UNIVERSIDADE FEDERAL DO RIO GRANDE DO SUL
INSTITUTO DE INFORMÁTICA
PROGRAMA DE PÓS-GRADUAÇÃO EM COMPUTAÇÃO

MARCELO BRANDALERO

MuTARe: A Multi-Target, Adaptive Reconfigurable Architecture

Thesis presented in partial fulfillment
of the requirements for the degree of
Doctor of Computer Science

Advisor: Prof. Dr. Antonio Carlos Schneider Beck
Coadvisor: Prof. Dr. Luigi Carro

Porto Alegre
March 2019

CIP — CATALOGING-IN-PUBLICATION

Brandalero, Marcelo

MuTARe: A Multi-Target, Adaptive Reconfigurable Architecture / Marcelo Brandalero. – Porto Alegre: PPGC da UFRGS, 2019.

142 f.: il.

Thesis (Ph.D.) – Universidade Federal do Rio Grande do Sul. Programa de Pós-Graduação em Computação, Porto Alegre, BR–Brazil, 2019. Advisor: Antonio Carlos Schneider Beck; Coadvisor: Luigi Carro.

1. Computer architecture. 2. Reconfigurable architecture. 3. Adaptable architecture. 4. Approximate computing. 5. Near-threshold voltage computing. I. Beck, Antonio Carlos Schneider. II. Carro, Luigi. III. Título.

“A reliable way to make people believe in falsehoods is frequent repetition, because familiarity is not easily distinguished from truth. [...] Our comforting conviction that the world makes sense rests on a secure foundation: our almost unlimited ability to ignore our ignorance.”

— Daniel Kahneman

AGRADECIMENTOS

Realizar um doutorado não é uma tarefa fácil. Requer tempo, esforço, e, acima de tudo, apoio. Muito apoio, pois é um caminho longo, com muitas quedas e obstáculos. Nunca vou esquecer do comentário que ouvi de dois professores sobre o processo, antes de iniciá-lo. O primeiro me disse que, durante um doutorado, eu iria solucionar um problema, mas que apenas uma solução não seria o suficiente. Eu teria que ir adiante, investigar alternativas, acertar e errar muitas vezes antes de estar satisfeito com uma das soluções atingidas, sendo necessária muita persistência durante o processo. O segundo me disse que, durante o doutorado, eu iria ver meus colegas que se formaram comigo prosperarem financeiramente com seus trabalhos na indústria, e eu teria que achar alguma forma de conforto com os salários inferiores pagos pela academia. Hoje, passado o processo todo, posso dizer que ambas as afirmações estavam corretas.

Gostaria de agradecer em primeiro lugar aos meus pais por todo o apoio fornecido nesta trajetória, tanto emocional como financeiro. Eles sempre me inspiraram a acreditar em um futuro melhor e seguir adiante, não importando o quão difícil o caminho à frente pudesse parecer, ao mesmo tempo que me ofereciam a mão sempre que eu precisasse.

Em segundo lugar, gostaria de agradecer ao meu orientador, Prof. Antonio Carlos Schneider Beck, por toda a paciência que teve em me orientar durante quatro anos como doutorando e seis anos como aluno. Sei que não foram poucos os meus tropeços durante o caminho, e ainda assim ele acreditou em mim e me apoiou. Posso dizer que levo muitos aprendizados destas experiências, não apenas técnicos, mas também pessoais. Também merecem um agradecimento especial o Prof. Luigi Carro e o Prof. Muhammad Shafique pelas contribuições técnicas ao desenvolvimento desta tese.

Em terceiro lugar, é muito difícil falar de apoio e não lembrar dos amigos. Nunca fui uma pessoa de muitas amizades, mas sempre valorizei muito cada uma delas. Não citarei nomes, pois corro o risco de omitir pessoas que foram muito importantes para mim nesta trajetória, mas creio que cada uma delas saiba o quão importante sua participação na minha vida foi até aqui. Um agradecimento especial também deve ser feito a todos os colegas de laboratório, companheiros de pesquisa e que dividiram comigo muitos dos desafios encontrados nos últimos anos.

Finalmente, um agradecimento especial à minha namorada Roberta por todo o apoio e carinho neste difícil último ano, e por ter mostrado a mim uma nova forma de viver a vida.

ABSTRACT

Power consumption, earlier a design constraint only in embedded systems, has become the major driver for architectural optimizations in all domains, from the cloud to the edge. Application-specific accelerators provide a low-power processing solution by efficiently matching the hardware to the application; however, since in many domains the hardware must execute efficiently a broad range of fast-evolving applications, unpredictable at design time and each with distinct resource requirements, alternatives approaches are required. Besides that, the same hardware must also adapt the computational power at run time to the system status and workload sizes. To address these issues, this thesis presents a general-purpose *reconfigurable accelerator* that can be coupled to a *heterogeneous set of cores* and supports *Dynamic Voltage and Frequency Scaling (DVFS)*, synergistically combining the techniques for a better match between different applications and hardware when compared to current designs. The resulting architecture, *MuTARe*, provides a coarse-grained regular and reconfigurable structure which is suitable for automatic acceleration of deployed code through dynamic binary translation. In extension to that, the structure of *MuTARe* is further leveraged to apply two emerging computing paradigms that can boost the power-efficiency: *Near-Threshold Voltage (NTV) computing* (while still supporting transparent acceleration) and *Approximate Computing (AxC)*. Compared to a traditional heterogeneous system with DVFS support, the base *MuTARe* architecture can automatically improve the execution time by up to $1.3\times$, or adapt to the same task deadline with $1.6\times$ smaller energy consumption, or adapt to the same low energy budget with $2.3\times$ better performance. In NTV mode, *MuTARe* can transparently save further 30% energy in memory-intensive workloads by operating the combinatorial datapath at half the memory frequency. In AxC mode, *MuTARe* can further improve power savings by up to 50% by leveraging *approximate functional units* for arithmetic computations.

Keywords: Computer architecture. reconfigurable architecture. adaptable architecture. approximate computing. near-threshold voltage computing.

MuTARe: Uma Arquitetura Multi-Proposito Adaptativa e Reconfigurável

RESUMO

Consumo de potência, antigamente um limitante no projeto apenas de sistemas embarcados, hoje é um dos principais objetivos de otimização em todos os domínios de dispositivos, desde a computação na núvem até a computação na borda. Aceleradores de propósito específico são capazes de fornecer uma solução para o processamento de baixa potência ao adequar o hardware à aplicação; porém, visto que, em diversos domínios, o hardware necessita executar uma ampla gama de aplicações, cada uma com diferentes requisitos computacionais, abordagens alternativas se fazem necessárias. Além disso, o mesmo hardware precisa se adequar, em tempo de execução, ao estado do sistema e tamanho da carga de trabalho, aumentando o poder computacional ao executar uma tarefa exigente e reduzindo-o quando inativo. De forma a resolver estes problemas, esta tese apresenta um acelerador de proposito geral que pode ser acoplado a um conjunto heterogeneo de cores e suporta DVFS, sinergisticamente combinando técnicas para uma melhor combinação entre diferentes aplicações e hardware quando comparado aos designs existentes hoje. A arquitetura resultante, *MuTARe*, provê uma estrutura regular e reconfigurável que é adequada para aceleração automática de código já exisente através de tradução binária. Além disso, *MuTARe* também provê uma estrutura adequada para aplicar dois emergentes paradigmas de computação que podem aumentar a eficiencia de potência: *computação no nível da tensão de threshold* (mantendo a capacidade de aceleração transparente) e *computação aproximativa*. Comparado a um sistema heterogeneo tradicional com suporte a DVFS, a arquitetura *MuTARe* base pode automaticamente melhorar o tempo de execução em $1.3\times$, ou adaptar-se para o mesmo baixo tempo de execução com uma redução de $1.6\times$ no consumo energético, ou adaptar-se para o mesmo baixo nível de energia com $2.3\times$ melhor performance. No modo *near-threshold*, *MuTARe* pode melhorar o consumo de potência de forma transparente em mais 30% em tarefas que exigem bastante memória operando o circuito combinacional à metade da frequencia da memória. No modo *computação aproximativa*, *MuTARe* consegue melhorar o consumo de potência em até mais 50% usando unidades funcionais aproximativas para as computações.

Palavras-chave: arquiteturas de computadores, arquiteturas reconfiguráveis, arquiteturas adaptativas, computação aproximativa, computação no nível da tensão de threshold.

LIST OF ABBREVIATIONS AND ACRONYMS

V_{dd} Operating Voltage.

V_{th} Threshold Voltage.

f Operating Frequency.

ALU Arithmetic-Logic Unit.

ANN Artificial Neural Network.

ASIC Application-Specific Integrated Circuit.

AxC Approximate Computing.

BB Basic Block.

BT Binary Translation.

CAD Computer-Aided Design.

CCA Configurable Compute Accelerator.

CGRA Coarse-Grained Reconfigurable Array.

CLB Configurable Logic Block.

CMP Chip Multi-Processor.

CPU Central Processing-Unit.

CReAMS Custom Reconfigurable Arrays for Multiprocessor Systems.

DCT Discrete Cosine Transform.

DFG Data-Flow Graph.

DIM Dynamic Instruction Merging.

DSE Design-Space Exploration.

DVFS Dynamic Voltage and Frequency Scaling.

DySER Dynamically Specializing Execution Resources.

EDP Energy-Delay Product.

FA Full Adder.

FGRA Fine-Grained Reconfigurable Array.

FP Floating-Point.

FPGA Field-Programmable Gate Array.

FU Functional Unit.

GPP General-Purpose Processor.

GPU Graphics Processing Unit.

HARTMP Heterogeneous Arrays for Reconfigurable and Transparent Multicore Processing.

HPC High-Performance Computing.

iACT Intel Approximate Computing Toolkit.

IDCT Inverse Discrete Cosine Transform.

ILP Instruction-Level Parallelism.

IMPACT IMPrecise adders for low-power Approximate CompuTing.

IoT Internet-of-Things.

ISA Instruction-Set Architecture.

ISE Instruction-Set Extension.

LSB Least-Significant Bit.

LUT Lookup Table.

ME Motion Estimation.

MuTARe Multi-Target Adaptive Reconfigurable Architecture.

NPU Neural Processing Unit.

NRE Non-Recurrent Engineering.

NTV Near-Threshold Voltage.

OoO Out-Of-Order.

OP Operating Point.

PC Program Counter.

PE Processing Element.

PSNR Peak Signal-to-Noise Ratio.

PSoC Programmable System-On-Chip.

PVT Process-Voltage-Temperature.

RISP Reconfigurable-Instruction-Set Processor.

RMSE Root-Mean-Squared Error.

ROB Re-Order Buffer.

RTL Register-Transfer Level.

RU Reconfigurable Unit.

SAD Sum of Absolute Differences.

SIMD Single Instruction Multiple Data.

SNNAP Systolic Neural Network Accelerator in Programmable Logic.

SoC System-on-Chip.

SRAM Static Random-Access Memory.

SSIM Structural Similarity Index.

STV Super-Threshold Voltage.

SVM Support Vector Machine.

TLP Thread-Level Parallelism.

VOS Voltage Over-scaling.

LIST OF FIGURES

LIST OF TABLES

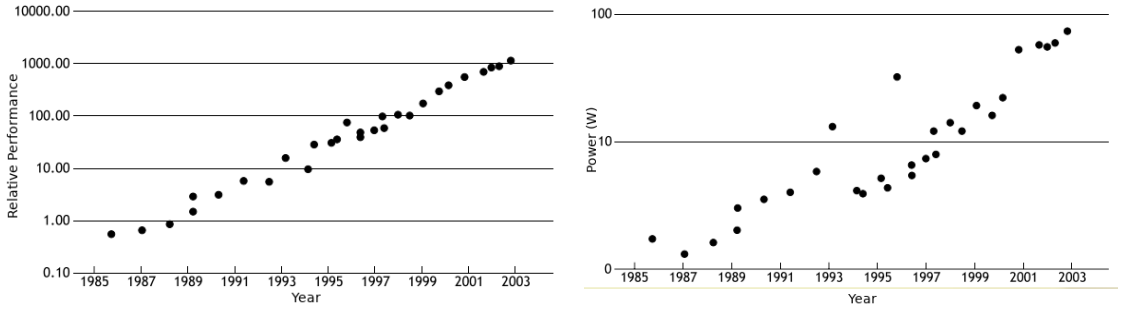
CONTENTS

1 INTRODUCTION

The microprocessor design landscape has changed dramatically over the last 40 years. For long after the first microprocessors were invented, improvements in CMOS transistor manufacturing followed Moore's and Dennard's scaling rules, allowing for smaller, more efficient and cheaper devices (MOORE, 1965; DENNARD et al., 1974). By that time, performance could be improved in a way that was entirely transparent to the programmers mainly for two reasons. First, smaller transistors presented less capacitance and could be charged faster, allowing circuits to operate at higher frequencies. Second, hardware designers were able to leverage the additional transistors to implement micro-architectural improvements that allowed multiple instructions to execute concurrently, exploiting the Instruction-Level Parallelism (ILP) that the applications presented (OLUKOTUN; HAMMOND, 2005). Given this transparency, programmers and end-users requiring more performance needed only wait for the next processor generation, as the performance was increasing exponentially (Fig. 1.1a). However, while the scaling rules promised constant power density as technology evolved, in practice device power has increased over the years (Fig. 1.1b), mostly because the operating voltage scaled slower than Dennard's predictions. For this reason, the contribution of static power (ignored in Dennard's original model) to the total power has significantly increased over the years (BOHR, 2007). However, since cooling technologies did not evolve proportionally, hardware designers reached a *power wall* - a limitation in the maximum power that any chip could sustain. Therefore, as transistors entered the nano-era, the strategy to keep increasing operation frequencies and providing transparent performance improvements proved to be unsustainable. Table 1.1 illustrates the power impacts of scaling, for a factor S, before and after Dennard's model broke down in the first years of the last decade.

Figure 1.1: Intel microprocessors evolution from 1985 to 2003.

(a) Performance. (b) Power.



Source: (OLUKOTUN; HAMMOND, 2005).

Table 1.1: Device scaling in the Dennard and Post-Dennard eras.

	Dennard Scaling	Post-Dennard Scaling
Device Count	S^2	S^2
Device Frequency	S	S
Device power (cap)	$\frac{1}{S}$	$\frac{1}{S}$
Device power (Vdd)	$\frac{1}{S^2}$	1
Power Density	1	S^2

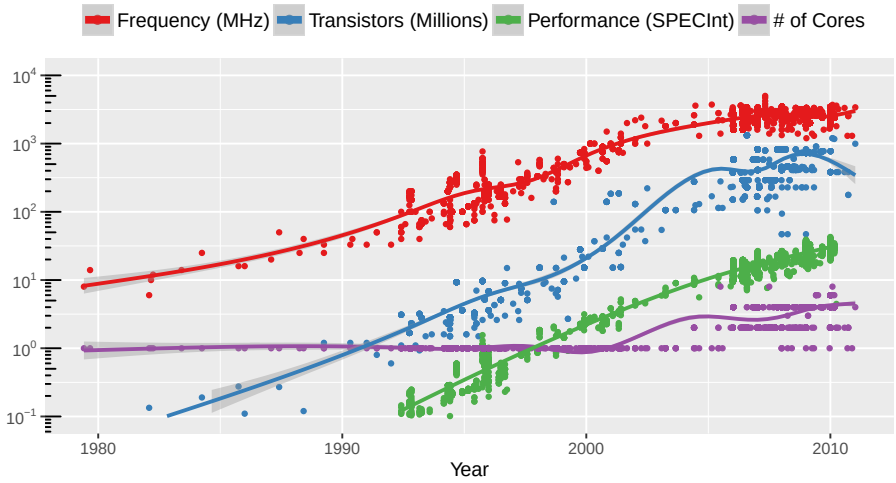
Source: (SHAFIQUE; GARG, 2017).

The *power wall* has led, in the last fifteen years, to a shift in the microprocessor design paradigm, as can be seen in Fig. 1.2. Even though the technology itself allowed for frequency increases, these would incur power overheads that would breach the power wall. Moreover, designers realized that processors had reached a stage in which exploiting additional ILP provided only marginal performance increases due to the practical limitation in the parallelism available from applications (WALL, 1991).

At the same time, however, new application domains emerged with the rise of High-Performance Computing (HPC) systems and the Internet, requiring processing of multiple independent tasks or requests simultaneously. These restrictions and demands led to the development of the first Chip Multi-Processors (CMPs), a solution integrating multiple processor cores with shared memory into the same die which could improve the performance in these emerging workloads without the power overheads of frequency increases (BORKAR et al., 2005). Besides accelerating applications with independent concurrent tasks, multicore processors offered the possibility of scalable linear speedups even to a single application. However, in that case, programmers were required to segment the applications into independent tasks to expose the Thread-Level Parallelism (TLP) that they presented - a task which is often non-trivial (BLAKE et al., 2010).

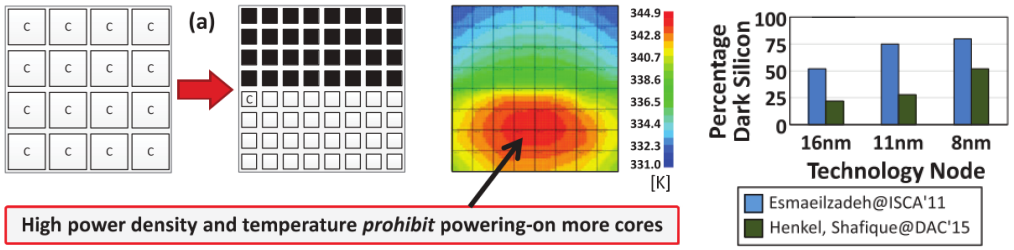
The CMPs solution just describedd provided for many years a scalable solution to increase performance, but this approach has also been challenged in the last decade (ESMAEILZADEH et al., 2012). Voltage scaling has significantly slowed down to avoid exacerbating static power, an issue which becomes more critical as technology scales (MUDGE, 2001) and has lead to even more power consumption increases across generations (SHAFIQUE; GARG, 2017). Therefore, modern designs have again reached a stage where the *power wall* prevents the scalability of the current approach. Because of that, not all cores in the chip can be turned on at the same time, an issue which was then named *dark silicon* (TAYLOR, 2013) and is illustrated in Fig. 1.3a. The figure shows the effect of turning on all cores in a contiguous region of the chip simultaneously: power density

Figure 1.2: Microprocessor evolution in the last 30 years.



Source: the author. Data provided by CPU DB (DANOWITZ et al., 2012).

Figure 1.3: The dark silicon problem.
(a) Power densities and temperature profile when all cores in a contiguous region are simultaneously on. (b) Predictions for the dark silicon amount.



Source: (SHAFIQUE; GARG, 2017).

and temperature increase up to a point where the chip may stop functioning correctly. Recent works estimate that the amount of *dark silicon*, i.e., the fraction of the chip that needs to be underutilized to keep it operating within the target power envelope, can reach up to 50% of the chip at the 8 nm technology node (HENKEL et al., 2015), as can be seen in Fig. 1.3b.

1.1 Current challenges in microprocessor design

The end of multicore scaling, in which the tight power envelopes prohibit the straightforward deployment of more cores to improve performance, has caused another significant paradigm change in microprocessor design. Now, more than before, microprocessors are moving away from the concept of general-purpose processing and being designed with particular applications in mind to sustain performance improvements with

power efficiency. To do so, without losing generality, General-Purpose Processors (GPPs) are *extended* with *accelerators*, application-specific designs tailored for efficiently exploiting the sort of parallelism available in each application (PATEL; HWU, 2008). While accelerators allow for the best possible efficiency by perfectly matching the application to the hardware, their use introduces a systemic cost impact (in hardware design and software development) which persists despite several advancements in the field (HWU; PATEL, 2018). From the hardware design perspective, since processors become specialized at design time to a particular (group of) application(s), the target market for each design is restricted, increasing the Non-Recurrent Engineering (NRE) costs. Besides, designers must now have an understanding of the application domain, making them a more valuable (and costly) resource than earlier. From the software development perspective, because accelerators must be programmed using special instructions, the code must be (re)structured in order to leverage the accelerator efficiently, or special (automatic) tools must be deployed to that end. In any case, extensive development effort or programmer's training in the tools is required, both of which increase production costs.

1.2 Scope of this thesis

The above discussion suggests that, while the excellent match between software and hardware makes accelerators highly-efficient execution units, capable of addressing the performance and power challenges just described, they introduce significant costs to the hardware and software development processes. Besides, since their design is fixed, they cannot adapt well to workload sets that change over time. In domains where applications are constantly evolving at a fast pace, such as in the Internet-of-Things (IoT), application-specific designs may have a too high cost that prohibits their utilization (ADEGBIJA et al., 2018). As a consequence, *accelerators* could be significantly improved if they were *more generic*, had *smaller programmability costs* and had better *run-time adaptation* capabilities.

Considering this context, *reconfigurable accelerators* present an alternative that can address most of the issues associated with application-specific ones. These devices typically consist of arrays of Processing Element (PE) with programmable interconnects, allowing customized datapaths that match each application's needs to be defined at run time. While not as efficient as an *application-specific* accelerator, they can provide considerable efficiency improvements compared to GPP while amortizing the design costs

and increasing the range of applications the device can run. Moreover, *reconfigurable accelerators* can be leveraged transparently for automatic acceleration of code that is already deployed with techniques such as dynamic binary translation, reducing the cost overheads in the development process. Moreover, as will be shown in this thesis, they provide a suitable structure for combining other techniques, such as *approximate computing* and *near-threshold voltage* computing.

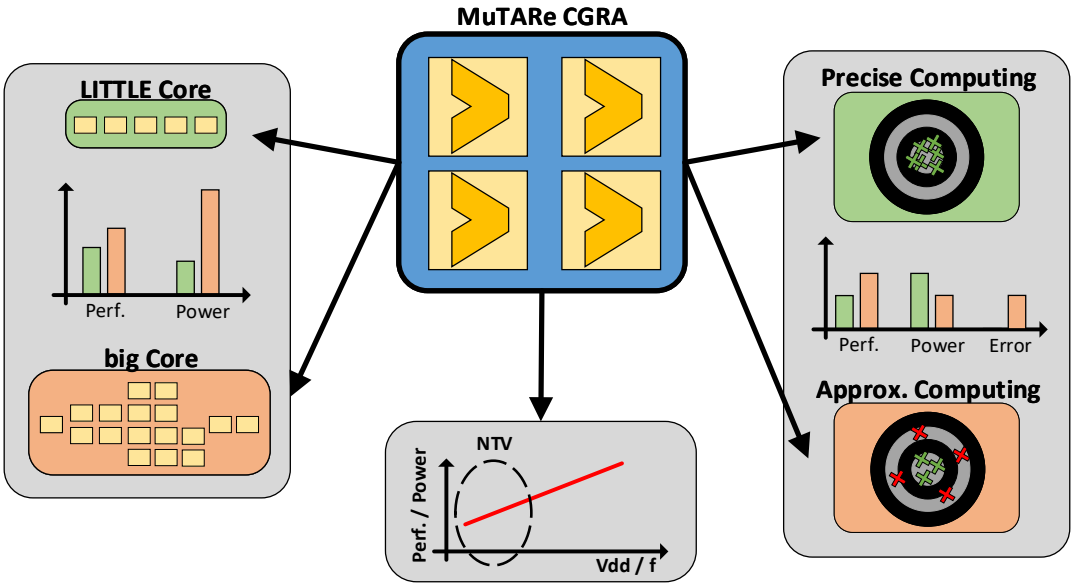
1.3 Contributions of this thesis

To address the challenges in designing and programming accelerators, and to provide a wider adaptability range than current reconfigurable accelerators, this thesis presents Multi-Target Adaptive Reconfigurable Architecture (MuTARe). MuTARe synergistically combines multiple adaptability techniques for transparent adaptation between different applications and the hardware when compared to current reconfigurable designs. It is *multi-target* in the sense that it can be *targeted* towards different application domains and, besides adapting to each application, can also dynamically adapt to different *target* metrics, such as meeting a performance target while saving power or meeting a power target while maximizing performance. MuTARe goes beyond traditional reconfigurable architectures by providing support for two emerging computing paradigms that can further boost the power efficiency: *near-threshold voltage computing* and *approximate computing*.

Fig. 1.4 presents an overview of the MuTARe architecture. The heart of MuTARe is a parametrisable and combinatorial Coarse-Grained Reconfigurable Array (CGRA) that can be coupled to different forms of GPP core: *in-order* cores (for low-power domains, such as IoT), *Out-Of-Order (OoO)* cores (for high-performance domains, such as HPC), or even both of them in a *big.LITTLE*-like setup (for mobile domains requiring a wide adaptability range). The acceleration capabilities can be combined with Dynamic Voltage and Frequency Scaling (DVFS) to precisely adjust for the performance levels required, lowering the Operating Frequency (f) and Operating Voltage (V_{dd}) when possible to reduce power consumption. With these techniques, MuTARe can work transparently for already deployed binaries by providing, as a dedicated hardware module, a dynamic binary translation algorithm that automatically maps recurring instruction sequences into the CGRA for acceleration.

In the first step to move beyond traditional reconfigurable architectures, MuTARe

Figure 1.4: Overview of the proposed MuTARe architecture.



Source: the author.

provides support for the operation in the Near-Threshold Voltage (NTV) range, where the lowest-energy operating point is typically found (MITTAL, 2015). In this challenging operating environment, memories become more prone to failure, and the effects of Process-Voltage-Temperature (PVT) variation are increased, requiring the use of special techniques during design to address these issues. MuTARe avoids these difficulties by providing a suitable structure for NTV computing: a combinatorial CGRA with a separate voltage domain from the memories, which is also a regular structure that may be designed with overprovisioned PEs to address variability issues. This operating mode can also be activated transparently, with no need to recompile existing code, and provides a considerable boost to power efficiency especially in memory-bound applications.

In a second step to improve over reconfigurable architectures, MuTARe provides support for Approximate Computing (AxC) to improve power efficiency in emerging error-tolerant workloads. While many works have shown the power benefits of deploying approximate functional units (SHAFIQUE et al., 2016), GPPs are typically unsuitable for this approach since the most significant fraction of power consumption is spent in control, rather than in processing. When moving the execution from the GPP to a combinatorial CGRA, however, the potential benefits of approximate functional units can be leveraged to their full extension. Support for approximate computing requires Instruction-Set Architecture (ISA) extensions to configure the accuracy, since that is a piece of semantic information that must be provided by the application developer, so this execution mode

introduces to the base architecture Instruction-Set Extensions (ISEs) that may be used towards that end.

The contributions of this thesis can be summarized as follows:

- A reconfigurable architecture based on a GPP-CGRA coupling, where the GPP may take any form of core (including heterogeneous arrangements with complex OoO cores), which supports DVFS to balance the acceleration capabilities and power consumption. Compared to existing reconfigurable architectures, MuTARe can transparently accelerate existing code, be coupled to any form of processor core and can leverage DVFS to match the performance improvements provided by automatic CGRA acceleration with the performance target, lowering the frequency if slack is available to save additional power.
- A reconfigurable architecture with a suitable structure for computing in the NTV domain. Compared to previous works that use NTV to save power, this one is the first where NTV is used in the context of single-threaded applications. The enhanced ILP exploitation provided by MuTARe's CGRA can partially compensate the performance loss from low-frequency operation, especially in workloads which are memory-bound, and its reconfigurable fabric enables simpler approaches to variablility management.
- A reconfigurable architecture that can leverage the benefits of AxC and provide additional performance improvements and power savings in emerging error-tolerant domains. Compared to existing works oin AxC, this is the first one where a *reconfigurable accelerator* is used for approximate computations, presenting the advantage of maintaining general-purpose processing capabilities while leveraging the full benefits of reduced power consumption in approximate functional units, since the power consumption is switched from control in the GPP to computatin in the CGRA's PEs.

The result is an architecture that can be tuned at design time to better adapt for low-power or high-performance, and at run-time adapt to the application being executed and improve the efficiency.

1.4 Structure of this thesis

The current chapter has presented an introduction to this thesis, with the problem it addresses, the scope and an overview of the solution: the MuTARe architecture. The remainder of this document is organized as follows.

Chapter 2 present background information for understanding this thesis. It covers reconfigurable accelerators, DVFS and NTV computing, and approximate computing, all of which are concepts exploited by MuTARe.

Chapter 3 presents the base architecture of MuTARe, a GPP core to which a reconfigurable accelerator with dynamic instruction mapping support is coupled and which can leverage DVFS to adapt to a performance target.

Chapter 4 presents extensions to increase the adaptability range of MuTARe, namely MuTARe with NTV support, and approximate MuTARe.

Chapter 5 presents the methodology and tools used to evaluate MuTARe under very distinct comparison scenarios.

Finally, Chapter 6 presents the conclusions of this work, summarizing the vital contributions and results, points towards future research directions and presents the publications by the author in the scope of this thesis.

2 BACKGROUND

This chapter presents background information essential for the understanding of this thesis. It covers reconfigurable architectures, DVFS, and NTV computing, and approximate computing. For each of these techniques, examples of systems that use them are presented and also the key novelties of the MuTARe architecture.

2.1 Reconfigurable Architectures

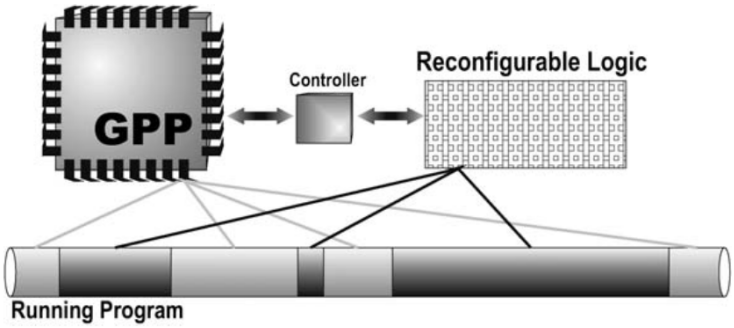
*Reconfigurable accelerator*s are hardware units with the ability to specialize to an application at run time by modifying their structure. Systems where these accelerators are coupled to GPP are referred to as Reconfigurable-Instruction-Set Processors (RISPs) (BARAT; LAUWEREINS, 2000)[1]. Through this specialization, reconfigurable accelerators are able to achieve better performance and energy consumption than the GPPs they are coupled to; however, given the flexibility to adapt the hardware structure at run time to different applications, they are usually not as efficient as dedicated Application-Specific Integrated Circuits (ASICs). In summary, these circuits fill a gap between software and hardware implementations of algorithms (COMPTON; HAUCK, 2002; BECK; CARRO, 2010).

Fig. 2.1 shows how execution works in a RISP. Instructions sequences are executed either in the GPP (when presenting low potential for acceleration, as shown in light gray in the figure) or in the reconfigurable accelerator (when they have high potential for acceleration, as shown in dark gray). A hardware controller handles the configuration, communication and synchronization process between the two execution units. To determine the code sequences which should execute in the accelerator, the hot regions of code (i.e. regions which account for a high fraction of the application's execution time or energy consumption) must be selected (step 1) and transformed into reconfigurable instructions (step 2).

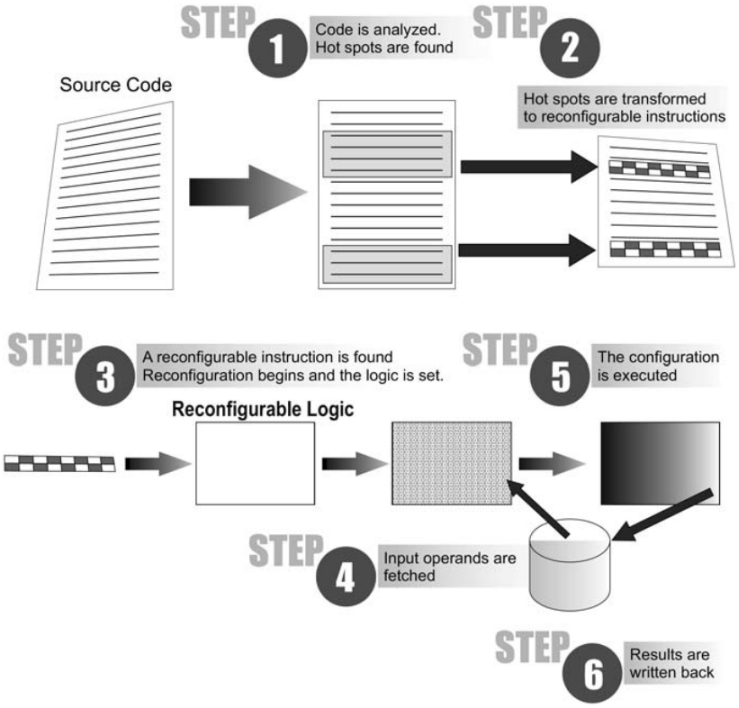
This section proceeds by discussing a classification scheme for RISPs, and then presents relevant implementation examples considering the scope of this work.

[1]There is no standard taxonomy in the field, so this form of system has been called many names in the past years, such as *reconfigurable architecture/system*, *spatial architecture*, and some authors even avoid using any specific terms at all, speaking instead of "*a system with configurable components*".

Figure 2.1: Overview of a RISP.
(a) Basic principle.



(b) Steps required to map, execute, and write back program regions in the reconfigurable accelerator.



Source: (BECK; CARRO, 2010).

2.1.1 Classification of Reconfigurable Architectures

This work uses a classification scheme similar to the one used by Beck and Carro, which covers (among others) three criteria: *code analysis and transformation*, *granularity*, and *processor coupling* (BECK; CARRO, 2010).

Fig. 2.1b presents a detailed view of each of the steps involved when executing an application in a RISP, from application source code specification to execution results.

The **code analysis and transformation** phase, depicted in steps 1 and 2, can be carried out either **statically** (*offline*, at compile time) or **dynamically** (*online*, at run time). Static schemes are simpler because the application source code is available, but they usually rely on programmer intervention (to identify regions of code that will be accelerated) or compiler modifications (to automatically select these regions). Dynamic schemes, on the other hand, although more complex to implement, present the following advantages:

- They take as input the application binary rather than the source code; therefore, both already-deployed as well as new applications are supported and can leverage the reconfigurable unit for acceleration.
- They have access to dynamic information (such as the frequency of executed code regions) and, therefore, optimization opportunities that are not available at compile time.

The process of dynamically transforming code from one ISA for execution in another one (e.g. a reconfigurable accelerator) is commonly named *binary translation* (ALTMAN; KAELI; SHEFFER, 2000). Most of these strategies use graph analysers that work on an application's Data-Flow Graph (DFG) to determine computations that can be transformed into reconfigurable instructions. To illustrate how this process is done, consider Fig. 2.2, which depicts the instructions in an application's basic block (graph nodes) and the data dependencies among them (graph edges). In this particular case, four code regions with data dependencies were transformed into reconfigurable instructions (denoted by CCA in the transformed DFG) that can execute in a single cycle, reducing the height of the DFG from 8 to 4. The height of the DFG represents the number of cycles it would take to execute the kernel in case an unlimited amount of functional units were available (, each of which can execute each instruction in a single cycle). Therefore, in this example, the potential speedup is $\frac{8}{2} = 2\times$.

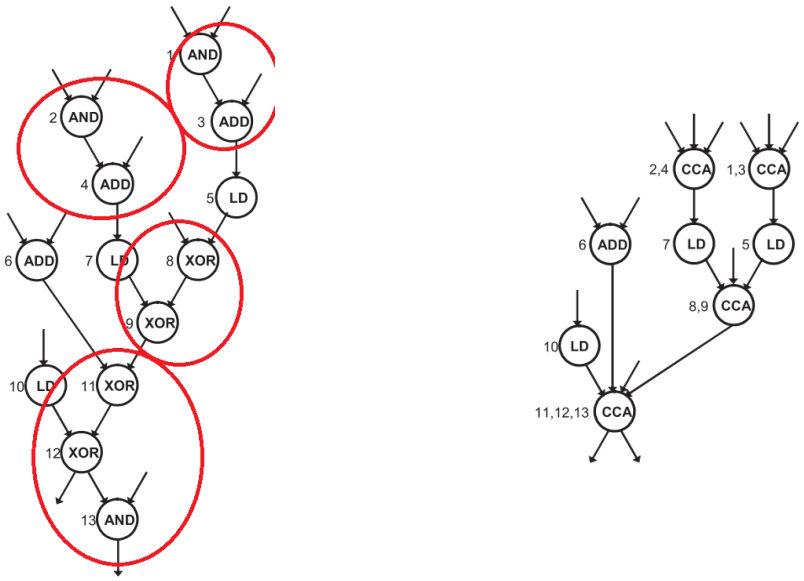
The accelerator's **granularity** refers to *size* of the reconfigurable logic blocks. There are two categories:

- **Fine-Grained Reconfigurable Arrays**, in which the reconfigurable logic blocks implement bit-level operations;
- **CGRAs**, in which the reconfigurable logic blocks implement word-level operations.

To illustrate the difference, consider Fig. 2.3, which compares both granularities. The Fine-Grained Reconfigurable Array is implemented using an FPGA, which consists of Configurable Logic Blocks (CLBs) and switch-boxes. Each CLB contains an N-input

Figure 2.2: Code transformation in a RISP.

(a) Original DFG. (b) Transformed DFG.



Source: (CLARK et al., 2004).

Lookup Table (LUT) and a flip-flop; they may be connected arbitrarily by configuring the switch-boxes and therefore implement logical functions of multiple inputs and outputs. The CGRA is a matrix of Arithmetic-Logic Units (ALUs) with crossbar connections between the rows.

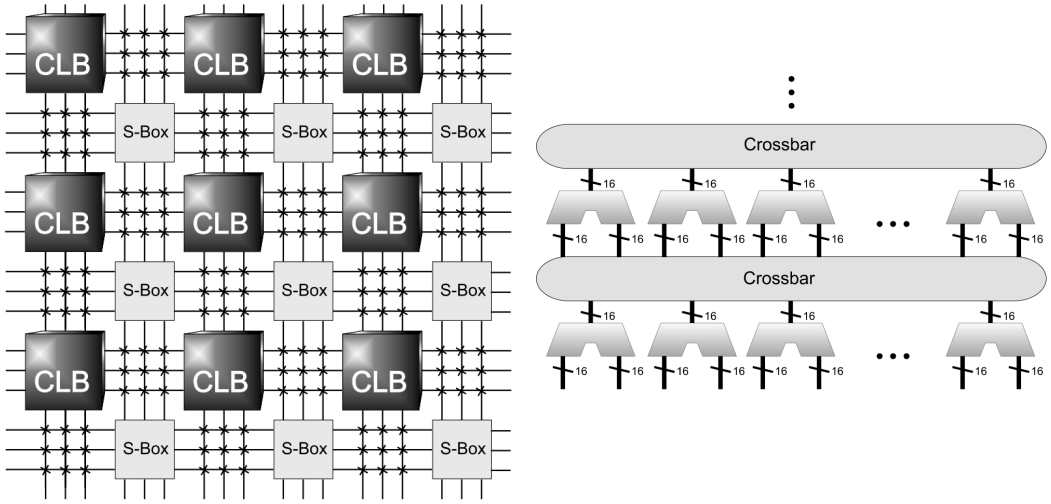
While Fine-Grained Reconfigurable Arrays (FGRAs) provide the the highest flexibility, there is a tradeoff: the configurations are significantly larger than in CGRAs (as more flexibility implies more configuration bits), and therefore they require larger storage space and also longer configuration times. For this reason, FGRAs are more effective in cases where a circuit needs to be reconfigured only a few times during execution, such as in filter applications (which execute a single hotspot the whole time), and CGRAs tend to be more efficient to implement applications consisting of multiple hotspots.

Coupling refers to how the reconfigurable accelerator synchronizes execution and communicates data with the GPP in the system. There are two forms: the accelerator can either be located inside the processor core (having direct access to the register file and first-level caches), in which case it is classified as *tightly-coupled*, or it can be located outside, in which case it is classified as *loosely-coupled*. There is a tradeoff in communication time and circuit speed. Loosely-coupled accelerators can include private memories and operate in a distinct frequency domain as the main processor, and therefore do not affect its critical path. On the other hand, communication must be done via higher-level caches or main memory, which can be significantly slower than in tightly-coupled accel-

Figure 2.3: Comparison between an FPGA and a CGRA.

(a) Field-Programmable Gate Array (FPGA).

(b) CGRA.



Source: (BECK; CARRO, 2010).

erators.

A recent work has investigated the tradeoffs involved in computing using tightly-coupled and loosely-coupled accelerators, and shows the performance advantages of loosely-coupled ones with private memory (COTA et al., 2015). This work, however, uses a tightly-coupled accelerator, since the goal is to provide transparent reconfigurable acceleration of alraedy deployed code (without needing to rewrite code for using local memories) and fast reconfiguration times.

2.1.2 Implementations of Reconfigurable Architectures

Reconfigurable architectures have been studied for quite some time and have been the subject of many surveys over the years (BARAT; LAUWEREINS, 2000; COMPTON; HAUCK, 2002; BECK et al., 2008; WIJTVLIET; WAEIJEN; CORPORAAL, 2016). Most of these architectures use static strategies to generate instructions for the reconfigurable fabric. Since this work uses a dynamic strategy, static architectures are only briefly described here due to their historical significance.

Concise (KASTRUP; BINK; HOOGERBRUGGE, 1999) and Chimaera (YE et al., 2000) use a tightly coupled Reconfigurable Unit (RU) that works as an ordinary functional unit and limited to combinational logic only. The GARP machine (CALLAHAN; HAUSER; WAWRZYNEK, 2000) comprises a MIPS-compatible processor with a

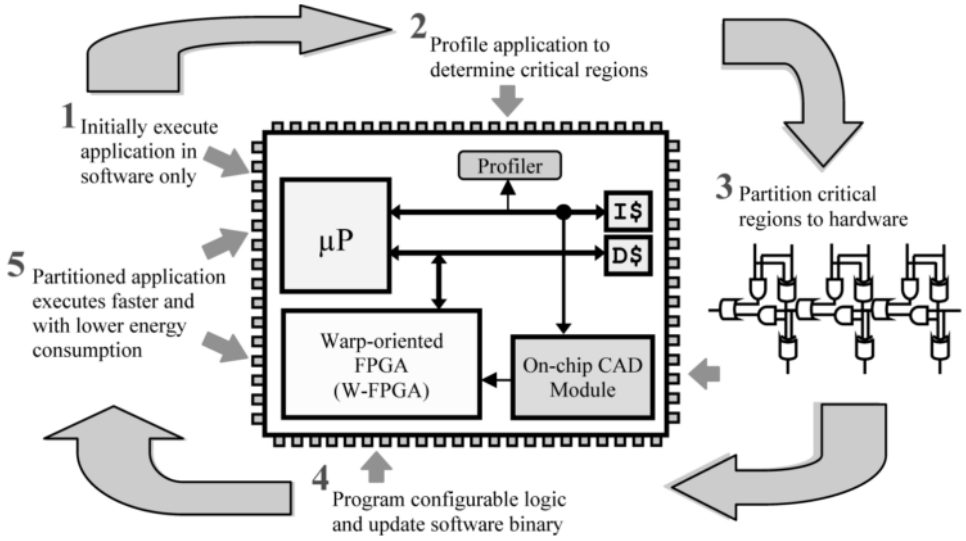
loosely-coupled and fine-grained RU. REMARC (MIYAMORI; MIYAMORI; OLUKOTUN, 1998) also uses a fine-grained RU and works as a loosely coupled coprocessor. RaPiD (CRONQUIST et al., 1998) and Piperench (GOLDSTEIN et al., 2000) are examples of architectures using coarse-grained RUs. The main novelty of the Piperench architecture is the concept of *pipelined reconfiguration*: given kernel is broken into pieces that can be reconfigured and executed on demand. Afterwards, in a process called virtualization, they are multiplexed in time and space to be executed in the reconfigurable logic. The Molen (VASSILIADIS et al., 2004) microcoded RU is fine-grained, loosely-coupled, and works together with a PowerPC processor core. DISC (WIRTHLIN; HUTCHINGS, 1995), OneChip (WITTIG; CHOW, 1996), PRISM-II (WAZLOWSKI et al., 1993) are other reconfigurable architectures that employ standard fine-grained FPGA resources. In the group of coarse-grained RUs, one could also include: Pact-XPP (BAUMGARTE et al., 2003), Morphosys (SINGH et al., 2000), Pleiades (ZHANG et al., 2000) and ADRES (MEI et al., 2003). Furthermore, there are reconfigurable architectures that are very similar to dataflow machines. For instance, TRIPS is based on a hybrid von-Neumann/dataflow architecture that combines an instance of coarse-grained, polymorphous grid processor core with an adaptive on-chip memory system (SANKARALINGAM et al., 2003). TRIPS uses three different execution modes, focusing on instruction-, data- or thread-level parallelism. Wavescalar (SWANSON et al., 2003) is another example, and its implementation is very similar to the structure found in TRIPS.

All of the systems just described require dedicated instructions implemented in the ISA to program the RU. Next, a few systems supporting automatic code generation are described in more detail due to their relevance to the scope of this thesis.

The **Warp Processor**, shown in Fig. 2.4, is one of the first systems to use a dynamic strategy to map application code while it executes into a RU (LYSECKY; STITT; VAHID, 2006). The Warp Processor consists of an ARM core, a profiler, a dedicated Computer-Aided Design (CAD) processor and a simplified FPGA, and works as follows. While the program executes normally in the ARM core, the profiler identifies the critical kernels within the application. Then, the dedicated CAD processor executes special CAD tools to transform the software regions into custom hardware using the FPGA's resources. The program binary is then updated to run critical kernels in the FPGA rather than the ARM core. A mutually-exclusive execution model is used to switch execution from the ARM processor to the custom hardware next time one of the critical kernels is executed. Since the main processor and the FPGA share the same data cache, issues with cache

coherency and consistency are avoided.

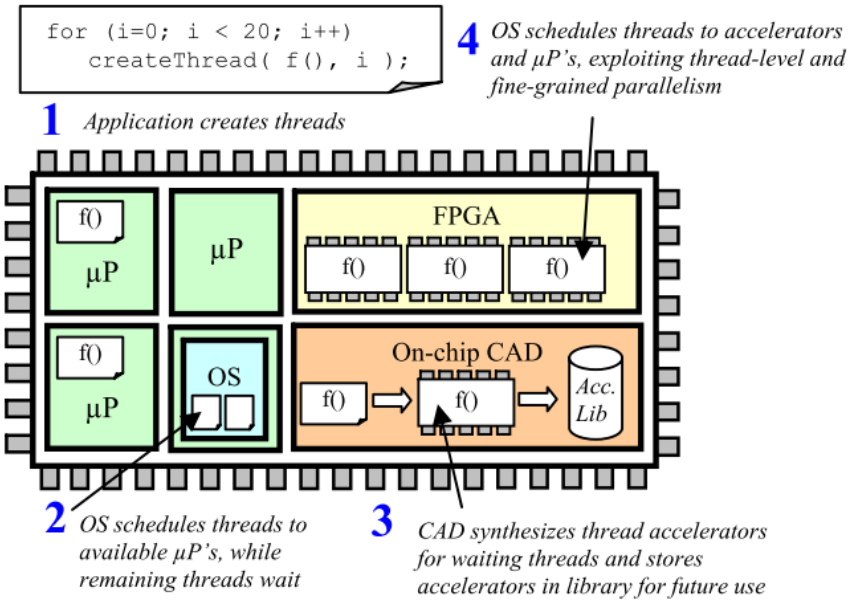
Figure 2.4: Overview of the Warp Processor.



Source: (LYSECKY; STITT; VAHID, 2006).

The **Thread Warping** approach, shown in 2.5, extends the Warp Processor to multicore systems, sharing a single reconfigurable FPGA fabric across multiple cores (STITT; VAHID, 2011). The approach uses operating system support to identify thread functions to map to the FPGA.

Figure 2.5: Overview of the Thread Warping approach.

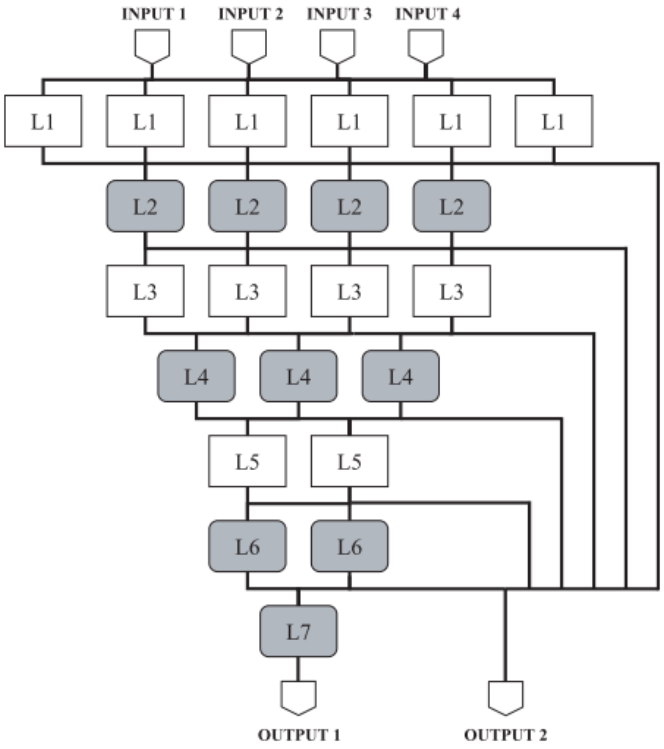


Source: (STITT; VAHID, 2011).

The **Configurable Compute Accelerator (CCA)** (CLARK et al., 2004), depicted in Fig. 2.6, uses a configurable matrix of functional units tightly coupled to the processor

pipeline as an additional functional unit. It can be used to implement regions of code with four inputs and two outputs consisting of basic logic and arithmetic operations. Static and dynamic approaches are proposed for generating custom CCA instructions from application kernels. In the dynamic approach, application kernels are discovered at runtime using a graph analyser and transformed into special CCA instructions, which replace the Central Processing-Unit (CPU) ones in the system's trace cache. CCA instructions may not include memory accesses, which restricts their scope, and are limited to very short regions with small input sizes.

Figure 2.6: Overview of the CCA hardware.

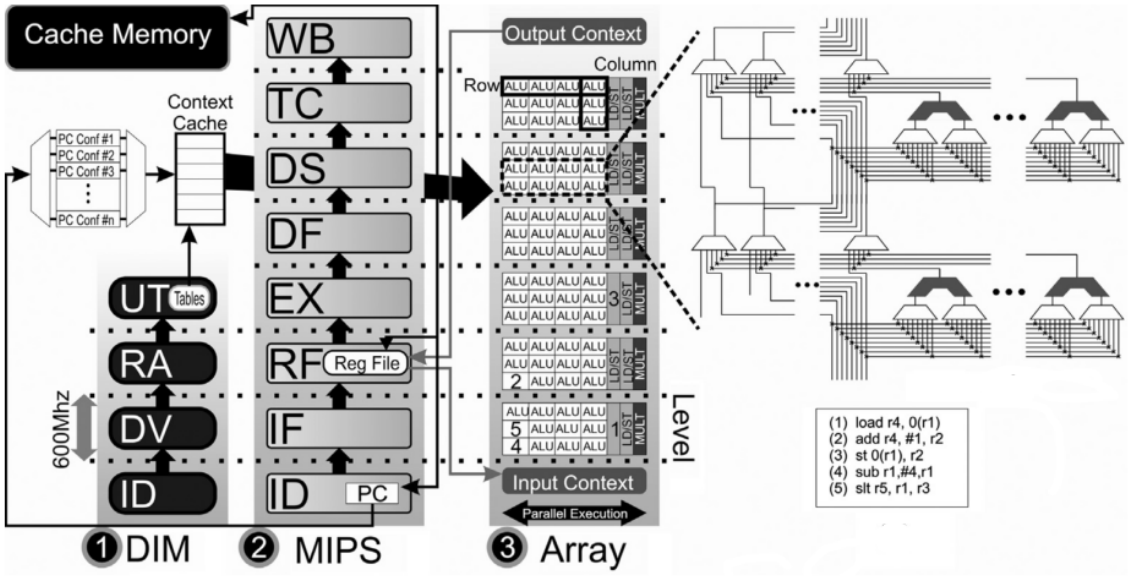


Source: (CLARK et al., 2004).

The **Dynamic Instruction Merging (DIM)** system, shown in 2.7, uses a CGRA that is tightly coupled to a MIPS processor (BECK; RUTZIG; CARRO, 2014). With a simplified hardware binary translation algorithm, application kernels are dynamically transformed into CGRA configurations and saved in a dedicated cache. Unlike the CCA, the system accesses the dedicated cache using the program counter Program Counter (PC) of the application kernels to check if they can execute in the array.

Custom Reconfigurable Arrays for Multiprocessor Systems (CReAMS) extends DIM by proposing an homogeneous system consisting of multiple reconfigurable processors, thereby also allowing the exploitation of TLP (RUTZIG; BECK; CARRO, 2013). **Heterogeneous Arrays for Reconfigurable and Transparent Multicore Pro-**

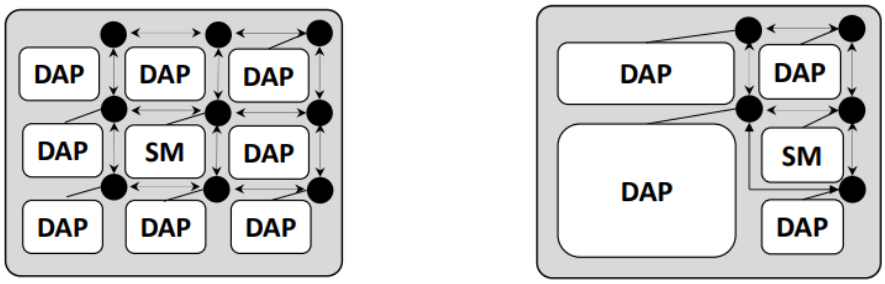
Figure 2.7: Overview of the DIM system.



Source: (BECK; RUTZIG; CARRO, 2014).

cessing (HARTMP) improves over CReAMS by considering a multi-core system of heterogeneous processors with homogeneous ISA (SOUZA et al., 2016). Fig. 2.8 illustrates the difference between both approaches.

Figure 2.8: Overview of the CReAMS and HARTMP systems.
(a) CReAMS. (b) HARTMP.



Source: (SOUZA et al., 2016).

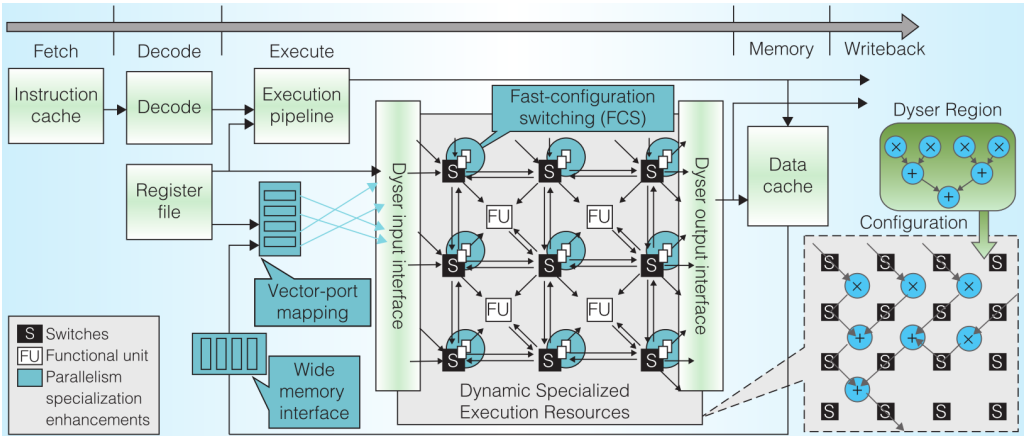
Dynamically Specializing Execution Resources (DySER) is a coarse-grained reconfigurable accelerator, tightly coupled to the processor's execution unit, with the claim to unify *functionality and parallelism specialization* into a single hardware (GOVINDARAJU et al., 2012). This is an interesting distinction made by the authors which deserves better clarification:

- *parallelism specialization* uses homogeneous hardware resources with wide and independent interconnects. Vector processors, Single Instruction Multiple Data (SIMD), and Graphics Processing Unit (GPU) are strategies that exploit this form

of specialization;

- *functionality specialization* uses heterogeneous, task-specific resources and routing. The works discussed in this section are examples of strategies to exploit this form of specialization.

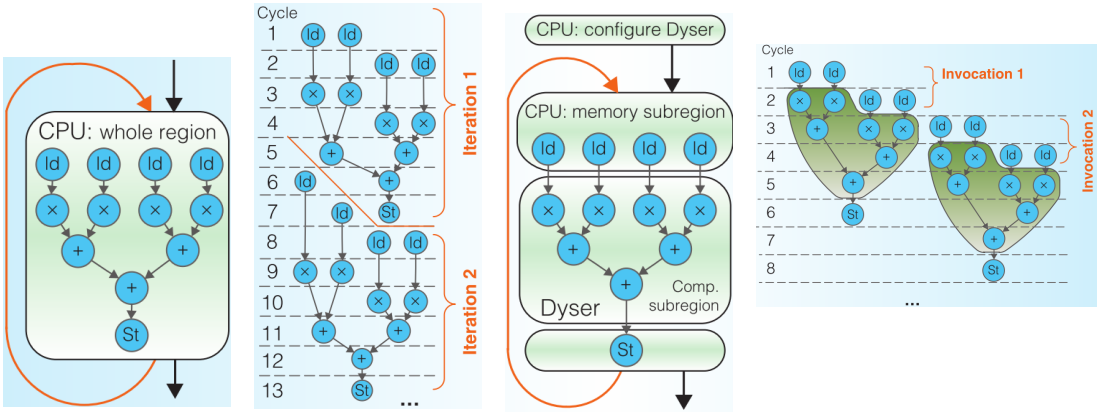
Figure 2.9: Overview of the DySER system.



Source: (GOVINDARAJU et al., 2012).

An overview of the DySER system is presented in Fig. 2.9. The reconfigurable accelerator consists of a matrix of heterogeneous functional units surrounded by programmable switches that can route data to the adjacent tiles, forming a hardware data path once configured. This accelerator is integrated to the processor's execution stage, which acts as a load/store engine to feed the DySER hardware.

Figure 2.10: Code transformation and execution in the DySER system.
(a) DFG. (b) CPU execution. (c) DySER transformation. (d) DySER execution.



Source: (GOVINDARAJU et al., 2012).

A comparison between CPU and DySER execution is provided in Fig. 2.10. In Fig. 2.10a, the DFG of a selected code region is presented. It is a 4-input, 1-output

code pattern (similar to a reduction) consisting of multiplications and sums. Fig. 2.10b shows how this sequence would execute in a 2-issue superscalar processor when invoked twice. A total of 13 cycles are required. Fig. 2.10c shows how this DFG is adapted for execution in DySER: the computation subregion is extracted, and the loads/stores are left for execution in the CPU to exploit its structures that support memory parallelism. Finally, Fig. 2.10d shows how this same execution occurs in DySER, and the opportuniity to pipeline both invocations resulting in a total of only 8 cycles to execute the region. DySER was originally programmed by using a special compiler that selected the regions that will be offloaded to the RU. Later, the **DORA** automatic code generation system was developed for DySER, with support for automatic code transformation (WATKINS; NOWATZKI; CARNO, 2016).

DynaSpAM, presented in Fig. 2.11, is a reconfigurable accelerator coupled to an OoO superscalar pipeline (LIU et al., 2015). The insight of the work is that the existing OoO scheduling logic can be leveraged, with some adaptions, to simultaneously allocate instructions in the pipeline and translate instruction sequences for execution in a CGRA. Moreover, an additional Re-Order Buffer (ROB) is used to hold results of the instructions executed in the CGRA, allowing for in-order commit as is done in the superscalar.

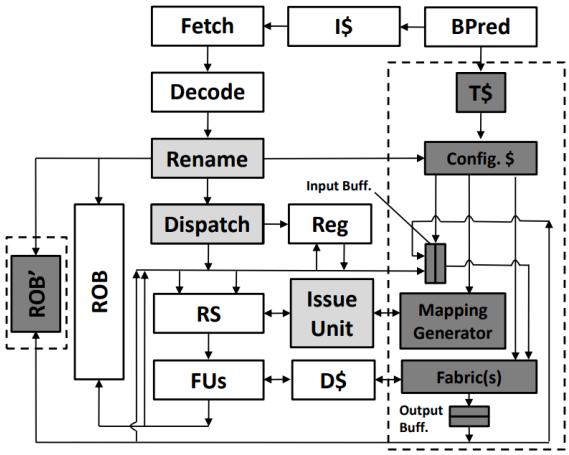
2.2 DVFS and NTV Computing

Considering the relationship between V_{dd} and power consumption, many modern processors support DVFS, a technique that allows tuning the processor's V_{dd} and frequency at run time, providing a knob to trade high performance (by setting the frequency to a high level) for improved power and energy consumption (by lowering the V_{dd} and the frequency) (BURD et al., 2000).

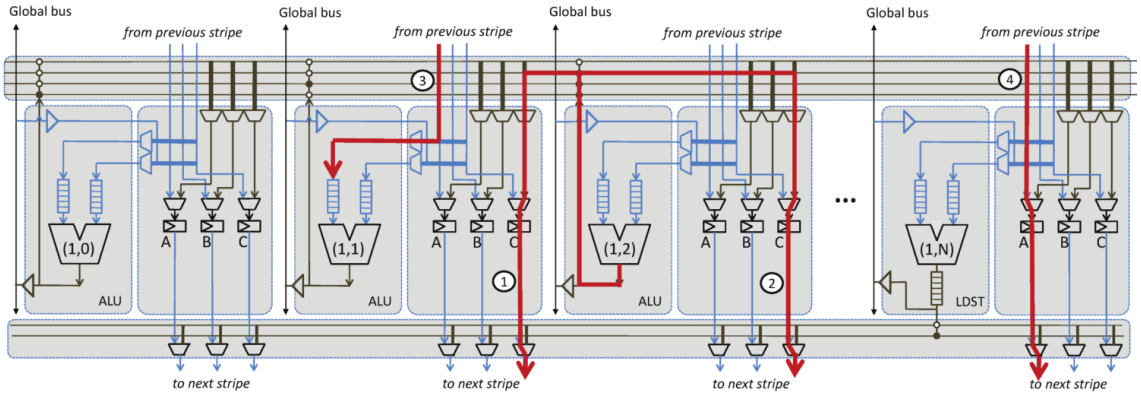
The Alpha-Power Law model provides a good approximation for the expected relationship between V_{dd}, f, and transistors' power consumption (SAKURAI; NEWTON, 1990). The primary knob in DVFS is V_{dd}, since it impacts both static and dynamic power consumption. As V_{dd} is reduced, however, capacitance charge time increase as a result from smaller currents. As a consequence, the circuit must run at a lower frequency (f). The relationship between delay $T = \frac{1}{f}$ and V_{dd} can be expressed as

$$T \propto \frac{V_{dd}}{(V_{dd} - V_{th})^{\alpha}}. \tag{2.1}$$

Figure 2.11: The DynaSpAM system.
(a) Overview.



(b) Reconfigurable fabric.



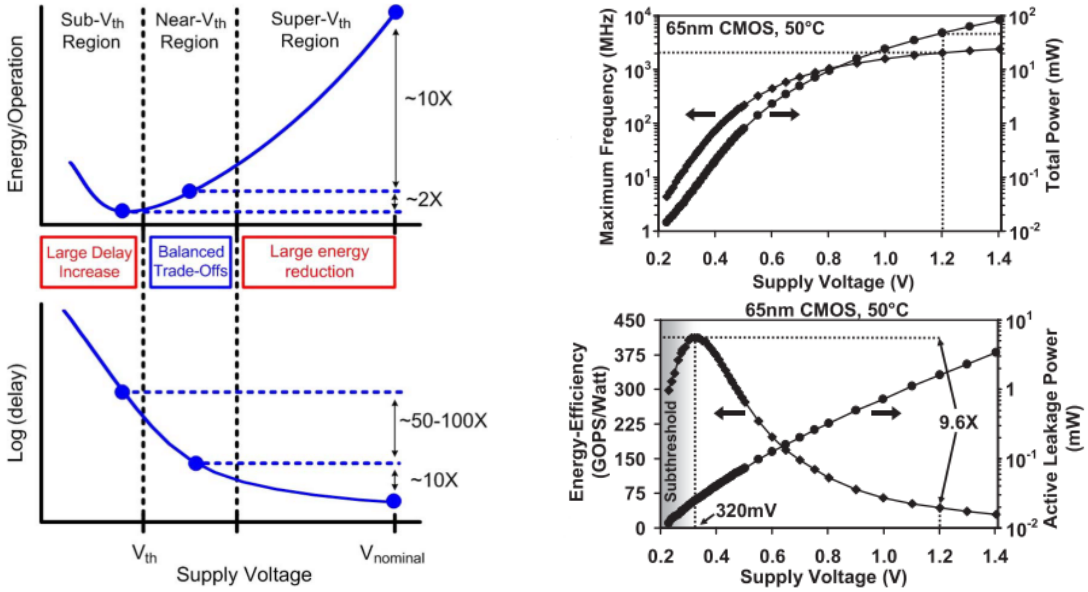
Source: (LIU et al., 2015).

In the equation above, α is a technology-dependent parameter named *velocity saturation index*, estimated at around 1.9 for current 7nm nodes (GUO et al., 2017), and V_{th} is the transistor's Threshold Voltage, estimated at about 159 mV for the 7nm FinFET node (PINCKNEY et al., 2017)[2]. This relationship shows that, as V_{dd} is lowered from the nominal voltage, the frequency initially descreases by only a small amount, V_{th} is still much smaller than V_{dd}. However, as V_{dd} approaches V_{th}, Eq. 2.1 becomes more sensitive to changes in V_{dd} and the impact in delay is larger. This effect can be visualized in the lower part of Fig. 2.12.

The result of DVFS is a major, monotonical reduction in static and dynamic power

[2]This is only a reference value for the Threshold Voltage (V_{th}), since V_{th} can actually be tuned at design time to balance between static power consumption and performance.

Figure 2.12: Effects of voltage scalability in delay and energy consumption.



Source: To the left, results by Univ. Michigan (DRESLINSKI et al., 2010); to the right, results by Intel (KAUL et al., 2012).

consumption (P_S, P_D), since

$$\begin{aligned} P_D &\propto V_{dd}^2 f \\ P_S &\propto V_{dd} I_S \end{aligned} \tag{2.2}$$

In Eq. 2.2, I_S is the current in the transistor's *source*, which is also a function of V_{dd}. While power consumption decreases with V_{dd} and f, however, the energy consumption depends on the ratio between the increase in task delay and the reduction in power consumption, since $E = PT$ (where $T = \frac{1}{f}$). Plugging together the relationships in Eq. 2.1 and Eq. 2.2 yields:

$$\begin{aligned} E_D &\propto V_{dd}^2 \\ E_S &\propto V_{dd} I_S T \end{aligned} \tag{2.3}$$

Eq. 2.3 shows that the dynamic energy monotonically decreases as V_{dd} is lowered, since the dynamic power depends on f. The static energy, however, initially decreases as V_{dd} is lowered (influence of V_{dd} and I_S, with small increase in T) but then presents a tipping point as the task delay increases faster when approaching V_{th} (large influence of T). Since $E(V_{dd}) = E_S(V_{dd}) + E_D(V_{dd})$, there's a value of V_{dd} that minimizes energy consumption by balancing the increase in static energy with the decrease in dynamic energy, and this tipping point happens exactly where

$$\frac{\partial E_S}{\partial V_{dd}} = -\frac{\partial E_D}{\partial V_{dd}} \tag{2.4}$$

Different works have investigated the operating point (value of V_{dd}) where Eq. 2.4 is satisfied and execution can be carried out with minimum energy consumption. This value is technology-dependent, however, and hard to estimate precisely. Nevertheless, some preliminary works estimated it to lie in the transistor's subthreshold region, where $V_{dd} \leq V_{th}$ (DRESLINSKI et al., 2010; KARPUZCU et al., 2012). More recent studies, on the other hand, estimate it to lie in the near-threshold region, with V_{dd} slightly higher than V_{th} (KAUL et al., 2012; KHARE; JAIN, 2013; PINCKNEY et al., 2017). These findings, shown in Fig. 2.12, have sparked interest in NTV computing as a means to address the power wall limitation in modern designs (MITTAL, 2015).

Operation in the NTV region, however, raises several challenges that must be addressed with special design strategies. As a consequence, DVFS in typical processors is restricted to the Super-Threshold Voltage (STV) region, with V_{dd} set between 70-100% of the nominal voltage (DRESLINSKI et al., 2010). The design challenges of NTV operation, as well as state-of-the-art techniques used to address those challenges, are presented in detail in the next subsections.

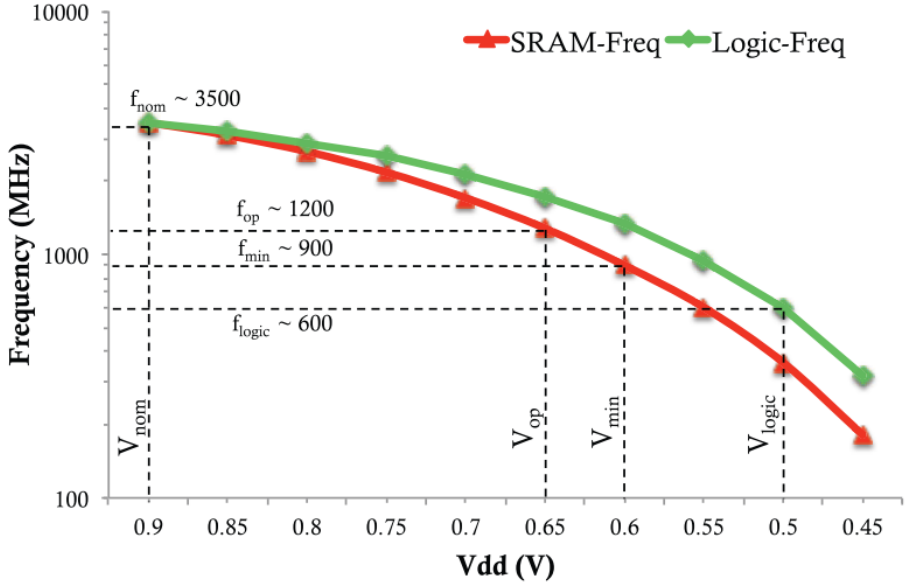
2.2.1 Challenges in NTV Operation

Performance degradation. The first issue with NTV computing is the significant performance degradation from the low-frequency operation, with previous works estimating the delay to increase by about $10\times$ when moving from STV to NTV (see Fig. 2.12). Modern applications, however, require processing speeds compatible with todays's standards. Therefore, alternative strategies must be used to compensate for that performance loss, otherwise NTV operation must be restricted to the few applications without strong performance requirements.

Scalability differences between logic and memory. As voltage is scaled down, the delay of logic and state-holding elements present different responses. In particular, the delay of memories start to degrate faster, and in practice cannot be lowered below a given limit due to increased error vulnerability (BACHA; TEODORESCU, 2014). As a consequence, memories require redesign for NTV or be strategically placed in a separate voltage islands operating at higher levels.

Fig. 2.13 illustrates this effect with experiments carried out in a recent work (GOPIREDDY et al., 2016). It shows how the maximum frequency a logic block and an Static Random-Access Memory (SRAM) block in a processor can run under the ef-

Figure 2.13: Impact in maximum frequency as V_{dd} is lowered (logic vs memory).



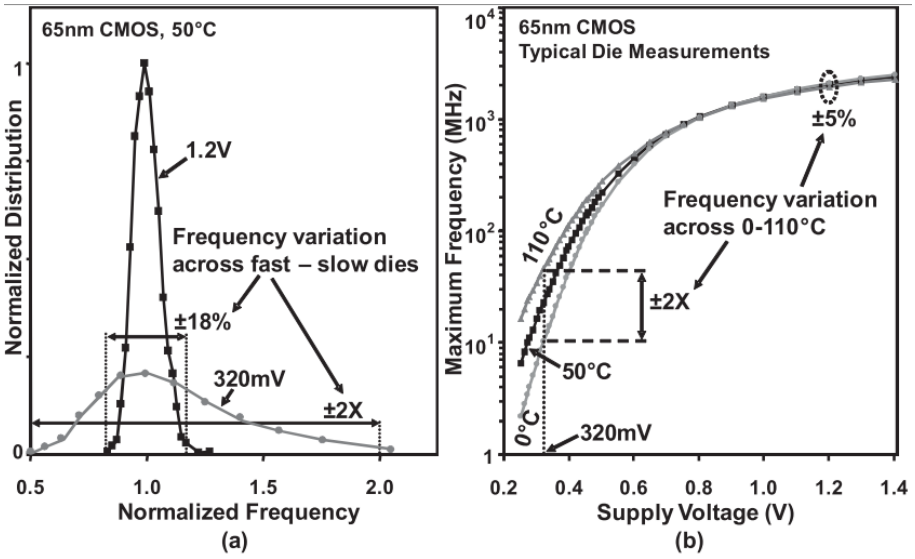
Source: (GOPIREDDY et al., 2016).

fects of voltage scaling, considering both are designed for a nominal *3500 MHz @ 0.9V* (labeled as V_{nom} in the plot). The SRAM voltage can only be lowered down until 0.6V (labeled as V_{min}) as levels below that start affecting the cell's ability to reliably store data. For this voltage level, the memory runs at about 900 MHz; the logic cell can run at the same frequency at a lower voltage level, 0.55V, which, according to equation 2.2, could yield a further 16% reduction in dynamic power consumption given the quadratic relationship with V_{dd}.

Increases in process variability. Process variations affect the transistor parameters non-uniformely across a manufactured processor. These variations result in frequency and power consumption differences among cores in a chip, and are exacerbated when operating in the NTV range. The reason for that is that small deviations in V_{th} (process variation) affect the ratio $\frac{V_{dd}}{V_{th}}$ and the difference $V_{dd} - V_{th}$ more when V_{dd} is closer to V_{th}.

Fig. 2.14 shows this effect precisely. The left plot shows the probability density function for the frequency a core can achieve after manufacturing. The x axis shows the normalized operating frequency and the y axis the probability density of that given frequency. Two curves are shown: one for $V_{dd} = 1.2$V (STV) and one for $V_{dd} = 320$mV (NTV). In the core designed for 1.2V , $\sigma = \pm 18\%$, which means that roughly 70% of all cores will run at a frequency between $0.82\times$ and $1.18\times$ the target frequency. When the chip is designed for $V_{dd} = 320$mV, however, $\sigma = \pm 2\times$, which means that roughly 70% of all cores will run at a frequency between $0.5\times$ and $2\times$. As a consequence, the unpredictabnility increases. As mentioned earlier, when V_{dd} lies closer to V_{th}, the impacts

Figure 2.14: Impacts of variability in STV and NTV voltage levels.



Source: (KAUL et al., 2012).

of V_{th} in frequency after manufacturing are exacerbated according to Eq. 2.1.

The right plot in the same figure shows the effects that temperature may have on the chip, since temperature also affects the V_{th} level. At STV, the difference will be only 5% between a chip operating at 50ºC or 0ºC. At NTV, however, this difference can also be as high as $2\times$.

VARIUS-NTV is a tool developed to model the microarchitectural variability in an NTV processor (KARPUZCU et al., 2012). It is used by most of the NTV works to predict the delay and power variations within a multi-core die.

2.2.2 Solutions for NTV Operation

Increasing performance with dim multi-cores. As discussed in Chapter 1, the scalability of multi-core processors is currently constrained by the power wall. NTV provides a solution to that issue, with nearly-quadritic improvements to power consumption (see Eq. 2.2). As a consequence, an increased number of cores can be simultaneously switched on. This is the key idea introduced by the work of Pinckney et al. (PINCKNEY et al., 2013): by rewriting an application for parallel execution and running it in a voltage-scaled multi-core operating at NTV, converting *dark silicon* (cores turned off) into *dim silicon* (all cores on, at low voltage/frequency), it is possible to reduce energy consumption by an avergage $4\times$ while mantaining the same performance level as in serial execution. A similar analysis considering the impacts of emerging FinFET devices was

carried out more recently (PINCKNEY et al., 2017). According to the study, the benefits of NTV in energy consumption (following the same idea of maintaining the performance levels of serial execution) have dropped significantly in the last nodes of planar transistors, but as the switch to FinFET occurred the gains were boosted due to a major decrease in V_{th} (372mV in the 20nm node and 165mV in the 14nm node). In 7nm, the energy gains from the same setup (dim multicore versus serial execution) are estimated at $8.2\times$.

Most state-of-the-art designs employing NTV use it as a means to reduce the power consumption of each core, enabling more cores to be switched on under the same power envelope. While this approach enables the effective acceleration of parallel workloads, it is not suitable for single-threaded workloads.

Separating voltage islands inside the core. Work by Zhai et al. proposes operating the pipeline and L1 caches in seperate voltage islands, considering the different scalability of V_{dd} for each structure demonstrated in Fig. 2.13. Caches run at twice the frequency as the cores, and therefore can sustain twice the memory bandwidth while still allowing the core logic to be scaled down to optimal levels (ZHAI et al., 2007).

ScalCore extends the approach not only to pipeline and caches, but also to critical structures inside the pipeline which are typically implemented as SRAM, such as the register file, the instruction queues and the ROB (GOPIREDDY et al., 2016). The key insight is to design a core that can achive voltage-scalability, being able to operate in the NTV range and leverage the benefits without, however, incurring any overheads when operating back in STV.

Separating voltage islands inside the chip. A few works have addressed the variability issue by defining multiple voltage and frequency domains in a chip and assigning them values post-manufacturing according to the varability results. Work of Silvano et al. (SILVANO et al., 2014) cites four strategies for this assignment: Single-Voltage Single-Frequency (SVSF), Single-Voltage Multi-Frequency (SVMF), Multi-Voltage Single-Frequency (MVSF), and Multi-Voltage Multi-Frequency (MVMF). SVSF is the simplest and most conservative approach, which assigns the same voltage and frequency to all cores and structures in a die. While the strategy is sufficient for an STV processor, considering the low variability, it would result in over-pessimistic assumptions for the maximum frequency considering the high variability in NTV. On the other hand, the Multi-Voltage or Multi-Frequency approaches are more flexible, but require the use of special components to interface between different domains, and the use of special techniques for determining the optimal voltage and frequency assignments for each domain.

In the same work, an MVSF strategy for sustaining performance from STV levels while computing at NTV (with higher number of cores, while still reducing the energy consumption) is used for this assignment: first, the lowest-required frequency for sustaining STV performance is computed, and then the voltage domains for achieving that frequency are allocated. The result is a frequency-homogeneous multi-core. A MVMF extention to this scheme, which results in a heterogeneous multi-core and can be particular efficient in applications with workload disbalance among threads is also presented (SILVANO et al., 2014).

Kaul et al. propose a simpler approach, suitable for many-core sytems (KAUL et al., 2012). The authors argue for a simple SVMF strategy where the nearest-possible operation frqeuency is assigned to each core. Then, according to the Law of Large Numbers, the overall throughput of the system should not be affected.

EnergySmart takes a similar, SVMF approach, arguing for the inneficiency of MV-based schemes due to the high amount of area and low energy efficiency of on-chip voltage regulators (KARPUZCU et al., 2013). The proposed strategy is also coupled to a scheduling algorithm assigns tasks to the performance-heterogeneous cores to maximize performance per watt.

Kiamehr et al. propose a temperature-aware voltage scaling approach, showing that ambient temperature information (which is typically very close to circuit temperature, when operating at NTV) can be leveraged to better tune the optimal V_{dd} level and reduce energy consumption compared to temperature-unaware schemes (KIAMEHR et al., 2017).

2.3 Approximate Computing

Approximate computing is a design paradigm that enables trade-offs in the quality of an application's results for improvements in other metrics such as performance or energy consumption. It is based on the observation that many modern application domains, such as computer vision, gaming, machine learning, data mining, dynamic simulations and data mining present a *forgiving nature* (i.e., they can tolerate controlled deviations in the output without compromising the functionality) due to at least one of the following characteristics (CHAKRADHAR; RAGHUNATHAN, 2010):

- **Noisy inputs**: Real-world data is, by nature, noisy due to measurement errors. Any

application processing noisy data has already introduced some degree of error in the output and, therefore, can tolerate approximation in its computations.

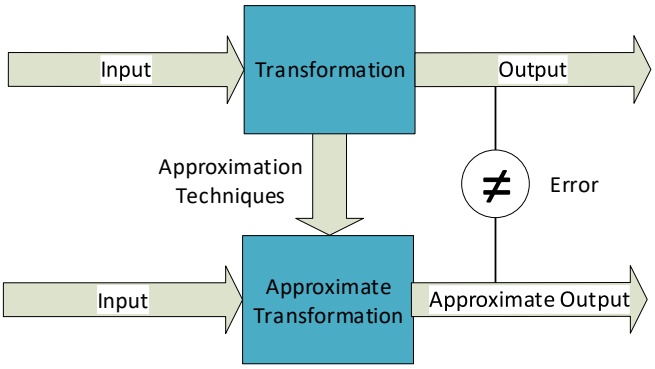
- **Large, redundant data-sets**: Some applications process large amounts of data with significant similarity between input elements, for instance image/video processing and machine-learning applications. In such scenarios, approximation can be used to remove this redundancy with only small impact in the quality.
- **No perfect result**: Heuristics, which are a form of approximating a computation, are often used when solving complex problems for which an exact solution is either unavailable or computationally intractable. Other approximation techniques may also be applied in such scenarios.
- **Limited perceptual ability of users**: In many cases, a *good-enough* application result is sufficient because the user would not be able to notice any difference between that and the *precise* result. Many lossy-compression algorithms are based on this idea ((ESR), 2011).

The observation above raises an opportunity to exploit these common characteristics as a design paradigm to enable performance or energy improvements. Fig. 2.15 shows an overview of the Approximate Computing paradigm. Given the abstract notion of an algorithm, which is a transformation from inputs into outputs, implemented either as software or hardware, approximate computing consists of finding an approximate, more efficient version of the same implementation by using approximation techniques. The approximate transformation produces an approximate output which can be compared with the precise one, and an error measurement can be established. An important assumption in approximate computing is that the output of the approximate transformation must contain only *small deviations* from the precise one. In this sense, approximation techniques that can cause the system to abruptly crash or change the *structure of the output* (i.e. change the number of output elements, or the output encoding) are still considered unacceptable.

Considering the above discussion, this work uses a terminology similar to the one defined in a previous work to classify when an approximation technique produces acceptable results or not (REHMAN; SHAFIQUE; HENKEL, 2012). An application output falls into one of the three categories:

- **precise output** : the output produced by the precise program.
- **approximate output**: an output that is incorrect, but acceptable as an approximation of the correct one. The *structure* of the output is preserved (e.g. if the precise

Figure 2.15: Overview of the approximate computing paradigm.



Source: the author.

output is an image, the approximate output must also be an image of the same size) and an error metric can be computed.

- **invalid output** : this is an output that is incorrect and unacceptable as an approximation. This is the case when the application terminates abruptly, crashes, produces an output without the correct *structure* or does not produce any output at all.

For understanding approximate computing, this section approaches the subject in a top-down fashion by reviewing recent works. Section 2.3.1 discusses how to assess numerically the quality of an approximate output, and acceptable thresholds are determined. Section 2.3.2 discusses how to determine computations that are tolerant to approximation and techniques for annotating these. Finaly Section 2.3.3 describes existing stategies to implement approximate computations across the system stack.

2.3.1 Assessing Application Quality

As was shown in Fig. 2.15, from a high-level of abstraction, each computer program can be described as a transformation of inputs into outputs which are encoded as bits. In order to assess quality, however, these bits must be transformed into meaningful information, since each position in the bitstream may have distinct impact in the application quality. What the output represents, and, therefore, how quality must be assesed, depends on the nature of the application under consideration and the type of output that it produces (e.g. numbers, images, text, . . .).

Fig. 2.16 presents an example of quality assessment for an image-processing application. In this case, the Root-Mean-Squared Error (RMSE) of the pixel difference

Figure 2.16: Distinct quality levels for an image when using RMSE as metric.

Source: (SAMADI et al., 2013).

between the precise and approximate outputs, normalized to the range $[0, 1]$, is used as a quality metric. This metric can be evaluated using Eq. 2.5, where N is the number of pixels in the image and X_i, $\widetilde{X}_i$ are the pixel values in the precise and approximate images, respectively. Considering an 8-bit greyscale image, pixels may take a value in the range $[0, 255]$. Given a particular N, a minimum and maximum value value for the RMSE exists, so it can be normalized to the range $[0, 1]$.

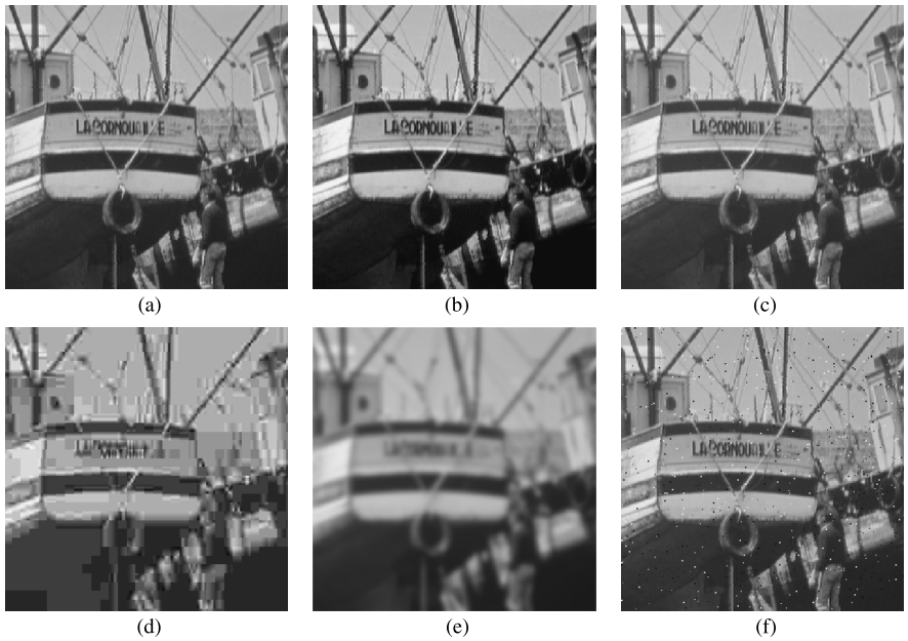
$$Q_{RMSE} = \sqrt{\frac{\sum_{i=0}^{N} (X_i - \widetilde{X}_i)^2}{N}} \tag{2.5}$$

The figure also highlights an important observation: distinct quality levels may be adequate under distinct circumstances (in this case, the resolution of the image). Table 2.1, extracted from a recent survey (MITTAL, 2016), shows applications and error metrics commonly used in approximate computing works. For applications producing numerical outputs, such FFT, Newton-Raphson and FIR filter, the average relative difference between precise and approximate outputs is typically used. For machine-learning applications, quality is assessed from the classification accuracy [3]. For applications producing images, pixel difference, Peak Signal-to-Noise Ratio (PSNR) or Structural Similarity Index (SSIM) (WANG et al., 2004) may be used.

The same table also highlights an important challenge when dealing with approximation. Very often, multiple error metrics with distinct sensitivities to the form of the

[3] It should be noted, however, that these applications present *no perfect result* as they are based on heuristics. Therefore, rather than assessing quality by comparing with the perfect result, approximate results are typically compared with the best one from an heuristic run.

Figure 2.17: Images affected by distinct forms of error, with the same RMSE value.



Source: (WANG et al., 2004).

error may be available for the same application. Fig. 2.17 provides an example. Six images are presented: the first one is the original one; the others have been distorted with various filters (in order: contrast-stretch, mean-shift, lossy JPEG compression, blur, and salt-pepper impulsive noise). All of the distorted images present the same RMSE value with respect to the original one, yet some may be acceptable in some contexts while others may not. For comparison with the SSIM metric, image (c) presents the highest mean SSIM value (0.99) while (d) presents the lowest one (0.69).

Given the complexity of assessing quality, all works in approximate computing assume that a function to measure the quality of the output has been defined by the application developer (the one who knows how the application will be used). Moreover, the developer must know the quality that is acceptable for the application, an issue which is often non-trivial.

So far, few works have investigated quality thresholds for modern applications, or how far approximation can be applied without compromising an application's functional requirements. AxGames represents one such attempt (PARK et al., 2016). In that work, a methodology based on crowdsourcing was used for understanding how an application's output quality, as expressed by a function written by the application developer, translates into a rate of users satisfied with that quality. This approach enables developers to systematically assess the effects of approximation from an end-user's perspective, rather than from an arbitrary metric, as depicted in Fig. 2.18.

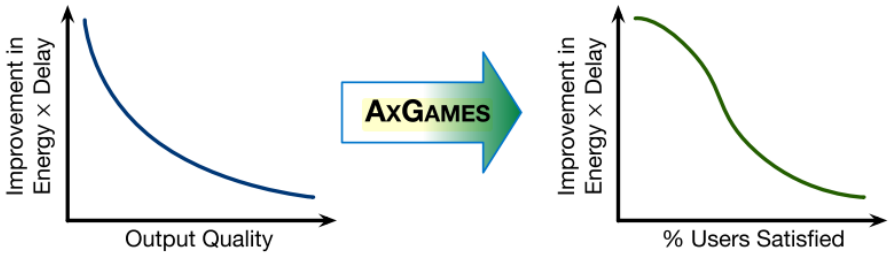
Table 2.1: Example of error metrics used for distinct applications.

Error Metrics(s)	Applications
Correct and incorrect decisions	Image binarization, *jmeint* (triangle intersection), *ZXing* (visual barcode recognition), Julia set fractals.
Classification or clustering accuracy	*Ferret* (compares an image for similarity against database of images), *streamcluster*, K-Nearest Neighbours, K-Means Clustering, LGVQ (Generalized Learning Vector Quantization), Convolutional neural networks, multi-layer perceptron networks, Support Vector Machine (SVM).
Energy conservation across scenes	Physics-based simulations (e.g. collision detection, constraint solving.
PSNR and SSIM	H.264, x264, MPEG, JPEG, rayshade, image resizer, image smoothing, OpenGL games.
Pixel difference	*bodytrack*, *eon*, *raytracer*, particle filter, volume rendering, Gaussian smoothing, mean filter, dynamic range compression, edge detection, raster image manipulation.
Ranking accuracy	Bing search, supervised semantic indexing document search.
Ratio of error of initial and final guess	(Differential) Equation solvers, Image Compression.
Relative difference or error from standard output	Fluidanimate, blackscholes, swaptions (PARSEC), Barnes, water, Cholesky, LU (Splash2), vpr, parser (SPEC2000), Monte Carlo, sparse matrix multiplication, Jacobi, discrete Fourier transform, MapReduce programs (e.g., page rank, page length, project popularity, and so forth), forward/inverse kinematics for 2-joint arm, Newton-Raphson method for finding roots of a cubic polynomial, n-body simulation, adder, FIR filter, conjugate gradient. H.264

Source: (MITTAL, 2016).

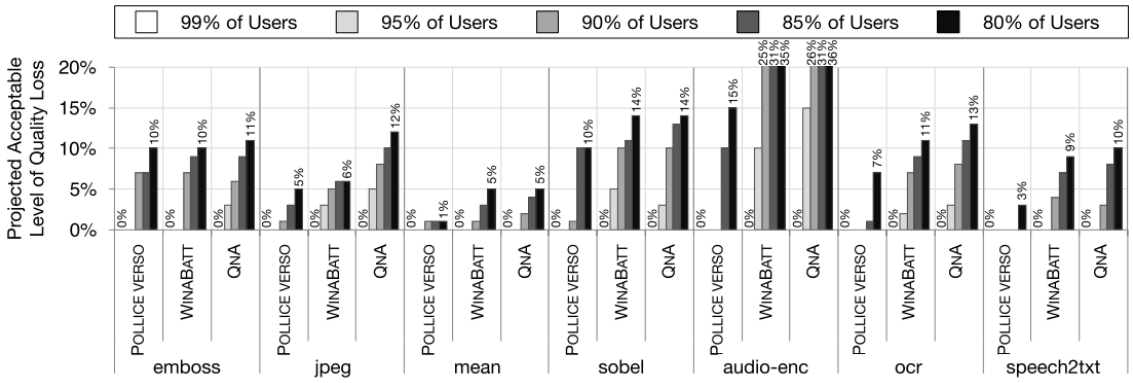
A study was conducted with 700 participants, in which seven applications were considered: *emboss*, *jpeg*, *mean*, *sobel* (image-processing filters); *audio-enc* (audio encoder); and *ocr*, *speech2txt* (optical character recognition and text-to-speech conversion, respectively - applications producing text as a result). Participants in the study were presented three games, named *Pollice Verso*, *WinABatt*, and *QnA*. In all games, players are given an initial amount of virtual money and must make a decision each turn that may increase or decrease their money. In the first game, players are presented each turn an approximate application result, along with the precise counterpart, and must bet a certain amount of money in asserting that the approximate output is good enough or unacceptable. In the second game, players are presented a low-quality approximate output and must spend money to increase the quality and make the output acceptable. In both these games, players are better rewarded when their answers are close to average of the pop-

Figure 2.18: Correlating output quality value with users' satisfaction.



Source: (PARK et al., 2016).

Figure 2.19: Results from the AxGames study.



Source: (PARK et al., 2016).

ulation (other players). In the third game, players are again presented a low-quality approximate output which can be improved by spending money, but must answer questions about features found in the result (for instance, if the output is an image, a player might be presented a multiple-choice question where one must identify whether the image contains a sports car, a truck or other structures).

Statistical inference was then used with the responses to produce the results shown in Fig. 2.19. Key conclusions to this study were as follows. First, distinct applications come with distinct quality requirements, and a single target quality threshold cannot be set for all applications considering that each application takes a distinct quality metric. That may be true even for applications with the same type of output, as can be seen when comparing the results of applying *mean* filter and *emboss*, *jpeg* and *sobel*. Second, users show higher tolerance to approximation when they consider cost or context (which may translate to energy consumption), as shown by the higher tolerance in the cost-aware games *WinABatt* and *QnA*.

2.3.2 Determining approximable computations

As was mentioned earlier, one key assumption of approximate computing is that the outputs produced by the approximate version of the program will contain only small deviations from the correct one. Situations where the program crashes or produces no output are, in most cases, still considered unacceptable. Therefore, for most of the approximation techniques described later in this section, it is generally true that not all of the computations will be amenable to approximate execution. So, in order to employ approximate computing, first the computations that are amenable to approximate execution must be identified.

Figure 2.20: Implementation of 2x2 matrix multiplication at distinct abstraction levels.

(a) Mathematical level.

$$C = A \times B$$

(b) High-Level language.

```
void mm(int A[2][2],
        int B[2][2],
        int C[2][2]) {

  for (int i = 0; i < 2; i++) {
    for (int j = 0; j < 2; j++) {
      C[i][j] = 0;
      for (int k = 0; k < 2; k++) {
        C[i][j] += A[i][k] * B[k][j];
      }
    }
  }

}
```

(c) Low-level language.

```
.LFB0:
  pushq %rbp
  xorl  %ecx, %ecx
  movq  %rsp, %rbp
.L2:
  leaq  (%rdx,%rcx), %r10
  xorl  %eax, %eax
.L3:
  movl  $0, (%r10,%rax)
  movl  (%rdi,%rcx), %r8d
  movl  (%rsi,%rax), %r9d
  imull %r8d, %r9d
  movl  %r9d, (%r10,%rax)
  movl  4(%rdi,%rcx), %r8d
  imull 8(%rsi,%rax), %r8d
  addl  %r9d, %r8d
  movl  %r8d, (%r10,%rax)
  addq  $4, %rax
  cmpq  $16, %rax
  jne   .L3
  addq  $8, %rcx
  cmpq  $32, %rcx
  jne   .L2
  popq  %rbp
  ret
```

Source: the author.

Fig. 2.20 illustrates this discussion, by presenting a very simple application (matrix multiplication) from distinct levels of abstraction: high level (mathematical model), mid-level (source code in C) and low level (x86 assembly code). An approximate output

for this computation would be a matrix $\widetilde{C}$ such that $\widetilde{C} \approx A \times B$, so would be tempting to think that simply approximating the $\times$ operation would solve the issue. However, implementing this computation as an algorithm introduces additional variables and computations besides A, B, and C. At a mid-level of abstraction, in Fig. 2.20b, it's noticeable the introduction of variables i, j, and k to control the loop and array indexes. From a low-level of abstraction, in Fig. 2.20b, this problem is highlighted. Out of 22 assembly instructions, only 3 (multiplications and sums at lines 12, 15, 16) are directly related to the application output. The remaining ones are overheads introduced to evaluate array indexes, calculate memory addresses, and control loop/jump addresses. These operations, when naively approximated, may cause the program to run forever, switch control flow to a random region in the address space or similarly access an unauthorized memory address, causing the program to crash. It is, therefore, critical to identify the highest number of operations that can be *safely* approximated, in order to achieve significant efficiency improvements without compromising application functionality.

Considering the above discussion, we use a definition similar to the one used by Rehman et al. to classify operations (REHMAN; SHAFIQUE; HENKEL, 2012):

- **non-crucial operations**: operations that can produce an approximate program output, but will never lead to an unacceptable one; these are operations whose results flow directly to the program output.
- **crucial operations** : operations that may lead to a software program failure and an unacceptable output; these are, for instance, operations involved in memory address calculation (load, stores, branches) or loop control, since they may cause segmentation fault or the program to run forever.

So far, three approaches for identifying approximate computations have been developed: data type annotation (Fig. 2.21a), code region annotation (Fig. 2.21b), and automatic detection (by the compiler or runtime). These approaches are described over the next sections.

Manual detection through data annotation. One way to specify what computations can be carried approximately is to inform the compiler the variables that may be imprecise by annotating them in the code. Then, approximate optimizations can be carried out in every computation affecting only those approximate variables.

Work by Sampson et al. propose extending programming languages with constructs that would allow programmers to specify approximate data types and operations

Figure 2.21: Variable and code region annotation for approximate computing.

(a) Variable annotation.

```
void mm(approx int A[2][2],
        approx int B[2][2],
        approx int C[2][2]) {

  for (int i = 0; i < 2; i++) {
    for (int j = 0; j < 2; j++) {
      C[i][j] = 0;
      for (int k = 0; k < 2; k++) {
        C[i][j] += A[i][k] * B[k][j];
      }
    }
  }
}
```

(b) Code region annotation.

```
void mm(int A[2][2], int B[2][2], int C[2][2]) {

#pragma approximate_begin(in=A[2][2],B[2][2])
  for (int i = 0; i < SIZE; i++) {
    for (int j = 0; j < SIZE; j++) {
      C[i][j] = 0;
      for (int k = 0; k < SIZE; k++) {
        C[i][j] += A[i][k] * B[k][j];
      }
    }
  }
#pragma approximate_end(out=C[2][2])

}
```

Source: the author.

Table 2.2: Language constructs for approximation in EnerJ.

Construct	Purpose
@*Approx*, @*Precise*, @*Top*	Type annotations: qualify a type as approximate, precise, or whether it should inherit the qualifier from the superclass.
endorse(e)	Cast: transform an approximate value to precise.
@*Approximable*	Class annotation: allows a class to have both precise and approximate instances.
@*Context*	Type annotation (allowed inside @*Approximable* classes): the precision of the type depends on the precision of the enclosing object.
_APPROX (method suffix)	Method naming convention: invoke this implementation of the method when the receiver has approximate type.

Source: (SAMPSON et al., 2011).

(SAMPSON et al., 2011). As a case study, a set of language extensions to Java, named EnerJ, is presented. The constructs defined in EnerJ are presented in Table 2.2 and described next.

In EnerJ, every variable includes, along with the data type, a precision qualifier, as defined by the @*Approx* and @*Precise* annotations. No annotation defaults to @*Precise* to maintain compatibility with legacy code. Computations whose results flow into approximate variables are subject to approximate software optimizations, such as the ones which will be presented in subsection 2.3.3. Precise and approximate variables have isolation guaranteed by the type system: while precise variables may flow into approximate ones, an approximate value assigned to a precise data type generates a semantic error. The programmer may circumvent this restriction by using a special type of cast, named *endorsement*. This design strategy certifies that the programmer is aware of how approximation may be impacting precise variables. Classes may have precise and imprecise instances when annotated with the @*Approximable* construct. In these instances, variables qualified with the @*Context* annotation will have same precision qualifier as the enclosing object. Finally, class methods can be overloaded to implement precise and approximate versions by using the _APPROX suffix; which implementation is called depends on the precison qualifier of the variable the result is assigned to.

Fig. 2.22 shows an usage example of the EnerJ framework. A class for storing floating point numbers is defined, with a method that computes the mean of the set. The precision qualifier of the floats stored in the *nums* array depends on the precision of the *FloatSet* object. Two *mean* methods are defined; the first one is called when the

Figure 2.22: Example of using the EnerJ framework.

```
@Approximable class FloatSet {

  @Context float[] nums;

  @Precise float mean() {
    float total = 0.0f;
    for (int i = 0; i < nums.length; i++) {
      total += nums[i];
    }
    return total / nums.length;
  }

  @Approx float mean_APPROX() {
    float total = 0.0f;
    // Perforated loop
    for (int i = 0; i < nums.length; i += 2) {
      total += nums[i];
    }
    return 2 * total / nums.length;
  }

}
```

Source: (SAMPSON et al., 2011).

instance of *FloatSet* is precise and the second one when it is approximate. The approximate implementation computes the result more efficiently by applying loop perforation, a software-level approximation technique which will be described in subsection 2.3.3.

Data variable annotation is commonly used with techniques based on precision scaling (changes the precision of specified variables) and approximate data storage, which will be described in subsection 2.3.3.

Manual detection through code region annotation. Another way to specify approximate computations is to specify regions of code that are amenable to approximate execution. Then, every computation carried out in that region is treated as approximate, as well as its results. This approach requires the programmer to identify *pure code regions*, where the following requirements are met:

- the region has a well-defined number of inputs and outputs, and this number is known at compile time;
- the region is deterministic; i.e. whenever it executes with the same input arguments, the same output is produced;
- the region processes no global variables.

For the sake of illustration, consider the code region annotated in Fig. 2.21b.

It contains eight inputs (four values in the matrix A, four values in the matrix B) and four outputs (the matrix C). This region contains internal variables, i, j, k, but these do not affect the program state anywhere outside of the region. Moreover, the region is deterministic, i.e. if it is executed multiple times with the same inputs A and B, it will produce the same result C. This is true because there are no state-holding elements in the region (values which are preserved across invocations, such as static or global variables) and no stochastic computations (such as a call to *rand()*).

This is the approach typically used by works targeting code transformation, such as neural acceleration and approximate memoization, which will be described in subsection 2.3.3.

Automatic Detection. Considering the difficulties of manually identifying approximate code, attempts have been made at automating this procedure.

Roy et al. propose a statistical method to determine non-critical application variables (i.e. variables into which approximate computations may can safely flow) by considering their contribution to the application output (ROY et al., 2014). The method is based on profiling all variables at run time, perturbing their values by a small amount and monitoring the impact in application quality by using statistical tests. The approach achieves 87% of the accuracy achieved by manual datatype annotation, such as the one achieved by EnerJ (SAMPSON et al., 2011).

Later work by Roy et al. improves the previous strategy by assigning a non-binary importance value to each variable, rather than classifying them into approximable or non-approximable (ROY; WANG; WONG, 2015). This expands the range of approximation-tolerant computation, but increases the risk of producing invalid outputs.

2.3.3 Strategies for approximation

This section presents the main techniques used for implementing approximate computing. These techniques can be classified into three levels (MITTAL, 2016; SHAFIQUE et al., 2016; XU; MYTKOWICZ; KIM, 2016):

- *software level*: techniques which transform the application source code into an approximate implementation and require no changes in existing hardware;
- *architectural level*: techniques which transform the source code into an approximate representation and use hardware support to accelerate the execution;

- *hardware level*: techniques which modify the circuit implementation (usually dedicated hardware) and require no changes in the application.

Software-level Techniques. *Loop perforation* is an approximate software optimization that modifies loops so they execute only a subset of their iterations, thereby reducing execution time and energy consumption. For instance, a loop of the form

```
for (i = 0; i < b; i += 1) { ... }
```

may be transformed into

```
for (i = 0; i < b; i += n) { ... }
```

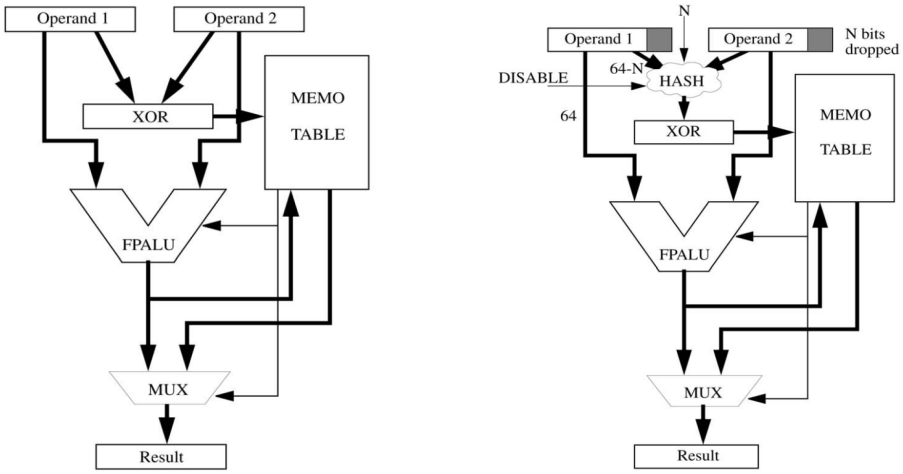
where n is taken based on the desired *perforation rate*, $r = 1 - \frac{1}{n}$ (the expected percentage of loop iterations to skip). Since the technique targets a very specific type of software construct (loops with a number of iterations known at compile time), it can be easily automated, as recent works have done.

In (SIDIROGLOU-DOUSKOS et al., 2011), an offline algorithm for automatically identifying the best opportunities for loop perforation in a program is presented. The algorithm first runs a *criticality testing* phase where it identifies candidate loops for perforation. These are the loops that account for a significant fraction of the executed instructions and, when perforated using a target r, are guaranteed not to produce unacceptable outputs, infinite loops or decrease the performance. After that, the algorithm proceeds to a space-exploration phase where all combinations of loops and perforations are evaluated and accuracy/performance tradeoffs are recorded. Evaluating this approach with benchmarks in the PARSEC suite demonstrated speedups of up to 5x in the applications under consideration.

In (SAMADI et al., 2014), loop perforation is used to approximate *reduction* code patterns running in CPU and GPU. Additionally, a variant of loop perforation named *subsetting*, which consists of intentionally ignoring some inputs in a computational kernel, is evaluated as well. These techniques can yield an average speedup of 3.5x for GPU code and 4.3x for CPU code.

Architecture-level Techniques. *Memoization* is the concept of storing the results of expensive computations so that they can be replaced by a table lookup when it should execute with the same inputs again. The term was coined by Donald Michie in a classic paper (MICHIE, 1968) and an extensive body of work has been developed in using

Figure 2.25: Memoization of floating-point instructions.
(a) Classic memoization. (b) Fuzzy memoization.



Source: (ALVAREZ; CORBAL; VALERO, 2005).

memoization for improving an application's execution (SODANI; SOHI, 1997; HUANG; LILJA, 1999; GONZÁLEZ et al., 1999; SURESH et al., 2015).

In this subsection, *approximate memoization*, an extension to the classic memoization scheme in which similar computations are allowed to share a same entry in the lookup table (CHAUDHURI et al., 2011), is discussed. This is typically achieved by dropping the Least-Significant Bits (LSBs) in the input and using only the remaining ones to index in the table. Therefore, the chances of a table hit are increased, boosting performance.

Alvarez et al. propose *fuzzy memoization* of Floating-Point (FP) operations at an instruction granularity (see Fig. 2.25) (ALVAREZ; CORBAL; VALERO, 2005). A hardware memo table is used to save results of FP multiplication and division instructions (simpler FP operations are shown not worth of memoizing because of their low latency). Implementing this scheme in an in-order 2-issue superscalar processor using a 24KB table improves Energy-Delay Product (EDP) on average by 20% in 4 benchmarks from the MediaBench suite.

Keramidas et al. take a similar approach by proposing a *clumsy value cache* for GPUs (KERAMIDAS; KOKKALA; STAMOULIS, 2015). The cache is a memoization table that can store the results of selected GPU instructions or sequences of instructions (blocks). Key conclusions of the work are as follows:

- *Value reuses are not equal between instructions*. Distinct instructions present distinct reuse rates, and these rates change differently when approximation is applied.
- *Value reuse potential in code segments does not exist*. Moving from instruction-level memoization to block-level memoization, although it increases the potential

benefits of reusing memoized values, it greatly decreases the hit rates due to the increased number of inputs.

- *Not all instructions in fragment shaders must be equally precise*. Distinct instructions present distinct quality thresholds and contribute differently to the final application quality.

Paraprox uses approximate memoization to accelerate two processing patterns commonly found in data-parallel applications: map and scatter/gather (SAMADI et al., 2014). A fixed-size table, generated at compile time, is used. Considering long input arguments, the authors exploit the fact that not all input arguments need to be equally precise, and use an offline algorithm to detect the optimum number of bits for each of the input arguments. An average speedup of 2.6x for CPU and 2.2x for GPU code is achieved over four applications with less than 10% quality degradation.

Work by Sato et al. presents an *approximate computing Stack based on computation reuse* (SATO et al., 2015). The programmer annotates using pragma directives the regions that are tolerant to approximation, specifying the input variables that can be approximated and a mask (an adapted code annotation scheme, similar to how it was described in the region annotation scheme). A compiler generates special instructions to signal the processor the inputs that can be approximated and the approximation mask. At runtime, a hardware mechanism updates a reuse table as the function executes and also checks this table for reuse before the function is executed. The approach was evaluated using a single image-processing benchmark from the MediaBench suite against a very simple SPARC processor, demonstrating speedup of 1.25x and increased reuse rates compared to non-approximate reuse.

Memoization is also one of the techniques supported in the iACT (MISHRA; BARIK; PAUL, 2014) and ACCEPT (SAMPSON et al., 2015) frameworks, which will be described later on.

Concerning *approximate data storage*, San Miguel et al. propose Doppelgänger, a cache for approximate computing (MIGUEL et al., 2015). The work uses the insight that in many scenarios, such as when processing an image with many similar blocks, there will be a lot of approximate redundancy in the data cache (similar blocks stored) and exploits this fact to compress the cache. Fig. 2.26 presents an example.

A continuation of this work is the Bunker Cache, which takes into account spatial approximation as well as value approximation (MIGUEL et al., 2016). It is based on the insight that elements that are approximately similar in value exhibit spatial regularity in

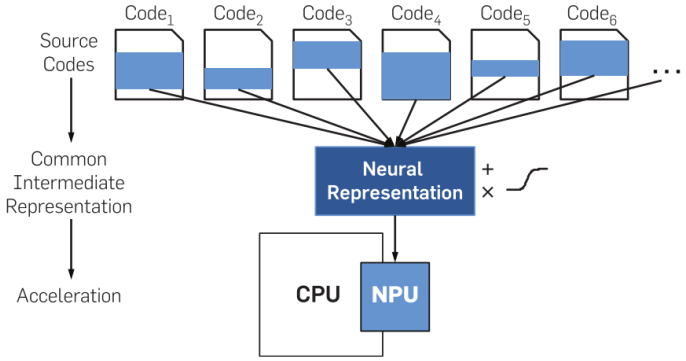
Figure 2.26: Data redundancy in image processing applications.
(a) Selected data blocks.

(b) Data block values.

	R	G	B	R	G	B
1	92	131	183	91	132	186
2	90	131	185	93	133	184
3	35	31	29	43	38	37

Source: (MIGUEL et al., 2015) .

Figure 2.27: Overview of the *parrot transformation*.



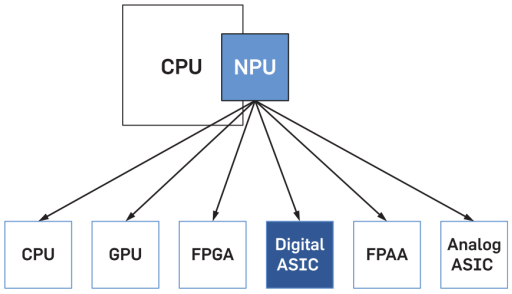
Source: (ESMAEILZADEH et al., 2014).

memory.

Of all techniques working at an architectural level, *neural acceleration* is the most promising approach. Esmaeilzadeh et al. propose the *Parrot Transformation*, a learning-based approach to accelerate approximate programs (ESMAEILZADEH et al., 2014). A framework was designed for transforming program regions that are tolerant to approximation into an Artificial Neural Network (ANN) representation which can then be either be executed in the processor or accelerated using a dedicated hardware, thereby presenting the ability of converting distinct code patterns into a common representation that can be accelerated. This idea is depicted in Fig. 2.27.

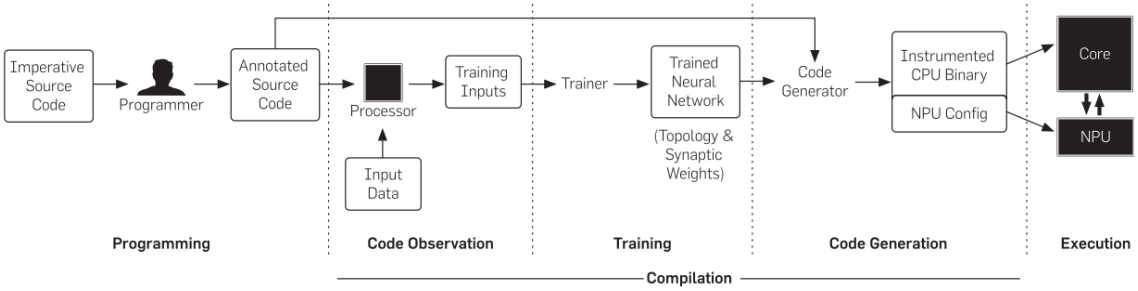
The design flow is show in Fig. 2.29. The programmer is responsible for selecting, during application development, approximate regions that can be turned into ANNs and using special code annotation to mark them for the framework using the technique presented in subsection 2.3.2. At compile time, these regions are profiled for common $< \text{input}, \text{output} >$ mappings and a neural network that mimics that behaviour is trained. The original code is replaced by a call to a programmable Neural Processing Unit (NPU)

Figure 2.28: Design space for implementing an NPU.



Source: (ESMAEILZADEH et al., 2014).

Figure 2.29: The *parrot transformation* framework in detail.



Source: (ESMAEILZADEH et al., 2014).

that executes the trained ANN more efficiently than the original code in the CPU.

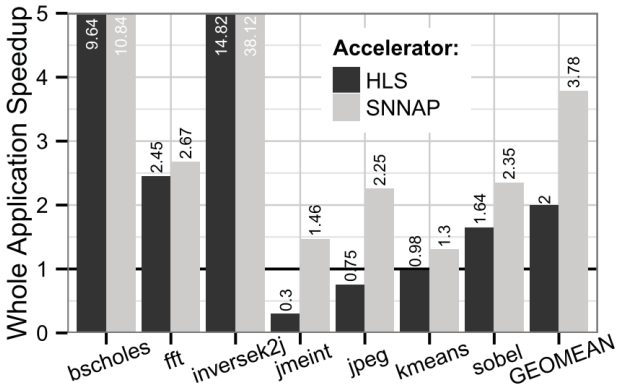
Different works have investigated implementations of NPU for executing the ANN that results from the Parrot Transformation. Some of these design choices are depicted in Fig. 2.28. In all of these works, the AxBench (YAZDANBAKHSH et al., 2016) suite was used for evaluation.

The original work investigated CPU and ASIC implementations (ESMAEILZADEH et al., 2014). CPU presents a geomean slowdown of 19.9x, while ASIC execution presents geomean speedups of 2.3x and energy savings of 3.0x. No results on the area overheads of the ASIC implementation are presented in the work.

Work by Yazdanbakhsh et al. presents the design of a neural accelerator named *NGPU* that can be gracefully integrated into GPU cores and used to accelerate applications that otherwise would run on the GPU (YAZDANBAKHSH et al., 2015) . Compared to the baseline GPU architecture, cycle-accurate simulation results for NGPU show a 2.4x average speedup and a 2.8x average energy reduction within 10% quality loss margin. These benefits are achieved by introducing less than 1% area overhead.

Work by Moreau et al. demonstrates implementation of an NPU named Systolic Neural Network Accelerator in Programmable Logic (SNNAP) for Programmable System-On-Chips (PSoCs), which are devices integrating a hard processor core with programmable logic on the same die (MOREAU et al., 2015). SNNAP can be programmed

Figure 2.30: Speedup achieved by SNNAP, compared to HLS.



Source: (MOREAU et al., 2015).

by using the same strategies described earlier, as well as by using dedicated low-level primitives which are available to the advanced user. The proposal was evaluated by using a Zynq ZC702 evaluation platform, which features a mobile-grade dual-core ARM Cortex-A9 and a Xilinx Artix-7 FPGA onto the same 28nm die, and applications from the AxBench suite. When comparing the use of SNAPP for each benchmark's target region against a software baseline, speedup ranges from 1.3x to 38.12x, with a geomean of 3.78x. Similarly, a geomean energy saving of 2.77x is achieved when considering the Zynq+DRAM and 1.82x when considering the core logic alone. Detailed results are presented in Fig. 2.30.

Hardware-level Techniques. Voltage Over-scaling (VOS) (MOHAPATRA; KARAKONSTANTIS; ROY, 2009; MOHAPATRA et al., 2011) scales a circuit's input voltage beyond (*over*) the level considered safe, thereby introducing possible timing problems but greatly reducing the power consumption (refer to Section 2.2 for a detailed explanation on the effects of V_{dd} in power and energy consumption). It uses statistical techniques to identify at compile time the *crucial* and *non-crucial* operations and changes the architecture at runtime by means of VOS to tradeoff between energy and quality. The evaluation was done based on a Motion Estimation (ME) processor implemented in 90nm CMOS. Simulation results show average power savings of 33% and a maximum quality loss of 1 dB.

Work by Mohapatra et al. presents a design methodology that makes circuits more scalable for VOS by applying two techniques: Dynamic Segmentation and Error Compensation and Delay Budgeting (MOHAPATRA et al., 2011). The methodology was evaluated by designing dedicated circuits for the kernels of three applications (ME, SVM classification and K-Means Clustering) and comparing against the traditional methodology in scenarios of iso-quality and iso-energy. At iso-quality, energy savings improve by

17% in ME, 52% 30% for SVM and 30% for K-Means Clustering. At iso-energy, quality improves by 2dB for ME, 52% for SVM and 6% for K-Means Clustering.

IMPrecise adders for low-power Approximate CompuTing (IMPACT) is an implementation of approximate Full Adders (FAs) that are used to build complex arithmetic units (GUPTA et al., 2011). The authors evaluated the approach in an ASIC flow with image and video compression kernels using the approximate FAs in the LSBs of multi-bit adders and compared with precise adders. In image compression, approximating the 7-9 LSBs (out of 20) for the Discrete Cosine Transform (DCT) and Inverse Discrete Cosine Transform (IDCT) computations allows for 20-35% area savings and shortens the circuit's critical path, allowing for a reduction of 12-22% in the supply voltage and 40-60% in power. The quality, in these cases, measured as PSNR, ranges from 15 to 30 dB. In video compression, approximating the 1-4 LSB (out of 16) in Sum of Absolute Differences (SAD) computation which is part of ME allows for up to 40% power savings with quality above 30 dB. The approach achieves significantly better quality ratios than simply truncating the LSBs, but less power savings.

While IMPACT requires defining at design time the desired quality degradation, other works propose approximate adders in which the quality can be controlled at runtime (KAHNG; KANG, 2012; YE et al., 2013; SHAFIQUE et al., 2015). While reconfigurability may introduce overheads that are absent in a static design, it introduces the ability to tune the quality to particular application phases, potentially allowing for additional power savings. Work by Shafique et al. (SHAFIQUE et al., 2016) provides a comparison between the runtime-configurable approximate adders implementation in the authors' own previous work (SHAFIQUE et al., 2015) and IMPACT (GUPTA et al., 2011).

Combining strategies for approximation. Zhang et al. propose a framework called ApproxANN for approximating neural computation in ANNs (ZHANG et al., 2015). A methodology for identifying critical and resilient neurons is presented, and then applied to a few selected applications to reduce the precision of neural computation by three means: skipping specific neural computations, discarding the LSBs and utilizing approximate Functional Units (FUs). The results show that for applications with varying ratios of energy consumption between computation and memory the overall consumption can be reduced by 34-51%. The claim is that not only does the computational part need to be approximated for improving energy efficiency, but also the memory accesses.

SAGE is a framework for implementing approximate computing kernels that run in a GPU (SAMADI et al., 2013) . The programmer writes a program in CUDA and

specifies a target output quality (TOQ). The framework proceeds in two phases, one offline and another online. In the first phase, the framework optimizes the kernels by using approximation techniques and derives multiple implementations with varying degrees of quality. In the online phase, the framework selects the approximate kernels that provide the TOQ by using a greedy algorithm and periodic calibration checks for accuracy. Three approximate optimizations are exploited to accelerate the CUDA kernels:

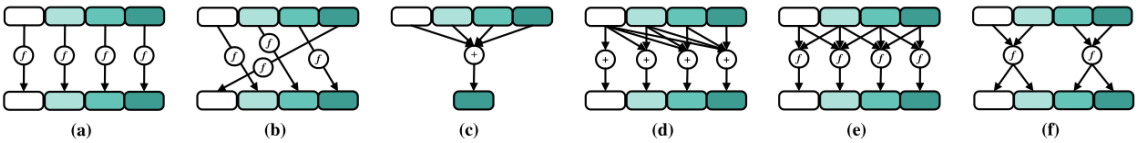
- selective discarding of atomic operations that cause frequent collisions and deteriorate performance as threads are sequentialized
- data packing, which reduces precision of input arrays in order to reduce the amount of memory operations
- thread fusion, in which similar threads are combined to produce a single result that is shared by both.

Samadi et al. presents a framework for optimizing computational patterns that execute in a GPU by applying approximation (SAMADI et al., 2014). Six commonly-found patterns in parallel programs are targeted, shown in Fig. 2.31: (a) map, (b) gather/scatter, (c) reduce, (d) stencil, (e) scan and (f) partition. For each of these, pattern-specific approximate optimizations are used. For (a) and (b), memoization is used, replacing expensive computation with memory accesses. For (c), subsetting is used - i.e. the reduction is evaluated by considering only part of the input. For (d) and (e), subsetting and replication is used - only a small part of the input is read from memory and then replicated to the remaining input, on the assumption that adjacent inputs are typically similar. For (f), only a subset is used for evaluation - the result is used to predict the results for the remaining of the input array.

Just like SAGE, the framework consists of an offline compiler, which detects the patterns and applies multiple optimizations - thereby generating multiple versions of each kernel - and an online monitor system that checks the accuracy of the results against the TOQ and dynamically selects the kernel that increases or reduces quality to meet the TOQ and improve performance.

Intel Approximate Computing Toolkit (iACT) (MISHRA; BARIK; PAUL, 2014) is a framework for understanding the impacts of approximation in applications. The framework consists of a modified LLVM compiler with support to #pragma directives which indicate regions that are tolerant to approximation and a PIN-based runtime that implements these approximations. The tool supports three sort of approximate computa-

Figure 2.31: Data-parallel patterns targeted by the Paraprox framework.



Source: (SAMADI et al., 2014).

tion: precision reduction in software variables, noisy ALU computations and approximate memoization.

ACCEPT (SAMPSON et al., 2015) is a framework for assisting the programmer in introducing approximation in one's code. It is implemented by modifying the LLVM compiler and supports C code. It analyses the software and identifies functions which are *pure* (see subsection 2.3.2), including the variables and function calls responsible for violating the purity criteria, and also marks loops that can be safely perforated.

2.4 Contributions to the State-of-the-Art

Contributions to the field of reconfigurable architectures. In all works where automatic code transformation is used to dynamically map instruction traces to a reconfigurable fabric, a potentially-significant fraction of the application must first execute in a GPP before being accelerated. Therefore, the form (in-order, OoO, heterogeneous) of the base processor may have a significant impact in the final performance, power and energy consumptions. Compared to previously proposed reconfigurable architectures, MuTARe is the only one that takes this into account and uses a CGRA that can be coupled to multiple arrangements of processor cores and supports DVFS to match the performance improvements provided by automatic CGRA acceleration with the performance target, lowering the frequency if slack is available to save additional power. Furthermore, MuTARe goes beyond traditional reconfigurable architectures by leveraging the regular structure of the CGRA to incorporates emerging techniques that improve the adaptability range: approximate computing, to further improve the power consumption in emerging error-tolerant workloads, and NTV computing, to further save power and make the architecture competitive for low-power domains.

Contributions to the field of NTV computing. The MuTARe Architecture proposed in this work provides a suitable structure for applying the concept of NTV computing, as it can effectively address the design challenges mentioned in Section 2.2.1. First,

MuTARe can exploit significantly more ILP than traditional OoO cores, which have the ILP exploitation constrained by the dynamic scheduling algorithm they employ. That additional ILP exploitation can partially compensate for the decreased operation frequency and enable NTV to be used in the context of single-threaded applications. Second, MuTARe's main execution unit is a combinatorial CGRA with state-holding elements located outside of the fabric. As such, they can be kept in a separate voltage island (STV), along with the caches, and operate at twice the speed from logic. The increased ILP exploitation from additional functional units in the CGRA causes memory to be the limiting resource. By operating the memory at twice the logic speed, balance is restored to the design. Finally, MuTARe's regular fabric allows for more easily dealing with process variability, for instance employing over-provisioned functional units, of whose the slowest are disabled post-manufacturing.

Contributions to the field of approximate computing. The MuTARe Architecture is a reconfigurable architecture that can leverage the benefits of approximate computing and provide additional performance improvements and power savings in emerging error-tolerant domains. Compared to existing approximate computing works, this is the first one where a reconfigurable accelerator is used for approximate computations, with the advantage that such a system can adapt to different approximate applications (unlike dedicated accelerators) and consumes a significant amount of power in the functional units instead of control (unlike GPPs), having sufficient margin for benefiting from approximate optimization.

3 MUTARE - BASE ARCHITECTURE

This chapter presents the base architectural framework of Multi-Target Adaptive Reconfigurable Architecture (MuTARe). MuTARe's goal is to provide a *generic* framework for deploying a reconfigurable architecture that can execute applications more efficiently than GPPs (either faster or with lower energy consumption) and transparently (without need of any changes in the software development process). MuTARe achieves this goal by transfering execution to a reconfigurable unit built around a CGRA, which, compared to GPP cores, allows for wider ILP exploitation, the acceleration of data-dependent operations (by eliminating intermediate register writes) and saving instruction schedules for reuse (rather than rescheduling every time the same sequence executes, as is the case in OoO cores). The CGRA, being a regular structure, can be easily customized according to the designs's area and power budgets, and can, therefore, also be applied to a wide range of domains. To achieve this acceleration goal transparently, MuTARe uses a dynamic, hardware-implemented Binary Translation (BT) algorithm to CGRA configurations that encode entire instruction traces on-the-fly. Finally, the benefits of enhanced ILP exploitation in the CGRA may lead to over-performance in particular classes of applications, where performance *beyond a certain target* is unnecessary. To that end, MuTARe leverages DVFS to balance the performance improvements from CGRA execution with improved power benefits, adjusting the frequency to the required performance levels.

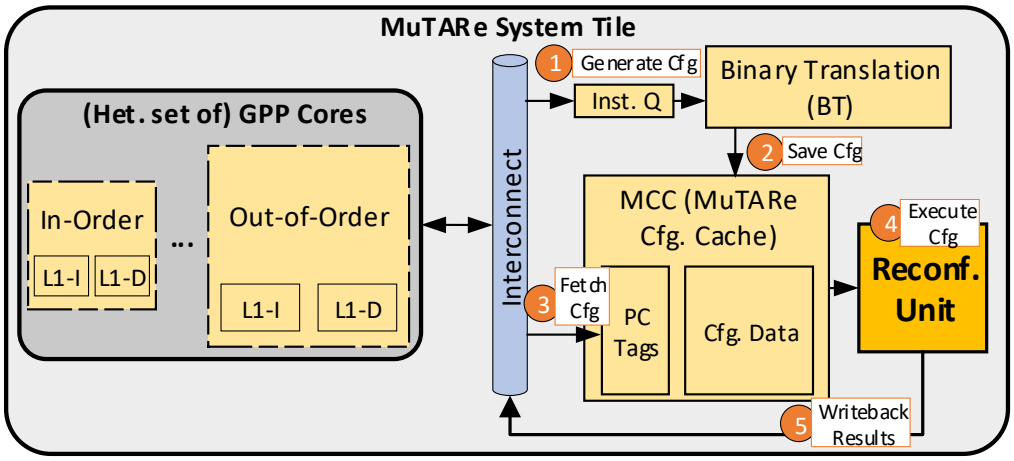
This section describes the base architecture of MuTARe. The two extensions of MuTARe for *NTV computing* and *approximate computing* require a few changes to the base architecture and are discussed afterward in Chapter 4.

3.1 Overview of MuTARe

An overview of the MuTARe hardware architecture is presented in Fig. 3.1. The key component in a MuTARe system is a *system tile*, which consists of a set of cores (a scalar core, or an OoO core, or an heterogeneous core arrangement such as ARM's big.LITTLE), a Reconfigurable Unit (RU) built around a CGRA, a configuration cache and a BT module. These *system tiles* can be replicated to build MuTARe-based CMPs.

The same Fig. 3.1 also provides an overview of the execution process in a MuTARe system tile. All instructions execute first in the GPP cores, as in a traditional GPP system. While they are executed, they are forwarded to an instruction queue that interfaces

Figure 3.1: Overview of MuTARe.



Source: the author.

between the GPP and the hardware-implemented BT module. The BT module translates incoming instruction traces into configurations for execution in the RU (see Step 1, in the figure) by allocating the operations into the CGRA fabric, and saving these configurations in the configuration cache for posterior acceleration (see Step 2). These configurations are indexed by the PC of the first instruction in the translated sequence, allowing the GPP fetch unit to lookup in the configuration cache (as well as its own instruction cache) whenever a new instruction sequence must be fetched for execution. When a match happens (the current PC is equal to that of a configuration), the configuration is fetched from the cache (see Step 3) and execution is offloaded to the RU (see Step 4). After execution in the RU completes, the results are written back to the GPP cores and commmitted *in program order* (see Step 5).

MuTARe is a fully customizable design where the GPPs, the CGRA sizes can be tuned for different domains. Since MuTARe automatically maps instruction sequences to the CGRA on-the-fly *after* they execute for the first time in the GPP, a fraction of the program instructions will inevitably execute in the base processor, so it is important to tune the base processor as well. The exact form of this arrangement will thus depend on the application domain: for high-performance computing, a large reconfigurable unit can be coupled to an OoO core; for mobile computing, a medium-sized reconfigurable unit can work with an heterogeneous arrangement such as ARM's *big.LITTLE* (ARM, 2013); for IoT devices, a simple single-issue core can be extended with a small CGRA.

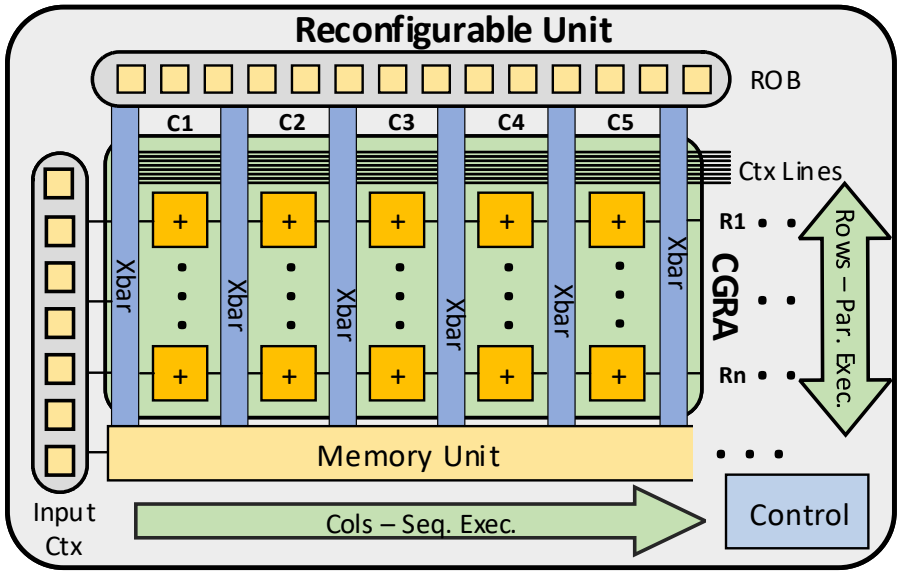
Next, each of the MuTARe components are described in greater detail.

3.2 MuTARe's Components

3.2.1 Reconfigurable Unit

An overview of MuTARe's RU is shown in Fig. 3.2. The main component is a combinatorial CGRA, organized as a matrix of integer FUs (represented as squares with + symbol, in the figure) without state-holding elements. In this matrix, columns represent sequential execution and rows represent parallel execution. FUs are connected via crossbars (Xbar) in a feed-forward, left-to-right fashion, i.e. FUs in a column propagate data to the ones to their right. Unused results in the adjacent column are made available to all subsequent columns by using context lines, wires that traverse the CGRA left-to-right. Since FUs typically have a smaller latency than one processor cycle, it is possible to fit more than one FU (and, subsequently, column) in sequence within a single processor cycle. Each set of grouped columns where data propagation takes one processor cycle corresponds to a *level*. The parameters that can be tuned in the design are the *number of columns*, the *number of rows*, the *number of columns within one level* and, as a consequence of all these, also the maximum *number of operations* per configuration.

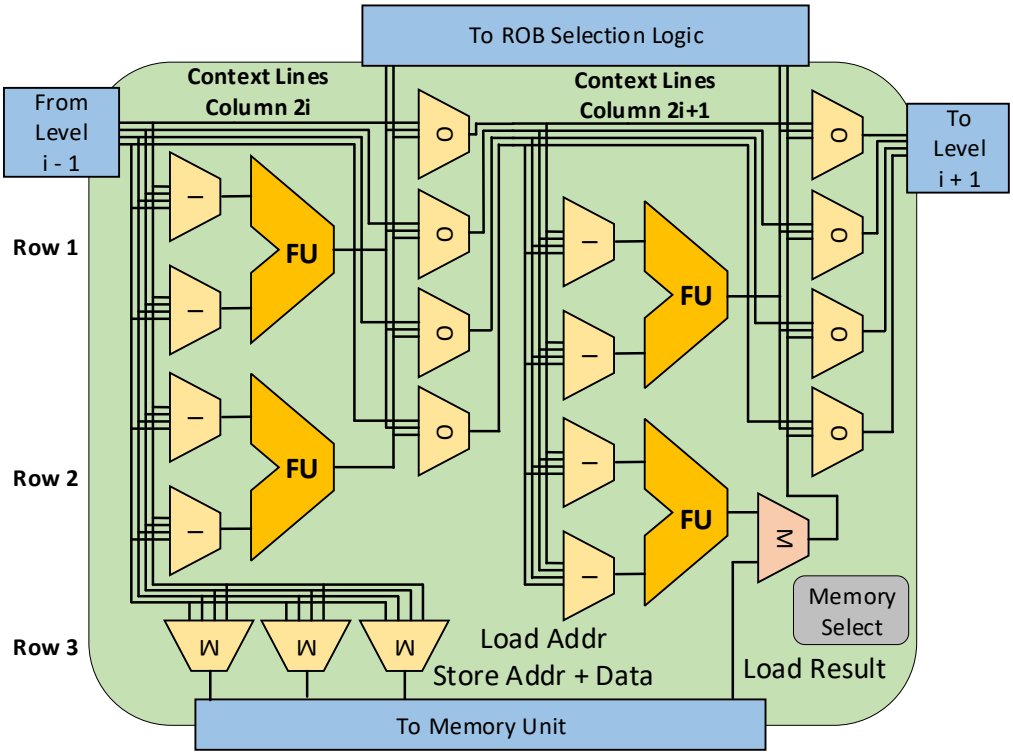
Figure 3.2: Overview of MuTARe's RU.



Source: the author.

To show the detailed structure of a level, a very simple 2×2 CGRA design (with two columns and two rows per level, for a maximum of four operations per cycle) is shown in Fig. 3.3 for illustration purposes. Each column has access to a set of *context lines*, from which *input multiplexers* (*I*, in the figure) select the data that will feed each *FU*. After each

Figure 3.3: Detailed view of MuTARe's CGRA.



Source: the author.

FU, *output multiplexers* (*O*, in the figure, one for each *context line*) select the value that will be propagated to the next column. This value can either be the corresponding value from the previous context line or one of the values produced by the FUs in this column. This particular form of interconnection, where a value written to a context line becomes available to all subsequent columns (if all subsequent output muxes are set to 0) and consumes the context line allows a greedy algorithm to be used for scheduling, as will be shown in Section 3.2.1.

Besides support for integer ALU operations, the CGRA also supports memory and other complex operations such as integer multiplication and division. Access to these structures and functional units is also provided through input multiplexers and output multiplexers, so the same algorithm that maps ALU instructions extends to other operations as long as they can be represented as a position in the CGRA matrix. The maximum number of concurrent memory operations is given by the data cache design.

In the same example from Fig. 3.3, one memory operation can be allocated into each level. To that end, two input muxes, in case of loads (one for selecting a base register value and the other for the selecting the immediate offset, summed up to compute an address), and three input muxes, in case of stores (two for the address and one for the value), select the data that will be forwarded to the memory unit (which will be detailed

in Section 3.2.4). Since a maximum of four operations can be allocated to the level, the memory operation must replace one of the ALU operations. This is done by including a mux that selects whether the result from the last operation (column 2, row 2) comes from the ALU or the memory. The same logic applies for other complex operations that occupy a slot in the CGRA's matrix structure.

The RU's context lines is initially fed with register values from the processor. These are stored in the RU's *input context*. The values produced inside the CGRA are forwarded to a ROB-like structure that stores results and holds the necessary information to enable in-order commit. This ROB receives information from one level each cycle using a ROB selection network (also shown in Fig. 3.3) that will be detailed later on in Section 3.2.4.

A RU configuration encodes the mapping of instructions in a trace (potentially spanning multiple basic blocks) to FUs in the CGRA and their corresponding interconnections.

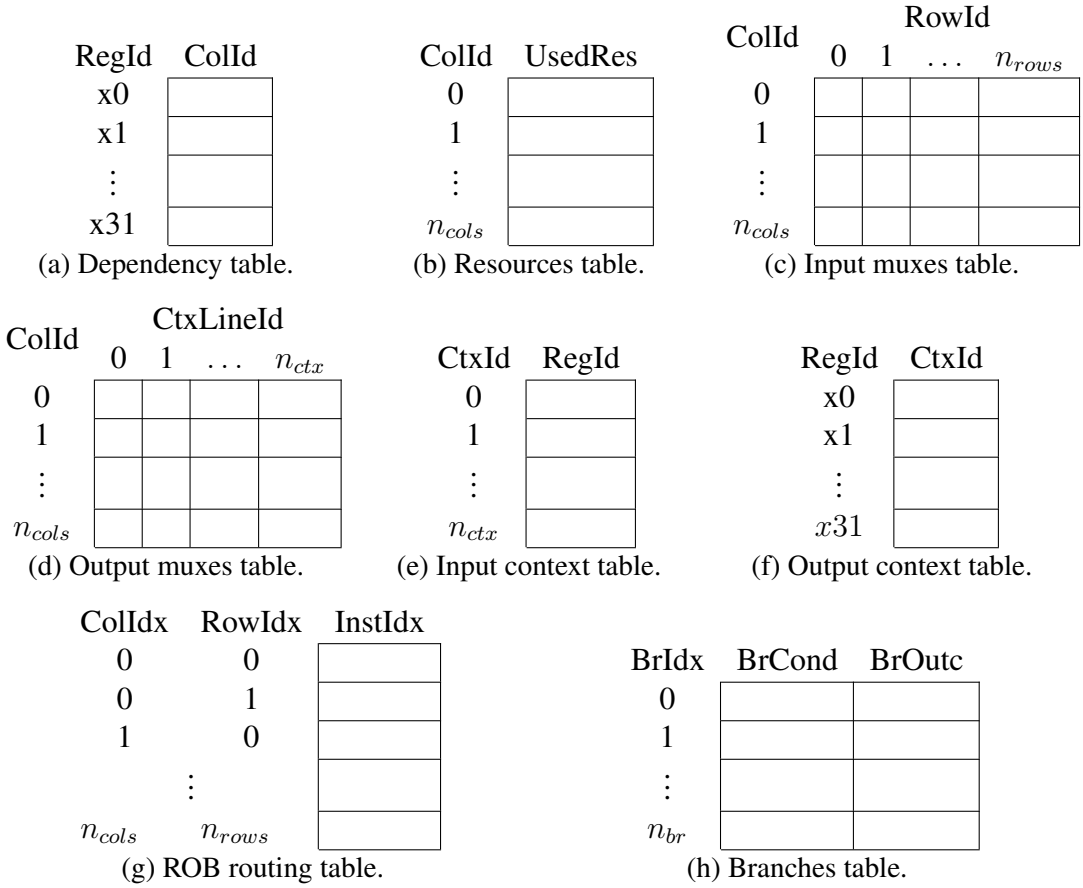
3.2.2 Binary Translation Module

One of the key challenges when deploying reconfigurable architectures is how to generate reconfigurable code. The process can be broken down into two steps:

- *region selection* requires choosing, in the original program, the code regions that are suitable targets for reconfigurable acceleration. Hot candidates are regions which are executed often, are easily predictable (few branches) and have few memory operations.
- *region mapping* consists in implementing that particular region as a reconfigurable datapath in the accelerator.

One of the key advantages of MuTARe over similar works is the ability to automatically generate CGRA configurations on-the-fly, eliminating the traditional mapping issues associated with accelerators (decribed earlier in Chapter 1). The translation process involves scheduling the incoming instructions into the FUs available in the CGRA matrix structure (described in Section 3.2.1), and is accomplished by the BT module, which is implemented as a hardware module operating after instruction commmit to avoid any potential performance hazards. Compared to OoO processors, which also dynamically select instructions and schedule them into functional units, the approach taken by Mu-

Figure 3.4: Tables used in the BT process.



Source: the author.

TARe generates these schedules only once and encodes them in a CGRA configuration, saving for future reuse and amortizing the power costs of dynamic scheduling.

A description of the translation algorithm starts first with a simplified version for mapping only ALU and branch operations into the CGRA's matrix structure depicted in Fig. 3.2. In this case, each supported instruction is identified by an *operation*, a *destination register* and two *input registers*. The extensions for immediate operations, memory operations and more complex operations will be decribed afterward.

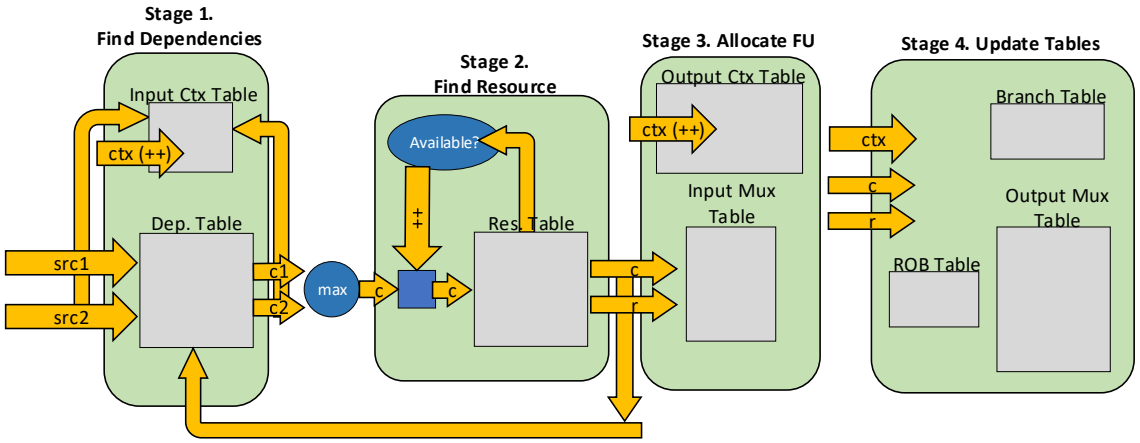
Algorithm 1 describes in detail the (simplified) translation process implemented by the BT module as a 4-stage pipeline with the support from 8 tables (detailed in Fig 3.4). When generating a configuration, each incoming instruction is allocated to the lowest available FU where its input operands are ready, and the allocation information (e.g. functional unit position, multiplexers setup) is saved. This step is repeated until the configuration is full or an unsupported operation is detected. The current implementation supports the execution the whole the RISC-V base integer instruction set, with integer computational instructions, control transfer instructions, loads and stores.

Algorithm 1: BT Algorithm.

Data: Empty configuration tables, an instruction trace stored in $inst\_Q$.
Result: CGRA configuration encoding the execution of the instruction trace.

$inst\_idx \leftarrow 0$;
$br\_idx \leftarrow 0$;
while $inst \leftarrow dequeue(inst\_Q)$ **do**
  $(op\_class, reg_{src1}, reg_{src2}, reg_{dst}, imm) \leftarrow$ decode$(inst)$;
  `// Determine smallest column index where dependencies are available. Update input context if needed.`
  $c_1 \leftarrow tbl_{dep}[reg_{src1}]$;
  if c_1 *is empty* **then**
    $c_1 \leftarrow 0$;
    $tbl_{ictx}[ctr_{ctx}] \leftarrow reg_{src1}$;
    $tbl_{octx}[reg_{src1}] \leftarrow ctr_{ctx}$;
    Increment ctr_{ctx};
  $c_2 \leftarrow tbl_{dep}[reg_{src2}]$;
  if c_2 *is empty* **then**
    $c_2 \leftarrow 0$;
    $tbl_{ictx}[ctr_{ctx}] \leftarrow reg_{src2}$;
    $tbl_{octx}[reg_{src2}] \leftarrow ctr_{ctx}$;
    Increment ctr_{ctx};
  $c \leftarrow \max(c_1, c_2) + 1$;
  `// Increase column index until a functional unit is available`
  while $tbl_{res}[c, op\_class] = 0$ **do**
    Increment c;
  `// Allocate FU, get row index and configure input mux`
  $r \leftarrow (max\_res) - tbl_{res}[c, op\_class]$;
  Decrement $tbl_{res}[c, op\_class]$;
  $tbl_{imux}[c, r] \leftarrow (tbl_{ctx}[reg_{src1}], tbl_{ctx}[reg_{src2}])$;
  `// Determine position in output context`
  if $tbl_{octx}[reg_{dst}]$ *is empty* **then**
    $tbl_{octx}[reg_{dst}] \leftarrow ctr_{ctx}$;
    $octx \leftarrow ctr_{ctx}$;
    Increment ctr_{ctx};
  else
    $octx \leftarrow tbl_{octx}[reg_{dst}]$;
  `// Update tables`
  $tbl_{omux}[c, octx] \leftarrow r$;
  $tbl_{dep}[reg_{dst}] \leftarrow c$;
  $tbl_{ROB}[c, r] \leftarrow inst\_idx$;
  if op_class *is branch* **then**
    $tbl_{br}[ctr_{br}] \leftarrow (inst\_idx, br_{cond}(inst), br_{outcome}(inst))$;
    Increment ctr_{br};
  Increment $inst\_idx$;

Figure 3.5: Overview of MuTARe's BT unit.



Source: the author.

A detailed view of the BT module with the pipeline stages is provided in Fig. 3.5. Next, a description of each of the pipeline stages follows.

Stage 1: Find Dependencies. The first stage consists in identifying the position in the CGRA where the values of the source registers are located. To that end, the dependency table (tbl_{dep}) (which holds the index of the column where each output register has been last written to) is accessed to find the column where each of the input register dependencies have been computed. The number represents the smallest column index where the incoming operation can be allocated (the result is not available earlier). If either dependency is not found in the dependency table, then it has not yet been produced, and must be fetched from the *input context* (the set of registers which are read from the core's register file), and the input context table (tbl_{ictx}) is updated accordingly.

Stage 2: Find FU. While the first stage finds the *lowest* possible column where an operation could be scheduled given the dependencies, the second stage finds, from the lowest column onwards, a column with an available FU for the operation. This process is accomplished by accessing the resoures table (tbl_{res}, a table storing the occupancy status of each functional unit in each column) with the column index. Typically, a single access (with the lowest column from stage 1) will result in an available FU. If that is not the case, this process causes a pipeline stall, as each cycle the column index is increased and the next column is scanned until the resource is found.

The result of this stage is a $(row, column)$ pair which identifies the position in the matrix where the operation can be scheduled. This information must be forwarded in this stage to the first stage in order to avoid pipeline bubbles, since it affects the dependencies table. Since the resources table is essentially a bitmap, it presents low latency and its

output can be forwarded with no latency overheads.

Stage 3: Allocate FU. With that information, the instruction is allocated to the corresponding FU. The resources table is updated to reflect the allocation, and the input muxes table (tbl_{imux}) is updated to select the proper input to the FU. The output context table (tbl_{octx}) is checked to determine the position in the context to allocate the destination register value (it is overwritten if previously produced inside the CGRA, otherwise a new entry is allocated).

Stage 4: Update Tables. In the final stage, all tables are updated. The output context table is updated with the position of the destination register in the context. The ROB routing table (tbl_{ROB}) is updated with the index of the instruction being currently mapped. Finally, if the instruction is a branch, the branches table (tbl_{branch}) is also updated with the index of the instruction in the sequence, the branch condition ($=, <, >, \ldots$) and the branch outcome (taken or not taken); these branches are converted into assertion statements. Branches whose destination may change the next time the same sequence is executed (i.e. branches with link register) are not supported, as that would require also storing the destination of each branch, making the translation process expressively more costly.

Terminating the Translation Process. The algorithm will terminate when an unsupported operation is detected or an attemped resource allocation fails. The first condition can be detected in the first pipeline stage. The second condition may be detected:

- in the first stage, when no more context lines are available (lines 9 and 15) or if no more levels are availabe (line 17, when the dependency is produced in the last CGRA level);
- in the second stage, if the last level is reached and no FU is found (line 18);
- in the third stage, if no more context lines are available (line 24);
- in the fourth stage, if no more branches are available (line 33).

When this process completes, a configuration for the CGRA is generated and stored in the configuration cache, along with the PC of the first instruction transformed. Each configuration contains the operations of the FUs, the crossbar setup and the logical destination registers of each instruction in the sequence so that the register file may be updated later on.

Next, a few important extensions to this basic scheduling algorithm described above (which ony handles operations of the form $reg_{dst} \leftarrow reg_{src1}$ op reg_{src2}) are dis-

cussed.

Handling Immediate Values. A dedicated table is used to store immediate values. These values are detected during the translation phase and at run time loaded to the input context to be available through the context lines from the first level.

Handling Memory Accesses. Memory accesses are allocated as ordinary ALUs, with a few exceptions. Two input muxes, in case of loads (base register and immediate value are summed to compute an address), and three, in case of stores (two for address and one for the value), are encoded in tbl_{imux}. A load operation replaces an ALU operation to preserve the maximum number of instructions in each level (see Fig. 3.3). To prevent stores from being reordered w.r.t. previous loads, a counter is used during the translation phase to store the last level where a store was allocated.

Exploiting speculative execution. The algorithm allows multiple branches, and, subsequently, basic blocks, to be allocated within the same configuration. Since the traslation process generates a configuration based on a single outcome of the branch (the one program path that executed in the first place), configurations containing multiple branches are natually speculative. A run-time mechanism must later on re-execute the branches to confirm if the expected program behaviour is the same as the one encoded in the configuration.

3.2.3 Configuration Cache

The configuration cache stores configurations for the CGRA. It is organized as a PC tag array (to index instruction traces already translated) and a data array (to store the configuration data), and must support enough bandwidth to configure at least one CGRA level each cycle in order to avoid stalls due to reconfiguration. Table 3.1 details this arrangement.

Since each configuration is indexed by a single PC, and encoded an instruction trace spanning multiple basic blocks, multiple traces starting from the same PC cannot coexist in the cache. This strategy prevents two traces starting at the same PC from occupying the cache at the same time, but also simplifies the lookup process. The lifetime of these configurations in the cache is handled using a 2-bit counter, which is incremented whenever the execution of the corresponding configuration is interrupted due to misspeculation (read ahead for how speculative execution in handled in the RU). When the counter saturates, the configuration is erased from the cache.

Table 3.1: Detailed structure of the Configuration Cache.

PC (Idx)	PC (Tag)	Cfg. Data
0		
1		
⋮		
n		

Source: the author.

The size of each configuration varies according to the size of the CGRA, so that should be taken into account during design to balance the configuration load times with the bandwidth offered by the cache. For instance, in sigle-issue, small pipeline processor the configuration load time is critical to achieve high performance. In superscalar cores, however, with larger instruction windows, it is typically less critical to load configurations fast as a higher number of instructions will execute.

3.2.4 Interface with GPP

Executing a CGRA configuration requires four steps:

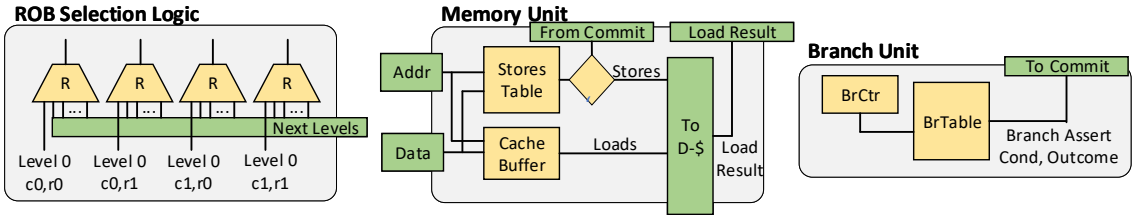
- Check if a valid configuration exists for the next instruction sequence;
- Load the configuration for the next instruction sequence;
- Execute the configuration;
- Commit the configuration;

The first step requires checking in the configuration cache if a configuration exists. Whenever the GPP attempts to fetche a new instruction block from its own instruction cache, a lookup with the same PC also takes place in the configuration cache. Since the configuration cache has a detached tag array to identify if a configuration exists for that PC or not, the power overheads of these additional checks if only marginal. As described in the previous section, each configuration stored in the cache encodes an instruction trace and is indexed by the PC of the first instruction in the translated sequence. When a match is found for a given PC, that means a previous trace starting at that same PC has already been previously translated. Then, the next block needs not be fetched from the instruction cache; instead, execution is offloaded to the CGRA.

The configuration load process starts by identifying, in the first configuration bits, the register values that must be transferred to the CGRA. This information is handled by the GPP, which must transfer the register values to the CGRA. This process may take a

few cycles until the results are available, and its speed is constrained by the number of read ports in the register file. The register values transferred to the RU are stored in the corresopnding position in the input context. While register values are loaded, the configuration process continues. Next, for each level, the input multiplexers, FU operations, output multiplexers and ROB routing are configured. Loading the configuration one level at a time allows the execution to start once the first level is loaded, without waiting for the whole load process to complete.

Figure 3.6: MuTARe's RU structures for handling OoO execution and speculation.



Source: the author.

Executing with In-Order Commit. As discussed in section 3.2.1, a ROB-like structure is used to commit instructions in program order. The left side of Fig. 3.6 shows the ROB selection logic, which routes data produced into each level into one of the ROB write ports. The structure of the ROB is shown in Table 3.2. Besides the value produced by each FU, the destination register of each operation (encoded in the configuration) is also written into the address encoded with that operation. Instruction commit can start as soon as the first (in-order) operation is completed in the CGRA. Moreover, using the ROB structure allows for maintaining precise exception behavior and eases the speculation process, allowing many operations to be committed even when a configuration misspeculation occurs.

Table 3.2: Structure of the simplified ROB used for storing operation's results.

	V	Br	RegDest	RegValue
0				
1				
⋮				
63				

Source: the author.

Handling memory accesses. Two tables are used to that end:

- the *stores table* is a 16-entry table holding store addresses and data. It is filled with a new address and data whenever a store executes in the array. When committing

a store from the ROB, this table is checked for the corresponding store, which is committed to the data cache.

- the *cache bufffer* is a 16-entry direct-mapped cache storing speculative cache accesses. This is required since stores are not sent to the cache until reaching the head of the ROB, but a succeeding load may try to fetch from the same address.

These structures are shown in the middle of Fig. 3.6

Handling branches. As discussed in Section 3.2.2, branches are converted into assertion statements during the configuration generation stage. When a configuration runs, these branches are verified using a special branches table, depicted on the right side of Fig. 3.6. The subtraction required for computing the branch outcome is performed by an ALU and the result written to the ROB. When reaching the head of the ROB, the branch unit is consulted with the result of the branch computation. The *branches table* contains information on the branch condition $(=, \leq, \geq, \ldots)$ and the branch outcome, so a lookup must check if the outcome encoded in the configuration matches the run-time outcome of the branch. If so, then the branch is successful and the execution continues. Otherwise, the last valid CGRA instruction has just executed and CGRA execution is aborted. The recovery process is activated, which works the same way as in traditional OoO processors with the advantage that no register renaming has been carried out.

4 MUTARE - EXTENDED ARCHITECTURE

This chapter describes two extensions to MuTARe. These two extensions push MuTARe into directions previously unexplored by reconfigurable architectures. These two extensions, MuTARe with support for NTV and Approximate computing, leverage the structure of MuTARe's reconfigurable fabric.

4.1 Approx-MuTARe

Approximate computing, the paradigm discussed earlier in section 2.3, consists in improving a system's performance, area, power or energy consumption by introducing controlled computation errors that do not compromise the functional requirements. One of the proposed hardware-level approaches consists in replacing the traditional precise FUs in the designs by approximate implementations that produce an error that can be predicted as a function of the input. Such a strategy can be applied, for instance, to adder designs (GUPTA et al., 2011; GUPTA et al., 2013; ALMURIB; KUMAR; LOMBARDI, 2016), multiplier designs (KULKARNI; GUPTA; ERCEGOVAC, 2011; REHMAN et al., 2016), and divider designs (HASHEMI; BAHAR; REDA, 2016), and can improve delay, area, and power by up to 70% (SHAFIQUE et al., 2016).

In the context of application-specific accelerators, hardware-based approximation (approximate FUs) can be easily deployed, considering that the nature of the workload is known. However, the concept remains challenging to apply in GPPs, since most of the power consumption is spent with control overheads (such as dynamic instruction scheduling) rather than computations. MuTARe, however, provides a suitable structure for applying this strategy in a general-purpose context since it can generate custom datapaths at run-time for different forms of applications and replace control overheads with computation. Therefore, as execution is switched from a GPP (large control overheads in power consumption) to a combinatorial CGRA (large amount of FU), the optimization margin for applying approximate FU is increased.

This section describes Approx-MuTARe, an extension to the base MuTARe architecture described in chapter 3, that can, using a single execution unit, provide transparent acceleration capabilities for precise, already-deployed applications (same benefits as the base MuTARe architecture), and also extract the benefits from approximate computations in emerging-error tolerant workloads. The proposed approach represents an improvement

over previous work in hardware-level approximation, which was restricted to application-specific designs, and also an improvement over traditional reconfigurable architectures, which are restricted to precise execution units. The emphasis in this work was given on the design of the approximate RU; however, the discussion below includes other aspects from the system implementation, from the architectural design to the compiler changes required to map approximate instruction sequences to the RU.

4.1.1 Architectural Changes

Approx-MuTARe is based on the base MuTARe architecture described in Chapter 3. Two changes are required in the architecture to provide support for *approximate computing*: approximate FUs and an approach to map applications into the approximate FUs with controllable quality guarantees.

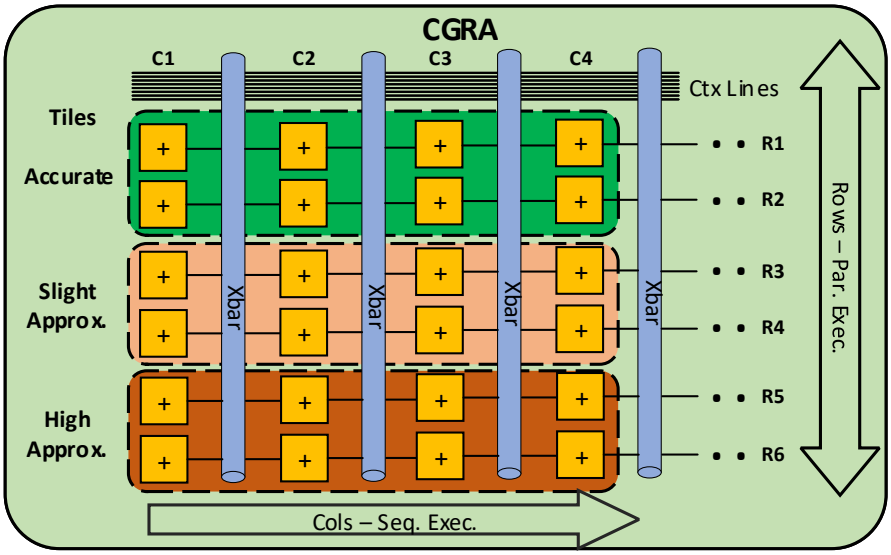
Deploying approximate FUs. The first step required for supporting approximate execution in the CGRA is to make approximate FUs available for use. Different applications present different degrees of error-tolerance, and, even within a single application, different regions or operations may present different error-tolerance. Therefore, an *approximate accelerator* must provide FUs with different error behaviors.

Considering this requirement, approximate FUs proposed earlier can be classified into two groups: those with a *configurable* accuracy (can be set only once, at design time) and those with *reconfigurable* accuracy (can change at run time). While the latter presents more flexibility and can better adapt to different applications, Approx-MuTARe uses accuracy-configurable FUs. The reason for that is that, under the *dark silicon* condition, where power presents a significantly higher cost than area, it is more reasonable to optimize for power than area.

To extend the base MuTARe architecture with approximate FU, the matrix arrangement in MuTARe's CGRA, shown earlier in Fig. 3.2, is organized into *accuracy tiles*. Each *accuracy tile* is a set of contiguous rows and columns implementing the same *accuracy mode*, i.e., containing accuracy-configurable FUs with the same accuracy level, and can be individually power-gated to avoid dissipating leakage power.

Fig. 4.1 provides an example. In this arrangement, three accuracy modes are provided: a precise one (for applications or code regions requiring precise execution), a slightly approximate one (small errors) and a high approximate one (large errors). Each accuracy tile covers four columns, with two rows in each. Since one of the modes is the

Figure 4.1: Approx-MuTARe's CGRA organized into accuracy tiles.



Source: the author.

precise one, this arrangement can also be used to execute applications already deployed, just like the base MuTARe architecture.

Extending the Instruction Set. As mentioned earlier, since approximate execution requires semantic information in order to assess the application quality, ISEs that allow manually programming the reconfigurable fabric. The configurations are generated offline and stored in the *heap* region of the program binary, and loaded at run time.

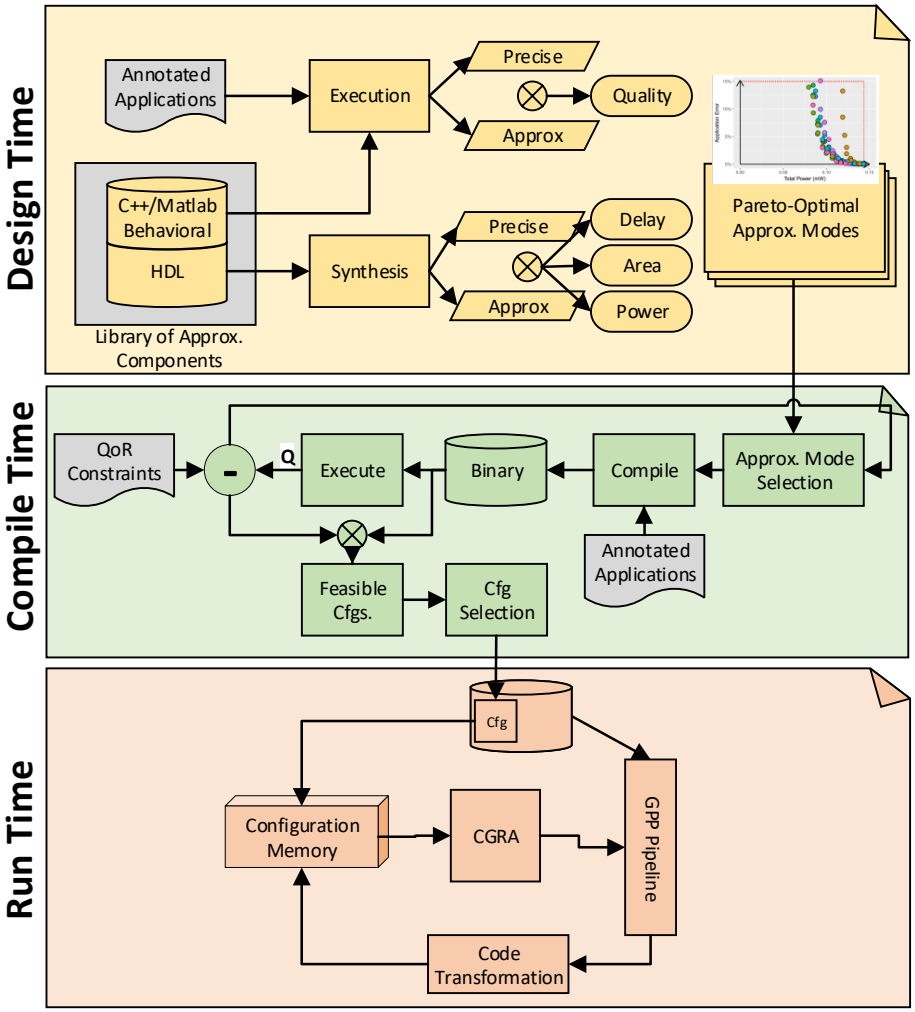
- *LD_MuTAReCfg Reg(Imm)*. Loads the configuration stored into address $Reg + Imm$ into the CGRA for execution.
- *Disable_MuTAReBT*. Disables the MuTARe transparent binary translation system, turning the configuration cache into a software-managed memory. Moreover, disables the automatic code fetch engine that compared the PC of the next fetch block with a possibly-existing configuration.
- *Enable_MuTAReBT*. (Re-)Enables the MuTARe transparent binary translation system.

The next section covers the required changes in the design flow (to select the approximation modes that should be implemented), the compilation flow (to select the approximation modes that should be used, given the Approx-MuTARe fabric) and the execution flow (how Approx-MuTARe behaves at run time).

4.1.2 Design, Compilation and Execution Flow Changes

A hybrid approach to generate configurations either statically or dynamically is presented next. The goal is to provide both the benefits of **transparent acceleration** (via dynamic code transformation, such as in (BECK; RUTZIG; CARRO, 2014; LIU et al., 2015)) **for already deployed non-approximate workloads** and also **improved power-efficiency** (via approximate configurations generated at compile time, such as in (VENKATARAMANI et al., 2013)) **for emerging approximate applications**. To do so, the transparent *binary translation* mechanism proposed in (BECK; RUTZIG; CARRO, 2014) is hardware-implemented and ISEs to configure and execute approximate application kernels in the CGRA are provided.

Figure 4.2: Design flow for Approx-MuTARe.



Source: the author.

Generating Configurations for Approximate Execution at Compile Time. One approach to this process is as follows: The compiler is provided with models of approximate computations (i.e. the effects of approximating particular operations) and their corresponding power savings. These are the approximate modes supported in the CGRA. It starts from an accurate CGRA configuration and changes blocks of operations in the application's DFG into their approximate counterparts which are mapped to the approximate CGRA tiles. This approximate implementation is compiled, simulated, and the quality achieved is compared against the constraint specified by the programmer; if the constraint is met, then this configuration is considered *feasible*; otherwise, it is discarded and a new one is generated. By the end of the process, the configuration yielding the lowest power consumption in the set of *feasible configurations* is selected.

For annotating approximate computations, three approaches are possible. The programmer may specify the beginning and end of pure regions of code, and all computations in that region are replaced by their approximate counterparts (this is the approach taken in (ESMAEILZADEH et al., 2014)). Alternatively, the programmer may annotated variables in the code which are subject to approximation (as in (SAMPSON et al., 2011)). Finally, as compiler techniques become more robust, some steps in this process can already be automated. For instance, the method in (MISAILOVIC et al., 2014) requires only accuracy and reliability constraints to find a suitable mapping of operations into approximation modes automatically.

In this work, an emphasis was given to the design of the approximate RU, so a manual annotation method of replacing operations flowing into approximate datatypes by their approximate implementations was used.

Generating Configurations for Precise Execution At Run Time The system works by default in the dynamic code transformation mode, in which the approximate tiles are disabled by power-gating. In this mode, the configuration memory starts empty and is filled as the program executes by the code transformation unit, which implements the algorithm described in (BECK; RUTZIG; CARRO, 2014).

Loading and Executing Configurations Each configuration is indexed in the memory by its instruction cache address (i.e., the PC of the first instruction). For new binaries, special ISEs are provided to load custom static configurations into the memory and disable the dynamic code transformation, if desired. Each time an instruction fetch occurs, both the *instruction cache* and *configuration memory* are simultaneously looked up. In case a matching configuration is found, it is loaded into the CGRA: the necessary

tiles are awakened from power-gated state, the FUs and crossbars are set up, and the input context is filled with values from the register file. Each execution may take multiple cycles, depending on the *depth* of the configuration. The results are written to the output context and, when the entire configuration has executed, these values are committed to the register file.

4.2 NTV-Aware MuTARe

MuTARe is capable of accelerating an application by offering a combinatorial reconfigurable fabric with a significant amount of FUs that can execute a large number of operations concurrently, exploiting the application's ILP. This strategy is capable to save significant power compared to OoO superscalar cores, since instruction schedules need to be generated only once and can exploit a greedy algorithm, and additionally, there is no need to store intermediate values for computation (only final results). On the other hand, the significant amount of FU operating concurrently (and potentially dissipating significant static power) may present a significant overhead when MuTARe is coupled to very simple processor cores, and tries to present itself as an alternative in these scenarios.

NTV computing was earlier presented in section 2.2 as a strategy to lower the power consumption to levels that are not achievable with traditional designs, exploiting the quadratic relationship between V_{dd} and power consumption. However, as discussed, a circuit's V_{dd} cannot be simply lowered to NTV levels without redesign, because structures will scale differently inside the core, limiting the range where DVFS can be safely applied without compromising the functional requirements (GOPIREDDY et al., 2016), and also outside the core, making chips run at different frequencies for the same V_{dd} level due to process variation (KAUL et al., 2012). Moreover, the frequency will drastically drop when moving from STV to NTV computing, incurring a significant performance loss in serial execution and requiring the use of multi-core to mitigate that performance loss (PINCKNEY et al., 2013).

This section will show how MuTARe provides a structure that is more suitable for NTV computing than traditional designs. As shown for the case of approximate computing in Section 4.1, since MuTARe's RU will be responsible for the majority of power consumption, significant optimizations can be carried out in it alone, which will translate to system-wide improvements. Traditional pipelined designs are not regular, with each stage containing different amounts of logic and state-holding elements. Redesign

for NTV can be significantly challenging in those cases. When designing MuTARe's CGRA for NTV, however, since it is a regular design, only one part of the array needs to be redesigned and can then be replicated.

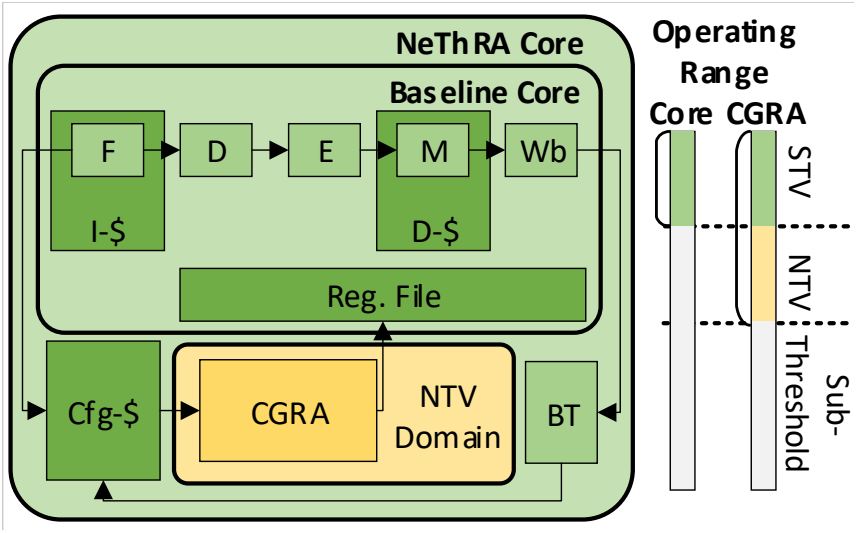
To wrap-up, **MuTARe-NTV exploits three key insights**:

- *performance losses* can be potentially compensated by the extra ILP exploited in the CGRA;
- the combinatorial CGRA structure can be voltage-scaled down to NTV since it presents no state-holding circuits;
- the remaining structures are kept in separate (STV) voltage-island that can run at twice the frequency from the CGRA, thus providing data to feed the CGRA at twice the speed;

4.2.1 Architectural Changes

To make MuTARe support NTV, two architectural modifications must be introduced. The first one involves bringing NTV to MuTARe; the second one involves dealing with one of the challenges of NTV computing.

Figure 4.3: NTV-MuTARe's voltage islands organization.



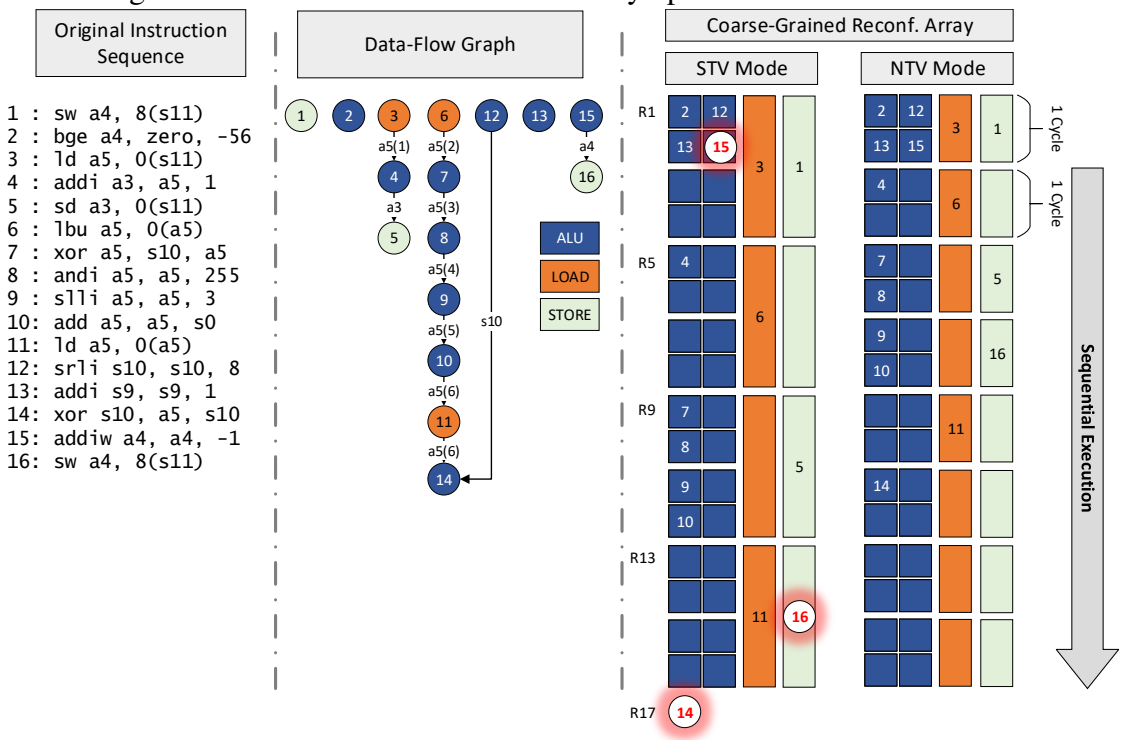
Source: the author.

NTV voltage rail. The premise of MuTARe is to provide a regular structure where the design changes required for supporting NTV operation are simplified compared to those in a complex processor pipeline. To that end, MuTARe's design is extended with one additional voltage rail which will operate at NTV level and feed the regular structures

(such as the CGRA), while the others will run in the lowest-possible STV level. While this may seem suboptimal at first, it should be noted that in MuTARe the RU will be responsible for a significant fraction of the run-time power consumption, as the switching activity and thus dynamic power consumption in the GPP is reduced due to CGRA acceleration.

One important decision is to select, then, the structures in the design that the NTV rail will feed and the ones that will run in STV. Different tradeoffs exist in that matter. The BT unit will operate in the same voltage domain (STV) as the base processor since it behaves like an additional pipeline stage and could cause stalls if operated at a lower frequency. The configuration cache can operate either in STV or in NTV; STV operation allows for larger configurations (as the load times can be improved by running the memory at twice the logic frequency), while NTV allows for lower power consumption. The same is true for the structure of the input context, the ROB and the additional structures for speculation described in section 3.2.4. Finally, the CGRA will operate in NTV level, as its structure will be responsible for a significant fraction of the total power consumption.

Figure 4.4: Accelerated chains of memory operations in NTV-MuTARe.



Source: the author.

This work has begun the investigation of these design decisions with the arrangement shown is shown in Fig. 4.3, where only the CGRA is operated at NTV and the

remaining structures are kept at STV. The V_{dd} of the CGRA is selected such that the delay of logic elements is half that of the caches, an idea similar to the one exploited in previous works (ZHAI et al., 2007; GOPIREDDY et al., 2016). This setup was selected because, since the base CGRA has the ability to accelerate chains of data-dependent ALU operations, the memory ones are usually the bottleneck, and therefore improvements are higher. Moreover, accelerating memory instructions allow the effective instruction window in the CGRA to increase.

This is exemplified in Fig. 4.4. This figure shows a RISC-V instruction trace from a real application (CRC32), the DFG for this trace and how it would be scheduled in a CGRA with the structure shown on the right, where each cycle 4 ALU, 1 load, and one store operation can be scheduled. When memory operations take two processor cycles, the instruction window is restricted to ten operations (as scheduling the 11th would require one additional cycle, unavailable in the structure under consideration), resulting in an ILP of 1.66 (ten operations in six cycles). If memory operations take a single cycle, however (which is the effect achieved by exploiting the frequency difference between the CGRA NTV and non-NTV domain of the caches), all 16 operations can be scheduled in the CGRA, resulting in an ILP of 2.66 (an increase of $1.6\times$).

Support for overprovisioning. As mentioned in section 2.2, one of the key challenges of NTV is variability, which affects the power consumption and latency of the manufactured devices.

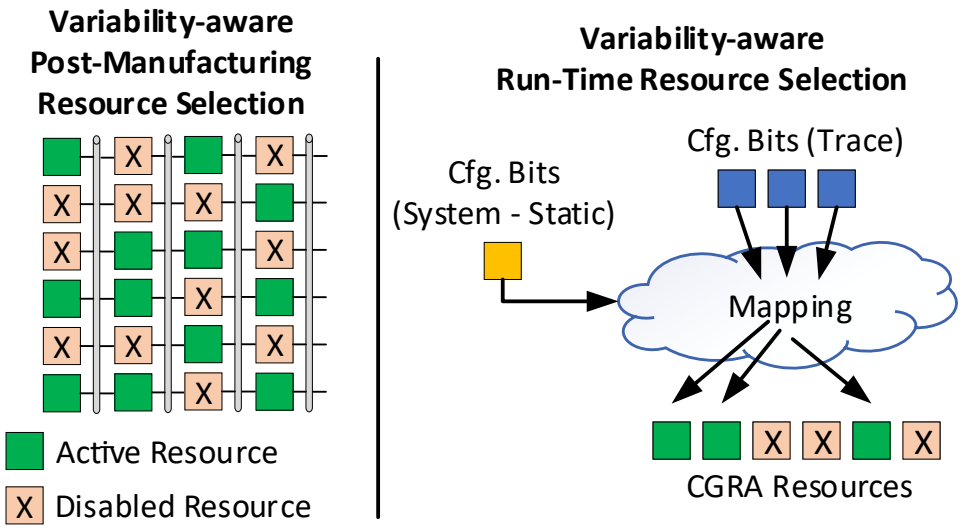
One of the advantages in using a regular design for dealing with variability is the ability to easily deploy an extended design with over-provisioned resources among which the best (faster, more power efficient) are selected after manufacturing while the remaining ones are disabled.

The challenge is to do so in a non-intrusive way, without compromising the BT algorithm.

The critical resource to be protected, in this case, are the functional units, since the multiplexers represent just a small part of the total latency. The circuit delay is affected by units chained together across a set of columns, so more units should be deployed in each column in order to select the fastest.

Fig. 4.5 shows the general strategy used to mitigate the effects of variability in NTV-MuTARe. It considers a MuTARe design where each CGRA column should contain 3 FUs. To avoid the effects of variability, however, increasing the chance that 3 FUs will meet the target latency constraint, the CGRA can be designed with twice (can be

Figure 4.5: Variability management strategy employed by NTV-MuTARe.



Source: the author.

determined based on the desired variability) the amount of FUs. Post-manufacturing, the three fastest blocks in each column can be identified and the remaining ones virtually disabled (by saving this information for later use). At run time, a multiplexer network can map the configuration bits to the appropriate resources, using to that end the static information obtained from post-manufacturing testing.

5 EVALUATION

This section describes how the base and extended MuTARe architectures were implemented and evaluated in different setups. First, a general overview of the evaluation methodology is presented, with all of the tools implemented or employed at some stage in this work. Then, separate sections will present different evaluation scenarios and results.

5.1 Methodology overview, metrics and tools

Different tools and methodologies have been used in each stage of this work. The first experiments used cycle-accurate performance simulators and architectural-level area and energy estimators, while the later ones used performance simulators combined with area and power estimations for hardware synthesis of real processor designs. This section presents a general overview of the tools that were used throughout the work, leaving setup and configurations details for each of the evaluation scenarios presented later in Section 5.2.

5.1.1 Methodology

MuTARe consists of a coupling between one or more GPP cores and a CGRA. In a way, MuTARe can be thought of as extending a GPP core with a reconfigurable accelerator, while maintaining the general-purpose processing capabilities and trying to achieve a better tradeoff between performance and power consumption. Therefore, execution in MuTARe is compared in all scenarios against execution in the GPP core it extends, which will be named *baseline processor* in the experiments that will follow.

Since the goal is to improve efficiency, this work evaluates the designs in four metrics: performance, area, power consumption, and energy consumption. In the case of the extended Approx-MuTARe architecture, one additional metric is taken into account: the accuracy loss from approximate computations.

Most of the benchmarks used in the evaluations come from two different sets. MiBench is a large benchmark set consisting of kernels that are representative of applications commonly found in the embedded systems domain (GUTHAUS et al., 2001). AxBench is a small benchmark set consisting of kernels that are typically found in appli-

cations supporting approximate computations (YAZDANBAKHSH et al., 2016). AxBench also includes, along with each benchmark, a standardized error metric to allow comparison across different works.

5.1.2 Tools: The gem5 Simulator

Different simulation platforms had been initially investigated to provide a base on top of which MuTARe could be implemented. In particular, the Multi2Sim (UBAL et al., 2012), ZSim (SANCHEZ; KOZYRAKIS, 2013) and **gem5** (BINKERT et al., 2011) simulator infrastructures have been considered. In the end, gem5 was chosen due to its detailed microarchitectural model of superscalar cores and widespread adoption in the architectural community, with developers from ARM, AMD and Google currently maintaining the code.

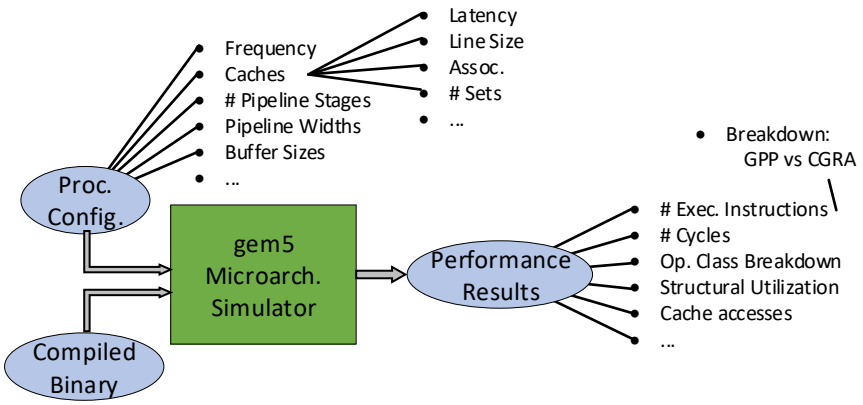
gem5 is a modular platform for computer systems simulation, encompassing both system-level architecture as well as processor microarchitecture simulation models. It supports two operating modes: system call emulation (SE) and full system (FS). In the first mode, calls to the operating system are emulated using a simplified library. The latter mode allows an entire operating system and applications to run on top of gem5.

The microarchitectural simulator uses a decoupled front-end, and back-end simulation engine, where the semantics of ISA instructions (functional model) are implemented in the front-end and processor microarchitecture (timing model) is implemented in the back-end. gem5 supports four CPU (backend) models:

- *AtomicSimple* is an instruction-level model, without any microarchitectural timing;
- *TimingSimple* is an instruction-level model extended with memory timing information; the processor can execute one instruction each cycle unless a stall caused by a memory access occurs;
- *MinorCPU* models the microarchitecture of an in-order, multiple-issue superscalar core; it supports branch prediction and uses the scoreboard technique for handling superscalar execution;
- *O3CPU* models the microarchitecture of a multiple-issue, out-of-order superscalar core; it supports branch prediction, register renaming and load/store reordering with memory dependence speculation.

MuTARe was implemented as an extension to the O3CPU model (and later adapted

Figure 5.1: gem5 simulation flow.



Source: the author.

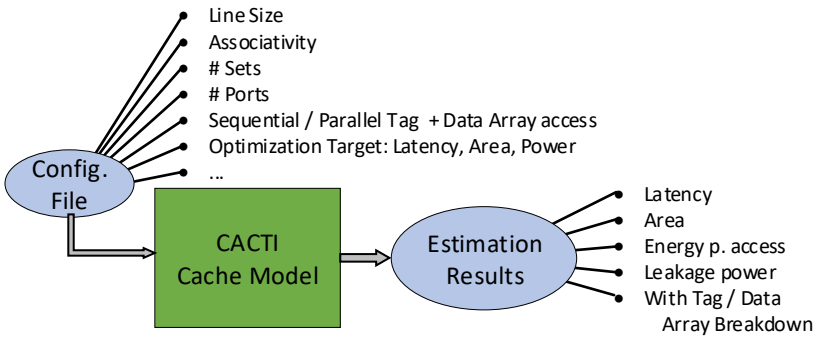
to work with the TimingSimple model as well) and tested and evaluated under the SE mode. An additional BT pipeline stage implementing a high-level version of Algorithm 1 was introduced after instruction commit to generate the CGRA configurations. This information is saved for posterior use in a high-level configuration cache. The simulator stores, for each translated sequence, the execution time in the CGRA in case of no misspeculation and no cache misses. The timing information of those instruction traces is evaluated when the application runs in the O3CPU, and then replaced by the timing from CGRA execution, corrected in case of cache misses or misspeculation. The implementation of MuTARe required 3500 lines of code, compared to the 26000 lines of code required to implement *O3CPU* microarchitectural model.

Fig. 5.1 provides an overview of the simulation flow in gem5, with the inputs and outputs to/from the simulator. The figure also shows typical configuration parameters and reported results.

5.1.3 Tools: CACTI

CACTI is an integrated cache and memory access/cycle time, area, leakage/dynamic power model (BALASUBRAMONIAN et al., 2017). Fig. 5.2 shows an overview of the tool. The user provides a high-level specification of the cache/memory design, including an optimization target (e.g., latency or area or power) and the tool finds an implementation for that target and outputs timing, area and power information. Since these designs are regular, results provided by CACTI tend to be highly accurate. CACTI is thus used in this work to estimate the area and power consumption of all cache implementa-

Figure 5.2: CACTI flow.



Source: the author.

tions.

While CACTI has undergone several revisions throughout the years, being currently in the 7th version, the results it produces are based on 90nm, 65nm, 45nm, and 32nm technology nodes. Therefore, the models do not capture the recent change from planar to FinFET transistors (NOWAK et al., 2004). **FinCACTI** is an extension to CACTI which also models FinFET devices in more recent technology nodes, and was also used in some analyzes in this work (SHAFAEI et al., 2014).

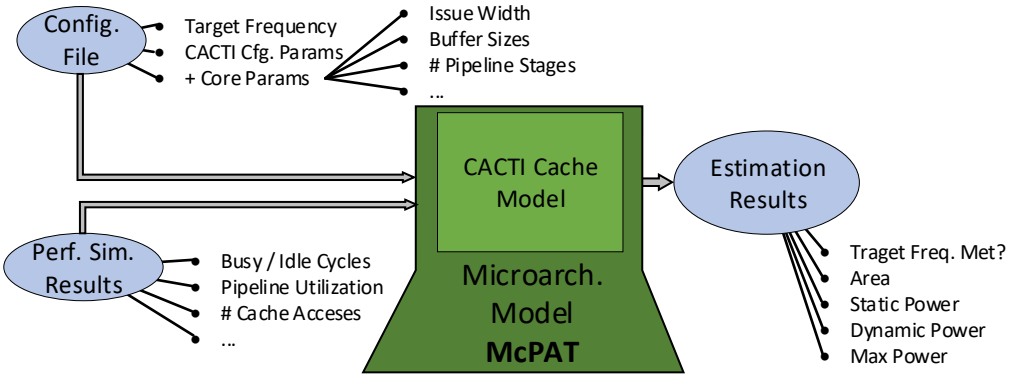
5.1.4 Tools: McPAT

McPAT is a Multi-Core Power, Area and Timing analyzer (LI et al., 2013). It is built around CACTI, including timing, area and power models for the entire CPU cores. To estimate the core power consumption, however, it requires additional information, since the dynamic power consumption has a dependency on structural utilization.

Fig. 5.3 shows the processing flow in McPAT. Two inputs are required by the tool: a processor configuration and performance simulation results. The configuration includes all the information required by CACTI to find a suitable cache implementation and also information on the core design, such as the target frequency, number of pipeline stages and buffer sizes. The performance simulation results can be provided by a simulator (such as gem5) and include the number of idle and busy cycles and how often each structure was utilized. This information is used to estimate dynamic power consumption.

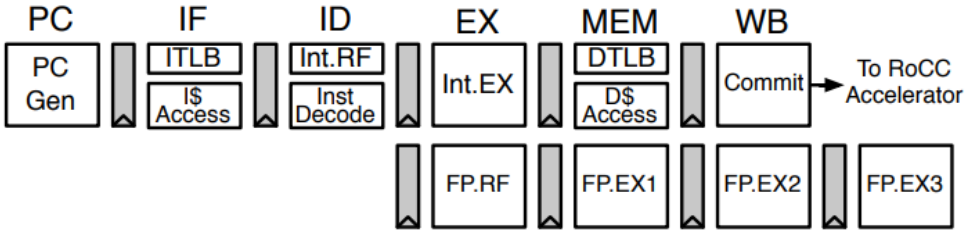
The tool also reports if it is possible to find a processor configuration that meets the target frequency, considering the delay of the caches as reported by CACTI.

Figure 5.3: McPAT flow.



Source: the author.

Figure 5.4: The Rocket core.



Source: (ASANOVIĆ et al., 2016).

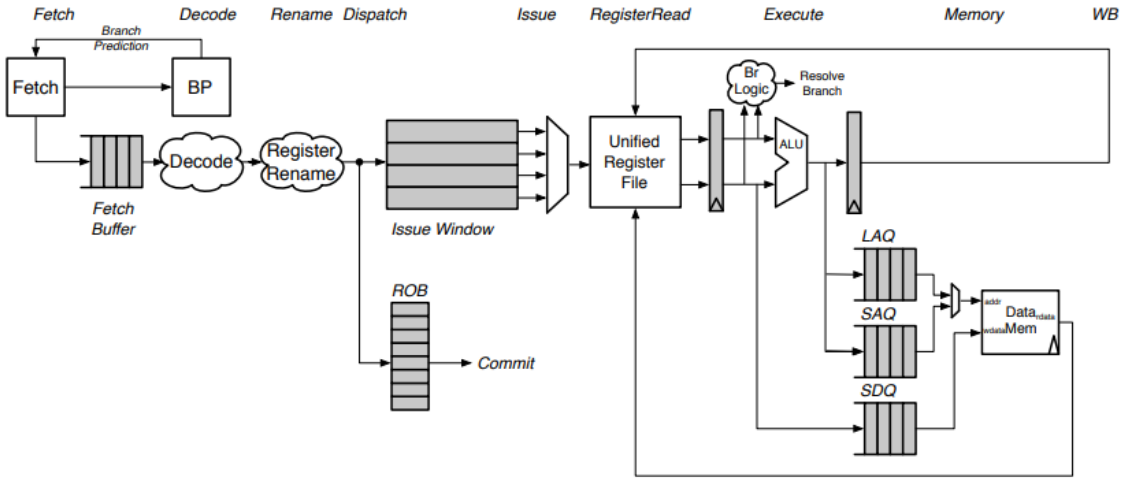
5.1.5 Tools: Rocketchip Generator

Some results from this thesis are based on implementations of real processor cores. In this case, the RISC-V ecosystem was adopted due to the vast number of tools already designed and implemented and free to use. Two cores designed in the University of California Berkeley have been used in this work: the Rocket Core and the BOOM Core (ASANOVIĆ et al., 2016; CELIO; PATTERSON; ASANOVIĆ, 2015).

The Rocket core implements a 5-stage scalar pipeline with branch prediction. It is a very simple processor core, as can be seen in Fig. 5.4. The BOOM core implements a parameterizable OoO superscalar pipeline with branch prediction and advanced structures for handling memory dependencies. An overview of this core is presented in Fig. 5.5.

We evaluate MuTARe in performance, area, power, and energy consumption running a benchmark set representing current IoT workloads. We compare with two GPP cores: a single-issue core, representative of current IoT processors, and a multiple-issue OoO core (2-wide), representing future IoT processors required to match the performance demands of future workloads. As this configuration closely resembles that of ARM's big.LITTLE systems (ARM, 2013), we will refer to the single-issue core as *LITTLE* and the OoO core as *big*.

Figure 5.5: The BOOM core.



Source: (CELIO; PATTERSON; ASANOVIĆ, 2015).

5.1.6 Tools: for Logic Synthesis

Logic synthesis involves transforming a circuit description in Register-Transfer Level (RTL) into a netlist of gates, which may be selected from a technology-specific library and include area and power characterization.

The synthesis was carried out using the Cadence (Encounter / Genus) and Synopsis (Design Compiler) toolsets, depending on the license available at each time. Two cell libraries were used: an industry-grade, 65nm cell library from ST Microelectronics, and 15nm predictive FinFET library designed by Silvaco (MARTINS et al., 2015).

5.1.7 Tools: DVFS model

Initial results to characterize the designs for different DVFS operating levels were based on cell libraries which themselves had been characterized for distinct voltage levels (XIE et al., 2015). However, due to the lack of additional documentation and reliable area estimations for these cells, the approach was abandoned.

Instead, the voltage-frequency relationship was estimated using an EKV-based model proposed in previous work by Markovic et. al (MARKOVIC et al., 2010), which is implemented in the VARIUS-NTV tool (KARPUZCU et al., 2012). This model takes into account the effects of Drain-Induced Barrier Lowering (an effect by which the transistor's effective V_{th} decreases when V_{dd} increases) and temperature (which also affects V_{th}) and can effectively model the voltage-frequency relationship in super-threshold, near-

threshold, and sub-threshold regions. The model was calibrated with results from a recent IRDS report for 15 nm ((IRDS); SYSTEMS, 2017).

5.1.8 Tools: Approximate FU models

The experiments with Approx-MuTARe required models for implementations of approximate FUs and the errors they introduce.

Multi-bit approximate adder designs were constructed by replacing the LSB FA circuits by their approximate counterparts. These designs were taken from previous works (GUPTA et al., 2011; GUPTA et al., 2013; ALMURIB; KUMAR; LOMBARDI, 2016). Most of the C/HDL models for these units are available in the open-source library lpACLib (SHAFIQUE et al., 2016); the remaining ones have been manually implemented.

Five different multiplier implementations were also taken from lpACLib, which designs from two different works (SHAFIQUE et al., 2016; KULKARNI; GUPTA; ERCEGOVAC, 2011); all these designs are built from approximate 2-bit multipliers and accurate adders, which can be combined for larger word lengths.

An *accurate* divider design was taken from OpenCores (HERVEILLE, 2011) and extended to implement the approximation described in a previous work (HASHEMI; BAHAR; REDA, 2016), for which the accuracy can be tuned by changing the number of approximate bits.

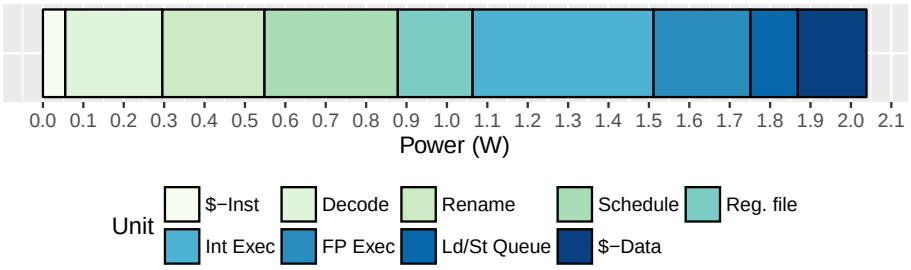
5.2 Results

The results of MuTARe are organized according to baseline system it is compared against. Each result is presented along with a motivation for the applicability of MuTARe in such a scenario. The first two scenarios evaluate the base MuTARe architecture against a wide-issue OoO superscalar core (typically employed in applications requiring high performance) and against a big.LITTLE-like system (typically employed in applications with hybrid requirements, such as in mobile systems). The last two scenarios involve the comparison of NTV-MuTARe and Approx-MuTARe against the base MuTARe architecture.

5.2.1 Scenario 1: High-Performance Computing for General-Purpose Domains

Motivation. The domain of the x86 ISA over the GPP computing market demonstrates the importance of maintaining binary compatibility to allow the execution of software already deployed in new processors. However, implementing a complex (CISC) ISA such as x86 is challenging because it contains over a thousand instructions with variable lengths and addressing modes. To cope with this, x86 processors have long been designed with a decoder that decomposes each x86 instruction into simpler RISC-like operations named μops (INTEL, 1997; HINTON et al., 2001). This step allows executing x86 in a pipelined organization, enabling dynamic scheduling and superscalar execution of ops to exploit high amounts of ILP. However, this whole process is costly: the decoding of each x86 instruction to multiple ops and their dynamically scheduling for concurrent execution require complex structures and logic. Previous works report that decode can account for up to 10% of the total package power (HIRKI et al., 2016), and scheduling for 10-20% (ISCI; MARTONOSI, 2003; FOLEGNANI; GONZALEZ, 2001). Experiments in the scope of this thesis, carried out using McPAT (the tool described earlier in section 5.1.4), have found similar results (Fig. 5.6).

Figure 5.6: Power breakdown in an OoO x86 core.

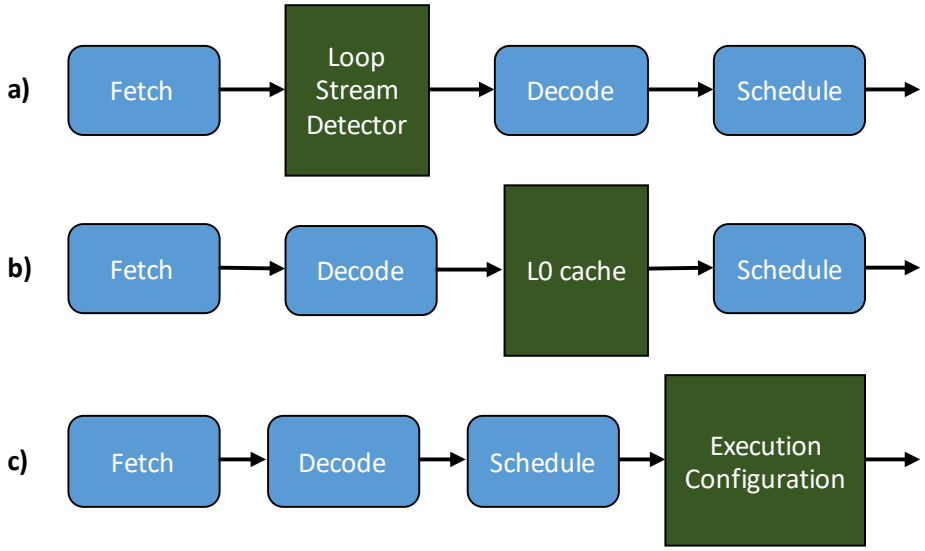


Source: the author.

To reduce the energy costs, recent Intel processors already store decoded μops in a special L0 cache inside the pipeline (DIXON et al., 2010) (GWENNAP, 2010), so that repeating instruction sequences need not be decoded multiple times. As can be seen in Fig. 5.7 (*a* and *b*), this mechanism has been moving deeper into the pipeline over the years, improving the amount of processing that is saved for reuse. While previous works have proposed efficient implementations of the scheduling logic (FOLEGNANI; GONZALEZ, 2001; PALACHARLA; JOUPPI; SMITH, 1997), however, no work has yet proposed to move one step further (Fig. 5.7c) to store the already scheduled μops inside the pipeline. By doing so, repeating instruction sequences could be automatically decoded and scheduled in a single step, skipping the complex pipeline stages that are involved, and

improving energy consumption and performance.

Figure 5.7: Using MuTARe's CGRA to cache instruction schedules in OoO cores.



Source: the author.

In this comparison, MuTARe was investigated as an alternative design strategy to reduce the utilization of the complex hardware structures responsible for decoding and scheduling of repeating instruction sequences in OoO x86 cores and thereby improve energy consumption. This improvement is achieved by encoding these processes into CGRA configurations, levering the structure of a CGRA to enable the recovery of µops already allocated in time and space. Since a CGRA is also capable of efficiently exploiting ILP, performance also improves.

Evaluation setup. In this evaluation, gem5 was used for performance simulation. The *O3CPU* model was used with the parameters shown in Table 5.1 to try to match an Intel Haswell design (HAMMARLUND et al., 2014). The CGRA was configured accordingly with the parameters shown in Table 5.2: the same number of multiplications, load and store operations each cycle, but with the ability to execute four chains of three data-dependent ALU operations in the latency of a single cycle. McPAT was afterward used for estimating the area and energy of the baseline core. These results were combined with the CGRA synthesis results from Cadence RTL Compiler, with the configuration cache being estimated using CACTI. Area and power/energy results are normalized to a 22 nm process technology. Multiple CGRA designs were experimented with by varying the number of levels (15, 30 and 60) and the trace length (from one to 10 BBs per configuration [1]).

MuTARe in this scenario was evaluated using a subset of 9 benchmarks from the Mibench suite, all compiled using *gcc 5.3.0* with the *-O3* optimization flag. A character-

[1]Larger traces were tested but provided high rates of branch misprediction.

Table 5.1: Baseline processor parameters.

Pipeline: 8-wide out-of-order, with 4 ALU ports, 2 mult. ports, 2 load ports and 1 store port. Instr. queue: 60 µops. Load buffer: 72 µops. Store buffer: 42 µops. ROB entries: 192 µops. Memory dependence prediction via store sets.
L1 D+I caches: 32kB each. 8-way set associative, 2 cycles hit latency.
L2 cache: 256kB, 8-way associative, 8 cycles hit latency.
L3 cache: 2MB, 16-way associative, 18 cycles hit latency.

Source: the author.

Table 5.2: CGRA parameters.

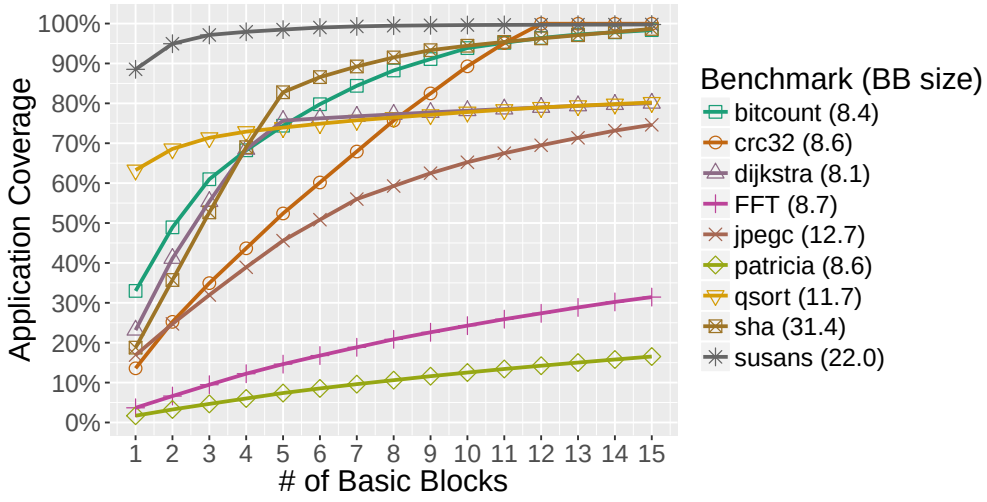
ALUs	12 per level, with a latency of $\frac{1}{3}$ of a cycle and organized as three columnns with four ALU rows each.
Multipliers	2 rows per level, with a latency of three cycles.
Load Units	2 rows per level, with a latency of two cycles.
Store Units	1 row per level, with a latency of one cycle.

Source: the author.

ization of these benchmarks, showing that they cover a broad range of applications with distinct dynamic behaviors, is presented in Fig. 5.8. Each application's Basic Blocks (BBs) are ordered increasingly by their contribution to the execution time (coverage); the figure shows how many BBs are required to achieve a particular coverage rate and also their average size. Some applications, such as susans, have an avg. BB size of 22 µops and a single very distinct kernel that covers 89% of the application, requiring four more BBs to achieve 98% coverage. Other applications, such as bitcount, have many distinct kernels (with a smaller avg. size of 8.4 µops), with the most significant one covering 33% and requiring 16 additional ones to cover 98%. Applications with smaller BBs are typically more difficult to accelerate, because they need better control prediction mechanism, as is the case with applications with too many kernels. MuTARe, however, just like the superscalar processor, can accelerate any application.

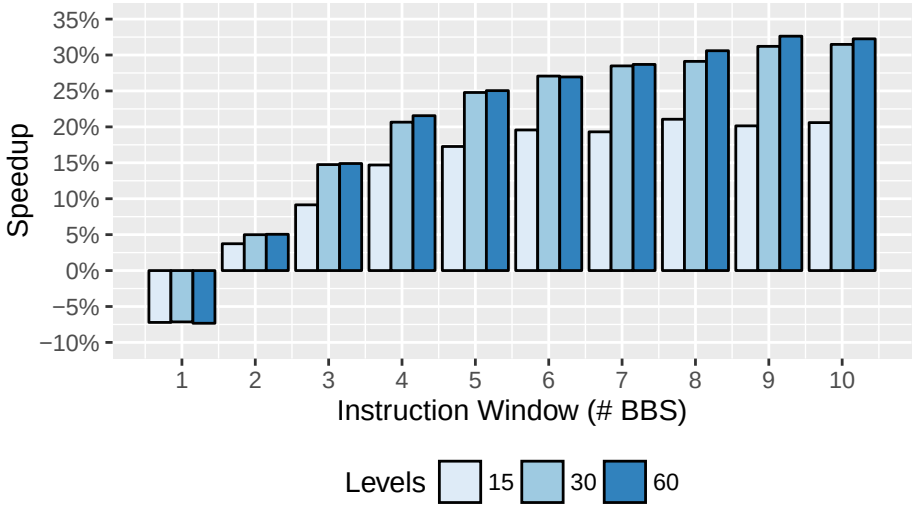
Performance results. Fig. 5.9 presents the geomean application speedup against the superscalar processor for each combination of trace length and number of levels. As can be seen, the speedup increases with both parameters, because more speculation allows better exploiting ILP across multiple basic blocks, and more levels allow it to support larger instruction sequences that amortize the reconfiguration costs. A highest mean speedup of 32.6% is achieved in the best case, with the design using 60 levels. The only slowdown occurs when not using speculation because, in this case, only the baseline processor can execute ops from multiple BBs simultaneously. The results show that the design with 30 levels achieves the best trade-off, given that it enables enough µops to be

Figure 5.8: Number of unique basic blocks required to cover an application.



Source: the author.

Figure 5.9: Geomean speedup for distinct CGRA sizes.



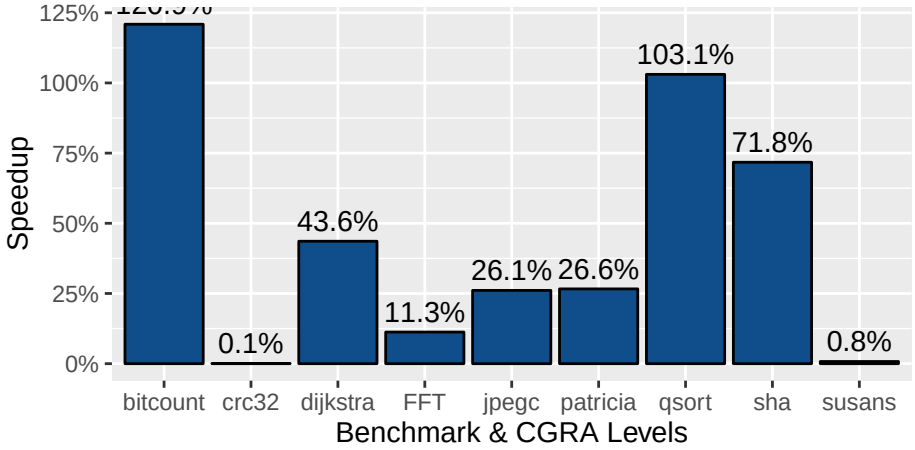
Source: the author.

allocated considering the degree of speculation exploited.

Considering benchmark-specific execution in the 30-level design, Fig. 5.10 presents the speedup and Table 5.3 additional results. There are significant improvements in nearly all applications, because of the CGRA's ability to speed up chains of data-dependent operations, thereby exploiting more ILP than the superscalar. *bitcount* is accelerated the most by 120.9%, because of the high rate of ALU operations (83.7%) and the low rate of µPC (µops per cycle) in the baseline processor (2.4), which indicates many dependencies among these operations.

The only exceptions are *crc32* and *susans*. In these applications, there are many memory dependencies, and the critical path in a configuration lies in chains of load operations. However, the baseline processor can accelerate these chains by using the load/store

Figure 5.10: Speedup achieved by each benchmark with the 30-level design.



Source: the author.

Table 5.3: Detailed results for the execution with 30 levels.

Benchmark	Base µPC	Avg. Configuration			Speedup	Coverage
		µops	BBs	Cycles		
FFT	3.3	61.7	7.6	12.8	1.11	44.0%
bitcount	2.4	66.1	7.8	11.2	2.21	94.9%
crc32	5.1	74.9	9.0	14.0	1.00	11.3%
dijkstra	3.3	69.9	8.7	13.8	1.44	81.4%
jpegc	2.2	60.5	4.6	17.7	1.26	65.4%
patricia	2.2	53.6	6.4	11.7	1.27	51.4%
qsort	1.7	93.6	7.4	18.2	2.03	69.5%
sha	2.3	71.0	2.5	11.7	1.72	70.3%
susans	2.6	69.9	3.0	27.0	1.01	19.3%

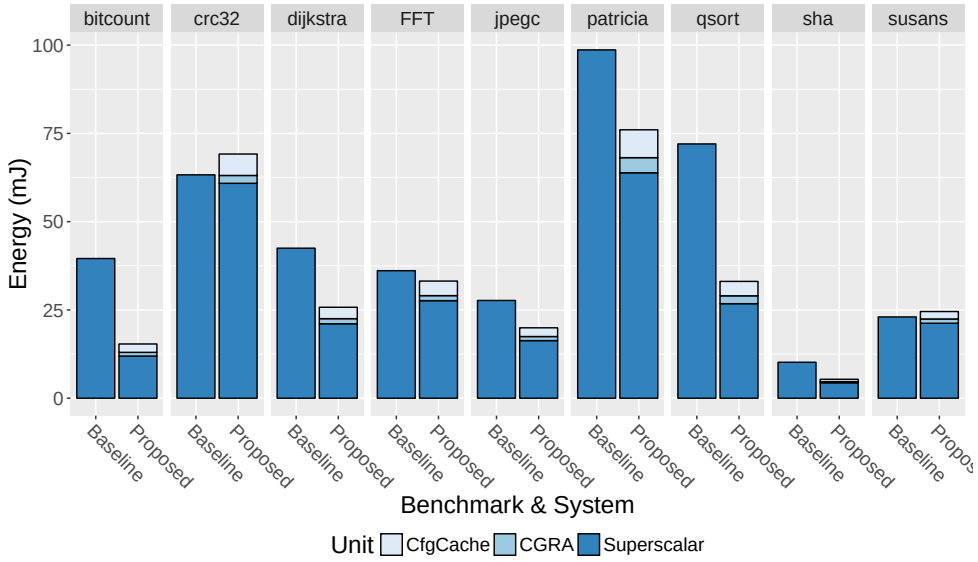
Source: the author.

queue; the code transformation module automatically detects that this is the case and does not save the configuration. Therefore, the coverage (rate of µops that execute in the CGRA) for these applications is small, and there are no performance losses. Additionally, *crc32* achieves the highest µPC in the baseline processor (5.1) and the average configurations in *susans* have the longest execution time (27 cycles) of all applications.

Energy consumption results. Fig. 5.6, presented earlier, shows a power breakdown of the superscalar processor. The costs of decode and scheduling can be amortized in MuTARe, as was previously stated, and also the ones associated with the load/store queue and integer execution, because memory operations are allocated only once when the configuration is generated and execution is more efficient in the CGRA.

Fig. 5.11 shows the energy consumed by each benchmark in the baseline processor and in the MuTARe system, separating the consumption of the superscalar core, the CGRA, and the configuration cache. The results are positive in almost all cases, and

Figure 5.11: Energy consumption by each benchmark with the 30-level design.

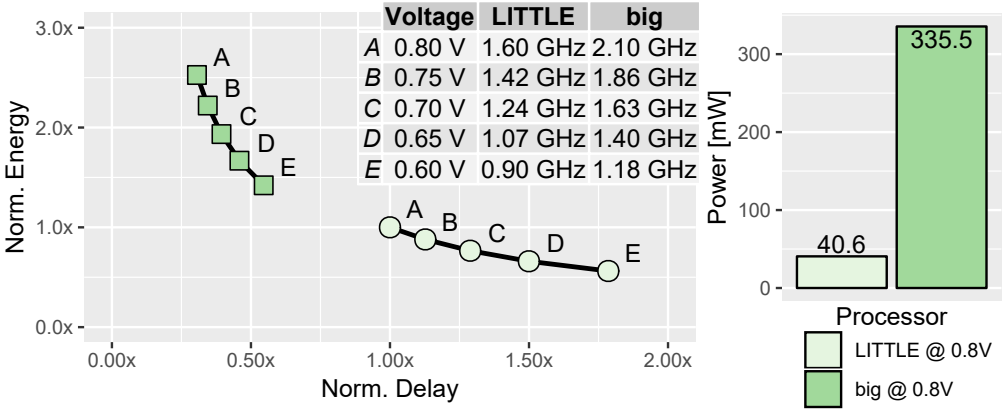


Source: the author.

a geomean reduction of 31.4% from the baseline is achieved. Two factors enable these gains. First, as already explained, the complex pipeline stages involved with instruction decoding and µop scheduling are bypassed, therefore reducing the energy that the superscalar consumes (*Superscalar* bars in the figure). Instead, this information is fetched from the configuration cache (*CfgCache* bars), and the code sequence is executed in the CGRA (*CGRA* bars). Second, the application execution time is reduced and, therefore, applications with substantial speedups (such as *bitcount* and *qsort*) also achieve the most significant energy reductions. There are only two cases with marginal increases, which are *crc32* and *susans*. As was previously stated, coverage for these benchmarks is small, so there is no speedup. However, looking up configurations in the configuration cache add a low energy overhead to these benchmarks.

Area requirements. The CGRA and the configuration cache have areas of 4.18mm^2 and 1.34mm^2, respectively. The superscalar core occupies 13.94 mm^2. Because of this small size, modern processors contain multiple cores and complex graphics processing units (GPUs) in the same die. Compared to Haswell 4770K, which occupies 177mm^2 and has four cores (SHIMPI, 2013), our design introduces only 12.5% area overhead when each of the cores gets replaced by a MuTARe tile.

Figure 5.12: Energy-delay tradeoffs and power in a heterogeneous system.



Source: the author.

5.2.2 Scenario 2: Heterogeneous Computing for Mobile Domains

Motivation. Adaptability is a key to modern mobile systems since a wide range of tasks must be executed with changing and sometimes unpredictable run-time performance and energy requirements. This scenario has driven the design of heterogeneous systems built from multiple GPP cores, each optimized for a different target. However, typically only two core choices are available in such an arrangement: (1) one GPP optimized for low energy, and (2) another one for high performance. An industrial example of such an architecture is ARM's big.LITTLE (ARM, 2013). Therefore, the range of architectural solutions is often limited and performance-energy tradeoffs suboptimal, even when DVFS is available. An example from the experiments conducted in this thesis is shown in Fig. 5.12: an application may execute in a *LITTLE* core, for low energy and slow performance, or migrate to a *big* core, with $3.3\times$ better performance and $2.8\times$ higher energy consumption at nominal voltage. However, besides the wide gap between the two operating ranges, both operating points provide nearly the same EDP. Reconfigurable architectures, on the other hand, can create customized datapaths at run time, thus providing a nearly-continuous range of architectural solutions. As reconfigurable accelerators are typically coupled to GPPs, they can be used to extend traditional heterogeneous designs and improve their adaptability, potentially achieving better performance-energy tradeoffs.

In this comparison, MuTARe was investigated as a more efficient alternative to heterogeneous single-ISA systems which can improve the range of architectural solutions and achieve better Pareto-optimal performance-energy tradeoffs without the need to recompile the program.

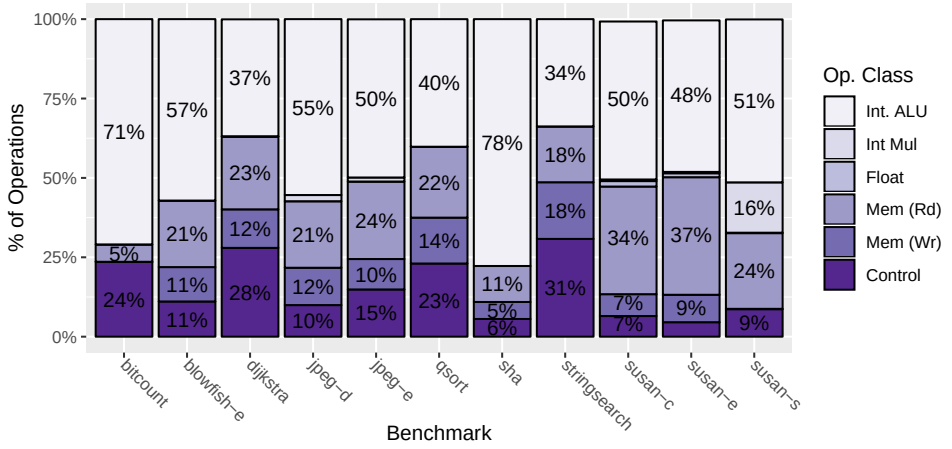
Evaluation setup. For performance evaluation and fast design-space exploration,

Table 5.4: Benchmark groups.

Application Group	Benchmarks
Sensing	bitcount, stringsearch
Communication	dijkstra, FFT
Image Processing	susan (edges, corners, smoothing)
Data Compression	jpeg encode
Security	aes encrypt, sha
Fault Tolerance	CRC32

Source: the author.

Figure 5.13: Benchmark operation class mix.



Source: the author.

the gem5 cycle-accurate simulator was used with the *TimingSimple* CPU to model the *LITTLE* core and the *O3* CPU to model the *big* core. The *O3* CPU was configured with the parameters shown in Table 5.8, and the CGRA with the parameters shown in Table 5.5. This design is significantly smaller than the one used in the previous analysis (Table 5.2). The reason for that is that the baseline processor is also significantly simpler (the *big* core is 2-issue wide, compared to the 8-issue wide core used in the previous analysis), making a smaller CGRA enough for performance improvements, and also in order to limit the power overheads.

11 benchmarks from mibench, representative of the IoT domain, compiled for RISC-V with -O3 and running the small input set were used in the evaluation. Table 5.4 classifies these benchmarks into the IoT application groups referred in a previous work (ADEGBIJA et al., 2018), and Fig. 5.13 shows the operation breakdown for each benchmark.

For area and power evaluation, logic synthesis of real processor designs (Rocket, representing the *LITTLE* core, and BOOM, representing the *big* core) was carried out,

Table 5.5: CGRA parameters.

ALUs	4 per level, with a latency of $\frac{1}{2}$ of a cycle and organized as two columnns with two ALU rows each.
Load Units	1 row per level, with a latency of two cycles.
Store Units	1 row per level, with a latency of one cycle.
Total length: 12 levels (24 columns).	

Source: the author.

using Cadence's RTL Compiler and Silvaco's 15nm standard cell library. FinCACTI was used to model the caches. Results for the CGRA were estimated based on BOOM's FUs and take into account the FUs, interconnects, BT module and configuration cache. BOOM was configured with the same parameters used for the gem5 simulator, depicted in Table 5.8. The Rocket core was targeted to 1.6 GHz and the BOOM core to 2.1 GHz, since similar frequency differences also appear in real big.LITTLE implementations such as in Samsung's Exynos 7420 and Qualcomm's Snapdragon 810 (EXYNOS..., 2015; SNAPDRAGON..., 2015).

Raw synthesis results for the processor cores and caches are presented in Tables 5.6 and 5.7, respectively. The 2-issue BOOM core occupies $10.5\times$ larger area and consumes $11.1\times$ more power than Rocket. As for the 4-issue BOOM, it increases area by $1.9\times$ and power by $1.62\times$ w.r.t. the 2-issue BOOM. The Rocket ALU, used to estimate the CGRA's area and power costs, occupies 3.8% of the Rocket area and is responsible for 3.7% of its power consumption.

Table 5.6: Rocket and BOOM synthesis results.

Processor	Target Freq [MHz]	Num Cells	Area [μm^2]	Power [mW]
Rocket	1,600	27,623	11,982	28.10
(ALU only)	1,600	1,458	455	1.03
BOOM-2W	2,100	287,793	125,178	311.42
BOOM-4W	2,100	593,717	233,109	506.16

Source: the author.

To account for the effects of DVFS, the model by Markovic et. al was used, adjusted to 15 nm, with nominal voltage set to 0.8V, and, based on the evaluation performed in a previous work (GOPIREDDY et al., 2016), considered it can be safely scaled down to 0.6V.

Execution in a Single Operating Point. The first analysis shows the performance-energy tradeoffs when an application executes entirely in a single Operating Point (OP),

Table 5.7: Cache parameters and results from FinCACTI.

Parameter	**LI-I**	**LI-D**
Line Size [B]	32	64
Associativity	2	4
Total Size [kB]	32	32
Cycle time [ns]	0.50	0.48
Leak. Power [mW]	1.92	1.92
Avg. Energy / Access [pJ]	8.5	12.9
Area [um^2]	11,352	11,409

Source: the author.

Table 5.8: Modeling parameters for the *big* core.

Parameter	
Inst-Q, Ld-Q, St-Q, ROB	24, 8, 8, 96
Issue Width	2-Wide
Issue Ports	2 ALUs, 1 Mult, 1 Ld, 1 St

Source: the author.

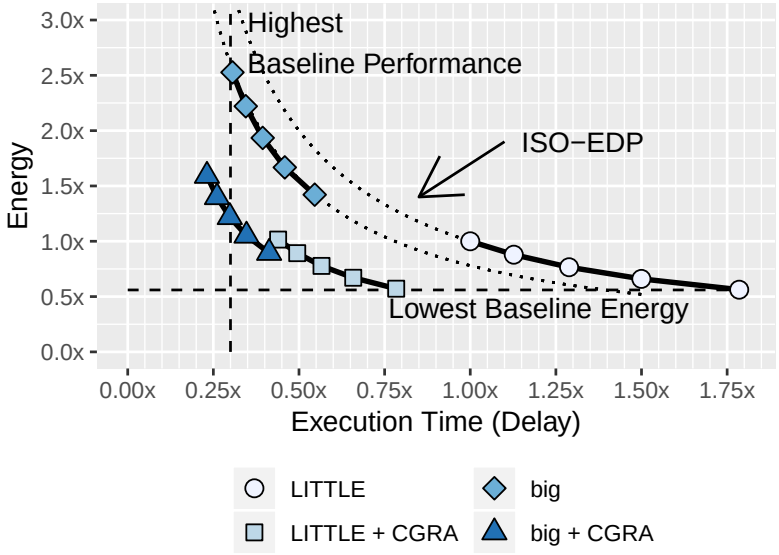
which consists of an *execution unit* and a DVFS level. In the baseline system, two execution units are available (*LITTLE* or *big*), and in MuTARe other two (*LITTLE* or *big* with automatic CGRA acceleration). Each of these four processing units may operate in five distinct DVFS levels, the same ones shown in Fig. 5.12.

Fig. 5.14 presents this analysis. In the figure, execution time (x-axis) and energy consumption (y-axis) are normalized w.r.t. execution in OP $(LITTLE, E)$: *LITTLE* core running at 0.9 GHz. Each point represents an OP; each curve groups together OPs with the same processing unit. Two dashed horizontal and vertical lines show the lowest energy and the best performance achievable in the baseline system (in OPs $(\textit{LITTLE}, E)$ and $(\textit{big}, A)$, respectively).

As can be seen in the figure, in the baseline system a huge $5.8\times$ performance gap (horizontal distance) exists between execution in OP $(LITTLE, E)$ and (big, A). MuTARe closes that gap: when using the proposed CGRA for on-demand acceleration, the *LITTLE* OPs (circles) are moved towards the left (squares). For nearly the same ($< 2\%$ overhead) lowest energy achieved in OP $(LITTLE, E)$, performance is improved by $2.3\times$ in OP $(LITTLE+CGRA, E)$. If DVFS is adjusted to the highest level, moving to OP $(LITTLE+CGRA, A)$, performance improvements increase to $4.0\times$ with no need to use the *big* core at all.

Despite the CGRA's ability to exploit significantly more ILP than *LITTLE*, by offering more FU operating concurrently in each cycle and also accelerating data-dependent

Figure 5.14: Energy-Delay curves for different execution units and DVFS levels.



Source: the author.

operations, the highest baseline performance (vertical dashed line) is still not reached when MuTARe operates in *LITTLE+CGRA* mode. This result suggests that a CGRA coupled only to the *LITTLE* core is not enough to cover all the operating range provided by an heterogeneous design. To further investigate this statement, detailed microarchitectural performance results are provided in Table 5.9. The coverage column (fraction of total instructions executed in the CGRA) shows that often a non-neglectable amount of application code will still execute in the base GPP. This is the case for $(100 - 54.1)\%$ of *jpeg-d*'s instructions, which will execute with an ILP of 0.42 at 1.6 GHz (when coupled to *LITTLE*) or 1.26 at 2.1 GHz (when coupled to *big*). Therefore, in this application, *LITTLE+CGRA* is unable to match *big*'s performance. Meanwhile, in other applications where the CGRA achieves high coverage and ILP, as in *bitcount*, *LITTLE+CGRA* can outperform *big*'s performance by $1.31\times$. In summary, *LITTLE+CGRA* can typically reach *big*'s performance if the coverage is high and the ILP difference from *LITTLE* to *big* is low.

Compared to execution in OP (big, A), which yields the highest baseline performance, leveraging the CGRA (curve with triangles) enables significant energy savings and the potential to reduce execution time. In the same high-frequency OPs, *big*'s performance can be improved by $1.32\times$ (leftmost triangle) while reducing energy consumption to 63% of its original value. Alternatively, by moving to OP $(big + CGRA, C)$, the same baseline high performance can be achieved and energy savings are further improved to 48% of the original value.

Table 5.9: Detailed performance results.

Benchmark	CGRA Perf.				GPP Perf.		
	Cov.	Avg. / Cfg.		IPC	IPC		
	[%]	Ops	Blks		LITTLE	big	bigger
bitcount	91.7	25.3	5.57	3.09	0.82	1.43	1.97
blowfish-e	87.1	18.2	2.13	1.67	0.44	1.31	1.52
dijkstra	84.9	15.0	4.03	1.34	0.41	1.03	1.27
jpeg-d	54.1	16.9	2.28	1.51	0.42	1.26	1.75
jpeg-e	67.9	13.3	1.97	1.28	0.41	1.07	1.40
qsort	68.9	13.4	3.03	1.52	0.32	0.73	0.86
sha	93.6	29.9	1.67	2.83	0.60	1.36	1.55
stringsearch	83.2	14.3	4.52	1.63	0.36	0.76	0.93
susan-c	80.9	12.2	0.76	1.19	0.34	0.94	1.20
susan-e	71.4	12.2	0.63	1.10	0.33	1.04	1.33
susan-s	84.0	10.5	1.01	0.92	0.51	1.21	1.66

Source: the author.

This initial analysis suggests that reconfigurable acceleration can improve a baseline *big.LITTLE*-like design in three distinct ways: 1) by **improving performance** in lowest-energy OP $(LITTLE, E)$, subject to the same energy budget; 2) by **improving energy consumption** compared to the highest-performance OP (big, A), still meeting the same execution time; 3) by **improving performance** in high-performance mode.

Table 5.10: Evaluation scenarios.

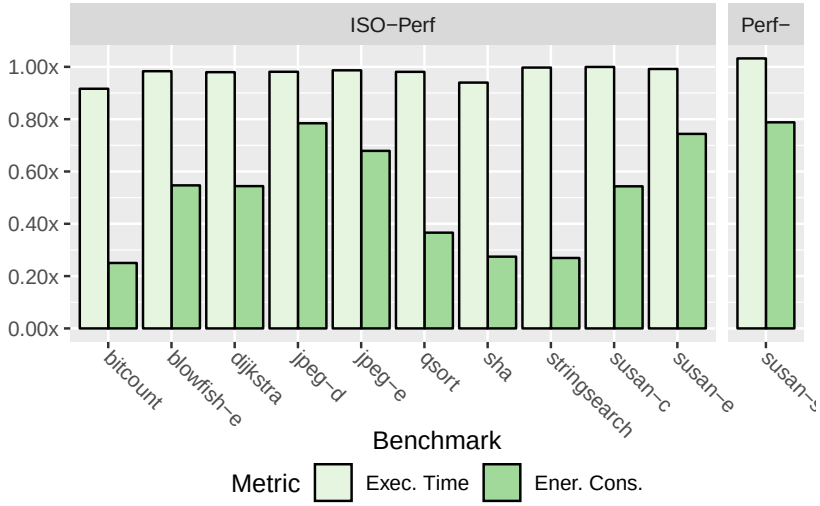
Scenario	System	Mode	Proc. Unit	DVFS	Constraint
I	Baseline	Low-Energ.	LITTLE	Lowest f	
II	Baseline	High-Perf.	big	Highest f	
III	MuTARe	Low-Energ.	LITTLE+CGRA	Dynamic	Max. e
IV	MuTARe	High-Perf.	big+CGRA	Dynamic	Max. t
V	MuTARe	Highest-Perf.	big+CGRA	Highest f	

Source: the author.

From the aforementioned discussion, the scenarios depicted in Table 5.10 are set for comparison. The baseline processor can operate in a *low-energy* mode $(LITTE, E)$, which yields an energy constraint e, or in *high-performance* mode (big, A), which yields a timing constraint t. The MuTARe system can also operate in these two-modes and an additional one: highest-performance. In *low-energy*, the *LITTLE* core is coupled to the CGRA and DVFS can be tuned to minimize execution time subject to the energy budget e. In *high-performance*, the *big* core is coupled to the CGRA and DVFS can be tuned to minimize energy consumption under the timing constraint t. Finally, in *highest-*

performance mode, the *big* core is coupled to the CGRA and operates in the highest frequency to improve and meet performance targets of emerging application unachievable with the *big* core. Results for these three scenarios are discussed next.

Figure 5.15: Results from t-constrained execution.

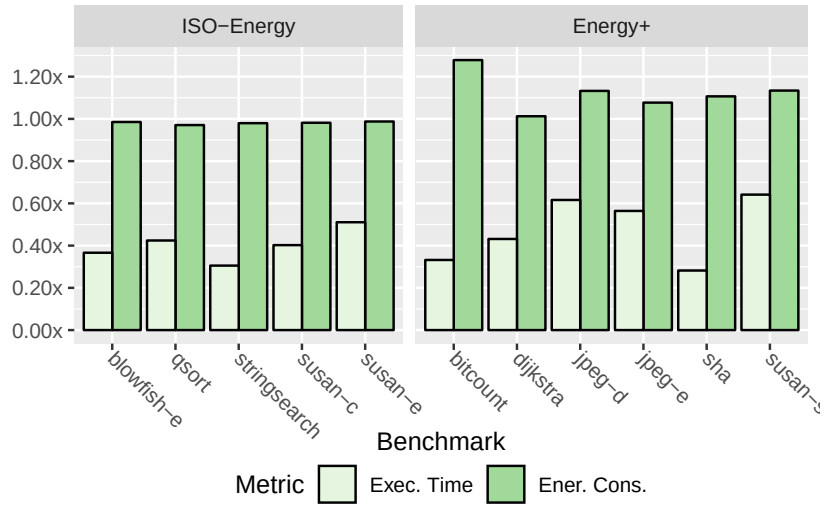


Source: the author.

Improving energy consumption in high-performance mode (Scenario IV). Fig. 5.15 show MuTARe's execution time and energy consumption when executing in the performance-constrained scenario just described. The execution time and energy consumption are normalized w.r.t. *big* operating in the highest frequency, i.e., w.r.t. Scenario II. In all but *susan-s* benchmark, MuTARe was able to meet the performance and save a significant amount of energy by exploiting ILP in a cost-effective way, i.e. generating instruction schedules only once and storing them into configuration for future use. *susan-s* presents a significant amount of multiplication operations (16%, as shown in Fig. 5.13), reducing the effective instruction window and ILP (only 10.5 operations per configuration, as shown in Table 5.9). Introducing a multiplier inside the reconfigurable unit may improve the performance in this benchmark, but will also introduce additional area and power overheads.

Improving performance in low-energy mode (Scenario III). Similarly to the previous analysis, Fig. 5.16 shows MuTARe's results in the energy constrained-scenario. The execution time and energy consumption are normalized w.r.t. *LITTLE* operating in the lowest frequency, i.e., w.r.t. Scenario I. Six of the applications are able to meet the energy budget, while five are not, introducing an energy overhead of less than 25%. Since power overheads introduced by the CGRA are proportionally larger for the *LITTLE* core than for the *big* one, maintaining the small energy budget is challenging in some appli-

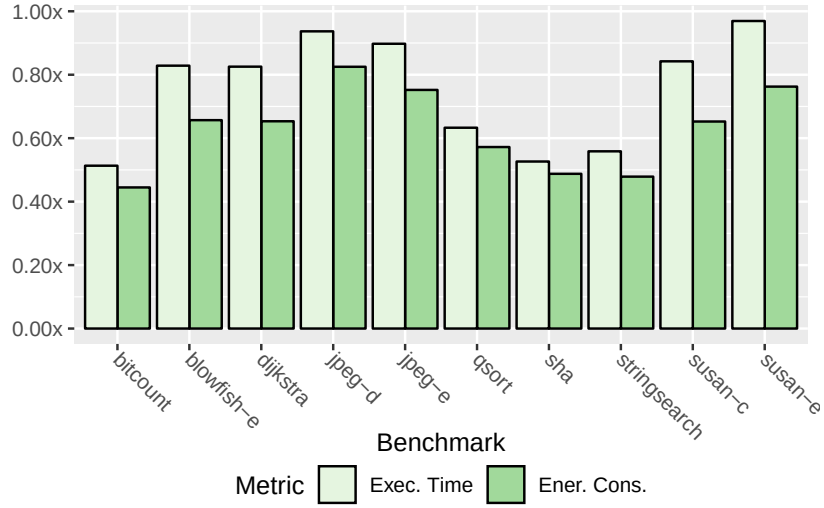
Figure 5.16: Results from e-constrained execution.



Source: the author.

cations, especially in the ones where the CGRA is significantly used. If energy is a tight constraint, however, the CGRA can be switched off automatically by determining when the ILP improvements provided by the CGRA are smaller than the power overheads introduced. In all execution cases, the performance-energy tradeoffs (EDP) improve, on average by $2.2\times$.

Figure 5.17: *High-performance* execution.



Source: the author.

Improving performance in high-performance mode (Scenario V). Finally, the results when executing MuTARe in high-performance mode are shown, again normalized to execution in *big* in the highest frequency (Scenario II). This is depicted in Fig. 5.17. Performance can be improved in all applications but *susan-s*. The best cases are highly-regular applications such as *bitcount* ($1.92\times$) and *sha* ($1.9\times$). In all cases, energy is

Figure 5.18: Performance and energy consumption under the effects of task migration.

Source: the author.

significantly reduced, and the EDP improves on average by $2.2\times$.

Results considering Task Migration. In heterogeneous systems, typically a scheduler migrates tasks between processing units according to the run-time requirements. In the previous analysis, we considered that the entire application runs in a single processing unit. Now, the performance-energy tradeoffs considering the effects of task migration is analyzed.

To show the role of ILP in an application's EDP, we rank all benchmarks based on the ILP improvement when switching from *LITTLE* to *big*, according to Table 5.9, and restrain the analysis to the worst case (*bitcount*), the median (*dijkstra*) and the best case (*susan-e*).

Fig. 5.18 shows performance and energy consumption in the Baseline (big.LITTLE) and MuTARe architectures depending on the fraction of the application that is scheduled for execution in *LITTLE* or *big*. Considering the baseline big.LITTLE system, results show that when the ILP improvements from *big* are small, as in the case of *bitcount*, energy consumption grows faster than speedup as execution is switched from the *LITTLE* to the *big* core. This phenomenon occurs when the application's data-dependencies prevent further ILP exploitation by the *big* core, which still introduces considerable power overheads. As a consequence, such an application will present better EDP when executing in the *LITTLE* core ($1.39\times$ w.r.t. *big*). On the other hand, *susan-e* presents the exact opposite behavior, with speedup increasing faster than energy consumption due to the availability of operations that can be executed concurrently. Therefore, EDP when executing in *big* is improved ($1.91\times$ w.r.t. *LITTLE*).

Considering now the MuTARe system, for all three benchmarks the speedup curves are above the baseline, and the energy consumption scales at a smaller rate as execution is switched from *LITTLE* to *big*. The reasons for that are the acceleration capabilities of the CGRA and the better power-efficiency compared to the *big* core, which overall enables better EDP than the baseline system in all operating points. In particular, the best EDP spot in MuTARe for *bitcount*, *dijkstra* and *susan-e* is found when the application executes entirely in the *big* core, providing an improvement of $3.14\times$, $1.86\times$ and $1.35\times$ (respectively) compared to the best case in the baseline system. This observation suggests that MuTARe can efficiently exploit an application's ILP without the overheads typically found in OoO cores.

Area costs. big.LITTLE-like systems often come in different arrangements, not only with different microarchitectures but also with a distinct number of cores (of each type) in the same System-on-Chip (SoC). For comparison, we use a typical scenario of four LITTLE cores and four big cores, as in Samsung's Exynos 7420 and Qualcomm's Snapdragon 810. Extending this system to MuTARe would require four additional MuTARe processing units, for a total area overhead of 32%. This overhead is smaller than 50%, which would be the cost of replacing the four *big* cores by (faster) *bigger* cores, and the choice for MuTARe presents better energy benefits with modest performance increases.

5.2.3 Scenario 3: Ultra Low-Power Computing for Emerging IoT Domains

Motivation. Battery-powered edge devices for IoT must present ultra-low energy consumption, competitive with application-specific designs, but at the same time be flexible enough to address the fast-evolving pace of IoT applications (BLAAUW et al., 2014). As these devices are now embracing the concept of *fog computing*, with a significant amount of processing being carried out locally rather than in the cloud (BONOMI et al., 2012; TAN et al., 2017), IoT processors now require a high range of adaptability: from ultra-low energy (when collecting) to high-performance (when processing data).

To balance between these requirements, current processors employ DVFS, adapting the V_{dd} f in the range between 100-70% of the nominal voltage to trade off between performance and energy/power. However, for future IoT workloads requiring extra performance under the same tight energy budgets, this range may not be wide enough. Because of this need, previous works have already shown that the optimum point for energy sav-

ings occurs for V_{dd} below the range traditionally achieved with DVFS, in the NTV range (KAUL et al., 2012; PINCKNEY et al., 2013).

An example from the experiments conducted in this thesis helps illustrate the idea. Consider an image processing pipeline (applying a filter, compressing the resulting image and finally encrypting it for secure transmission) which must execute in a very simple processor core with DVFS support, with a task deadline of 2 seconds. Table 5.11 supports the following discussion. When the system must process a small image (480p, nowadays' scenario), it can run at the lowest frequency and still meet the target deadline. However, in the near future, as image sizes are expected to increase (720p, future scenario), the task deadline is not met, even if the processor executes at the maximum operating frequency. Even if it did, the energy overhead would be $6.3\times$. While the provided example covers a single application domain (image processing), where input sizes are expected to increase as IoT becomes more widely adopted, the rationale applies to other domains as well. For example, routing algorithms (as the number of nodes in a sensor network increases), voice and object recognition (more patterns to identify) and many others.

Table 5.11: Execution times for an image processing pipeline in two scenarios.

Application	Exec. Time	
	480p (Low f)	720p (High f)
Smooth. Filter	1.765 s	2.432 s
Compression	187 ms	243 ms
Encryption	12 ms	12 ms
Sum	1.964 s ✓	2.687 s ✗

Source: the author.

An ideal solution to the variable performance demand problem should (1) provide energy-efficient acceleration capabilities and (2) do this for a broad range of applications. Superscalar OoO cores can exploit ILP, addressing the latter requirement, but consume significant power in doing so, which may be unacceptable in battery-powered devices. Dedicated accelerators meet the first requirement, but are not generic and are unsuitable for the fast-evolving pace of the IoT applications. Reconfigurable accelerators present an interesting alternative: they can create customized datapaths at run time, approaching the performance of a dedicated solution, and encode these for later reuse, saving significant power compared to OoO cores.

In this evaluation scenario, MuTARe is configured with a single-issue processor core extended with an energy-efficient CGRA for improving performance in emerging IoT workloads. The comparison is carried out with two adaptive processors: one energy-

efficient single-issue core, using DVFS for adaptability, and a 2-issue OoO core, also with DVFS. It will be shown that MuTARe in this setup is capable of achieving the same task deadline as IoT processors, but adapting to a lower energy budget and power.

Finally, this evaluation scenario also shows the benefits of NTV-MuTARe compared to the base MuTARe architecture. In NTV-MuTARe, all system components are scaled down to the lowest safe DVFS level (0.6V @ 0.90 GHz), and the CGRA is further scaled down to NTV levels (0.43V @ 0.45 GHz).

Evaluation setup. The evaluation methodology here is the same mentioned earlier in Section 5.2.2, with the following differences: while earlier MuTARe was configured with a set of heterogeneous cores and compared best performance and best energy consumption against this baseline, in here, the proposed system used the same CGRA but coupled to a LITTLE-like processor core for ultra-low-power domains. Still, a comparison shows what would be the benefits of using a big-like processor core instead of MuTARe. Table

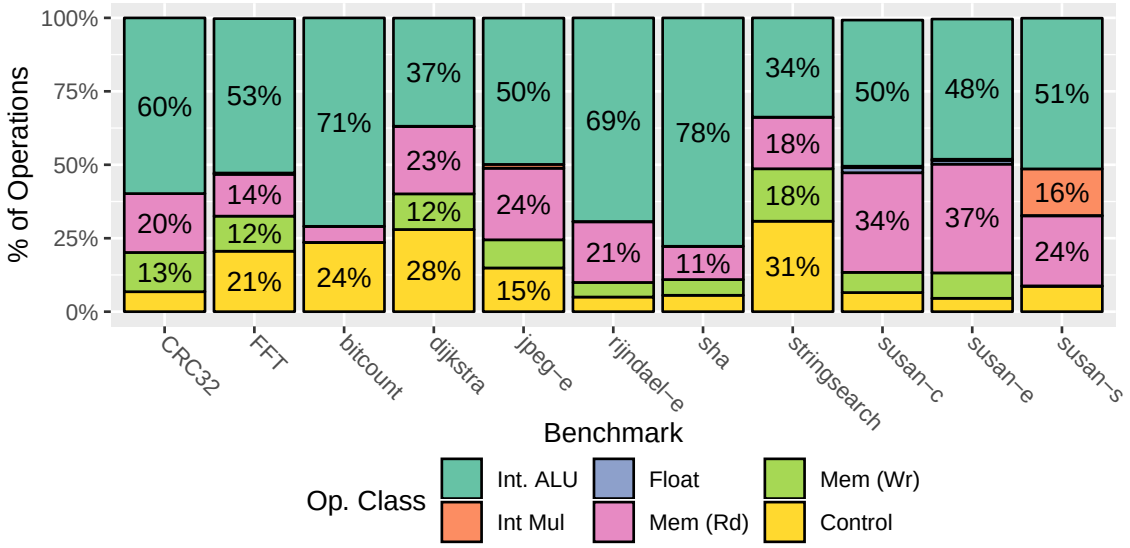
The benchmarks used here are similar, with three changes: AES encryption was included in the benchmark set for its relevance in nowadays' applications, as well as CRC32 and FFT (which includes floating point operations). Table 5.12 classifies these benchmarks into the IoT application groups referred in (ADEGBIJA et al., 2018), and Fig. 5.19 shows the operation breakdown for each benchmark.

First, the proposed and baseline system's behaviour across different DVFS OPs (i.e. voltage/frequency conditions) is investigated. Then the benchmarks' performance and energy consumption in each of the corner OPs (highest and lowest) are analyzed to show the limits of adaptability in each architecture. Additionally, a scenario with a set task deadline (defined as the performance achievable for each benchmark in the baseline core) is shown to compare how the performance improvements in the CGRA can be transformed into energy benefits when lowering the voltage. Finally, marginal improvements in the design metrics when further lowering the CGRA's OP to NTV levels are also shown.

Table 5.12: Benchmark groups.

Application Group	Benchmarks
Sensing	bitcount, stringsearch
Communication	dijkstra, FFT
Image Processing	susan (edges, corners, smoothing)
Data Compression	jpeg encode
Security	aes encrypt, sha
Fault Tolerance	CRC32

Figure 5.19: Benchmarks operation class mix.



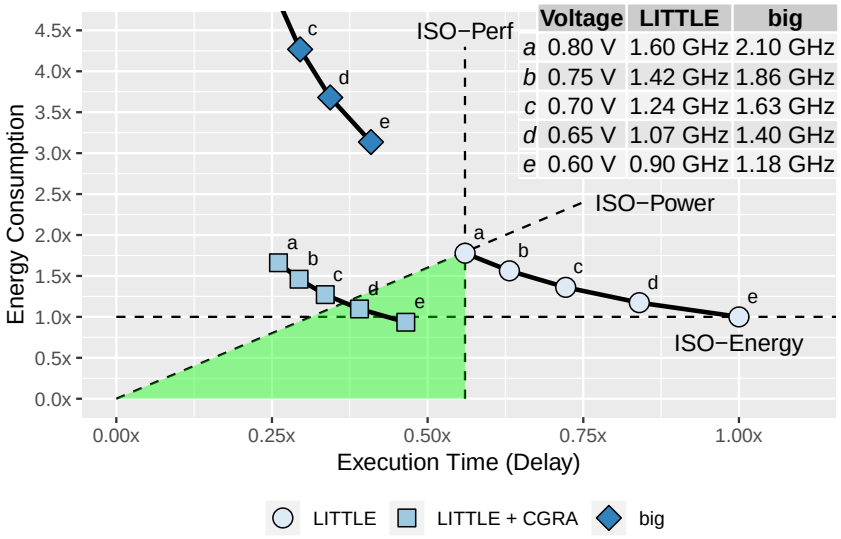
Source: the author.

Average Results (all benchmarks) in a single Operating Point. The comparison starts with the average performance-energy tradeoffs across all benchmarks in three systems: the baseline (*LITTLE*), the proposed one (*LITTLE* core extended with CGRA acceleration), and a future IoT processor (*big*). Fig. 5.20 presents the average results considering all applications, for the different systems and OPs, in the form of an EDP plot. In this plot, execution time is plotted against the x axis and energy consumption against the y axis, both normalized w.r.t. execution in the baseline system (LITTLE), in OP (0.6V, 0.90 GHz). Each curve shows a different system, with each point representing a different OP.

This plot allows comparison in performance (horizontal axis), energy (vertical axis), power (diagonal lines through the origin, where the energy/performance ratio is the same), and EDP (hyperbolas centered at the origin). The horizontal and vertical dashed lines show, respectively, the lowest energy consumption and the highest performance achievable when executing in the baseline system, and the diagonal line shows the power consumed when executing with the highest baseline performance. The shared (green) region in the plot denotes an interesting region, where all three metrics (performance, power and energy consumption) are improved w.r.t. the baseline core operating in the highest frequency (OP a).

The same plot shows that, when running in the same high frequency as the baseline, the proposed system can improve performance by $2.15\times$, while maintaining similar energy levels (7% reduction). In summary, the EDP curve is shifted to the left. Under this

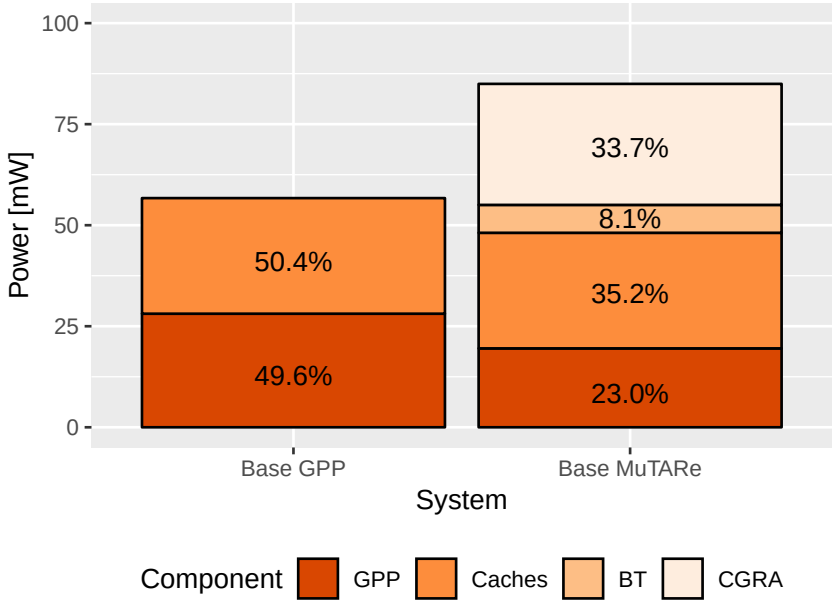
Figure 5.20: Energy-Delay tradeoffs and optimal operating region.



Source: the author.

condition, power is increased by $2\times$. However, the slack (performance improvements beyond the target task deadline) allows DVFS to be used to reduce the power and energy consumption while still meeting the deadline. Moving from OP (a) in the proposed system curve to OP (e) allows the target system to meet the task deadline still (with $1.20\times$ performance improvement, i.e., slack) and consuming 48% less energy, reducing overall power consumption by 37%.

Figure 5.21: Power breakdown with and without reconfigurable acceleration.



Source: the author.

Fig. 5.21 shows the average power consumed in the baseline and proposed systems

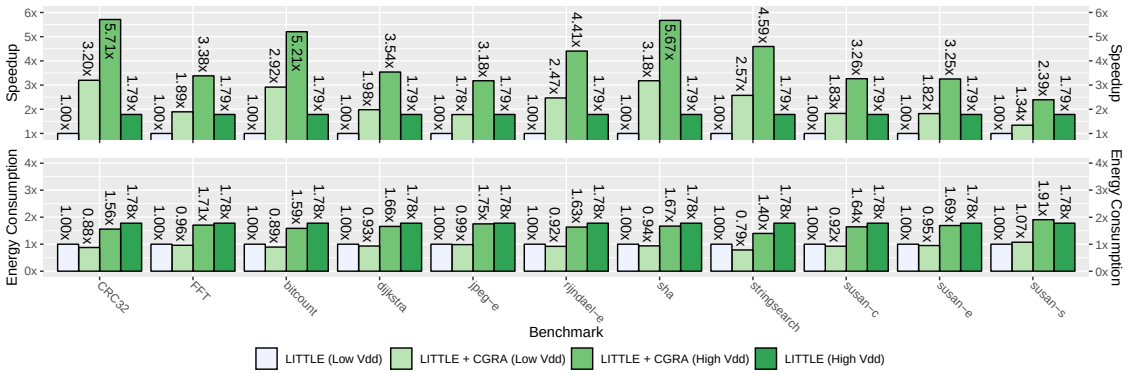
when running in the same OP (a), with a breakdown into the GPP core, the caches, the CGRA and the BT unit (the latter two costs are absent in the baseline system). The text labels show the percentual contribution of each component in the total system power.

When switching from the baseline to MuTARe, the GPP power consumption is reduced by nearly 30% because execution has been switched to the CGRA. This reduction is limited due to three factors:

- the coverage is limited (not all instructions execute in the CGRA - first execution and unsupported ops);
- even when executing in the CGRA, not all of the GPP components can be disabled - e.g., the register file;
- instructions executed in the CGRA are accelerated, which means the CGRA will spend less time executing those instructions than they would in the baseline core. As a consequence, the CGRA is active for less time than the baseline;

While the CGRA provides neat acceleration capabilities and reduces power in the GPP core, it also introduces a large overhead from the reconfigurable fabric and a small overhead from the binary translation unit, which runs only on traces executed in the GPP, leading to the power increased of $2.0\times$ previously mentioned. Since the size of each configuration is kept proportional to the size of the instructions allocated, the cache power stays roughly the same.

Figure 5.22: Speedup and energy consumption in a LITTLE core and in MuTARe.



Source: the author.

Per-Benchmark Results in a Single Operating Point. Fig. 5.22 shows the performance and energy consumption when running in the lowest and highest OP in each system. Again, results are normalized with respect to the execution in the baseline in the lowest OP. In all scenarios, performance is substantially increased compared to the *LITTLE* core in the same frequency condition.

To support this analysis, Table 5.13 provides detailed microarchitectural performance results (i.e., the ILP that each system can achieve, the coverage and the average configurations sizes).

The coverage column (fraction of total instructions executed in the CGRA) shows that the BT mechanism proposed in the work can match a high amount of instructions for CGRA execution, from 64.0% (in *FFT*) to 93.6 (in *sha*). The ILP exploited is also substantially higher than that achieved in the *LITTLE* core, and also better than the one in a *big* core, especially in highly-regular applications and those with data dependencies (e.g., *bitcount* and *sha*).

Table 5.13: Microarchitectural performance for each benchmark.

Benchmark	Cov	Avg / Cfg		IPC		
	[%]	[Ops]	[Blks]	CGRA	LITTLE	big
aes-e	87.4	21.0	0.89	2.01	0.65	1.43
bitcount	91.7	25.3	5.57	3.09	0.82	1.43
CRC32	92.5	27.5	1.99	2.32	0.60	1.07
dijkstra	84.9	15.0	4.03	1.34	0.41	1.03
FFT	64.0	17.3	3.53	2.49	0.65	1.43
jpeg-e	67.9	13.3	1.97	1.28	0.41	1.07
sha	93.6	29.9	1.67	2.83	0.60	1.36
stringsearch	83.2	14.3	4.52	1.63	0.36	0.76
susan-c	80.9	12.2	0.76	1.19	0.34	0.94
susan-e	71.4	12.2	0.63	1.10	0.33	1.04
susan-s	84.0	10.5	1.01	0.92	0.51	1.21

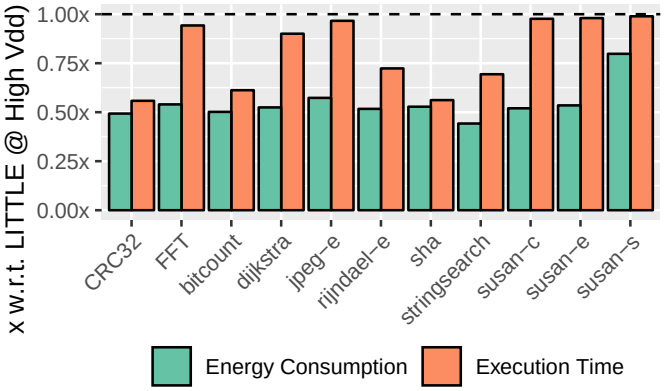
Source: the author.

Performance-Constrained Execution. Fig. 5.23 shows the performance and energy consumption of the proposed system, normalized to baseline in OP a, when a task deadline is present. The analysis assumes here that the task deadline is the execution time of the task in the baseline system at the highest frequency (i.e., the baseline system can meet the deadline just enough).

For many tasks, the CGRA accelerator provides such speedup that even lowering the operating voltage still provides a margin for improved performance. This is the case, for instance, in the *CRC32* and *sha* applications, where operating in the lowest voltage setting still allows for nearly $2\times$ performance improvements - a consequence of the higher ILP exploitation as shown in Table 5.13. The energy consumption, in all these cases, can also be significantly reduced.

Area costs. We estimate the reconfigurable unit, with caches and the BT unit to occupy 45,000micro m^2, an overhead of $3.75\times$ compared to the baseline single-issue

Figure 5.23: MuTARe's behavior in performance-constrained execution.

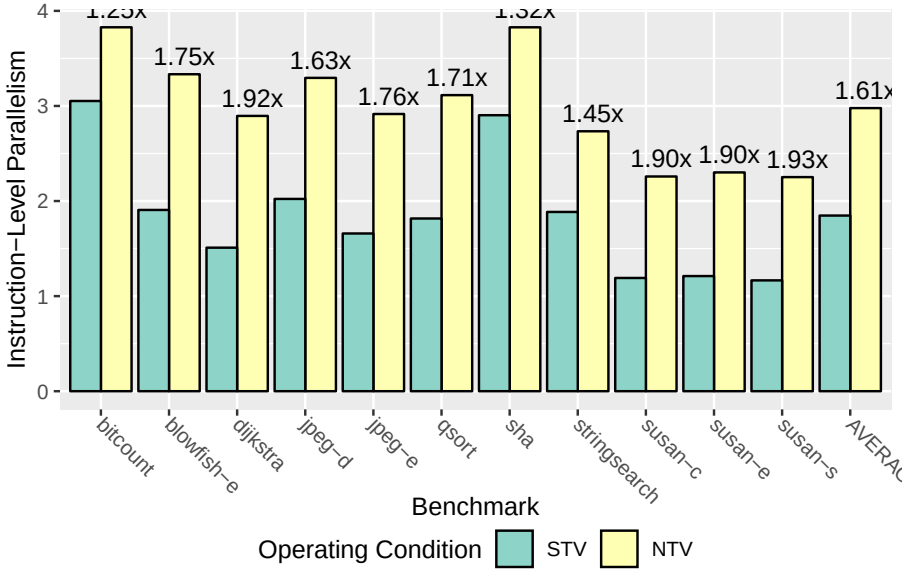


Source: the author.

core. However, this cost is still cheaper than switching to a more powerful core, such as the *big* core considered in this work, which would incur an overhead of more than $10\times$, as our synthesis results show.

NTV-MuTARe Results. In this condition, the power overheads introduced by the CGRA are reduced, following Eq. (1), since the frequency of the CGRA is lowered to half that of the remaining processor components. As a consequence, cache accesses can be executed in a single (CGRA) cycle, increasing the CGRA's ILP (at the cost of increased cycle latency). The goal is to try to recover the performance loss from the lower frequency with increased ILP.

Figure 5.24: Improvements in CGRA ILP when operating in NTV.

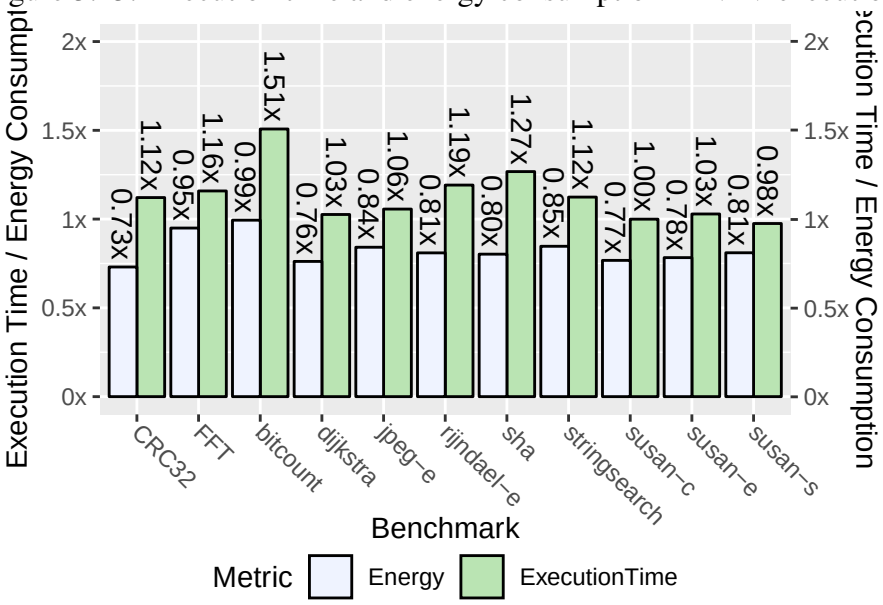


Source: the author.

Fig. 5.24 shows the results of this effect, which were illustrated earlier in Fig. 4.4. On the y-axis, for each benchmark, the average number of instructions that can be

executed in each (CGRA) cycle in the STV and NTV scenarios. The labels above the NTV axis show the ILP improvements. With a frequency reduction of $2.0\times$, the ILP improvements should be higher than $2.0\times$ in order to avoid performance losses. Although this level cannot be achieved, for all applications there is a significant improvement, averaging $1.61\times$ across all of them, suggesting that part of the performance loss from lower frequency can be recovered with higher ILP exploitation.

Figure 5.25: Execution time and energy consumption in NTV execution.



Source: the author.

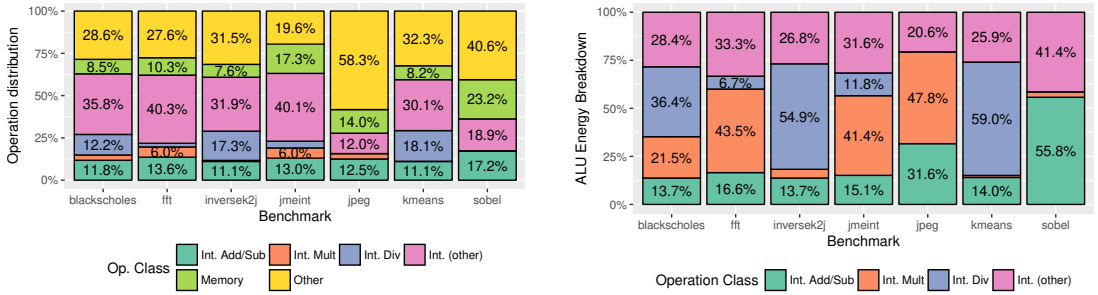
Finally, Fig. 5.25 compares the execution time and energy consumption in NTV-MuTARe against a baseline setup of MuTARe operating in the lowest DVFS level. In most cases, a performance overhead is still found, but smaller than the $2\times$ frequency penalty imposed by NTV operation. Since the CGRA is the only structure operating in NTV, the larger its utilization, the more significant the power and energy benefits. For instance, in *CRC32* and *sha*, which are two benchmarks with more than 90% coverage, NTV-MuTARe leads to 27% and 20% energy reduction, respectively. On the other hand, the energy benefits also depend on the ILP improvement. *bitcount* also presents over 90% coverage but, due to the increased execution time, presents less than 1% energy reduction.

5.2.4 Scenario 4: Approximate Computing for Error-Tolerant Domains

Motivation. As transistor scalability approaches the deep nano-era, power density issues have become of major concern. In the *dark silicon* era, the number of transistors

(i.e., area) is only a minor design constraint since a significant fraction of them must be switched off to avoid exacerbating power density and increasing chip temperature (ESMAEILZADEH et al., 2012; SHAFIQUE; GARG, 2017). Therefore, novel accelerator designs can be leveraged to exchange area for power, mitigating the inefficiencies of GPPs (HAMEED et al., 2011).

Figure 5.26: Breakdown of operations and ALU energy consumption into op classes.



(a) Number of operations.

(b) Energy consumption.

Source: the author.

CGRAs can improve the execution time and energy consumption of a wide range of applications by configuring customized datapaths at runtime using a single hardware. While this approach has been extensively investigated in the past (WIJTVLIET; WAEIJEN; CORPORAAL, 2016), recent research in approximate computing (SHAFIQUE et al., 2016), a paradigm enabling additional performance-area-energy improvements at an expense in quality, allows for new directions in power/energy-efficient design of CGRAs. Works on approximate functional units have demonstrated the potential of the imprecise adder, subtractor, multiplier and divider designs to improve performance, area, and power (GUPTA et al., 2011; KULKARNI; GUPTA; ERCEGOVAC, 2011; KAHNG; KANG, 2012; GUPTA et al., 2013; ALMURIB; KUMAR; LOMBARDI, 2016; HASHEMI; BAHAR; REDA, 2016). As shown in Fig. 5.26, although applications typically contain only about 25% of these operations, they are responsible, on average, for nearly 90% of the total ALU energy consumption, since the remaining operations (such as logic and shifts) are extremely low-power (we present the methodology for these experiments in a later section). However, only a few works (RAHA; JAYAKUMAR; RAGHUNATHAN, 2014; EL-HAROUNI et al., 2017; VENKATARAMANI et al., 2013; AKBARI et al., 2018) have targeted combining these individual designs to build accelerators, and none has yet combined them into a generic architectural solution such as a CGRA.

This comparison shows how Approx-MuTARe allows designers to trade area for improved power-efficiency in approximate applications. First, an extensive Design-Space

Exploration (DSE) of state-of-the-art approximate adders/subtractors, multipliers, and dividers is reported, with their area/power/error tradeoffs. Then, the design flow proposed earlier in Section 4.1.2 for selecting approximation modes to be implemented, compiling applications and managing reconfiguration under precise and approximate modes is evaluated.

Evaluation setup. Results are presented for the entire flow shown earlier in Fig. 4.2. All FUs have been synthesized to a 450 MHz clock frequency using Synopsys Design Compiler targeting UMC's 65 nm standard cell library.

1-bit FA implementations used to build designs 1-3 are taken from the 2011 IMPACT paper (GUPTA et al., 2011), for designs 4-7 from the 2013 IMPACT paper (GUPTA et al., 2013) and designs 8-10 from the work by Almurib et. al (ALMURIB; KUMAR; LOMBARDI, 2016). Multiplier designs 1-4 listed here are taken from the 2016 work by Shafique et al. (SHAFIQUE et al., 2016) and the design named *Lit* from the work by Kulkarni et al. (KULKARNI; GUPTA; ERCEGOVAC, 2011). The *accurate* divider design was configured for multiple accuracy modes following the approximation technique for division described in previous work (HASHEMI; BAHAR; REDA, 2016).

For quality estimation, both synthetic benchmarks that test all input combinations to each FU and also the benchmarks from AxBench (YAZDANBAKHSH et al., 2016) were used. The AxBench benchmarks were further modified for fixed-point execution. In the latter case, the analysis was restricted to replacing all non-critical instructions (i.e., all that flow directly into the application output and are not used for memory address or loop index computation) by a single approximation mode. For categorizing these critical and non-critical instructions, an approach similar to the one proposed by Rehman et al. was used (REHMAN et al., 2011).

Finally, the performance of Approx-MuTARe in this setup was simulated using gem5 and McPAT (LI et al., 2013) with the configuration shown in Table 5.14.

Table 5.14: Baseline processor parameters.

Pipeline: 8-wide OoO, with 4 ALU ports, 2 mult. ports, 2 load ports and 1 store port. Instr. queue: 60 micro ops. Load buffer: 72 micro ops. Store buffer: 42 micro ops. ROB entries: 192 micro ops. Memory dependence prediction via store sets.
L1 D+I caches: 32kB each, 8-way set associative, 2 cycles hit latency.
L2 cache: 256kB, 8-way associative, 8 cycles hit latency.
L3 cache: 2MB, 16-way associative, 18 cycles hit latency.

Source: the author.

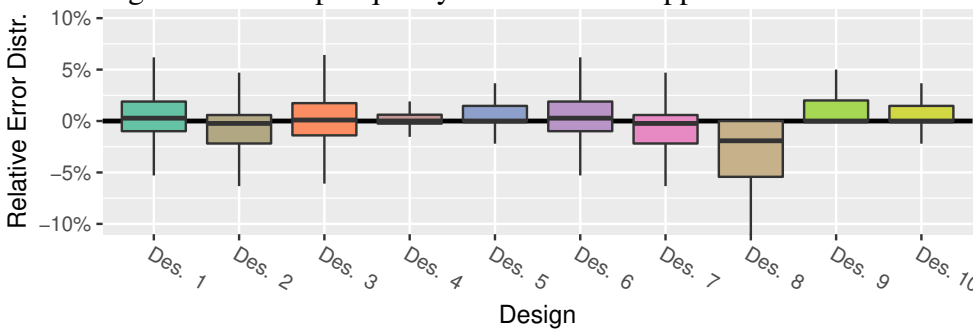
Design-Time Results. When designing approximate CGRA blocks, the designer

is presented with a library of approximate FUs described both in software (C++/Matlab) as well as hardware level (HDL). Each of these components provides distinct tradeoffs between delay, area, power, and quality; the goal of this stage is to select Pareto-optimal design points.

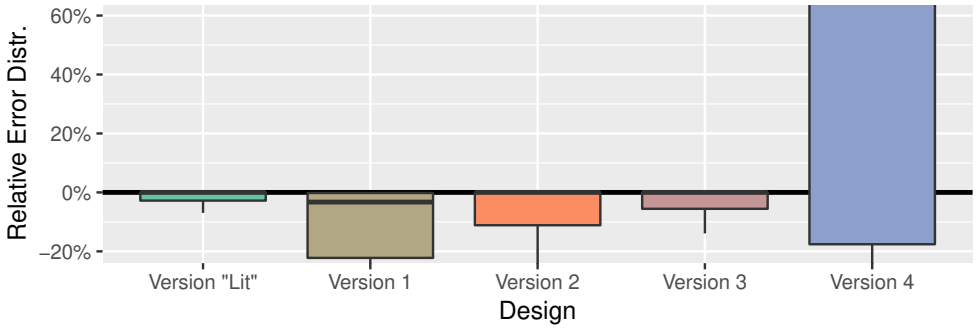
To get delay, area and power measurements, the designer must synthesize to the target technology available. For quality, two options are available:

- evaluate the output quality of each FU by testing all input combinations or using statistical models (such as (MAZAHIR et al., 2017b; MAZAHIR et al., 2017a));
- evaluate an application's output quality when replacing all accurate operations by their approximate counterparts.

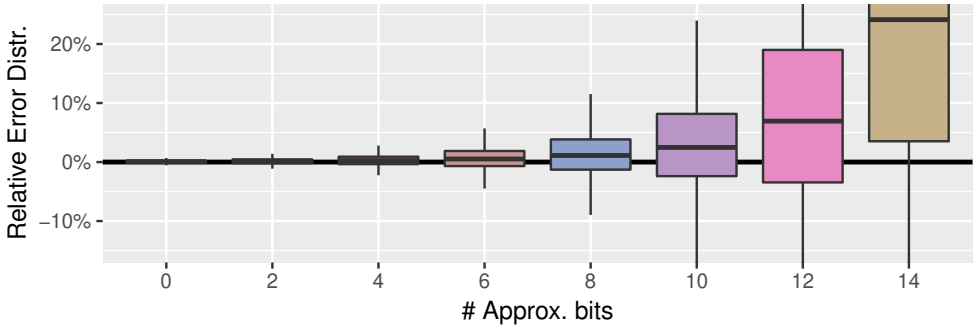
Figure 5.27: Output quality distribution of approximate FUs.



(a) Approximate 8-bit adders, with 4 LSBs computed approximately.



(b) 8x8 multipliers.

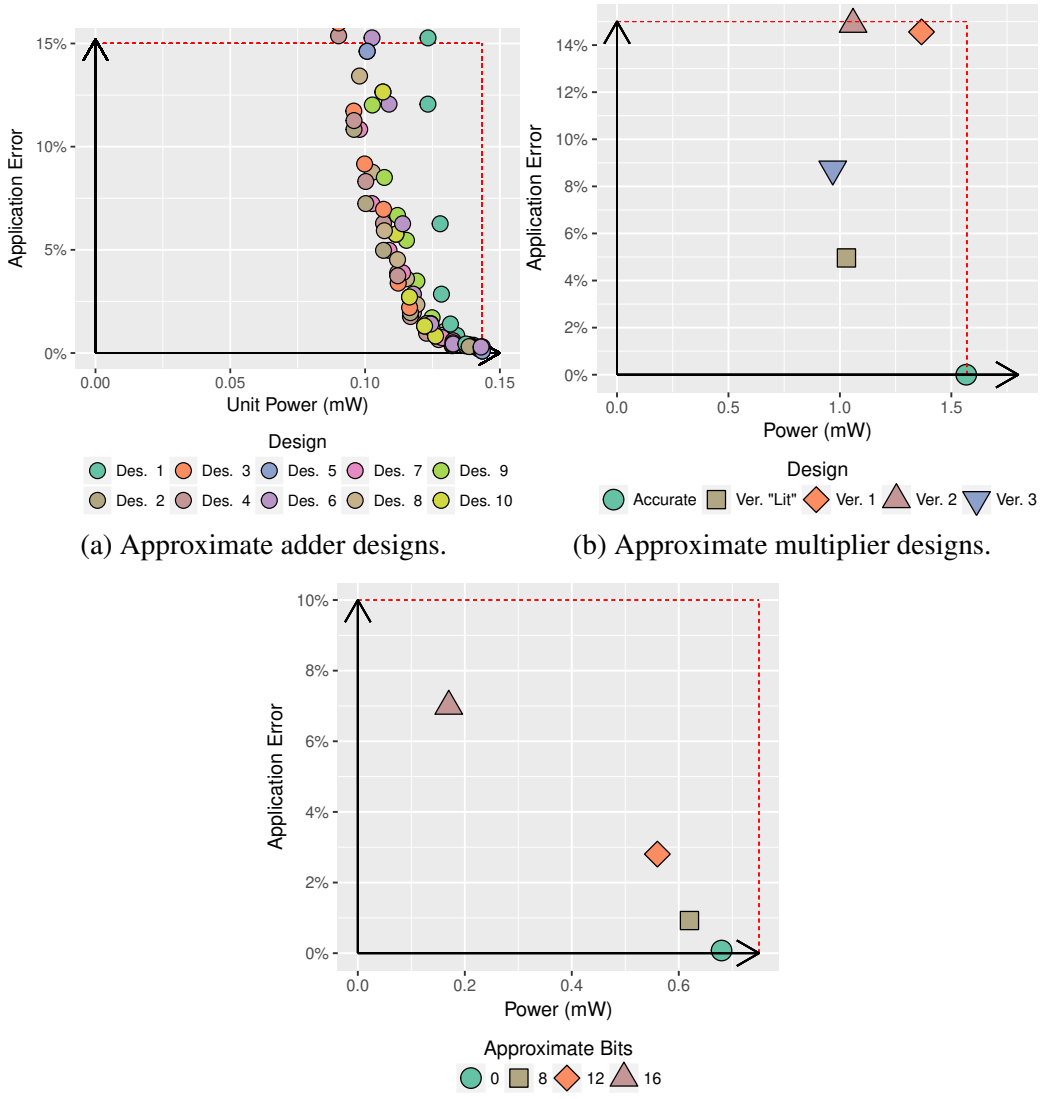


(c) 32x16 dividers.

Source: the author.

While the first option enables designers to do a faster design space exploration, it ignores application characteristics and how errors may propagate, accumulate or cancel each other out during execution. Fig. 5.27 shows the error distribution of each approximate FU design when tested under all possible input combinations. It shows that approximate FUs may have an error distribution biased towards positive or negative values, and this may impact applications differently. Therefore, these designs must be evaluated in the context of applications.

Figure 5.28: Design space of approximate multi-bit adders and multipliers.



(a) Approximate adder designs.

(b) Approximate multiplier designs.

(c) Approximate divider designs.

Source: the author.

Fig. 5.28, instead, shows the power-quality tradeoffs of approximate adders, multipliers, and dividers, now considering the average *application output quality* of all AxBench benchmarks and each FU's power. For each adder implementation in the figure, distinct levels of approximation (tuned by setting the number of approximation bits) are

tested.

To select *the designs with lowest power consumption* in the Pareto-Frontier that will be implemented in hardware, distinct quality loss constraints (e.g., $< 5\%$, $< 10\%$, ...) for each operation were set. Since adders are smaller than multipliers and dividers, four approximation modes for them and two for the remaining operations were selected. Table 5.15c shows the power/energy advantages of the approximate designs selected and Fig. 5.16 shows the logic function they implement.

Table 5.15: Approximate FUs modes selected in the DSE step.

(a) Approximate adder modes.

Design	Error	LSBs	Area (μm^2)	Power (mW)
Accurate	0%	-	1437	0.143
Design 2	$< 5\%$	8	1053 (-26.7%)	0.107 (-25.2%)
Design 3	$< 10\%$	9	992 (-31.0%)	0.100 (-30.0%)
Design 3	$< 15\%$	10	944 (-34.3%)	0.096 (-32.9%)

(b) Approximate multiplier modes.

Design	Error	Area (μm^2)	Power (mW)
Accurate	0%	13797	1.57
Version "Lit"	$< 5\%$	6552 (-52.5%)	1.02 (-35.0%)

(c) Approximate divider modes.

# Approx. bits	Error	Area (μm^2)	Power (mW)
0 (Accurate)	0%	18376	0.68
16	$< 10\%$	4036 (-78.0%)	0.17 (-75.0%)

Source: the author.

Compile-Time Results. In this stage, the application designer has a set of approximation modes available for which each operation may be replaced, along with power savings of using that mode and a final application accuracy constraint. The goal is to find the combination of approximation modes that yields the highest power savings while still meeting quality constraints. As mentioned in section 5.1, we restrict our analysis to replacing all operations by a single approximation mode. However, the CGRA can also support sequences of operations with alternating approximation modes.

Table 5.17 shows the approximation modes selected for each benchmark and the error they introduce. A quality constraint of maximum 10% error was enforced. The results show that most applications support approximate additions without any significant loss in quality. We found that *fft* and *inversek2j* present low tolerance to approximate

Table 5.16: Logic function implemented by the approximate FUs selected.

(a) Approximate Adders.

A	B	Cin	Accurate Sum	Accurate Cout	Design 2 Sum	Design 2 Cout	Design 3 Sum	Design 3 Cout
0	0	0	0	0	0	0	0	0
0	**0**	**1**	**1**	**0**	**1**	**0**	**0**	**0**
0	**1**	**0**	**1**	**0**	**0**	**0**	**1**	**0**
0	**1**	**1**	**0**	**1**	**1**	**0**	**1**	**0**
1	**0**	**0**	**1**	**0**	**0**	**1**	**0**	**1**
1	0	1	0	1	0	1	0	1
1	**1**	**0**	**0**	**1**	**0**	**1**	**1**	**1**
1	1	1	1	1	1	1	1	1

(b) Approximate Multiplier.

A	B	= Out	≈ Out
00	00	0000	0000
00	01	0000	0000
00	10	0000	0000
00	11	0000	0000
01	00	0000	0000
01	01	0001	0001
01	10	0010	0010
01	11	0011	0011
10	00	0000	0000
10	01	0010	0010
10	10	0100	0100
10	11	0110	0110
11	00	0000	0000
11	01	0011	0011
11	10	0110	0110
11	**11**	**1001**	**0101**

Source: the author.

Table 5.17: Selected approximation modes and corresponding error.

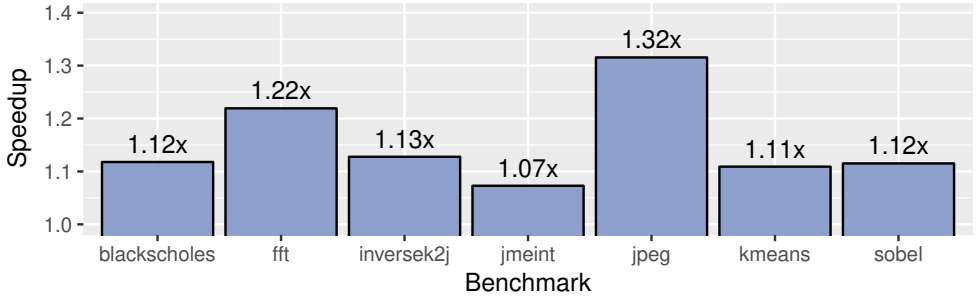
Benchmark	Add Cfg.	Mul Cfg.	Div Cfg.	Error (%)
blackscholes	Design 3 (10)	Accurate	Accurate	8.01
fft	Design 3 (10)	Accurate	Accurate	7.75
inversek2j	Design 3 (10)	Accurate	Accurate	8.26
jmeint	Design 2 (8)	Version "Lit"	Accurate	6.47
jpeg	Design 2 (8)	Version "Lit"	Accurate	5.01
kmeans	Design 3 (10)	Version "Lit"	Approx (16)	4.56
sobel	Design 3 (10)	Version "Lit"	Accurate	3.13

Source: the author.

division since the results of these operations are used to evaluate inverse trigonometric functions, which turned out to be very sensitive.

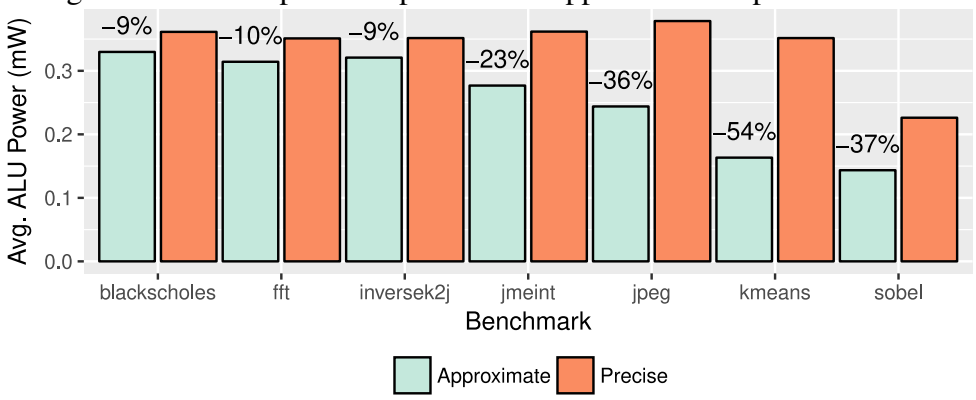
Run-Time Results. Fig. 5.30 shows the reductions in average ALU power when using the approximation modes selected, considering the operation distribution for each benchmark (presented in Fig 5.26) and the power savings of each approximation from Table 5.17. In two cases, this reduction can reach more than 50%. In some cases, the reduction achieved was below 10%; this is due to simplification in our analysis, which replaces all operations by a single approximation mode. In some cases, that replacement causes the entire application to exceed the quality constraint, preventing it from taking any benefit from approximate execution. Expanding the analysis to distinct approximation modes within the same application could provide valuable insight into the power savings potentials in these cases.

Figure 5.29: Application speedup when executing in the CGRA.



Source: the author.

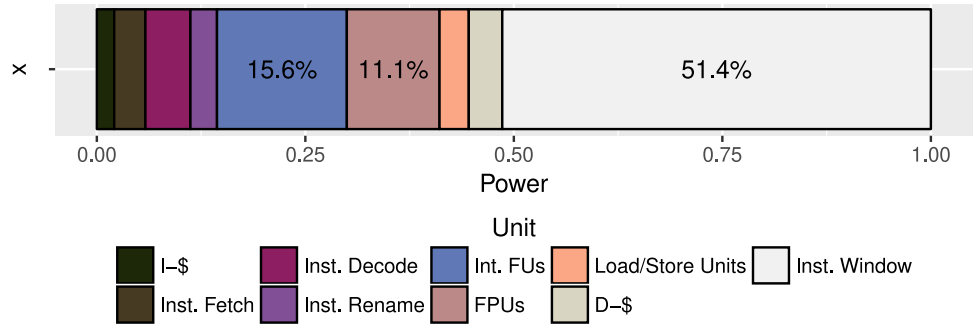
Figure 5.30: ALU power in precise and approximate implementations.



Source: the author.

We have also coupled the CGRA to an 8-issue OoO superscalar processor using the approach presented in (LIU et al., 2015) to investigate the potential for speedup and energy improvements. Fig. 5.31 shows the processor power distribution. Notice that, since it is a dynamically scheduled processor, a large amount of power is spent in control logic. Switching execution to the CGRA avoids most of these costs since each configuration already contains scheduled instructions; moreover, it increases the contribution of ALUs to the total processor energy, since the CGRA is essentially a matrix of FUs. Finally, execution in the CGRAs allows for significant energy improvements by reductions in the execution time, as shown in Fig. 5.29.

Figure 5.31: Power distribution in a superscalar processor.



Source: the author.

6 CONCLUSIONS

This work has presented MuTARe, the Multi-Target Adaptive Reconfigurable Architecture.

MuTARe was born as an extension to the DIM system, which consisted of a scalar MIPS core to which a combinatorial CGRA capable of accelerating data-dependent operations was coupled (BECK; RUTZIG; CARRO, 2014). The CGRA was designed with a simple 1-D interconnection structure that enabled greedily mapping instruction sequences executing in the scalar core to the Functional Units on-the-fly. By leveraging a dedicated hardware module to perform this mapping at run time, the overheads of dynamic Binary Translation were amortized and the hardware was able to maintain binary compatibility, enabling transparent acceleration of software already deployed.

The first step was to make the system competitive with high-end OoO processor cores, which revealed the first challenges: beating the performance levels of such cores and, most importantly, maintaining the precise exception behavior despite OoO execution. Since instruction sequences always execute for the first time in the GPP, before being mapped to the CGRA, it was vital that the first execution was fast to maintain a high-performance level. To that end, the system was extended to enable coupling to an OoO core. In this process, the mechanism described in Section 3.2.4 to communicate with the OoO core using a ROB-like structure was designed.

After that, attention in this work turned to heterogeneous systems, such as ARM's big.LITTLE, for mobile domains. In these systems, two processors optimized for different targets (low power vs high performance) are used to provide a better adaptation between application requirements, optimization target, and hardware. A reconfigurable architecture supporting the transparent acceleration of legacy code and capable of operating with a scalar or superscalar core was, in this context, a natural competitor. The proposed system was then extended for supporting coupling to a set of heterogeneous cores, and the performance, area, power, and energy tradeoffs were then investigated. Still, in this same context, the system was further extended to support DVFS, allowing a nearly-continuous adaptation knob to balance between the optimization targets (in contrast with the choice of heterogeneous core, which provided only a discrete knob). This extension enabled the proposed system to adapt to the same performance or energy levels as the baseline system (heterogeneous arrangement of GPP cores) while improving the other metric.

While this work was carried out, the emerging paradigm of *approximate comput-*

ing was flourishing. One question that emerged was then how to extend the proposed system with support to approximate computing. The natural solution was to used *approximate functional units* to that end, replacing the precise ones. To that end, however, modifications in the compiler toolchain had to be devised, considering that the accuracy of an application had to be specified by the programmer (and, therefore, transparent support for approximation in legacy code was unavailable). This led to the development of Approx-MuTARe, which has all of the advantages of MuTARe but can also provide additional power improvements in emerging error-tolerant workloads.

Finally, the last step in extending MuTARe was thought after contact with the *NTV computing* paradigm. MuTARe presented several features that made it a suitable structure for NTV computing, capable of addressing most of the challenges: the ability to extract significant amounts of ILP, the use of a combinatorial datapath (with larger voltage margins compared to sequential elements) and regularity (which simplifies variability management).

6.1 Future Work

This thesis has opened many opportunities for future work. In the scope of the base MuTARe architecture described in Chapter 3, coupling a RU to a set of heterogeneous cores raises interesting challenges. For instance, a strategy to find the optimal design of such an arrangement, given a set of applications, is still to be found. Considering the motivation for this design being that a significant fraction of applications still execute in the base GPP, an interesting research question concerns the optimal set of GPP cores, and the optimal design of the CGRA to enable the widest adaptability range (migrating from a high-performance, high-energy level to a low-performance, low-energy level) and/or most efficient adaptability range (high performance and low energy). The same motivational point also raises the issue of better selecting instruction sequences to map to the CGRA, considering that performance improvements depend on the balance between ILP improvements provided by the CGRA and the distribution of instructions streams executed in the GPP and the CGRA. Finally, considering that MuTARe is Multi-Target and can currently optimize for performance or energy consumption by coupling a Reconfigurable Unit to a performance-efficient or energy-efficient GPP core, another interesting research question is whether CGRA configurations can be optimized for performance or energy-efficiency. For instance, one may wonder whether it would be possible to tune

the BT algorithm at run time to generate configurations that are optimized either for low power or high performance.

In the context of Approx-MuTARe, the idea of using a reconfigurable hardware for approximate computations also raises interesting questions. The analysis in this work has restricted itself to *approximate Functional Units* with a single approximation mode. This greatly simplifies the process of mapping instruction sequences to the CGRA, since the hardware is set; on the other hand, it limits the choice of approximation strategies. In constrast, several works have proposed *approximate Functional Units* whose accuracy can be tuned at run time. While these units present an overhead in precise execution compared to non-approximate ones, one may wonder if these units wouldn't increase the efficiency of the mapping as it makes it more flexible and increases the range of options. The downside of this approach is also that more complex mapping techniques would need to be devised, as now not only the program is flexible (the accuracy of each operation can be selected) but also the hardware is flexible. Moreover, another issue that wasn't covered with approximate functional units in this thesis work was the latency advantages of this form of approximation.

In the context of NTV computing, several works can also be carried out. First of all, devising novel strategies for mitigating variability in regular combinatorial structures such as CGRAs. Then one may try to find optimal strategies to maximize the power consumption switch from the main processor to the CGRA, in order to increase the range that is optimized when computing in NTV. Finally, one may also try to find ways to improve the ILP exploitation to compensate for the frequency losses in NTV.

6.2 Publications and Presentations

6.2.1 Publications in the Scope of this Thesis

The following works have been published in peer-reviewed venues:

- **M. Brandalero**, M. Shafique, L. Carro, A. C. S. Beck. *"TransRec: Improving Adaptability in Single-ISA Heterogeneous Systems with Transparent and Reconfigurable Acceleration"*. Design, Automation & Test in Europe. Florence, 2019.
- **M. Brandalero**, L. Carro, A. C. S. Beck, M. Shafique. *"Approximate On-the-Fly Coarse-Grained Reconfigurable Acceleration for General-Purpose Applications"*.

Design Automation Conference. San Francisco, 2018.

- **M. Brandalero**, A. C. S. Beck. *"A Mechanism for Energy-efficient Reuse of Decoding and Scheduling of x86 Instruction Streams"*. Design, Automation & Test in Europe. Dresden, 2017.

The following works have been published in peer-reviewed journals:

- **M. Brandalero**, A. C. S. Beck. *"Potential analysis of a superscalar core employing a reconfigurable array for improving instruction-level parallelism"*. Design Automation For Embedded Systems, v. 20, p. 155-169, 2016. Springer US.

6.2.2 Publications as a Result from Collaborations

Besides the above mentioned publications, directly related to this thesis work, the following works were also published during the author's time as a Ph.D. student/candidate. These are the result of collaboration between other students in the group.

In peer-reviewed venues:

- **M. Brandalero**, G. M. Malfatti, G. F. Oliveira, L. R. Gonçalves, L. A. Silveira, B. C. da Silva, L. Carro and A. C. S. Beck. *"Efficient Local Memory Support for Approximate Computing"*. Brazilian Symposium on Computing Systems Engineering. Salvador, 2018.
- G. F. Oliveira, L. R. Gonçalves, **M. Brandalero**, A. C. S. Beck and L. Carro. *"Employing Classification-based Algorithms for General-Purpose Approximate Computing"*. Design Automation Conference. San Francisco, 2018.
- L. A. da Silveira, **M. Brandalero**, J. D. de Souza and A. C. S. Beck. *"The Potential of Accelerating Image-Processing Applications by Using Approximate Function Reuse"*. Brazilian Symposium on Computing Systems Engineering. Joao Pessoa, 2016.

And in peer-reviewed journals:

- **M. Brandalero**, L. A. da Silveira, J. D. Souza, A. C. S. Beck. *"Accelerating error-tolerant applications with approximate function reuse"*. Science of Computer Programming, v. 160, p. 54-67, 2017. Elsevier.

6.2.3 Presentations

The author has also presented this thesis work in two forums:

- **M. Brandalero**. *MuTARe: A Multi-Target Adaptive Reconfigurable Architecture*. Presented in the 2019 Design, Automation and Test & Europe's Ph.D. Forum. Florence, 2019.
- **M. Brandalero**. *Approximate On-The-Fly Coarse-Grained Reconfigurable Acceleration for General-Purpose Applications*. Presented in the 2019 Design Automation Conference Ph.D. Forum. San Francisco, 2018.

REFERENCES

ADEGBIJA, T. et al. Microprocessor optimizations for the internet of things: A survey. **IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems**, v. 37, n. 1, p. 7–20, jan 2018. ISSN 02780070. Disponível em: <http://ieeexplore.ieee.org/document/7954016/>.

AKBARI, O. et al. PX-CGRA: Polymorphic Approximate Coarse-Grained Reconfigurable Architecture. In: **IEEE/ACM Design Automation and Test in Europe Conference & Exhibition**. [S.l.: s.n.], 2018.

ALMURIB, H. A. F.; KUMAR, T. N.; LOMBARDI, F. Inexact designs for approximate low power addition by cell replacement. In: **2016 Design, Automation Test in Europe Conference Exhibition (DATE)**. Dresden, Germanydoi: [s.n.], 2016. p. 660–665. Disponível em: <http://ieeexplore.ieee.org/document/7459392/>.

ALTMAN, E.; KAELI, D.; SHEFFER, Y. Welcome to the opportunities of binary translation. **Computer**, IEEE, v. 33, n. 3, p. 40–45, mar 2000. ISSN 00189162. Disponível em: <http://ieeexplore.ieee.org/lpdocs/epic03/wrapper.htm?arnumber=825694>.

ALVAREZ, C.; CORBAL, J.; VALERO, M. Fuzzy Memoization for Floating-Point Multimedia Applications. **IEEE Transactions on Computers**, IEEE Computer Society, v. 54, n. 7, p. 922–927, jul 2005. ISSN 0018-9340. Disponível em: <http://dl.acm.org/citation.cfm?id=1070605.1070702>.

ARM. **big.LITTLE Technology: The Future of Mobile**. [S.l.], 2013. 1–12 p.

ASANOVIĆ, K. et al. **The Rocket Chip Generator**. [S.l.], 2016. 1–11 p. Disponível em: <http://www2.eecs.berkeley.edu/Pubs/TechRpts/2016/EECS-2016-17.html>.

BACHA, A.; TEODORESCU, R. Using ECC Feedback to Guide Voltage Speculation in Low-Voltage Processors. In: **2014 47th Annual IEEE/ACM International Symposium on Microarchitecture**. IEEE, 2014. p. 306–318. ISBN 978-1-4799-6998-2. Disponível em: <http://ieeexplore.ieee.org/document/7011397/>.

BALASUBRAMONIAN, R. et al. CACTI 7: New Tools for Interconnect Exploration in Innovative Off-Chip Memories. **ACM Transactions on Architecture and Code Optimization**, ACM, v. 14, n. 2, p. 1–25, jun 2017. ISSN 15443566. Disponível em: <http://dl.acm.org/citation.cfm?doid=3086564.3085572>.

BARAT, F.; LAUWEREINS, R. Reconfigurable instruction set processors: a survey. In: **Proceedings 11th International Workshop on Rapid System Prototyping.** IEEE Comput. Soc, 2000. p. 168–173. ISBN 0-7695-0668-2. Disponível em: <http://ieeexplore.ieee.org/document/855217/>.

BAUMGARTE, V. et al. PACT XPP—A Self-Reconfigurable Data Processing Architecture. **The Journal of Supercomputing**, Kluwer Academic Publishers, v. 26, n. 2, p. 167–184, 2003. ISSN 09208542. Disponível em: <http://link.springer.com/10.1023/A:1024499601571>.

BECK, A. C. S.; CARRO, L. **Dynamic Reconfigurable Architectures and Transparent Optimization Techniques**. Springer, 2010. Disponível em: <http://www.springer.com/engineering/circuits+{&}+systems/book/978-90-481-391>.

BECK, A. C. S.; RUTZIG, M. B.; CARRO, L. A transparent and adaptive reconfigurable system. **Microprocessors and Microsystems**, Elsevier Science Publishers B. V., v. 38, n. 5, p. 509–524, jul 2014. ISSN 01419331. Disponível em: <http://www.sciencedirect.com/science/article/pii/S0141933114000313http://dl.acm.org/citation.cfm?id=2644902.2644967>.

BECK, A. C. S. et al. Transparent Reconfigurable Acceleration for Heterogeneous Embedded Applications. In: **Proceedings of the conference on Design, automation and test in Europe - DATE '08**. New York, New York, USA: ACM Press, 2008. p. 1208–1213. ISBN 9783981080. ISSN 15301591. Disponível em: <http://dl.acm.org/citation.cfm?id=1403375.1403669>.

BINKERT, N. et al. The gem5 simulator. **ACM SIGARCH Computer Architecture News**, ACM, v. 39, n. 2, p. 1, aug 2011. ISSN 01635964. Disponível em: <http://dl.acm.org/citation.cfm?id=2024716.2024718>.

BLAAUW, D. et al. IoT design space challenges: Circuits and systems. In: **2014 Symposium on VLSI Technology (VLSI-Technology): Digest of Technical Papers**. IEEE, 2014. p. 1–2. ISBN 978-1-4799-3332-7. Disponível em: <http://ieeexplore.ieee.org/document/6894411/>.

BLAKE, G. et al. Evolution of thread-level parallelism in desktop applications. **ACM SIGARCH Computer Architecture News**, ACM, v. 38, n. 3, p. 302, jun 2010. ISSN 01635964. Disponível em: <http://dl.acm.org/citation.cfm?id=1816038.1816000>.

BOHR, M. A 30 Year Retrospective on Dennard's MOSFET Scaling Paper. **IEEE Solid-State Circuits Newsletter**, v. 12, n. 1, p. 11–13, jan 2007. ISSN 1098-4232. Disponível em: <http://ieeexplore.ieee.org/document/4785534/>.

BONOMI, F. et al. Fog Computing and its Role in the Internet of Things. In: **Proceedings of the first edition of the MCC workshop on Mobile cloud computing - MCC '12**. New York, New York, USA: ACM Press, 2012. p. 13. ISBN 9781450315197. ISSN 978-1-4503-1519-7. Disponível em: <http://dl.acm.org/citation.cfm?doid=2342509.2342513>.

BORKAR, S. Y. et al. **Platform 2015: Intel Processor and Platform Evolution for the Next Decade**. [S.l.], 2005.

BURD, T. et al. A dynamic voltage scaled microprocessor system. **IEEE Journal of Solid-State Circuits**, v. 35, n. 11, p. 1571–1580, nov 2000. ISSN 0018-9200. Disponível em: <http://ieeexplore.ieee.org/document/881202/>.

CALLAHAN, T.; HAUSER, J.; WAWRZYNEK, J. The Garp architecture and C compiler. **Computer**, v. 33, n. 4, p. 62–69, apr 2000. ISSN 00189162. Disponível em: <http://ieeexplore.ieee.org/document/839323/>.

CELIO, C.; PATTERSON, D. A.; ASANOVIĆ, K. **The Berkeley Out-of-Order Machine (BOOM): An Industry-Competitive, Synthesizable, Parameterized RISC-V Processor**. [S.l.], 2015. 1–5 p. Disponível em: <http://www2.eecs.berkeley.edu/Pubs/TechRpts/2015/EECS-2015-167.html>.

CHAKRADHAR, S.; RAGHUNATHAN, A. Best-effort Computing: Re-thinking Parallel Software and Hardware. In: **Design Automation Conference (DAC), 2010 47th ACM/IEEE**. [S.l.: s.n.], 2010. p. 865–870. ISSN 0738-100X.

CHAUDHURI, S. et al. Proving Programs Robust. In: **Proceedings of the 19th ACM SIGSOFT symposium and the 13th European conference on Foundations of software engineering - SIGSOFT/FSE '11**. New York, New York, USA: ACM Press, 2011. p. 102. ISBN 9781450304436. Disponível em: <http://dl.acm.org/citation.cfm?id=2025113.2025131>.

CLARK, N. et al. Application-Specific Processing on a General-Purpose Core via Transparent Instruction Set Customization. In: **37th International Symposium on Microarchitecture (MICRO-37'04)**. IEEE, 2004. p. 30–40. ISBN 0-7695-2126-6. ISSN 1072-4451. Disponível em: <http://ieeexplore.ieee.org/lpdocs/epic03/wrapper.htm?arnumber=1550980>.

COMPTON, K.; HAUCK, S. Reconfigurable computing: a survey of systems and software. **ACM Computing Surveys**, ACM, v. 34, n. 2, p. 171–210, jun 2002. ISSN 03600300. Disponível em: <http://dl.acm.org/citation.cfm?id=508352.508353>.

COTA, E. G. et al. An Analysis of Accelerator Coupling in Heterogeneous Architectures. In: **Proceedings of the 52nd Annual Design Automation Conference on - DAC '15**. ACM, 2015. p. 202. ISBN 978-1-4503-3520-1. Disponível em: <http://dl.acm.org/citation.cfm?id=2744769.2744794>.

CRONQUIST, D. et al. Specifying and compiling applications for RaPiD. In: **Proceedings. IEEE Symposium on FPGAs for Custom Computing Machines (Cat. No.98TB100251)**. IEEE Comput. Soc, 1998. p. 116–125. ISBN 0-8186-8900-5. Disponível em: <http://ieeexplore.ieee.org/document/707889/>.

DANOWITZ, A. et al. CPU DB: Recording Microprocessor History. **Communications of the ACM**, ACM, v. 55, n. 4, p. 55, apr 2012. ISSN 00010782. Disponível em: <http://dl.acm.org/citation.cfm?doid=2133806.2133822>.

DENNARD, R. et al. Design of ion-implanted MOSFET's with very small physical dimensions. **IEEE Journal of Solid-State Circuits**, v. 9, n. 5, p. 256–268, oct 1974. ISSN 0018-9200. Disponível em: <http://ieeexplore.ieee.org/document/1050511/>.

DIXON, M. et al. The Next Generation Intel® Core™ Microarchitecture. **Intel Technology Journal**, v. 14, n. 3, p. 8–28, 2010.

DRESLINSKI, R. G. et al. Near-Threshold Computing: Reclaiming Moore's Law Through Energy Efficient Integrated Circuits. **Proceedings of the IEEE**, v. 98, n. 2, p. 253–266, feb 2010. ISSN 0018-9219. Disponível em: <http://ieeexplore.ieee.org/document/5395763/>.

EL-HAROUNI, W. et al. Embracing approximate computing for energy-efficient motion estimation in high efficiency video coding. In: **Design, Automation & Test in Europe Conference & Exhibition (DATE), 2017**. IEEE, 2017. p. 1384–1389. ISBN 978-3-9815370-8-6. Disponível em: <http://ieeexplore.ieee.org/document/7927209/>.

ESMAEILZADEH, H. et al. Dark Silicon and the End of Multicore Scaling. **IEEE Micro**, v. 32, n. 3, p. 122–134, may 2012. ISSN 0272-1732. Disponível em: <http://ieeexplore.ieee.org/document/6175879/>.

ESMAEILZADEH, H. et al. Neural Acceleration for General-Purpose Approximate Programs. **Communications of the ACM**, ACM, v. 58, n. 1, p. 105–115, dec 2014. ISSN 00010782. Disponível em: <http://dl.acm.org/ft{_}gateway.cfm?id=2589750{&}type=htmlhttp://ieeexplore.ieee.org/document/64>.

(ESR), E. S. o. R. Usability of irreversible image compression in radiological imaging. A position paper by the European Society of Radiology (ESR). **Insights into Imaging**, Springer Berlin Heidelberg, v. 2, n. 2, p. 103–115, apr 2011. ISSN 1869-4101. Disponível em: <http://link.springer.com/10.1007/s13244-011-0071-x>.

EXYNOS 7420 Product Specification. 2015. Disponível em: <https://www.samsung.com/semiconductor/minisite/exynos/products/mobileprocessor/exynos-7-octa-7420/>.

FOLEGNANI, D.; GONZALEZ, A. Energy-effective issue logic. In: **International Symposium on Computer Architecture (ISCA)**. IEEE Comput. Soc, 2001. p. 230–239. ISBN 0-7695-1162-7. ISSN 1063-6897. Disponível em: <http://ieeexplore.ieee.org/lpdocs/epic03/wrapper.htm?arnumber=937452>.

GOLDSTEIN, S. et al. PipeRench: a reconfigurable architecture and compiler. **Computer**, v. 33, n. 4, p. 70–77, apr 2000. ISSN 00189162. Disponível em: <http://ieeexplore.ieee.org/document/839324/http://ieeexplore.ieee.org/lpdocs/epic03/wrapper.htm?arnumber=839324>.

GONZÁLEZ, A. et al. Trace-level reuse. In: **Proceedings of the 1999 International Conference on Parallel Processing**. IEEE Comput. Soc, 1999. p. 30–37. ISBN 0-7695-0350-0. ISSN 0190-3918. Disponível em: <http://ieeexplore.ieee.org/lpdocs/epic03/wrapper.htm?arnumber=797385>.

GOPIREDDY, B. et al. ScalCore: Designing a core for voltage scalability. In: **2016 IEEE International Symposium on High Performance Computer Architecture (HPCA)**. IEEE, 2016. p. 681–693. ISBN 978-1-4673-9211-2. Disponível em: <http://ieeexplore.ieee.org/document/7446104/>.

GOVINDARAJU, V. et al. DySER: Unifying Functionality and Parallelism Specialization for Energy-Efficient Computing. **IEEE Micro**, IEEE Computer Society, v. 32, n. 5, p. 38–51, sep 2012. ISSN 0272-1732. Disponível em: <http://ieeexplore.ieee.org/document/6235947/http://www.computer.org/csdl/mags/mi/2012/05/mmi2012050038.html>.

GUO, X. et al. Back to the Future: Digital Circuit Design in the FinFET Era. **Journal of Low Power Electronics**, v. 13, n. 3, p. 338–355, sep 2017. ISSN 1546-1998. Disponível em: <http://www.ingentaconnect.com/content/10.1166/jolpe.2017.1489>.

GUPTA, V. et al. Low-Power Digital Signal Processing Using Approximate Adders. **IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems**, v. 32, n. 1, p. 124–137, jan 2013. ISSN 0278-0070. Disponível em: <http://ieeexplore.ieee.org/document/6387646/>.

GUPTA, V. et al. IMPACT: IMPrecise adders for low-power approximate computing. In: **IEEE/ACM International Symposium on Low Power Electronics and Design**. IEEE, 2011. p. 409–414. ISBN 978-1-61284-658-3. Disponível em: <http://ieeexplore.ieee.org/document/5993675/>.

GUTHAUS, M. et al. MiBench: A free, commercially representative embedded benchmark suite. In: **Proceedings of the Fourth Annual IEEE International Workshop on Workload Characterization.** IEEE, 2001. p. 3–14. ISBN 0-7803-7315-4. Disponível em: <http://ieeexplore.ieee.org/lpdocs/epic03/wrapper.htm?arnumber=990739>.

GWENNAP, L. Sandy Bridge Spans Generations. **Microprocessor Report**, n. 328, p. 8, 2010.

HAMEED, R. et al. Understanding sources of ineffciency in general-purpose chips. **Communications of the ACM**, ACM, v. 54, n. 10, p. 85, oct 2011. ISSN 00010782. Disponível em: <http://dl.acm.org/ft{_}gateway.cfm?id=2001291{&}typ>.

HAMMARLUND, P. et al. Haswell: The Fourth-Generation Intel Core Processor. **IEEE Micro**, IEEE Computer Society, v. 34, n. 2, p. 6–20, mar 2014. ISSN 0272-1732. Disponível em: <http://www.computer.org/csdl/mags/mi/2014/02/mmi2014020006.html>.

HASHEMI, S.; BAHAR, R. I.; REDA, S. A low-power dynamic divider for approximate applications. In: **Proceedings of the 53rd Annual Design Automation Conference on - DAC '16**. New York, New York, USA: ACM Press, 2016. p. 1–6. ISBN 9781450342360. Disponível em: <http://dl.acm.org/citation.cfm?doid=2897937.2897965>.

HENKEL, J. et al. New trends in dark silicon. In: **Proceedings of the 52nd Annual Design Automation Conference on - DAC '15**. New York, New York, USA: ACM Press, 2015. p. 1–6. ISBN 9781450335201. Disponível em: <http://dl.acm.org/citation.cfm?doid=2744769.2747938>.

HERVEILLE, R. **Hardware Division Unit**. 2011. Disponível em: <https://opencores.org/project,divider>.

HINTON, G. et al. The Microarchitecture of the Pentium 4 Processor. **Intel Technology Journal**, v. 1, n. 1, p. 1–13, 2001. Disponível em: <http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.14.2991>.

HIRKI, M. et al. Empirical Study of the Power Consumption of the x86-64 Instruction Decoder. In: **USENIX Workshop on Cool Topics on Sustainable Data Centers (CoolDC 16)**. [s.n.], 2016. Disponível em: <https://www.usenix.org/conference/cooldc16/workshop-program/presentation/hirki>.

HUANG, J.; LILJA, D. D. J. Exploiting basic block value locality with block reuse. In: **High-Performance Computer Architecture, 1999. Proceedings. Fifth International Symposium On**. IEEE, 1999. p. 106–114. ISBN 0-7695-0004-8. Disponível em: <http://ieeexplore.ieee.org/lpdocs/epic03/wrapper.htm?arnumber=744342>.

HWU, W.-m.; PATEL, S. Accelerator Architectures —A Ten-Year Retrospective. **IEEE Micro**, v. 38, n. 6, p. 56–62, nov 2018. ISSN 0272-1732. Disponível em: <https://ieeexplore.ieee.org/document/8585394/>.

INTEL. **Intel Architecture Optimization Manual**. 1997. Disponível em: <http://www.intel.com/design/pentium/MANUALS/24281603.pdf>.

(IRDS), I. R. f. D.; SYSTEMS. **IRDS 2017: More Moore**. [S.l.], 2017. Disponível em: <https://irds.ieee.org/roadmap-2017>.

ISCI, C.; MARTONOSI, M. Runtime Power Monitoring in High-End Processors: Methodology and Empirical Data. In: **Proceedings of the 36th International Symposium on Microarchitecture (MICRO-36'03)**. IEEE Computer Society, 2003. p. 93. ISBN 0-7695-2043-X. Disponível em: <http://dl.acm.org/citation.cfm?id=956417.956567>.

KAHNG, A. B.; KANG, S. Accuracy-configurable adder for approximate arithmetic designs. In: **Proceedings of the 49th Annual Design Automation Conference on - DAC '12**. New York, New York, USA: ACM Press, 2012. p. 820. ISBN 9781450311991. Disponível em: <http://dl.acm.org/citation.cfm?doid=2228360.2228509>.

KARPUZCU, U. R. et al. VARIUS-NTV: A microarchitectural model to capture the increased sensitivity of manycores to process variations at near-threshold voltages. In: **IEEE/IFIP International Conference on Dependable Systems and Networks (DSN 2012)**. IEEE, 2012. p. 1–11. ISBN 978-1-4673-1625-5. Disponível em: <http://ieeexplore.ieee.org/document/6263951/>.

KARPUZCU, U. R. et al. EnergySmart: Toward energy-efficient manycores for Near-Threshold Computing. In: **2013 IEEE 19th International Symposium on High Performance Computer Architecture (HPCA)**. IEEE, 2013. p. 542–553. ISBN 978-1-4673-5587-2. Disponível em: <http://ieeexplore.ieee.org/document/6522348/>.

KASTRUP, B.; BINK, A.; HOOGERBRUGGE, J. ConCISe: a compiler-driven CPLD-based instruction set accelerator. In: **Seventh Annual IEEE Symposium on Field-Programmable Custom Computing Machines (Cat. No.PR00375)**. IEEE Comput. Soc, 1999. p. 92–101. ISBN 0-7695-0375-6. Disponível em: <http://ieeexplore.ieee.org/document/803671/>.

KAUL, H. et al. Near-threshold voltage (NTV) design - Opportunities and Challenges. In: **Proceedings of the 49th Annual Design Automation Conference on - DAC '12**. New York, New York, USA: ACM Press, 2012. p. 1153. ISBN 9781450311991. Disponível em: <http://dl.acm.org/citation.cfm?doid=2228360.2228572>.

KERAMIDAS, G.; KOKKALA, C.; STAMOULIS, I. Clumsy Value Cache: An Approximate Memoization Technique for Mobile GPU Fragment Shaders. In: **1st Workshop On Approximate Computing (WAPCO 2015)**. Amsterdam: [s.n.], 2015. p. 6. Disponível em: <http://wapco.inf.uth.gr/2015/program.html>.

KHARE, S.; JAIN, S. Prospects of Near-Threshold Voltage Design for Green Computing. In: **2013 26th International Conference on VLSI Design and 2013 12th International Conference on Embedded Systems**. IEEE, 2013. p. 120–124. ISBN 978-1-4673-4639-9. Disponível em: <http://ieeexplore.ieee.org/document/6472625/>.

KIAMEHR, S. et al. Temperature-Aware Dynamic Voltage Scaling to Improve Energy Efficiency of Near-Threshold Computing. **IEEE Transactions on Very Large Scale**

Integration (VLSI) Systems, v. 25, n. 7, p. 2017–2026, jul 2017. ISSN 1063-8210. Disponível em: <http://ieeexplore.ieee.org/document/7875441/>.

KULKARNI, P.; GUPTA, P.; ERCEGOVAC, M. Trading Accuracy for Power with an Underdesigned Multiplier Architecture. In: **2011 24th Internatioal Conference on VLSI Design**. IEEE, 2011. p. 346–351. ISBN 978-1-61284-327-8. Disponível em: <http://ieeexplore.ieee.org/document/5718826/>.

LI, S. et al. The McPAT Framework for Multicore and Manycore Architectures. **ACM Transactions on Architecture and Code Optimization**, ACM, v. 10, n. 1, p. 1–29, apr 2013. ISSN 15443566. Disponível em: <http://dl.acm.org/citation.cfm?id=2445572.2445577>.

LIU, F. et al. DynaSpAM : Dynamic Spatial Architecture Mapping using Out of Order Instruction Schedules. In: **Proceedings of the ACM/IEEE International Symposium on Computer Architecture**. [S.l.: s.n.], 2015. p. 541–553. ISBN 9781450334020.

LYSECKY, R.; STITT, G.; VAHID, F. Warp Processors. **ACM Transactions on Design Automation of Electronic Systems**, ACM, v. 11, n. 3, p. 659–681, jul 2006. ISSN 10844309. Disponível em: <http://dl.acm.org/citation.cfm?id=1142980.1142986>.

MARKOVIC, D. et al. Ultralow-Power Design in Near-Threshold Region. **Proceedings of the IEEE**, v. 98, n. 2, p. 237–252, feb 2010. ISSN 0018-9219. Disponível em: <http://ieeexplore.ieee.org/document/5395771/>.

MARTINS, M. et al. Open Cell Library in 15nm FreePDK Technology. In: **Proceedings of the 2015 Symposium on International Symposium on Physical Design - ISPD '15**. New York, New York, USA: ACM Press, 2015. p. 171–178. ISBN 9781450333993. Disponível em: <http://dl.acm.org/citation.cfm?doid=2717764.2717783>.

MAZAHIR, S. et al. Probabilistic Error Analysis of Approximate Recursive Multipliers. **IEEE Transactions on Computers**, p. 1–1, 2017. ISSN 0018-9340. Disponível em: <http://ieeexplore.ieee.org/document/7935435/>.

MAZAHIR, S. et al. Probabilistic Error Modeling for Approximate Adders. **IEEE Transactions on Computers**, v. 66, n. 3, p. 515–530, mar 2017. ISSN 0018-9340. Disponível em: <http://ieeexplore.ieee.org/document/7558229/>.

MEI, B. et al. ADRES: An architecture with tightly coupled VLIW processor and coarse-grained reconfigurable matrix. In: **13th International Conference on Field-Programmable Logic and Applications**. [S.l.]: Springer International Publishing, 2003. p. 61–70.

MICHIE, D. Memo Functions and Machine Learning. **Nature**, v. 218, n. 5136, p. 19–22, 1968.

MIGUEL, J. S. et al. The Bunker Cache for Spatio-Value Approximation. In: **2016 49th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)**. [S.l.: s.n.], 2016. p. 1–12.

MIGUEL, J. S. et al. Doppelgänger : A Cache for Approximate Computing. In: **Proceedings of the 48th Annual IEEE/ACM International Symposium on**

Microarchitecture (MICRO-48). [S.l.: s.n.], 2015. ISBN 9781450340342. ISSN 10724451.

MISAILOVIC, S. et al. Chisel: Reliability- and Accuracy-Aware Optimization of Approximate Computational Kernels. In: **Proceedings of the 2014 ACM International Conference on Object Oriented Programming Systems Languages & Applications - OOPSLA '14**. New York, New York, USA: ACM Press, 2014. v. 49, n. 10, p. 309–328. ISBN 9781450325851. ISSN 0362-1340. Disponível em: <http://dl.acm.org/citation.cfm?doid=2660193.2660231>.

MISHRA, A. K.; BARIK, R.; PAUL, S. iACT: A software-hardware framework for understanding the scope of approximate computing. In: **Workshop on Approximate Computing Across the System Stack (WACAS)**. [S.l.: s.n.], 2014.

MITTAL, S. A Survey of Architectural Techniques for Near-Threshold Computing. **ACM Journal on Emerging Technologies in Computing Systems**, ACM, v. 12, n. 4, p. 1–26, dec 2015. ISSN 15504832. Disponível em: <http://dl.acm.org/citation.cfm?doid=2856147.2821510>.

MITTAL, S. A Survey of Techniques for Approximate Computing. **ACM Computing Surveys**, ACM, v. 48, n. 4, p. 1–33, mar 2016. ISSN 03600300. Disponível em: <http://dl.acm.org/citation.cfm?doid=2891449.2893356>.

MIYAMORI, T.; MIYAMORI, T.; OLUKOTUN, K. REMARC: Reconfigurable Multimedia Array Coprocessor. **IEICE TRANSACTIONS ON INFORMATION AND SYSTEMS E82-D**, v. 82, p. 389—-397, 1998. Disponível em: <http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.56.607>.

MOHAPATRA, D. et al. Design of voltage-scalable meta-functions for approximate computing. In: **2011 Design, Automation & Test in Europe**. IEEE, 2011. p. 1–6. ISBN 978-3-9810801-8-6. Disponível em: <http://ieeexplore.ieee.org/document/5763154/>.

MOHAPATRA, D.; KARAKONSTANTIS, G.; ROY, K. Significance driven computation: A Voltage-Scalable, Variation-Aware, Quality-Tuning Motion Estimator. In: **Proceedings of the 14th ACM/IEEE international symposium on Low power electronics and design - ISLPED '09**. New York, New York, USA: ACM Press, 2009. p. 195. ISBN 9781605586847. Disponível em: <http://portal.acm.org/citation.cfm?doid=1594233.1594282>.

MOORE, G. E. Cramming More Components Onto Integrated Circuits. **Electronics**, p. 114–117, jan 1965. ISSN 0018-9219. Disponível em: <http://ieeexplore.ieee.org/lpdocs/epic03/wrapper.htm?arnumber=658762>.

MOREAU, T. et al. SNNAP: Approximate Computing on Programmable SoCs via Neural Acceleration. In: **2015 IEEE 21st International Symposium on High Performance Computer Architecture (HPCA)**. IEEE, 2015. p. 603–614. ISBN 978-1-4799-8930-0. Disponível em: <http://ieeexplore.ieee.org/document/7056066/>.

MUDGE, T. Power: a first-class architectural design constraint. **Computer**, v. 34, n. 4, p. 52–58, apr 2001. ISSN 00189162. Disponível em: <http://ieeexplore.ieee.org/document/917539/>.

NOWAK, E. et al. Turning silicon on its edge. **IEEE Circuits and Devices Magazine**, v. 20, n. 1, p. 20–31, jan 2004. ISSN 8755-3996. Disponível em: <http://ieeexplore.ieee.org/document/1263404/>.

OLUKOTUN, K.; HAMMOND, L. The Future of Microprocessors. **Queue**, ACM, v. 3, n. 7, p. 26, sep 2005. ISSN 15427730. Disponível em: <http://dl.acm.org/citation.cfm?id=1095418>.

PALACHARLA, S.; JOUPPI, N. P.; SMITH, J. E. Complexity-effective superscalar processors. In: **Proceedings of the 24th annual international symposium on Computer architecture**. ACM, 1997. p. 206–218. ISBN 0-89791-901-7. Disponível em: <http://doi.acm.org/10.1145/264107.264201>.

PARK, J. et al. AxGames: Towards Crowdsourcing Quality Target Determination in Approximate Computing. In: **Proceedings of the Twenty-First International Conference on Architectural Support for Programming Languages and Operating Systems - ASPLOS '16**. New York, New York, USA: ACM Press, 2016. v. 51, n. 4, p. 623–636. ISBN 9781450340915. ISSN 0362-1340. Disponível em: <http://dl.acm.org/citation.cfm?doid=2872362.2872376>.

PATEL, S.; HWU, W.-m. W. Accelerator Architectures. **IEEE Micro**, v. 28, n. 4, p. 4–12, jul 2008. ISSN 0272-1732. Disponível em: <http://ieeexplore.ieee.org/document/4626814/>.

PINCKNEY, N. et al. Limits of Parallelism and Boosting in Dim Silicon. **IEEE Micro**, v. 33, n. 5, p. 30–37, sep 2013. ISSN 0272-1732. Disponível em: <http://ieeexplore.ieee.org/document/6560066/>.

PINCKNEY, N. et al. Impact of FinFET on Near-Threshold Voltage Scalability. **IEEE Design & Test**, v. 34, n. 2, p. 31–38, apr 2017. ISSN 2168-2356. Disponível em: <http://ieeexplore.ieee.org/document/7747444/>.

RAHA, A.; JAYAKUMAR, H.; RAGHUNATHAN, V. A Power Efficient Video Encoder Using Reconfigurable Approximate Arithmetic Units. In: **2014 27th International Conference on VLSI Design and 2014 13th International Conference on Embedded Systems**. IEEE, 2014. p. 324–329. ISBN 978-1-4799-2513-1. Disponível em: <http://ieeexplore.ieee.org/document/6733151/>.

REHMAN, S. et al. Architectural-space exploration of approximate multipliers. In: **Proceedings of the 35th International Conference on Computer-Aided Design - ICCAD '16**. New York, New York, USA: ACM Press, 2016. p. 1–8. ISBN 9781450344661. Disponível em: <http://dl.acm.org/citation.cfm?doid=2966986.2967005>.

REHMAN, S. et al. Reliable software for unreliable hardware. In: **Proceedings of the seventh IEEE/ACM/IFIP international conference on Hardware/software codesign and system synthesis - CODES+ISSS '11**. New York, New York, USA: ACM Press, 2011. p. 237–246. ISBN 9781450307154. Disponível em: <http://dl.acm.org/citation.cfm?doid=2039370.2039408>.

REHMAN, S.; SHAFIQUE, M.; HENKEL, J. J. Instruction scheduling for reliability-aware compilation. In: **Proceedings of the 49th Annual Design Automation Conference on - DAC '12**. New York, New York, USA: ACM Press, 2012. p. 1292. ISBN 9781450311991. Disponível em: <http://dl.acm.org/citation.cfm?doid=2228360.2228601>.

ROY, P. et al. ASAC: Automatic Sensitivity Analysis for Approximate Computing. In: **Proceedings of the 2014 SIGPLAN/SIGBED conference on Languages, compilers and tools for embedded systems - LCTES '14**. New York, New York, USA: ACM Press, 2014. v. 49, n. 5, p. 95–104. ISBN 9781450328777. ISSN 0362-1340. Disponível em: <http://dl.acm.org/citation.cfm?doid=2597809.2597812>.

ROY, P.; WANG, J.; WONG, W. F. **PAC: program analysis for approximation-aware compilation**. IEEE Press, 2015. 69–78 p. ISBN 9781467383202. Disponível em: <http://dl.acm.org/citation.cfm?id=2830700>.

RUTZIG, M. B.; BECK, A. C. S.; CARRO, L. A transparent and energy aware reconfigurable multiprocessor platform for simultaneous ILP and TLP exploitation. In: **Proceedings of the Conference on Design, Automation and Test in Europe**. EDA Consortium, 2013. p. 1559–1564. ISBN 978-1-4503-2153-2. Disponível em: <http://dl.acm.org/citation.cfm?id=2485288.2485659>.

SAKURAI, T.; NEWTON, A. Alpha-power law MOSFET model and its applications to CMOS inverter delay and other formulas. **IEEE Journal of Solid-State Circuits**, v. 25, n. 2, p. 584–594, apr 1990. ISSN 00189200. Disponível em: <http://ieeexplore.ieee.org/document/52187/>.

SAMADI, M. et al. Paraprox: pattern-based approximation for data parallel applications. In: **Proceedings of the 19th International Conference on Architectural Support for Programming Languages and Operating Systems**. ACM, 2014. p. 35–50. Disponível em: <http://dl.acm.org/citation.cfm?id=2541948{&}CFID=955970568{&}CFTOKEN=67>.

SAMADI, M. et al. SAGE: Self-Tuning Approximation for Graphics Engines. In: **Proceedings of the 46th Annual IEEE/ACM International Symposium on Microarchitecture - MICRO-46**. New York, New York, USA: ACM Press, 2013. p. 13–24. ISBN 9781450326384. Disponível em: <http://dl.acm.org/citation.cfm?doid=2540708.2540711>.

SAMPSON, A. et al. **ACCEPT: A programmer-guided compiler framework for practical approximate computing**. [S.l.], 2015. 12 p.

SAMPSON, A. et al. EnerJ: Approximate Data Types for Safe and General Low-Power Computation. In: **Proceedings of the 32nd ACM SIGPLAN conference on Programming language design and implementation - PLDI '11**. New York, New York, USA: ACM Press, 2011. v. 46, n. 6, p. 164. ISBN 9781450306638. ISSN 0362-1340. Disponível em: <http://portal.acm.org/citation.cfm?doid=1993498.1993518>.

SANCHEZ, D.; KOZYRAKIS, C. ZSim: Fast and Accurate Microarchitectural Simulation of Thousand-Core Systems. **ACM SIGARCH Computer Architecture News**, ACM, v. 41, n. 3, p. 475–486, jul 2013. ISSN 01635964. Disponível em: <http://dl.acm.org/citation.cfm?id=2508148.2485963>.

SANKARALINGAM, K. et al. Exploiting ILP, TLP, and DLP with the polymorphous TRIPS architecture. In: **30th Annual International Symposium on Computer Architecture, 2003. Proceedings.** IEEE Comput. Soc, 2003. p. 422–433. ISBN 0-7695-1945-8. ISSN 1063-6897. Disponível em: <http://ieeexplore.ieee.org/lpdocs/epic03/wrapper.htm?arnumber=1207019>.

SATO, Y. et al. An Approximate Computing Stack Based on Computation Reuse. In: **2015 Third International Symposium on Computing and Networking (CANDAR)**. IEEE, 2015. p. 378–384. ISBN 978-1-4673-9797-1. Disponível em: <http://ieeexplore.ieee.org/document/7424742/>.

SHAFAEI, A. et al. FinCACTI: Architectural Analysis and Modeling of Caches with Deeply-Scaled FinFET Devices. In: **2014 IEEE Computer Society Annual Symposium on VLSI**. IEEE, 2014. p. 290–295. ISBN 978-1-4799-3765-3. Disponível em: <http://ieeexplore.ieee.org/document/6903378/>.

SHAFIQUE, M. et al. A low latency generic accuracy configurable adder. In: **Proceedings of the 52nd Annual Design Automation Conference on - DAC '15**. New York, New York, USA: ACM Press, 2015. p. 1–6. ISBN 9781450335201. Disponível em: <http://dl.acm.org/citation.cfm?doid=2744769.2744778>.

SHAFIQUE, M.; GARG, S. Computing in the Dark Silicon Era: Current Trends and Research Challenges. **IEEE Design & Test**, v. 34, n. 2, p. 8–23, apr 2017. ISSN 2168-2356. Disponível em: <http://ieeexplore.ieee.org/document/7762141/>.

SHAFIQUE, M. et al. Cross-Layer Approximate Computing: From Logic to Architectures. In: **2016 53nd ACM/EDAC/IEEE Design Automation Conference (DAC)**. [S.l.: s.n.], 2016. p. 1–6.

SHIMPI, A. L. **The Haswell Review: Intel Core i7-4770K & i7-4670K Tested**. 2013. Disponível em: <http://www.anandtech.com/show/7003/the-haswell-review-intel-core-i74770k-i54560k-tested/5>.

SIDIROGLOU-DOUSKOS, S. et al. Managing Performance vs. Accuracy Trade-offs with Loop Perforation. In: **Proceedings of the 19th ACM SIGSOFT symposium and the 13th European conference on Foundations of software engineering - SIGSOFT/FSE '11**. New York, New York, USA: ACM Press, 2011. p. 124–134. ISBN 9781450304436. Disponível em: <http://dl.acm.org/citation.cfm?doid=2025113.2025133>.

SILVANO, C. et al. Voltage island management in near threshold manycore architectures to mitigate dark silicon. In: **Design, Automation & Test in Europe Conference & Exhibition (DATE), 2014**. New Jersey: IEEE Conference Publications, 2014. p. 1–6. ISBN 9783981537024. Disponível em: <http://ieeexplore.ieee.org/xpl/articleDetails.jsp?arnumber=6800415>.

SINGH, H. et al. MorphoSys: an integrated reconfigurable system for data-parallel and computation-intensive applications. **IEEE Transactions on Computers**, v. 49, n. 5, p. 465–481, may 2000. ISSN 00189340. Disponível em: <http://ieeexplore.ieee.org/document/859540/>.

SNAPDRAGON 810 Product Specification. 2015. Disponível em: <https://www.qualcomm.com/products/snapdragon/processors/810>.

SODANI, A.; SOHI, G. S. Dynamic instruction reuse. **ACM SIGARCH Computer Architecture News**, ACM, v. 25, n. 2, p. 194–205, may 1997. ISSN 01635964. Disponível em: <http://dl.acm.org/citation.cfm?id=384286.264200>.

SOUZA, J. D. et al. A reconfigurable heterogeneous multicore with a homogeneous ISA. In: **Proceedings of the 2016 Conference on Design, Automation & Test in Europe**. EDA Consortium, 2016. p. 1598–1603. ISBN 9783981537062. Disponível em: <http://dl.acm.org/citation.cfm?id=2972181>.

STITT, G.; VAHID, F. Thread Warping: Dynamic and Transparent Synthesis of Thread Accelerators. **ACM Transactions on Design Automation of Electronic Systems**, ACM, v. 16, n. 3, p. 1–21, jun 2011. ISSN 10844309. Disponível em: <http://portal.acm.org/citation.cfm?doid=1970353.1970365>.

SURESH, A. et al. Intercepting Functions for Memoization. **ACM Transactions on Architecture and Code Optimization**, ACM, v. 12, n. 2, p. 18:1–18:23, jun 2015. ISSN 15443566. Disponível em: <http://dl.acm.org/citation.cfm?id=2775085.2751559>.

SWANSON, S. et al. WaveScalar. In: **Proceedings of the 36th annual IEEE/ACM International Symposium on Microarchitecture**. IEEE Computer Society, 2003. p. 291. ISBN 076952043X. Disponível em: <http://dl.acm.org/citation.cfm?id=956546>.

TAN, C. et al. LOCUS: Low-Power Customizable Many-Core Architecture for Wearables. **ACM Transactions on Embedded Computing Systems**, ACM, v. 17, n. 1, p. 1–26, nov 2017. ISSN 15399087. Disponível em: <http://dl.acm.org/citation.cfm?doid=3136518.3122786>.

TAYLOR, M. B. A Landscape of the New Dark Silicon Design Regime. **IEEE Micro**, v. 33, n. 5, p. 8–19, sep 2013. ISSN 0272-1732. Disponível em: <http://ieeexplore.ieee.org/lpdocs/epic03/wrapper.htm?arnumber=6583151>.

UBAL, R. et al. Multi2Sim: a simulation framework for CPU-GPU computing. In: **Proceedings of the 21st international conference on Parallel architectures and compilation techniques - PACT '12**. New York, New York, USA: ACM Press, 2012. p. 335. ISBN 9781450311823. Disponível em: <http://dl.acm.org/citation.cfm?id=2370816.2370865>.

VASSILIADIS, S. et al. The MOLEN Polymorphic Processor. **IEEE Transactions on Computers**, v. 53, n. 11, p. 1363–1375, nov 2004. ISSN 0018-9340. Disponível em: <http://ieeexplore.ieee.org/document/1336759/>.

VENKATARAMANI, S. et al. Quality programmable vector processors for approximate computing. In: **Proceedings of the 46th Annual IEEE/ACM International Symposium on Microarchitecture - MICRO-46**. New York, New York, USA: ACM Press, 2013. p. 1–12. ISBN 9781450326384. Disponível em: <http://dl.acm.org/citation.cfm?doid=2540708.2540710>.

WALL, D. W. Limits of instruction-level parallelism. **ACM SIGPLAN Notices**, ACM, v. 26, n. 4, p. 176–188, apr 1991. ISSN 03621340. Disponível em: <http://dl.acm.org/citation.cfm?id=106973.106991>.

WANG, Z. et al. Image Quality Assessment: From Error Visibility to Structural Similarity. **IEEE Transactions on Image Processing**, v. 13, n. 4, p. 600–612, apr 2004. ISSN 1057-7149. Disponível em: <http://ieeexplore.ieee.org/document/1284395/>.

WATKINS, M. A.; NOWATZKI, T.; CARNO, A. Software transparent dynamic binary translation for coarse-grain reconfigurable architectures. In: **2016 IEEE International Symposium on High Performance Computer Architecture (HPCA)**. IEEE, 2016. p. 138–150. ISBN 978-1-4673-9211-2. Disponível em: <http://ieeexplore.ieee.org/document/7446060/>.

WAZLOWSKI, M. et al. PRISM-II compiler and architecture. In: **Proceedings IEEE Workshop on FPGAs for Custom Computing Machines**. IEEE Comput. Soc. Press, 1993. p. 9–16. ISBN 0-8186-3890-7. Disponível em: <http://ieeexplore.ieee.org/document/279484/>.

WIJTVLIET, M.; WAEIJEN, L.; CORPORAAL, H. Coarse Grained Reconfigurable Architectures in the Past 25 Years: Overview and Classification. In: **2016 International Conference on Embedded Computer Systems: Architectures, Modeling and Simulation (SAMOS)**. IEEE, 2016. p. 235–244. ISBN 978-1-5090-3076-7. Disponível em: <http://ieeexplore.ieee.org/document/7818353/>.

WIRTHLIN, M.; HUTCHINGS, B. A dynamic instruction set computer. In: **Proceedings IEEE Symposium on FPGAs for Custom Computing Machines**. IEEE Comput. Soc. Press, 1995. p. 99–107. ISBN 0-8186-7086-X. Disponível em: <http://ieeexplore.ieee.org/document/477415/>.

WITTIG; CHOW. OneChip: an FPGA processor with reconfigurable logic. In: **Proceedings IEEE Symposium on FPGAs for Custom Computing Machines FPGA-96**. IEEE, 1996. p. 126–135. ISBN 0-8186-7548-9. Disponível em: <http://ieeexplore.ieee.org/document/564773/>.

XIE, Q. et al. Performance Comparisons Between 7-nm FinFET and Conventional Bulk CMOS Standard Cell Libraries. **IEEE Transactions on Circuits and Systems II: Express Briefs**, v. 62, n. 8, p. 761–765, aug 2015. ISSN 1549-7747. Disponível em: <http://ieeexplore.ieee.org/document/7012086/>.

XU, Q.; MYTKOWICZ, T.; KIM, N. S. Approximate Computing: A Survey. **IEEE Design & Test**, IEEE, v. 33, n. 1, p. 8–22, feb 2016. ISSN 2168-2356. Disponível em: <http://ieeexplore.ieee.org/articleDetails.jsp?arnumber=7348659>.

YAZDANBAKHSH, A. et al. AxBench: A Benchmark Suite for Approximate Computing. **IEEE Design and Test**, n. special issue on Computing in the Dark Silicon Era 2016, 2016. Disponível em: <http://hdl.handle.net/1853/54485>.

YAZDANBAKHSH, A. et al. Neural Acceleration for GPU Throughput Processors. In: **Proceedings of the 48th International Symposium on Microarchitecture - MICRO-48**. New York, New York, USA: ACM Press, 2015. p. 482–493. ISBN 9781450340342. Disponível em: <http://dl.acm.org/citation.cfm?doid=2830772.2830810>.

YE, R. et al. On Reconfiguration-Oriented Approximate Adder Design and Its Application. In: **Proceedings of the International Conference on Computer-Aided**

Design. IEEE Press, 2013. p. 48–54. ISBN 9781479910694. Disponível em: <https://dl.acm.org/citation.cfm?id=2561838http://ieeexplore.ieee.org/document/6691096/>.

YE, Z. A. et al. CHIMAERA: A High-Performance Architecture with a Tightly-Coupled Reconfigurable Functional Unit. In: **Proceedings of the 27th annual international symposium on Computer architecture - ISCA '00**. New York, New York, USA: ACM, 2000. v. 28, n. 2, p. 225–235. ISBN 1-58113-232-8. ISSN 01635964. Disponível em: <http://portal.acm.org/citation.cfm?doid=339647.339687http://dl.acm.org/citation.cfm?id=342001.339687>.

ZHAI, B. et al. Energy efficient near-threshold chip multi-processing. In: **Proceedings of the 2007 international symposium on Low power electronics and design - ISLPED '07**. New York, New York, USA: ACM Press, 2007. p. 32–37. ISBN 9781595937094. Disponível em: <http://portal.acm.org/citation.cfm?doid=1283780.1283789>.

ZHANG, H. et al. A 1-V heterogeneous reconfigurable DSP IC for wireless baseband digital signal processing. **IEEE Journal of Solid-State Circuits**, v. 35, n. 11, p. 1697–1704, nov 2000. ISSN 0018-9200. Disponível em: <http://ieeexplore.ieee.org/document/881217/>.

ZHANG, Q. et al. ApproxANN: An Approximate Computing Framework for Artificial Neural Network. In: **Proceedings of the 2015 Design, Automation & Test in Europe Conference & Exhibition**. [S.l.]: EDAA, 2015. p. 701–706. ISBN 9783981537048.