an ISA extension allows the compiler to guarantee the state of the program through retrying or discarding computations. Relax also provides software support that allows programmers to annotate code blocks that may experience hardware faults and specify recovery computations in the case of a fault.

EnerJ [103] features type qualifiers that split data into approximate and precise levels of data reliability through annotations on variables at high-level programming language. This allows the programmer to protect critical application data from errors, marking them as precise. EnerJ also provides a system that guarantees the isolation of the precise from the approximate components.

Energy Types [17] allows more levels of approximate operation through energy specifications that describe phases and modes of type information that are inferred by the compiler. The phases indicate the behavior of the program fragment regarding the energy consumption with different logical goals depending on the workload. A mode is a programmer-defined and typed energy state, indicating the expected energy usage context to be used with the specified data and their operations.

DECAF [8] allows error tolerance degrees in the type system on high-level programming language and extends quality constraints to non-annotated data. This work proposes a compilation system that infers data affected by operations with the annotated data. DECAF, therefore, allows the programmer to annotate only the most crucial data and to omit annotations where the accuracy requirements can be implied. This system reaches this implication through static and dynamic analysis of the application.

Truffle [23] is a micro-architecture implementing dual-voltage operation that supports ISA extensions for data protection. A high voltage guarantees reliable operations with precise data, while a low voltage allows energy savings for approximate operations. Thus, approximations are possible in operations, registers, and memory.

Stazi *et al.* [111] propose a characterization of the heap memory region for targeting various levels of approximation. This strategy was applied to a video encoding algorithm and the application was modified to allocate only resilient data into the approximate region. The authors also propose isolation of the MSB from errors to increase the approximation levels and obtain higher energy savings.

ApproxSymate [21] identifies program approximation paths using symbolic execution and dynamic analysis to decrease approximation-domain metrics. However, the instrumentation for the symbolic execution requires changes in the application to identify each possible execution path to approximate. The symbolic execution fed a sensitivity analysis to calibrate the error insertion according to the symbolic expressions.

Aloe [53] aims for static analysis of the reliability of programs with recovery blocks, extending a programming language interface with checkers for detection and recovery from selective SDC. The authors propose methods to support unreliable error detection and approximate re-executions for code blocks exposed to errors.

Protection interfaces usually rely on annotations or changes in the source code of the application to indicate critical data. While effective to protect against crashes, these interfaces bring additional complexity by requiring approximation domain by the programmer and changing the application, which hinders portability and maintainability of the code. `AxRAM` protects regions that store critical data in common for several types of applications and acts transparently without annotations or changes in the source code. Our interface aims for execution resilience, protecting and treating data to avoid execution crashes.

We propose an addressing scheme that avoids access violations on the memory, which represent a major part of the execution crashes on the use of approximate memories. We also propose hardware support through an architectural model that divides memory into reliable and approximate regions controlled by memory-mapped registers.

2.4.2 Runtime Interfaces

Runtime systems control and recover from errors by managing the execution flow to recover from catastrophic errors, which may cause the loss of significant data. This control aims to avoid premature interruptions of the program that cause an execution crash and prevent the application to generate a result. Error management at runtime aims to avoid unexpected execution behaviors by dynamically capturing the execution flow.

Green [7] is a framework that allows programmers to approximate functions and specify statistical quality requirements for applications. An offline calibration phase builds a quality model to be reconfigured at runtime by a quality monitor. The quality monitor proposed by Green checks the quality after a specified number of executions, comparing the approximate and accurate outputs.

PowerDial [51] controls knobs at the software level to mitigate fluctuations of the quality control. This system adapts static parameters of the applications into control variables to use as approximation knobs that are dynamically changed according to the quality constraints. This proposal aims for a particular type of application, deployed to produce results at a target frequency.

Ringenburg *et al.* [99] propose a dynamic quality monitor with offline debugging instrumentation that tracks the data flow of approximate operations. The quality monitor uses correlations identified by the offline tool between individual operations and output quality. The online mechanisms allow adjusting the approximation levels to satisfy quality constraints and re-executing code to recover data.

Rumba [56] aims for online control of large errors in an environment with approximate accelerators through error predictors. The authors explore predictors based on a decision tree, moving average, and linear models. To recover from detected errors, Rumba has a mechanism of selective re-execution of code regions and dynamic parameters tuning to adjust the system to the required output quality.

STAxCache [96] combines circuit and architectural techniques to control the impact of errors from a cache memory implementation controlled by user instructions. This ISA extension allows the user to specify quality requirements on data arrays of the cache memory. STAxCache proposes an architecture organization for an STT-MRAM cache that introduces a quality table that contains quality constraints for each memory address range. Different approximation techniques explore the error tolerance according to read, write, and refresh operations of the STT-MRAM cache to maximize energy benefits.

Crash Skipping [118] avoids overheads of recovering computations by skipping instructions that lead to crashes. If the current instruction causes the interruption of the execution, it is replaced by a `nop`, and the control flow continues. The continuation of the execution occurs based on the granularity of the skip, which can be by instruction or function. In the instruction granularity, the program continues from the faulting instruction to the next sequential instruction. In the function granularity, the program jumps to the return address of the function currently executing, exiting the function. Furthermore, this proposal has a counter of avoided crashes and prevents application stalling by imposing a limit of avoided crashes. Despite improving execution resilience, Crash Skipping does not protect critical data of the application and neither proposes dynamic changes on the approximation knob.

Interfaces that implement runtime systems can monitor and recover from computations that cause execution crashes or lower than required quality. This type of system usually adds an execution overhead that decreases approximation benefits on energy or performance with checkpoint and rollback mechanisms. The addressing scheme of `AxRAM` is a runtime system that does not need checkpoints, recalibration, or instrumentation. This makes the interface a lightweight system that does not add significant overheads in the execution environment. `AxRAM` avoids redirecting the control flow as a feature since operations that cause these crashes would deviate the control flow to an unrecoverable memory region. Furthermore, we explore alternatives for a supervisor system to treat incorrect executions for recovering application data without changes in the application.

2.4.3 Configuration Interfaces

The control of approximation levels refers to the relation between the approximation knob and the error rate. In hardware, the approximation knob can be some parameter that depends on the architecture and applied technique. An approximation level is related to a value of the knob and represents the amount of error that is inserted into application data during the execution.

EDEN [61] is a framework for improving energy efficiency and performance for DNN inference using approximate DRAM. The authors use a retraining mechanism to improve the accuracy of a DNN when executed on approximate DRAM and explore parameters of access delay, retention time, and voltage as approximation knob to meet a user-specified accuracy target based on characterizations of the DRAM error properties. EDEN allows the characterization of unknown DRAM devices through an offloading that emulates the errors injected by the target hardware into the DNN through different error models that are representative of most of the error patterns observed in approximate DRAMs.

Masadeh *et al.* [78] propose an approach based on Machine Learning (ML) techniques to change the approximate design according to different input data. The authors use training inputs to build a cluster of input characteristics and predict the error tolerance of an application to meet a defined quality. Their proposal is targeted at deterministic approximation techniques, where the input precisely determines the impact of errors on the application.

AdAM [112] proposes to determine approximation levels among the memory hierarchy on the use of STT-MRAM and PCM. The authors explore approximations to change the parameters of these memories through an integer linear programming optimization that considers the current workload of the application. At runtime, AdAM adjusts the approximation according to the response time of applications that are directly affected by the errors from these memories.

Ranjan *et al.* [95] allow programmers to identify data amenable to approximation and specify a quality target and propose a runtime controller to configure the error constraints. The runtime controller modulates error by interleaving exploration, in which it dynamically learns values for the error constraints, and evaluation phases, where the system operates with the learned configuration.

DART [130] is a framework for determining approximation levels among the memory hierarchy considering different approximation techniques on SRAM caches and DRAM main memory. This framework has an annotated application as input with specified maximum error magnitude for each approximate variable and extracts memory footprints to analyze data significance and infer approximation levels through a search tree.

OPTIMA [129] aims an online control for approximation levels of multiple cache memories in many-core systems. The authors propose to control output quality by software routines provided by the programmer. An online controller samples the application outputs to manage a tuning procedure that adjusts the approximation knobs. These routines sample the output of the application from time to time, analyze its quality, and adjust the approximation knobs accordingly.

SEAMS [77] is a runtime interface for configuring approximation knobs among memory hierarchy dynamically and based on the workload-specific error tolerance. The authors propose to tune memory knobs without prior observation of the hardware or application workload, which makes the interface technology-agnostic and application-independent. SEAMS require domain-specific metrics to indicate the quality degradation and annotations in non-critical data elements of the application. A quality monitor computes the degradation at runtime and reports to the system for dynamic configuration.

Interfaces for the configuration of memory approximation usually demand several executions in the approximate environment or are guided by domain-specific components, such as input interpretation and quality metrics, requiring annotations or changes in the applications. In `SmartApprox`, no specifications from the programmer are necessary and the quality specification comes from a training phase. Furthermore, our interface considers variables that affect the error rate at runtime and adapts the configuration according to the current error scenario. None of the related proposals consider the error from approximate hardware as dynamic nor configure the approximation technique without changes in the application that require domain knowledge by the user. Our runtime system has no interleaving with the training phase, once a configuration is learned, the detected error scenario determines what is the approximation level of the application, reducing overheads. `SmartApprox` predicts the approximation level that meets the error tolerance based on the execution of training applications, thus adding a training phase without annotations in the source code. This training phase can be performed at design time, alleviating the runtime overhead. Moreover, our proposal considers the error as a dynamic element and comprises different error scenarios according to hardware characterization and sensors at runtime. Thus, once a new error scenario is perceived, it determines a new approximation level without performing a new search, removing the overhead of checkpointing and rollback mechanisms or several executions of the input application among adjusted knobs.

2.4.4 Summary

In this work, we propose interfaces for the protection and configuration of approximate memories that expose data to nondeterministic errors. Table 2.2 shows the list of the features of the related works in comparison to our proposal. The control of other interfaces usually relies on annotations in data or code blocks, ISA extensions, and domain-specific quality metrics that represent the context of the application. Annotations in data or code blocks bring additional code that should be maintained during the application lifetime [21], while ISA extensions may demand changes that increase the system complexity [73]. Furthermore, requiring domain or approximation control by the programmer jeopardizes the applicability of the solution on requiring expert knowledge about application and approximation technique. The transparent control of our interfaces allows for no changes in the application level to execute in our environment, where the configuration, runtime, and protection mechanisms act without annotations.

The configuration of the memory parameters is required to be dynamic to maximize energy efficiency. Static configurations assume that the approximation level for the application does not change in its execution phase. However, interfaces with dynamic configuration [78, 95, 112, 130, 129] usually adjust the approximation knob according to the workload or the different quality thresholds required per input. `SmartApprox` adjusts the knob according to the error changes at runtime, which may be affected by environment variables, such as temperature. Thus, our interface chooses the configurations amongst several approximation levels for each application.

The implementation of an interface for hardware approximation requires modifications at the architecture level. The hardware support considers whether the interface proposes an architectural model to implement its features. Some interfaces restrict their approach at the software level and do not propose changes in hardware components.

Some interfaces assume specific contexts, such as a specific video encoding algorithm [111] or DNNs [61]. Our interfaces aim the utilization for general purpose applications that tolerate some inaccuracy in their results, in an environment with repeatedly executions with different inputs but without specific restrictions for the application.

Our proposed interfaces improve execution resilience by recovering data that would cause crashes. Despite providing mechanisms of protection for critical application data, some interfaces do not allow for recovering lost data or executions. The addressing scheme of `AxRAM` tries to restore an invalid memory address to its original value into the allowed boundaries. Furthermore, we propose mechanisms for attenuating the overhead of the recovery through approximate re-executions, which decreases the average quality but improves energy savings.

Table 2.2: Features of the listed related interfaces. *We propose transparent interfaces that require no application changes and dynamically configure approximate memories for general applications.*

	Work	Year	Control	Config.	HW support	App. changes	Approx. Levels	Recovery	App. purpose	Approximation
protection	Relax [20]	2010	block annotation	static	✓	high	✓	✓	general	generic configurable
	EnerJ [103]	2011	data annotation	–		high			general	generic
	Energy Types [17]	2012	data annotation	static	✓	high	✓		general	generic configurable
	Truffle [23]	2012	ISA extensions	–	✓	high			general	Voltage-scaling
	DECAF [8]	2015	data annotation	dynamic		medium	✓		general	generic configurable
	Stazi *et al.* [111]	2018	data allocation region	static	✓	high	✓		Video encoding	DRAM refresh rate
	ApproxSymate [21]	2019	symbolic execution	dynamic		low		✓	general	generic
	Aloe [53]	2020	block annotation	static		high	✓	✓	general	generic configurable
runtime	Green [7]	2010	function annotation	dynamic		high	✓		general	generic configurable
	PowerDial [51]	2012	software knobs	dynamic		high	✓	✓	general	software knobs
	Ringenburg *et al.* [99]	2015	instrumentation, offline debugging	dynamic		high	✓	✓	general	operations
	Rumba [56]	2016	block annotation, parameter tuning	dynamic	✓	high		✓	general	accelerators
	STAxCache [96]	2017	ISA extensions	dynamic	✓	high	✓		general	STT-MRAM parameters
	Crash Skipping [118]	2019	transparent	static	✓	none		✓	general	generic
configuration	AdAM [112]	2018	workload and data characterization	dynamic	✓	medium	✓	✓	general	STT-MRAM, PCM
	EDEN [61]	2019	DNN characterization	static	✓	high	✓		DNN	DRAM parameters
	Masadeh *et al.* [78]	2019	input clustering	dynamic		low	✓		general	deterministic
	DART [130]	2020	memory footprints, data annotation	dynamic	✓	high	✓		general	SRAM voltage, DRAM refresh rate
	Ranjan *et al.* [95]	2020	data annotation	dynamic	✓	medium	✓	✓	general	compression
	OPTIMA [129]	2021	sample quality routines	dynamic	✓	medium	✓	✓	general	caches for many-core systems
	SEAMS [77]	2021	block annotation, parameter tuning	dynamic	✓	low	✓		general	nondeterministic approx. memories
	This work	2022	transparent	dynamic	✓	none	✓	✓	general	nondeterministic approx. memories

Chapter 3

Interfaces for Approximate Data Access

Practical use of data approximation requires interfaces for data protection that act between the application and the approximation technique, controlling what data can or cannot be exposed to errors, or triggering recovery strategies when critical errors occur [53, 104]. Transparent interfacesprotect application data without requiring programmer annotations [118], acting mainly on execution resilience to improve the chances for a valid result.

This chapter presents proposals of transparent interfaces for approximate memories aiming for improving execution resilience. First, we present an interface that protects common critical data regions, treats data that cause execution crashes, and implements an architectural model that allows for error isolation of memory regions. Secondly, we explore alternatives for transparent resilience to detect and recover from lost executions and implement our interface in an environment that reduces the effects of errors by pushing the approximations to other levels of the memory hierarchy. Lastly, we propose an approximation scheme that also is an execution environment that works transparently or with easily portable support to approximation control by the application.

3.1 Summary of Protection Interfaces

In this chapter, we present three proposals of transparent interfaces for the protection of critical data. Despite protecting data considering different aspects regarding to the memory hierarchy and environment, the three interfaces are implemented with the same architectural model. Thus, the three setups consider memory approximation controlled by a knob that exposes the stored data to nondeterministic errors and allows for the protection of some critical data.

3.1.1 Embedded SRAM Main Memory

Section 3.2 proposes an interface considering an embedded systems environment where an application has access to the physical addresses of the data array, and the memory is a single entity that is solely responsible for storing all data, without cache levels. The environment of Section 3.2 comprises a simple scenario applicable to embedded processors with limited Static Random Access Memory (SRAM) memory that are often used in sensing, control, and Internet of Things (IoT) applications. The evaluation is performed with a voltage-scaled SRAM main memory on a MIPS32 architecture executing in the ArchC simulator [98]. The relation between the error rate and the supply voltage is based on the characterization from Wang and Calhoun [119].

3.1.2 Approximation in Memory Hierarchy

Section 3.3 proposes an interface that explores features from a supervisor system that controls the memory space and offers access to virtual addresses to the applications. Thus, the environment considered for this interface needs an Operating System (OS) to convert virtual addresses to the applications. Furthermore, the OS controls the memory regions by supervisor or application privileges. Only application accesses are exposed to errors, which is defined and supported by status registers provided by the architecture. A runtime system manages the execution and triggers re-executions when necessary. A lost output may therefore be recovered by re-execution mechanisms.

The scenario in Section 3.3 includes a memory hierarchy with multiple cache levels that attenuate the access into a main Dynamic Random Access Memory (DRAM). Furthermore, caches are not approximated, and therefore the memory hierarchy alleviates the impact of errors from the approximate DRAM. Experiments are based on the RISC-V 64 architecture executing in the Spike simulator [66]. The relation between DRAM voltage and the error rate is based on the characterization from Chang *et al.* [12], and the energy savings calculated through simulations in Ramulator [59] and DRAMPower [11].

3.1.3 Protection based on Program Sections

Section 3.4 executes in a similar environment to Section 3.3, except for the inclusion of status registers to filter supervisor instructions as error-free accesses. The interface proposed on Section 3.4 considers well-defined data regions of the program and the supervisor system, where only data in the program region is exposed to errors. Thus, if the supervisor executes a system call manipulating data in the program region, for example, this access will be exposed to errors.

The experimental setup and the energy parameters are the same as Section 3.3. Additional to the transparent mechanisms, Section 3.4 offers optional annotations to the application. Although annotated applications are non-transparent, the interface proposes these annotations as non-mandatory, maintaining compatibility with applications designed for traditional environments.

3.1.4 Comparison between Interfaces

The results from each evaluation are not directly comparable due to the differences among environments. However, a high-level comparison is possible through an analysis of the implementation and methodology of each one. Table 3.1 exhibits a comparison of the evaluated features from each environment. Despite Section 3.2 applying a simpler environment than others, without virtual addressing and cache levels, it proposes general hardware support on an architectural model for the approximate memory. The simpler

Table 3.1: Features of the setup for the protection interfaces from each section of this chapter. *A simpler environment allows for higher energy savings, while an approximation source more distant from the CPU allows for a lower occurrence of crashes.*

	main control	hw handl.[1]	cache levels	allow anno.[3]	virtual addr.	crash occur.[4]	rel. sav.[5]	evaluation env.
3.2	Memory-mapped registers	✓				high	higher	SRAM / ArchC / MIPS32
3.3	Re-execution trigger		✓		✓	low	high	DRAM / Spike / RV64
3.4	Partition of sections on loading ELF		✓	✓	✓	lower	low	DRAM / Spike / RV64

[1] Implements handling of errors in hardware
[2] Implements acceptance tests of the results
[3] Allows annotations (in addition to the transparent features)
[4] Occurrence of crashes amongst error rates compared to the other interfaces (lower is better)
[5] Relative energy savings in its respective setup compared with the other interfaces (higher is better)

environment from 3.2 allows for higher relative energy savings, however, the energy consumption in the other setups is higher due to the use of cache levels and DRAM main memory, which indicates that lower energy savings in relative measures may mean higher in the absolute value.

The higher occurrence of crashes on the setup from Section 3.2 is due to the error source being closer to the CPU than on the other environments. Thus, the interface from 3.2 has a higher decrease of crashes but also in the relative occurrence in its environment, which has higher occurrence than others. Despite that, applying this interface to the other environments without adaptations would achieve worst results than other implementations. In Sections 3.3 and 3.4, an addressing scheme is adapted implementing features for the virtual addressing of the RV64 Sv39 page system, where the page table is a radix-tree that should be traversed by the supervisor system to find physical addresses.

Despite having lower relative savings than other interfaces, the execution environment from Section 3.4 allows optional annotations from the programmer, which reduces the occurrence of crashes. Thus, this setup allows applying the proposal to contexts where the recovery from crashes is more costly or not possible.

3.2 AxRAM: A Lightweight Transparent Interface for Approximate Data

In this section, we present `AxRAM`, a high-level interface for approximate data access that improves execution resilience by allowing coarse-grained control of the approximate state of a data region. We found that invalid memory references are the main cause of crashes for many applications running on systems with approximate memories. To protect memory references without annotations in these data, we identify operations with invalid addresses and protect critical memory regions that store pointers. Thus, `AxRAM` corrects memory access violations by only accessing addresses within memory bounds, avoiding interruptions on the execution flow. To protect other references and critical data, `AxRAM` divides the data array into fixed-size memory regions, where each region can choose between an approximate level and an accurate state. To isolate a memory region transparently, we identify the system stack as an area that stores control pointers and other critical application data. Thus, our implementation isolates this memory location without any user annotation to identify critical data.

We propose an architecture model that allows the implementation of the `AxRAM` interface. This model is based on voltage overscaling approximation on an SRAM that has two global levels of supply voltage. The first, higher, voltage level is the nominal value for the memory cell, which guarantees the execution of memory operations at the minimal designed error rate. An adjustable voltage regulator provides the second global voltage level, at which lower voltages lead to energy savings with higher error rates in the memory operations. Despite our interface being built to work with a voltage-overscaled SRAM, it is suitable to other approximate memories that exhibit related error models, such as DRAMs with voltage, timing, or energy changes.

We evaluated the design by simulating the execution of 12 selected applications from various computing domains. For each application, we defined as error-free the minimum number of regions to store the application stack, keeping locally allocated data safe and avoiding execution crashes on subroutine returns. We detail our evaluation with an analysis of execution crashes, output quality, average energy cost, and quality-energy efficiency metrics. `AxRAM` contributions, built upon previous work [25, 26, 27], include:

- An addressing scheme for data stored in approximate memories that avoids execution crashes;
- Implicit protection of a memory region that stores critical data;
- A memory architecture design that allows the coexistence of accurate and approximate memories of variable sizes in the same system;

Our experimental evaluation compares `AxRAM` with the use of a voltage-overscaled approximate memory, employing no data protection. Our results show that `AxRAM` eliminates data crashes, reducing 51% of total execution crashes. When comparing `AxRAM` with an approximate memory without any data protection, `AxRAM` offers energy savings of 9% at a 95% average quality threshold.

3.2.1 AxRAM design

`AxRAM` is an interface to improve execution resilience, maximizing the benefits provided by approximate memories. Our proposal improves the average output quality to increase energy efficiency by avoiding execution crashes in approximate data environments. This approach considers that, in a production scenario, an application typically runs multiple times with different inputs. Every single execution instance is subject to some errors from the approximate memory, which may lead to quality degradation and, eventually, an execution crash. By avoiding crashes, we allow many of the previously unsuccessful execution instances to last longer and produce some output. Thus, the average quality amongst a whole batch of executions is increased not by better quality for every single instance, but mainly by producing more results. This reduces the amount of energy spent on unsuccessful computations, increases efficiency, and potentially allows the whole application to be subject to higher error levels.

We propose two modifications in the memory design to avoid execution crashes. Our improvement focuses on execution resilience by (1) treating accesses out of allowed memory boundaries and (2) protecting critical data regions commonly found on many applications. The remainder of this section classifies execution crashes and discusses the fundamentals of these two approaches.

Types of Crashes

Execution crashes are premature terminations of execution flow that lead to no output production. Without an output, the quality cannot be computed and is perceived as zero, which reduces the average quality of executions. These terminations usually are caused by errors in critical application data. We classify execution crashes into three types:

- **Data crashes**: when an attempt to fetch data from an invalid memory address causes an access violation.
- **Flow crashes**: when the control flow tries to jump to an incorrect region of memory.
- **Timeouts**: when there is no valid result produced after some reasonable, application-specific, amount of time.

A `load` or `store` operation with an out-of-bounds pointer causes a data crash. An attempt to jump to an invalid control address causes a flow crash. Timeouts happen on applications that rely on data convergence, when errors accumulate and prevent the execution to meet the stop criteria. Besides, the use of data structures based on memory references may cause these crashes: an error may produce a wrong pointer, causing infinite iterations over random or irrelevant memory locations.

Treatment of Incorrect Pointers

Memory boundaries are defined by the size of the memory in embedded systems and by the application limits in virtual memory. A considerable number of crashes are caused by memory operations on addresses that are out of application boundaries. These addresses were data pointers, stored in memory, that had one or more bits flipped due to an error. An attempt to operate with an invalid address makes the system throw an access violation signal. This signal stops the control flow and discards the remaining computation, causing an execution crash. This leads to no output being produced and decreases the average quality of results in the approximate environment.

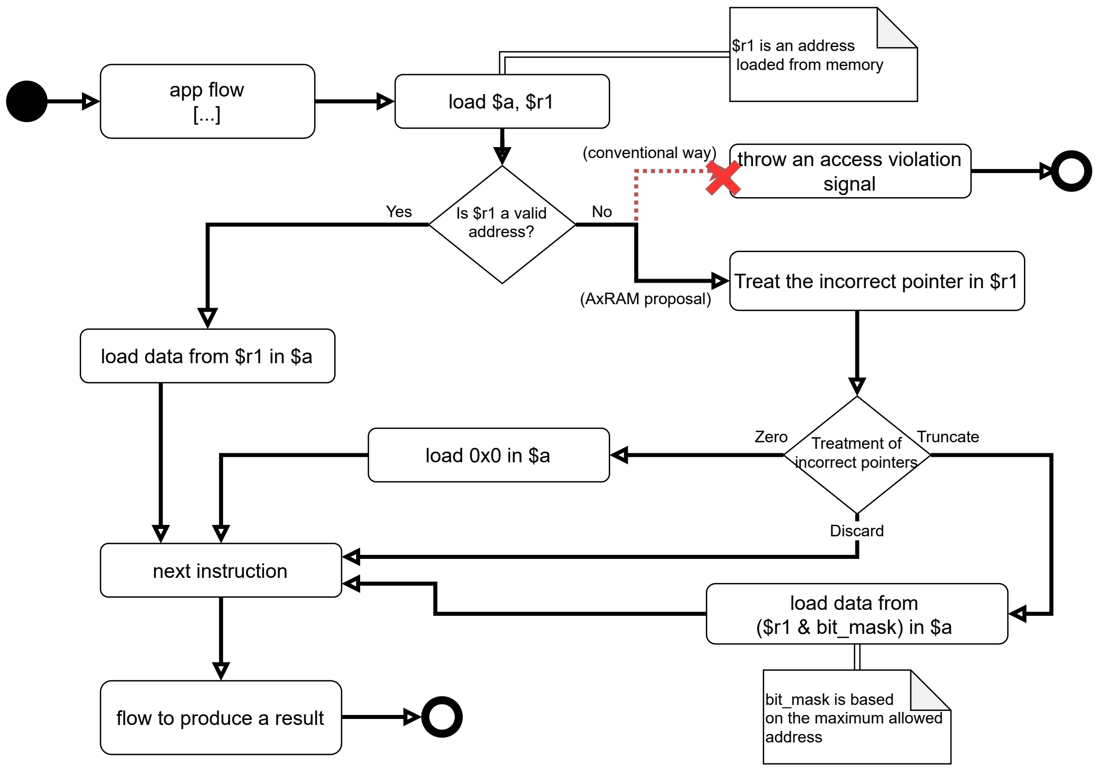
To correct invalid addresses, we need to identify data pointers stored in memory. Nonetheless, pointers are indistinguishable from any other data at the memory level, and identifying them requires additional information from the application level, adding significant overhead. Hence, we propose to detect data pointers through instructions that manipulate pointers. When `load` or `store` instructions are executed, these instructions receive a data pointer as a parameter.

Our proposal is an addressing scheme to treat invalid pointers that are out of allowed memory boundaries. AxRAM identifies data pointers on memory operations and verifies whether these addresses are within memory bounds or not. Instead of throwing an access violation signal, we proceed with the computation after treating the incorrect pointer. We evaluate three forms of treatment for incorrect pointers during the execution: (1) discarding the current instruction, (2) zeroing the destination register, and (3) truncating the address indicated by the pointer within bounds.

Figure 3.1 shows an example of the workflow of the treatment of incorrect pointers with a `load` instruction. Depending on the implementation of this treatment, different values are loaded into the destination register, but, in all cases, the execution flow continues to the next instruction. This is different in the conventional way, where the execution flow stops and throws an access violation signal. However, in the case of a `branch` instruction, the incorrect pointer refers to the next instruction in the execution flow and, therefore, the remaining instructions may be lost through a flow crash.

Discarding the current instruction leaves unchanged the value in the destination register, and the remaining computation proceeds without any change in context. This treatment can be advantageous in the case of a loop that uses the destination register to load temporary values, for example, because the computation proceeds with a value from the previous iteration. If these values are pixels of an image, the value from a previous iteration can be similar to the current one.

Loading zero in the destination register can be advantageous in the case of some data structures like linked lists. These structures depend on pointers that indicate the next



app flow
[...]
load $a, $r1
$r1 is an address loaded from memory
(conventional way)
throw an access violation signal
Is $r1 a valid address?
Yes
No
(AxRAM proposal)
Treat the incorrect pointer in $r1
load data from $r1 in $a
Zero
Treatment of incorrect pointers
Truncate
load 0x0 in $a
Discard
next instruction
load data from ($r1 & bit_mask) in $a
bit_mask is based on the maximum allowed address
flow to produce a result

Figure 3.1: Treatment of incorrect pointers working in a `load` instruction. *Instead of throwing an access violation signal,* AxRAM *treats the incorrect pointer to continue the execution in the flow to produce a result.*

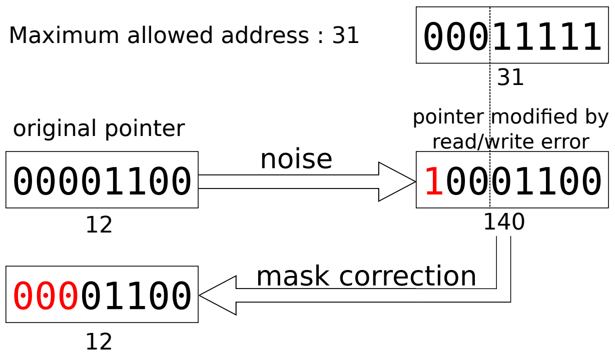


Figure 3.2: Example of how the protection of memory boundaries works. *A bitflip in the MSB is corrected by a mask that considers the memory limits.*

element. A pointer reading zero is conveniently used to indicate the end of the list. When finding an incorrect address while iterating over such a structure, there is no way of finding where the next position is stored, thus zeroing the value would indicate the end of the list and allow computation to proceed towards some, not necessarily correct, output.

Finally, truncating the pointer value is a more generic approach. The truncation applies a mask to the value based on the characteristics of the memory space. All bits that would represent an invalid memory location are zeroed, forcing the address to be valid. This truncation is an attempt to correct the address to the original value, considering the common case represented in Figure 3.2, in which the memory is smaller than the addressing capability of the data word, where invalid addresses would be triggered by an incorrect reading of higher magnitude bits of the address. This treatment can be advantageous in more general contexts since it tries to recover the original value of the pointer. Nevertheless, there is no guarantee that an incorrect pointer will return to its original value after the mask is applied because an error may occur on the less significant bits or more than one error may change the pointer. In the case of an incorrect (but valid) pointer, the computational work proceeds with the wrong data fetched from this address.

The choice for which treatment to perform depends on the implementation of the interface. To simplify the usage of this feature, `AxRAM` loads the configuration at boot time and performs the same treatment for all applications. Thus, the treatment of incorrect pointers requires no user intervention since the execution environment triggers it automatically. This addressing scheme can be implemented on the architecture, OS, or as a runtime system that encapsulates memory accesses. `AxRAM` implements an error recovery mechanism that does not need checkpoints or program instrumentation. Furthermore, the implementation of `AxRAM` as a runtime system represents a lightweight avoidance of and recovery from crashes without programmer intervention or program modifications. The energy cost of this implementation is lower than runtime systems that recover from errors by checkpointing, monitoring the execution, and re-executing entire functions or computational tasks. Moreover, the hardware implementation of this treatment is as simple as an AND gate in the memory input to modify pointer values, which represents a negligible overhead in performance or energy.

Critical Data Protection

Incorrect data pointers cause a significant part of execution crashes. Nevertheless, these pointers are not the only critical application data. Flow pointers or control indexes, e. g. return addresses of functions, file headers, and loop control indexes, are critical values that are not treated by the addressing scheme. Several works [8, 17, 101, 103] propose the error isolation of critical data by extending the programming language to include annotations to classify how critical each data portion is. These annotations require, from the programmer, expert knowledge of the approximated environment, the control mechanism, and a full understanding of the application data, reducing the portability of the solution. Thus, the automatic identification of critical data is essential to improve execution resilience without programmer intervention.

We identify the system stack as a contiguous region, usually small, that contains some critical data in many types of applications. Compilers use the system stack to store temporary execution values, such as shorter-lived automatic variables. Errors on these indexes may cause a loop to execute indefinitely, which leads to a timeout. Furthermore, return addresses of functions are commonly stored in the stack region. These pointers indicate where the execution flow must return after the end of a called function. An error on these data makes the execution flow try to jump to an incorrect address causing a flow crash immediately. Furthermore, the system stack boundaries are easily traceable at the architecture level. These characteristics make the system stack a natural candidate region to be protected from errors without programmer intervention.

3.2.2 Implementation

A memory interface acts between an approximation technique and the application. The approximation technique depends on an environment implementing an architectural model that allows the approximation. In this section, we discuss the implementation of our interface with approximation techniques and other issues that could be faced with the usage of `AxRAM`.

Architectural Model

`AxRAM` offers two main features to protect and recover an application from crashes in an approximate environment. The treatment of incorrect pointers operates directly at the memory addressing scheme, independently from the architectural model. Nevertheless, critical data isolation requires some architecture support to isolate some parts of the memory from errors.

`AxRAM` architecture model for data isolation defines two reliability levels in memory storage. One of these levels should be an operation considered free from errors. Several proposed architectures [23, 70, 75, 116] separate memory regions by reliability levels to isolate some data from errors. `AxRAM` is compatible with the architectural model of Truffle [23], but without the usage of ISA extensions, which would demand changes in some level of the application. Therefore, we propose an architectural model to avoid excessive application changes.

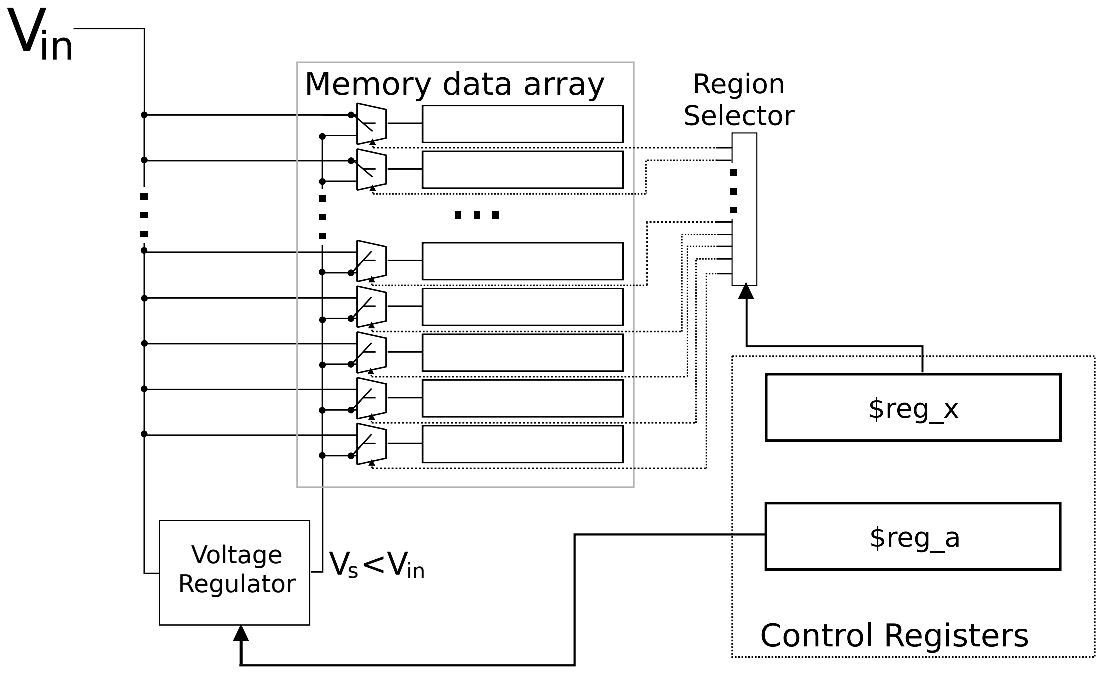


Figure 3.3: Memory architecture of AxRAM. *Memory-mapped registers control where and how much error is allowed on the approximate memory.*

Our architectural model also uses voltage scaling as an approximation source based on an embedded SRAM main memory. Despite that, `AxRAM` is suitable for other memory approximations that exhibit nondeterministic data errors and allow the division of memory into approximate and non-approximate regions. In our architectural model, illustrated in Figure 3.3, the memory is divided into regions on the data array, where each region has a common supply voltage. One single configurable voltage regulator supplies a voltage level below the nominal specification of the memory regions, that is, a voltage level at which the stored data is more susceptible to errors [119]. A switch in the power line of each region specifies whether the region should use the nominal, higher, error-free V_{in} voltage or the lower error-prone voltage supplied from the regulator.

To control the approximation, the model includes two memory-mapped registers. Register $reg_a defines the voltage level supplied by the voltage regulator and register $reg_x controls the gate switching to define which regions are in the approximate state. These memory-mapped registers work as knobs to control approximations. In the case of an embedded systems environment, without the supervision of an OS, these knobs can be configured before the execution of the application. Thus, the main work to port an application to this environment is to find the tolerable error rate for the application. This error rate represents the limit of imprecision tolerated.

OS Support

The architectural model from 3.2.2 supports the execution without application changes in an embedded systems environment with `AxRAM` as a runtime system. If an OS supervises the application, the control knobs should be configured by the OS since other applications could use the same memory. The OS is the runtime system that implements the interface, in this case.

To make the configuration of the approximate memory simpler, our architectural model supports only two reliability levels concurrently – one considered as precise and another that exposes data to a configurable error rate. Thus, all data stored in approximate regions are exposed to the same error rate at a given point in time. This error rate is the same for all applications running in this environment that have data exposed to errors. Therefore, a syscall with a probability as a parameter should be available to configure the memory error rate when necessary.

The reliable regions of the approximate memory contain the program stack, which belongs to an application in an environment with an OS. The application stack is previously allocated by the OS. Thus, to transparently protect this region, the OS has to specify in the $reg_x register the memory area reserved to store the stack. The OS has control over all memory pages allocated to the applications, thus the stack protection works without changes in the application.

3.2.3 Methodology

This work relies on the study of errors' impact at the application level. This study needs to evaluate different techniques and error models. A fast alternative to make this evaluation is the modeling of data errors at higher-level abstractions in a simulation environment. A simulation environment allows the implementation of different approximation techniques on several technologies.

Setup

The experimental evaluation of our proposal is in an embedded systems environment where one single-threaded application runs in bare metal in the CPU without the supervision of an OS. In this environment, the application has access to the entire memory array through an SRAM main memory with the architectural model described in 3.2.2. This implementation is a model with no influence of other applications or a middleware on the effect of errors and energy savings.

In our modeling approach, an approximate state changes the supply voltage of the data memory and exposes data to dynamic errors according to a uniform probability. This model is implemented in an ADeLe-generated [36] CPU model for the ArchC architectural simulator [98] using a MIPS32 architecture. The simulator replaces all read and write operations on the data memory with a software model that is susceptible to error, by performing a single bit flip in a random position of the data word. This bit flip represents an error according to a uniform probability specified in $reg_a.

We compare `AxRAM` with an approximate memory in a scenario of a voltage-overscaled SRAM that implements the same approximate states, architecture, and error model. This environment implements neither `AxRAM` addressing scheme nor critical data protection. The implementation of the same approximate states makes this a fair comparison because it allows the applications the same data error probability on both techniques, with and without our proposed features. We refer to this environment as "`approximate memory`" in our results section.

Table 3.2: Applications, their respective type, and quality metrics of the experiments. *We evaluate applications from several computational domains.*

Application	Type	Quality metric
2mm	Memory-bound	MAPE
nbody spectralnorm	CPU-bound	
reg_detect	Signal processing	
bunzip2 bzip2 dijkstra floyd-warshall qsort	Memory-bound	FEE
fft	Signal processing	FEM
jpeg		SSI
mandelbrot	CPU-bound	

Applications

Table 3.2 shows the applications we use on the evaluation, their type, and respective quality metrics. More details of the quality metrics are presented in Section 2.1.4. These applications represent a wide range of usages of computational systems, such as linked lists, function pointers, floating points, compressing, and arithmetic operations. Furthermore, some applications represent different implementations of a solution to the same problem, like dijkstra and floyd-warshall (the shortest path problem), or manipulate the same kind of data inversely, like bzip2 and bunzip2 (compressing/decompressing). We classify applications into three types: signal processing, CPU-bound, and memory-bound. Signal processing applications are commonly applied to the context of approximate computing and include image processing. CPU-bound applications are kernels that stress the CPU in the majority of its processing with few memory accesses. Memory-bound applications spend major time of the execution with accesses to memory into the kernel.

Voltage-overscaling is a nondeterministic approximation technique [80], and therefore we need to perform several executions to evaluate the impact of errors in the application. Thus, we execute 100 times each application at each error rate. The error rate determines the approximation level of the technique. We evaluate the applications at 40 error rates in logarithmic intervals from 1E-9 to 1E-4. The 1E-9 represents an error rate where most of the evaluated applications do have execution crashes, and, in the 1E-4 error rate, most of the evaluated applications obtain results with quality equals zero. For simulation purposes, the approximation technique was applied only to the code in the main computation stage of each application, to avoid errors happening during I/O phases that emulate some peripheral behavior. The applications are compiled by ellcc with the flags -O3, -static, and -target mips32r2-linux.

Quality Control and Energy

The usage of approximate memory implies errors in some application data. These errors can result in some quality loss in the output of each execution. Since the minimum acceptable quality depends on the context of the application, the tolerable limit of error also depends on this context. Nevertheless, there is no way to measure the output quality without the accurate result, obtained through a non-approximate execution. To calculate the energy savings, we define a threshold in the quality metric that each execution output must obey. Thus, the energy cost is based on several quality thresholds of execution outputs. To avoid application stalling, we set a timeout as twice the accurate execution time for each application.

The energy savings are based on a relative value to the nominal voltage of the SRAM. The data to infer the relative voltage is extracted from Wang and Calhoun [119], where the authors present error rates for a voltage range of SRAM cells calculated through the static noise margin of these cells. The data used in this work is from a 45nm 6T SRAM, where the error rate is independent of the stored data. We implement the errors in memory read and write operations considering the rate of the highest error probability at each voltage.

3.2.4 Evaluation

We present in our experimental evaluation a comparison between the three forms of treatment of incorrect pointers. Further, we consider the treatment by truncation as the implementation of `AxRAM` in our chosen environment and illustrate in a case study the analysis of the impact of addressing and data protection on execution crashes. Lastly, we evaluate and discuss `AxRAM` considering four main metrics: number of execution crashes, output quality, energy savings, and quality-energy efficiency.

Treatment of Incorrect Pointers

The purpose of this treatment is to avoid data crashes caused by incorrect pointers. Figure 3.4 shows execution crashes of all applications with the evaluated error rates to the

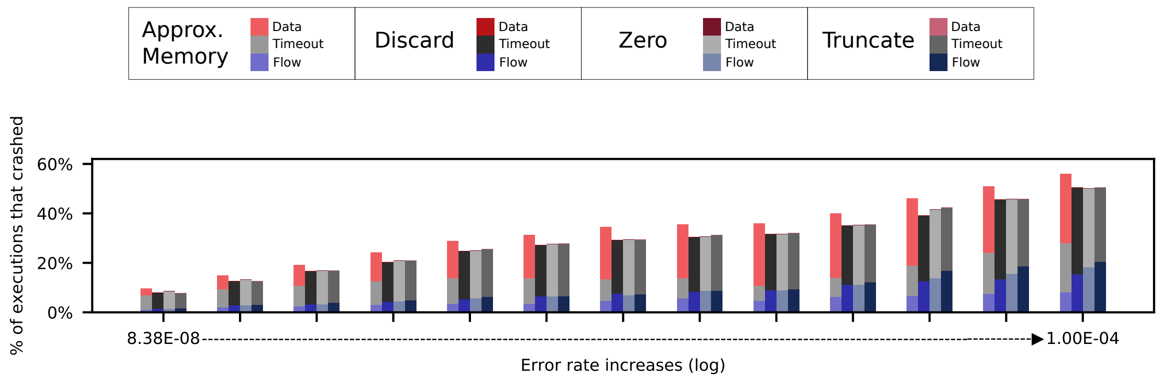


Figure 3.4: Execution crashes implementing techniques to treat incorrect pointers. *The three techniques reduce data crashes but increase timeouts by insisting on executions with (sometimes) incorrect data.*

three forms of treating incorrect pointers in isolation. `Truncate` refers to the treatment by truncating incorrect pointers into allowed memory boundaries, `zero` refers to writing the value zero in the destination register, and `discard` refers to the treatment by discarding the instruction with the incorrect pointer. The left-most bar shows the crashes for the use of an approximate memory without any protection or treatment.

As the error rate increases, the number of execution crashes also grows, however to a lesser extent when applying techniques for the treatment of pointers. The three addressing schemes have similar behavior considering the type and the number of execution crashes. Data crashes are do not happen with any application, while flow crashes and timeouts increase together with the error rate. However, `truncate` exhibits fewer timeouts and more flow crashes than the other treatments among several error rates. Timeouts are energy costly executions that do not produce any result. To avoid such wasted resources and given that the three techniques are similar in other aspects, we focus our evaluation on an implementation of `AxRAM` that uses `truncate`.

Impact of Addressing and Data Protection

The addressing scheme and the stack protection of `AxRAM` intend to reduce execution crashes that the accesses to incorrect memory addresses cause. Figure 3.5 shows the observed crashes for a case study application in three chosen scenarios. `Approximate memory` refers to a voltage-overscaled approximate memory without `AxRAM` protections. `Truncate` refers to the use of the addressing mask to treat incorrect pointers by truncation, in isolation. At last, `stack` refers to the `AxRAM` implementation of critical data isolation with stack protection only. We omit the results of `AxRAM` implementation with both techniques since this scenario eliminates all crashes in the studied application.

The trending scenario of crashes is that the higher error rates determine the higher number of crashes. Nevertheless, the errors are nondeterministic and may occur at any point in the execution. An error at a critical point may cause an execution crash. Therefore, some higher error rates may show a smaller number of crashes, but without effects on the trending line.

The addressing mask of `truncate` corrects only pointers that would fall into invalid memory locations because of a memory error when fetching the pointer. When a pointer

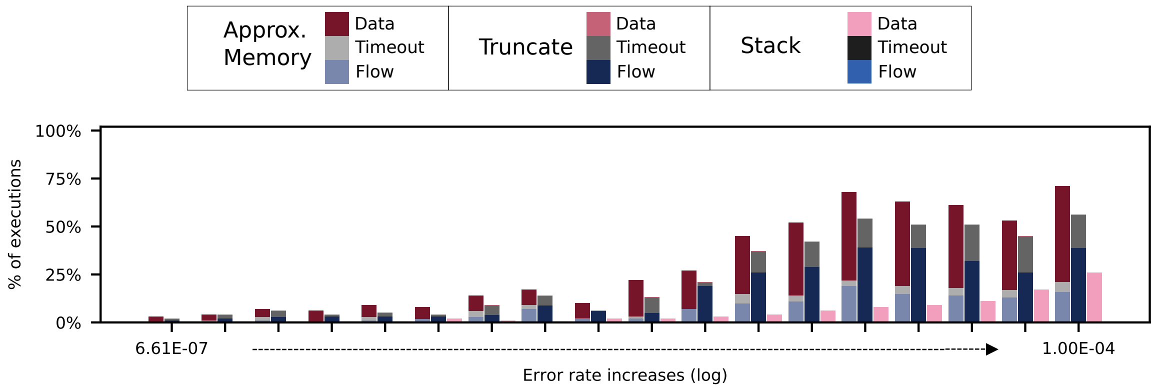


Figure 3.5: Execution crashes implementing AxRAM protection techniques in the application jpeg. *No crash happens in combining both Truncate and Stack.*

read from memory contains some error but still falls within a valid memory region, this error is undetectable by the interface, and the execution proceeds as if no error had happened. Thus, although valid, the address may point to a memory location that does not contain the expected value, causing some consequence according to how this value is used during execution:

- The value is used as part of a data region of the application (e.g., a pixel of an image for jpeg), the computation proceeds with the wrong value, and some quality degradation is perceived. This is the most common behavior perceived in our evaluated applications, where data crashes are eliminated, and the execution proceeds with impact in quality;
- The value is used as a reference to a code region (e.g., a function pointer) and the application jumps to the incorrect address, breaking execution flow and/or causing a flow crash. Our experimentation scenario includes applications such as qsort that make use of function pointers and exhibit such flow crash behavior;
- The value is used as a reference to another data region (e.g., a data pointer in a linked list) and the application fetches and uses data incorrectly, causing quality degradation or, eventually, entering an infinite loop, a timeout crash. Our experimentation includes applications such as dijkstra that make use of data structures that rely on data pointers and, thus, exhibit this timeout crash behavior.

The addressing scheme of `truncate` eliminates data crashes, while these crashes interrupt 50% of executions in approximate memory evaluations at a 1E-4 error rate. Nevertheless, flow and timeout crashes sum 21% of approximate memory executions and 56% of mask executions. The causes of more crashes of these types are the result of the scenarios discussed above. Nevertheless, the number of successful executions on mask is 44% in comparison to 39% on approximate memory at the maximum error rate evaluated.

The stack protection implementation achieves an aggressive elimination of crashes compared to both other techniques. `Stack` eliminates flow and timeout crashes at all error rates. On this technique, the interruptions due to data crash are 26% of executions at maximum error rate, while `truncate` eliminates data crashes at this error rate, but exhibits 17% and 39% of timeout and flow crashes, respectively.

The evaluation demonstrates that `stack` and `truncate` attack different types of crashes and in different ways. The combination of both techniques eliminates all crashes on the studied application. Therefore, it shows potential improvements in application resilience in the evaluated scenario.

Execution Crashes

Execution crashes cause premature interruptions in the execution flow, without producing an output. Thus, an execution crash implies a zero quality output and increases the energy cost since a new execution is necessary to recover the lost data. The `AxRAM` protection cannot eliminate execution crashes for all application domains. Especially when applications heavily rely on memory references or convergence, an execution flow

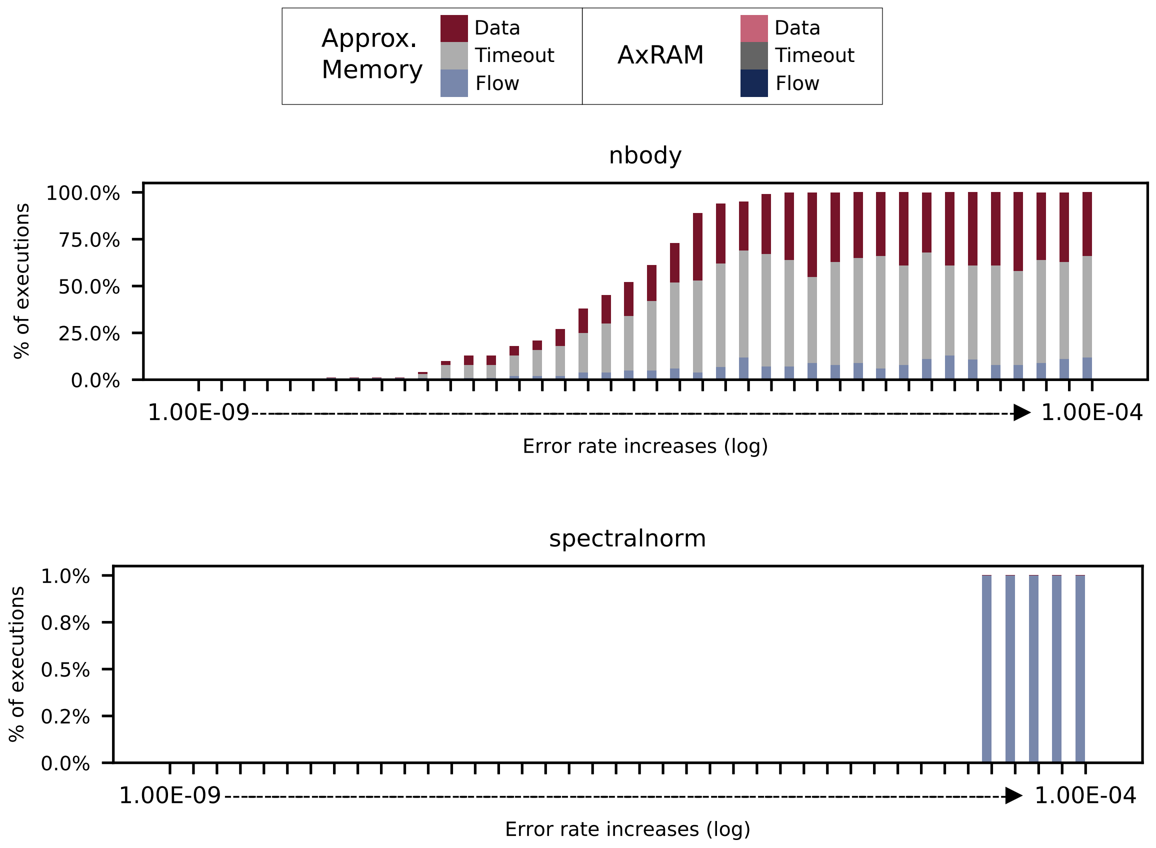


Figure 3.6: Crashes of CPU-bound applications. *These applications have a non-intensive memory usage, then the number of execution crashes is reduced.*

deviation may lead to an interruption by timeout. In the use of data approximation, the number of crashes tends to increase with the error probability. There are downside deviations at some points, because of the nondeterminism of the errors. A higher number of executions may soften these deviations.

`AxRAM` shows no data crashes at any error probability in our evaluation. The addressing mask of our design avoids data crashes due to truncating out-of-bounds addresses instead of crashing the execution. To analyze the behavior of all applications, we separate the analysis by application type. Applications of the same type have some characteristics in common, but this does not determine their crash behavior.

CPU-bound applications exhibit fewer crashes than other types due to a non-intensive use of memory, the approximate component. Figure 3.6 shows the execution crashes of these applications. In general, the use of `approximate memory` does not strongly affect CPU-bound applications since just nbody shows a significant number of execution crashes in this environment. Nbody uses data pointers to iterate over its vectors, which explains the high occurrence of data crashes. The use of `AxRAM` highly benefits nbody, once all executions produce results up to the 1E-4 error rate.

Figure 3.7 shows execution crashes to **signal-processing applications**. These applications have many error-tolerant data but crashes affect their executions more than CPU-bound applications in `approximate memory`. `AxRAM` eliminates all execution crashes in the evaluated error rates to these applications.

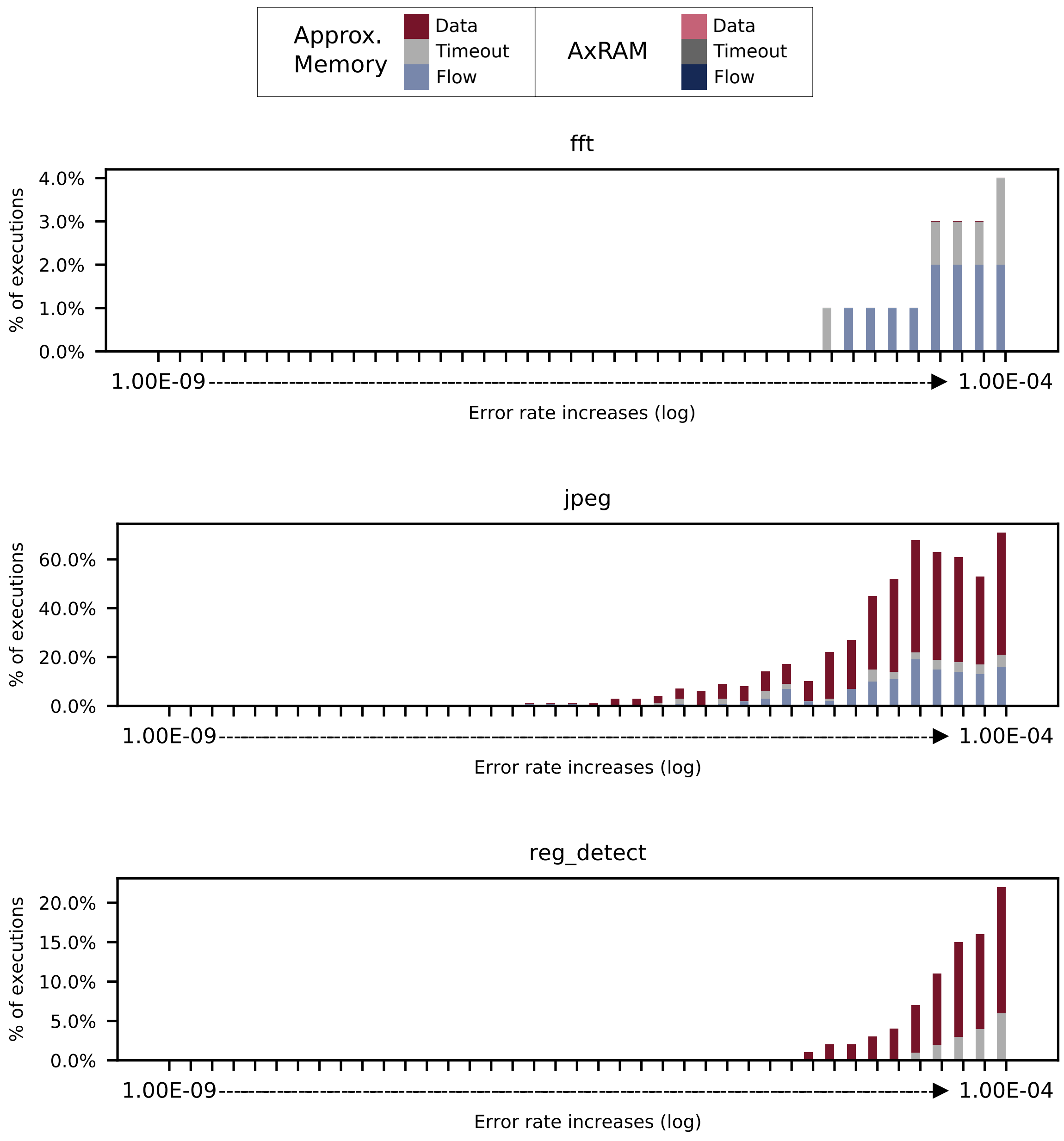


Figure 3.7: Crashes of signal-processing applications. `AxRAM` *eliminates all execution crashes in the evaluated error rates.*

Memory-bound applications are commonly more susceptible to crashes due to the intensive use of memory. Figure 3.8 shows crashes for these applications. The nondeterminism of crashes strongly acts on bunzip2 due to error recovery mechanisms in its kernel. The application bzip2 has similar operations to bunzip2 but does not have the error recovery mechanisms and is more affected by execution crashes. Dijkstra and qsort also represent corner cases in the `AxRAM` technique. Dijkstra uses a list-like structure that strongly relies on pointers to store its information. If one of these is incorrectly read, the application would loop over random data in memory, causing a timeout crash – thus although many data crashes are eliminated, timeouts take their place. Qsort uses function pointers to call the comparison routine within the sorting algorithm. Differently from data pointers, these are not recovered by the addressing mechanism and end up causing flow crashes. These are more noticeable in the `AxRAM` scenario because, after eliminating data

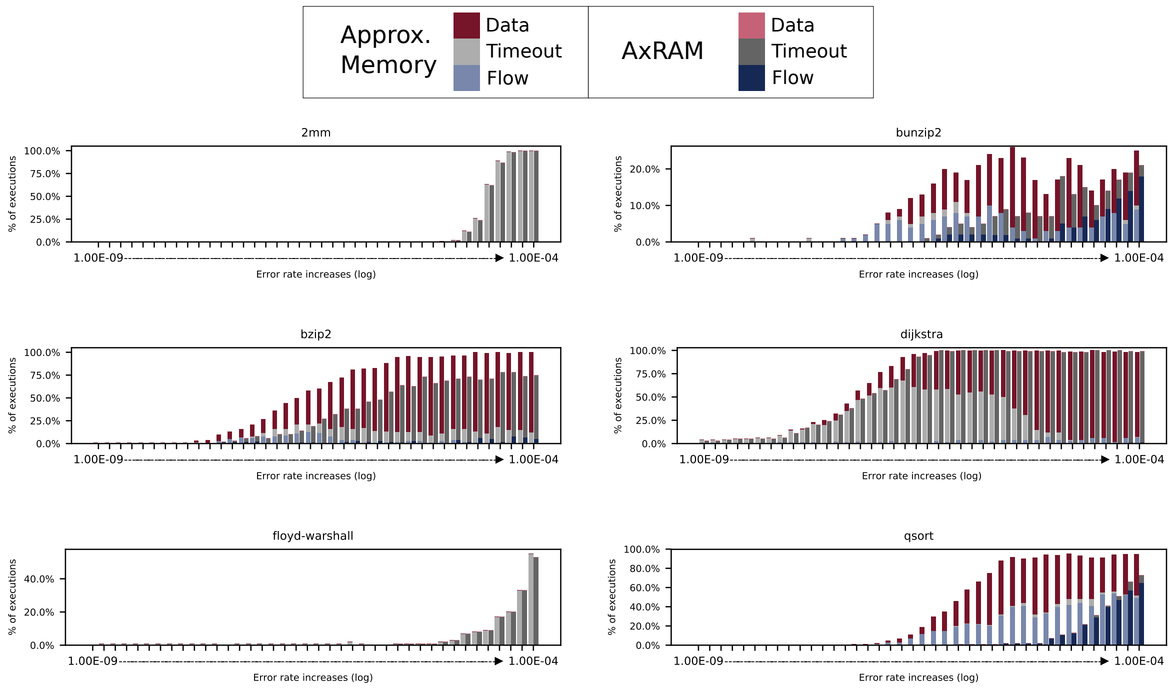


Figure 3.8: Crashes of memory-bound applications. *Some applications of this type, like dijkstra and 2mm, use data structures that make them more susceptible to execution crashes.*

crashes, the application ends up lasting longer and increasing the probability of an error affecting a function pointer. Memory-bound applications are in general strongly affected by execution crashes in `approximate memory`, and `AxRAM` protections show the potential to decrease and significantly postpone the number of crashes in three applications: bunzip2, bzip2, and qsort.

Quality

Execution crashes highly influence the average quality since each one represents a null-quality output. Nevertheless, errors in non-critical data influence the quality degradation as well and, therefore, also impact the average quality. When compared to `approximate memory`, `AxRAM` allows a higher error rate to achieve the same expected average quality for the application output, which potentially translates into higher energy savings. Figure 3.9 shows the average output quality to all executions of evaluated applications. Since each error rate point in the X-axis is a logarithm interval, a small displacement to the right that `AxRAM` allows in the curve represents several degrees of energy-quality adjustment. The results show that 8 out of 12 applications exhibit significant quality improvements with `AxRAM`.

CPU-bound applications show different patterns of the quality line behavior. Mandelbrot has an almost null impact with the use of approximate memories. Spectralnorm executions show lower quality without a significant number of crashes. Nbody gets low quality early on an abrupt fall because of execution crashes. `AxRAM` is capable of postponing the quality decrease of nbody and holds up the quality of spectralnorm.

Signal-processing applications usually tolerate more errors than other applications. The behavior of jpeg and reg_detect applications with `AxRAM` protections is the

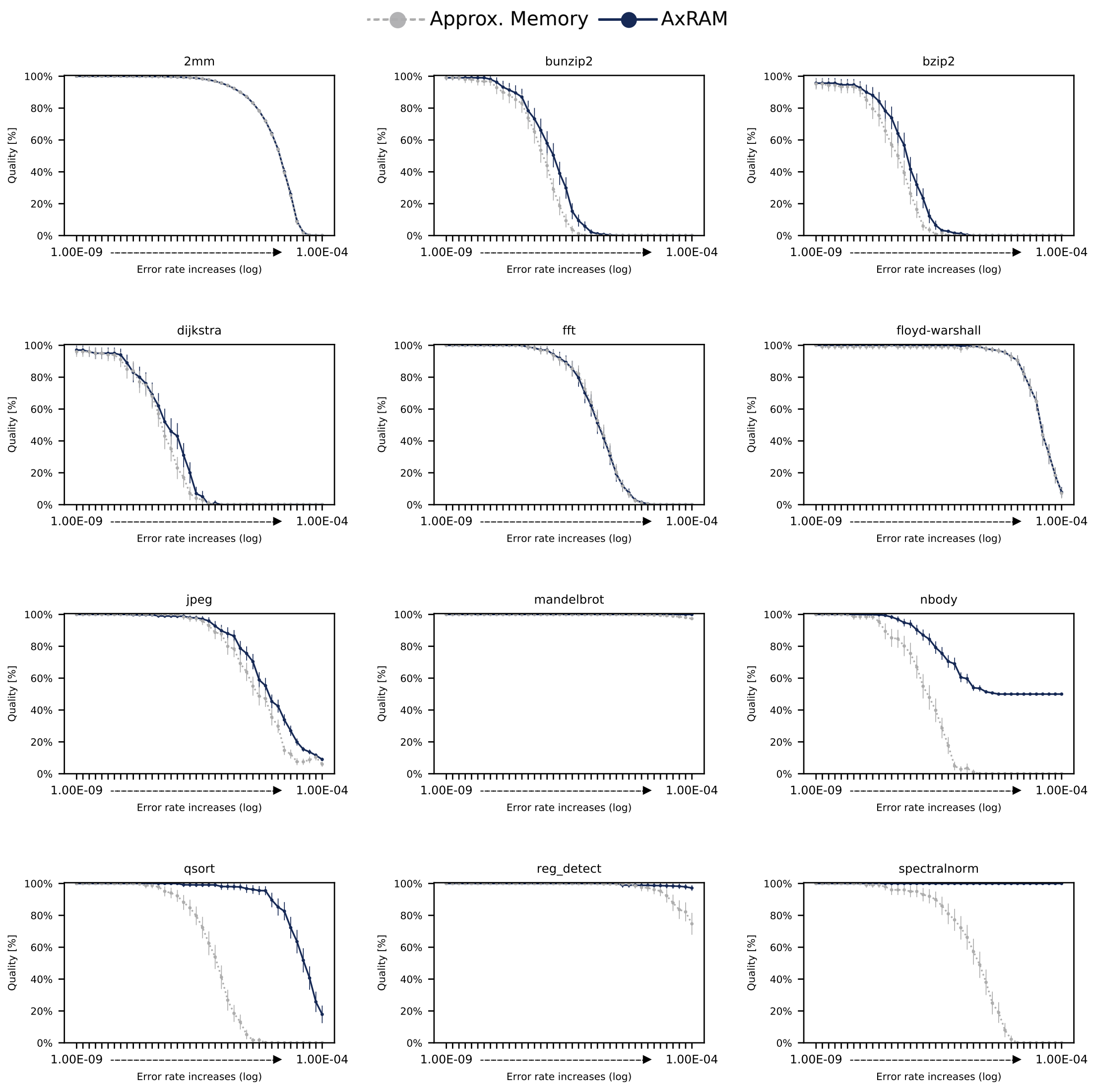


Figure 3.9: Average quality of each application. *Most of the applications have a quality improvement in several error rates with* `AxRAM`*.*

postponement of quality abrupt decreases in some steps of error rates compared with `approximate memory`. The application fft has a similar quality in all error rates in `AxRAM` and `approximate memory`.

Most of the **memory-bound applications** have a similar quality behavior on `AxRAM` and `approximate memory` environments, with or without error rate steps offset. The memory-bound applications that do not suffer significant differences in quality with `AxRAM` are 2mm and floyd-warshall. Applications bzip2, bunzip2, and dijkstra have an offset of some error rates, depending on the threshold. Qsort achieves the highest improvements in quality with `AxRAM` among memory-bound applications with several error rates offset.

In general, our results show that crashes strongly influence the average output quality of application executions. Nevertheless, some applications show a different behavior between crash increases and quality depreciation. This evidences that crashes are not the only influence on the average output quality.

Energy

Increasing the average output quality also applications to be executed at higher error rates while still meeting a quality requirement. Thus, instead of increasing quality at an energy budget, `AxRAM` can also save energy for a given quality constraint. To evaluate this, we calculate the relative energy consumption considering a quality threshold to profile the application. The baseline can be called an accurate memory, defined as a memory that yields the very low probability of error of 1 in 10^{-12} operations. The profiling of the application in this environment statistically guarantees an average quality on certain relative energy consumption. The accurate memory region of the protected stack is negligible compared to the energy consumption of the entire memory array. In our experiments, the size of the stack of all applications is at most 250 kB.

To find the relative energy consumption to an average quality, we associate each error rate with the respective energy consumption. The expected behavior without the influence of crashes is that energy consumption decreases smoothly as the error rate increases. Nonetheless, the growth of data and flow crashes causes abrupt decreases in energy consumption due to the termination of executions earlier than expected. Moreover, a large number of timeouts cause an increase in energy consumption.

Figure 3.10 shows the geometric mean of the minimum required energy to achieve average quality thresholds from 90% to 100% at a 1% step. `AxRAM` achieves more energy savings than approximate memory at all quality thresholds. Nevertheless, depending on the quality threshold `AxRAM` does not exhibit energy gains to all applications compared to `approximate memory`, e. g. at a 95% quality threshold that `AxRAM` saves energy to 9 out of 12 applications with an 8.92% mean of less energy.

Some applications are not achieving 100% quality with both `approximate memory` and `AxRAM`. Thus, an accurate execution is needed to reach this quality, which causes a considerable increase in energy consumption. Nevertheless, 100% quality represents an accurate output and a requisite to applications that execute in approximate environments is that some inaccuracy is tolerated.

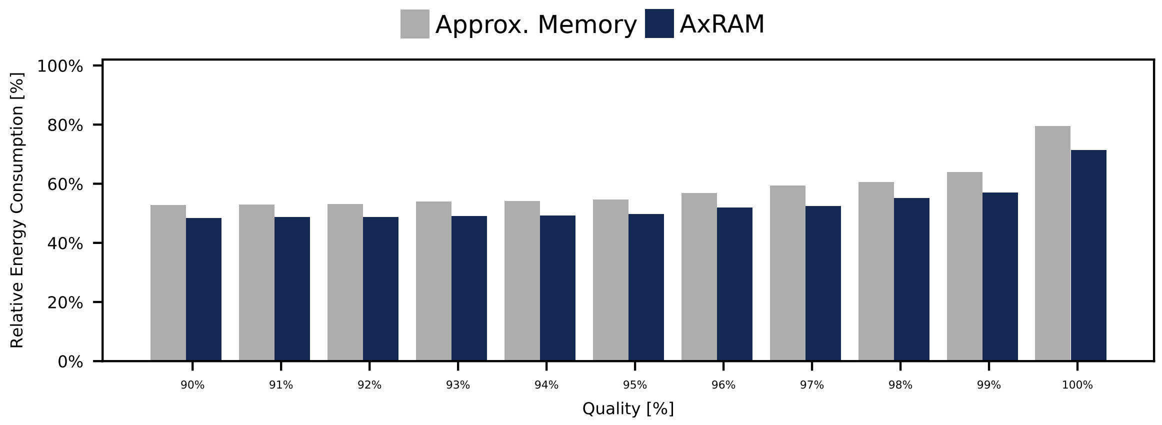


Figure 3.10: Mean of relative energy to achieve average qualities thresholds. `AxRAM` *increases energy savings by improving the error resilience for a given quality threshold.*

Quality-Energy Efficiency

Quality metrics show how much the output deviates from the original result. Energy metrics show to what extent the approximation provides benefits. These two types of metrics show different aspects of the results. Thus, we define a combined metric that represents both aspects, the Quality-Energy Efficiency (QEE). This metric is defined by $\frac{Q}{E}$, where Q is the normalized quality and E is the percentage of energy relative to the consumption of a reliable memory. Figure 3.11 shows the average QEE to all evaluated applications and error rates.

An accurate memory has QEE equals 1.0 since the quality of its outputs is 100% and its relative energy consumption is 100%. Thus, the results of QEE less than 1.0 are inefficient due to being below the results of an accurate memory. All evaluated applications show some error rates with QEE higher than 1.0. `AxRAM` has a higher peak of QEE than approximate memory for 10 out of the 12 evaluated applications due to the postponement of the QEE fall to higher error rates. The QEE line of mandelbrot, reg_detect, and spectralnorm shows that these applications are not affected by memory errors with `AxRAM` protections.

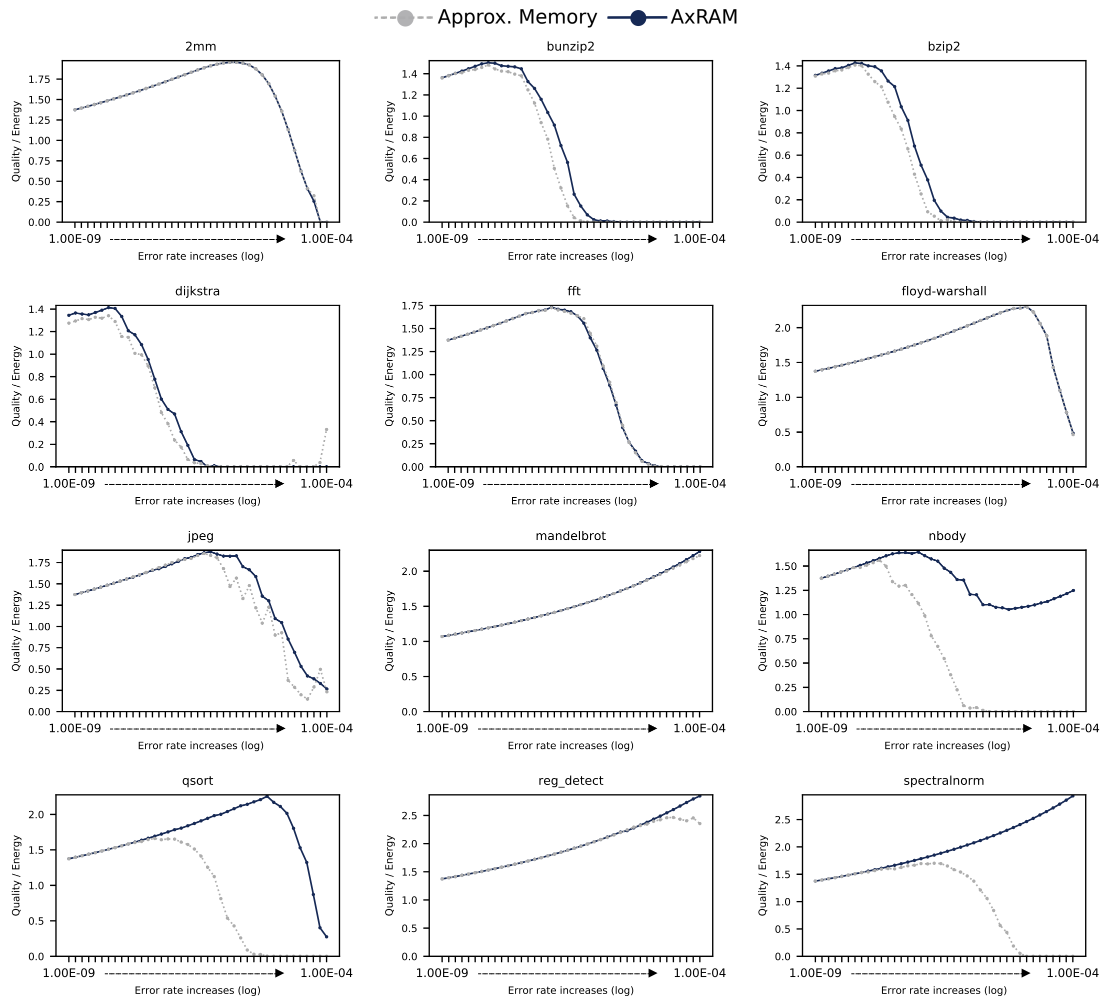


Figure 3.11: Quality-energy efficiency for the evaluated applications. *Without a quality threshold, the maximum efficiency of quality and energy depends on the error resilience of each application.*

Despite QEE representing a combined metric of quality and energy, it does not show individual values for quality or energy. A very low energy consumption may represent an increase of QEE even with low quality. Applications dijkstra and jpeg show this increase in the higher error rates with approximate memory, where the average energy consumption is very low due to crashes at the beginning of the executions, but the average quality is almost null. Despite that, the peak of QEE in our evaluated scenario to all applications achieve quality higher than 90% for both `AxRAM` and `approximate memory`.

3.2.5 Discussion

`AxRAM` proposes a generic memory architecture that implements a set of approximate states, which are operating points that induce read and write errors in the stored data. By controlling the error rate and the region of the memory array that is affected, the `AxRAM` interface allows an external agent to control the degree of approximation provided, inducing energy savings by tolerating some quality depreciation. In our simulated evaluation, however, we employ `AxRAM` in a limited scenario in which one single-threaded embedded application runs in bare metal in the CPU, with full control over the entire memory array and `AxRAM` control knobs. In this scenario, each induced memory error is a single random bit flip in the memory data word per operation.

`AxRAM` exposes to the environment – the application, in the simpler embedded system scenario, an application-level library, the OS, or the middleware – control knobs in the form of memory-mapped registers. Although these registers can be written at any point at execution time, changing the approximate state or the approximated memory banks is a potentially time-consuming operation, similar to changing power states or Dynamic Voltage and Frequency Scaling (DVFS). For this reason, it is desirable to perform coarse-grained control of approximations, reducing state changes to a minimum, such as in the evaluated scenarios, where we set the approximate region and state at application kernel start-up and keep the setting until the end.

The proposed interface avoids execution crashes transparently and without significant performance and energy costs. `AxRAM` has two main features that increase execution resilience: an addressing scheme to treat incorrect memory references and critical data protection of the system stack. Our experimental evaluation shows that the critical data transparent protection decreases the number of flow crashes, while the addressing schemes can avoid data crashes. An implementation of `AxRAM` in an embedded computing scenario featuring a dual-voltage SRAM memory with configurable reliability shows a reduction of 51% of execution crashes across all error rates compared to an unprotected approximate memory. At 95% average output quality, `AxRAM` shows energy savings of 9% and a higher peak of quality-energy efficiency for 10 out of 12 applications when compared to unprotected approximate memory. For the same 95% average output quality, when compared to a system with non-approximate memory producing exact results, `AxRAM` reduces energy consumption in half.

3.3 Transparent Resilience for Approximate DRAM

Checkpointing and rollbacking mechanisms can reduce the error impact from the approximate memories on results, improving average quality by recovering broken results, but at the cost of instrumentation or modifications in the programming language [53, 122]. A transparent re-execution restarts the execution without these interventions but requires twice initialization overheads [118]. An approximate re-execution can alleviate the cost of restarting an application [53], but memory approximations are probabilistic, and re-executing into the same approximation level may lead to another invalid result.

In this section, we explore alternatives for transparent resilience of applications with approximate main memory and recovering by approximate re-executions. We propose to predetermine approximation levels to perform re-executions at a higher level with a lower probability of a new invalid result. Furthermore, we introduce acceptance tests with simple verification that detect invalid results produced by execution crashes or Silent Data Corruption (SDC). These functions check, without a golden accurate output, whether an approximate execution result is valid and contains the required data format.

Our previous interface [26] models an error-prone memory as a single high-level entity that is solely responsible for storing all data, which limits the approximation model on architectures without hierarchical design. We reduce the impact of errors by pushing the approximation to DRAM, a more energy-intensive point in the memory hierarchy, which is accessed through error-free caches that alleviate this impact. Furthermore, we adapt the addressing scheme of `AxRAM` for this environment with a software implementation that considers the virtual addressing space controlled by a supervisor system. The main contributions, built upon previous work [28, 29], are:

- An approximate re-execution mechanism for instances that generate invalid results considering the nondeterministic errors in the execution environment;
- Detection of invalid results through lightweight acceptance tests without having a golden accurate output;
- An evaluation of the impact of error from different memory hierarchy levels and a comparison between state-of-the-art alternatives for transparent resilience in an execution scenario more compatible with systems with multi-level memory hierarchy.

Mechanisms of transparent protection have the potential to eliminate execution crashes at some operating points. Nevertheless, to transform these crashes into instances that fit a quality requirement, these executions have to generate a valid and higher quality result. We compare and mix features from the interfaces AxRAM [26] and Crash Skipping [118] in our proposed environment. Our results show that approximate re-execution improves energy savings by up to 4 pp when compared to accurate re-execution, with negligible impact on quality. A combination of features from transparent resilience interfaces avoids up to 70% of crashes and achieves energy savings from 14% to 31%, depending on the application, with acceptable quality degradation.

3.3.1 Design

Transparent protection mechanisms attempt to automatically protect applications from critical data errors without programmer intervention. We alleviate the impact of errors on application quality by triggering approximate re-executions when invalid outputs are detected. Furthermore, we evaluate transparent hardware and software-level resilience mechanisms for approximate memory that can avoid a large fraction of critical errors.

Impact of Errors in the Memory Hierarchy

Approximations in different levels of the memory hierarchy could represent different impacts on the application in terms of benefits and quality deprecation [76]. In general, caches alleviate the number of accesses into a more energy-expensive and slower main memory. In a scenario with an approximate DRAM, a precise cache also alleviates the impact of errors from this main memory because of the reduced number of accesses on the main memory. Although previous proposals mitigate approximation among more than one level in the memory hierarchy [77, 130], our preliminary experiments, shown in Section 3.3.4, evidence that a precise cache can reduce the number of application-visible data errors from the last-level DRAM. Thus, the approximation only at the DRAM main memory could maximize its energy gains with less impact on the output quality using precise caches. Moreover, an approximate DRAM maximizes energy benefits since this memory level represents the most energy-hungry point of the hierarchy, which can represent more than 60% of total memory energy breakdown [130]. Thus, this potentially represents an improvement of orders of magnitude on error resilience without programmer intervention.

Approximate Re-execution

To measure the quality of the results, we need to compare them with a reference output. The reference output is the result of an accurate execution that has no energy benefits, and, thus, is not feasible at the system-level design. Nevertheless, invalid results can be detected when caused by execution crashes or errors in critical data. To this end, we need an evaluation function for each application that distinguishes outputs that are not valid. Every invalid output is indicated with null quality and certainly needs a re-execution to generate a valid result. The re-execution is a simple mechanism that can be triggered without programmer intervention by an OS or a runtime system. Although an accurate re-execution generates an accurate output, it adds an energy overhead that may reduce the approximation benefits. Furthermore, in an approximate environment, an accurate output is not necessary in the first place.

To maximize the benefits of the approximation, we propose to re-execute, in approximate mode, each execution instance that produced an invalid result. Approximate re-execution has been proposed through specialized source code and programming language that re-execute parts of the application in the same approximation level [53]. However, re-executing in the same approximation level exposes the application to the same error level and the same probability of a crash or invalid outcome, and a specialized source code demands changes in applications developed to commodity hardware. To overcome these issues, we propose to perform the re-execution of a failed instance at a lower level of approximation than the original execution. If another invalid result is retrieved, another re-execution is scheduled in the next level, successively. A guard-banded adjustment of memory parameters defines the approximation level zero, which results in error-free execution. As a transparent process-level mechanism, this re-execution does not demand changes in the source code or instrumentation by checkpointing. However, each re-execution adds process initialization overheads since the application is re-executed from its beginning. On the proposed approximate re-execution method, the approximation level is decreased while there is no valid output from the last execution. In the worst case, the last execution is performed at approximation level zero without errors and thus a valid result is guaranteed. Furthermore, we can check the integrity of execution outputs based on a simple verification of the data. Thus, we propose to use acceptance tests to detect invalid results even in the case of SDC. This function returns whether an output is evaluable by a quality metric without having an accurate reference, checking labels and critical information required by the application. The simplicity of such a checking mechanism results in negligible runtime overhead to determine whether the results are valid or not.

Transparent Interface Mechanisms

Transparent protection mechanisms of critical data, in the literature, focus on resilience, trying to maintain the execution flow and converge to a valid output. Incorrect memory references are the main cause of crashes, thus protection of control flow pointers and treatment of data pointers have been proposed in AxRAM [26]. The loss of the control flow is also a concern in approximate memory environments. Mechanisms of instructions replacement by a no-operation (`nop`) instruction have been proposed to overcome this problem in Crash Skipping (CS) [118].

We propose a combination of transparent mechanisms from AxRAM and CS into an interface that avoids execution crashes by protecting the control flow, treating data pointers, skipping faulty instructions, and preventing application stalling. The allocation of the system stack addresses into reliable memory protects control flow pointers with a minimum penalty in energy savings due to the usually small size of the stack compared to the entire application data. The treatment of data pointers into an environment with virtual addressing with MSB truncation validates non-existing addresses but is not sufficient to avoid all data crashes. Thus, we combine this addressing scheme with the replacement by `nop` instructions. However, these mechanisms can increase the number of instances that fall in indefinite execution, thus another mechanism counts the replaced instructions and stops the execution if a threshold of avoided crashes is reached.

3.3.2 Implementation

The mechanisms of transparent resilience evaluated in this work are based on AxRAM [26] and CS [118]. Therefore, we implement these interfaces to show and compare the results achieved with their mechanisms in our proposed environment. We consider mechanisms of hardware and software implementation of the transparent interfaces. The hardware-level implementation of AxRAM allocates the system stack addresses into a memory region with the approximate level zero, protecting control flow pointers stored into these addresses. The other main feature of AxRAM is a treatment to out-of-bounds memory accesses that we implement with the truncation mask of 39 bits in our virtual memory environment, following the RV64 Sv39 standard page-based virtual-memory system. The CS implementation (referred to as `CSi`) considers the hardware mechanisms, a skipping threshold of 20 for all evaluated applications, and implements instruction granularity to skip crashes, which is a configuration calibrated with a unique execution of a single random application for achieving energy savings.

The base AxRAM hardware implementation to treat out-of-bounds addresses is not enough to match the expected results in a virtual memory scenario. Truncating the most significant bits covers only pointers that fall out of the allowed addressing space but does not validate whether they are a match for an existing virtual memory page. Thus, a software-level implementation of this scheme considers the page allocation to find a likely correspondence for the virtual address. In the RV64 Sv39 page-based virtual-memory system, the 39-bit addressed virtual memory space is divided into 4 KiB pages and organized into three levels, allocating a 9-bit Virtual Page Number (VPN) identifier within each level and a 12-bit intra-page offset [125]. We consider that, when the hardware raises an access violation exception, one of these partial VPNs suffered a bitflip that corrupted the virtual address. Thus, we search the Page Table Entry (PTE) for a VPN at a hamming distance $= 1$ in comparison to the virtual address that caused the exception. If a correspondence is found, we create a new PTE pointing the faulty virtual address to the correspondent physical address, allowing the execution to proceed. This PTE avoids another search in the case of an error with the same incorrect address but adding data that occupies memory and a possible entry in the Translation Lookaside Buffer (TLB). If no correspondence is possible, the execution crashes in a segmentation fault. This alternative implementation is referred to as `SW-AxRAM`.

3.3.3 Methodology

Transparent interfaces allow for controlling error rates at the hardware level, the OS, or a runtime system using approximation knobs. We adopt a model where non-user level instructions are protected, thus only the application is exposed to errors and the OS runs accurately. Different from the methodology of Section 3.2.3, we evaluate the impact of the error with multiple cache levels in the memory hierarchy, in which the OS manages a virtual addressing scheme where multiple applications have access to the memory array. Furthermore, the energy savings on an approximate DRAM consider the proportional access to the error-free data, thus, some application that allocates a small error-free region but has intensive use of it increases the overhead of the protections.

Simulation Environment

Our environment is built upon the Spike RISC-V reference ISA simulator [66], and user-privileged memory accesses are replaced by software models that can expose data to errors [34]. These errors are bitflips persisted in memory that can occur at any bit of the row buffer with a given probability. The RISC-V Proxy Kernel controls virtual memory addresses and the execution environment. To enforce a higher stress on the approximate memory, our simulator performs a cache flush after 10^5 instruction cycles to increase the number of memory accesses. In our simulation, traces of DRAM accesses are transformed into DRAM commands and timestamp marks by Ramulator [59] to evaluate the energy consumption through DRAMPower [11]. Our simulated memory hierarchy has two independent 32KB L1 instruction and data caches and a single 128KB L2 cache, and the DRAM specification is DDR3 1600Mhz 64bit, 8 banks, 2 ranks, 1024 columns, 8 bytes burst length, 16384 rows, tRCD 13.75 ns, tRP 13.75 ns, and 1.35 V nominal VDD. To account for energy, we consider that each equally-sized fraction of the memory contributes equally to the aggregated dynamic energy cost. Thus, if certain memory regions use a different operating point to protect or expose data to errors, their energy cost is proportional to their utilization.

Error and Energy Model

Our error scenario considers a controlled environment, where the approximation level refers to a calibrated static error rate. To calculate the energy impact of exposing data to a certain error rate, we derive a relation between the DRAM supply voltage of the DRAM array and the bit error probability from data collected with scaled voltage and fixed temperature and latency parameters [13]. Current commercially-available DRAM does not support dynamic changes in the supply voltage of the DRAM arrays, thus, this environment requires minor changes in the power delivery of the DIMMs similar to Voltron [13]. These changes should avoid errors in the peripheral circuitry, maintaining the nominal voltage on these components while allowing dynamic adjusts in the supply voltage of the DRAM array. We assume that these modifications have insignificant energy impact and model a median scenario of error probabilities. In steps of approximation that are not covered by the parameters from extracted data, we consider the exponential relation between voltage and error to design regressions in the form

$$error = A \times e^{(B \times vdd)} \tag{3.1}$$

where *error* is the bit error probability and *vdd* is the supply voltage of the DRAM, validating the *error* value between 0 and 1. The values for a median error scenario are $A = 1.796 \times 10^{68}$ and $B = -155.87$ with coefficient of determination $R^2 = 98.51\%$. We did not consider other variables that could affect the error as a dynamic component of the memory, such as the temperature. However, we intend to profile the behavior and output generated by the application exposed to errors, thus a calibration over other factors is applicable to proposed techniques and also other error models.

Applications and Quality Functions

We evaluate applications from AxBench [131], cBench [38], and Polybench [88] on our experiments. The considered applications are atax, correlation, dijkstra, fft, jpeg, and sobel, which represent the behavior of general-purpose applications that manipulate data that tolerate approximation in their results. For simulation purposes, we use standard input and output as the source and the destination of data. However, no other modifications were implemented within the applications, such as annotations in the source code, instructions, or data to control approximations, maintaining the transparency of the interfaces.

The selected quality metrics are Fraction of Equal Elements (FEE) for correlation and dijkstra; Structural Similarity Index (SSI) for jpeg and sobel; and Mean Absolute Percentage Error (MAPE) for atax and fft. More details of these metrics are given in Section 2.1.4. The acceptance test of each application verifies if the respective output is valid without having the accurate output to trigger a re-execution. The acceptance test for jpeg and sobel checks whether the data contains a valid image header in the expected dimensions. For atax, correlation, fft, and dijkstra, the acceptance test verifies if the number of elements in the output is coherent with the size of the input data.

Approximation Levels and Metrics

We consider the approximation level zero as 1.35 V, the nominal voltage of DRAM. The predetermined approximation levels are 10 between 1.02 V and 1.11 V with 10 mV steps. The applications are compiled with GCC/G++ 9.2.0 for RISCV-V from riscv-gnu-toolchain with flags -O3 and -static. To measure the energy benefits and quality degradation at the approximation levels, we perform 100 executions of each application at each approximation level. These executions intend to profile the application to determine its behavior in the approximate DRAM environment. Since the memory approximations are nondeterministic, executions in the same approximation level may produce different outputs. Thus, the expected quality and energy for each level are the average from all executions.

Each of the 100 random execution instances, at a given approximation level l, of the target application account for a relative energy cost, measured by DRAMPower, and an output quality, given by the quality metrics. We aggregate the average energy consumption and average quality at each level as μ_{W_l} and μ_{Q_l}, respectively. We also observe the outcome of each execution instance to produce the probability of a re-execution to be triggered at the given level, δ_l. Thus, the Expected Quality (E_Q^l) and Expected Energy (E_W^l) for each level l are taken as the statistical expected value of the random variables energy cost and quality, considering the mean values μ_{Q_l} and μ_{W_l}, as shown in Equations 3.2 and 3.3, respectively.

$$E_Q^l = (1 - \delta_l) \times \mu_{Q_l} + \delta_l \times E_Q^{(l-1)} \quad (3.2) \qquad\qquad E_W^l = \mu_{W_l} + \delta_l \times E_W^{(l-1)} \quad (3.3)$$

Considering the relative energy savings are the difference between the energy consumption from accurate (100%) and approximate executions, the Expected Energy Savings (E_S^l) are similarly given by Equation 3.4.

$$E_S^l = 1 - E_W^l \tag{3.4}$$

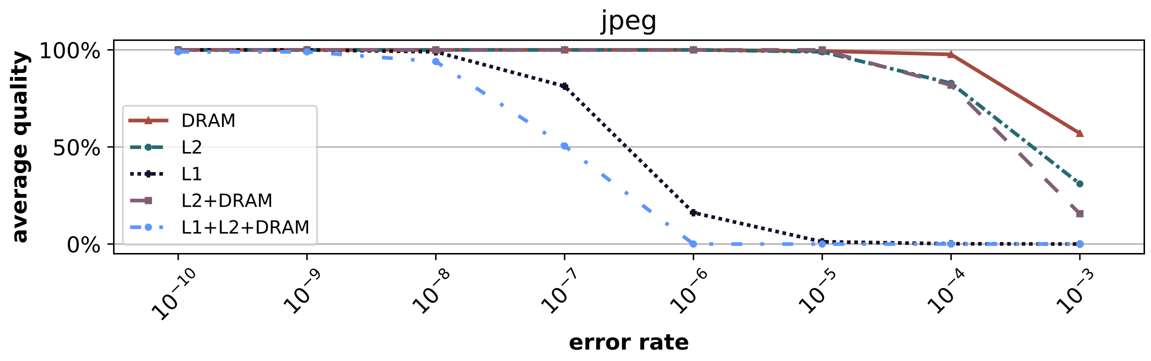
3.3.4 Evaluation

The best operating point for each application is the approximation level that achieves the highest energy savings while still fitting in a quality requirement. Our experiments search for the best operating point by analyzing several executions in each approximation level. The remainder of this section presents our evaluation results with the proposed transparent mechanisms..

Impact of Errors from Levels of the Memory Hierarchy

Figure 3.12 shows how errors injected at three different levels in the memory hierarchy impact the average quality of results for the jpeg application, for error rates in logarithmic intervals from 10^{-10} to 10^{-3}. The highest impact occurs when errors are combined in L2 and L1 caches and the DRAM, zeroing quality at the 10^{-6} level. For error rates of 10^{-5}, the L1 alone nullifies the quality of results. Avoiding errors at the L1 cache allows the applications to survive up to three orders of magnitude more errors. Therefore, amongst the evaluated hierarchy levels, the error source from cache L1 presents the highest impact on decreasing the quality of results.

The fewer cache misses and the flush performed by the context switching operation soften the difference between the impact of errors from L2+DRAM, L2, and DRAM. However, an improvement of at least one order of magnitude can be noticed with errors only from DRAM. Thus, the further away is the error source from the CPU, the less significant is the impact of the error on application results. Taking into consideration a breakdown of energy consumption of these hierarchy levels where DRAM corresponds to the major energy consumption in memory hierarchy [130], our evidence shows that the minor impact of error source is in the most energy-hungry level. Therefore, the use of error-free L1 and L2 caches have the potential to soften the impact of an approximate DRAM by several orders of magnitude.



.

Figure 3.12: Average output quality for jpeg when errors are present in different levels of the memory hierarchy. *Quality improves when errors occur only in the main memory, but not in the caches.*

Acceptance Tests

Transparent resilience interfaces avoid behaviors that would otherwise result in a crash. However, these behaviors may result in SDC, which leads to invalid outputs. Interfaces avoid execution crashes but increase the number of instances that generate results with useless or null quality. Thus, other invalid outputs that cannot be detected by the OS may need a re-execution to generate a valid result.

Figure 3.13 exhibits the percentage of re-executions triggered by crashes detection, the proposed output validation with acceptance tests plus crashes detection, and an oracle that detects every execution that resulted in quality lower than 80%. Such an oracle implementation is not achievable, since it would need an accurate output as a reference to validate and measure the output quality, which nullifies the energy gains of the approximate memory. For all applications and error rates, the inclusion of acceptance tests covers more instances than crash detection only. Although the crash detection may cause

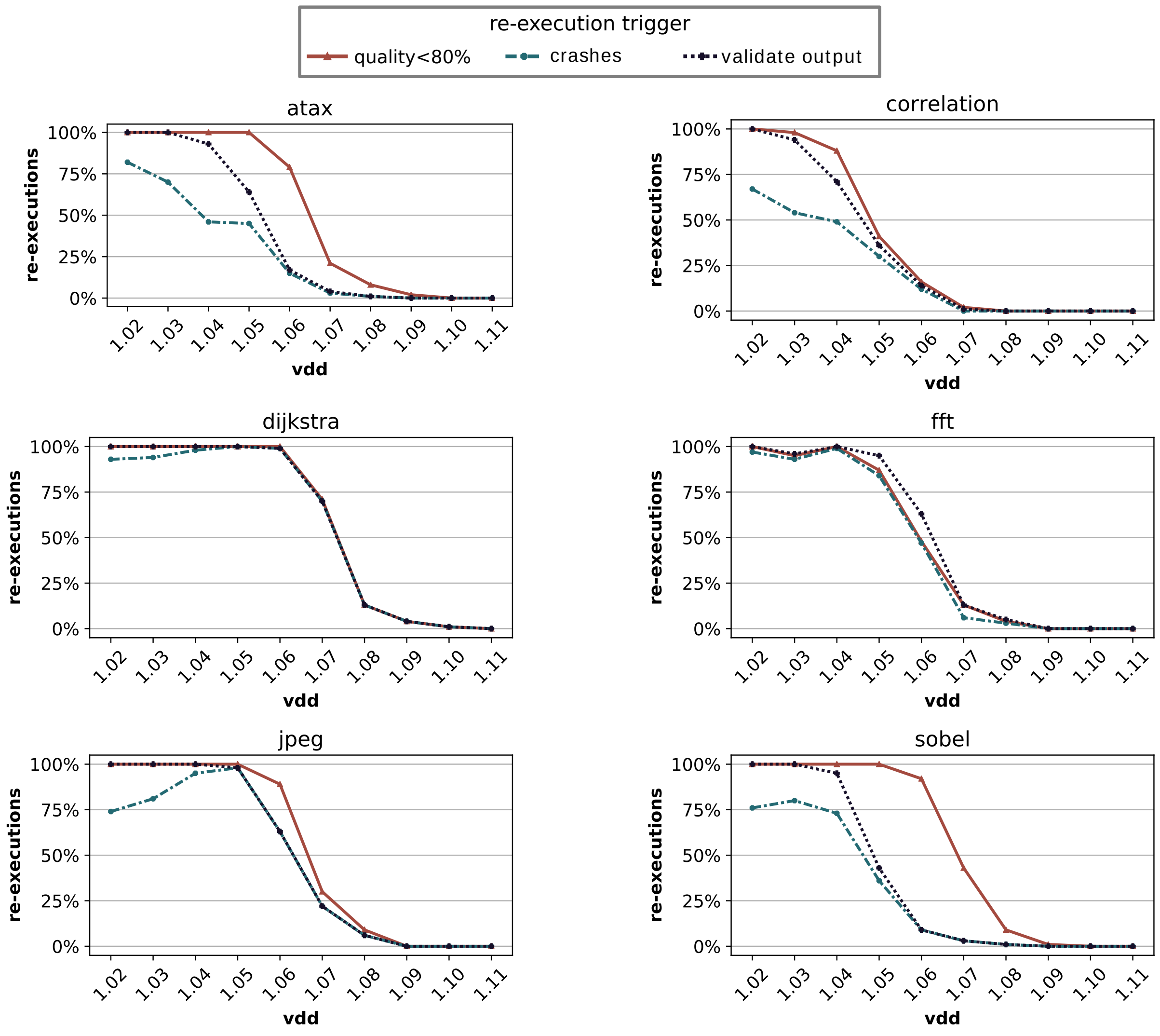


Figure 3.13: Percentage of executions invalidated by crash detection only, crash detection and output validation, and an oracle with 80% target quality. *Output validation improves upon crash detection performance for all operating points. Reaching oracle performance on all applications would require costly quality assessment.*

false positives, such as when a crash happens after a valid result is generated, as in fft application where this trigger surpasses the oracle at some levels, it prevents false negatives from the acceptance test, such as when an image header is correctly written and the execution crashes within the pixel data region. Thus, the combination of acceptance tests and crashes detection is more efficient as a re-execution trigger than crash detection only.

Approximate Re-execution

The re-execution methods bring another component to the energy-quality trade-off, which adds energy overhead for each re-execution but increases the average quality by recovering invalid results. For the approximate method, this energy overhead can be softened with lower re-execution probabilities in the next approximation level, or be raised when the chances for another invalid result are high on this level. Nonetheless, higher chances for other invalid results occur at approximation levels where the low average quality of results and the nonexistent energy savings prevent these operating points to be valid levels of execution. Table 3.3 shows the expected savings (E_S) and quality (E_Q) for all applications without any transparent resilience mechanisms (`AM` model), in approximation levels from 1.07 V to 1.10 V, in which most applications present the more significant energy savings maintaining high-quality results. In all these levels and applications, the energy overhead does not make the approximate re-execution surpass the energy cost of accurate re-executions on our experiments, where the E_S of the approximate method is at most the same as the one of the accurate method.

The probability of a re-execution is a key characteristic to achieve energy savings using re-execution methods, especially for the approximate re-execution that relies on the re-execution probability of the next lower vdd. Sobel presents the highest E_S values because of fewer crashes and invalid results on lower vdds, increasing the E_S of the approximate method up to 4 pp at 1.07 V. Dijkstra, fft, and jpeg nullify the energy gains at 1.07 V through accurate re-execution, because of the higher re-execution probability that adds an overhead based on the nominal energy consumption. However, this overhead is decreased by the approximate re-execution, which can present slight energy savings on jpeg and dijkstra at this approximation level. The higher number of delayed crashes and invalid results produced by SDC increases the energy overhead of fft, which postpones its energy

Table 3.3: Expected Energy Savings (E_S) and Expected Quality (E_Q) for approximate and accurate re-execution. *The approximate method achieves higher savings with slight differences in quality.*

	vdd →	1.07		1.08		1.09		1.10	
application	re-execution	E_S	E_Q	E_S	E_Q	E_S	E_Q	E_S	E_Q
atax	approximate	**10.7%**	74.8%	**18.9%**	91.1%	19.3%	98.0%	18.51%	100.00%
	accurate	7.7%	**76.3%**	17.8%	**91.2%**	19.3%	98.0%	18.51%	100.00%
correlation	approximate	**15.3%**	96.6%	20.0%	98.9%	19.3%	100.0%	18.52%	100.00%
	accurate	13.3%	**96.7%**	20.0%	98.9%	19.3%	100.0%	18.52%	100.00%
dijkstra	approximate	**1.0%**	68.5%	**12.5%**	83.5%	**18.0%**	98.7%	**17.10%**	99.67%
	accurate	–	80.7%	8.3%	**83.8%**	17.6%	98.7%	16.92%	99.67%
fft	approximate	–	44.9%	**15.6%**	91.6%	**17.2%**	100.0%	18.52%	100.00%
	accurate	–	49.0%	12.9%	91.6%	16.6%	100.0%	18.52%	100.00%
jpeg	approximate	**3.0%**	88.6%	**20.1%**	89.3%	19.3%	99.6%	18.52%	100.00%
	accurate	–	93.2%	18.6%	**89.4%**	19.3%	99.6%	18.52%	100.00%
sobel	approximate	**30.5%**	68.5%	**22.6%**	92.9%	19.6%	99.3%	18.52%	99.99%
	accurate	26.5%	**69.8%**	22.3%	92.9%	19.6%	99.3%	18.52%	99.99%

savings to 1.08 V on both re-execution methods. Correlation presents equal E_S peaks to approximate and accurate re-executions due to the fewer invalid results and crashes in its best approximation level. Some operating points show no difference between the accurate and the approximate methods because of having small chances of triggering re-executions. For 5 out of 6 applications, the peak of E_S occurs with approximate re-execution at the cost of a decrease on the E_Q.

In general, the approximate re-execution shows higher savings at the approximation level closer to the edge of a high probability of crashes because of the minimum overhead added when compared to the accurate method. Energy savings are significantly high in comparison to the depreciation in quality. In all cases when quality and energy metrics show different results, the approximate method achieves higher efficiency in the combined efficiency. Therefore, our proposed approximate re-execution method achieves the highest energy savings for the 6 applications with a minimal impact on quality.

Transparent Interfaces

The number of valid results varies depending on the execution with transparent interfaces. These interfaces are orthogonal to the re-execution methods, which can recover invalid results even when no resilience mechanisms are used, as the results with `AM`. Resilience interfaces insist on executions that would crash or result in invalid outputs, trying to generate valid and higher quality results.

Valid Executions

Figure 3.14 shows the percentage of executions that produce valid results for two evaluated features, `SW-AC` and `SW-ACw`, besides an approximate memory without implementing resilience interfaces (`AM`). `SW-AC` implements software and hardware addressing schemes from `AxRAM` plus `CSi` features and protection of the stack region from errors. `SW-ACw` implements all these features, except for stack protection. `SW-AC` shows the higher number of valid executions, and, consequently, the lower re-execution probability.

In general, the use of interfaces decreases the need for re-executions at lower vdds, however, higher quality results are usually generated at higher vdds, where crashes and invalid outputs are not so frequent using our proposed architecture. Furthermore, in our environment, the error impact is alleviated by error-free L1 and L2 caches, which can protect small temporary variables, and the approximate re-execution decreases the impact of recovering invalid outputs, thus reducing any gains of protecting the application stack. Moreover, triggering fewer re-executions does not necessarily infer more energy savings. Other factors influence these gains, such as timeouts (the more energy-wasting execution crash) and the execution phase at which a crash happens. The more delayed an execution crash is, the higher is the energy overhead of an instance.

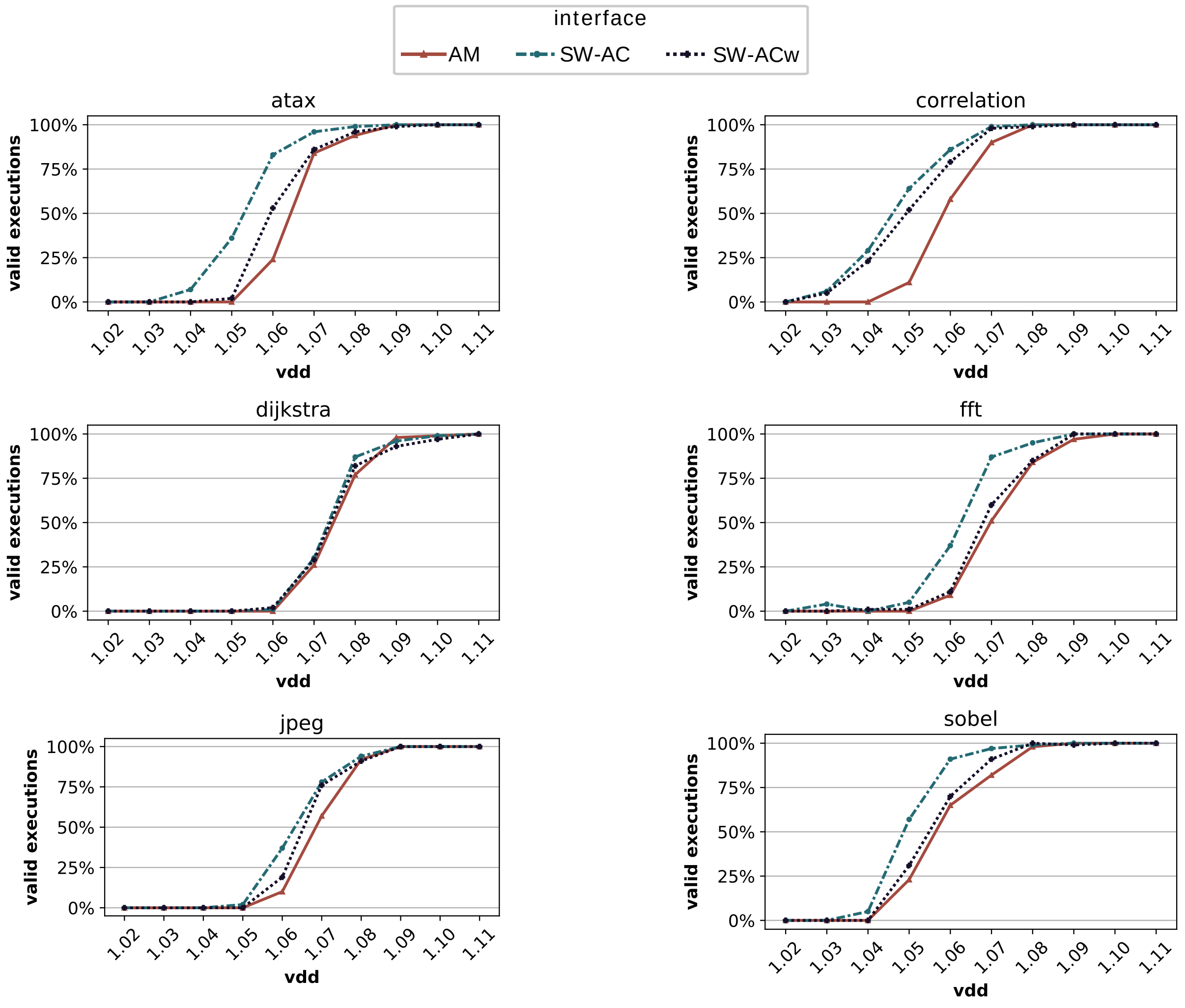


Figure 3.14: Executions that produce valid outputs with the evaluated interfaces. *In general, resilience mechanisms increase the percentage of these valid executions.*

Expected Energy Savings and Quality

In our environment, the best operating points are in the vdds where the probability for a re-execution is relatively low, and, when a re-execution is needed, the approximate re-execution is performed at a level with this probability almost equal to zero. These operating points range from 1.07 V to 1.10 V. Comparing data from this range, Table 3.4 shows the operating points with the highest energy savings achieved for all applications per interface, where the protection mechanisms match the highest expected quality for 5 out of 6 applications. However, the decreased cost of recovering invalid outputs or crashed instances by approximate re-executions results in higher energy savings for `AM`, which matches the highest savings for 3 out of 6 applications.

The software-level addressing scheme achieves the most benefits on quality and energy for 5 out of 6 applications. However, the stack protection can impact differently depending on the overhead and structures used by the application. Only for sobel this feature increases the combined benefits. The stack protection, however, is derived from correct memory allocation for the stack region and can be configured at runtime [26]. Thus, an analysis of the impact of each protection feature and its respective overhead is necessary before associating an application to an interface.

Table 3.4: Operating points of the best E_S for each application.

application	interface	vdd	E$_Q$	E$_S$	application	interface	vdd	E$_Q$	E$_S$
atax	`AM`	1.09	98.0%	19.3%	fft	`AM`	1.10	100%	18.5%
	`SW-AC`	1.07	80.3%	**20.0%**		`SW-AC`	1.09	100%	17.7%
	`SW-ACw`	1.09	**99.9%**	18.8%		`SW-ACw`	1.09	100%	**19.3%**
correlation	`AM`	1.08	**98.9%**	**20.0%**	jpeg	`AM`	1.08	89.3%	**20.1%**
	`SW-AC`	1.07	98.6%	**20.0%**		`SW-AC`	1.09	99.6%	18.9%
	`SW-ACw`	1.07	98.5%	19.3%		`SW-ACw`	1.09	**99.7%**	19.3%
dijkstra	`AM`	1.09	98.7%	**18.0%**	sobel	`AM`	1.07	68.5%	30.5%
	`SW-AC`	1.10	99.0%	10.0%		`SW-AC`	1.07	**74.9%**	30.5%
	`SW-ACw`	1.11	**100%**	17.8%		`SW-ACw`	1.07	71.6%	**30.6%**

Comparison with Other Interfaces

Figures 3.15 shows the expected energy savings for the proposed interfaces compared with hardware-only implementations of `AxRAM` and `CSi`. From a certain point, execution crashes are rare, and thus resilience interfaces have no significant benefits and add energy

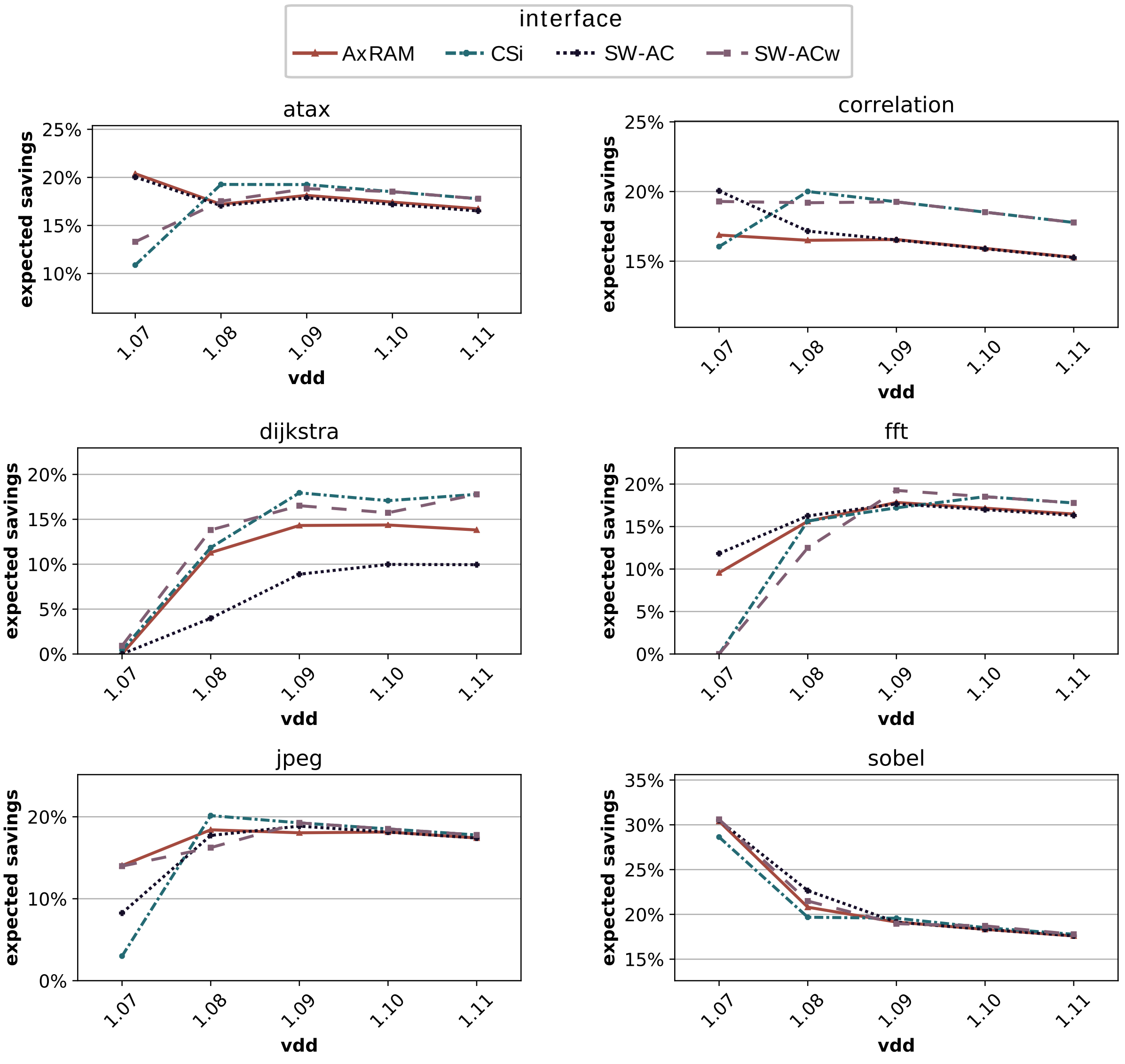


Figure 3.15: Expected energy savings for evaluated interfaces. *The combined interfaces benefit from the lower overhead of* `CSi` *at higher vdds, and the higher savings of* `AxRAM` *at lower vdds.*

overhead, especially when protecting memory regions from errors. This behavior can be noticed on the decreased benefits with sobel starting from 1.08 V.

Dijkstra is a representative application that shows the highest difference between the four interfaces. This application performs a lot of accesses on the application stack, resulting in significant overhead on its protection and favors the energy benefits of `CSi`, in isolation. Moreover, `CSi` is a low-overhead hardware-only mechanism, in comparison to the software implementation `SW-AxRAM`. However, if the stack protection is removed from `SW-AxRAM`, the energy benefits get closer to pure `CSi`, while still providing high-quality results and, thus, maximizing the energy-quality trade-off. This behavior pattern is perceived in most applications.

Thus, the unprotected stack is determinant for achieving high energy savings for most of the applications at higher vdds. However, interfaces that protect some critical memory regions achieve higher savings at lower vdds, as seen in atax. The combined interfaces have a lower overhead at higher vdds and protect critical data at lower vdds, achieving most of the benefits of `CSi` and `AxRAM` and maintaining high quality results for a wider range of operations.

3.3.5 Discussion

Approximate memories operate below guard-banding parameters trading energy for quality degradation of results. In the border of this guard-banding, the execution of some applications brings low or even no errors. However, as the approximation level grows, errors may lead to invalid results or crashed executions. Mechanisms for transparent resilience aim to avoid execution crashes to generate valid results and improve the chances for a high-quality result in higher approximation levels without any program annotation.

Despite transparent, these mechanisms require some runtime system to manage the approximation. Thus, the approximation control is carried to the supervisor layer, where an OS or a middleware detects invalid executions and triggers recovery methods. The transparent protection of critical data is also managed by a supervisor system that controls regions of the memory that usually contains these data, but the higher energy savings with this protection are achieved at the operating points in the margin of the occurrence of crashes.

This section compared and explored state-of-the-art mechanisms for transparent resilience, detecting invalid results to trigger re-execution mechanisms. Our results show that approximate re-executions achieve energy savings of 4 pp with negligible loss in quality. Furthermore, the proposed combined interfaces, that merge transparent mechanisms, benefit from higher energy savings with lower overhead than interfaces that implement isolated mechanisms. Despite being application-specific, the use of acceptance tests as re-execution triggers can improve the detection of invalid outputs up to 30% by simple and lightweight validations.

3.4 Approximate Dynamic Allocation

Environments with a supervisor system like an OS have information about memory allocation, data access patterns, and addressing schemes of each application at runtime and can act as a middleware in the approximation control. Previous work [111] proposes the control of an approximate memory through data allocation on the heap region but restricted to a specific application that needed to be adapted to this environment through annotations. However, being the dynamically allocated space a region that usually contains most of the data resilient to errors for some applications, the supervisor system can control the approximate region employing the information of the program allocation.

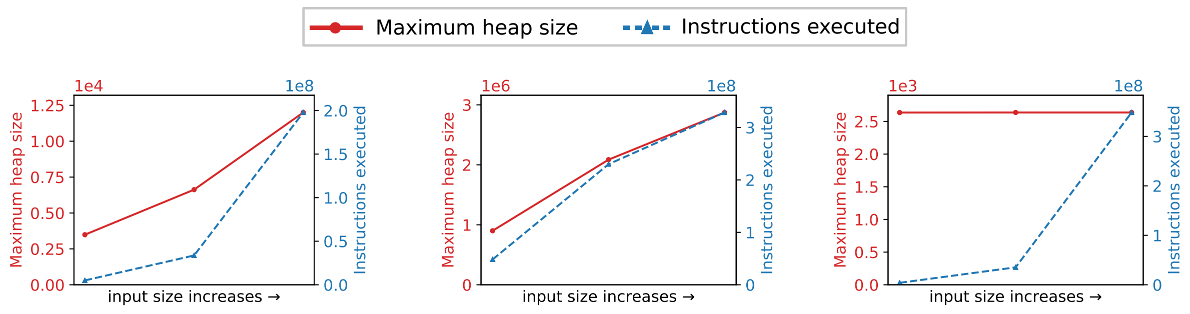
In this section, we propose an allocation scheme for the approximate data controlled by the supervisor system. This scheme acts without annotations in the application data and stores all dynamically allocated data in the approximate region by default. The program is divided into a new layout of virtual memory segments, adding the approximate heap section, which refers to a contiguous physical region of memory addresses, where the stored data is exposed to controlled errors. Moreover, we provide a method of reliable dynamic allocation to protect critical data in the case of annotated applications. This method acts through a system call that increments the program break (the end of the program's data segment) and returns the pointer to the beginning of the new allocated accurate data region, virtually contiguous with the approximate region but physically at another memory region. Thus, we benefit both portability from and to non-approximate environments and keep the execution environment compatible with off-the-shelf architectures. The main contributions are:

- A transparent allocation scheme that uses the heap memory region to store approximate data and achieve energy savings;
- A data approximation model compatible with and easily portable from off-the-shelf environments;
- Approximation controls offered to the application through dynamic allocation into error-free memory regions.

We evaluate our proposal on a voltage-scaled DRAM and compare the results with transparent interfaces and annotated applications. Our proposal achieves a reduction of up to 25 pp of crashes, with energy savings up to 20% with improved quality when compared with other transparent interfaces.

3.4.1 Design

A typical layout of the data allocation of a program divides memory regions into sections of text, data, Block Starting Symbol (bss), heap, and stack. The text section contains read-only data of the executable instructions from the program. These instructions are fetched from the memory by the processor to decode and execute them. The data section contains initialized data and the bss section contains uninitialized static variables and constants of the application. The heap section contains dynamically allocated data, while



(a) dijkstra (b) jpeg (c) pi

Figure 3.16: Maximum heap size according to three different input sizes and their corresponding executed instructions. *Some applications, like jpeg, increase the heap space according to instructions executed, thus the dynamically allocated region has a direct relation with the input size.*

and the stack section contains return addresses and temporary data. Both heap and stack sizes change during the execution.

Applications that manipulate common resilient data, such as images and audio files, usually have an input of unknown size to compute. These applications, when memory-intensive, usually appeal to dynamic allocation to store the data to be processed or the partial results of their computation. Figure 3.16 shows the size of the heap region according to the number of instructions executed with different input sizes for three applications, jpeg, dijkstra, and pi, with distinct memory access patterns. The jpeg application has the most heap-intensive utilization because the dynamically allocated data grows together with the number of executed instructions according to the input size. Dijkstra has a more slightly growing of the heap, while the workload of pi is indifferent to the input size to allocate data in the heap region.

We propose to store the dynamically allocated data into the approximate region of the memory by default, providing a transparent approximation of this region and an easily portable execution environment. Our proposal takes more advantage of applications like jpeg, where the dynamically allocated data contains most of the data that are resilient to errors to be processed. This approximation is transparent since there is no need for annotations or interventions by the application to ensure data approximation. The execution environment is easily portable because it exploits a previous memory organization layout of common environments. Thus, applications built aware of our environment run in previous commercial off-the-shelf environments without changing their source code.

Transparent Approximation Scheme

The heap section stores the dynamically allocated data of the program. When the OS loads the binary program, an Executable and Linkable Format (ELF), after the loading of the instructions, data, and bss sections of the application, the memory has an empty heap. We propose to allocate the heap section into a memory location of unreliable storage. In this location, memory parameters are changed to operate with lower energy consumption, but the stored data are exposed to errors.

When the program is loaded and the heap section is empty, we change the physical region of the new memory spaces requested by the application to allocate data into the approximate region of the memory. The references to the program continue contiguous but the physical addresses are allocated into different parts. Thus, instructions, data, and bss sections are stored in the reliable memory region, and the heap section is stored in the approximate memory region. The stack section also is stored into the reliable region, hence, just the dynamically allocated data is approximate by default, in exchange for energy savings in the memory operation. From the application's view, this approximation is transparent since no explicit changes are needed to enable it.

Error-free Allocation

The heap section contains only approximate data by default, which makes this technique transparent to the application. However, programmers can build or port an application that is aware of this environment and may need to dynamically allocate some critical data. Therefore, we propose a method to provide reliable allocation to the application, which may ask for error-free dynamic allocation to the supervisor system. This method is performed through system calls that provide a way to dynamically increase the size of the allocated reliable memory for the application.

In our proposal, the default behavior when the program requires dynamic allocation is to perform a syscall and mark the addresses as approximate, allocating them into a different region than the previous memory sections, which are stored into an error-free region. The program break is a reference of the OS that indicates where the data segment of the application ends. Unix-based systems have the system call SBRK that allows the application to increment the program break, receiving an integer value (the increment) as a parameter [55]. In our environment, the SBRK syscall is available and also increments the program break but only in the approximate data segment. We propose a new system call with similar behavior to SBRK, receiving an increment as a parameter but, instead of allocating approximate data, the newly allocated addresses are marked to point to an error-free memory region. Thus, instead of providing an allocator for approximating data and error-free allocation by default, we provide an explicit allocator for reliable storage and allocate approximate data by default.

3.4.2 Implementation

Controlling the dynamically allocated approximate data involves mapping where to store these data in the physical memory and where to find the reliable sections of the program. An OS usually makes available to the program a virtual address space, a range of memory references that are converted to the physical addresses. In the virtual address space, the program may have a contiguous memory region that is detached in the physical memory. The implementation of our proposal involves mapping two physical heaps that are in the same location from the program's view but are separated in the physical memory.

Figure 3.17 shows the proposal of our allocation scheme, where the heap section may contain some blocks of reliable storage when the application allocates them explicitly. All blocks reliably allocated are contiguous in the virtual address space, but they increment

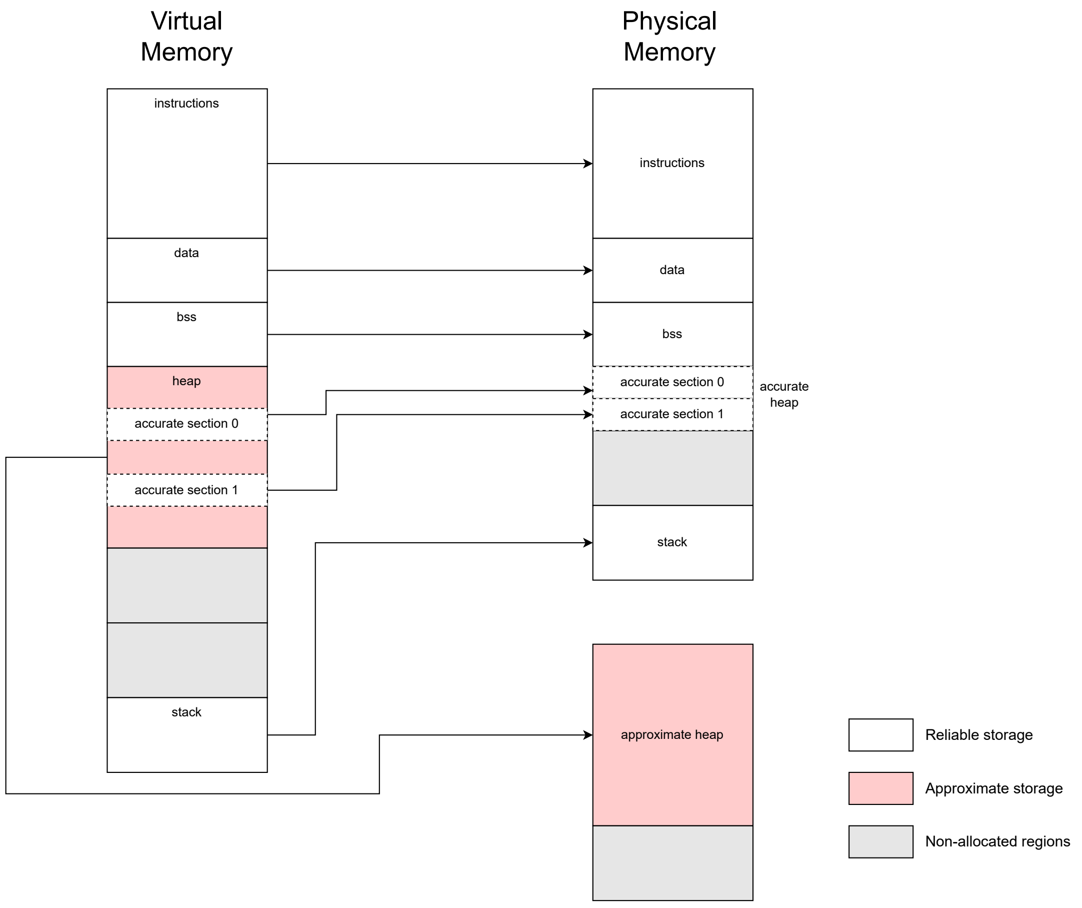


Figure 3.17: Allocation scheme of our proposal. *By default, the virtual heap contains only approximate data, however, the application may require reliable blocks in this section that are allocated in a different physical location.*

another reference of the program break in the physical memory. Thus, our system takes advantage of the reliable memory regions to provide a non-transparent option for the application to dynamically allocate critical data.

The OS should maintain two references for the program break, one for the approximate allocation and other for the error-free heap. This proposal takes advantage of the error-free memory allocated to the application at the cost of dividing the heap addresses. Depending on the size of each allocation, the page tables size may have detached fragments that could impact the performance of the data access and the size of the stored data.

3.4.3 Methodology

Our environment is built upon the AxPIKE simulator [35], which replaces memory accesses within the user region with software models that can expose data to errors. The simulation environment and configuration are the same as Section 3.3.3, except for the filter of the user-privileged access that is controlled by user memory regions. Thus, instead of the supervisor system protecting memory accesses from privileged instructions, it controls memory regions on user and supervisor spaces, where only accesses in the user space are exposed to errors.

Error Model

We consider a hardware characterization where the supply voltage of the DRAM data array is controlled according to the specified error rate. We evaluate fixed error rates of 9 logarithmic intervals from 10^{-9} to 10^{-1}. To calculate energy from a specific supply voltage, we formulate a logarithmic regression based on a characterization from the literature [13] relating Vdd with error:

$$vdd = \log_e(\sqrt[B]{\frac{error}{A}}) \tag{3.5}$$

where A and B are constants taken from the characterization with the respective values of 1.796×10^{68} and -155.87.

Applications

We evaluate applications available in AxBench [131] and MiBench [45]. Table 3.5 shows these applications, their considered quality metric (explained in Section 2.1.4), the validation functions (which validate the results for re-execution), the respective input size on the experiments, and the number of executed instructions with each input. The validation functions represent the minimum data requirements to make the execution results evaluable. Each combination of application and input is executed 100 times to ascertain nondeterministic error behavior. The applications are compiled with GCC/G++ 9.2.0 for RISCV-V from riscv-gnu-toolchain with flags -O3 and -static.

Table 3.5: Applications used to evaluate the proposal of a transparent approximate heap.

application	input size	quality metric	exec. insts.	acceptance test
dijkstra	compl. graph, |V|=100	FEE	2.0×10^8	at least 10 paths 5 nodes
inversek2j	10,000 coordinates	MAPE	9.1×10^8	25% of the expected size
jmeint	5,000 coordinates	FEE	4.7×10^8	25% of the expected size
jpeg	768x512 image	SSI	3.4×10^8	check image size
mm	512x512 int matrices	MAPE	3.5×10^8	25% of the expected size
sobel	512x512 image	SSI	4.1×10^8	check image size

Comparison Interfaces

We compare our proposal with `AxRAM` [26] and Crash Skipping (`CSi`) [118]. The implementation of `AxRAM` considers the virtual address space and treats incorrect pointers by truncating them as explained in Section 3.3.2. `CSi` is calibrated at instruction granularity with a threshold of 5 skipped instructions, parameters obtained through a single execution of a random subset of the applications to maximize energy savings.

We also evaluate our transparent interface (with implicit approximate heap) against a method of explicit approximation with annotated data through an approximate malloc function (where the application has error-free data by default and asks for every approximate data). The annotated application is not a transparent interface but serves as an oracle for their average quality. Moreover, we evaluate this annotation strategy with our proposal of explicit reliable allocation (where the application is annotated to ask for error-free allocation). We modify the applications jpeg [131] and dijkstra [45] to perform this comparison because jpeg allocates a vector of pointers to then allocate the image pixels and dijkstra uses an adjacency list to represent the graph. Thus, both applications have significant critical data that is stored in the heap section.

3.4.4 Evaluation

Transparent interfaces protect critical data in common for many applications. However, annotations protect specific critical data of the applications, being more effective to protect against executions crashes and obtaining higher average quality. However, as the number of error-free accesses increases, also the energy overhead of these protections decreases the approximation benefits. In the remainder of this section, we analyze the results obtained by the proposed transparent approximate heap face to annotations that explicitly identify the data amenable to approximation and two transparent interfaces, `AxRAM` and `CSi`. Lastly, we change applications to explicitly protect some critical data allocated in the heap section and compare strategies that explicitly protect and approximate data by default versus another that explicitly approximate and protect data by default.

Crashes

Figure 3.18 shows the percentage of invalid executions when executing the application in Vdds that expose data to errors with protections of our proposed interface (approximate heap), `AxRAM`, and `CSi`. Except for dijkstra at 1.05 V, the transparent approximate heap exhibits lower crashes than `AxRAM` and `CSi`. Thus, protecting non-dynamically allocated data from errors avoids a large number of crashes.

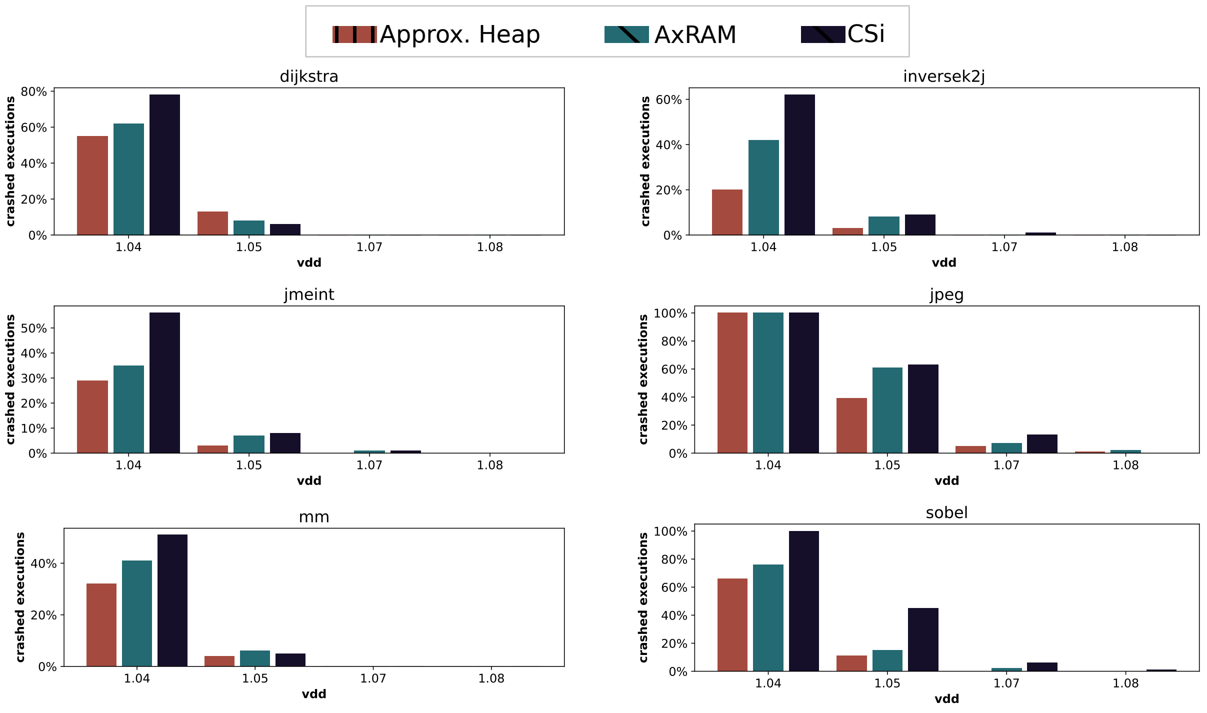


Figure 3.18: Invalid results generated by executions when executing with different transparent interfaces. *In general, protecting all data but the heap achieves fewer crashes.*

At lower Vdds, the error probability is higher and also the occurrence of crashes and production of invalid results. For example, at 1.04 V 50% of the executions produce invalid results with approximate heap, on average. In the same approximation level, `AxRAM` produces 59% and `CSi` 75% of the execution results as invalids. As the error probability decreases, crashes and invalid results tends to zero. For applications inversek2j, jmeint, jpeg, and sobel the approximate heap presents zero invalid results at the same approximation level that at least one interface still generates invalid results.

Quality

Figure 3.19 exhibits the average quality of the interfaces obtained executing amongst several Vdds, where the approximate heap is the transparent protection mechanism that is closer to an annotated application. The annotated application has identified data structures and variables that are critical through programmer interventions and modifications in the source code, thus this approach is not transparent. A transparent interface will not surpass the average output quality of an annotated application in general cases, however, the results achieved through annotations serve as references to a higher quality oracle to the transparent interfaces.

The approximate heap has different data isolation from `AxRAM`, while `AxRAM` exposes all application data to errors but the program stack, the approximate heap protects all data regions but the heap. `CSi` does not protect any data region from errors, just avoid execution crashes by skipping instructions that would crash the application. Thus, the data isolation offered by `AxRAM` and approximate heap usually improves the average quality

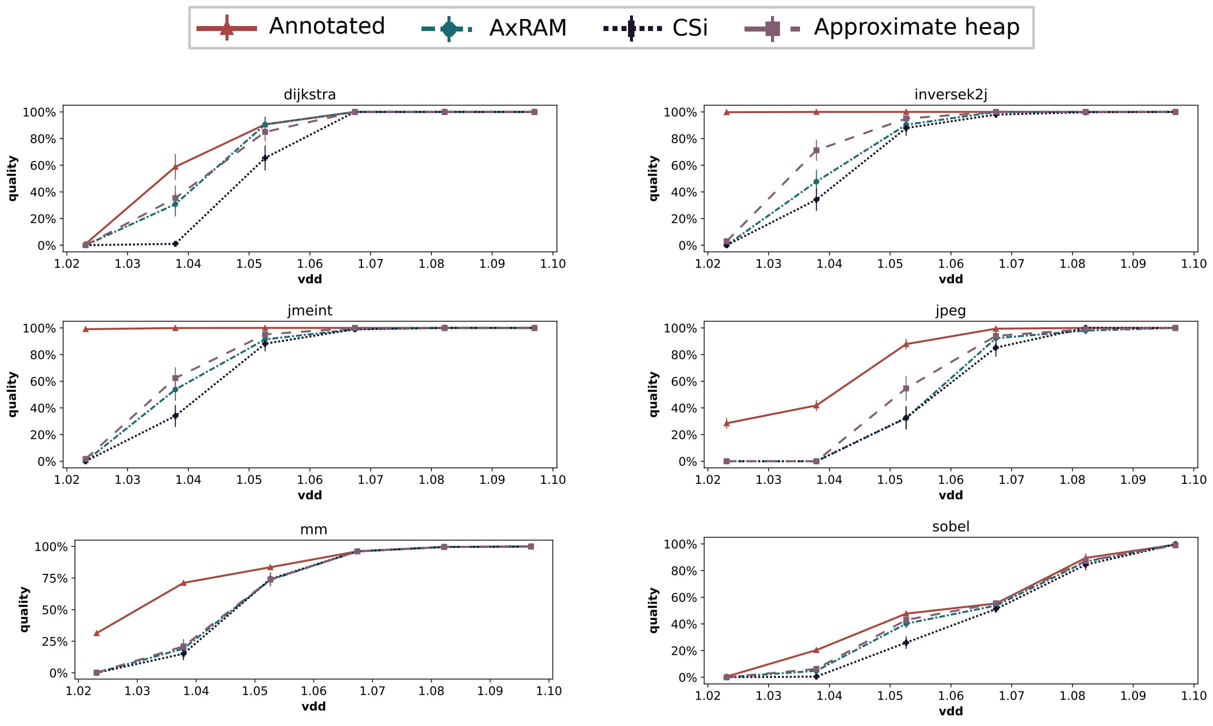


Figure 3.19: Average output quality of the evaluated applications among several Vdds. *A transparent interface with an approximate heap achieves quality closer to annotated applications than other transparent interfaces for most of the Vdds.*

of the results. The general case for all applications is the annotation of an approximate malloc obtaining the higher average quality and the approximate heap being the closer transparent interface for most of the Vdds.

Energy

Figure 3.20 shows the energy savings relative to an execution at nominal voltage among different error probabilities, where an annotated application achieves higher savings at lower vdds due to non producing invalid results, which reduces the number of re-executions. However, the energy savings depend on how the application produces the approximate results and accesses the data. For example, dijkstra has 61% of its memory accesses into the stack section but protecting this region does not represent a corresponding reduction in the invalid results. Therefore, protecting the stack adds an overhead that decreases energy savings for this application with approximate heap, `AxRAM`, and annotations.

The general behavior is the approximate heap having closer curves to the annotated application. However, in applications inversek2j, jmeint, and jpeg the energy savings of approximate heap is surpassed by other interfaces at higher Vdds due to the fewer errors which makes harmless the probability of critical errors. Thus, in these Vdds, the protection of non-heap sections increases the energy overhead.

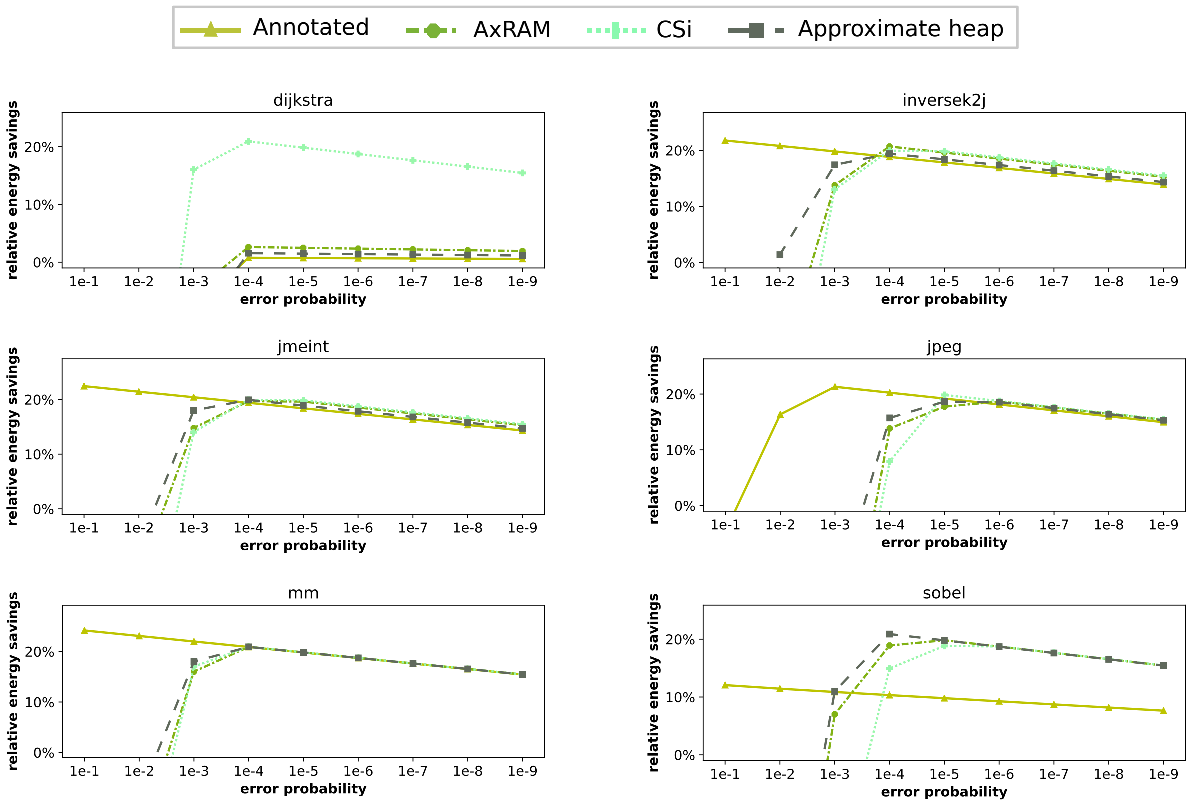


Figure 3.20: Average energy savings relative to a nominal execution. *Some applications, like dijkstra, benefit from lower protections on their data.*

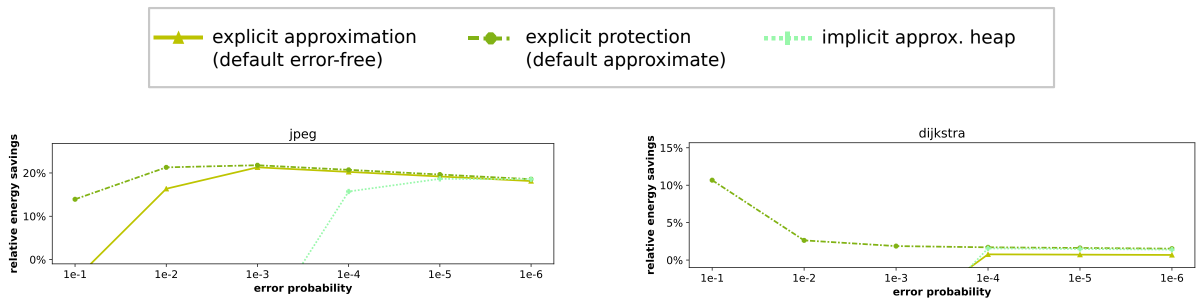


Figure 3.21: Energy savings with explicit approximation, explicit error isolation, and implicit approximate heap. *Approximating dynamically allocated data by default may guide to higher energy savings.*

Changed applications

Our proposal involves the approximation of the heap section by default. However, an application may be changed aware of this feature and request reliable allocation for some critical data stored in the heap. Figure 3.21 presents the relative energy savings for jpeg and dijkstra applications annotated to request reliable allocation of critical data in the heap, compared to an environment with explicit approximation (where all data but the annotated are error-free) and the implicit approximate heap. For both applications, the explicit annotations for protecting data achieve higher energy savings for most of the error probabilities. This proposal allows for protecting non-contiguous critical data, like in dijkstra, adding an overhead of memory space but that increases the energy savings with the approximation.

3.4.5 Discussion

The layout of the program allocation divides the memory into traceable sections. The heap section contains data allocated on demand by the application. In this section, we explored the approximation of the heap section by default, exposing all dynamically allocated data of the applications to errors. This proposal performs transparent approximation without needing programmer interventions or changes in the application. We also explored an annotation scheme where the programmer could ask for error-free dynamic allocation in the environment that approximates all dynamically allocated data by default.

The error-free allocation involves another reference to the program break, thus the application has two physical heap regions, one in the approximate and error-exposed memory and another with reliable and error-free storage. This allocation may cause fragmentation of the memory data which impacts the access delay and memory performance. Further implementations of this method are required to ascertain these impacts and the actual effect on memory storage.

The transparent approximate heap is a transparent approximation that decreases the number of invalid results obtained in the execution with an approximate DRAM and improves the average quality for most of the evaluated applications. This makes this mechanism fit to applications that require improved execution resilience, leading to energy savings closer to annotated applications than other transparent interfaces at higher error probabilities.

Chapter 4

Learning-based Configuration of Approximate Memories

The proposed interfaces for data protection isolate from error common critical data for many types of applications to improve execution resilience and increase the average quality of the results. However, the energy savings or performance improvements from memory approximation depend also on how much error is allowed for each application due to the configuration of the approximation knob having a relation with the benefits provided. The errors from approximate memories typically are nondeterministic, thus a single execution does not represent the predictable behavior of an application even with the same input. Our protection interfaces require several iterations over multiple configurations to find the approximation level with the maximum benefits for an application respecting the quality threshold. Nonetheless, these iterations add a considerable overhead to configure an unknown application at runtime.

The error tolerance depends on how the application manipulates data and how the error affects quality in different degrees. An approximation level is configured by the relation between the approximation knob, the value of the operating parameter, and the amount of error it triggers. This relation depends on an error scenario that represents environment variables, such as fabrication process, locality, temperature, and access delay.

A specific output quality requirement determines a threshold of error that, if surpassed, may result in wasted computing efforts. The best approximation for an application is, therefore, the one that has the highest energy or performance benefits and respects this threshold. Previous proposals of configuration interfaces [78, 112, 130] determine the approximation level of an application by performing several executions among approximation levels or measuring the error impact on the results through quality metrics. These metrics are application-specific and require some domain knowledge. Furthermore, the approximation level may change dynamically according to the error scenario of the hardware components.

We propose `SmartApprox`, a framework for determining approximation levels based on previous knowledge of application features and error tolerance. `SmartApprox` builds a knowledge base in a training phase that analyzes a series of measurable features of applications in the face of a configuration of approximation levels to fulfill quality specifications. These application features are measurable execution statistics from general-purpose architecture counters that can be collected through a single accurate execution. At runtime, features are extracted from the input application to determine the approximation level for the same configuration that built the previous knowledge. The knowledge base evaluates the impact of errors according to a correlation between application features and the approximation level on the current error scenario of the environment. Thus, no quality specifications or metrics are required for the input application, and its approximation level is determined based on the error tolerance of the training applications and the variables that influence the error rate. Our main contributions, built upon previous work [30], are:

- A framework for determining the configuration of an approximate memory compatible with the error tolerance of the application without requiring annotations or domain knowledge from the programmer;
- A study of classes of application features and their impact on the execution with approximate memories;
- A runtime system that controls approximation considering different error scenarios and hardware configurations with negligible overhead on the reconfiguration;
- An evaluation featuring applications from different computing domains amongst configurations, learning models, and sets of features.

We evaluate `SmartApprox` on a simulation with a voltage-scaled Dynamic Random Access Memory (DRAM), wherein three configurations of approximation levels describe error scenarios that change dynamically according to hardware characterization and sensors. The evaluated features of applications achieve energy savings of 36% with acceptable quality degradation, depending on the application and error scenario. Our evaluation finds the sets of features that best correlate with application resilience under different error scenarios, which score 97% of an exhaustive search for ideal configurations, with significantly lower effort.

4.1 SmartApprox Design

`SmartApprox` is a framework that determines approximation levels for an application, at runtime, based on previous knowledge that correlates execution features and error tolerance. Figure 4.1 shows the system overview, in which a training phase builds the knowledge base for a specific configuration and set of features, and a runtime system uses the same features and a learning model to determine the approximation level. In the training phase, a representative set of applications is executed several times over the approximation levels of each error scenario. When an unknown application is given as input to `SmartApprox`, it is executed in an error-free mode to collect the features and compare them with the same attributes from the previous knowledge. According to the error tolerance of these applications, our framework associates the binary of the input application to an approximation level on each error scenario. The error scenario is detected at runtime, and the environment is configured according to the corresponding approximation level of the input application.

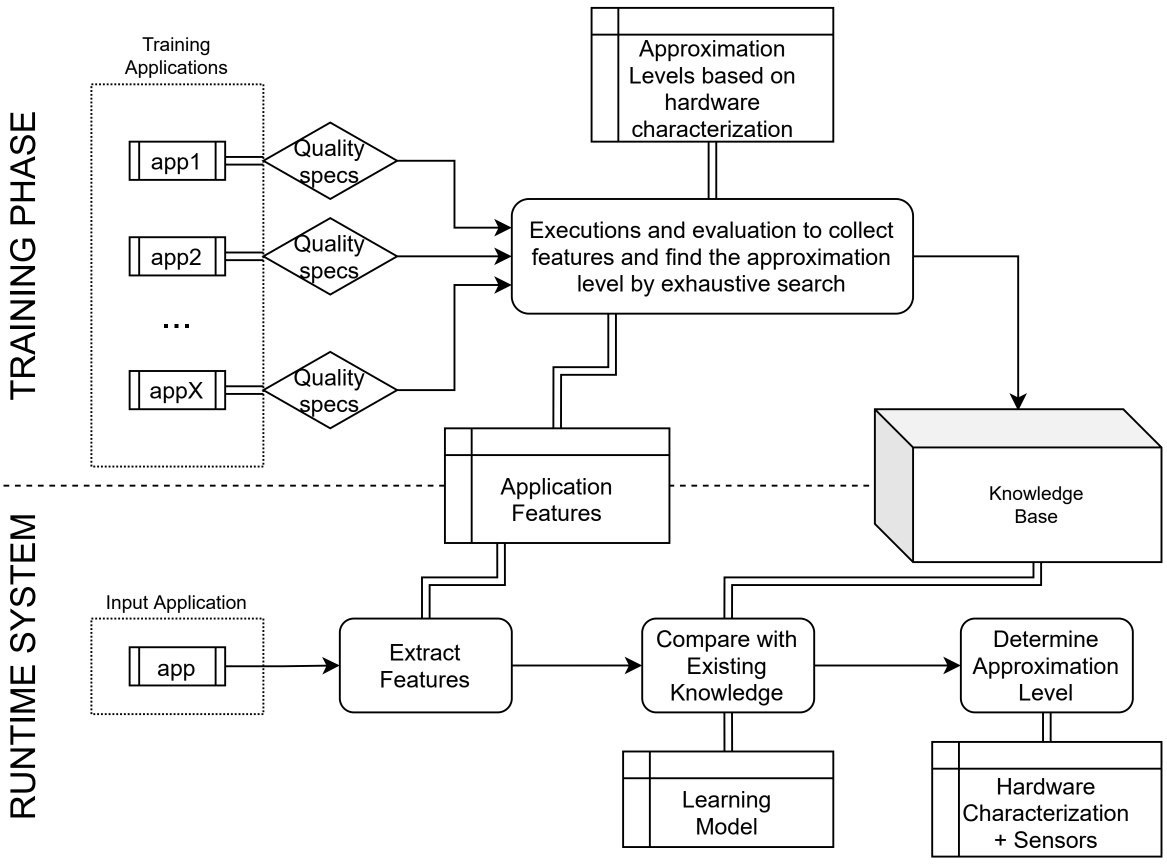

TRAINING PHASE
Training Applications
app1
app2
...
appX
Quality specs
Quality specs
Quality specs
Approximation Levels based on hardware characterization
Executions and evaluation to collect features and find the approximation level by exhaustive search
Application Features
Knowledge Base
RUNTIME SYSTEM
Input Application
app
Extract Features
Compare with Existing Knowledge
Determine Approximation Level
Learning Model
Hardware Characterization + Sensors

Figure 4.1: SmartApprox overview to build a knowledge base and determine the approximation level for an input application. *No quality metrics are required at runtime.*

The error tolerance of an application depends on the execution behavior at an approximation level. Higher tolerance implies higher energy savings with acceptable results. An approximation level contains the configuration of the approximation knob for an error rate, while the quality evaluation depends on each application and its data manipulation. An application that compresses audio, image, or video, for example, can measure the quality loss as the error in the resulting signal, while an application that searches for paths can measure quality by counting the correct paths in the resulting array. Thus, the quality metric is domain-specific and requires intervention based on application knowledge. However, features of applications may indicate how the data is manipulated, to what extent it is exposed to error, and other properties of the execution, thus suggesting how much error the application tolerates.

`SmartApprox` determines the approximation levels based on the quality metrics of training applications. Thus, our interface does not require domain-specific metrics for the input application to infer its error tolerance. The training phase performs an exhaustive search to find the approximation level towards the acceptable quality of the training applications. `SmartApprox` acts at runtime based on previous knowledge built offline.

4.1.1 Building the Knowledge Base

The Knowledge Base of `SmartApprox` is a collection of results obtained by the execution of applications in the training phase. This phase requires training applications and their correspondent quality metrics and requirements, besides a hardware configuration that represents the supported approximation levels and a set of application features to be extracted from an accurate execution. All applications are executed several times in each of the configured approximation levels to profile their behavior, since different executions may have different results in nondeterministic approximation techniques.

The behavior of each application at an approximation level is given by the quality of its results in the level. A single execution at an error-free level is performed to collect the features of the applications to compare these values at runtime. Then, the Knowledge Base accumulates all the collected information and is used in the runtime system to support the prediction of the appropriate approximation level for a target application.

4.1.2 Hardware Characterization

The configuration of approximation levels depends on hardware characterization to find the relation between parameters used as approximation knob and the probability of an error. This relation may include other variables in addition to the approximation knob, such as temperature, process variation, and environmental parameters. For instance, the same voltage adjustment in two different DRAM chips may lead to different error probabilities because of process variations [13], and the same timing parameters may cause different error configurations depending on temperature changes [71].

Approximation levels refer directly to the tradeoff between the energy gains and the error that the application data is exposed to. The supported approximation levels should cover the possible dynamic changes in the source of error or the approximation knob by sensors and hardware attributes. Thus, the training phase considers the characterization of the hardware to profile behavior in different error scenarios. Each error scenario is determined by ranges in variables that can be detected at runtime.

In the runtime system, the hardware characterization and sensors detect the dynamic changes in variables that affect the error rate. For instance, consider that two error scenarios are determined, lower error rates with temperatures below 30°C and higher error rates otherwise. When a temperature sensor detects that the environment surpasses 30°C, the approximation knob should be adjusted according to the scenario with higher error rates. Therefore, `SmartApprox` can adjust the configuration to the corresponding error scenario without the overhead of profiling or re-executing the application.

4.1.3 Features of Applications

While environment and hardware characteristics influence how much error is generated, characteristics of applications influence how much error is allowed. `SmartApprox` extracts features of applications, which are measurable characteristics that represent how the hardware is used, data structures, and the behavior on the execution of an application. The quality of the output depends on the data manipulation and the execution behavior since the output represents how the application handles the data exposed to errors.

The application features of `SmartApprox` are numerical parameters that translate the execution behavior to support the decision of the approximation level for each application. Features of different classes and sources can be applied depending on the approximation technique and error scenario. For instance, energy statistics from memory usage may guide to a different configuration than the executed instructions of the application.

4.1.4 Learning Model and Error Impact

Changing parameters of approximate memories infers a nondeterministic perturbation in the application data [80]. The perturbation outcome problem consists of guaranteeing a limit on the impact in the output and energy savings, automatically, for any general-purpose application [115]. To assure the impact with probabilistic errors, approximate systems should control the error rate that application data are exposed to. Nondeterministic perturbation models jeopardize the prediction of the error consequences since executions with the same input could behave differently. Once an error is introduced in the application data and the execution flow manipulates these data, that may have consequences on the output and in the execution flow itself. Thus, a single execution does not determine the behavior of an application in a nondeterministic approximate system, however, the behavior of a new application may be predicted through several executions of other applications.

The behavior of the training applications is summarized in the features stored in the knowledge base that is visible to the runtime system. At runtime, the same features are collected from the input application through a single accurate execution. The approximation level for each application is determined relating these features and the information available on the knowledge base. To find this relation, a learning model is needed to control and decide based on the previous knowledge.

A learning model can act as a classifier or regressor. A classifier treats the information about the approximation level as a label and aims to find a similarity between data from applications of the knowledge base and the input application. The best approximation level from the most similar application is predicted to the input. A regressor maps a function between numeric features and the configured approximation levels. The information of the knowledge base can be mitigated by single or multiple regression. A single regression tries to map a function to predict the best approximation level, thus the numeric value of the approximation knob is related to the extracted features. A multiple regression tries to map a function to predict the achieved quality at each approximation level to the extracted features, thus the expected quality is based on the metric of the training applications, and the same quality requirement is applied. The chosen approximation level is the one characterized by the maximum error rate that meets the acceptability criteria of the results based on a quality metric threshold.

4.2 Implementation

`SmartApprox` is an interface for determining levels in approximate memory systems that exposes the application stack to nondeterministic and probabilistic errors. The implementation of our proposal involves a software control and an approximation knob that modifies hardware parameters. The software control is responsible for building the knowledge base at the training phase and determining the approximation level at the runtime system. The hardware modifications should comprise sensors for detecting the current error scenario, management for the knobs in the approximate memory, and counters for extracting the features. Some sensors, such as temperature, are present in commodity hardware and can be used in the runtime system of `SmartApprox`, as well as some performance counters used as application features.

4.2.1 Classes of Application Features

Some features have characteristics in common, as referring to the same aspect of computation (e.g., executed instructions or memory accesses). To group common characteristics and measure their influence on the impact of the approximation, we divide features into classes. Each class has a different impact on the error tolerance depending on the approximation technique. We propose eight classes of application features on `SmartApprox`:

Executed Instructions (EI): Instructions describe the execution behavior and how the application manipulates the data exposed to errors. This manipulation can detail the error impact in the execution flow or the output. For instance, a `jump-to-register` instruction may deviate the control flow to an incorrect address, leading to execution crashes [26]. Although instructions depend on the architecture, a general classification can represent common commands. This classification groups similar instructions according to their behavior, such as memory, floating-point, control flow, and other instructions that expose execution to crashes or manipulate data exposed to errors.

Memory Instructions (MI): Instructions that directly refer to memory accesses model the insertion of errors into the execution. Mainly, `load` and `store` instructions represent the memory usage behavior of the application. However, other instructions may affect memory accessing and should be taken into account, such as `fence` instructions that impose memory access barriers.

Memory Accesses (MA): While instructions show how an application behaves and manipulate data exposed to errors, memory accesses are the only source of the actual errors. Even hold errors are only visible to the application after reading the affected data. Accesses on all levels of the memory hierarchy are relevant to measure the impact in the output, even when only one level of the memory hierarchy is approximated since each level alleviates the number of accesses to the next one [29]. Moreover, memories may be divided into reliable and approximate regions [23] and the proportion of accesses in each region can show the error avoidance or the potential energy improvement.

Approximate Memory Accesses (AMA): Features of accesses performed only at the approximate part of memory describe the number of accesses exposed to errors in the execution flow. This class also contains the fraction of approximate accesses on memory, which expresses how many accesses need to be performed to expose data to errors.

Cache Efficiency (CE): Even when the main memory is subjected to errors, caches perform an important role in alleviating the impact of errors [29]. The hit ratio refers to the efficiency of an error-free cache level on avoiding accesses to the approximate main memory. Each access to the main memory exposes data to errors, thus the cache efficiency is also a metric of error protection at which higher efficiency decreases the impact on the execution results. That is, an application that has more effective use of cache has also a lower impact of errors on its data and produces higher quality results.

Memory Energy Breakdown (MEB): The energy spent at each memory mode of operation indicates memory usage patterns. For instance, an execution that spends more dynamic than static energy has relatively more memory read and write operations during the data lifetime. Furthermore, these data show how intensive is the use of memory throughout the application execution. Other features on the energy breakdown that refer to the memory usage patterns are the average power, the total energy at a given time slot, and the energy spent on some memory operations, such as DRAM refreshes or row activations.

Data Size (DS): The memory architecture specifies limits on the amount of data that can be transferred. For example, the size of the row buffer limits the data that can be read or written at once in a DRAM bank. The size of the data manipulated by the application influences directly the number of memory accesses. Furthermore, these features have a direct influence on cache efficiency, the number of instructions, and energy consumption. Therefore, bytes read or written in the memory hierarchy are relevant to show usage patterns of the approximate memory and application behavior.

Memory Controller Commands (MCC): Main memory systems usually have a controller that manages the data access and maximizes the data flow. This management is performed through commands that depend on memory technology. For instance, a DRAM controller uses commands of row activation (ACT), precharge (PRE), read (RD), write (WR), and refresh (REF). These commands are related to the memory usage by the application but include technology-dependent details that are not described by the application-level features.

4.2.2 Runtime System

The runtime system determines the current configuration for an input application. This configuration may change dynamically according to sensors and hardware characterization. The features collected from the input applications are used in the configurations amongst all supported error scenarios in the lifetime of the current execution of the application. These configurations are stored into a list to be applied at runtime without new executions. Therefore, the first workload should be representative of the context of the following executions of the same application.

Algorithm 4.1 exhibits the implementation of the runtime system of `SmartApprox`, where a single accurate execution of the input application collects the features to apply the learning model and predict its appropriate approximation level. If the application is known, the global variable *Levels* contains the determined level for the application and

Algorithm 4.1: Execution of the runtime system of `SmartApprox`. *If the error scenario changes, a new configuration is applied with negligible overhead.*

```
Inputs
  App: an application and a representative input.
  KB: the knowledge base built at the training phase.
  LM: a learning model.
  Fts: a set of application features.
  HWCS: hardware characterization and sensors that identify the current error scenario.
Result: An approximation level.

SmartApprox_Runtime(App, KB, LM, Fts, HWCS):
    if not(KnownApps contains App) then
        execute(App, ErrorFree) $\rightarrow$ output, stats;
        ftValues $\leftarrow$ extract(stats, Fts);
        appLevels $\leftarrow \emptyset$;
        foreach errScn in scenarios(KB) do
            approxLvl $\leftarrow$ LM(KB, errScn, ftValues);
            config $\leftarrow$ < errScn, approxLvl >;
            append config in appLevels;
        end
        append App in KnownApps;
        append < App, appLevels> in Levels;
    end
    errScn $\leftarrow$ getCurrentScenario(HWCS);
    level $\leftarrow$ Levels[App][errScn];
    return level;
```

there is no need for another collection of features. On the search for the approximation level for a new application, all error scenarios considered in the knowledge base are taken into account and different levels can be associated with each scenario. After finding the approximation level for all scenarios, the current error scenario of the system is detected by reading hardware sensors or by characterization of the components, and the determined approximation level for this scenario is returned. Therefore, `SmartApprox` determines the approximation level indexed by the application according to the error scenario with negligible overhead.

4.3 Methodology

To demonstrate `SmartApprox`, we simulate an environment with the training phase and runtime system. To represent the hardware, we model data errors as high-level software abstractions. Our simulation replaces accesses on the main memory with these models, where the data word is exposed to a random bitflip, at the error rate observed by the knob adjustment in different error scenarios. These errors are persisted in the memory and may occur at any bit of the data word. The simulation environment is the same as Section 3.3.3, except for the filter of privileged accesses. In this evaluation, we consider a protection interface that implements mechanisms from `AxRAM`, where all accesses into the data region are exposed to errors, except for the stack addresses.

4.3.1 Approximation and Energy Models

The approximation based on the `AxRAM` interface [26], which acts transparently to define the protected data from the application and has no annotations requirements. To account for the energy of protected regions, the fraction of accurate accesses relies on their proportional energy at the nominal memory specification. The approximation technique used in our evaluation is based on the voltage scaling of DRAM, where the Vdd is lowered to values that expose all nonprotected data to errors. Although errors can be dependent on data patterns or spatial distribution, a uniform bitflip model sufficiently represents the error models for content and region dependency [61].

We limit the approximation technique to reducing the supply voltage of the DRAM due to the potential of reducing power across different memory operations. To account for energy, we consider the exponential relation between the supply voltage and the error rate to model exponential regressions based on data from Chang *et al.* [13]. These data were collected from chips of a unique vendor (specified as "Vendor B") with fixed temperature and delay parameters, thus the error fluctuation represents the process variation among memory chips. Exponential regressions were formulated based on these data to instantiate error scenarios. To comprise aspects of process variability, three scenarios were considered, based on the minimum, median, and maximum error probability points of the collected data, named here as best, median, and worst error scenarios. The exponential regressions are limited between 0 (error-free) and 1 (always injects error) and are expressed in the form described by Equation 3.1, wherein A values are 5.15×10^{78}, 1.80×10^{68}, and 3.55×10^{59}, and B values are -175.47, -155.87, and -142.63 for worst, median, and best scenarios, respectively. Figure 4.2 shows regressions for each considered scenario, where the R^2 is 95% for the worst, 99% for the median, and 77% for the best error scenarios.

Since the relation between error and voltage is exponential, a small voltage difference reflects a significant difference in error. Furthermore, voltage as an approximation knob is limited by the physical capabilities and variability of regulators and transistors. To account for these variables, we consider 10 approximation levels on each error scenario, from 1.02 to 1.11 V, at a 10 mV step. Thus, 30 error rates were considered in our evaluation.

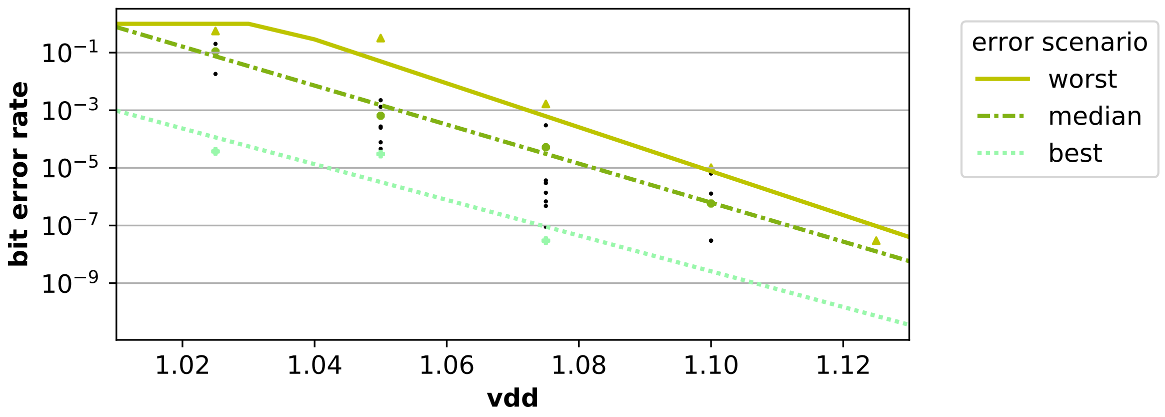


Figure 4.2: The error scenarios of our evaluation. *The error probability of each voltage level is obtained according to the current scenario characterized on the hardware.*

4.3.2 Applications and Quality Metrics

We select applications from different computing domains from the benchmark sets described in the AxBench [131], Benchmarks Game [44], CBench [38], Mibench [45], and Polybench [88]. The evaluated applications represent the expected general behavior of applications that tolerate error in their results (e.g., manipulating images, floating-points, and data statistics) [128]. The average execution time of each application is approximately 55 seconds in our environment. Table 4.1 shows the complete list of evaluated applications, their input workload, and quality metrics. The applications are compiled with GCC/G++ 9.2.0 for RISCV-V from riscv-gnu-toolchain with flags -O3 and -static. These quality metrics are detailed in Section 2.1.4.

To evaluate the learning model and features, we build a test methodology using a knowledge base containing data from 26 applications. The applications are divided into test and training. Features and quality measures from training applications abstract the behavior of different classes of programs to compose the knowledge base. The evaluated applications build a relatively small dataset to abstract the several classes of the test applications. We therefore selected 7 of the 26 applications to be employed in the test.

Table 4.1: Applications used in our evaluation. *Some applications could be not well-resilient but have representative behavior in the approximate environment.*

Application	Workload	Quality metric
2mm	32x32 double	MAPE
atax	500x500 double	FEE
blackscholes	1.000 entries	MAPE
bunzip2	256x256 image	SSI
bzip2	256x256 image	SSI
correlation	64x64 int	FEE
covariance	32x32 double	FEE
dijkstra	complete graph, V=60	FEE
fannkuch-redux	N=9	MAPE
fasta	N=50.000	FEE
fft	8 waves, size 1024	MAPE
floyd-warshall	complete graph, V=60	FEE
inversek2j	1.000 coordinates	MAPE
jacobi-2d-imper	2x 32 double	FEE
jmeint	1.000 coordinates	FEE
jpeg	512x512 image	SSI
k-nucleotide	2x 50.000 nucleotides	MAPE
mandelbrot	N=500	SSI
mm	100x100 int	MAPE
nbody	N=100.000	MAPE
pi	N=1.000.000	MAPE
qsort	10.000 strings	FEE
reg_detect	10.000 iterations	MAPE
reversecomplement	2x 50.000 nucleotides	FEE
sobel	256x256 image	SSI
spectralnorm	N=500	MAPE

For each experiment, we remove from the training set only the data from its respective test application, similarly to other methods for cross-validation used with a small number of instances in a dataset [127]. The selected applications for testing are blackscholes, dijkstra, k-nucleotide, mandelbrot, jpeg, pi, and sobel. The results are obtained by the average of the results of 7 evaluations with each one of the test applications out of the training dataset that has data from the 25 remaining applications.

The exhaustive search to the best approximation level iterates over 100 executions of each application at each approximation level. In the presence of nondeterministic errors, these executions are necessary to measure the impact of the probabilistic errors in the execution output. Therefore, the process of building the knowledge base comprised 100 individual executions of each of the 26 evaluated applications, at each of the 10 approximation levels, for each of the 3 energy scenarios. With approximately 55 seconds of execution per application, an error scenario with 10 approximation levels takes more than 15 hours of iterative executions to configure a single application through the exhaustive search.

The error tolerance depends on each application. However, since `SmartApprox` is interested in the behavior of each application when executing in the approximate environment, we have to define a requirement of acceptability for the execution results. Thus, the quality requirement was defined as 90%, which leads to a reasonable quality of the output when considering the number of incorrect elements, relative error margin, and acceptable image noise.

4.3.3 Features and Learning Models

Table 4.2 shows the features of our evaluation following each class from Section 4.2.1. Our instructions classification considers the RISC-V ISA specification [124] and possible hardware counters that produce data about the execution. The DRAM commands are generated by Ramulator [59] and the energy data are provided by DRAMPower [11]. All values are normalized to the highest value between all training applications.

The features are used to learn the application resilience. The evaluated learning models are the Nearest Neighbor Algorithm (NNA), Neural Network (NN), Random Forest (RF), and Support Vector Machine (SVM). NNA is a classifier that calculates the euclidean distance between the values of features from the input and the knowledge base, where the approximation level is taken from the best level of the application with the lower distance to the input application. SVM is a classifier with the Radial Basis Function (RBF) kernel to predict the approximation levels. NN and RF are considered as classifiers, single and multiple regressors. NN uses the Multi-Layer Perceptron (MLP) implementation with 5,000 maximum iterations, where all the evaluated data has been converged, and the solver kernel of the Broyden–Fletcher–Goldfarb–Shanno algorithm (BFGS) with an alpha of 1e-5 and hidden layers sizes of (5, 2). RF constructs decision trees based on the application features, with a maximum depth of 2, to predict labels or function values, depending on the learning type. A random choice is also taken into consideration as a reference, choosing N random applications of the knowledge base and using their best approximation level.

Table 4.2: Classes of applications features of our evaluation. *Each feature may belong to different classes.*

Class	Features
AMA	Approximate DRAM accesses
	Fraction of approx. accesses on DRAM
CE	IC miss rate
	L1 miss rate
	L2 miss rate
DS	IC bytes read
	IC bytes written
	L1 bytes read
	L1 bytes written
	L2 bytes read
	L2 bytes written
MCC	ACT commands
	Auto-refresh cycles
	PRE commands
	RD commands
	REF commands
	Total trace length (clock cycles)
	WR commands
MEB	Active idle energy
	Auto-refresh energy
	Average power
	Total idle energy
EI	Number of barrier instructions
	Number of control flow instructions
	Number of floating-point instructions
	Number of jump-to-register instructions
	Number of load instructions
	Number of store instructions
	Sum of loads and stores
	Total number of instructions
MA	Approximate DRAM accesses
	DRAM accesses
	IC accesses
	L1 read accesses
	L1 write accesses
	L2 read accesses
	L2 write accesses
	L2 writebacks
MI	Number of barrier instructions
	Number of load instructions
	Number of store instructions
	Sum of loads and stores

4.3.4 Oracle and Evaluation Metrics

The best approximation level for a test application is determined through an exhaustive search among the configured levels for each of the training applications. Therefore, we define an oracle choice that indicates the level that has the minimum energy consumption that fits the quality requirement for each application. This operating point may not be the one chosen by `SmartApprox`, however, causing losses on energy savings or quality. Our evaluation intends to find the best learning models and features set, thus we analyze each tuple <learning model, features> indicating the approximation level for the input application. To account for the energy score of this tuple, we consider:

$$E_{score}(T) = \begin{cases} \frac{E(oracle)}{E(level)}, & \text{if } E(level) > E(oracle) \\ 1.0, & \text{otherwise} \end{cases} \tag{4.1}$$

, where $E(x)$ is the relative energy consumption, compared to nominal, on the approximation level x, and *oracle* and *level* indicate the operating points determined by the oracle and the used tuple T of <learning model, features>, respectively.

The quality score follows the same logic when the average quality on the operating point determined by the tuple is above those on the level determined by the oracle. Thus, the quality score is given by:

$$Q_{score}(T) = \begin{cases} \frac{Q(level)}{Q(oracle)}, & \text{if } Q(level) < Q(oracle) \\ 1.0, & \text{otherwise} \end{cases} \tag{4.2}$$

, where $Q(x)$ is the average quality achieved on the operating point x. To evaluate the tradeoff between energy and quality, the final score is given by:

$$Score(T) = Q_{score}(T) \times E_{score}(T) \tag{4.3}$$

When the application is executed at a higher approximation level than the oracle, in general, it consumes less energy but outputs lower than the required quality. Higher energy consumption is achieved when the application is executed in a lower approximation level than the oracle, in general achieving higher average quality. Concomitantly lower quality and higher energy are possible in variations that account for different execution crashes (e.g., indefinite loops execution caused by errors).

4.3.5 Search for the Best Features

`SmartApprox` has a set of application features as a parameter, which is considered in the learning model to determine the approximation level of an input application. We propose several classes of features that influence the error tolerance of applications. Nonetheless, each subset guides to different results that depend on several factors, such as the error scenario and training applications. Furthermore, features from different classes can be combined to generate the best possible subset. The search space for this set is combinatorial regarding the number of application features (e.g., with 37 features, we have more than 10^{11} subsets), which makes an exhaustive search unfeasible. Therefore, we perform this search through a Genetic Algorithm (GA), a metaheuristic that reduces the search space stochastically by iterating over crossover operations.

We execute this search in the three error scenarios considering as fitness function the Score metric (Equation 4.3) that comprises aspects of energy and quality. This search encodes the optimization problem in variable-length chromosomes, each representing a subset of the features listed in Table 2. The initial population includes all classes that score more than 90% in the fitness function in Section 4.5 and random individuals. For example, considering class CE, the chromosome [“IC miss rate”, “L1 miss rate”, “L2 miss rate”] is a valid member of the initial population. The initial population excludes all features from classes that did not achieve the 90% score and were not randomly selected, thus reducing the solutions generated through the crossover operations. Our mutation operator randomly chooses one of (1) removing, (2) adding, or (3) replacing a feature from the subset. The mutation probability is calibrated to 5% on non-stagnated generations and grows 20% after each stagnation to avoid iterating only over similar solutions in the population. The metaheuristic stops after 5,000 generations after producing at least 200,000 subsets or by stagnating the population on 250 generations, evaluating at least 0.5% of the total subsets produced by the GA. These numeric parameters were determined by iteratively analyzing the progress of the GA and the scores of the solutions.

4.4 Evaluation

In this section, we present our evaluation results with a voltage-scaled DRAM. First of all, we show our results for training applications on three error scenarios and the oracle that determines the approximation levels on our knowledge base. Further, the learning models are evaluated with the sets of features listed in Table 4.2, and the results are detailed for 7 selected test applications. Lastly, an evaluation of the application features shows the most relevant in the three scenarios for all applications in the runtime system.

4.4.1 Oracle and Knowledge Base

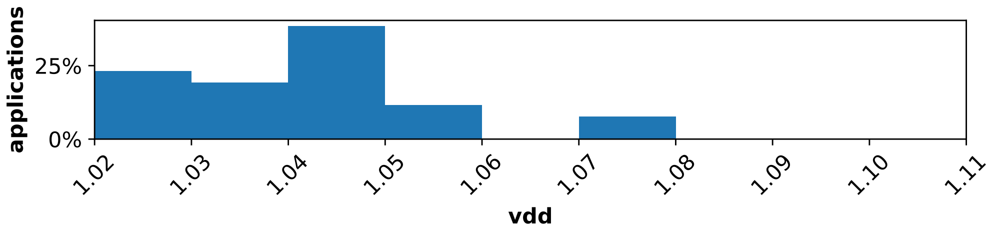
Our knowledge base is built through multiple executions of applications at each configured approximation level. A quality metric is taken into consideration for the training applications, and the average quality is calculated at each approximation level. The level with the maximum energy savings that satisfy the quality requirement for an application represents its optimal configuration and the best approximation level. Thus, an optimal configuration is associated with the application features in the knowledge base. A good knowledge base should cover different error tolerances, so the applications should be distributed among the possible approximation levels. We consider an oracle that performs an exhaustive search among configurations to evaluate the disposition of the best approximation levels at each energy scenario. More details of the oracle and evaluation metrics are available in Section 4.3.4.

Figure 4.3 shows the distribution of the best approximation levels to achieve 90% quality across all evaluated applications, for each error scenario, where a higher approximation (lower voltage) means higher energy savings. This plot indicates the best approximation level, thus levels that have a higher energy cost (e.g., higher voltage) also fulfill the quality requirements. Each scenario determines different error rates for each Vdd and affects the distribution of applications. As the error scenario tends to introduce more errors, the distribution of the applications is prone to higher Vdds and lower approximation levels, as shown on the differences between best, median, and worst scenarios.

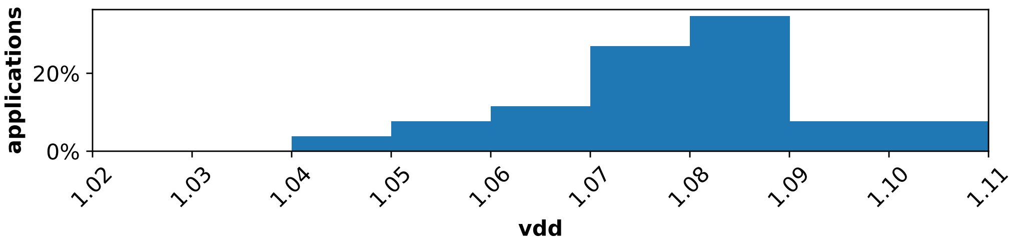
Applications that are less memory-intensive tend to perform better at lower Vdds. Examples of these applications in our training set are spectralnorm, pi, nbody, and mandelbrot. Applications that manipulate sensitive data, such as bzip2, bunzip2, and dijkstra, tolerate fewer errors and thus have their best approximation level at higher Vdds.

On the best error scenario, the best approximation levels tend to be distributed between 1.02 V and 1.05 V with some deviation at 1.07 V. The peak of this distribution is at 1.04 V, which is the best approximation level for atax, blackscholes, covariance, inversek2j, jacobi-2d-imper, k-nucleotide, mm, qsort, and sobel. In the best error scenario, the energy savings are 31% on average.

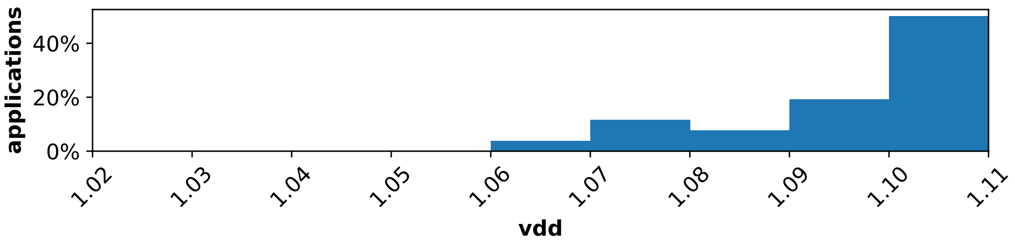
Considering the median error scenario, most applications are spread in distribution with peaks concentrating between 1.06 V and 1.08 V. Below 1.03 V, none of the applications achieved average quality higher than 90%. Spectralnorm is the only application that tolerates 1.04 V at the median scenario, but its energy gains are limited due to its reduced number of memory accesses. For instance, mm achieves energy savings of 36%



(a) best-case error scenario



(b) median error scenario



(c) worst-case error scenario

Figure 4.3: Frequencies of best approximation level to achieve 90% quality requirement throughout applications. *When the error scenario worsens, applications tend to tolerate lower approximation levels with higher Vdd.*

at 1.08 V, while spectralnorm saves 28% at 1.04 V. The energy savings for the median scenario are 29% on average.

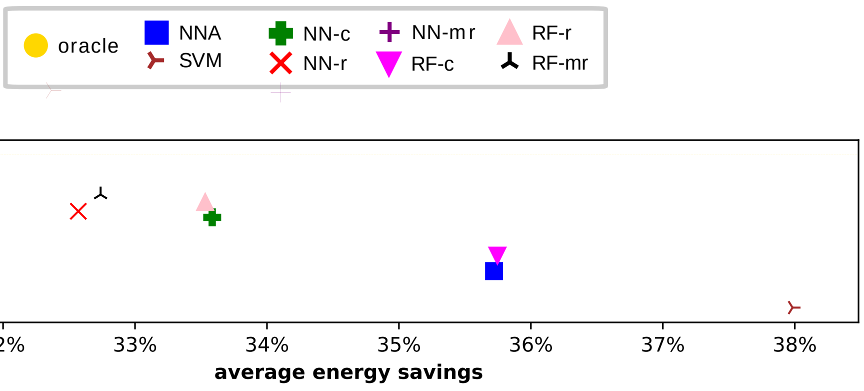
In the worst error scenario, the distribution of best approximation levels is accumulated in the higher Vdds, where half of the applications tolerate only 1.11 V to achieve 90% quality. None of the applications tolerate less than 1.05 V. The average energy savings in the worst scenario is 28%.

In each of the error scenarios, we built a knowledge base with different distributions of the best approximation level of the applications. Thus, an evaluation of a different error scenario also evaluates other distributions of the error tolerance and different energy savings for all applications.

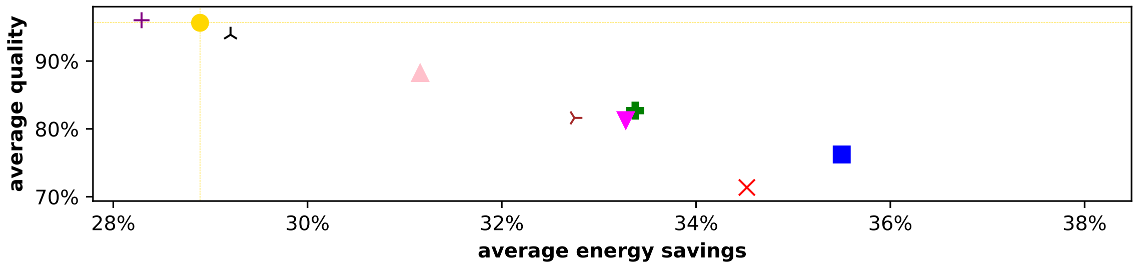
4.4.2 Learning Model

Determining an approximation level for an application may result in energy savings and quality depreciation. Our evaluation oracle represents the level with the maximum energy savings that meet the required quality. However, `SmartApprox` uses a learning model that can determine a different approximation level than the oracle, increasing energy or losing quality. In this section, we evaluate and compare the learning models and the random choice, considering energy and quality and the aggregated Score metric from Section 4.3.4.

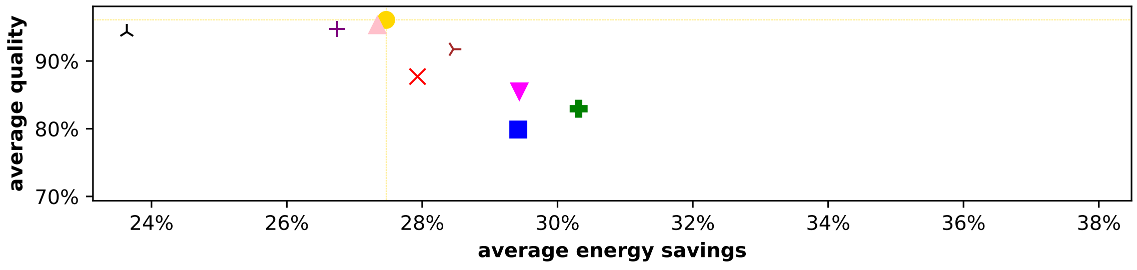
Figure 4.4 shows the average quality and energy savings that each learning model achieves for all test applications. NNA and SVM are classifiers, while NN and RF are also regressors with single and multiple predictions. The lines crossing the oracle mark the benefits on energy savings and quality on axes X and Y, respectively. In general, higher energy savings means lower quality, which has some points of deviation on the average quality for each learning model. The oracle represents the maximum efficiency, thus no point achieves higher quality and savings. In the three error scenarios, the multiple regressors of NN and RF resemble the oracle better than the other models. The single regressor RF achieves results closer to the oracle in the worst-case scenario but has inferior results on best and median error scenarios.



(a) best error scenario



(b) median error scenario



(c) worst error scenario

Figure 4.4: Average quality and energy savings of learning models in the three error scenarios. *NN and RF were applied as classifiers (-c), single regressors (-r), and multiple regressors (-mr). In general, higher savings means lower quality.*

Table 4.3: Average score of learning models for the best, median, and worst error scenarios. *The learning models with multiple regressors achieve the best average scores.*

		average score		
type	model	best	median	worst
classifier	NNA	81%	77%	76%
	NN	91%	86%	83%
	RF	86%	85%	87%
	SVM	78%	85%	94%
single regr.	NN	91%	73%	87%
	RF	91%	90%	95%
multiple regr.	NN	91%	95%	91%
	RF	**93**%	**96**%	**97**%
random		81%	74%	77%

Table 4.3 shows the average score of the evaluated learning models in each of the error scenarios. The random choice shows that the achieved average score depends on the error scenario, where the chances for a better score are improved according to lower error rates per approximation level. The best error scenario has more chances to waste energy than depreciate quality. Considering that a single step on voltage may affect quality more than energy, a learning model that applies conservative approximation levels tends to achieve a higher average score.

Classifiers treat the approximation knob as a label and do not consider its numeric relation to the application features. The average score of NNA is similar to a random choice in the three evaluated scenarios, thus a relation of this type is the least representative. SVM is surpassed by random on the best error scenario but achieves better results as the error rate of the approximation levels gets worse. NN and RF classifiers can achieve better results than a random choice but have inferior performance in terms of the score for most scenarios.

Single regressors evaluate the relation between the numeric value of the approximation knob and the application features, while multiple regressors explore the relation between the expected quality in each approximation level with these features. The multiple regressors predict, then, one value per approximation level, 10 per error scenario, in our environment. Thus, in our evaluation, multiple regressors achieve better results by considering more aspects of the approximate environment. Both NN and RF present promising results with multiple regression, but only RF maintains high scores with single regression on the three evaluated error scenarios. RF with multiple regressions achieves a higher average score than all the evaluated learning models also due to a conservative determination of approximation, where it usually chooses lower energy savings in a linear magnitude over quality loss in an exponential magnitude. Therefore, in the remaining evaluations, we adopt multiple-regression RF as the learning model.

In our experiments, running the learning model to determine an approximation level for the input application takes seconds, while the simulated exhaustive search for the golden approximation level determined by the oracle takes hours. Moreover, the learning model does not require any information on how to obtain the quality of the results of the input application. Assuming characterization in real hardware, the exhaustive search still requires hundreds of executions in several knob configurations for reliable and statistically significant results, while the learning model can be executed offline, in a more powerful machine. These observations highlight the reduced effort of `SmartApprox` compared to determining the appropriate approximation level on a per-application basis.

4.4.3 Classes of Features

To evaluate the influence of each class of application features, we analyze them grouped by classes (Section 4.3.3). The selected test applications are different in structure and manipulate different data. Table 4.4 shows the average score of the test applications in the three error scenarios. We consider high scores when the value is higher than 90%, which means quality and energy close to the oracle.

Table 4.4: Average score of multiple-regression RF. *A set with features from a single class has quality and energy results that depend on the particularities of the input application.*

Class	avg score		
	best	median	worst
AMA	83%	88%	89%
CE	97%	97%	86%
DS	90%	**99%**	94%
MCC	83%	84%	78%
MEB	88%	95%	92%
EI	**99%**	98%	**96%**
MA	89%	**99%**	**96%**
MI	97%	98%	**96%**

The class AMA has an 87% average score. This class has not achieved high scores with dijkstra, which uses linked lists with sensitive data to reference its elements. These are not distinguished by features of approximate DRAM, thus the approximation level determined using this class depreciates quality more than tolerated, lowering the score.

CE shows a metric of error avoidance from the approximate DRAM. This class achieves a high 97% average score on the best and median error scenarios. Only for jpeg, the approximation levels result in too low quality because of the poor information about the data size provided by CE features.

The DS class achieves 99%-average scores on the median error scenario. However, in the best scenario, this class misses the error sensitivity of dijkstra, and it is too conservative in the worst scenario for k-nucleotide and sobel. The more conservative choices have less influence on the score than the quality-deprecating ones, thus the average score, considering all applications, are 90% and 94% to best and worst error scenarios, respectively.

MCC expresses application behavior on the use of memory but cannot detect the error sensitivity of dijksta. Furthermore, this class of application features determines approximation levels of low quality to jpeg and conservative levels to k-nucleotide and sobel. The scores are 83%, 84%, and 78% for the best, median, and worst scenarios, respectively.

MEB achieves average scores of 88%, 95%, and 92% for best, median, and worst error scenarios. In the median scenario, just jpeg does not have a high average score, as this class has some information about the data size but is not sufficient to detect the appropriate level. In the best error scenario, the level for the dijkstra application decreases the average score, depreciating the average quality.

EI represents the entire behavior of execution, data manipulation, and memory usage, thus achieving a 97% average score. Except for the overly conservative choice for k-nucleotide, this class of features achieves high scores for all applications.

MA represents some of the data size information, thus detecting the workload for jpeg. However, the specific needs of dijkstra were perceived only in the median and worst error scenarios, decreasing the score on the best scenario. Some energy savings are achieved in sobel application but none in k-nucleotide, where the conservative determination of the approximation level decreases the average score. The average scores are 89%, 99%, and 96% for the best, median, and worst scenarios, respectively.

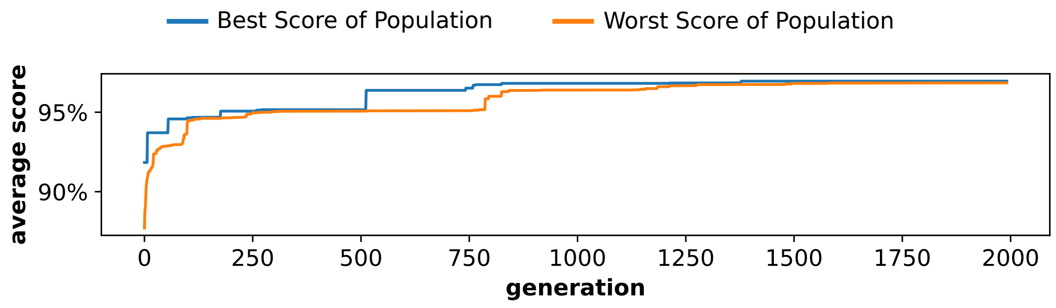
MI achieves the same average scores that EI, considering all applications and error scenarios, 97%. There is no significant difference between error scenarios, averaging 97% for best, 98% for median, and 96% for worst. Except for jpeg in the worst scenario, this class of application features achieves high scores for all applications and error scenarios.

4.4.4 Features Search

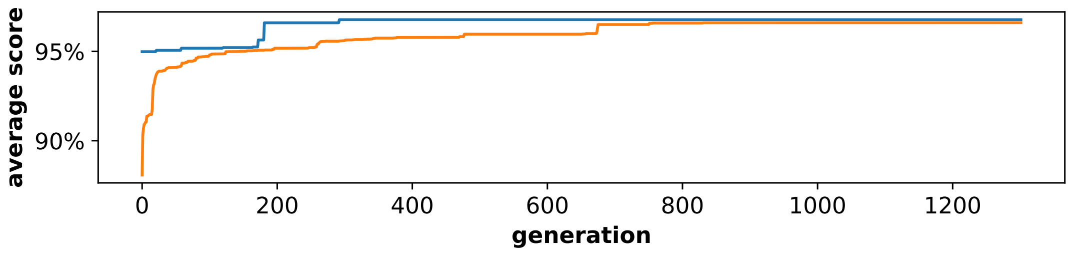
Each class of features represents a different impact on the tolerated error rate and approximation level. However, combining them can improve the sensitivity of how the approximation levels impact application structures, workloads, and behavior on approximate memories. In this section, we evaluate the determination of approximation levels for all applications, thus the learning model should indicate the appropriate approximation for the 26 evaluated applications at each iteration of the Genetic Algorithm (GA) in Section 4.3.5. The GA searches for the best combination for the evaluated applications.

We start our search with an initial population that has the features in classes CE, DS, MEB, EI, MA, and MI as individuals. These classes achieve an average score higher than 90% at all error scenarios. The population is completed with random size sets with randomly chosen features, considering all classes. Figure 4.5 shows the best and the worst scores in the population for all generations of the GA in the three evaluated error scenarios. In all scenarios, the GA stopped after the maximum stagnation. The initial population is the same for all scenarios, however with a different score since the resulting quality and energy depend on the configuration of approximation levels. Thus, the median error scenario, which has the higher score of 95.0% in the initial population, converges after 1301 generations. The worst error scenario has more limited search space than other scenarios, where 50% of applications have the same best approximation level due to the higher number of errors. Therefore, even with a higher score of 92.7% in the initial population, this scenario converges after 1347 generations. With a more complex search space, the best error scenario starts the search with a higher score of 91.8% and converges after 1991 generations.

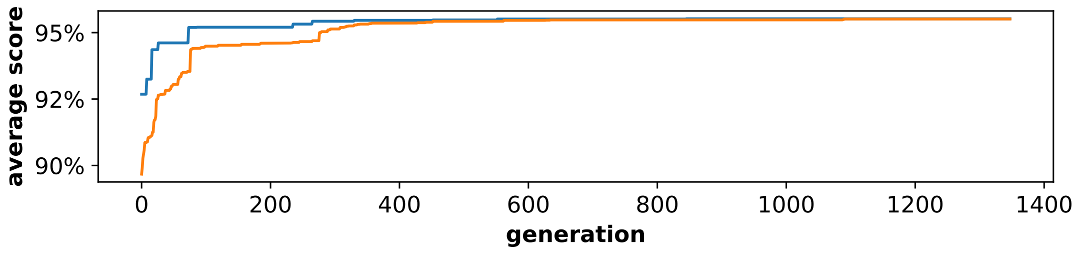
The initial population of the GA has an average score of 89.7% ($\sigma = 1.2$ p.p.), 91.5% ($\sigma = 1.8$ p.p.), and 90.9% ($\sigma = 0.92$ p.p.) for best, median, and worst error scenarios, respectively. The lower score of the population tends to grow after each generation and,



(a) best error scenario



(b) median error scenario



(c) worst error scenario

Figure 4.5: The best and the worst average score on the population of sets of features among all generations of the GA. *The best scenario has a more complex search space and lower initial scores, thus showing delayed convergence.*

according to the stagnation of the best solution, the standard deviation of the population tends to decrease. The standard deviations of the final population are below 0.05 p.p. for all error scenarios. The low standard deviation shows that the solution converges to values near the best score found, and, then, the GA stops after the maximum stagnation with average scores of 96.9%, 96.7%, and 95.5% for best, median, and worst scenarios.

The set of features that has the best average score for each error scenario is listed in Table 4.5. Features from classes MEB and EI are on sets of all scenarios, which shows how energy and instructions features affect the error tolerance of applications. Although directly related to memory use, features from class MI are not in the resulting set for any scenario.

In the best scenario, higher energy savings are achieved due to fewer errors at lower Vdds. The search space is distributed among the higher approximation levels, and the average energy savings are 30.6% with 95.8% average quality. In this scenario, most of the features on the best resulting set are related to lower levels of cache. The idle energy of DRAM is considered and the number of floating-point instructions shows some of the

Table 4.5: Best features set found for each error scenario. *In the best scenario, caches are determinant, while the median scenario considers more energy features, and the worst considers the usage among memory hierarchy.*

scenario	features	class
best	IC miss rate	CE
	L1 miss rate	CE
	L1 bytes read	DS
	Total Idle Energy	MEB
	Number of floating-point instr.	EI
median	Approximate DRAM accesses	AMA, MA
	L1 bytes written	DS
	Active idle Energy	MEB
	Auto-refresh Energy	MEB
	Total Idle Energy	MEB
	Number of floating-point instructions	EI
worst	Approximate DRAM accesses	AMA, MA
	L1 miss rate	CE
	L2 bytes read	DS
	L1 bytes written	DS
	RD commands	MCC
	Auto-Refresh Energy	MEB
	Number of floating-point instructions	EI
	IC accesses	MA
	L2 read accesses	MA

behavior of application manipulating data.

The median scenario shows an almost normal distribution of best approximation levels of applications, tending to determine higher Vdds than average. Considering all applications with the best set of features, the average energy savings is 28.0% and output quality is 95.7%. Most of the features of the set for this scenario consider the energy breakdown of DRAM, including the energy of the auto-refresh operation, which changes on different voltages. As the set of the best scenario, floating-point instructions are considered, however without the major focus on information from caches of lower levels.

In the worst error scenario, the appropriate approximation levels are concentrated in the higher Vdds to most of the applications. Therefore, lower energy savings are achieved, being the best set with 25.9% of average savings and 95.8% of average quality. In this scenario, the resulting set has a higher number of features than other scenarios, and the resulting set does not have a general reference of the memory hierarchy. Thus, to determine conservative approximation levels, `SmartApprox` considers features from L1, L2, IC, and DRAM to achieve the highest score of the worst error scenario.

The resulting sets of each error scenario evidence that aggressive approximations tend to consider more features from lower cache levels, while conservative approximations consider the usage and behavior on the entire memory hierarchy. On a likely normal distribution, as in the median scenario, the energy breakdown of DRAM operations is important to determine better approximation levels.

4.5 Discussion

`SmartApprox` determines approximation levels without requiring specific metrics or annotations in the source code. Our work proposes a generic memory interface that senses modifications in the error scenario and changes the approximation level according to application features. In our simulated environment, however, `SmartApprox` is evaluated into a limited scheme where the approximate memory is a voltage-scaled DRAM with the execution of a single application and normalized quality metrics with the same acceptability threshold for all applications.

Approximate memories achieve energy savings at the cost of error exposition of the application data. The impact of such errors depends on the data manipulation and behavior of the executed application. Furthermore, errors are tolerable only until a certain limit. In this chapter, we analyze and propose a relation between application features and error tolerance. `SmartApprox` explores different application features to determine approximation levels based on previous knowledge. To this end, we list classes of numeric features that refer to details of memory usage and instructions that could affect the error tolerance and can be extracted from a single accurate execution. We evaluate three error scenarios that vary according to dynamic components of the approximate memory. We detail how learning models can be used to determine the appropriate approximation level for a new application and perform a search for the best set of features that expresses the error tolerance. Our results show that `SmartApprox` achieves 28% average energy savings with 96% output quality in a median error scenario and the best found set of features.

Chapter 5

Conclusion

Errors in the computational results are often tolerated and sometimes the accurate results are even not possible. Approximate Computing (AC) exploits degrees of error tolerance while maintaining acceptable results. However, applications exposed to errors may have unexpected behavior or trespass a limit of the tolerated inaccuracy when executing in approximate environments. Thus, interfaces are required to control the error and maintain the quality of the results and the energy gains. In this work, we proposed interfaces that explore approximations at the memory level and raise the likelihood of tolerated inaccuracy to achieve energy gains. `AxRAM` is an interface with data protection and recovery, besides some mechanisms explored in this work being independent of this interface. `SmartApprox` is an interface to configure approximate memories that infers the adjustment of the approximation knob according to an input application. The protection and recovery mechanisms act transparently without requiring approximation-specific changes in the applications. Nonetheless, these interfaces assume some controls in software and changes in the hardware to perform. In this chapter, we discuss the implications of such control and the applicability of our interfaces, listing requirements, limitations, and the future directions of this research.

5.1 Discussion

In our evaluation scenario, we target applications or kernels that repeatedly execute over many different inputs. Therefore, it is possible to recover from lower than tolerable quality outputs by re-executing or discarding useless outputs. Additionally, our interfaces apply to other scenarios of data approximation that could benefit from their protections or configurations.

5.1.1 Applicability

The applicability of the general AC techniques depends on the context of the applications. Thus, our interfaces also depend on this context that determines the inaccuracy limit of the application results. `AxRAM` assumes that applications executed in its platform are resilient to errors. `SmartApprox` infers the context through the applications from the knowledge base that reflect the domain of the input applications. Therefore, our interface configures the approximation knob without requiring domain-specific metrics at runtime and the executing application must correspond to the domain abstracted from the training phase.

The data that tolerate errors should be substantial to achieve the benefits that the protection mechanisms provide. Thus, the energy savings that our interfaces provide are limited to how intensive accessing and storing the application is with the data resilient to errors. The evaluated voltage-scaling technique reduces power across activation, precharge, and refresh operations, besides the portion of the static power that comes from the data array [12]. Thus, different patterns of access and data storage benefit from this technique with a reduction in static and dynamic power from the memory. However, the features of our interfaces are suitable to other techniques for memory approximation that have nondeterministic error behavior and allow for memory partition into error-free and approximate.

5.1.2 System Implementation

Our interfaces execute in approximate memory systems where the application stack is exposed to nondeterministic and probabilistic errors. The implementation of our proposal involves a software control and an approximation knob that modifies hardware parameters. The software control is responsible for recovery and data treatment, by triggering re-executions or treating incorrect virtual memory addresses, but also for configuration steps, such as building the knowledge base and determining the approximation level at runtime. The hardware modifications should comprise the architectural model of `AxRAM` to provide isolation of some memory regions, and also sensors for detecting the current error scenario, and counters for extracting the features of `SmartApprox`. Some sensors, such as temperature, are present in commodity hardware and can be used in the runtime system of `SmartApprox`, as well as some performance counters used as application features.

5.1.3 Features Extraction

`SmartApprox` uses features of the applications to determine a relation between their error tolerance. These features are statistics from the execution in the same hardware that could be collected through general-purpose architecture counters. Depending on the features set to be collected, some instrumentation or analysis tool is necessary. However, the extraction of the features is needed on a single accurate execution of a new application only and, thus, approximate executions can be performed without this tool or instrumentation. Instrumentation can be automatically performed by compilers, while analysis tools collect data through the OS or hardware counters.

5.1.4 Multi-application Environment

When multiple applications are competing for an approximate memory that admits only one approximation level, the knob could be adjusted according to the level that has the lower error rate. Although this policy decreases the energy benefits of the application more resilient to errors, it maintains the execution producing outputs with acceptable quality for all concurrent running applications.

These decreased benefits could be mitigated through other approximation techniques that do not depend on memory regions to perform, such as STT-MRAM write parameters, or other architectural models that allow for multiple regions with different approximation levels. However, the use of the techniques configured at access restricts the memory approximations, while the complexity and overheads of the architectural model increase with the supported approximation levels.

5.2 Limitations

We propose an architectural model for memories that implement a set of approximate states, which are operating points that induce errors in the stored data. By controlling the error rate and the region of the memory array that is affected, this interface allows an external agent to control the degree of approximation provided, inducing energy savings by tolerating some errors in the stored data. In our evaluation, however, we employ the interface in a limited simulation scenario. In the remainder of this section, we discuss some limitations and implications of the execution with our interfaces in production with real hardware conditions.

5.2.1 Perturbation Model

We restrain our approach on nondeterministic and probabilistic perturbation models, where different executions with the same input may generate different results. Thus, the configuration has to correspond to a statistical threshold of inaccuracy. Our evaluation considers the supply voltage as the knob of an approximate memory, which allows energy savings at several memory operations. Other memory approximations with nondeterministic error behavior could be applied with our interfaces. These parameters are used as approximation knobs to obtain energy savings, and the probability of an error occurrence has a direct relation with the changes in their values. Our proposal mitigates the perturbation outcome problem to predict how an application reacts in several approximation levels in the presence of these errors.

The error from adjusting memory parameters may present behavior dependent on data patterns, spatial distribution, and even sensitivities that affect nearby cells [12, 86]. Our simulated evaluation scenario considers single-bit soft errors with uniform bitflips on software models to simplify the error representation. The uniform bitflip sufficiently represents the modeling for data patterns and region dependency [61], while the effects with our interfaces under single-bit errors are similar to those of multi-bit errors. Incorrect pointers resulting from multi-bit errors would also be truncated by the `AxRAM` addressing mask that would avoid data crashes caused by these pointers. The correction provided by this mask would achieve similar results on both scenarios, with the same level of approximation, in the same order of magnitude on single and multiple-bit errors. Furthermore, evidence suggests that the single bitflip model is enough in resilience studies for less pessimistic fault injection scenarios since the outcomes are similar in most cases [105].

5.2.2 Hardware Characterization

`SmartApprox` configures the approximate memory based on approximation levels that set the values of the approximation knob. The approximation levels depend on a hardware characterization to understand the relation between error and the approximation knob, as well as the changes in this relation that can be detected at runtime. For example, the fabrication process of the memory can affect some cells to be more susceptible to errors. These cells are weaker than others and usually exhibit spatial concentration at certain regions of the memory [13, 70]. Thus, if some data are stored into a memory region that concentrates more weak cells, the approximation knob should be adjusted according to a characterized error scenario that realizes this situation. In addition to detecting regions of weaker cells, the hardware characterization should comprise other variables that cause dynamic changes on the error probability according to the approximation technique, such as temperature, aging, and data density.

The complexity of the characterization increases the supported error scenarios by the combination of the considered variables (e.g., a characterization that previews one change in N variables supports 2^N error scenarios). Our evaluation simplifies the characterization in favor of the understanding with only three error scenarios, however, an implementation of `SmartApprox` depends on such complexity to detect the current error scenario at runtime. Furthermore, the training phase should include the characterization and the possible changes that the application will be exposed in the real hardware.

5.2.3 Quality Control in Production Scenarios

In production environments, applications accept some quality loss in approximate environments, but with a tolerable limit. Thus, our configuration interface adjusts the application to an approximation level that respects this limit. To calculate the relative energy savings of our technique, we consider an average quality threshold achieved in each error rate, where it is possible to obtain lower than required quality in individual executions. However, we ensure through profiling that, on average, the application meets the required quality target.

After deployment, the output quality cannot be computed for every execution instance. Nevertheless, we configure the approximation knob to correspond to the observed error rate that achieves each average quality threshold. Since `AxRAM` decreases the number of crashes and quality has already been considered in the training phase, a watchdog technique can be employed to avoid application stalling (timeout), leading quality and energy to converge to the observed in the training phase. In our evaluation, the application runs in a computation environment where one input represents the workloads of the entire lifetime of the application. In case a more dynamic evaluation is required, some execution instances may be sampled and elected for non-approximate computation, fine-tuning the operating point selection and configuring the approximation knobs according to another round of features collection.

5.2.4 Acceptable Quality

The configuration of the approximation knob depends on an application-specific quality threshold defined in a training phase. We simplify the evaluation with normalized quality metrics with a unique threshold for all applications (90%). We consider that this threshold is enough for a useful result for the tested applications, however, depending on the context, an application may tolerate more or less quality depreciation.

`SmartApprox` does not require quality metrics in the runtime system, thus the quality obtained by the training applications should be treated as analogous in the input application. Thus, the interface depends on a correspondence of the acceptability criteria between the input and the training applications. When this correspondence cannot be directly defined, a sufficiently high-quality threshold is enough to guide the search for useful results.

5.3 Future Directions

The proposed interfaces mitigate the energy-accuracy tradeoff on the exploration of memory approximations that prospect for proposals that could present contributions and advances on the state-of-art. In this section, we discuss future work that could emerge from this research.

5.3.1 Data Protection at Instruction Granularity

The memory architecture model of `AxRAM` controls the data error exposition by the division of the memory array into reliable and approximate regions. This partition allows for protecting coarse-grained critical data identified previously. As we state in our work, some critical data may be identified at runtime, and then we propose a treatment for incorrect memory references. However, the control only through the partition of the memory restricts the protection at the storage region.

A more detailed characterization of the voltage-induced errors in DRAMs shows that longer access latency may soften the error at lower voltages [13]. Therefore, even with a lower supply voltage, a memory cell can be reliably read or written if the latency parameters were changed. This change impacts the performance but allows for error protection at each instruction. Further studies would mitigate the tradeoff between error protection at this granularity and the performance overheads that it causes.

5.3.2 Configuration with Multiple Knobs

Our configuration interface adjusts the approximation knobs according to approximation levels from the training phase. To consider more than one approximation knob amongst the memory hierarchy, each approximation level should be defined in the training phase with these multiple knobs. For example, a combined approximation with scaled voltage in an SRAM cache and adjusted refresh rate on a DRAM main memory should have each approximation level with a correspondent value on the voltage and refresh rate.

The knowledge base would be built with the approximation level of the correspondent values of the knobs and the hardware support should allow the adjustment of these multiple knobs at runtime. Future work can explore the benefits of multiple knobs to improve energy savings or to explore multi-objective gains, combining knobs to achieve performance and energy benefits.

5.3.3 Domain-oriented Knowledge Base

The knowledge base of `SmartApprox` guides the configuration of each new application at runtime. The knowledge base is built in a training phase that requires a representative set of applications and their quality specifications that reflect the context of the input applications. Our evaluation is based on quality metrics and enough threshold to define a useful result. However, the definition of a useful result depends on the computational outline that the application is inserted [33]. For instance, a domain of object detection in images has more well-delimited acceptability with an algorithm that identifies objects than an SSI value of the quality threshold. Therefore, we propose as future work an environment with a specific domain on the acceptability criteria of the training applications on `SmartApprox`, which would contribute to showing the correspondence between a more precise definition of useful results with the input applications.

5.3.4 Error Tolerance Benchmarks

The context of the configuration that `SmartApprox` applies depends on the representativeness of the training applications, as also the effectiveness of the proposed protection and recovery mechanisms. Our experimental evaluation considers applications from several computing domains from available benchmarks where we specify quality requirements. However, the available benchmarks have general applications that may not represent a specific domain or analyzed representativeness in the context of AC.

An analysis of several contexts of applications that tolerate errors with different quality specifications and workloads would contribute to a more complete evaluation of different environments of error tolerance with configuration and protection of critical data. This analysis may comprise aspects of different data access patterns and the entire context of data usage, with well-defined and specific thresholds of quality acceptability.

5.4 Final Remarks

We propose interfaces for configuration, protection, and recovery for applications that execute in environments with approximate memories. We introduce transparent interfaces that do not require changes in the applications to indicate computational elements amenable to approximations. Our protection interface isolates from errors common critical data to many types of applications, improve execution resilience by treating data that would cause crashes, and recover from lost executions through re-execution and validation mechanisms. Our configuration interface determines approximation without requiring specific quality metrics for the input application and senses modifications in the error scenario according to hardware characterization and sensors.

Our interfaces explore different application features to determine approximation levels based on previous knowledge. We detail how learning models can be used to determine the appropriate approximation level for a new application and perform a search for the best set of features that expresses the error tolerance. Our results evidence that the proposed interfaces configure approximations with energy savings and average quality close to the achieved by an exhaustive search, while improve execution resilience and reduce a significant part of the crashes, depending on the application and environment. The sources from `AxRAM` implementation are available with the AxPIKE simulator [1], while the data, evaluated applications, and source code from `SmartApprox` are available on the page of our research group [2].

[1]`https://github.com/VArchC/axpike-isa-sim`
[2]`https://varchc.github.io/smartapprox/`

Bibliography

[1] Agrawal, A., Choi, J., Gopalakrishnan, K., Gupta, S., Nair, R., Oh, J., Prener, D. A., Shukla, S., Srinivasan, V., and Sura, Z. Approximate computing: Challenges and opportunities. In Proceedings of the IEEE International Conference on Rebooting Computing (ICRC) (2016), IEEE.

[2] Amanollahi, S., Kamal, M., Afzali-Kusha, A., and Pedram, M. Circuit-Level Techniques for Logic and Memory Blocks in Approximate Computing Systems. Proceedings of the IEEE 108, 12 (2020), 2150–2177.

[3] Anajemba, J. H., Ansere, J. A., Sam, F., Iwendi, C., and Srivastava, G. Optimal soft error mitigation in wireless communication using approximate logic circuits. Sustainable Computing: Informatics and Systems 30 (2021), 100521–100527.

[4] Aponte-Moreno, A., Moncada, A., Restrepo-Calle, F., and Pedraza, C. A review of approximate computing techniques towards fault mitigation in HW/SW systems. In Proceedings of the IEEE Latin-American Test Symposium (LATS) (2018), IEEE.

[5] Ataei, S., and Stine, J. E. A 64 kB Approximate SRAM Architecture for Low-Power Video Applications. IEEE Embedded Systems Letters 10, 1 (2017), 10–13.

[6] Avanaki, A. N. Exact global histogram specification optimized for structural similarity. Optical Review 16, 6 (2009), 613–621.

[7] Baek, W., and Chilimbi, T. M. Green: A Framework for Supporting Energy-Conscious Programming using Controlled Approximation. Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI) (2010), 198–209.

[8] Boston, B., Sampson, A., Grossman, D., and Ceze, L. Probability type inference for flexible approximate programming. In Proceedings of the Conference on Object-Oriented Programming, Systems, Language, and Applications (OOPSLA) (2015), ACM.

[9] Carbin, M., Misailovic, S., and Rinard, M. C. Verifying quantitative reliability for programs that execute on unreliable hardware. Communications of the ACM 59, 8 (2016), 83–91.

[10] Chandrasekar, K., Weis, C., Akesson, B., Wehn, N., and Goossens, K. Towards variation-aware system-level power estimation of DRAMs. In Proceedings of the Design Automation Conference (DAC) (2013), ACM.

[11] Chandrasekar, K., Weis, C., Li, Y., Akesson, B., Wehn, N., and Goossens, K. DRAMPower: Open-source DRAM power & energy estimation tool, 2012. `http://www.drampower.info`.

[12] Chang, K. K., Kashyap, A., Hassan, H., Pekhimenko, G., Khan, S., and Mutlu, O. Understanding latency variation in modern DRAM chips: Experimental characterization, analysis, and optimization. In Proceedings of the SIGMETRICS/IFIP Performance Conference (SIGMETRICS/ Performance) (2016), ACM.

[13] Chang, K. K., Yağlikçi, A. G., Ghose, S., Agrawal, A., Chatterjee, N., Kashyap, A., Lee, D., O'Connor, M., Hassan, H., and Mutlu, O. Understanding Reduced-Voltage Operation in Modern DRAM Devices. In Proceedings of the ACM on Measurement and Analysis of Computing Systems (POMACS) (2017), ACM.

[14] Chen, Y., Chhugani, J., Dubey, P., Hughes, C. J., Kim, D., Kumar, S., Lee, V. W., Nguyen, A. D., and Smelyanskiy, M. Convergence of recognition, mining, and synthesis workloads and its implications. Proceedings of the IEEE 96, 5 (2008), 790–807.

[15] Cherubin, S., and Agosta, G. Tools for Reduced Precision Computation: A Survey. ACM Computing Surveys 53, 2 (2020), 1–35.

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

[17] Cohen, M., Zhu, H. S., Senem, E. E., and Liu, Y. D. Energy types. ACM SIGPLAN Notices 47, 10 (2012), 831–850.

[18] Cui, J., Liu, J., Huang, J., Zhu, H., and Yang, L. T. ApproxRefresh: Enabling Uncorrectable Data Reuse on Flash Memory with Approximate Read. In Proceedings of the SIGPLAN/SIGBED Conference on Languages, Compilers, Tools, and Theory of Embedded Systems (LCTES) (2020), ACM.

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

[20] De Kruijf, M., Nomura, S., and Sankaralingam, K. Relax: An architectural framework for software recovery of hardware faults. In Proceedings of the International Symposium on Computer Architecture (ISCA) (2010), ACM.

[21] De Silva, H., Ho, N. M., Santosa, A. E., and Wong, W. F. ApproxSymate: Path sensitive program approximation using symbolic execution. In Proceedings of the SIGPLAN/SIGBED Conference on Languages, Compilers, Tools, and Theory of Embedded Systems (LCTES) (2019), ACM.

[22] Esmaeilzadeh, H., Blem, E., St. Amant, R., Sankaralingam, K., and Burger, D. Dark silicon and the end of multicore scaling. IEEE Micro 32, 3 (2012), 122–134.

[23] Esmaeilzadeh, H., Sampson, A., Ceze, L., and Burger, D. Architecture support for disciplined approximate programming. In Proceedings of the International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS) (2012), ACM.

[24] Esmaeilzadeh, H., Sampson, A., Ceze, L., and Burger, D. Neural Acceleration for General-Purpose Approximate Programs. In Proceedings of the International Symposium on Microarchitecture (MICRO) (2012), IEEE.

[25] Fabrício Filho, J., Felzmann, I., Azevedo, R., and Wanner, L. A Resilient Interface for Approximate Data Access. In Proceedings of the Brazilian Symposium on Computing Systems Engineering (SBESC) (2019), IEEE.

[26] Fabrício Filho, J., Felzmann, I., Azevedo, R., and Wanner, L. AxRAM: A lightweight implicit interface for approximate data access. Future Generation Computer Systems 113 (2020), 556–570.

[27] Fabrício Filho, J., Felzmann, I., and Wanner, L. Tratamento de Ponteiros Incorretos armazenados em Memórias Aproximadas. In Anais da Escola Regional de Alto Desempenho de São Paulo (ERAD-SP) (2019).

[28] Fabrício Filho, J., Felzmann, I., and Wanner, L. Sensibilidade a erros em aplicações na arquitetura RISC-V. In Anais da Escola Regional de Alto Desempenho de São Paulo (ERAD-SP) (2020).

[29] Fabrício Filho, J., Felzmann, I., and Wanner, L. Transparent Resilience for Approximate DRAM. In Proceedings of the International Conference on Architecture of Computing Systems (ARCS) (2021), C. Hochberger, L. Bauer, and T. Pionteck, Eds., vol. 12800 of Lecture Notes in Computer Science, Springer.

[30] Fabrício Filho, J., Felzmann, I., and Wanner, L. SmartApprox: Learning-based Configuration of Approximate Memories for Energy-efficient Execution. Sustainable Computing: Informatics and Systems (2022).

[31] Fatemieh, S. E., Farahani, S. S., and Reshadinezhad, M. R. LAHAF: Low-power, area-efficient, and high-performance approximate full adder based on static CMOS. Sustainable Computing: Informatics and Systems 30 (2021), 100529.

[32] Felzmann, I., Fabrício Filho, J., Azevedo, R., and Wanner, L. Impact of memory approximation on energy efficiency. In Proceedings of the Symposium on High-Performance Computing Systems (WSCAD) (2018), IEEE.

[33] Felzmann, I., Fabrício Filho, J., Oliveira, J. R., and Wanner, L. How much quality is enough quality? A case for acceptability in approximate designs. In Proceedings of the IEEE International Conference on Computer Design (ICCD) (2021).

[34] Felzmann, I., Fabrício Filho, J., and Wanner, L. Risk-5: Controlled Approximations for RISC-V. IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems 39, 11 (2020), 4052–4063.

[35] Felzmann, I., Fabrício Filho, J., and Wanner, L. AxPIKE: Instruction-level Injection and Evaluation of Approximate Computing. In Proceedings of the Design, Automation, and Test in Europe Conference (DATE) (2021), EDA Consortium.

[36] Felzmann, I., Susin, M. M., Duenha, L., Azevedo, R., and Wanner, L. F. ADeLe: A description language for approximate hardware. Future Generation Computer Systems 102 (2020), 245–258.

[37] Froehlich, S., Grosse, D., and Drechsler, R. Approximate memory: Data storage in the context of approximate computing. In Information Storage: A Multidisciplinary Perspective. Springer International, 2019, pp. 111–134.

[38] Fursin, G. Collective Benchmark (cBench), 2008. `http://ctuning.org/cbench`.

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

[40] Ghose, S., Agrawal, A., O'Connor, M., Mutlu, O., Yağlikçi, A. G., Gupta, R., Lee, D., Kudrolli, K., Liu, W. X., Hassan, H., Chang, K. K., and Chatterjee, N. What Your DRAM Power Models Are Not Telling You. ACM SIGMETRICS Performance Evaluation Review 46, 1 (2019), 110–150.

[41] Ghosh, A., Raha, A., and Mukherjee, A. Energy-Efficient IoT-Health Monitoring System using Approximate Computing. Internet of Things 9 (2020), 100166–100183.

[42] Gottscho, M., BanaiyanMofrad, A., Dutt, N., Nicolau, A., and Gupta, P. DPCS: Dynamic power/capacity scaling for SRAM caches in the nanoscale era. ACM Transactions on Architecture and Code Optimization 12, 3 (2015), 1–26.

[43] Gottscho, M., Shoaib, M., Govindan, S., Sharma, B., Wang, D., and Gupta, P. Measuring the Impact of Memory Errors on Application Performance. IEEE Computer Architecture Letters 16, 1 (2017), 51–55.

[44] Gouy, I. The Computer Language Benchmarks Game, 2015. `https://benchmarksgame-team.pages.debian.net/benchmarksgame/`.

[45] Guthaus, M. R., Ringenberg, J. S., Ernst, D., Austin, T. M., Mudge, T., and Brown, R. B. MiBench: A free, commercially representative embedded benchmark suite. In IEEE International Workshop on Workload Characterization (WWC) (2001), IEEE.

[46] Ha, M., Hwang, S., Kim, J., Lee, Y., and Lee, S. Hierarchical Approximate Memory for Deep Neural Network Applications. In Proceedings of the Asilomar Conference on Signals, Systems, and Computers (ACSSC) (2020), IEEE.

[47] Hajizadeh, F., Binesh Marvasti, M., Asghari, S. A., Abbas Mollaei, M., and Rahmani, A. M. Configurable DSI partitioned approximate multiplier. Future Generation Computer Systems 115 (2021), 100–114.

[48] Heijmen, T. Soft Errors from Space to Ground: Historical Overview, Empirical Evidence, and Future Trends. In Soft Errors in Modern Electronic Systems, M. Nicolaidis, Ed. Springer, 2011, ch. 1, pp. 1–25.

[49] Ho, N.-M., Manogaran, E., Wong, W.-F., and Anoosheh, A. Efficient Floating Point Precision Tuning for Approximate Computing. In Proceedings of the Asia and South Pacific Design Automation Conference (ASP-DAC) (2017).

[50] Hoffmann, H., Misailovic, S., Sidiroglou, S., Agarwal, A., and Rinard, M. Using code perforation to improve performance, reduce energy consumption, and respond to failures. Tech. rep., MIT, 2009.

[51] Hoffmann, H., Sidiroglou, S., Carbin, M., Misailovic, S., Agarwal, A., and Rinard, M. Dynamic knobs for responsive power-aware computing. ACM SIGPLAN Notices 47, 4 (2012), 199.

[52] Jordan, M. G., Brandalero, M., Malfatti, G. M., Oliveira, G. F., Lorenzon, A. F., da Silva, B. C., Carro, L., Rutzig, M. B., and Beck, A. C. S. Data clustering for efficient approximate computing. Design Automation for Embedded Systems 24, 1 (2020), 3–22.

[53] Joshi, K., Fernando, V., and Misailovic, S. Aloe: Verifying reliability of approximate programs in the presence of recovery mechanisms. In Proceedings of the International Symposium Code Generation and Optimization (CGO) (2020), ACM.

[54] Jothin, R., and Mohamed, M. P. High Performance Approximate Memories for Image Processing Applications. Journal of Electronic Testing: Theory and Applications (JETTA) 36, 3 (2020), 419–428.

[55] Kerrisk, M. SBRK(2) - Linux man page. man7.org. `https://linux.die.net/man/2/sbrk`.

[56] Khudia, D. S., Zamirai, B., Samadi, M., and Mahlke, S. Rumba: an online quality management system for approximate computing. ACM SIGARCH Computer Architecture News 43, 3S (2016), 554–566.

[57] Kim, Y., Daly, R., Kim, J., Fallin, C., Lee, J. H., Lee, D., Wilkerson, C., Lai, K., and Mutlu, O. Flipping bits in memory without accessing them: An experimental study of DRAM disturbance errors. In Proceedings of the International Symposium on Computer Architecture (ISCA) (2014).

[58] Kim, Y., Venkataramani, S., Chandrachoodan, N., and Raghunathan, A. Data Subsetting: A Data-Centric Approach to Approximate Computing. In Proceedings of the Design, Automation, and Test in Europe Conference (DATE) (2019), IEEE.

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

[60] Kolhapure, A., and Kumar, A. SRAM in hold-operation: Modeling the interaction of soft-errors and switching power-supply noise. In Proceedings of the IEEE International Symposium on Circuits and Systems (ISCAS) (2012).

[61] Koppula, S., Orosa, L., Yaglikci, A. G., Azizi, R., Shahroodi, T., Kanellopoulos, K., and Mutlu, O. EDEN: Enabling energy-efficient, high-performance deep neural network inference using approximate DRAM. In Proceedings of the International Symposium on Microarchitecture (MICRO) (2019), IEEE.

[62] Kugler, L. Is "good enough" computing good enough? Communications of the ACM 58, 5 (2015), 12–14.

[63] Kulkarni, P., Gupta, P., and Ercegovac, M. Trading Accuracy for Power with an Underdesigned Multiplier Architecture. In Proceedings of the International Conference on VLSI Design (VLSID) (2011), IEEE.

[64] Kumar, A., Rabaey, J., and Ramchandran, K. SRAM supply voltage scaling: a reliability perspective. In Proceedings of the International Symposium on Quality Electronic Design (ISQED) (2009), IEEE.

[65] Lee, D., Khan, S., Subramanian, L., Ghose, S., Ausavarungnirun, R., Pekhimenko, G., Seshadri, V., and Mutlu, O. Design-Induced Latency Variation in Modern DRAM Chips. In Proceedings of the ACM on Measurement and Analysis of Computing Systems (POMACS) (2017), vol. 1.

[66] Lee, Y., Waterman, A., Avizienis, R., Cook, H., Sun, C., and Stojanovic, V. Spike, a RISC-V ISA Simulator, 2014. `https://github.com/riscv/riscv-isa-sim`.

[67] Li, S., Park, S., and Mahlke, S. Sculptor: Flexible approximation with selective dynamic loop perforation. In Proceedings of the International Conference on Supercomputing (ICS) (2018), ACM.

[68] Li, X., and Yeung, D. Application-level correctness and its impact on fault tolerance. In Proceedings of the International Symposium on High-Performance Computer Architecture (HPCA) (2007), IEEE.

[69] Liu, J., Jaiyen, B., Kim, Y., Wilkerson, C., and Mutlu, O. An experimental study of data retention behavior in modern DRAM devices. In Proceedings of the International Symposium on Computer Architecture (ISCA) (2013), ACM.

[70] Liu, J., Jaiyen, B., Veras, R., and Mutlu, O. RAIDR: Retention-Aware Intelligent DRAM Refresh. ACM SIGARCH Computer Architecture News 40, 3 (2012), 1–12.

[71] Liu, S., Pattabiraman, K., Moscibroda, T., and Zorn, B. G. Flikker: Saving DRAM refresh-power through critical data partitioning. ACM SIGPLAN Notices 47, 4 (2012), 213–224.

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

[73] Lopes, B. C., Auler, R., Ramos, L., Borin, E., and Azevedo, R. SHRINK: Reducing the ISA Complexity via Instruction Recycling. In Proceedings of the International Symposium on Computer Architecture (ISCA) (2015), ACM.

[74] Lou, L., Nguyen, P., Lawrence, J., and Barnes, C. Image Perforation: Automatically Accelerating Image Pipelines by Intelligently Skipping Samples. ACM Transactions on Graphics 35, 5 (2016), 1–14.

[75] Lucas, J., Alvarez-Mesa, M., Andersch, M., and Juurlink, B. Sparkk: Quality-Scalable Approximate Storage in DRAM. In Proceedings of the Memory Forum, ISCA (2014), CS, UTAH.

[76] Maity, B., Donyanavard, B., Surhonne, A., Rahmani, A., Herkersdorf, A., and Dutt, N. AXES: Approximation Manager for Emerging Memory Architectures. Tech. rep., UCLA Irvine, 2020.

[77] Maity, B., Donyanavard, B., Surhonne, A., Rahmani, A., Herkersdorf, A., and Dutt, N. SEAMS: Self-Optimizing Runtime Manager for Approximate Memory Hierarchies. ACM Transactions on Embedded Computing Systems 20, 5 (2021), 1–26.

[78] Masadeh, M., Hasan, O., and Tahar, S. Using Machine Learning for Quality Configurable Approximate Computing. In Proceedings of the Design, Automation, and Test in Europe Conference (DATE) (2019), IEEE.

[79] Mittal, S. A survey of techniques for approximate computing. ACM Computing Surveys 48, 4 (2016), 62:1–62:33.

[80] Moreau, T., San Miguel, J., Wyse, M., Bornholt, J., Alaghi, A., Ceze, L., Enright Jerger, N., and Sampson, A. A Taxonomy of General Purpose Approximate Computing Techniques. IEEE Embedded Systems Letters 10, 1 (2018), 2–5.

[81] Moreau, T., Wyse, M., Nelson, J., Sampson, A., Esmaeilzadeh, H., Ceze, L., and Oskin, M. SNNAP: Approximate computing on programmable SoCs via neural acceleration. In Proceedings of the International Symposium on High-Performance Computer Architecture (HPCA) (2015), IEEE.

[82] Muralimanohar, N., Shafiee, A., and Srinivas, V. CACTI 7.0: A Tool to Model Caches/Memories, 3D stacking, and off-chip IO, 2009. `https://www.hpl.hp.com/research/cacti/`.

[83] Nguyen, D. T., and Chang, I.-J. An Approximate DRAM Architecture for Energy-efficient Deep Learning. Journal of Semiconductor Engineering 1, 1 (2020), 31–37.

[84] Nodeh, M. T. T., Bazzaz, M., and Ejlali, A. Exploiting approximate MLC-PCM in low-power embedded systems. ACM Transactions on Embedded Computing Systems 17, 1 (2017), 1–25.

[85] Oboril, F., Shirvanian, A., and Tahoori, M. Fault tolerant approximate computing using emerging non-volatile spintronic memories. In Proceedings of the VLSI Test Symposium (VTS) (2016), IEEE.

[86] Orosa, L., Yağlikçi, A. G., Luo, H., Olgun, A., Park, J., Hassan, H., Patel, M., Kim, J. S., and Mutlu, O. A deeper look into RowHammer's sensitivities: Experimental analysis of real DRAM chips and implications on future attacks and defenses. In Proceedings of the International Symposium on Microarchitecture (MICRO) (2021), ACM.

[87] Osta, M., Ibrahim, A., Chible, H., and Valle, M. Inexact Arithmetic Circuits for Energy Efficient IoT Sensors Data Processing. In Proceedings of the IEEE International Symposium on Circuits and Systems (ISCAS) (2018), IEEE.

[88] Pouchet, L.-N. Polybench: The polyhedral benchmark suite, 2012. `https://web.cse.ohio-state.edu/~pouchet.2/software/polybench/`.

[89] Pozidis, H., Papandreou, N., Sebastian, A., Pantazi, A., Mittelholzer, T., Close, G. F., and Eleftheriou, E. Enabling Technologies for Multilevel Phase-Change Memory. In Proceedings of the European Phase Change and Ovonic Symposium (E\PCOS) (Zürich, Switzerland, 2011).

[90] Qiu, K., Luo, J., Gong, Z., Zhang, W., Wang, J., Xu, Y., Li, T., and Xue, C. J. Refresh-aware loop scheduling for high performance low power volatile STT-RAM. In Proceedings of the IEEE International Conference on Computer Design (ICCD) (2016), IEEE.

[91] Raha, A., and Raghunathan, V. qLUT: Input-Aware Quantized Table Lookup for Energy-Efficient Approximate Accelerators. ACM Transactions on Embedded Computing Systems 16, 5s (2017), 1–23.

[92] Raha, A., Sutar, S., Jayakumar, H., and Raghunathan, V. Quality Configurable Approximate DRAM. IEEE Transactions on Computing 66, 7 (2017), 1172–1187.

[93] Raha, A., Venkataramani, S., Raghunathan, V., and Raghunathan, A. Energy-Efficient Reduce-and-Rank Using Input-Adaptive Approximations. IEEE Transactions on Very Large Scale Integration Systems 25, 2 (2017), 462–475.

[94] Rahimi, A., and Gupta, R. K. Hardware/Software Codesign for Energy Efficiency and Robustness: From Error-Tolerant Computing to Approximate Computing. In Dependable Embedded Systems, N. Dutt, G. Martin, and P. Marwedel, Eds. Springer, 2021.

[95] Ranjan, A., Raha, A., Raghunathan, V., and Raghunathan, A. Approximate Memory Compression. IEEE Transactions on Very Large Scale Integration Systems 28, 4 (2020), 980–991.

[96] Ranjan, A., Venkataramani, S., Pajouhi, Z., Venkatesan, R., Roy, K., and Raghunathan, A. STAxCache: An approximate, energy efficient STT-MRAM cache. In Proceedings of the Design, Automation, and Test in Europe Conference (DATE) (2017), IEEE.

[97] Rashidi, S., Jalili, M., and Sarbazi-Azad, H. Improving MLC PCM Performance through Relaxed Write and Read for Intermediate Resistance Levels. ACM Transactions on Architecture and Code Optimization 15, 1 (2018), 1–31.

[98] Rigo, S., Araújo, G., Bartholomeu, M., and Azevedo, R. ArchC: A SystemC-based architecture description language. In Proceedings of the Symposium on Computer Architecture and High Performance Computing (SBAC-PAD) (2004), IEEE.

[99] Ringenburg, M., Sampson, A., Ackerman, I., Ceze, L., and Grossman, D. Monitoring and debugging the quality of results in approximate programs. In Proceedings of the International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS) (2015), ACM.

[100] Rosa, F. R., Brum, R. M., Wirth, G., Ost, L., and Reis, R. Impact of dynamic voltage scaling and thermal factors on FinFET-based SRAM reliability. In Proceedings of the IEEE International Conference on Electronics, Circuits, and Systems (ICECS) (2015), IEEE.

[101] Roy, P., Ray, R., Wang, C., and Wong, W. F. ASAC: Automatic sensitivity analysis for approximate computing. In Proceedings of the SIGPLAN/SIGBED Conference on Languages, Compilers, Tools, and Theory of Embedded Systems (LCTES) (2014), ACM.

[102] Samadi, M., Jamshidi, D. A., Lee, J., and Mahlke, S. Paraprox: pattern-based approximation for data parallel applications. In Proceedings of the International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS) (2014), ACM.

[103] Sampson, A., Dietl, W., Fortuna, E., Gnanapragasam, D., Ceze, L., and Grossman, D. EnerJ: Approximate data types for safe and general low-power computation. In Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI) (2011), ACM.

[104] Sampson, A., Nelson, J., Strauss, K., and Ceze, L. Approximate storage in solid-state memories. In Proceedings of the International Symposium on Microarchitecture (MICRO) (2013), ACM.

[105] Sangchoolie, B., Pattabiraman, K., and Karlsson, J. One bit is (not) enough: An empirical study of the impact of single and multiple bit-flip errors. In IEEE/IFIP International Conference on Dependable Systems and Networks (DSN) (2017).

[106] Sayed, N., Bishnoi, R., and Tahoori, M. B. Approximate Spintronic Memories. ACM Journal on Emerging Technologies in Computing Systems 16, 4 (2020), 1–22.

[107] Semiconductor Research Corporation. Decadal Plan for Semiconductors: Full Report. Tech. rep., Semiconductor Research Corporation, 2021.

[108] Seong, N. H., Woo, D. H., Srinivasan, V., Rivers, J. A., and Lee, H.-H. S. SAFER: Stuck-At-Fault Error Recovery for Memories. In Proceedings of the International Symposium on Microarchitecture (MICRO) (2010), IEEE.

[109] Shafique, M., Hafiz, R., Rehman, S., El-Harouni, W., and Henkel, J. Invited - Cross-layer approximate computing: from logic to architectures. In Proceedings of the Design Automation Conference (DAC) (2016), ACM.

[110] Silveira, J. E., Felzmann, I., Fabrício Filho, J., and Wanner, L. RV-Across: An Associative Processing Simulator. In Proceedings of the Symposium on High-Performance Computing Systems (WSCAD) (2020).

[111] Stazi, G., Adani, L., Mastrandrea, A., Olivieri, M., and Menichelli, F. Impact of Approximate Memory Data Allocation on a H.264 Software Video Encoder. In Proceedings of the International Conference on High Performance Computing (ISC) (2018), R. Yokota, M. Weiland, J. Shalf, and S. Alam, Eds., vol. 11203 LNCS of Lecture Notes in Computer Science, Springer.

[112] Teimoori, M. T., Hanif, M. A., Ejlali, A., and Shafique, M. AdAM: Adaptive approximation management for the non-volatile memory hierarchies. In Proceedings of the Design, Automation, and Test in Europe Conference (DATE) (2018), IEEE.

[113] Tolentino, M. E., Turner, J., and Cameron, K. W. Memory miser: Improving main memory energy efficiency in servers. IEEE Transactions on Computers 58, 3 (2009), 336–350.

[114] Tovletoglou, K., Nikolopoulos, D. S., and Karakonstantis, G. Access-aware DRAM failure-rate estimation under relaxed refresh operations. In Proceedings of the International Conference on Embedded Computer Systems: Architectures, Modeling, and Simulation (SAMOS) (2017), IEEE.

[115] Venkatagiri, R., Mahmoud, A., Hari, S. K. S., and Adve, S. V. Approxilyzer: Towards a systematic framework for instruction-level approximate computing and its application to hardware resiliency. In Proceedings of the International Symposium on Microarchitecture (MICRO) (2016), IEEE.

[116] Venkataramani, S., Chippa, V. K., Chakradhar, S. T., Roy, K., and Raghunathan, A. Quality programmable vector processors for approximate computing. In Proceedings of the International Symposium on Microarchitecture (MICRO) (2013), ACM.

[117] Venkataramani, S., Sabne, A., Kozhikkottu, V., Roy, K., and Raghunathan, A. SALSA: Systematic logic synthesis of approximate circuits. In Proceedings of the Design Automation Conference (DAC) (2012).

[118] Verdeja Herms, Y., and Li, Y. Crash skipping: A minimal-cost framework for efficient error recovery in approximate computing environments. In Proceedings of the ACM Great Lakes Symposium on VLSI (GLSVLSI) (2019), ACM.

[119] Wang, J., and Calhoun, B. H. Minimum supply voltage and yield estimation for large SRAMs under parametric variations. IEEE Transactions on Very Large Scale Integration Systems 19, 11 (2011), 2120–2125.

[120] Wang, J., Singhee, A., Rutenbar, R. A., and Calhoun, B. H. Statistical modeling for the minimum standby supply voltage of a full SRAM array. In Proceedings of the European Solid-State Circuits Conference (ESSCIRC) (2007), IEEE.

[121] Wang, J., Singhee, A., Rutenbar, R. A., and Calhoun, B. H. Two fast methods for estimating the minimum standby supply voltage for large SRAMs. IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems 29, 12 (2010), 1908–1920.

[122] Wang, T., Zhang, Q., and Xu, Q. ApproxQA: A unified quality assurance framework for approximate computing. In Proceedings of the Design, Automation, and Test in Europe Conference (DATE) (2017).

[123] Wang, Y., Deng, J., Fang, Y., Li, H., and Li, X. Resilience-aware frequency tuning for neural-network-based approximate computing chips. IEEE Transactions on Very Large Scale Integration Systems 25, 10 (2017), 2736–2748.

[124] Waterman, A., and Asanović, K. The RISC-V Instruction Set Manual, Volume I: User-Level ISA, Document Version 20190608-Base-Ratified. Tech. rep., RISC-V Foundation, 2019.

[125] Waterman, A., and Asanović, K. The RISC-V Instruction Set Manual, Volume II: Privileged Architecture, Document Version 20190608-Priv-MSU-Ratified. Tech. rep., RISC-V Foundation, 2019.

[126] Weis, C., Jung, M., Zulian, E. F., Sudarshan, C., Mathew, D. M., and Wehn, N. The Role of Memories in Transprecision Computing. In Proceedings of the IEEE International Symposium on Circuits and Systems (ISCAS) (2018), IEEE.

[127] Wong, T.-T. Performance evaluation of classification algorithms by k-fold and leave-one-out cross validation. Pattern Recognition 48, 9 (2015), 2839–2846.

[128] Xu, Q., Mytkowicz, T., and Kim, N. S. Approximate Computing: A Survey. IEEE Design and Test 33, 1 (2016), 8–22.

[129] Yarmand, R., Kamal, M., Afzali-Kusha, A., Esmaeli, P., and Pedram, M. OPTIMA: An Approach for Online Management of Cache Approximation Levels in Approximate Processing Systems. IEEE Transactions on Very Large Scale Integration Systems 29, 2 (2021), 434–446.

[130] Yarmand, R., Kamal, M., Afzali-Kusha, A., and Pedram, M. DART: A Framework for Determining Approximation Levels in an Approximable Memory Hierarchy. IEEE Transactions on Very Large Scale Integration Systems 28, 1 (2020), 273–286.

[131] Yazdanbakhsh, A., Mahajan, D., Esmaeilzadeh, H., and Lotfi-Kamran, P. AxBench: A multiplatform benchmark suite for approximate computing. IEEE Design and Test 34, 2 (2017), 60–68.

[132] Zhang, X., Zhang, Y., Childers, B. R., and Yang, J. DrMP: Mixed Precision-Aware DRAM for High Performance Approximate and Precise Computing. In Proceedings of the International Conference on Parallel Architectures and Compilation Techniques (PACT) (2017), IEEE.

[133] Zhang, Z., He, Y., He, J., Yi, X., Li, Q., and Zhang, B. Optimal slope ranking: an approximate computing approach for circuit pruning. In Proceedings of the IEEE International Symposium on Circuits and Systems (ISCAS) (2018), IEEE.

[134] Zhao, W., Tong, W., Feng, D., Liu, J., Chen, Z., Xu, J., Wu, B., Wang, C., and Liu, B. Improving the energy efficiency of STT-MRAM based approximate cache. In Proceedings of the Design, Automation, and Test in Europe Conference (DATE) (2021), IEEE.