Universidade Estadual de Campinas
Instituto de Computação

Isaías Bittencourt Felzmann

A framework for modeling and simulation of approximate computing in hardware

Uma ferramenta para modelagem e simulação de computação aproximada em hardware

CAMPINAS
2019

Isaías Bittencourt Felzmann

A framework for modeling and simulation of approximate computing in hardware

Uma ferramenta para modelagem e simulação de computação aproximada em hardware

Dissertação apresentada ao Instituto de Computação da Universidade Estadual de Campinas como parte dos requisitos para a obtenção do título de Mestre 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 Master in Computer Science.

Supervisor/Orientador: Prof. Dr. Lucas Francisco Wanner

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

CAMPINAS
2019

Agência(s) de fomento e nº(s) de processo(s): FAPESP, 2017/08015-8
ORCID: https://orcid.org/0000-0003-3048-8310

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

Felzmann, Isaías Bittencourt, 1992-
F349f A framework for modeling and simulation of approximate computing in hardware / Isaías Bittencourt Felzmann. – Campinas, SP : [s.n.], 2019.

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

1. Computação aproximada. 2. Computação consciente de energia. 3. Arquitetura de computador. 4. Hardware - Linguagens descritivas - Métodos de simulação. I. Wanner, Lucas Francisco, 1981-. II. Universidade Estadual de Campinas. Instituto de Computação. III. Título.

Informações para Biblioteca Digital

Título em outro idioma: Uma ferramenta para modelagem e simulação de computação aproximada em hardware
Palavras-chave em inglês:
Approximate computing
Energy-aware computing
Computer architecture
Computer hardware description languages - Simulation methods
Área de concentração: Ciência da Computação
Titulação: Mestre em Ciência da Computação
Banca examinadora:
Lucas Francisco Wanner [Orientador]
Antonio Carlos Schneider Beck Filho
Rodolfo Jardim de Azevedo
Data de defesa: 28-02-2019
Programa de Pós-Graduação: Ciência da Computação

Universidade Estadual de Campinas
Instituto de Computação

Isaías Bittencourt Felzmann

A framework for modeling and simulation of approximate computing in hardware

Uma ferramenta para modelagem e simulação de computação aproximada em hardware

Banca Examinadora:

- Prof. Dr. Lucas Francisco Wanner
  IC/UNICAMP
- Prof. Dr. Antonio Carlos Schneider Beck Filho
  INF/UFRGS
- Prof. Dr. Rodolfo Jardim de Azevedo
  IC/UNICAMP

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

Campinas, 28 de fevereiro de 2019

Acknowledgements

The author greets and thanks everyone who somehow contributed and allowed this work to be completed: Lucas Wanner, Rodolfo Azevedo, Andressa Marchesan, Matheus Susin, João Fabrício Filho, Jonathas Evangelista, Marcelo Jara, Liana Duenha and every colleague at the Computer Systems Laboratory, family member or friend for their help, advice or company.

In particular, thanks to São Paulo Research Foundation (FAPESP) for the financial support (grant #2017/08015-8). Any opinions, findings and conclusions or recommendations expressed in this material are those of the author and do not necessarily reflect the views of the Foundation.

Resumo

Pesquisas recentes têm introduzido unidades de hardware que produzem resultados incorretos de maneira determinística ou probabilística para um pequeno conjunto de entradas. Por outro lado, permitem um maior desempenho ou um consumo de energia significativamente menor em comparação com versões precisas das mesmas unidades. Como integrar, validar e avaliar essas alternativas em uma arquitetura ou processador, porém, permanece um desafio. A falta de ferramentas para representar e avaliar hardware aproximado leva desenvolvedores a verificar suas soluções de maneira independente, sem considerar interações com outros componentes, exigindo um grande esforço em modelagem e simulação. Neste trabalho, introduzimos *ADeLe*, uma linguagem de alto nível para descrever, configurar e integrar unidades de hardware aproximado em um processador. *ADeLe* reduz o esforço de desenvolvimento de hardware aproximado por modelar aproximações em um alto nível de abstração e injetá-las automaticamente em um modelo de processador para simulação arquitetural. Na ferramenta relacionada a *ADeLe*, aproximações podem modificar ou substituir completamente o comportamento de instruções de hardware através de políticas definidas pelo usuário. As instruções podem ser modificadas deterministicamente ou probabilisticamente (por exemplo, baseado em tensão de operação e frequência). Para proporcionar um ambiente de teste controlado, as aproximações podem ser ligadas e desligadas a partir do software em execução. O consumo de energia é automaticamente computado com base em modelos customizáveis no sistema. Assim, a ferramenta proporciona um método de verificação genérico e flexível, permitindo uma fácil avaliação da troca entre energia e qualidade de aplicações sujeitadas a hardware aproximado. Demonstramos a ferramenta pela introdução de variadas técnicas de aproximação em um modelo de processador, com o qual aplicações selecionadas foram executadas. Ao modelar módulos de hardware aproximado dedicados, mostramos como *ADeLe* representa unidades aritméticas aproximadas e unidades funcionais de precisão reduzida executando 4 aplicações de processamento de imagens e 2 de computação de ponto flutuante. Com outro método de aproximação, também mostramos como a ferramenta é utilizada para estudar o impacto de memórias alimentadas por tensão ajustável sobre 9 aplicações. Nossos experimentos demonstram as capacidades da ferramenta e como ela pode ser utilizada para gerar processadores virtuais aproximados e avaliar o equilíbrio entre energia e qualidade para diferentes aplicações com esforço reduzido.

Abstract

Recent research has introduced approximate hardware units that produce incorrect outputs deterministically or probabilistically for some small subset of inputs. On the other hand, they allow significantly higher throughput or lower power than their error-free counterparts. The integration, validation, and evaluation of these approximate units in architectures and processors, however, remains challenging. The lack of tools to represent and evaluate approximate hardware leads designers to verify their solutions independently, not considering interactions with other components, demanding high-effort modeling and simulation. In this work, we introduce *ADeLe*, a high-level language for the description, configuration, and integration of approximate hardware units into processors. *ADeLe* reduces the design effort for approximate hardware by modeling approximations at a high level of abstraction and automatically injecting them into a processor model for architectural simulation. In the *ADeLe* framework, approximations may modify or completely replace the functional behavior of instructions according to user-defined policies. Instructions may be approximated deterministically or probabilistically (e.g., based on operating voltage and frequency). To allow for controlled testing, approximations may be enabled and disabled from software. Energy is automatically accounted for based on customizable models that consider the potential power savings of the approximations that are enabled in the system. Thus, the framework provides a generic and flexible verification method, allowing for easy evaluation of the energy-quality trade-off of applications subjected to approximate hardware. We demonstrate the framework by introducing different approximation techniques into a processor model, on top of which we run selected applications. Modeling dedicated hardware modules, we show how *ADeLe* can represent approximate arithmetic and reduced precision computation units executing 4 image processing and 2 floating point applications. Using a different method of approximation, we also show how the framework is used to study the impact of voltage-overscaled memories over 9 applications. Our experiments show the framework capabilities and how it may be used to generate approximate virtual CPUs and to evaluate energy-quality trade-offs for different applications with reduced effort.

List of Figures

Contents

Chapter 1

Introduction

Approximate computing has emerged as a promising solution to design issues in the Dark Silicon era [Esmaeilzadeh et al., 2011]. As power dissipation becomes the limiting factor for further increasing clock frequency on integrated circuits [Borkar and Chien, 2011], the exploration of an energy-quality trade-off can potentially allow further energy downscaling in comparison to traditional techniques. In addition to diverse software-based approximation approaches, the design of approximated hardware modules has attracted attention [Mittal, 2016; Xu et al., 2016]. The lack of specific tools for approximate-hardware development, however, increases the effort of validating and evaluating such hardware.

The common hardware design approach to develop approximate hardware modules typically consists in running input- and output-constrained circuit-level simulation to obtain energy and time metrics, followed by modeling the module behavior using a higher-level language or tool to estimate the quality of results. The results are then validated in a benchmark application substituting parts of the code for the behavioral model [Mittal, 2016]. In this method, the target application is explicitly changed to introduce the approximations, which lacks generality, given that applying the approximation in a different application may require extensive modification. Furthermore, architecture-level interactions between hardware modules may not be captured by modeling the behavior of approximate hardware at the application level.

Alternatives to higher-level software modeling include software instrumentation and architecture simulation. The latter has the advantage of generically representing the interaction between components in the target hardware and being transparent to the final application. Full-CPU simulators or emulators such as Wattch [Brooks et al., 2000], gem5 [Binkert et al., 2011], MARSS [Patel et al., 2011] and ArchC [Rigo et al., 2004] can model the process behavioral execution at varied detail levels, but lack a design framework to aid in the injection of hardware approximations.

In order to offer a reduced-effort method to model approximate architectures, we propose *ADeLe – The Approximation Description Language* – a high-level descriptive modeling language for hardware approximation, and demonstrate its use by injecting approximations into an off-the-shelf CPU model [Rigo et al., 2004]. The *ADeLe* abstraction focuses on representing approximations in a set of generic self-contained models designed to be automatically consumed by a CPU simulator, according to a high-level

description. Thus, the effort of validating a new approximation design is reduced to designing the models and the description file. *ADeLe* eliminates the effort of modifying a CPU simulator directly, and the models allow reuse in multiple target architectures. The method also embeds customizable energy models to estimate energy consumption, at the designer discretion, using abstractions of physical parameters that may influence in the simulation flow, and a flexible control structure to isolate resilient areas of the test application to be subjected to approximations.

In this work, *ADeLe* was implemented as an extension to the ArchC framework [Rigo et al., 2004] to show how it translates into simulation tools, allowing the execution of benchmark applications with minimal modification to the original source code, and the generation of comprehensive quality and energy consumption results. The *ADeLe* framework generates verifiable, uniform, reproducible, and reusable energy-quality results, and we show that by expanding known results to a set of different target applications. Moreover, the generic standardization herein proposed allows a fair and easier comparison of different approximation techniques, currently limited by the usage of incompatible or undisclosed modeling details and energy metrics [Mittal, 2016; Xu et al., 2016].

Thus, in this work, we summarize the following contributions:

- A high-level modeling language to describe how approximations affect an application at the architecture level;
- A framework to compile and translate the language in a simulation tool;
- A set of software models to represent common hardware approximations;
- A demonstration of how different applications behave when subjected to approximations.

The remainder of this text is organized as follows: In Chapter 2, we introduce the Approximate Computing paradigm and summarize related work on simulation techniques. The *ADeLe* language and its design flow is presented in Chapter 3, describing how to use it to model approximations. Our experiments demonstrating the language and the implementation applicability to model approximate hardware are described in Chapter 4, which includes descriptions of the selected hardware approximations modeled and benchmarks executed, as well as the results obtained. We conclude in Chapter 5.

Chapter 2

Related Work: Approximate Computing in the *Dark Silicon* era

For decades since the first microprocessor was conceived, the forecast exponential growth on the number of transistors [Moore, 1998] within a chip has been sustained by miniaturization [Dennard et al., 1974]. A smaller transistor can achieve a higher frequency which, when associated to microarchitecture developments and better memory systems, led to the exponential improvement in performance of computing systems overall [Borkar and Chien, 2011]. The Dennard model [Dennard et al., 1974] also defines a lower supply voltage in the scaling procedure. Although this resulted in better energy efficiency at first, the open-circuit leakage current increases exponentially the lower is the supply voltage [Borkar and Chien, 2011]. As a result, the power dissipated by the circuit increases to a point at which packaging and cooling techniques cannot handle [Shafique et al., 2014b].

This limitation in power dissipation forbids a chip to be used at its full capabilities for a long period of time. Thus, a significant part of the system runs at a lower frequency, or is even left powered off, in a situation often referred in the literature as *Dark Silicon* [Shafique et al., 2014a]. In a 22 nm architecture, the area affected by *Dark Silicon* exceeded in up to two times earlier forecasts [Esmaeilzadeh et al., 2011; Kapadia and Pasricha, 2017], and this amount is going towards 80% of the chip for recent manufacturing processes [Shafique et al., 2014b].

Despite being a waste of resources, leaving silicon dark does not necessarily improves power efficiency. Especially in many-core architectures, active computing cores should be kept at a distance from each other to maintain the temperature distribution within the chip and reduce their influence in one another. However, the longer the distance between the nodes, the further away data need to propagate, which increases latency, reduces performance, and negatively affects efficiency. Moreover, a longer and slower dataflow requires faster computation nodes to maintain throughput, resulting in more power dissipation [Yang et al., 2017; Kapadia and Pasricha, 2017].

Approximate Computing has been studied as an alternative design to improve power efficiency without negative effects in performance. Instead, Approximate Computing explores a trade-off between energy and quality, since many applications do not require computation to be exact and precise all the time [Kugler, 2015]. Recent studies

have shown how Approximate Computing techniques can achieve energy savings and improve power efficiency both in the software and hardware levels [Xu et al., 2016; Mittal, 2016].

Software-level Approximate Computing techniques include precision scaling, loop perforation, memoization, task skipping, and function replacement [Mittal, 2016]. They all have in common the intent of reducing computation time, thus reducing energy consumption but not power dissipation. These software techniques are usually specific to a target application and need to be redesigned to others. As a result, they achieve better controlled quality results and limited energy savings [Chippa et al., 2014].

The hardware-level techniques, on the other hand, affect a wider range of applications. These include replacing functional units by simpler ones or adjusting operating parameters, mainly supply voltage, below the nominal level. These modifications have direct impact on power dissipation and usually allow extended impact on the hardware by fine tuning the operating parameters, thus the energy savings are significant and extensible throughout the project [Chippa et al., 2014]. Their impact on quality, since they are not specific, however, can even be unpredictable in advance, thus requiring further evaluation and simulation for validation. The following sections present an overview of hardware-level approximation techniques (Sec. 2.1) and existing tools that allow such validation (Sec. 2.2).

2.1 Hardware-level approximation techniques

2.1.1 Memory access approximation

Load value prediction is a well known technique to minimize the cache miss penalty on a memory access. Miguel et al. [2014] followed this idea to create a load value approximation, a technique that augments prediction by not rolling back the application execution if a value is incorrectly predicted. Instead, the actual value read is only used to train the system and improve prediction quality. Furthermore, the data are not fetched from memory at every cache miss, but at a determined rate, saving energy on memory fetch. The study shows up to 8.5% better performance and 12.6% energy savings on average.

Similarly, Yazdanbakhsh et al. [2016b] also propose rollback-free value approximation, breaking the memory fetch process for each cache miss. The system is optimized for graphic processors, maintaining data consistency for all processing cores. The authors report 36% better performance and average 27% energy savings. Both Miguel et al. [2014] and Yazdanbakhsh et al. [2016b] used adapted simulation software to evaluate their designs. The former considers every memory read within a code region as a cache miss, computing the approximation, while the latter relates the memory reads with hot memory regions.

2.1.2 Inexact hardware

Inexact hardware are modules developed, in logic level, to produce incorrect results at some determined input conditions, trading quality for energy and performance [Kahng and Kang, 2012]. The circuit is usually designed to reduce the critical path, allowing better performance and reducing energy consumption.

Kulkarni et al. [2011] propose an approximate multiplier hardware. The authors take advantage of the fact that there are only 8 possible products of 2-bit integers to create a module that outputs a 3-bit integer, instead of 4. In this concept, the result of the operation 3×3 is 7, not 9. All the other possible products can be represented in a 3-bit integer, so the output is correct in 15 out of 16 possible input patterns. Besides, the multiplier can be connected to others, in a Wallace Tree topology, offering larger number multiplication. Moreover, the system was designed with a error correction mechanism to be used if the application demands. The statistical analysis shows that the multiplier can save up to 45% energy at an average calculation error in the order of 3%.

Kahng and Kang [2012] present the design of an approximate adder. In the project, the adder carry chain is sliced to reduce the circuit critical path, allowing it to operate in a higher frequency and use less area, thus reducing power dissipation. The authors describe how to set up the adder precision and a possible auxiliary circuit for error correction, if necessary. The results show up to 24.6% throughput and 37% energy savings in comparison with a regular exact adder.

EvoApprox8b [Mrazek et al., 2017] is a library of approximate hardware adders and multipliers evolved by genetic programming. The objective is to offer a common point of comparison for benchmarking approximate circuits, a library containing almost a thousand approximate implementations of commonly used hardware. The authors report energy and quality metrics, obtained individually, for each of the alternatives, and disclose high-level software and hardware description models for them.

The design projects of inexact hardware modules commonly use computer aided design to validate timing and power characteristics. This method, although necessary and very precise, limits the validation to a narrow set of operations, making it unlikely the test of a full application, let alone real-world applications. As alternatives, Kulkarni et al. [2011] and Mrazek et al. [2017] remodeled their designs in high level software, which allows for timely feasible verification but limits the representation of the designs in association with other modules in the architecture level.

2.1.3 Voltage overscaling

The Dennard model [Dennard et al., 1974] describes the relation between a transistor size, threshold and supply voltage and maximum operating frequency. As power is proportional to voltage, it is possible to reduce power dissipation by voltage reduction. The voltage is said “overscaled” when it is adjusted below the operating point, which causes timing and switching failures [Chippa et al., 2014]. Depending on the consequences of such failures and the resiliency of the application, in the context of Approximate Computing, they could be referred as approximations.

Voltage overscaling is the technique used by Chippa et al. [2014] and Rahimi et al. [2015]. The former uses the technique to approximate the adders in a multiply-and-accumulate (MAC) system, causing timing constraint violation and output errors. The authors employ overscaling in association with other techniques, and results show that the energy savings achieved by voltage overscaling alone are very similar to those perceived when other techniques are used together for smaller error tolerances. The larger the tolerance, however, the worse it performs when compared with the other, still achieving 50% energy savings over precise computation.

Rahimi et al. [2015] also use voltage overscaling in association with other techniques. They propose an auxiliary memory module at each Floating Point Unit of a graphics processor. This module stores data from common computations, avoiding them to be recomputed, a memoization technique. The auxiliary unit itself has the supply voltage overscaled, reporting average energy savings of 32%.

Both works by Chippa et al. [2014] and Rahimi et al. [2015] use integrated circuits design software to obtain the power characteristics of their propositions. To allow real-world simulation, Chippa et al. [2014] wrote their own dedicated system simulator, while [Rahimi et al., 2015] modified an existing GPU simulator to extract input patterns, without including simulation features of the designed module. In both cases, an architectural simulator would allow the modules to be integrated in the system, offering results validation in real conditions.

2.2 Modeling techniques for Approximate Computing

Modeling abstractions and methodologies have been proposed in both software [Sampson et al., 2011; Carbin et al., 2013; Sampson et al., 2015; Barbareschi et al., 2017] and hardware [Rahimi et al., 2013; Nepal et al., 2014; Yazdanbakhsh et al., 2015] levels. They enhance the design flow of a technique identifying target code or circuit areas that may benefit from approximation and providing a model of modifications. However, such methodologies, particularly the hardware-level ones, are focused on representing the approximations themselves in a self-contained fashion, with limited modeling of their integration in a target system. Their validation uses the typical methodology to validate custom-design hardware modules, which involves high-effort time-consuming Register Transfer Level simulation in *ABACUS* [Nepal et al., 2014] and *Axilog* [Yazdanbakhsh et al., 2015] or customization of a high-level simulation tool to represent the system [Rahimi et al., 2013].

Simulation or instrumentation can be used to estimate the final behavior of an application, when a hardware module is modified, in association with the rest of the system. Fault injection simulators are tools of consolidated research interest to represent possible modifications in a system [Hsueh et al., 1997; Kooli and Natale, 2014; Kooli et al., 2015]. In the context of Approximate Computing, consolidated simulators such as *FERRARI* [Kanawati et al., 1992], *DOCTOR* [Han et al., 1995], *FTAPE* [Tsai et al., 1996], *Xception* [Carreira et al., 1998], *FAUmachine* [Potyra et al., 2007] and *LIFTING* [Bosio and Natale, 2008] are some of the options to simulate approximations injection.

Application-directed fault injection simulators, however, typically work at application level. Thus, the injection is limited to each use-case application, sometimes requiring it to be individually modified, and the scope is usually limited to hardware regions directly accessed by the software, such as memory and storage locations. Moreover, such tools provide information of the computation result, but do not provide an interface to extract measurements such as energy, limiting the approximation analysis to quality of results.

Instrumentation tools such as *Pin* [Luk et al., 2005] can be used to inject approximations at the application level, usually for the same platform that runs the software. The *React* framework [Wyse et al., 2015] uses Pin to introduce approximations into multiple applications, some of them representing approximate hardware, and to account energy using a simplified linear model. The framework, however, does not provide a generic or flexible modeling method to represent approximations, limiting its coverage mostly to the provided models.

System simulators such as *Wattch* [Brooks et al., 2000], *gem5* [Binkert et al., 2011] and *MARSS* [Patel et al., 2011] were already used by researchers to validate approximation techniques, although they do not offer a mechanism to directly modify the simulated hardware to inject approximations and are limited to a specific target architecture. The emulator *QEMU* [Bellard, 2005] has been used, in association with *SystemC*, as a virtual platform to represent a full system in hardware-software co-design [Monton et al., 2007; Yeh and Chiang, 2010; Chiang et al., 2011; Kleinert et al., 2016]. Thus, SystemC augments QEMU emulation capabilities adding customized hardware modules and creating a communication interface which allows high-performance emulation of fault-injected systems [Geissler et al., 2014; Ferraretto and Pravadelli, 2015; Höller et al., 2015]. The framework *VarEMU* [Wanner et al., 2013] extends QEMU with fault and power models to evaluate variability-aware software and supports the injection of faults into the emulated hardware. VarEMU and other QEMU-based techniques take advantage of binary translation to achieve high-performance verification. On the other hand, QEMU limits the representation of details in the architecture, such as multiple cores, thus the high performance comes at the expense of generality and control.

ArchC [Rigo et al., 2004], despite not performing as well as QEMU does to their target architectures, is an *Architecture Description Language* that can virtually represent any target system, allowing custom ISA extensions independently of any approximation injection mechanism. In association with SystemC Hardware Description Language support, ArchC can be expanded with custom functional units or peripherals at the designer discretion. The composition of *ADeLe* and ArchC as a framework provides a complete and general verification system, where ArchC represents a customized target CPU and its peripherals, and *ADeLe* handles approximation injection and approximation-aware simulation control, allowing full control and complete representation of the system in the verification process to designers.

Chapter 3

A Framework for Approximate Computing

The *ADeLe* language describes how approximations, associated with their energy models and probabilities, are injected into instructions in a CPU model. In this section, we describe a use case introducing the concepts and the approximation design flow using *ADeLe*. Our example targets a designer who wants to evaluate the impact of a new register bank design, powered by an adjustable supply voltage.

In the typical hardware development flow, the designer would model the hardware using a Hardware Description Language (HDL), evaluate it in a Register Transfer Level (RTL) simulator and synthesize it to obtain power and timing data, back annotating this information to the netlist for further simulation [Mittal, 2016]. At this point, the designer has data on how the register bank behaves under different voltage levels and characterization of soft errors when reading and writing data to certain locations, as well as their occurrence distribution [Tagliavini et al., 2017; Slayman, 2011; Calhoun and Chandrakasan, 2005]. In the context of Approximate Computing, errors under overscaled supply voltages may be seen as approximations that trade accuracy for energy savings in the computation [Chippa et al., 2014].

The HDL model and RTL simulation data alone cannot provide any information about how the new design impacts real applications. A CPU simulator may help in evaluating and validating the design, and its integration in a system. Nevertheless, extensive effort would be required to adapt the simulator in order to inject code that represents the faulty register bank, to develop a control mechanism that parameterizes it according to the supply voltage, and to aggregate the results of RTL simulation to represent energy metrics.

ADeLe proposes an enhanced design flow, summarized in Fig. 3.1, that eliminates the effort of modifying the CPU simulator source code directly. Taking, as inputs, a generic model of the approximation and a high-level description of how it interacts with the target CPU model, an *ADele*-compatible CPU simulator automatically modifies its execution flow to represent the approximated behavior.

The *ADeLe* design flow requires three *models*: the *approximation model* (Sec. 3.1.1) describes the black-box behavior of the approximation at instruction-level, in terms of its input data and output results; the *probability model* (Sec. 3.1.2) defines whether

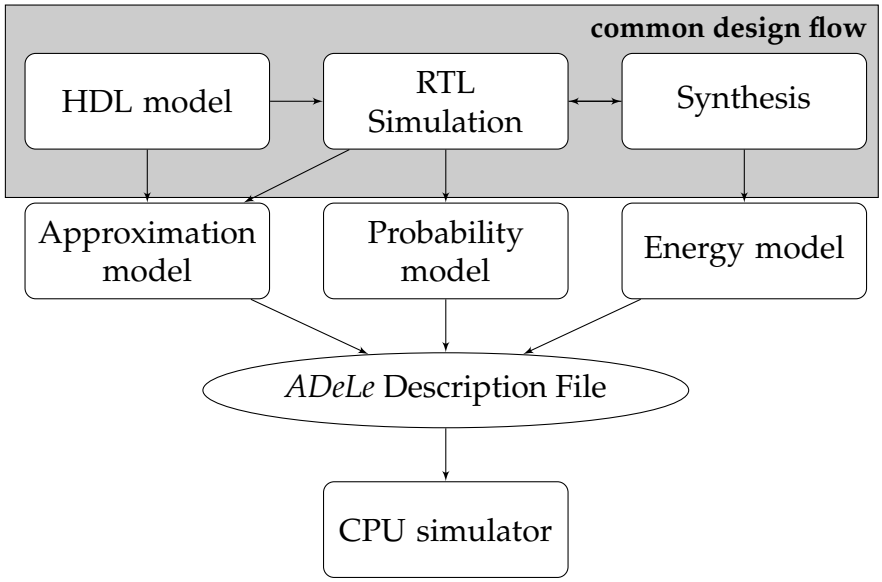


Figure 3.1: *ADeLe* design flow showing how it affects the CPU simulator

```
1 void RandomBitFlip(source_t source, word &data) {
2   int bit = rand() % (8 * sizeof data);
3   data    = data ^ (0x1 << bit);
4 }
```

Figure 3.2: Sample implementation of RandomBitFlip data modifier.

the described behavior should happen instead of the default execution behavior in the simulator; and the *energy model* (Sec. 3.1.3) computes how much energy was spent to execute the instruction, defined at the designer discretion.

In our use-case scenario described at the beginning of this Section, the designer would implement the approximation model as a method that takes as input the handled data and outputs it with some random modification, modeling the unexpected bit flips perceived during RTL simulation when the register bank is powered by a lower supply voltage [Calhoun and Chandrakasan, 2005] – in the *ADeLe* language, such an approximation model is called *data modifier* (Sec. 3.1.1). The sample in Fig. 3.2 illustrates the C++ implementation of a data modifier that flips a random bit in the data.

Similarly, the probability model uses the statistical fault occurrence distribution to determine whether an approximation is supposed to happen, given the applied supply voltage. Finally, the designer uses consolidated dynamic and static power models [Rabaey et al., 2002; Kim et al., 2003] to determine, in the energy model, the energy cost of each single instruction.

These models are generic and self-contained in the sense that they should not require nor use, by definition, any information specific to the target simulator or even to the target architecture. However, to represent the architecture and integrate all three models together, they share information encoded in a set of *operating parameters* (Sec. 3.1.4) that represent, for example, the current supply voltage, frequency and temperature of the system.

```
1  DM RandomBitFlip();
2  PM LowVddProbability();
3  EM DefaultEM();
4
5  OP default_op  = {voltage   = 1.0,    // V
6                    frequency = 400.0}; // MHz
7  OP low_vdd_op  = {voltage   = 0.8};   // V
8
9  energy     = DefaultEM();
10 parameters = default_op;
11
12 approximation LOW_VDD_REGBANK {
13   initial       = off;
14   parameters    = low_vdd_op;
15   regbank_read  = RandomBitFlip();
16   regbank_write = RandomBitFlip();
17   probability   = LowVddProbability();
18 }
```

Figure 3.3: *ADeLe* description of an adjustable supply voltage register bank.

All models and operating parameters are related to each other and to the target architecture, at instruction level, using the *ADeLe Description File*. Being a bridge between the generic, self-contained models and the target simulator, this description file is the only part of the design flow that is written specifically for a target.

Fig. 3.3 exemplifies the *ADeLe* Description File that represents the adjustable supply voltage register bank as an approximation that affects all instructions in the ISA. Lines 1-3 declare the methods that implement the three required models: the approximation model, as a data modifier (DM), the probability model (PM) and the energy model (EM). Lines 5-7 define two sets of operating parameters that represent the regular CPU execution and a lower voltage – and low-power – state. The energy model and parameters that should be used to compute the energy cost in non-modified executions are set in lines 9,10. Finally, lines 12-18 define how the approximation itself is composed: all instructions are injected with the data modifier implementation when reading or writing to the register bank with the probability defined by the probability model. Moreover, when the execution is approximated, the target processor uses the lower voltage set of operating parameters, which affects energy computation, as well as the other models.

To enhance control of the final application execution, *ADeLe* defines that the approximations may be activated or deactivated at execution time, avoiding that non-error-resilient sections of the application are submitted to approximation. An *ADeLe*-compatible CPU simulator generates, after processing the description file, a library containing code to be included into the test application. This library defines a set of constants representing each approximation defined in the *ADeLe* Description File and control functions to interface with the CPU simulator control mechanism. Fig. 3.4 il-

```
1 data_t *data;
2
3 data = acquire_data();
4
5 adele_activate(ADELE_APPROX_LOW_VDD_REGBANK);
6 data = computation_kernel(data);
7 adele_deactivate(ADELE_APPROX_LOW_VDD_REGBANK);
8
9 store_data(data);
```

Figure 3.4: Simulation control interface.

lustrates how a resilient computation kernel may be isolated from non-resilient data acquisition and storage operations, allowing just the kernel to be affected by approximations.

3.1 The ADeLe Language

The approximation representation proposed by *ADeLe* is composed of a set of models and a description file that relates them. The *ADeLe* Description File declares all the models to be injected into a CPU design, defines sets of operating parameters, groups instructions in the ISA to support approximation injection, and describes the approximations themselves. The model declarations (Fig. 3.3, lines 1-3) contain the signatures of the method implementations, prefixed with an identifier to classify them in approximation (DM or IM – Sec. 3.1.1), probability (PM – Sec. 3.1.2), or energy models (EM – Sec. 3.1.3).

The definition of operating parameters (Fig. 3.3, lines 5-7), identified by the *OP* keyword, is the main integration mechanism between models. Not all operating parameters in a set need to be explicitly defined, in which case *ADeLe* keeps the previous value of the given parameter unchanged. Sec. 3.1.4 details the operating parameters structure and its integration with the other models defined by *ADeLe*.

Regardless of whether approximations are activated in the final simulation, *ADeLe* requires the definition of a method to compute the energy spent by the simulated CPU regular operation. Thus, a default energy model and a default set of operating parameters need to be defined, such as exemplified in Fig. 3.3 in lines 9 and 10. These defaults are used at the beginning of the simulation and overwritten whenever an approximation that uses different configurations is activated.

Finally, the *ADeLe* Description File defines two types of groups: instruction groups and approximation groups. The instruction groups are arbitrary sets of instructions in the ISA that share some features in the description, for example, all floating-point instructions that deal with single-precision data may be subject to a different set of operating parameters that indicates that their energy cost is higher [Tong et al., 2000]. The example in Fig. 3.5 introduces the description of another approximation in which the CPU includes a Floating-Point Unit that adjusts itself to a lower-precision (IEEE

```
1  DM Float2HalfPrecision();
2  EM DefaultEM();
3
4  OP default_op  = {voltage   = 1.0,     // V
5                    frequency = 400.0}; // MHz
6  OP fpu_op      = {scaling   = 1.2};   // Scalar
7  OP half_fpu_op = {scaling   = 0.6};   // Scalar
8
9  energy     = DefaultEM();
10 parameters = default_op;
11
12 group SINGLE_FP { // Single-Precision Floating-Point
13   parameters  = fpu_op;
14   instruction = {add.s, sub.s, ...}; // List instructions
15 }
16
17 approximation HALF_PRECISION {
18   initial = off;
19   group SINGLE_FP {
20     regbank_read  = Float2HalfPrecision();
21     regbank_write = Float2HalfPrecision();
22     parameters    = half_fpu_op;
23   }
24 }
```

Figure 3.5: *ADeLe* description of half-precision FPU.

754-2008 Half Precision [IEEE, 2008]) low-power state, in which an instruction group (lines 12-15) defines that floating point instructions spend more energy than the other ones in the ISA, using a scaling factor [Tong et al., 2000; Ho et al., 2017].

The approximation groups – or simply approximations – define and name the approximations injected in the CPU simulator themselves. Each approximation has a default state (Fig. 3.3, line 13 and Fig. 3.5, line 18) that indicates whether the given approximation is activated or not at the simulator startup. This state may be changed by the final application at simulation execution time, thus selecting areas of the application that are resilient to approximations. The defined approximations can be injected into all instructions in the ISA (Fig. 3.3), one or more of the previously defined instruction groups (Fig. 3.5), or to specific instructions, at each case all the models and operating parameters applied to the instructions are selected. This configuration tells the simulator at which point the execution flow needs to deviate to an approximated behavior.

3.1.1 Approximation modeling

The methods that define approximation models are divided into two categories: data modifiers, that model alterations in the data used by an instruction, and implementation

modifiers, that change the execution behavior of the affected instruction. Data and implementation modifiers are identified by the keywords *DM* and *IM*, respectively, and can receive arbitrary data as parameters to model the approximation.

Data modifiers (DM)

These are methods that receive data accessed by an instruction and can apply some modification in the data. Data modifiers can be applied to read or write operations on any data source in the target architecture. When using data modifiers, the instruction is executed without any knowledge that the data was modified. These approximations are injected into instructions using the set of keywords <*source*>_<*operation*>, where *source* is the data source and *operation* is the data-handling operation. Fig. 3.5 exemplifies the inclusion of data modifiers in register bank read and write operations (lines 20 and 21). This approach can be used to model, for example, incorrect reads in the register bank due to voltage overscaling [Tagliavini et al., 2017; Slayman, 2011; Calhoun and Chandrakasan, 2005], data precision tuning [Ho et al., 2017; Sampson et al., 2011] and other memory access approximations [Miguel et al., 2014; Yazdanbakhsh et al., 2016b; Ganapathy et al., 2015].

The *ADeLe* description example in Fig. 3.5 declares the data modifier *Float2HalfPrecision* and then associates it with a group of instructions that contains all single-precision floating-point operations (lines 19-23), provided that the respective approximations are activated at execution time. The implementation of *Float2HalfPrecision* (Fig. 3.6) uses bitwise operations on floating-point values to read and write single-precision data as half-precision [IEEE, 2008]. Thus, the floating-point operations can be executed by the original models defined in the CPU modeling, but the results are taken as if the model had a half-precision floating-point unit.

A data modifier method implicitly receives a data structure containing information about the source of the data it is modifying and a reference to the data itself. The source of the data is encoded in a data structure containing the type of the data source (memory, register bank, special registers), the name of the data source (in case of multiple sources of the same type), the address of the data (in case of memory or register bank) and the operation being performed (read or write). This information may be used to determine whether the approximation should be applied to the data or not. In the data modifier illustrated in Fig. 3.6, for example, the data modifier is only targeted at the register bank called "RBF", representing a register bank dedicated to storing floating-point data. Thus, although all instructions that deal with floating-point data are affected by the approximation, only operations in the floating-point register bank are actually modified.

Implementation modifiers (IM)

Instructions in the CPU model are usually represented as methods that implement their behavior, performing the action that the operation would execute on hardware. Implementation modifiers augment this behavioral description by adding code before

```
1 void Float2HalfPrecision(source_t source, word &data) {
2     if (source.type == REGBANK && source.name == "RBF") {
3     // Bitwise operations to convert data
4         data = BitwiseOps(data);
5     }
6 }
```

Figure 3.6: Example of a data modifier.

the original model (*pre-behavior*), after (*post-behavior*), or replacing completely the behavior by another function (*alt-behavior*). *AdeLe* associates implementation modifiers to instructions using the keywords *pre_behavior*, *post_behavior* and *alt_behavior*, indicating the model that implements the approximation and its parameters, which are data objects available in the processor model scope, such as registers and register bank.

Alt-behavior models apply to dedicated hardware approximation techniques, such as approximate arithmetic [Lau et al., 2009; Kahng and Kang, 2012; Kulkarni et al., 2011; Camus et al., 2015; Mrazek et al., 2017], and floating-point [Lee et al., 2009; Camus et al., 2016b,a; Yin et al., 2016] modules, that replace (or replicate) functional units in the architecture by approximate counterparts, thus generating, in a modeling perspective, behaviors that are similar but cannot be represented solely by modifications in the data. Pre- and post-behavior models can be applied combined with alt-behavior, when some common procedure needs to take place before or after the approximation to better represent the results and support code maintainability, or individually, when the approximation affects operands or results and the instruction itself is executed following its original behavior, similar to a data modifier, but more specific to a single instruction.

The example in Fig. 3.7 injects Kulkarni's multiplier [Kulkarni et al., 2011] into integer multiplication instructions. The multiplier implementation receives the operand data from the register bank and references to the target registers where the product is written to. We took advantage of C++ method overloading to represent Kulkarni's method for both 32-bit and 64-bit (divided into halves) results, according to the MIPS ISA specification. Considering that Kulkarni's multiplier is unsigned, sign extension needs to be performed in order to correctly represent signed multiplication with signed operands, which is modeled as a post-behavior for the signed multiplication instruction (line 21).

3.1.2 Probability models (PM)

The method that implements a probability model, identified by the keyword *PM*, should return a boolean defining whether or not the approximation, at an instruction level, should take place. Probability models, as all other methods defined by *ADeLe*, have access to the operating parameters data structure, that can be used to determine the probability. For example, consider the function *LowVddProbability* in Fig. 3.3 (line 2): a lower supply voltage may increase the probability of a data-modifier approximation

```
1  IM Kulkarni(word a, word b, word &hi, word &lo);
2  IM Kulkarni(word a, word b, word &r);
3  IM SignExtend(word a, word b, word &hi, word &lo);
4  EM DefaultEM();
5
6  OP default_op  = {voltage   = 1.0,    // V
7                    frequency = 400.0}; // MHz
8  OP kulkarni_op = {scaling   = 0.7};   // Scalar
9
10 energy     = DefaultEM();
11 parameters = default_op;
12
13 approximation KULKARNI_MUL {
14   initial = on;
15   instruction multu {
16     alt_behavior  = Kulkarni(RB[rs], RB[rt], hi, lo);
17     parameters    = kulkarni_op;
18   }
19   instruction mult {
20     alt_behavior  = Kulkarni(RB[rs], RB[rt], hi, lo);
21     post_behavior = SignExtend(RB[rs], RB[rt], hi, lo);
22     parameters    = kulkarni_op;
23   }
24   instruction mul {
25     alt_behavior  = Kulkarni(RB[rs], RB[rt], RB[rd]);
26     parameters    = kulkarni_op;
27   }
28 }
```

Figure 3.7: *ADeLe* description of Kulkarni's multiplier.

affecting the register bank [Calhoun and Chandrakasan, 2005; Tagliavini et al., 2017], thus a function that accounts for supply voltage when rolling the dice to determine the occurrence may be implemented as the probability model for this approximation.

The probability model is coupled to an approximation using the keyword *probability*, which may be included either within the approximation group, when the same probability applies to all instructions that receive the approximation, or within an instruction block. Considering that probability models represent the likelihood of an approximation, they are not always required. While in the example of the adjustable voltage register bank (Fig. 3.3, line 17) the model decides if the lower supply voltage should affect the register bank operation at a given time, in the following half-precision FPU (Fig. 3.5) and Kulkarni multiplier (Fig. 3.7) examples, the occurrences are deterministic – the results are always approximated – and the probability model is not required.

3.1.3 Energy models (EM)

The methods that implement energy models, as defined by *ADeLe*, can take arbitrary parameters (such as a functional unit input operands), as well as the operating parameters, and return a number that represents, in Joules, how much energy one single instruction execution uses. Such flexible approach when computing energy allows designers to model their approximate circuits consumption at various levels of simplification. That is, the energy model abstraction can accommodate, but not be limited to, simplified models that assume every instruction to contribute equally and proportionally to the energy consumption [Tiwari et al., 1996; Guedes et al., 2013; Gupta et al., 2010; Jaiantilal et al., 2010], consolidated models that take into account physical variables such as voltage and frequency [Rabaey et al., 2002; Kim et al., 2003], common assumptions that energy is determined by the functional units involved in computation, disregarding control structures [Kulkarni et al., 2011; Camus et al., 2015; Kahng and Kang, 2012], or even fully customized models targeted at specific scenarios to represent precisely the energy consumption of a real system [Isci and Martonosi, 2003; Bertran et al., 2012].

Regardless of the chosen modeling approach, the reliability of the computed energy consumption resides on the model, and not in the abstraction proposed by *ADeLe*, and thus care must be taken to adequate and evaluate the model for the targeted scenario. At this point, *ADeLe* still contributes as a powerful tool, in a sense that models can be easily replaced at a high-level description and tested against multiple scenarios, at the designer discretion.

Energy models are declared using the *EM* keyword within the *ADeLe* description file. Models can be associated with instruction groups (where they are used for those instructions regardless of an active approximation), approximation groups, or approximated instructions, using the *energy* keyword. Considering that the simulator needs to compute energy for the whole execution, an energy model needs to be associated outside any groups, approximations or instructions in the description, and it is taken as the default energy model when no other is set.

3.1.4 Operating parameters (OP)

The operating parameters structure represent a collection of physical conditions in which the simulated processor is running, such as supply voltage, operating frequency, and temperature. The operating parameters represent the primary integration mechanism between all three types of models defined by *ADeLe*. For example, the voltage at which the CPU is running can be used to determine whether an approximation should happen, in the probability model, compute the energy used by the operation, in the energy model, and define what parts of the execution are affected and how, in the approximation model, affecting multiple implemented approximations at the same time [Mineo et al., 2016; Gautschi et al., 2016; Tagliavini et al., 2017; Calhoun and Chandrakasan, 2005].

Considering that operating parameters are dependent on the implemented approximations, they can also be changed at execution time. This can, for example, represent dynamic voltage-frequency scaling (DVFS) features in the simulated CPU and trigger, using probability and approximation models, different behaviors in the functional units due to overscaling [Chippa et al., 2014].

The operating parameters are declared using the *OP* keyword, and associated with groups of instructions (Fig. 3.5, line 13), approximations (Fig. 3.3, line 14), instructions (Fig. 3.7, lines 17, 22, 26), or as the default parameters using the *parameters* keyword.

3.2 *ADeLe* implementation into a CPU simulator

ADeLe was designed to represent approximations according to their behavior in the target architecture, resembling a functional simulator. To implement these concepts, we extended the simulation tools in the ArchC framework [Rigo et al., 2004] with the VArchC module, that allows the generation of *ADeLe*-compatible simulators. Based on the SystemC Hardware Description Language, the ArchC language generates processor models written in C++ by describing the behavior of each instruction in a customized software method, and thus *ADeLe* is a good fit for these models. Moreover, the integration with the ArchC framework allows the representation of other target architectures, by modeling them in the ArchC abstraction, and associated peripherals, in their SystemC description, creating an extensible and generic framework. Although ArchC is a convenient framework to represent a generic simulator, the proposed extension modifies common structures in functional simulators – the instruction decoder and storage accesses – and could similarly by applied to other simulators. Fig. 3.8 shows a simplified class diagram of a original ArchC CPU model and, highlighted, the VArchC extensions.

The ArchC Language describes processor models according to their structure and the behavior of the instructions. The structural information contains declarations for higher-level architecture characteristics, such as word size, number of words fetched from memory at each instruction, instruction formats, register bank size and memory connectivity. These provide the required information for a behavioral description of the instructions [Rigo et al., 2004]. Each instruction in the ISA is represented by a C++

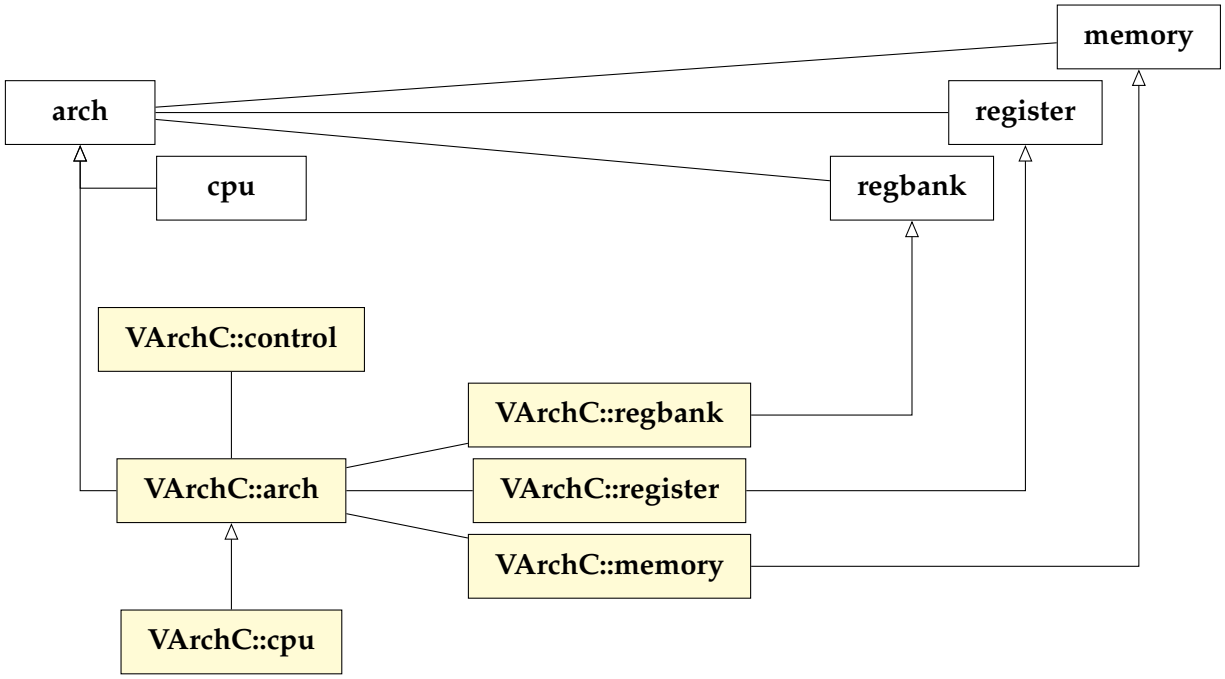


Figure 3.8: Simplified class diagram of ArchC extension.

method describing the operations executed, which the simulator instruction decoder calls when the corresponding instruction is executed.

This model has a straight-forward relation with the modeling of implementation modifiers using *ADeLe*. The instruction decoder was modified to deviate from the instruction default behavior. It verifies whether an implementation modifier is activated for the given instruction, using internal control structures, and runs the probability model. If an approximation is activated for the given instruction, it dispatches function calls to pre-behavior, the default behavior, alt-behavior, and/or post-behavior accordingly, as well as to the energy model.

Storage structures, such as memories, register banks and dedicated-purpose registers in the architecture, are modeled, in ArchC, using specific C++ classes that implement their function. Then, each storage structure in the CPU model is represented as an instance of the class that models the respective structure. To inject data modifiers, we use approximation-enabled C++ classes that interface with the original storage classes, and use method and operator overloading to produce approximate reads and writes, calling the selected data modifier implementation method for each operation. This modeling approach is transparent to the original instruction behavior description, in a sense that data accesses are still performed the same way, not requiring any modification.

3.2.1 Software interface

ADeLe defines that approximation groups may be enabled or disabled at execution time, to isolate, in the application, regions that are resilient and more likely to benefit from the energy-quality trade-off. However, to allow such feature, a communication interface

was defined between the final application and the CPU model, controlling the internal structures that define which approximations are enabled for which instructions.

In the current ArchC-based implementation, this interface is defined through a memory-mapped hardware register. In the application side, an auto-generated library header defines a pointer to the hardware register and methods that allow control over the simulation, activating and deactivating approximations. The simulator recognizes that the application uses the hardware register and triggers the control subroutines whenever an instruction accesses the pointer.

The memory-mapped hardware register alternative has the advantages of being transparent to the final application and independent of architectural characteristics. However, since the control relies on memory operations in the application side, the control subroutine may be affected by approximations applied to memory operations. To avoid such unlikely situation, the *ADeLe* framework offers the alternatives of creating a dedicated instruction for control or using a set of existing instructions with pre-defined operands. These alternatives improve the simulation control, but may require further architectural modeling and compiler support in the application level.

Chapter 4

Demonstration and results

To demonstrate how the *ADeLe* framework can be employed in the validation of Approximate Computing techniques, we built two simulation scenarios. In the first one, *underdesigned hardware* (Sec. 4.1), we show how to model modified functional units in a processor, and how such modified hardware affects common applications [Felzmann et al., 2018b]. The second scenario, *voltage-overscaled memories* (Sec. 4.2), brings a adjustable-voltage memory architecture in which we show how various supply voltage levels affect the execution of multiple applications from various computing domains [Felzmann et al., 2018a].

We executed our experiments on an ArchC-generated MIPS processor model. The MIPS model (v. 2.4.0) and ArchC software (v. 2.4.1), both compiled with GCC 4.9.2, are available at the ArchC webpage. Benchmarks were cross-compiled using ELLCC v. 0.1.34, with target set to big-endian, hard-float 32-bit MIPS.

4.1 Underdesigned hardware

In *underdesigned hardware*, we use the *ADeLe* framework to simulate a processor that had two functional units, the hardware multiplier and the Floating Point Unit, replaced by approximate counterparts. The multiplier replacements were five selected alternatives from the EvoApprox8B library [Mrazek et al., 2017], over which we executed four image processing applications. The FPU was configured to operate in Half Precision [IEEE, 2008] to run two floating point applications. The results show significant error saving with negligible quality degradation for most individual cases.

4.1.1 Implemented approximations

Approximation techniques were selected to cover both integer and floating-point applications and model instruction and data modifiers. Due to their energy consumption, time restrictions and perceived resiliency to approximations we target multiplication and floating-point instructions [Kulkarni et al., 2011; Mrazek et al., 2017; Camus et al., 2016b; Yin et al., 2016].

```
IM Evo479(word a, word b, word hi, word lo, bool sign);
IM Evo479(word a, word b, word r);
EM SimpleEM();

OP default_op = {scaling = 1.0000};
OP evo479_op  = {scaling = 0.7005};

energy     = SimpleEM();
parameters = default_op;

approximation EVOAPPROX_479 {
  initial = off;
  instruction multu {
    alt_behavior  = Evo479(RB[rs], RB[rt], hi, lo, false);
    parameters    = evo479_op;
  }
  instruction mult {
    alt_behavior  = Evo479(RB[rs], RB[rt], hi, lo, true);
    parameters    = evo479_op;
  }
  instruction mul {
    alt_behavior  = Evo479(RB[rs], RB[rt], RB[rd]);
    parameters    = evo479_op;
  }
}
```

Figure 4.1: *ADeLe* description of one EvoApprox8B instance.

EvoApprox8b multipliers

EvoApprox8b [Mrazek et al., 2017] is a benchmarking library of approximate 8-bit adders and multipliers evolved by genetic programming. The authors provide Verilog HDL descriptions and C models for all units in the library. We consider only the approximate multipliers, and built 32-bit modules from partial products using a Wallace Tree architecture. Energy savings were estimated using Cadence RTL Compiler over a baseline multiplier built completely from the tool optimization of a Verilog regular multiplication operation. The approximated multiplication instruction was represented as an implementation modifier, as illustrated in Fig. 4.1, which shows the representation of one multiplier instance in the library (479). The overloaded methods *Evo479()* build the Wallace Tree from multiple copies of multiplier instance 479 to allow 32-bit multiplication. The Energy Model *SimpleEM()* returns the scaling operating parameter as the instruction energy cost, thus providing a relative energy counter.

Single- to half-precision floating-point

Both operands and results of single-precision operations are converted to half-precision according to the IEEE 754 pattern [IEEE, 2008]. Tong et al. [2000] analyzed the energy

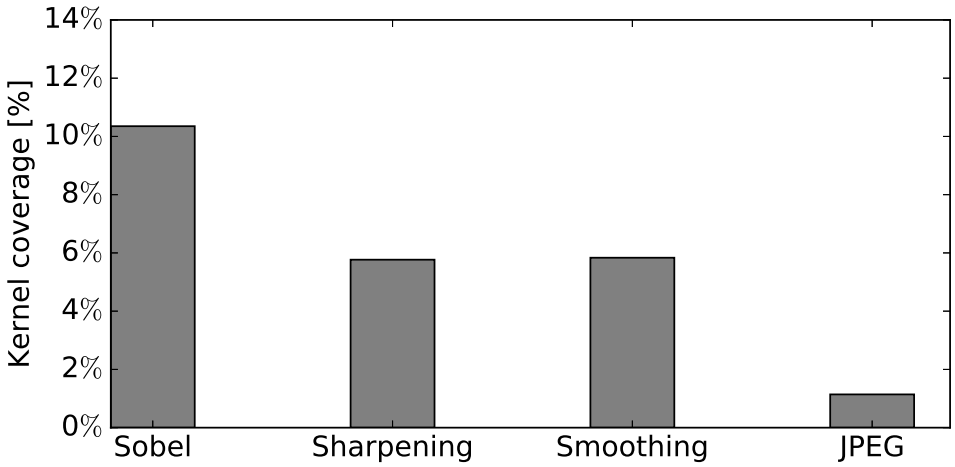


Figure 4.2: Percentage of kernel multiplications approximated.

consumption of multiple bitwidth floating-point architectures and reported 30% energy savings when using 16-bit operands and half-precision compatible significand fields. The approximated half-precision floating-point operations were represented as a data modifier, as described in the example in Sec. 3.1.1, Fig. 3.5.

4.1.2 Selected benchmarks

To represent the selected classes of approximations, applications on Image Processing – that usually rely on integer arithmetic, multiplication included – and floating point computation were selected and adapted from the AxBench [Yazdanbakhsh et al., 2016a] benchmark suite, as well as other freely distributed software. Image Processing applications were executed on 26 images from the USC-SIPI image database [SIP, 1997].

Image Processing

The selected Image Processing applications were the *Sobel edge detection* algorithm, as employed in [Wanner et al., 2013], the Gauss filter implementation for *Image Sharpening* and *Image Smoothing*, as described in [Lau et al., 2009] and [Kulkarni et al., 2011], and the JPEG compression algorithm from AxBench [Yazdanbakhsh et al., 2016a]. In the convolution-based algorithms (Sobel, Sharpening, and Smoothing), we replaced the multiplication in the convolution for selected approximate multipliers from EvoApprox8b [Mrazek et al., 2017]. For JPEG, AxBench provides additional information, in the source code, about which data structures are resilient to approximations, so we approximate all the multiplication operations where products were written to these structures. Fig. 4.2 shows the number of approximate operations relative to the application kernel. The resulting images for all approximate applications were compared with accurate executions using a usual Peak Signal-to-Noise Ratio (PSNR) metric.

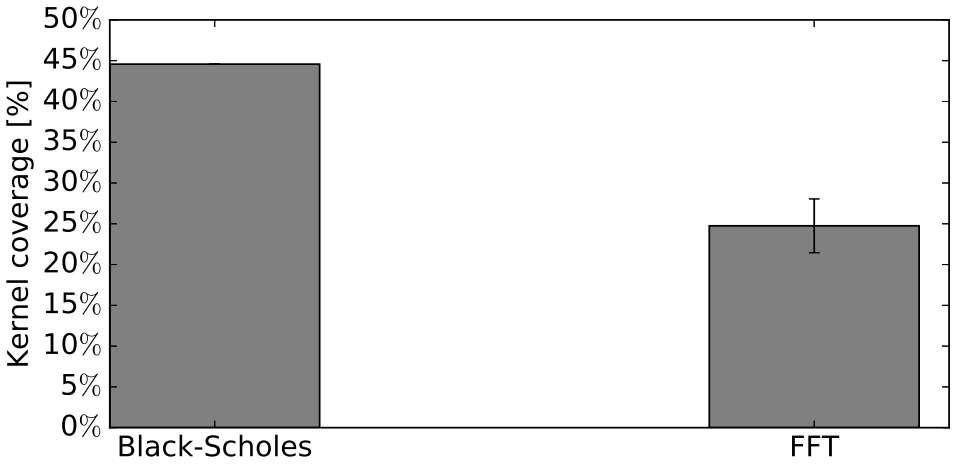


Figure 4.3: Percentage of kernel floating point operations approximated.

Floating-point

Two applications were selected from the AxBench [Yazdanbakhsh et al., 2016a] benchmark suite: *Black-Scholes* option price computing and *Fast Fourier Transform*. In both applications, we replaced all single-precision floating-point operations with half-precision operations (Fig. 4.3), truncating the operands with a data modifier as discussed in Sec. 3.1.1. Black-Scholes was executed over one thousand options, generated using the input generator from PARSEC [Bienia, 2011] suite, and results were evaluated according to the absolute error. The FFT execution was repeated 50 times over vectors of length 1K to 64K random values each, and results compared using the average relative error between the accurate (single-precision) and inaccurate (half-precision) computations.

4.1.3 Results

Applications were executed in the simulator in order to obtain the quality of results and relative energy consumption, comparing accurate and inaccurate computations. Except for the JPEG application, Image Processing applications showed resiliency to relative error rates on integer multiplication from 2% to 5%, allowing energy savings at a visually indistinguishable loss in quality. The floating-point approximation achieved higher instruction coverage than the integer one in the tested applications and thus exhibited a higher potential to save energy at low error rates.

Quality – Image Processing

EvoApprox8b [Mrazek et al., 2017] is an extensive library, thus we selected a subset of the multipliers based on their theoretical, non-specific, energy-quality trade-off. After expanding the original 8-bit modules to 32-bit ones in a Wallace Tree organization, we used the software models distributed in the library to describe the same multiplier organization in software and evaluate the mean relative error for all multipliers, using an extensive uniform dataset. Fig. 4.4 shows the energy-quality trade-off for the mul-

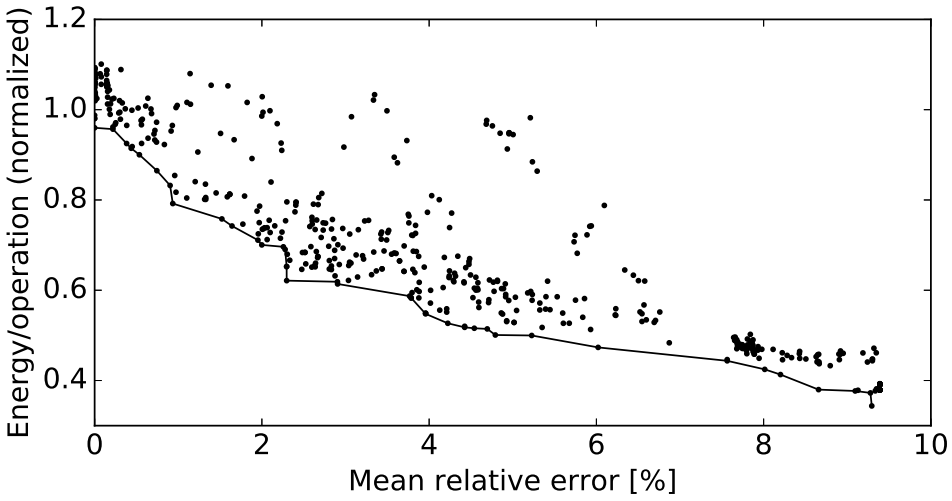


Figure 4.4: Energy-quality trade-off of 32-bit multipliers.

Table 4.1: Selected EvoApprox8b multipliers.

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

tipliers, where energy per operation is normalized at the consumption of a multiplier fully optimized by the synthesis tool from the regular HDL multiplication operator.

At higher quality metrics, the approximate multipliers tend to have higher energy consumption than the baseline. This shows the energy overhead of associating multiple smaller multipliers to build larger ones. However, as quality decreases, it is clear that the architectures consume less energy, achieving savings of up to 60%. In addition to the baseline accurate multiplier, we selected the five multipliers that achieve the lowest energy consumption at given maximum error metrics, and ran the benchmark applications after modeling their behavioral description in our processor model. The selected multipliers are summarized on Table 4.1.

Fig. 4.5 shows the resulting image quality after computation. Except for the JPEG application, PSNR was calculated between the accurately computed image and the approximated one. For JPEG, the images were compared directly with the original uncompressed image. Fig. 4.6 demonstrates the image outputs.

The approximate multipliers performed particularly well in the algorithms based on the Gauss filter – Sharpening and Smoothing. Using *mul8_303*, 10 out of 26 are identical to the accurately computed ones for Image Sharpening. For Image Smoothing, the results show that up to 1% mean relative error in multiplication has no effect on the resulting image. Higher error rates, despite lowering the PSNR, are still visually indistinguishable (Figs. 4.6b and 4.6c).

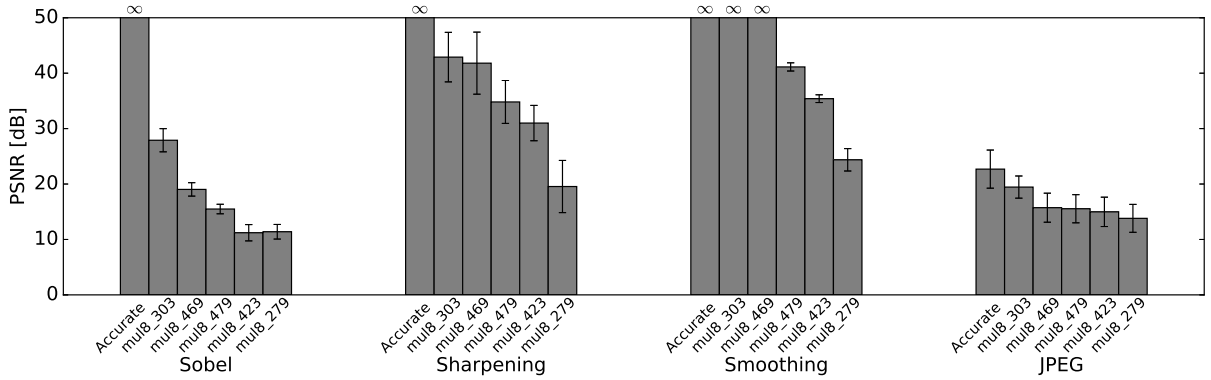
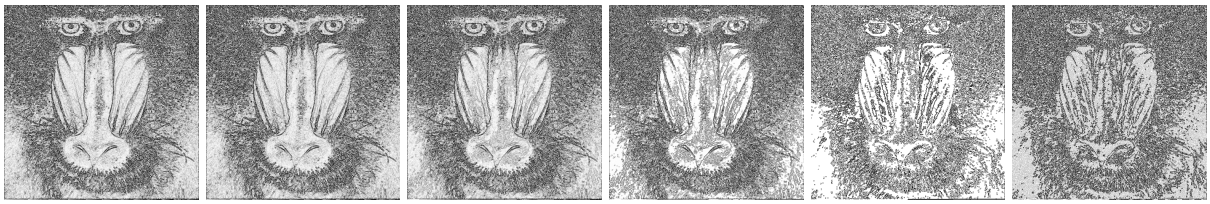
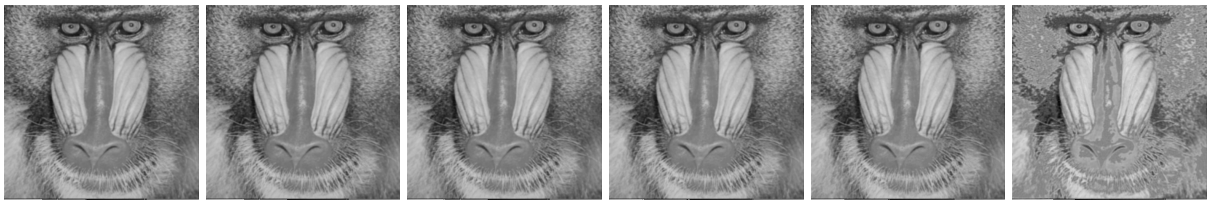


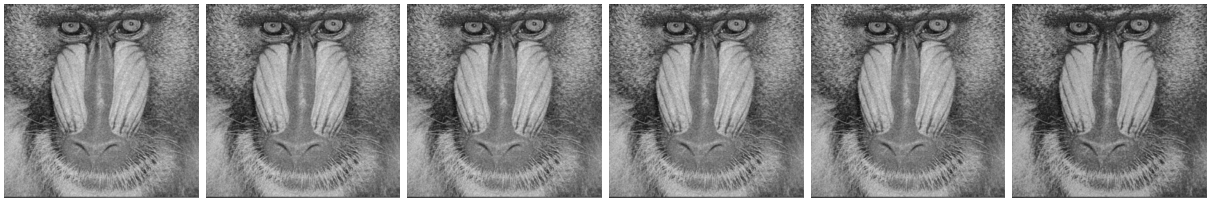
Figure 4.5: Peak Signal-to-Noise Ratio of computed images.



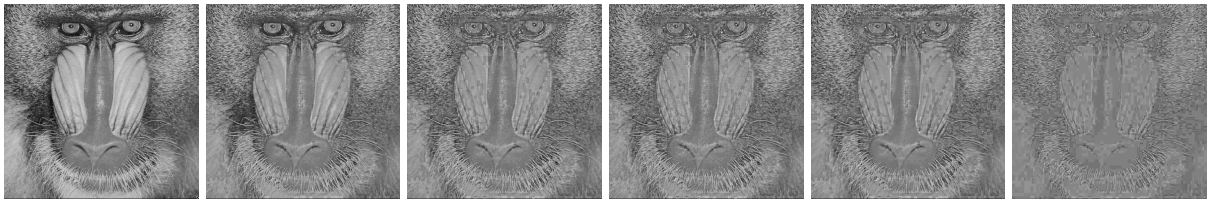
(a) Sobel



(b) Sharpening



(c) Smoothing



(d) JPEG

Figure 4.6: Image Processing applications: Accurate and inaccurate multiplications.

The other convolution-based algorithm, Sobel edge detection, despite presenting smaller Signal-to-Noise ratio than the Gauss-based ones, also showed some visual resiliency to approximations, as demonstrated in Fig. 4.6a. JPEG compression was the least resilient application (Fig. 4.6d), even though multiplication represents only 1.1% of the computation kernel (Fig. 4.2).

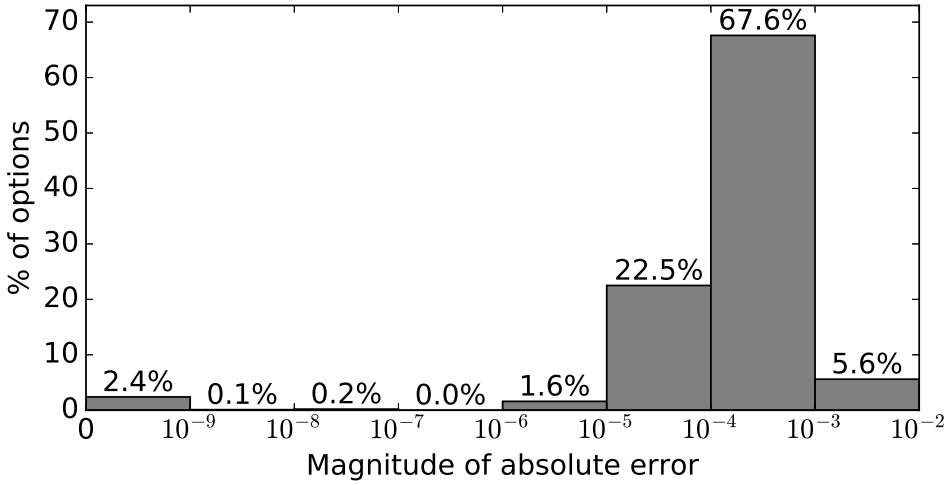


Figure 4.7: Black-Scholes: Average relative error.

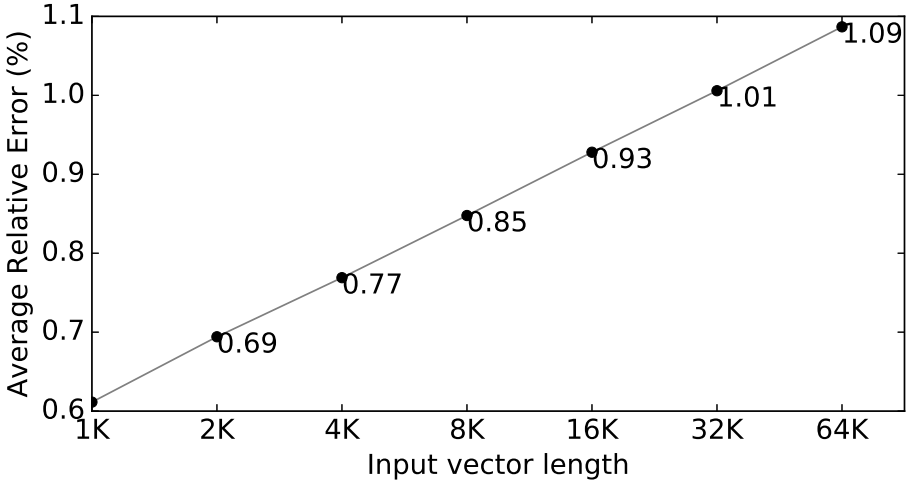


Figure 4.8: Fast Fourier Transform: Average relative error.

Quality – Floating-point

For Black-Scholes, approximations in floating-point instructions covered 44.6% of the instructions in the applications kernel (Fig. 4.3). The maximum absolute error observed was in the order of 10^{-2}, and 2.4% of the options resulted in a value identical to the accurately computed ones. The majority (90.1%) was in an error range from 10^{-5} to 10^{-3} (Fig. 4.7). PARSEC's input generator [Bienia, 2011] just replicates the default 1000 options input set to generate larger ones and the modeled approximation is deterministic. Therefore, increasing the size of the input set has no effect on the quality of results.

In the Fast Fourier Transform application, the approximated floating-point instructions represent 24.8% of the application kernel (Fig. 4.3). All the approximate computations are in a range ±1% relative to the accurately computed equivalents. Fig. 4.8 also shows that the relative error increases logarithmically with the size of the input set. The small relative error demonstrates the resiliency of FFT to this kind of approximation.

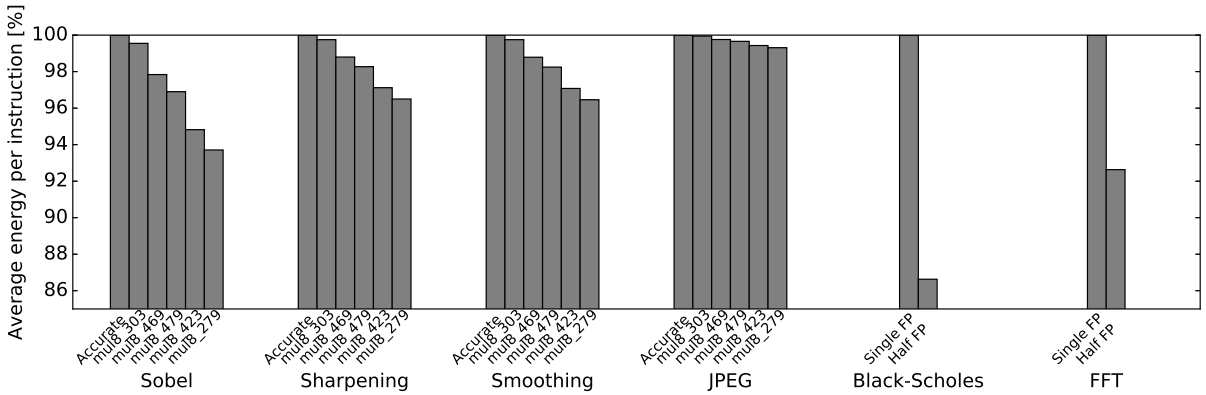


Figure 4.9: Effect of approximations on average energy per instruction.

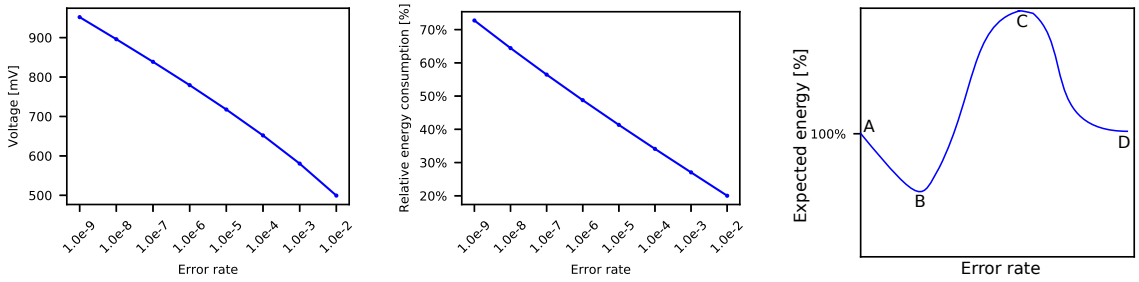
Energy consumption

To compute energy, we use a simplified energy model that defines the energy consumption of all instructions as uniformly spent by the functional units used on computation, as an approximation for a scalar in-order microarchitecture [Guedes et al., 2013]. Thus, the energy savings in a single instruction is proportional to the savings in the executed operation, so the approximation lowers the average energy-per-instruction consumption of the application as a whole according to the approximation coverage in the computation kernel. Using *ADeLe*, this behavior is easily reproduced by changing the scaling operating parameter (Section 3.1.4). Although *ADeLe* can represent more complex and accurate energy models, we adopt such a simplified alternative for a plain comparison, considering it is commonly used in approximate hardware verification [Kulkarni et al., 2011; Camus et al., 2015; Kahng and Kang, 2012].

Fig. 4.9 shows how approximations affect the average energy per instruction relative to the accurate computation. As expected, the applications in which approximated instructions present higher coverages in the application kernel benefit more on energy savings, particularly the floating-point applications.

The Image Processing applications presented lower energy savings mostly due to the low coverage that multiplication instructions represent in the computation kernel. The convolutions nested loops impose a control overhead, lowering the relative number of calculations executed in the kernel. The multiplication operation itself, however, can potentially save up to 30% energy at a 2% mean relative error (Table 4.1). This suggests that more multiplication intensive applications can potentially benefit more from such approximations.

Furthermore, simpler hardware for both integer and floating-point approximations can potentially use a lower voltage supply, which lowers the energy consumption [Borkar, 2016; Chippa et al., 2014]. Considering that both the integer multiplication hardware and the floating-point unit may be in the processor critical path, the lower supply voltage could be applied not only to the particular module, but to the CPU as a whole, allowing even lower energy consumption [George et al., 2006]. Using *ADeLe*, this situation can be easily modeled with the proper energy model by changing the



(a) Operating voltage (b) Energy (c) Energy trade-off

Figure 4.10: Relation between memory errors and energy.

value of the voltage operating parameter (Sec. 3.1.4), making straight-forward energy estimations possible.

4.2 Voltage-overscaled memories

In our second scenario, *voltage-overscaled memories*, we use the *ADeLe* framework to demonstrate the energy-quality trade-off in multiple applications when their executions are subjected to error-susceptible memories. We injected three different types of errors – BitFlip, StuckAt(0) and StuckAt(1) – in memory read and write operations at given error rates and estimated a supply-voltage level according to a statistical model in the literature [Wang and Calhoun, 2011]. Figs. 4.10a and 4.10b show the relation between the error rate and operating voltage and energy.

The occurrence of memory errors can cause the application to crash, in which case that no output is computed, or to produce low-quality results. To recover unusable results we define a mechanism based on error-free re-execution, such as in [Carlisle and George, 2018] and [Khudia et al., 2015]. Fig. 4.10c shows the average expected energy cost of computation, considering that some results may require re-execution. At the start point *A*, the error rate is so low that it compares to an error-free, precise, computation, with equivalent energy consumption. As the error rate rises, some energy is saved until the equilibrium point *B*, where the most energy is saved and few re-executions are needed. The more re-executions, the more energy is spent with them, until the maximum point *C*, where many approximate executions complete in a lower than acceptable quality. Finally, execution crashes cause the premature end of some execution instances, reducing the energy cost until point *D*, where re-execution is dominant.

Our experiments show the optimal equilibrium point between error rate and energy for 5 out of 9 test applications that represent common tasks in computing systems. The results demonstrate the energy profile of each application and the energy-quality trade-off, with up to 30% energy savings for a 80% quality threshold.

```
1  DM BitFlip();
2  EM OverscaledEM();
3  PM OverscaledPM();
4
5  energy = OverscaledEM();
6
7  approximation BITFLIP_MEM {
8    initial       = on;
9    probability   = OverscaledPM();
10   mem_read      = BitFlip();
11   mem_write     = BitFlip();
12 }
13
14 approximation BITFLIP_REG {
15   initial       = on;
16   probability   = OverscaledPM();
17   regbank_read  = BitFlip();
18   regbank_write = BitFlip();
19 }
```

Figure 4.11: *ADeLe* description of one EvoApprox8B instance.

4.2.1 Implemented approximations

This scenario is based in a hypothetical low-power embedded processor. This conceptual model implements a set of "approximation states" that directly influence the supply voltage of both the register bank and data memory. Each approximation state is associated with an error rate, or probability of occurrence of one error during a read or write operation [Wang and Calhoun, 2011].

To represent the approximation states, all read and write operations in both register bank and memory were replaced by an augmented software model using an *ADeLe* data modifier (Sec. 3.1.1). The software model select one random bit in the data word and applies one of three modifications: a bit flip, a stuck at zero, or a stuck at one fault. Fig. 4.11 shows a sample *ADeLe* Description File that injects bit flips in both memory and register bank operations. The sample abstracts the Energy and Probability models, that are implementations of the method in [Wang and Calhoun, 2011].

4.2.2 Selected benchmarks

Nine different applications were selected to represent a set of common elements in embedded systems. For each application, the code in the execution kernel was isolated, using the *ADeLe* software control interface, and the errors were applied only to this region. Thus, the input and output operations, which are part of the simulation environment, were executed in a non-approximated state. The selected applications and their quality metrics are:

- **Typical applications:** Typically, related work on Approximate Computing use multimedia processing algorithms to demonstrate results, mostly due to their perceived resiliency to approximations [Mittal, 2016]. To represent these, we selected the JPEG compression algorithm from AxBench [Yazdanbakhsh et al., 2016a] and Fast Fourier Transform from MiBench [Guthaus et al., 2001]. JPEG quality was estimated by comparing accurate- and approximate-computed using the Structural Similarity Index [Wang et al., 2004; Avanaki, 2009]. For FFT, we accounted for the number of samples out of a tolerance margin of 10^{-9} after reconstruction.

- **CPU-Bound applications** Mandelbrot, N-Body e SpectralNorm were selected from [Gouy, 2004?]. These applications have in common their higher use of CPU and less access to memory, potentially demonstrating higher resiliency to memory approximations. The bitmaps generated by Mandelbrot were compared using the Structural Similarity Index [Wang et al., 2004; Avanaki, 2009], and we computed the Mean Relative Error of the N-Body and SpectralNorm numeric outputs.

- **Memory-Bound applications:** Since these present higher memory usage, the applications Dijkstra, QSort and bzip compression and decompression (bunzip), selected from MiBench [Guthaus et al., 2001] e cBench [Fursin, 2010?], are more susceptible to memory approximations. Quality for QSort was computed accounting for the number of correctly ordered elements. For Dijkstra, the output was modeled in the form of a routing table, in which each element in line i and column j is the next hop to destination j from i. Thus, quality is the fraction of correctly computed hops. The bzip algorithms were used to compress and decompress text files, and quality computed by the similarity of their contents – length of similar substrings.

4.2.3 Results

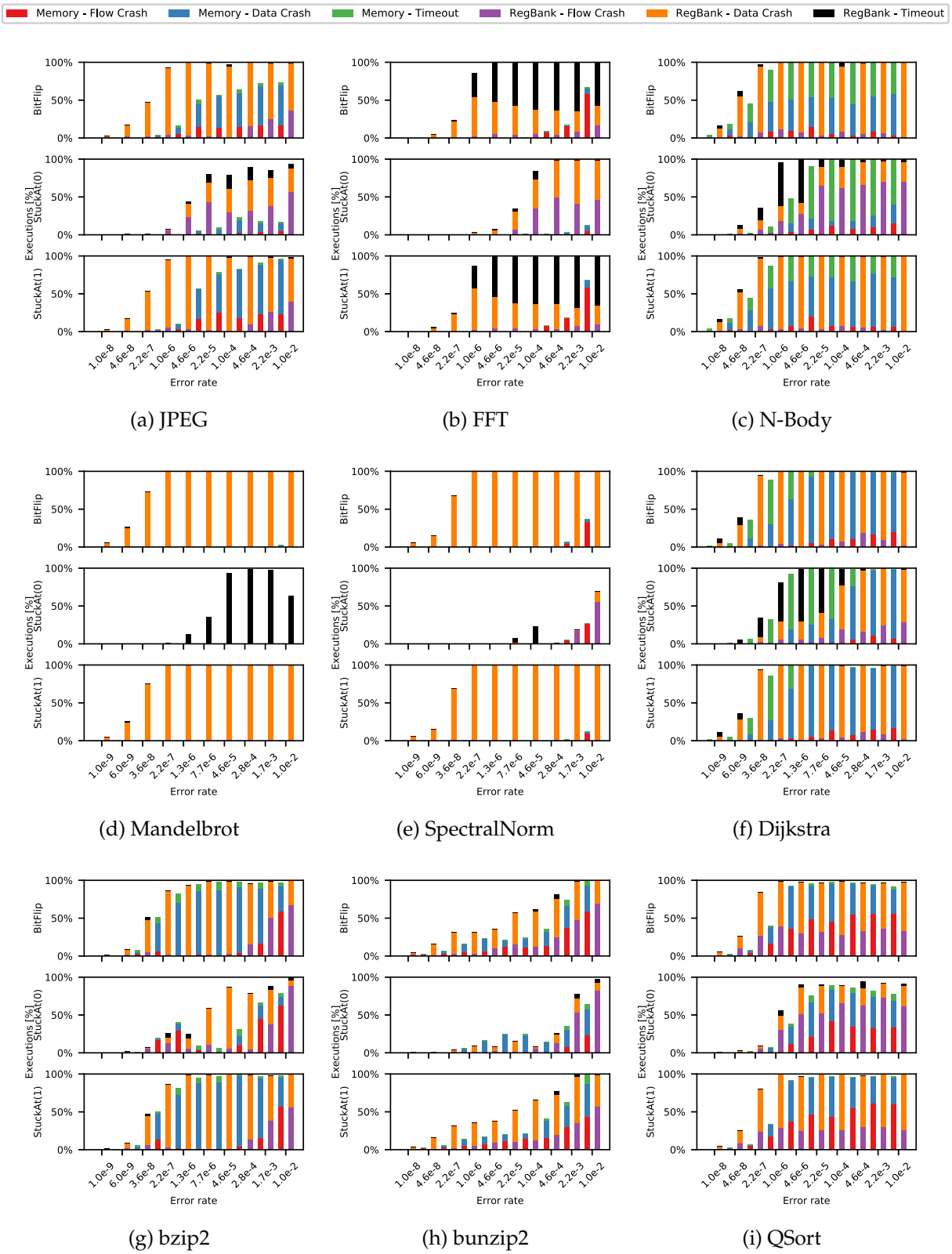
Each application was run 100 times at 10 different error rates. For each execution, we analyzed the application resilience – or the probability of an error to make the application crash – and the final quality of results. From resilience and quality, we estimated the likelihood of an instance to require a re-execution, and energy was accounted using the method in [Wang and Calhoun, 2011].

Resilience analysis

In the resilience analysis, the occurrence of a crash means that an execution was interrupted before the computation produced a valid result. The crashes were classified in three categories:

- Flow crash: occurs when a branch target is incorrectly read, causing the program to be deviated to an invalid address.

- Data crash: occurs when data is fetched from an invalid memory address (Segmentation Fault).



(a) JPEG (b) FFT (c) N-Body

(d) Mandelbrot (e) SpectralNorm (f) Dijkstra

(g) bzip2 (h) bunzip2 (i) QSort

Figure 4.12: Resilience analysis.

- Timeout: occurs when a valid result is not computed within the time limits. The maximum time an approximate execution can take was fixed in 5 times the accurate execution of the same instance.

Fig. 4.12 shows the resilience analysis. In general, errors in the register bank provoke more crashes in the control flow. The register bank stores, additionally to local values, memory addresses, loop control variables, and function return addresses. Also, the timeout crashes concentrate mainly in situations where errors are applied to the register bank, demonstrating the low resilience of control structures to approximations.

The stuck at zero errors are perceived easily masked by the applications in terms of execution crashes. This kind of error, when affecting memory addresses, tends to change the datum to an address that is part of the same program, possibly in the same memory page, mitigating data and flow crashes. On the other hand, this same behavior, when applied to branch target addresses or loop control variables, increases the application execution time, possibly creating endless loops, increasing timeout crashes, especially in convergence-based applications such as N-Body (Fig. 4.12c) and Dijkstra (Fig. 4.12f).

Application resilience is a limiting factor in the energy-quality trade-off, since an execution crash results in useless computation and, consequently, waste of energy budget. Additionally, even when a crash does not occur, control flow issues may lead to a higher execution time, which also negatively affects the energy cost.

Most crashes caused by the errors injected in memory are data crashes. This kind of crash may be masked by isolating critical data regions containing pointers and branch targets, such as the application stack. Thus, applications would potentially show better resilience, since the errors would affect only memory words representing general data. On the other hand, this critical memory region should be kept in error-free mode, which impacts the final energy cost of operation.

Quality of results

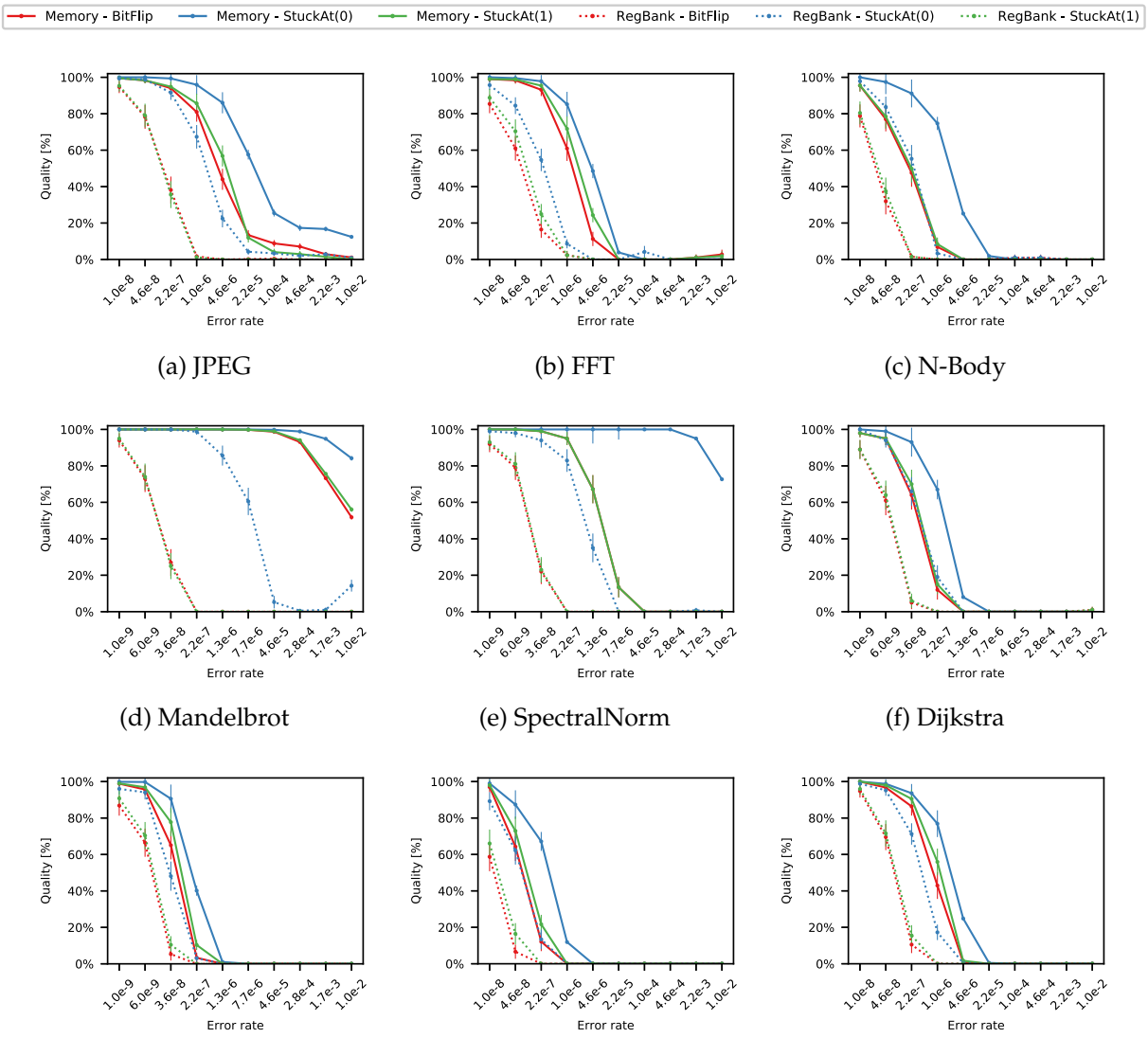
Fig. 4.13 shows the average quality of results for each application, in comparison with accurate executions. The averages were taken considering all 100 executions for each error rate with a confidence interval of 95%. Executions that resulted in an execution crash were considered as 0% quality, meaning no result was obtained.

Execution crashes are dominant factors in final quality. The isolation of control structures may avoid such crashes, directing the impact of memory errors to the final results. The analysis of the initial points in each plot of Fig. 4.13 shows that the loss in quality is smoother in executions successfully completed, what indicates that higher energy savings could be achieved.

Another effect of low resilience of applications is the higher impact on quality caused by approximations in registers. Yet, not considering crashes, they cause higher quality degradation. Since any application would operate much more times in registers than in memory, these end up being more affected by errors. This indicates a drawback on the approximation technique when applied to the register bank.

Energy

Although approximated memory structures offer power savings, the occurrence of execution crashes and the quality degradation may require the re-execution of certain



(a) JPEG (b) FFT (c) N-Body

(d) Mandelbrot (e) SpectralNorm (f) Dijkstra

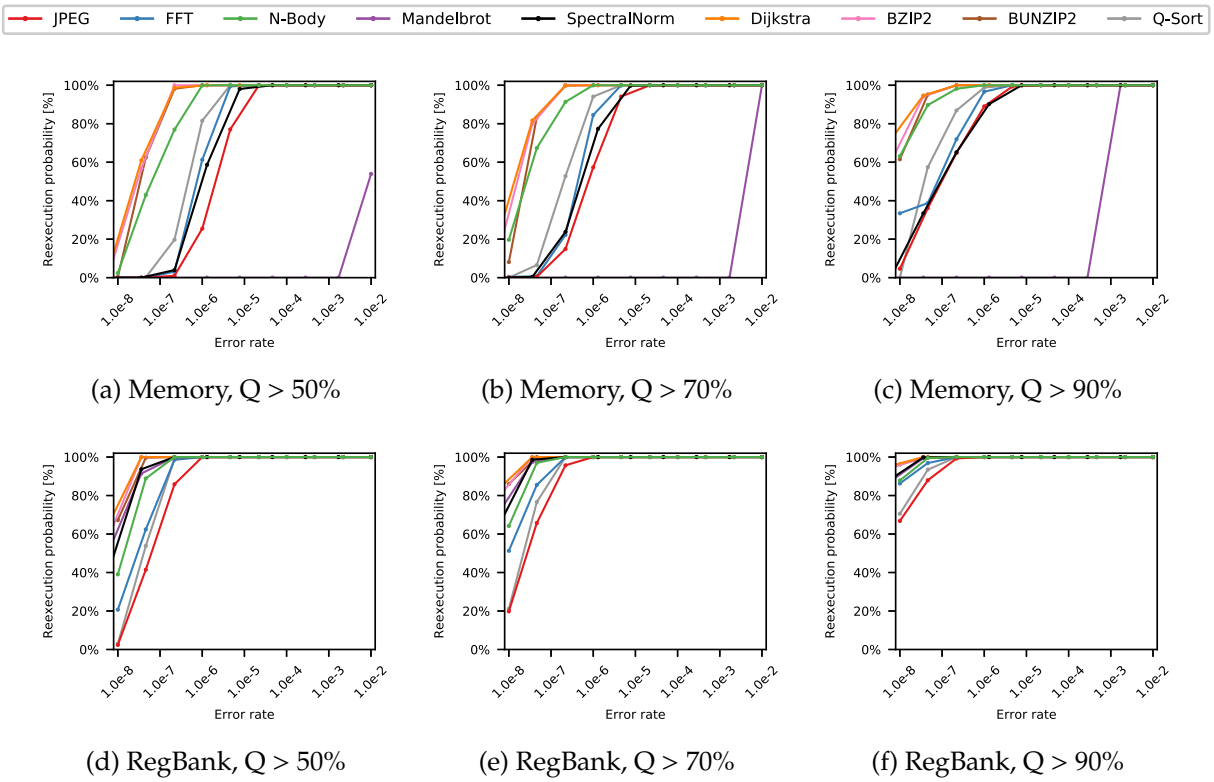
(g) bzip2 (h) bunzip2 (i) QSort

Figure 4.13: Quality of results.

instances. When a re-execution in accurate more is needed, it impacts the energy consumption. Based on the quality of results (Fig. 4.13), we estimate the probability of an execution to result in lower than acceptable quality, triggering a re-execution. Fig. 4.14 shows this probability for different quality thresholds, considering errors applied to memory and registers.

An accurate-mode re-execution results in a fixed energy penalty for each application. On the other hand, an approximate-mode execution energy depends on the elapsed time before obtaining a result or occurrence of a crash. This, the expected relative energy was computed according to the number of instructions executed in both accurate and approximate modes.

Fig. 4.15 relates the error rate to the energy cost for each application in a 95% confidence interval. Generally, all applications show energy savings by lowering the acceptable quality threshold, considering errors applied to memory. For the registers,



(a) Memory, Q > 50%

(b) Memory, Q > 70%

(c) Memory, Q > 90%

(d) RegBank, Q > 50%

(e) RegBank, Q > 70%

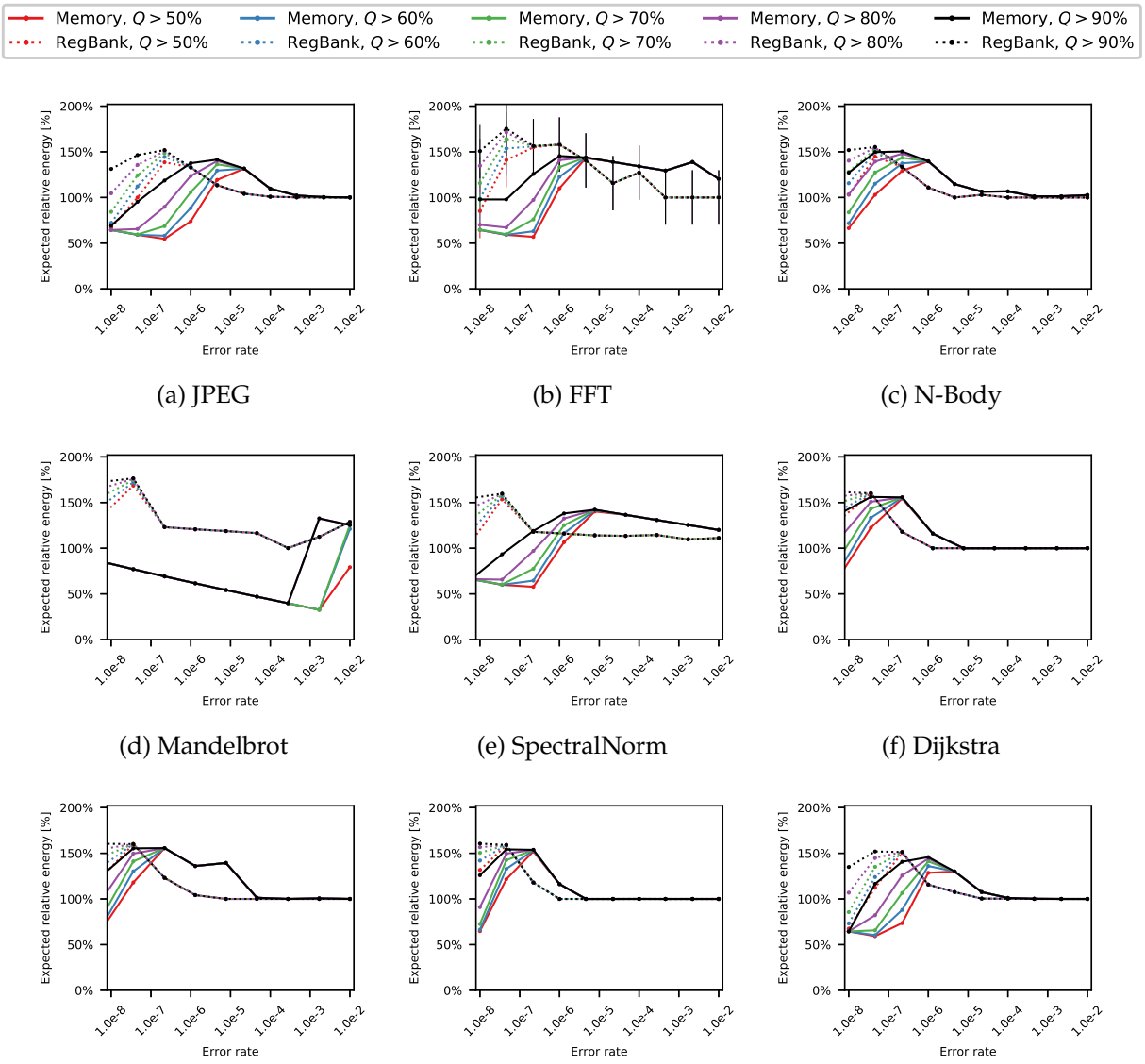
(f) RegBank, Q > 90%

Figure 4.14: Re-execution probability.

however, most results do not even demonstrate any savings, mostly due to the low resilience of applications.

Particularly, including errors in registers for the FFT application caused a great fluctuation on the number of executed instructions in approximate mode between executions. A similar behavior is observed analyzing the large number of timeout crashed for this applications (Fig. 4.12b). The nested loops in the FFT algorithm does not favor errors affecting control structures, such as registers.

The energy results show a behavior similar to the expected one (Fig. 4.10c), in which the energy cost is reduced up to a point where quality losses are dominant, requiring re-execution. This inflection point in energy consumption – point *B* in Fig. 4.10c – is visible for 5 out of the 9 applications in our test case. The other applications present a positive bias in the initial points, indicating that an equilibrium point exists, even if smoother, in a lower error rate. Lower rates, however, attenuate energy savings and the potential benefits of approximations.



(a) JPEG (b) FFT (c) N-Body

(d) Mandelbrot (e) SpectralNorm (f) Dijkstra

(g) bzip2 (h) bunzip2 (i) QSort

Figure 4.15: Energy trade-off.

Chapter 5

Conclusion

We presented *ADeLe*, a high-level descriptive language for hardware approximations. Our language translates user-defined models of hardware approximations into higher-level structures and injects them into CPU models for architectural simulations. The modeling abstraction offered by *ADeLe* can represent both data- and operation-based approximations, which include a large subset of approximation techniques available in the literature. Furthermore, it supports the representation of energy models and operating parameters, which allows for straight-forward energy consumption analysis.

ADeLe, as a verification tool, is complementary to design abstractions such as ABACUS [Nepal et al., 2014] and Axilog [Yazdanbakhsh et al., 2015]. These are focused on designing or representing approximations as self-contained features and leave validation to expensive simulation processes, while the *ADeLe* framework represents the approximations while in interaction with a real system. The generality of our language abstraction also makes it an extension to existing CPU and fault injection simulators. Thus, an approximation designer could take advantage of a high-level tool to explore possible hardware approximations, and then use *ADeLe* to validate its behavior over a range of target architectures and applications. Even when in comparison with existing approximation-aware frameworks such as VarEMU [Wanner et al., 2013] and React [Wyse et al., 2015], *ADeLe* proposes a higher-level comprehensive abstraction to represent the approximations and their interactions with the system, offering flexibility and full control to the designer on the verification process.

To demonstrate the applicability of our method on modeling approximations, we adapted an existing CPU simulator based on an *ADeLe* description, injecting approximations, and ran multiple test-cases, obtaining quality and energy metrics. Our results, in addition to demonstrating how each approximation trades quality for energy for selected applications, show that the simulations produced using our language framework can generate comprehensive results with reduced design effort. We further argue that other simulation configurations would be easily represented, thus extending our language capability.

The execution of this work made clear that the feasibility of Approximate Computing techniques depends on architecture-level integration. Although the literature presents work on developing approximate hardware modules and hardware-level techniques to generate approximations, their interaction with a complete system is still

uncertain. Particularly, a real-world "approximatable" processor would require a dedicated control mechanism to allow isolation of critical areas on target applications, while introducing low overhead to allow energy savings. While *ADeLe* represents a powerful framework to simulate such a system executing applications, we foresee, as future work, the proposition and evaluation of architecture-level alternatives that allow integration of Approximate Computing in a real system.

5.1 Production

This work originated the production of the following articles:

- *ADeLe: Rapid Architectural Simulation for Approximate Hardware*
  Isaías Felzmann, Matheus Susin, Liana Duenha, Rodolfo Azevedo and Lucas Wanner.
  Presented at the 30th International Symposium on Computer Architecture and High Performance Computing (SBAC-PAD 2018), Lyon, France, September 2018.
  Best Paper Award.

- *Impact of memory approximation on energy efficiency* (In Portuguese)
  Isaías Felzmann, João Fabrício Filho, Rodolfo Azevedo and Lucas Wanner.
  Presented at *XIX Simpósio em Sistemas Computacionais de Alto Desempenho* (WSCAD 2018), São Paulo, October 2018.
  Original title: *Impacto de memórias aproximadas na eficiência energética*.

- *ADeLe: A Description Language for Approximate Hardware*
  Isaías Felzmann, Matheus Susin, Liana Duenha, Rodolfo Azevedo and Lucas Wanner.
  Submitted to Future Generation Computer Systems – Best of SBAC-PAD 2018 (Special Issue).

Bibliography

SIPI Image Database, 1997. URL `http://sipi.usc.edu/database/database.php`.

Alireza Nasiri Avanaki. Exact histogram specification optimized for structural similarity. *CoRR*, 2009.

Mario Barbareschi, Antonino Mazzeo, Domenico Amelino, Alberto Bosio, and Antonio Tammaro. Implementing Approximate Computing Techniques by Automatic Code Mutation. In *3rd Workshop On Approximate Computing (WAPCO), 2017*, 2017.

Fabrice Bellard. QEMU, a Fast and Portable Dynamic Translator. In *Proceedings of the Annual Conference on USENIX Annual Technical Conference*, ATEC '05, page 41, Berkeley, CA, USA, 2005. USENIX Association.

Ramon Bertran, Yolanda Becerra, David Carrera, Vicenç Beltran, Marc Gonzàlez, Xavier Martorell, Nacho Navarro, Jordi Torres, and Eduard Ayguadé. Energy accounting for shared virtualized environments under DVFS using PMC-based power models. *Future Generation Computer Systems*, 28(2):457–468, 2012. ISSN 0167-739X. doi: 10.1016/j.future.2011.03.007.

Christian Bienia. *Benchmarking Modern Multiprocessors*. PhD thesis, Princeton University, 2011.

Nathan Binkert, Bradford Beckmann, Gabriel Black, Steven K Reinhardt, Ali Saidi, Arkaprava Basu, Joel Hestness, Derek R Hower, Tushar Krishna, Somayeh Sardashti, Rathijit Sen, Korey Sewell, Muhammad Shoaib, Nilay Vaish, Mark D Hill, and David A Wood. The Gem5 Simulator. *ACM SIGARCH Computer Architecture News*, 39(2):1–7, 2011. ISSN 0163-5964. doi: 10.1145/2024716.2024718.

Shekhar Borkar. Extreme Energy Efficiency by Near Threshold Voltage Operation. In Michael Hübner and Cristina Silvano, editors, *Near Threshold Computing: Technology, Methods and Applications*, pages 3–18. Springer International Publishing, Cham, 2016. ISBN 978-3-319-23389-5. doi: 10.1007/978-3-319-23389-5_1.

Shekhar Borkar and Andrew A Chien. The future of microprocessors. *Communications of the ACM*, 54(5):67, may 2011. ISSN 00010782. doi: 10.1145/1941487.1941507.

A Bosio and G D Natale. LIFTING: A Flexible Open-Source Fault Simulator. In *2008 17th Asian Test Symposium*, pages 35–40, nov 2008. doi: 10.1109/ATS.2008.17.

D Brooks, V Tiwari, and M Martonosi. Wattch: a framework for architectural-level power analysis and optimizations. In *Proceedings of 27th International Symposium on Computer Architecture - ISCA'00*, pages 83–94, 2000.

B H Calhoun and A Chandrakasan. Analyzing static noise margin for sub-threshold SRAM in 65nm CMOS. In *Proceedings of the 31st European Solid-State Circuits Conference, 2005. ESSCIRC 2005.*, pages 363–366, 2005. doi: 10.1109/ESSCIR.2005.1541635.

V Camus, J Schlachter, and C Enz. Energy-efficient inexact speculative adder with high performance and accuracy control. In *2015 IEEE International Symposium on Circuits and Systems (ISCAS)*, pages 45–48, 2015. doi: 10.1109/ISCAS.2015.7168566.

V Camus, J Schlachter, C Enz, M Gautschi, and F K Gurkaynak. Approximate 32-bit floating-point unit design with 53% power-area product reduction. In *ESSCIRC Conference 2016: 42nd European Solid-State Circuits Conference*, pages 465–468, 2016a. doi: 10.1109/ESSCIRC.2016.7598342.

Vincent Camus, Jeremy Schlachter, and Christian Enz. A Low-power Carry Cut-back Approximate Adder with Fixed-point Implementation and Floating-point Precision. In *Proceedings of the 53rd Annual Design Automation Conference*, DAC '16, pages 127:1—-127:6, New York, NY, USA, 2016b. ACM. ISBN 978-1-4503-4236-0. doi: 10.1145/2897937.2897964.

Michael Carbin, Sasa Misailovic, and Martin C Rinard. Verifying Quantitative Reliability for Programs That Execute on Unreliable Hardware. *SIGPLAN Not.*, 48(10):33–52, 2013. ISSN 0362-1340. doi: 10.1145/2544173.2509546.

E. Carlisle and A. D. George. Cache fault injection with DrSEUs. In *IEEE Aerospace Conference*, March 2018.

J Carreira, H Madeira, and J G Silva. Xception: a technique for the experimental evaluation of dependability in modern computers. *IEEE Transactions on Software Engineering*, 24(2):125–136, 1998. ISSN 0098-5589. doi: 10.1109/32.666826.

M C Chiang, T C Yeh, and G F Tseng. A QEMU and SystemC-Based Cycle-Accurate ISS for Performance Estimation on SoC Development. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 30(4):593–606, 2011. ISSN 0278-0070. doi: 10.1109/TCAD.2010.2095631.

V K Chippa, D Mohapatra, K Roy, S T Chakradhar, and A Raghunathan. Scalable Effort Hardware Design. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, 22 (9):2004–2016, 2014. ISSN 1063-8210. doi: 10.1109/TVLSI.2013.2276759.

R H Dennard, F H Gaensslen, V L Rideout, E Bassous, and A R LeBlanc. Design of ion-implanted MOSFET's with very small physical dimensions. *IEEE Journal of Solid-State Circuits*, 9(5):256–268, 1974. ISSN 0018-9200. doi: 10.1109/JSSC.1974.1050511.

Hadi Esmaeilzadeh, Emily Blem, Renee St. Amant, Karthikeyan Sankaralingam, and Doug Burger. Dark silicon and the end of multicore scaling. In *Proceedings of the 38th annual international symposium on Computer architecture - ISCA '11*, volume 39, page 365, New York, 2011. ACM Press. ISBN 9781450304726. doi: 10.1145/2000064.2000108.

I B Felzmann, J Fabrício Filho, R Azevedo, and L F Wanner. Impacto de memórias aproximadas na eficiência energética. In *Anais do Simpósio em Sistemas Computacionais de Alto Desempenho (WSCAD)*, pages 111–122, 2018a.

I B Felzmann, M M Susin, L Duenha, R Azevedo, and L F Wanner. ADeLe: Rapid Architectural Simulation for Approximate Hardware. In *2018 30th International Symposium on Computer Architecture and High Performance Computing (SBAC-PAD)*, pages 9–16, 2018b. doi: 10.1109/CAHPC.2018.8645875.

D Ferraretto and G Pravadelli. Efficient fault injection in QEMU. In *2015 16th Latin-American Test Symposium (LATS)*, pages 1–6, 2015. doi: 10.1109/LATW.2015.7102401.

Grigori Fursin. Collective Benchmark, 2010? URL `https://ctuning.org/cbench/`.

Shrikanth Ganapathy, Georgios Karakonstantis, Adam Teman, and Andreas Burg. Mitigating the Impact of Faults in Unreliable Memories for Error-resilient Applications. In *Proceedings of the 52Nd Annual Design Automation Conference*, DAC '15, pages 102:1—-102:6, New York, NY, USA, 2015. ACM. ISBN 978-1-4503-3520-1. doi: 10.1145/2744769.2744871.

M Gautschi, M Schaffner, F K Gürkaynak, and L Benini. 4.6 A 65nm CMOS 6.4-to-29.2pJ/FLOP@0.8V shared logarithmic floating point unit for acceleration of nonlinear function kernels in a tightly coupled processor cluster. In *2016 IEEE International Solid-State Circuits Conference (ISSCC)*, pages 82–83, jan 2016. doi: 10.1109/ISSCC.2016.7417917.

F A Geissler, F L Kastensmidt, and J E P Souza. Soft error injection methodology based on QEMU software platform. In *2014 15th Latin American Test Workshop - LATW*, pages 1–5, 2014. doi: 10.1109/LATW.2014.6841910.

J George, B Marr, B E S Akgul, and K V Palem. Probabilistic Arithmetic and Energy Efficient Embedded Signal Processing. In *Proceedings of the 2006 International Conference on Compilers, Architecture and Synthesis for Embedded Systems*, CASES '06, pages 158–168, New York, NY, USA, 2006. ACM. ISBN 1-59593-543-6. doi: 10.1145/1176760.1176781.

Isaac Gouy. The Computer Language Benchmarks Game, 2004? URL `https://benchmarksgame-team.pages.debian.net/benchmarksgame/`.

Marcelo Guedes, Rafael Auler, Liana Duenha, Edson Borin, and Rodolfo Azevedo. An automatic energy consumption characterization of processors using ArchC. *Journal of Systems Architecture*, 59(8):603–614, 2013. ISSN 1383-7621. doi: 10.1016/j.sysarc.2013.05.025.

T Gupta, C Bertolini, O Heron, N Ventroux, T Zimmer, and F Marc. High Level Power and Energy Exploration Using ArchC. In *2010 22nd International Symposium on Computer Architecture and High Performance Computing*, pages 25–32, 2010. doi: 10.1109/SBAC-PAD.2010.13.

M. R. Guthaus, J. S. Ringenberg, D. Ernst, T. M. Austin, T. Mudge, and R. B. Brown. Mibench: A free, commercially representative embedded benchmark suite. In *IEEE WWC*, 2001.

Seungjae Han, K G Shin, and H A Rosenberg. DOCTOR: an integrated software fault injection environment for distributed real-time systems. In *Proceedings of 1995 IEEE International Computer Performance and Dependability Symposium*, pages 204–213, 1995. doi: 10.1109/IPDS.1995.395831.

N M Ho, E Manogaran, W F Wong, and A Anoosheh. Efficient floating point precision tuning for approximate computing. In *2017 22nd Asia and South Pacific Design Automation Conference (ASP-DAC)*, pages 63–68, jan 2017. doi: 10.1109/ASPDAC.2017.7858297.

A Höller, A Krieg, T Rauter, J Iber, and C Kreiner. QEMU-Based Fault Injection for a System-Level Analysis of Software Countermeasures Against Fault Attacks. In *2015 Euromicro Conference on Digital System Design*, pages 530–533, 2015. doi: 10.1109/DSD.2015.79.

Mei-Chen Hsueh, Timothy K Tsai, and Ravishankar K Iyer. Fault injection techniques and tools. *Computer*, 30(4):75–82, 1997.

IEEE. IEEE Standard for Floating-Point Arithmetic. *IEEE Std 754-2008*, pages 1–70, 2008. doi: 10.1109/IEEESTD.2008.4610935.

C Isci and M Martonosi. Runtime power monitoring in high-end processors: methodology and empirical data. In *Proceedings. 36th Annual IEEE/ACM International Symposium on Microarchitecture, 2003. MICRO-36.*, pages 93–104, 2003. doi: 10.1109/MICRO.2003.1253186.

Abhishek Jaiantilal, Yifei Jiang, and Shivakant Mishra. Modeling CPU Energy Consumption for Energy Efficient Scheduling. In *Proceedings of the 1st Workshop on Green Computing*, GCM '10, pages 10–15, New York, NY, USA, 2010. ACM. ISBN 978-1-4503-0450-4. doi: 10.1145/1925013.1925015.

Andrew B Kahng and Seokhyeong Kang. Accuracy-configurable Adder for Approximate Arithmetic Designs. In *Proceedings of the 49th Annual Design Automation Conference - DAC'12*, pages 820–825, New York, 2012. ACM. ISBN 978-1-4503-1199-1. doi: 10.1145/2228360.2228509.

G A Kanawati, N A Kanawati, and J A Abraham. FERRARI: a tool for the validation of system dependability properties. In *Digest of Papers. FTCS-22: The Twenty-Second International Symposium on Fault-Tolerant Computing*, pages 336–344, 1992. doi: 10.1109/FTCS.1992.243567.

N Kapadia and S Pasricha. A Runtime Framework for Robust Application Scheduling With Adaptive Parallelism in the Dark-Silicon Era. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, 25(2):534–546, 2017. ISSN 1063-8210. doi: 10.1109/TVLSI.2016.2594238.

D. S. Khudia, B. Zamirai, M. Samadi, and S. Mahlke. Rumba: An online quality management system for approximate computing. In *ISCA*, 2015.

N S Kim, T Austin, D Baauw, T Mudge, K Flautner, J S Hu, M J Irwin, M Kandemir, and V Narayanan. Leakage current: Moore's law meets static power. *Computer*, 36(12):68–75, 2003. ISSN 0018-9162. doi: 10.1109/MC.2003.1250885.

Bruno Kleinert, Simone Weiß, Franziska Schäfer, Jupiter Bakakeu, and Dietmar Fey. Adaptive Synchronization Interface for Hardware-Software Co-Simulation Based on SystemC and QEMU. In *Proceedings of the 9th EAI International Conference on Simulation Tools and Techniques*, SIMUTOOLS'16, pages 28–36, Brussels, Belgium, 2016. ICST. ISBN 978-1-63190-120-1.

M Kooli and G Di Natale. A survey on simulation-based fault injection tools for complex systems. In *2014 9th IEEE International Conference on Design Technology of Integrated Systems in Nanoscale Era (DTIS)*, pages 1–6, 2014. doi: 10.1109/DTIS.2014.6850649.

M Kooli, A Bosio, P Benoit, and L Torres. Software testing and software fault injection. In *2015 10th International Conference on Design Technology of Integrated Systems in Nanoscale Era (DTIS)*, pages 1–6, 2015. doi: 10.1109/DTIS.2015.7127370.

Logan Kugler. Is "Good Enough" Computing Good Enough? *Communications of the ACM ACM*, 58(5):12–14, 2015. ISSN 0001-0782. doi: 10.1145/2742482.

Parag Kulkarni, Puneet Gupta, and Miloš D Ercegovac. Trading accuracy for power in a multiplier architecture. *Journal of Low Power Electronics*, 7(4):490–501, 2011.

Mark S K Lau, Keck-Voon Ling, and Yun-Chung Chu. Energy-aware Probabilistic Multiplier: Design and Analysis. In *Proceedings of the 2009 International Conference on Compilers, Architecture, and Synthesis for Embedded Systems*, CASES '09, pages 281–290, New York, NY, USA, 2009. ACM. ISBN 978-1-60558-626-7. doi: 10.1145/1629395.1629434.

Yongsoon Lee, Younhee Choi, Seok Bum Ko, and Moon Ho Lee. Performance analysis of bit-width reduced floating-point arithmetic units in FPGAs: A case study of neural network-based face detector. *Eurasip Journal on Embedded Systems*, 2009(1):258921, jul 2009. ISSN 16873955. doi: 10.1155/2009/258921.

Chi-Keung Luk, Robert Cohn, Robert Muth, Harish Patil, Artur Klauser, Geoff Lowney, Steven Wallace, Vijay Janapa Reddi, and Kim Hazelwood. Pin: Building Customized Program Analysis Tools with Dynamic Instrumentation. *ACM SIGPLAN Notices*, 40(6):190–200, 2005. ISSN 0362-1340. doi: 10.1145/1064978.1065034.

J S Miguel, M Badr, and N E Jerger. Load Value Approximation. In *Proceedings of the 47th Annual IEEE/ACM International Symposium on Microarchitecture - MICRO-47*, pages 127–139, 2014. doi: 10.1109/MICRO.2014.22.

A Mineo, M Palesi, G Ascia, P P Pande, and V Catania. On-Chip Communication Energy Reduction Through Reliability Aware Adaptive Voltage Swing Scaling. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 35(11):1769–1782, nov 2016. ISSN 0278-0070. doi: 10.1109/TCAD.2016.2524556.

Sparsh Mittal. A Survey of Techniques for Approximate Computing. *ACM Computing Surveys - CSUR*, 48(4):62:1—-62:33, 2016. ISSN 0360-0300. doi: 10.1145/2893356.

M Monton, A Portero, M Moreno, B Martinez, and J Carrabina. Mixed SW/SystemC SoC Emulation Framework. In *2007 IEEE International Symposium on Industrial Electronics*, pages 2338–2341, 2007. doi: 10.1109/ISIE.2007.4374971.

G E Moore. Cramming More Components Onto Integrated Circuits. *Proceedings of the IEEE*, 86(1):82–85, jan 1998. ISSN 0018-9219. doi: 10.1109/JPROC.1998.658762.

V Mrazek, R Hrbacek, Z Vasicek, and L Sekanina. EvoApprox8b: Library of approximate adders and multipliers for circuit design and benchmarking of approximation methods. In *Design, Automation Test in Europe Conference Exhibition (DATE), 2017*, pages 258–261, 2017. doi: 10.23919/DATE.2017.7926993.

K Nepal, Y Li, R I Bahar, and S Reda. ABACUS: A technique for automated behavioral synthesis of approximate computing circuits. In *2014 Design, Automation Test in Europe Conference Exhibition (DATE)*, pages 1–6, 2014. doi: 10.7873/DATE.2014.374.

Avadh Patel, Furat Afram, Shunfei Chen, and Kanad Ghose. MARSS: A Full System Simulator for Multicore x86 CPUs. *Proceedings of the 48th Annual Design Automation Conference - DAC'11*, pages 1050–1055, 2011. ISSN 0738100x. doi: 10.1145/2024724.2024954.

S Potyra, V Sieh, and M Dal Cin. Evaluating Fault-tolerant System Designs Using FAUmachine. In *Proceedings of the 2007 Workshop on Engineering Fault Tolerant Systems*, EFTS '07, New York, NY, USA, 2007. ACM. ISBN 978-1-59593-725-4. doi: 10.1145/1316550.1316559.

Jan M Rabaey, Anantha P Chandrakasan, and Borivoje Nikolic. *Digital integrated circuits*, volume 2. Prentice hall Englewood Cliffs, 2002.

A Rahimi, A Marongiu, R K Gupta, and L Benini. A variability-aware OpenMP environment for efficient execution of accuracy-configurable computation on shared-FPU processor clusters. In *2013 International Conference on Hardware/Software Codesign and System Synthesis (CODES+ISSS)*, pages 1–10, 2013. doi: 10.1109/CODES-ISSS.2013.6659022.

Abbas Rahimi, Amirali Ghofrani, Kwang-Ting Cheng, Luca Benini, and Rajesh K Gupta. Approximate Associative Memristive Memory for Energy-efficient GPUs. In *Proceedings of the 2015 Design, Automation & Test in Europe Conference & Exhibition - DATE '15*, pages 1497–1502, San Jose, CA, USA, 2015. EDA Consortium. ISBN 978-3-9815370-4-8.

S Rigo, G Araujo, M Bartholomeu, and R Azevedo. ArchC: a systemC-based architecture description language. In *Proceedings of the 16th Symposium on Computer Architecture and High Performance Computing - SBAC-PAD'04*, pages 66–73, 2004. doi: 10.1109/SBAC-PAD.2004.8.

Adrian Sampson, Werner Dietl, Emily Fortuna, Danushen Gnanapragasam, Luis Ceze, and Dan Grossman. EnerJ: Approximate Data Types for Safe and General Low-power Computation. *SIGPLAN Not.*, 46(6):164–174, 2011. ISSN 0362-1340. doi: 10.1145/1993316.1993518.

Adrian Sampson, André Baixo, Benjamin Ransford, Thierry Moreau, Joshua Yip, Luis Ceze, and Mark Oskin. ACCEPT: A Programmer-Guided Compiler Framework for Practical Approximate Computing. 2015.

Muhammad Shafique, Siddharth Garg, Jörg Henkel, and Diana Marculescu. The EDA Challenges in the Dark Silicon Era: Temperature, Reliability, and Variability Perspectives. In *Proceedings of the 51st Annual Design Automation Conference*, DAC '14, pages 185:1—-185:6, New York, 2014a. ACM. ISBN 978-1-4503-2730-5. doi: 10.1145/2593069.2593229.

Muhammad Shafique, Siddharth Garg, Tulika Mitra, Sri Parameswaran, and Jörg Henkel. Dark Silicon As a Challenge for Hardware/Software Co-design: Invited Special Session Paper. In *Proceedings of the 2014 International Conference on Hardware/Software Codesign and System Synthesis*, CODES '14, pages 13:1–13:10, New York, 2014b. ACM. ISBN 978-1-4503-3051-0. doi: 10.1145/2656075.2661645.

C Slayman. Soft error trends and mitigation techniques in memory devices. In *2011 Proceedings - Annual Reliability and Maintainability Symposium*, pages 1–5, jan 2011. doi: 10.1109/RAMS.2011.5754515.

G Tagliavini, D Rossi, A Marongiu, and L Benini. Synergistic HW/SW Approximation Techniques for Ultra-Low-Power Parallel Computing. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, page 1, 2017. ISSN 0278-0070. doi: 10.1109/TCAD.2016.2633474.

V Tiwari, S Malik, A Wolfe, and M T C Lee. Instruction level power analysis and optimization of software. In *Proceedings of 9th International Conference on VLSI Design*, pages 326–328, jan 1996. doi: 10.1109/ICVD.1996.489624.

J Tong, D Nagle, and R Rutenbar. Reducing power by optimizing the necessary precision/range of floating-point arithmetic. *IEEE Transactions on VLSI Systems*, 8, 2000. doi: 10.1109/92.845894.

T K Tsai, R K Iyer, and D Jewitt. An approach towards benchmarking of fault-tolerant commercial systems. In *Proceedings of Annual Symposium on Fault Tolerant Computing*, pages 314–323, jun 1996. doi: 10.1109/FTCS.1996.534616.

J Wang and B H Calhoun. Minimum Supply Voltage and Yield Estimation for Large SRAMs Under Parametric Variations. *TVLSI*, 2011.

Zhou Wang, A. C. Bovik, H. R. Sheikh, and E. P. Simoncelli. Image quality assessment: from error visibility to structural similarity. *IEEE TIP*, 2004.

L Wanner, S Elmalaki, L Lai, P Gupta, and M Srivastava. VarEMU: An Emulation Testbed for Variability-aware Software. In *Proceedings of the Ninth IEEE/ACM/IFIP International Conference on Hardware/Software Codesign and System Synthesis (CODES+ISSS)*. IEEE Press, 2013. ISBN 978-1-4799-1417-3. doi: 10.1109/CODES-ISSS.2013.6659014.

Mark Wyse, Andre Baixo, Thierry Moreau, Bill Zorn, James Bornholt, Adrian Sampson, Luis Ceze, and Mark Oskin. React: A framework for rapid exploration of approximate computing techniques. In *Workshop on Approximate Computing Across the Stack - WAX'2015*, 2015.

Q Xu, T Mytkowicz, and N S Kim. Approximate Computing: A Survey. *IEEE Design Test*, 33(1):8–22, 2016. ISSN 2168-2356. doi: 10.1109/MDAT.2015.2505723.

Lei Yang, Weichen Liu, Weiwen Jiang, Chao Chen, Mengquan Li, Peng Chen, and Edwin H M Sha. Hardware-software collaboration for dark silicon heterogeneous manycore systems. *Future Generation Computer Systems*, 68:234–247, 2017. ISSN 0167-739X. doi: http://dx.doi.org/10.1016/j.future.2016.09.012.

A Yazdanbakhsh, D Mahajan, B Thwaites, J Park, A Nagendrakumar, S Sethuraman, K Ramkrishnan, N Ravindran, R Jariwala, A Rahimi, H Esmaeilzadeh, and K Bazargan. Axilog: Language support for approximate hardware design. In *2015 Design, Automation Test in Europe Conference Exhibition (DATE)*, pages 812–817, 2015. doi: 10.7873/DATE.2015.0513.

A Yazdanbakhsh, D Mahajan, P Lotfi-Kamran, and H Esmaeilzadeh. AXBENCH: A Multi-Platform Benchmark Suite for Approximate Computing. *IEEE Design and Test, special issue on Computing in the Dark Silicon Era*, pages 1–7, 2016a.

Amir Yazdanbakhsh, Gennady Pekhimenko, Bradley Thwaites, Hadi Esmaeilzadeh, Onur Mutlu, and Todd C Mowry. RFVP: Rollback-Free Value Prediction with Safe-to-Approximate Loads. *ACM Transactions on Architecture and Code Optimization - TACO*, 12(4):62:1—-62:26, 2016b. ISSN 1544-3566. doi: 10.1145/2836168.

T C Yeh and M C Chiang. On the interface between QEMU and SystemC for hardware modeling. In *2010 International Symposium on Next Generation Electronics*, pages 73–76, nov 2010. doi: 10.1109/ISNE.2010.5669197.

P Yin, C Wang, W Liu, and F Lombardi. Design and Performance Evaluation of Approximate Floating-Point Multipliers. In *2016 IEEE Computer Society Annual Symposium on VLSI (ISVLSI)*, pages 296–301, 2016. doi: 10.1109/ISVLSI.2016.15.