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

GUILHERME DOS SANTOS KOROL

A Resource-Aware Multicore CGRA Architecture for Edge Applications

Thesis presented in partial fulfillment
of the requirements for the degree of
Master of Microeletronics

Advisor: Prof. Dr. Antonio Carlos Schneider Beck
Coadvisor: Dr. Marcelo Brandalero

Porto Alegre
July 2020

CIP — CATALOGING-IN-PUBLICATION

Korol, Guilherme dos Santos

A Resource-Aware Multicore CGRA Architecture for Edge Applications / Guilherme dos Santos Korol. – Porto Alegre: PGMICRO da UFRGS, 2020.

104 f.: il.

Thesis (Master) – Universidade Federal do Rio Grande do Sul. Programa de Pós-Graduação em Microeletrônica, Porto Alegre, BR–RS, 2020. Advisor: Antonio Carlos Schneider Beck; Coadvisor: Marcelo Brandalero.

1. CGRA. 2. Resource management. 3. Power gating. 4. Reconfigurable architectures. I. Beck, Antonio Carlos Schneider. II. Brandalero, Marcelo. III. Título.

“The universe is under no obligation to make sense to you”

— NEIL DEGRASSE TYSON

AGRADECIMENTOS

Gostaria de expressar minha gratidão a todos que tornaram esse trabalho possível.

Em primeiro lugar, agradeço a minha família. Para Leo, Iris e Ricardo, que sempre me apoiaram, meu mais sincero muito obrigado. Sem vocês, eu não teria chegado até aqui.

Para meu orientador, Prof. Dr. Antonio Carlos Schneider Beck, gostaria de agradecer pela paciência e pelos ensinamentos passados, que não somente técnicos, levarei para minha carreira. Também, ao meu co-orientador, Dr. Marcelo Bradalero, que facilitou enormemente a realização desse trabalho, estando sempre disposto a solucionar minhas dúvidas.

Aos meus colegas de laboratório, com quem pude dividir momentos de alegria e trabalho, meu muito obrigado. Estes, que sempre ofereceram ajuda, foram fundamentais ao longo do ano. Em especial, ao meu amigo, Michael Guilherme Jordan, que desde a primeira semana, têm se empenhado nesse projeto.

Finalmente, quero agradecer a minha namorada Joice. Minha sorte foi poder contar contigo ao longo desse caminho. Muito obrigado pelo teu suporte e amor.

ABSTRACT

Edge devices on the Internet of Things (IoT) are intelligent, cloud-connected, usually battery-operated systems that are increasing in numbers and that are used in many applications, including smart homes, agriculture, healthcare, transportation, security, and telecommunication. In particular, these performance-hungry devices have exposed designers to the problem of achieving high throughput and low latency in environments with limited power supply. Moreover, as applications migrate from the cloud to the edge, these devices must concurrently execute a broad range of applications, and now feature multiple cores to address this demand. However, as multicore systems do not provide the best adaptability between the hardware and the various applications with distinct resource requirements, many works have investigated the use of reconfigurable architectures, in particular, Coarse-Grained Reconfigurable Architectures (CGRAs), to enable energy-efficient processing. Currently, this adaptation is limited to either homogeneous organizations, aiming at the highest performance, or heterogeneous organizations, aiming at improved energy efficiency. We propose in this work a novel approach to increase the energy efficiency of CGRAs by dynamically monitoring their underutilized resources and applying power gating to them in order to save power with minimal impact on performance. Then, we extend the approach to a multicore architecture featuring multiple CGRAs, where a central controller detects which applications are underutilizing their CGRAs and cleverly tunes the power gating of each unit to meet a system-wide power constraint. By using the proposed approach, we enable a homogeneous CGRA architecture to achieve the energy consumption levels of a heterogeneous one, since the extra degree of adaptability offered by the online management transparently matches the system's resources to the applications at hand. Overall, it is possible to achieve average reductions in EDP of over to 40% when the proposed architecture is compared to its homogeneous and heterogeneous counterparts.

Keywords: CGRA. resource management. power gating. reconfigurable architectures.

Uma arquitetura Multiprecessada CGRA com Gerenciamento de Recursos para Aplicações *Edge*

RESUMO

Dispositivos *Edge* na Internet das Coisas (IoT) são sistemas inteligentes, conectados à nuvem, normalmente movidos a bateria que estão aumentando em número e são utilizados em muitas aplicações, que incluem casas inteligentes, agricultura, saúde, transporte, segurança, e telecomunicações. Em especial, esses dispositivos de alta demanda computacional vem expondo projetistas ao problema de alcançar alta vazão e baixa latência em ambientes de alimentação limitada. E mais, com aplicações migrando da nuvem para *Edge*, esses dispositivos estão executando paralelamente mais e mais aplicações, e hoje em dia possuem múltiplos processadores para este tipo de processamento. Entretanto, como as arquiteturas multiprocessadas não oferecem a melhor adaptabilidade entre o *hardware* e as várias aplicações com requisitos de recursos diversos, muitos trabalhos têm investigado o uso de arquiteturas reconfiguráveis. em especial, as Arquiteturas Reconfiguráveis de Grão Grosso (CGRA), para prover processamento eficiente energeticamente. Atualmente, essa adaptabilidade está limitada a ter organização homogênea, objetivando maior desempenho, ou a ter organização heterogênea, objetivando melhoria da eficiência energética. É proposta neste trabalho uma nova abordagem para aumentar a eficiência energética das CGRAs que dinamicamente monitora os recursos subutilizados e aplica *power gate* de forma a economizar potência com mínimos impactos em performance. Então, ampliamos a abordagem para uma arquitetura multiprocessada com múltiplas CGRAs, onde um controlador centralizado detecta quais aplicações estão subutilizando suas CGRAs e regula o *power gate* de cada unidade para satisfazer uma restrição de potência em nível de sistema. Usando a abordagem proposta, promovemos uma arquitetura de CGRAs homogêneas para atingir níveis de consumo energético de uma heterogênea, uma vez que o grau adicional de adaptabilidade oferecido pelo gerenciamento *online* adequa de forma transparente os recursos do sistema às aplicações em uso. No geral, é possível atingir reduções em EDP de mais de 40% quando a arquitetura proposta é comparada com suas equivalentes homogênea e heterogênea.

Palavras-chave: CGRA, gerência de recursos, *power gating*, arquiteturas reconfiguráveis.

LIST OF ABBREVIATIONS AND ACRONYMS

ALU	Arithmetic Logic Unit
AMU	Active Mapping Unit
ASIC	Application-Specific Integrated Circuit
BT	Binary Translator
CC	Configuration Cache
CCA	Configurable Compute Accelerator
CGRA	Coarse-Grained Reconfigurable Architectures
CLB	Configurable Logic Block
CMOS	Complementary MOS
CPU	Central Processing Unit
CReAMS	Custom Reconfigurable Arrays for Multiprocessor System
DAP	Dynamic Adaptive Processors
DDH	Dynamic Detection Hardware
DFG	Data Flow Graph
DIM	Dynamic Instruction Merging
DORA	Dynamic Optimizer for Reconfigurable Architectures
DVFS	Dynamic Voltage and Frequency Scaling
DynaSpAM	Dynamic Spatial Architecture Mapping
DySER	Dynamically Specialized Execution Resource
FPGA	Field Programmable Gate Arrays
FS	Full System
FU	Functional Unit
GPP	General-Purpose Processors
HARTMP	Reconfigurable and Transparent Multicore Processing

ILP	Instruction Level Parallelism
IoT	Internet of Things
IPC	Instruction Per Cycle
IPS	Instructions Per Second
ISA	Instruction Set Architecture
ISE	Instruction Set Extension
LCB	Local Clock Buffers
LUT	LookUp Table
MMU	Memory Management Unit
MOS	Metal Oxide Semiconductor
NoC	Network-on-Chip
NRE	Non-Recurring Engineering
OoO	Out-of-Order
PC	Program Counter
PCPG	Per-Core Power Gate
PE	Processing Elements
PMU	Power Management Unit
PX-CGRA	Polymorphic Approximate CGRA
RA	Reconfigurable Architecture
ReMAP	Reconfigurable Multicore Acceleration and Parallelization
RFU	Reconfigurable Functional Units
RISP	Reconfigurable-Instruction-Set Processors
RISPP	Rotating Instruction Set Processing Platform
RTL	Register Transfer Level
SE	System call Emulation
SFU	Shift and Mask Unit

SMT	Simultaneous Multithreading
SoC	System on Chip
TLP	Thread Level Parallelism
TRePU	TransRec Processing Unit
TTL	Transistor-Transistor Logic
VLIW	Very Large Instruction Word
VMU	Virtual Mapping Unit

LIST OF FIGURES

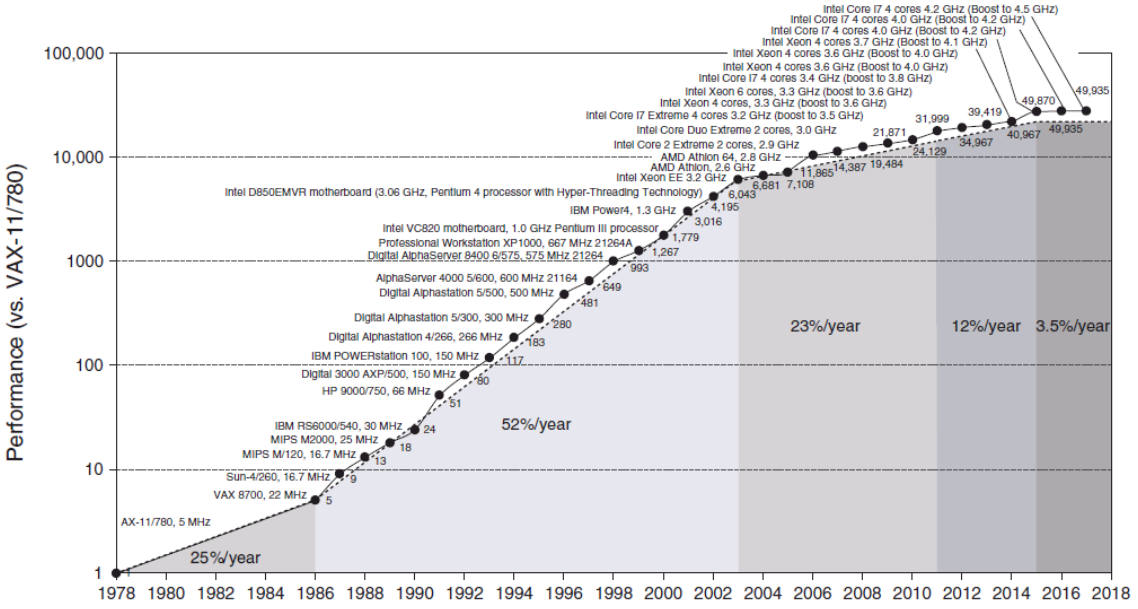
LIST OF TABLES

CONTENTS

1 INTRODUCTION

Two main forces have driven an exponential increase in microprocessors' performance over the last years (Figure 1.1). Process technology and microarchitectural advances have granted speedups for users that could simply wait for the next technology generation (OLUKOTUN; HAMMOND, 2005). Precisely, the technological aspect is pushed by Moore's Law (MOORE, 2006) that, in conjunction with Dennard Scaling (DENNARD et al., 1974), ruled that for every new technology node, transistor integration doubles, circuits are 40% faster, and power dissipation stays the same. Technology scaling alone has enabled a three-orders-of-magnitude performance increase in the past forty years (BORKAR; CHIEN, 2011).

Figure 1.1: Growth in performance relative to the VAX 11/780 processor from 1978 to 2018.



Source: (HENNESSY, 2017)

However, a continuous reduction of the supply and threshold voltage is required in order to keep up with the projections of Dennard scaling. As it turns out, the transistor is not a perfect switch. Hence, a challenge that was not foreseen in Dennard et al. (1974) - leakage - has raised its participation in the total chip's power dissipation, thereby both threshold and supply voltage do not scale at the desired rate, a fact that has put an end to the Dennard Scaling around 2003 (HENNESSY, 2017). In practice, power dissipation has exhibited an opposite trend from what was predicted by Dennard Scaling (Figure 1.2). Power dissipation has increased instead of being kept at constant values. The phenomenon

is best explained in Table 1.1 that presents power density as a function of the scaling factor S from before and after the Dennard Scaling breakdown. More recently, a slow down in Moore's Law is also observed. The market leader, Intel, has taken the time interval between new technologies nodes from the Moore-predicted, two years, to a slower pace. For example, the 32nm node took two years after the launch of the 45nm to start production. Later, the Intel's 22nm technology took 2.25 years to be launched after the 32nm, and the 14nm had its launch further delayed to a 2.5 years interval, showing a clear shift from the trend predicted by Moore (FLAMM, 2017). Consequently, the industry can no longer rely exclusively on technology scaling to provide a continuous performance increase for new generations of general-purpose processors.

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

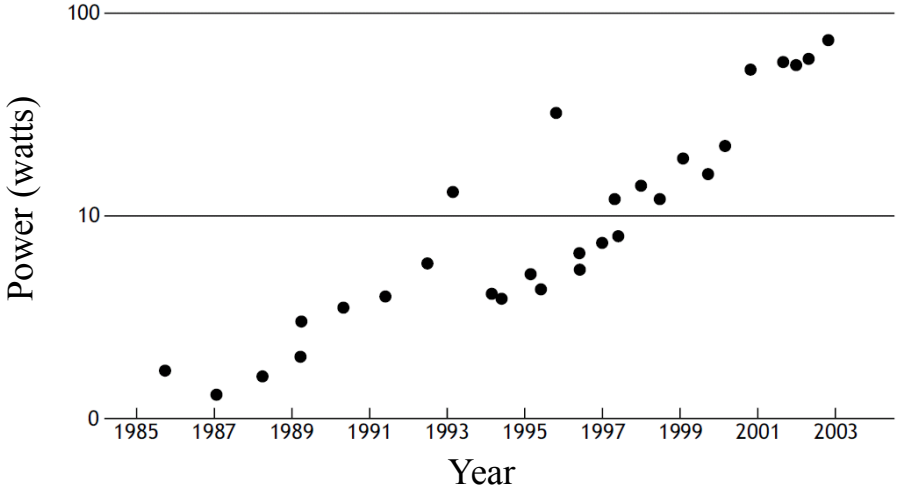
	Dennard Scaling	Post-Dennard Scaling
Device Count	S^2	S^2
Device Frequency	S	S
Device Power (cap)	$1/S$	$1/S$
Device Power (Vdd)	$1/S^2$	1
Power Density	1	S^2

Source: (SHAFIQUE; GARG, 2017)

On the other hand, microarchitectural advances have also contributed to the increase in performance observed over the last years. The additional transistor density has enabled advanced microarchitectural techniques to enhance performance even further. For example, techniques like pipelining, branch prediction, out-of-order execution, and speculation were proposed to increase the capability of extracting Instruction Level Parallelism (ILP) from applications' code. However, as was empirically described by Pollack's rule, performance gains due to extra circuitry are bounded by the square root of the number of transistors or area (POLLACK, 1999). For example, from a new microarchitectural advance that quadruples the processor die area, a performance improvement of only about $2\times$ is expected by Pollack's rule. Additionally, Wall (1991) advocated that relying only on applications' intrinsic instruction parallelism, as done in these single-processor architectures, could not sustain gains indefinitely. In other words, performance gains due to traditional microarchitecture techniques, such as the ones transparent to the application (e.g., superscalarity and dynamic scheduling) have stalled in recent microprocessor generations.

Inevitably, due to both technology stagnation, represented by the end of Dennard scaling and Moore's Law, as well as the reduction in microarchitectural-driven performance gains, the industry has signaled that the trend in general-purpose processors' per-

Figure 1.2: Power dissipation of Intel processors over the years.



Source: (OLUKOTUN; HAMMOND, 2005)

formance as observed in the last thirty years is likely to be discontinued, as shown in Figure 1.1. This change of paradigm forced a historic switch from single-processor architectures to Chip Multi-Processors (CMP) (OLUKOTUN et al., 1996). With CMPs, the focus shifted to throughput improvements. Attending multiple requests have become as, or more, important than attending a single request at a lower latency (OLUKOTUN; HAMMOND, 2005). For example, in a web server attending requests from multiple users, throughput may be more relevant than latency. Moreover, parallelization in CMPs is not restricted to the execution of independent tasks like users' requests in a web server. Even a single task can be broken into parallel threads. Exploring Thread Level Parallelism (TLP) in CMP systems has transferred responsibilities for performance improvements from the hardware designer to the software developer, who has to find in the application parts of the code that can execute in parallel - which is not an easy task (BLAKE et al., 2010).

However, as a wide range of applications, such as multimedia, encryption, and network, run on these systems, each of which with a particular computational requirement, a machine of single architecture and organization will not satisfy all the diverse computational requirements equally well (MITTAL, 2016). Indeed, the fact that applications' behavior varies greatly (e.g., in terms of ILP) is explored to the development of power and energy-efficient architectures. In heterogeneous computing, the strategy of mixing up cores with high computing capability with less powerful ones, as presented in (KUMAR et al., 2003), opened up many research opportunities to save energy by providing architectures with a better match to what was required by the different applications. Heterogeneity may come as processing elements with different architectures or organizations. For example, processors that implement the same Instruction Set Architecture (ISA) can

provide multiple levels of energy efficiency by having different organizations. On the other hand, the heterogeneous approach was taken even further with systems that include elements that vary their architectures. For instance, the use of customized hardware has proven itself to be a useful approach to increase efficiency (CHUNG et al., 2010). The abundance of transistors is employed to implement hardware customization based on hardwired or customized processing elements and data movement, creating highly energy-efficient computation units called accelerators (SHAFIQUE et al., 2014; KHAN; SHAFIQUE; HENKEL, 2015).

Works coupling accelerators to General-Purpose Processors (GPP) report orders-of-magnitude gains in performance (HAMEED et al., 2010; KOÇBERBER et al., 2013; CONG et al., 2014). Hardware accelerators provide the best efficiency since they are tailored to particular tasks. However, the use of fixed-function, or *application-specific*, accelerators comes with costs across all the design stack. From a hardware perspective, designing an accelerator like video and cryptography engines requires knowledge of the application by the hardware designers, which configures a more expensive design cycle when compared to the development of general-purpose, application-independent, machines. Also, as most accelerators are tailored for performing a specific task, they lack generality. Moreover, even though, at the time they are launched into the market, economics justify their implementation, future general-purpose solutions may, eventually, fill the original application's niche (PATEL; HWU, 2008). Whereas from a software perspective, the development cost involved in programming the accelerator might exceed the performance gains. Historically, software compatibility has played a significant role in the microprocessor industry. For instance, it has played a major role in the success of Intel's x86 ISA and the failure of its attempted replacement, the IA-64 (HENNESSY, 2017). Despite the efforts that have been made for making accelerators' programming interface easier to programmers, it is still a burden to software development (HWU; PATEL, 2018).

Alternatively, dynamically customizable, or *reconfigurable*, accelerators were proposed to bridge the gap between the efficiency offered by application-specific accelerators and the generality provided by GPPs. Precisely, the usage of Processing Elements (PEs) interconnected by programmable networks ensures that a wide range of applications can be executed over the same hardware fabric, addressing the generality issue and the already mentioned prohibitive Non-Recurring Engineering (NRE) costs of application-specific accelerators. Also, by customizing their datapath for specific computations,

reconfigurable architectures such as the Field Programmable Gate Arrays (FPGA) and the Coarse-Grained Reconfigurable Architectures (CGRA) address the lack of efficiency of GPPs. Even though CGRAs are more generic, have smaller programmability costs, and have better runtime adaptability than application-specific accelerators (WIJTVLIET; WAEIJEN; CORPORAAL, 2016; LIU et al., 2019), there is still room for improvement. This work investigates these architectures, aiming to reduce their power dissipation and energy consumption in order to enable their use in constrained environments such as the one presented below.

1.1 Motivation and Scope

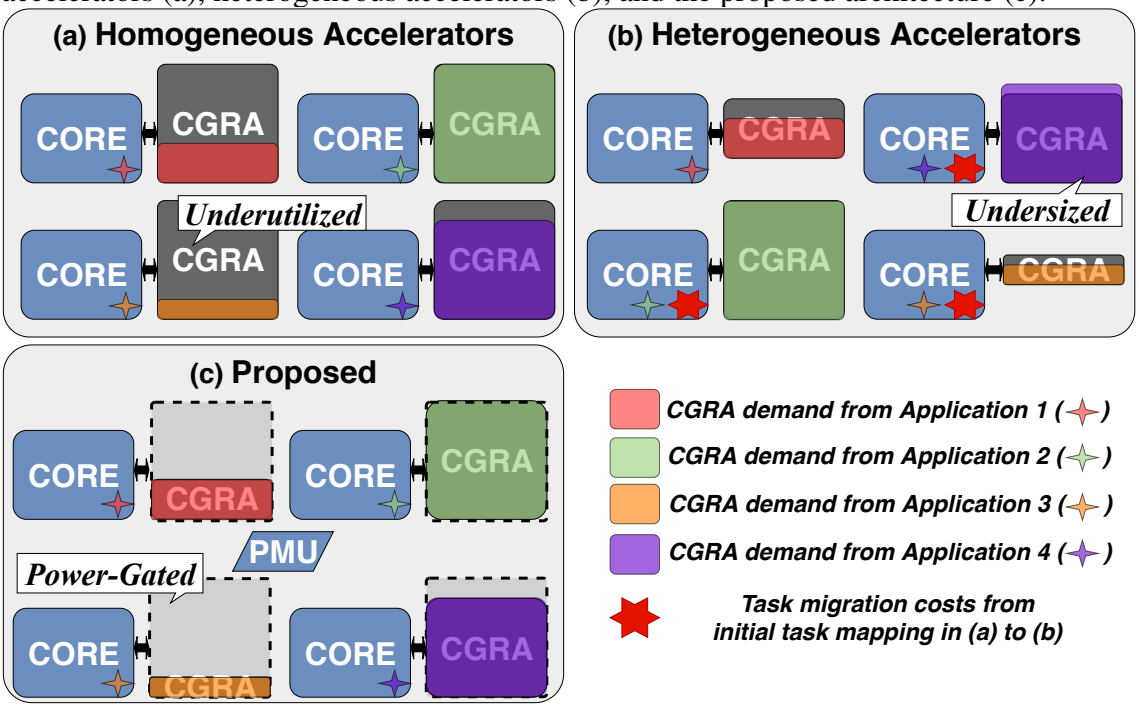
In recent years, computing devices have moved closer to sensors (PAGANI et al., 2017; ADEGBIJA et al., 2018). Internet of Things (IoT) and Edge devices are examples of where complexity has increased to attend users' expectations on performance, which further aggravates the issue, earlier discussed, of power density since these devices are usually constrained in power supply. Nowadays, the number of interconnected devices in IoT keeps increasing at a steady and fast pace. Estimations report that by 2025 the IoT will compose an economic impact ranging from $2.7 to $6.2 trillion, including more than 50 billion low-power devices world-wide (ADEGBIJA et al., 2018; MOHAMMADI et al., 2018). A significant part of those devices are capable of complex computations and are seen in almost every aspect of our daily lives: from cellphones, coffee makers, wearables, health appliances to washing machines, there is a growing need for fast, low latency responses. Thus, processing on the IoT has largely been transferred from the cloud to the network edge (SHI; DUSTDAR, 2016). These so-called Edge devices aim to alleviate the latency associated with sending and receiving data to/from the cloud, as well as other non-functional requirements like security and privacy. To that end, edge applications need to process data locally instead of remotely through the network. Since many of these devices are portable and battery-operated, they cannot keep up with the near-constant increase in processing requirements. Although techniques for energy efficiency have been developed, they cannot adequately satisfy the requirements for the new generation of IoT Edge applications.

Initially, to support the applications' demands for low latency, high throughput, and devices' physical limitations, homogeneous multicore architectures were extensively employed (ADEGBIJA et al., 2018). However, rising performance demands and signif-

icant energy requirements of Edge devices have pushed the use of heterogeneous architectures (SHI; DUSTDAR, 2016; FAN et al., 2019). Hence, works like (BAUER et al., 2010; KULKARNI et al., 2018; LI et al., 2019) that use either FPGA or ASIC devices are getting more popular for implementing energy-efficient Edge computing platforms. Still, few works have explored the use of general-purpose reconfigurable accelerators to enable modern multicore systems to work under Edge-tolerable energy consumption levels.

Besides the requirement of energy efficiency imposed by Edge applications, another crucial aspect of Edge devices is that they are inserted in a fast-evolving environment with new applications being continually launched into the market (SHI et al., 2016; MOHAMMADI et al., 2018). Hence, the programmability of Edge devices becomes a key aspect for economic reasons (SHI et al., 2016; SHAFIQUE et al., 2018). Aiming to simultaneously cope with the requirements of energy efficiency, performance, programmability, and adaptability, reconfigurable architectures present themselves as an attractive alternative for the new Edge applications.

Figure 1.3: Example of four sample applications running on systems with homogeneous accelerators (a), heterogeneous accelerators (b), and the proposed architecture (c).



Source: the author

Usually, multicore reconfigurable systems are *(1) homogeneous*, with reconfigurable accelerators of the same size distributed across the system (GUPTA et al., 2011; TAN et al., 2016), or *(2) heterogeneous*, with different accelerators coupled to the cores (SOUZA et al., 2016; TAN et al., 2018). Homogeneous architectures often incur in low

energy efficiency for applications with low ILP due to the underutilized processing resources. Figure 1.3(a) shows a multicore architecture based on a homogeneous set of reconfigurable accelerators. In the example, applications 1 and 3 are not leveraging the full potential of their CGRAs due to their low ILP, causing underutilization. On the other hand, with heterogeneous approaches, both performance and power efficiency will depend on the workload characteristics to fit in the available accelerators and on an efficient scheduler (and the associated costs of task migration) to rightly map the applications (MÜCK; SARMA; DUTT, 2015). An example of a heterogeneous multicore reconfigurable architecture is depicted in Figure 1.3(b). It is possible to note that to better accommodate the ILP demands of applications 2, 3, and 4, they had to be migrated to smaller accelerators in order to avoid underutilization, bringing together significant migration costs. Moreover, even though accelerators of multiples sizes are available, it will not always be possible to perfectly match acceleration demands from every application to the available heterogeneous CGRAs since their size are fixed after deployment (e.g., application 4 in Figure 1.3(b)).

Considering the discussion above, this work proposes a resource-aware multicore CGRA architecture tailored for Edge applications. While physically implemented as a *homogeneous* multicore reconfigurable architecture, it is as energy-efficient as a heterogeneous architecture, since it can automatically adapt its heterogeneity to the workload at hand and, by doing so, avoid the expensive energy costs of task migration (Figure 1.3(c)). This is achieved by employing a novel ILP/TLP-aware resource management module that dynamically supervises the acceleration provided by each of the CGRAs and the overall system TLP. That information is gathered cooperatively and used to automatically power gate underutilized CGRA resources and adapt the system at runtime to the tasks at hand, keeping the power dissipation at edge-tolerable levels. Precisely, when the proposed architecture is evaluated against its homogeneous and heterogeneous counterparts for a diverse set of edge applications, the average energy consumed by accelerators is reduced in 43.95% and 19.76%, respectively.

1.2 Contributions

With respect to the earlier discussion, the primary purpose of this work is to improve over existing CGRA architectures on energy efficiency by employing an intelligent power gating scheme, enabling its use on more energy-restricted environments. In sum-

mary, the contributions are as follows:

- This work proposes a resource-aware multicore reconfigurable architecture tailored for Edge applications. The work includes its hardware implementation and a global power management unit for CGRAs that, with no need for software schedulers, efficiently adapts the hardware resources to the applications at hand;
- Considering a relevant set of Edge applications, we assess the proposed architecture on performance, power, and energy, over a modern multicore system that includes four Out-of-Order 2-issue BOOM processors. The results show an average reduction of 63.35% in the total energy consumption with an average speedup of 1.11x over the multicore system.
- We also compare the proposed architecture to the Stitch (TAN et al., 2018) reconfigurable architecture, a state-of-the-art system for Edge applications, showing that the it is possible to achieve better speedups under the same power envelope with the advantage of being fully transparent and adaptive.

1.3 Structure of this dissertation

Chapter 2 presents a background on the concepts necessary to understand the contributions of this work. It begins by outlining some of the issues caused by the diversity of behavior presented in modern workloads. Next, Chapter 2 discusses the main techniques for implementing resource and power management. Concluding this chapter, adaptable architectures are outlined with a focus on reconfigurable architectures.

Chapter 3 presents the architecture proposed in this work. First, a top overview and description of the main modules are given. Later, the hardware modules first introduced in this work are described in more detail.

Chapter 4 presents two evaluations of the proposed system. In the first scenario, the system is assessed in a single-core architecture with a larger reconfigurable fabric. While in the second scenario, the full architecture is evaluated. In a multicore reconfigurable architecture, the proposed resource management techniques are evaluated against homogeneous and heterogeneous counterparts. Additionally, a comparison against a state-of-the-art reconfigurable architecture is proposed.

Finally, Chapter 5 concludes this document summarizing the main points and contributions. Then, some directions for future works are given, and the publications pro-

duced throughout this work are listed.

2 BACKGROUND

This chapter presents the main concepts and terminology required to understand this dissertation. The first section discusses particularities encountered in a modern workload scenario. Specifically, **Section 2.1 explores the dynamic behavior of modern applications** since this work leverages such characteristic to perform dynamic adaptations on the system's resources. Next, **Section 2.2 outlines the main sources of power dissipation** in current digital CMOS circuits. The discussion is followed by a presentation of the usual techniques used for dealing with the power issue in modern digital circuits, including the technique used in this work: power gating. Further, **Section 2.3 shows the architectural models used for increasing power and energy efficiency of computing devices**. Initially, the section presents the concept of heterogeneous computing that leverages the dynamic behavior of workloads for mixing into the same system processing elements with different processing capabilities. Later, **still in Section 2.3, reconfigurable architectures, the target of this work, are presented in greater detail**. Throughout the following sections, the state-of-the-art works on each subject are also given.

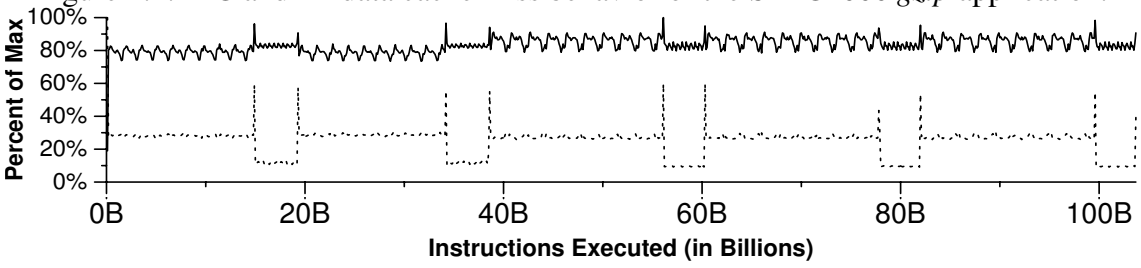
2.1 Dynamic Behavior of Workloads

Many programs present widely different behavior during their execution. For instance, a program may experience portions with an intensive memory-bound behavior followed by portions of intensive computing. The property known as *program phase* consists of a set of intervals within a program execution where the behavior (according to some metric) is regular (SHERWOOD et al., 2002). Although a program phase can reappear multiple times during a single execution, they are identified regardless of temporal adjacency.

Sherwood et al. (2002) analyzed the behavior of a set of applications from the SPEC 2000 benchmark suite (HENNING, 2000). Figure 2.1 displays the instruction per cycle (IPC) and L1 data cache miss for the *gzip* application. The behavior is analyzed over a 100 billion instructions interval (x-axis) with samples taken every 100 million instructions. The y-axis represents the percentage of maximum value that each metric had during execution. The first observation that can be made about Figure 2.1 is the clear identification of distinct phases. Program phases are a direct function of the way a program traverses its code. Hence, most program phases happen repeatedly during the

program execution.

Figure 2.1: IPC and L1 data cache miss behavior of the SPEC2000 *gzip* application.

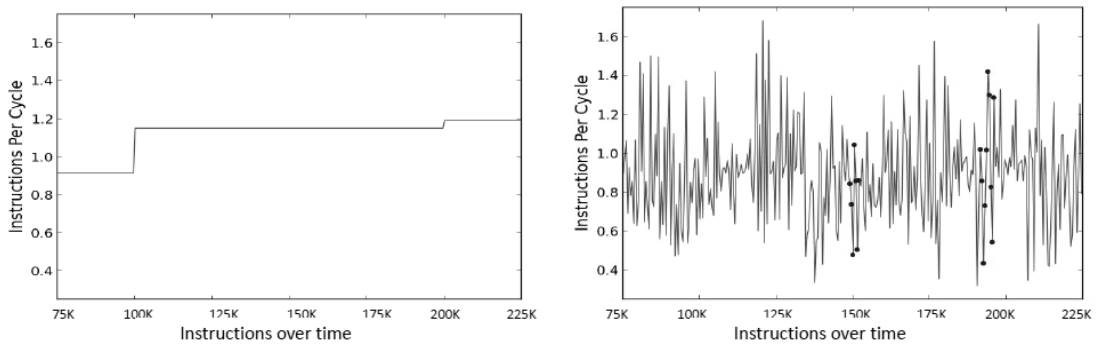


Source: (SHERWOOD et al., 2002)

The granularity in which samples are taken during the execution of a program is a crucial factor that contributes to the accuracy of program phase identification. In the work by Sheerwood et al., the authors adopted a coarse grain sampling (100 million instructions). However, other works have made attempts to the identification of program phases at finer grains (XU; ALBONESI, 1999; WUNDERLICH et al., 2003; RANGAN; WEI; BROOKS, 2009). The work in (PADMANABHA et al., 2013) explores the variance due to the sampling granularity for program phase identification. In Figure 2.2, the same trace with 300 thousand instructions of the *gcc* application is analyzed by two phase detectors. The first one (Figure 2.2(a)) samples the processor IPC every 100 thousand instructions. As for the second one, in Figure 2.2(b), samples are taken every 500 instructions. When the application is analyzed based on larger intervals, it shows a more stable behavior than what is extracted by the detector with smaller sampling intervals (fine grain). These differences in the analysis of a program can have a significant impact on a system's energy consumption, for example. Resource managers based on program phase detection must pay attention to these particularities. In Figure 2.2(b), some consecutive points with drastic IPC variance have been circled. The circles indicate moments in the execution where a decision based on data gathered previously could have a non-optimal outcome.

As a case-study on the impact of sampling granularity, Padmanabha et al. give the scheduling on a heterogeneous architecture of the type Big.LITTLE. In the example, the scheduler's sole objective is to schedule phases with low IPC to the energy-efficient in-order Little processor and performance demanding phases to the out-of-order Big processor. They show that when the sampling granularity is reduced from one million to one thousand instructions, the time spent on the Little processor increases 40% on average for the SPEC06 benchmark suite. If the granularity is further reduced to one hundred instructions, the average time running with Little is further increased by 45%.

Figure 2.2: Variance in IPC for the same 300 thousand *gcc* instructions with two sampling granularities.



(a) IPC sampled every 100K instructions (Coarse) shows stable performance between quanta

(b) IPC sampled every 500 instructions (Fine) shows highly variant performance between quanta

Source: (PADMANABHA et al., 2013)

Conclusively, identifying program phases enable software and hardware optimizations. Resource management is an excellent example of where the online detection of program phases makes it possible to perform adaptation of the system's resources in accordance with their usage by the running applications.

2.2 Resource and Power Management Techniques

2.2.1 CMOS Power Dissipation

After the first bipolar polar transistor (HARRIS, 1956), logic families like the Transistor-Transistor Logic (TTL) have widespread as the leading choice for implementing logic designs. Furthermore, until the 1980s, bipolar-based technologies have propelled the integrated circuit revolution (RABAEY; CHANDRAKASAN; NIKOLIC, 2004). Despite the high integration that bipolar has offered, its high power dissipation per gate hinders the implementation of more complex designs (that integrate a larger number of transistors).

The alternative to the power dissipation problem of bipolar devices arose with the MOS (Metal Oxide Semiconductor) technology. The first MOS family to be employed by industry was the PMOS-only. It enabled designs with unprecedented complexity, for example, the Intel 8008 microprocessor (SHIMA; FAGGIN; MAZOR, 1974). Later, the NMOS-only devices with their faster switching speeds were proposed to replace the slower PMOS-only devices. However, the same problem that caused the end of the bipolar era - power dissipation - also affected the NMOS-only logic family. The ma-

turity of fabrication processes and the continuous search for higher integration and lower power dissipation have driven the industry to adopt, the decades early proposed, CMOS (complementary MOS) logic family (WANLASS; SAH, 1963).

To this day, CMOS is the primary choice for implementing digital circuits. However, the power issue that struck prior technologies and logic families is still a major concern. Not only that, but as there is no new technology to shift to, designers have to find ways to overcome the power dissipation problem so the industry and academia can continue to innovate in the area.

In the following sections, the sources for power dissipation in CMOS circuits are presented along with techniques that aim to alleviate the problem in modern circuits.

Sources of Power Dissipation in CMOS. The total power dissipation (as in Equation 2.1) of a CMOS device can be decomposed into three contributing factors.

$$P_{total} = P_{dyn} + P_{sc} + P_{st} \tag{2.1}$$

The first factor (P_{dyn}) is the dynamic power. It represents the power dissipated by the switching activity of the circuit gates and its transistors. In conventional static CMOS, a gate load is charged by the PMOS transistors (pull-up network) and discharged by the NMOS transistors (pull-down network). When either action is performed, a certain amount of energy is drawn from the source and dissipated by either the PMOS transistor network (when charging) or dissipated through the NMOS transistor network (when discharging). Equation 2.2 details the CMOS dynamic power

$$P_{dyn} = C_L * {V_{DD}}^2 * P_{0\rightarrow1} * f \tag{2.2}$$

where: C_L denotes the gate's load capacitance, V_{DD} gives the supply voltage, $P_{0\rightarrow1}$ represents the probability that a clock event causes the gate's output to switch its current state (switching activity), and f is the operating frequency.

The second contributing factor to the total CMOS power dissipation (Equation 2.1) is the short-circuit power (P_{sc}). In practical CMOS circuits, when a transition happens at the input of a logic gate, both PMOS and NMOS networks may be simultaneously on due to non-ideal input slopes (input signal is above the NMOS voltage threshold and below the PMOS voltage threshold). Having both pull-up and pull-down networks conducting creates a direct path (short-circuit) between V_{DD} and $ground$. According to Veendrick (1984), the power dissipated by short circuits in CMOS designs can be modeled as

$$P_{sc} = K * (V_{DD} - 2V_{TH})^3 * \tau * f \tag{2.3}$$

where: K is a constant dependent on transistor size, V_{DD} is the supply voltage, V_{TH} is the threshold voltage, τ is the input signal rise or fall time (representing the time that both PMOS and NMOS are conducting), and f is the switching frequency.

The final contribution to the total CMOS power is due to static power (P_{st}). A CMOS gate in steady-state would ideally consume no energy since there is no direct path between V_{DD} and $ground$. However, in actual implementations, a small current that flows through the transistors is observed (I_{st}). Equation 2.4 gives the power due to the static current (flowing between V_{DD} and $ground$ when no switching activity is occurring).

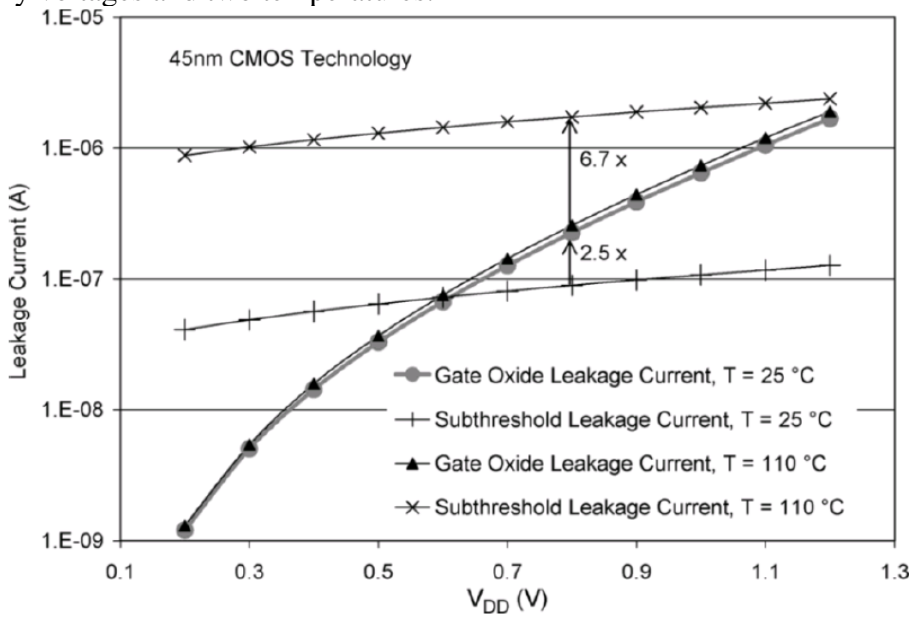
$$P_{st} = I_{st} * V_{DD} \tag{2.4}$$

There are various mechanisms affecting the static current. For example, *reverse-biased pn-junction leakage*, *gate-induced drain leakage*, and *punchthrough leakage* currents are part of the static current, but can be neglected for normal operation (AGARWAL et al., 2005). However, for recent technology nodes (below 45nm), two main sources can be attributed to the static current I_{st}: *subthreshold leakage* current (increased with the reduction of the threshold voltage) and *gate tunneling leakage* current (increased with the scaling of the gate oxide thickness) (LIU; KURSUN, 2006; MUKHOPADHYAY et al., 2003). Figure 2.3 shows the contribution of both currents in the total static current for a 45nm NMOS transistor. At high temperature and nominal supply voltage ($V_{DD} = 0.8$), the subthreshold is 6.7x larger than the gate leakage. Whereas at room temperature, the subthreshold corresponds to 2.5x of the current due to gate leakage. Indeed, for smaller technology nodes, it is observed an equal contribution between subthreshold and gate leakage currents.

As the voltage applied to the transistor gate gets below V_{TH}, it does not form an inversion channel. However, the drain-body junction is reversed biased (assuming that source and body are shorted together), and the movement of carriers forms the subthreshold leakage current due to diffusion. Moreover, as it can be modeled by $I_s * e^{\frac{q(V_{GS}-V_{TH})}{nKT/q}} * (1 - e^{\frac{V_{DS}}{KT/q}})$ the subthreshold current is said to be exponentially related to the gate voltage V_{GS}.

The second source of static current, gate tunneling leakage, is caused by the tunneling of electrons through from oxide layer to the gate and vice-versa. The problem is

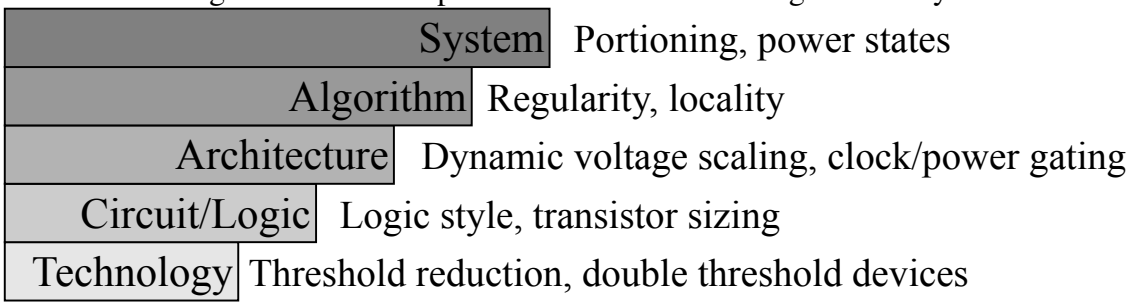
Figure 2.3: Subthreshold and gate oxide leakage currents in a NMOS transistors for varying supply voltages and two temperatures.



Source: (LIU; KURSUN, 2006)

aggravated in recent technology nodes since the probability of tunneling is enhanced by the high electrical fields due to thinner oxide layers.

Figure 2.4: Power optimizations across the design hierarchy.



Source: Adapted from (RABAEY; PEDRAM; LANDMAN, 1996)

Conclusively, both dynamic and static power are the major concerns in current designs since short-circuit current can be kept within bounds (since it represents less than 20% of the dynamic power (VEENDRICK, 1984)). Nowadays, every CMOS design employs, at some degree or another, efforts for reducing dynamic and static power dissipation. Optimizations at all levels of the design hierarchy have been used to reduce power dissipation, as illustrated in Figure 2.4. There are examples from technology level optimizations that approach the problem improving on traditional CMOS technology like double threshold transistors (WEI et al., 1998) or promoting the use of new materials like the high-k gate dielectrics (CHOI et al., 2002), to system-level optimizations that may implement some form of power management in processors like the Intel's StrongArm

SA-100 (INTEL, 1998) or over entire data-centers like the Facebook's Dynamo (WU et al., 2016).

Next, two specific techniques at the architecture level employed to mitigate power and energy consumption, which are relevant to the scope of this work, will be discussed.

2.2.2 Clock Gating based techniques

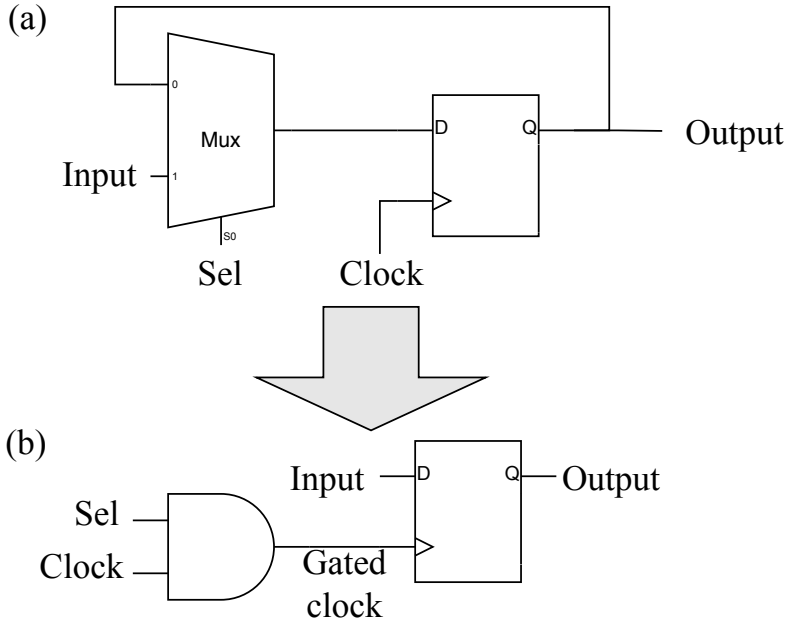
The clock signal is used by the majority of circuit blocks in a synchronous design. Since the clock switches every cycle, it has a switching activity of 1, which implicates that the sequential parts of a circuit are a great source of power dissipation (refer to Equations 2.1 and 2.2). A popular technique to reduce dynamic power is the disruption of the clock signal from the idle parts of the circuit. Specifically, the power savings obtained with the clock gating technique come from the reduction to zero of the switching activity $P_{0\rightarrow1}$. Moreover, the power savings due to clock gating can be broken into three major components (PANDA et al., 2010; WU; PEDRAM; WU, 2000):

1. the power dissipated by the combinational circuits being fed by the gated sequential elements gets reduced once their input is stable (not switching);
2. the input clock of memory elements, like flip-flops, is not triggered, reducing these elements' dynamic power;
3. and, the power dissipated by the clock distribution network (or clock tree), which has its capacitive load reduced (with clock buffers) also reduces.

Clock gating techniques can be classified into fine or coarse-grained. In its simplest form, fine-grained implementations are usually automatically inserted by specialized tools during the design flow. Take the circuit in Figure 2.5(a), for example. Once the enable signal is identified (*sel* in the example), it is possible to transform the circuit to a clock-gated one. The resulting circuit in Figure 2.5(b) is a possible solution that keeps the functionality of its predecessor, but dissipates lower dynamic power.

In coarse-grained clock gating, or architectural level clock gating, clock signals of entire circuit blocks are gated altogether. At higher levels of the design hierarchy, automated insertion of the clock-gate circuitry becomes prohibitive (PANDA et al., 2010). Usually, at the architectural level, clock gating is manually implemented by the designer in the RTL (register transfer level). However, the gains can be much higher when bigger portions of the circuit are clock-gated.

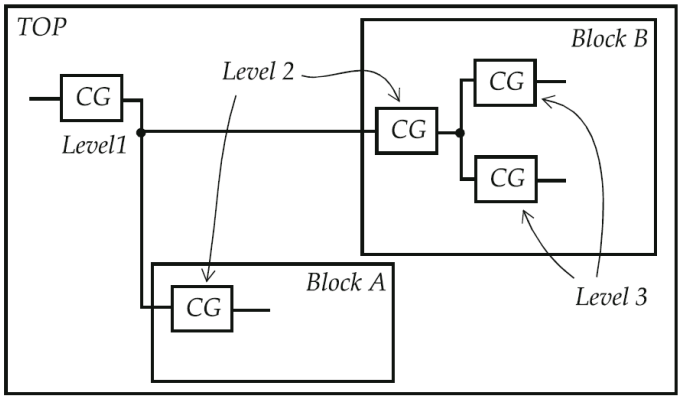
Figure 2.5: Fine-grained clock gating.



Source: the author.

Architectural clock gating can also be implemented in a hierarchical manner. Usually, there is a top-level clock feeding the multiple clock gates, which are responsible for their blocks (or even feed another level of clock gating). An example of a hierarchical clock gating is depicted in Figure 2.6. At the top level, the main clock signal that, besides feeding the circuit elements in its level, is the source for the clocks under the next level. The pattern could continue indefinitely and is attributed to the designer to keep in mind the issues that this method could give rise to. For example, the overheads created by simultaneous clock gating large groups of blocks must be assessed in order to avoid glitches, additional clock skew, and di/dt problems (i.e., large current swings in the power rail when the clock signal is reestablished for a large block).

Figure 2.6: Architectural clock gating.



Source: (CHADHA; BHASKER, 2012).

An extensive number of designs that use some form of clock gating as means to

save power have been proposed. For example, the Intel low-power processor, XScale, uses clock gating as one way to save power (CLARK et al., 2001). Besides an extensive use of Dynamic Voltage and Frequency Scaling (DVFS), the XScale employs clock gating in three architectural levels. First, it employs a top-level clock gating that freezes all core in the *idle mode*. Second, XScale implements a global clock level that has 83 enable signals spread across the core. And, at a final level, XScale uses clock gating at its LCB units (Local Clock Buffers, or LCB, are units responsible for clock distribution at the design's smallest block level, at least five latches).

Another commercial design that uses clock gating is the IBM Power5 processor (CLABES et al., 2004). While there are enable signals for both global and local clock gating, in the IBM Power5 processor, the logic generating the clock interruptions are taken locally. It is constituting, in fact, a fine-grained clock gating approach. Savings of more than 25% in the processor switching power are reported.

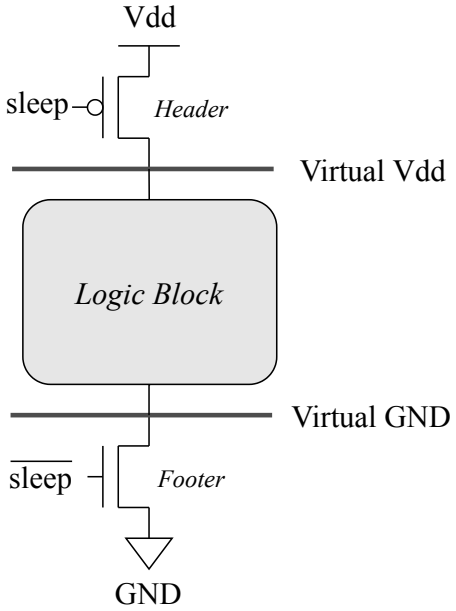
2.2.3 Power Gating based techniques

Despite the savings in dynamic power achieved by clock gating, it does not prevent static power dissipation from occurring even when the circuit is clock gated. For early CMOS technologies reducing dynamic power was sufficient. However, as it was already seen in Subsection 2.2.1, for more recent technologies, static power is responsible for a significant part of the total power dissipation. A widely adopted alternative to reduce power dissipation due to leakage is power gating. Essentially, the power gating cuts-off power supply of non-active parts of the circuit. To that end, power switches (in the form of sleep PMOS and NMOS transistors) and additional circuitry for generating control signals are used.

In a design that makes use of power gating, the power network is split into two parts: the always-on and the virtual power rails. When PMOS headers are inserted, a *virtual Vdd* is created isolating the gated block from the always-on Vdd. In a similar manner, when NMOS footers are used, a *virtual ground* is created (Figure 2.7).

One of the big issues in the implementation of power gated circuits is the sleep transistor sizing. At the same time that the sleep transistors (header and/or footer) must cope with the switching current drained by the block they are feeding, the sleep transistors are also a source of static power dissipation. Also, as those transistors are, typically, large, they produce high slew rates - requiring long time intervals for being charged and

Figure 2.7: Power gate implementation schematic.



Source: the author.

discharged. However large the sleep transistors are, the savings in static power due to power gating come from the fact that, collectively, the effective transistor length of the logic power being gated is larger and consumes more static power.

The effectiveness of the power gating is the result of the various aspects, such as correct transistor sizing, control logic behind the sleep signals (controlling the header and footer transistors), power gate topology, and controller design. The following discussion will address those central issues.

Power Gating Granularity. Two approaches can be taken for including power gate in a design: fine or coarse-grained power gating.

For fine-grained power gating, the sleep transistor is included inside the standard cell. It facilitates the power gate implementation since EDA tools can handle the additional circuitry, and the power gating overheads can be characterized together with the standard cells by the traditional design flow. Nevertheless, the logic required to control the massive number of sleep transistors makes fine-grained power gate expansive in terms of area (sleep transistors takes usually $2\times$ to $4\times$ the area of the original cell (PAL, 2014)).

On the other hand, in the coarser approach, the circuit is divided into blocks or modules, making it possible to share the virtual supply network by groups of cells. The area overhead is smaller since fewer sleep transistors are required, and is less sensitive to process variations (PANDA et al., 2010).

Overall, due to the smaller area overhead, the coarse-grained power gate is preferred over the fine-grained. Additionally, a coarse-grained power gate has been used

as means for increasing the adaptability of designs, increasing their power (and energy) efficiency. For example, the simultaneous multithread (SMT) processor, Power9, can power gate half of its execution units for its microarchitecture to sequential workloads (SADASIVAM et al., 2017). At a coarser grain power gate, the authors in (LEVERICH et al., 2009) advocate in favor of the per-core power gate (PCPG) in multicore systems. Leverich et al. (2009) show that power gate can be beneficial even for traditional general-purpose processors used in datacenters. Such modern systems usually employ DVFS as the primary technique for power saving. However, the fact that DVFS is commonly applied to all cores uniformly does not allow a more precise adaptation to the workload at hand, is added to the observed low utilization of cores in a datacenter (ranging from 10 to 50% (BARROSO; HÖLZLE, 2007)). It is making the use of PCPG a great alternative to increase a system's adaptability. Leverich et al. (2009) report savings in energy consumption of up to 40% with no significant performance losses due to PCPG (results that are 30% better than due to DVFS-only). Additionally, as DVFS and power gate are orthogonal techniques, energy savings can achieve up to 60% when both techniques are used.

Power Gating Topology. Closely related to the granularity, the power gate topology is another crucial aspect. According to Pal (2014), there are three categories of power gate topology: Global, local, and switch topologies.

A global topology refers to power gate implementations that share the virtual supply network across all logic blocks (applied only to coarse-grained power gate). Effectively, it incurs in a single *power domain*[1]. The second topology, local, is also used for a coarse-grained power gate. However, this arrangement results in a set of multiple power domains. It is implemented by segmenting the virtual supply networks at each sleep transistor. Hence, in a local topology, each sleep signal independently controls the state of specific logic blocks. Finally, the switch topology is particular to fine-grained power gate, where each cell has its own sleep transistor.

State Retention for Power Gating. Some designs may employ some form of state retention to recover from sleep modes in an "unmodified" state (not in reset state). The system state may refer to the state of flip-flops or contents of memory modules. Generally, retention mechanisms can be categorized into four approaches (CHADHA; BHASKER, 2012):

[1] A power domain is a group of logic blocks, or cells, that share a common sleep signal and virtual supply network.

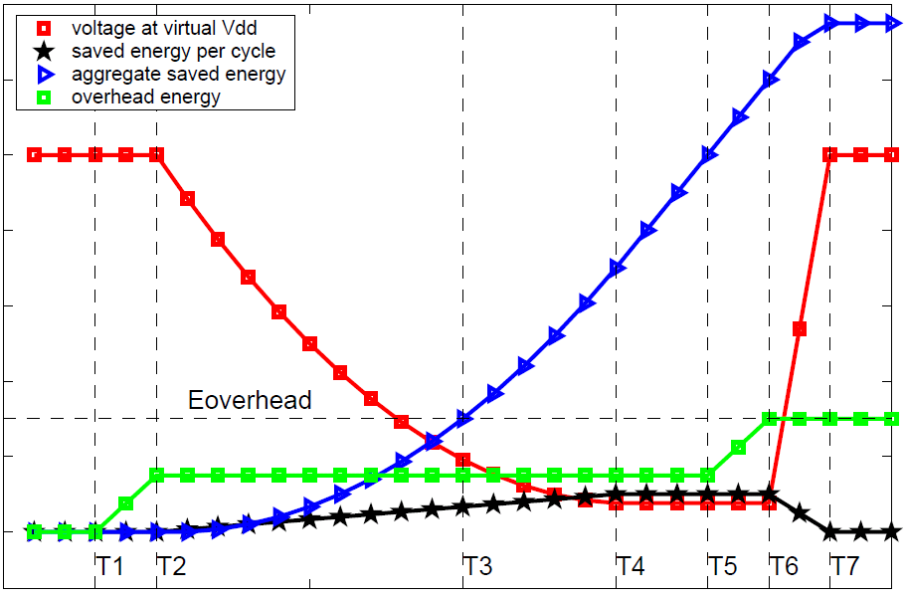
1. *System state saved externally.* Before powering down the logic block, its state is scanned via scan chains and the content saved to external memory. When recovering from the sleep mode, the contents are shifted back from the external memory to the flip-flops.
2. *Retention cells.* A logic block can employ retention cells, flip-flops or latches, for keeping their states throughout the sleep mode. The retention cells have a dual-rail power supply, connecting the cell to both virtual and always-on power networks. Hence, the designer may select some memories elements in the circuit to be kept on during sleep mode, retaining their state.
3. *Memory retention.* This approach is similar to the one used by retention cells. However, it employs dual-rail memories, so their contents are kept even if the blocks they are inserted in get power gated.

Power Gating Controllers. An essential aspect of power gating physical implementations is controlling the turning on and off the logic blocks. The in-rush current caused by multiple blocks being powered up together can cause severe damages to the circuit. Consequently, some orchestration has to be used when large portions of the circuit are entering and exiting sleep modes. The most common approach is to daisy-chain the sleep signals to the headers and footers transistors. The main result of daisy-chaining sleep signals is that between the sleep signal assertion and the actual power-up of a logic block, some additional time is taken to ensure safe operation.

Power Gating Operation. In (HU et al., 2004), the authors detailed a time-based approach for power gating execution units of a processor. Hu et al. (2004) explain the time characteristics involved in activating and deactivating parts of a design via power gating. Figure 2.8 summarizes the process.

The interval in Figure 2.8 starts at $t = 0$ with the header transistor fully charged (high virtual Vdd). After some time passes, the control circuit makes a decision to power gate the block. Hence, in $T1$, the sleep signal is brought low, and the overhead energy (green curve) starts to rise due to the energy consumed by the signal propagating up to the sleep transistor gate. Only at $T2$, the sleep signal is delivered at the sleep transistor gate, and it starts the switching-off. As soon as the virtual Vdd (red curve) starts the decline, the leakage current of power gated block also starts to go down (rising the aggregate energy saved - blue curve). The full discharge time is the $[T2, T4]$ interval. However, as mentioned earlier, sleep transistors present leakage current themselves and, then, the virtual Vdd does not, necessarily, get to zero Volts. Now, the block under the sleep transistor

Figure 2.8: Power gating cycle.



Source: (HU et al., 2004)

is said to be power gated, or in sleep mode. When the control logic detects an upcoming busy state, in $T5$, the sleep signal is de-asserted. From $T5$ to $T6$ the sleep signal propagates trough the power control network until the header transistor gate is achieved. When that happens, $T6$, the header starts to charge (switch-on) again. The charging up of the header also causes an energy overhead to occur. Only at $T7$ it power on is completed, virtual Vdd is at Vdd level, and the process can be repeated whenever the control logic finds another opportunity to power gate.

Additionally, Hu et al. (2004) propose the *break-even* point for operating power gated circuits. At this point, the aggregate energy savings equal the overheads. Precisely, the break-even point is given by $T3 - T2$. After the development of an analytical model, the authors state that the break-even point for a typical modern technology (with static to dynamic power ratio of 33/67) is ten cycles.

2.3 Adaptability for Energy Efficiency

2.3.1 Heterogeneous Computing

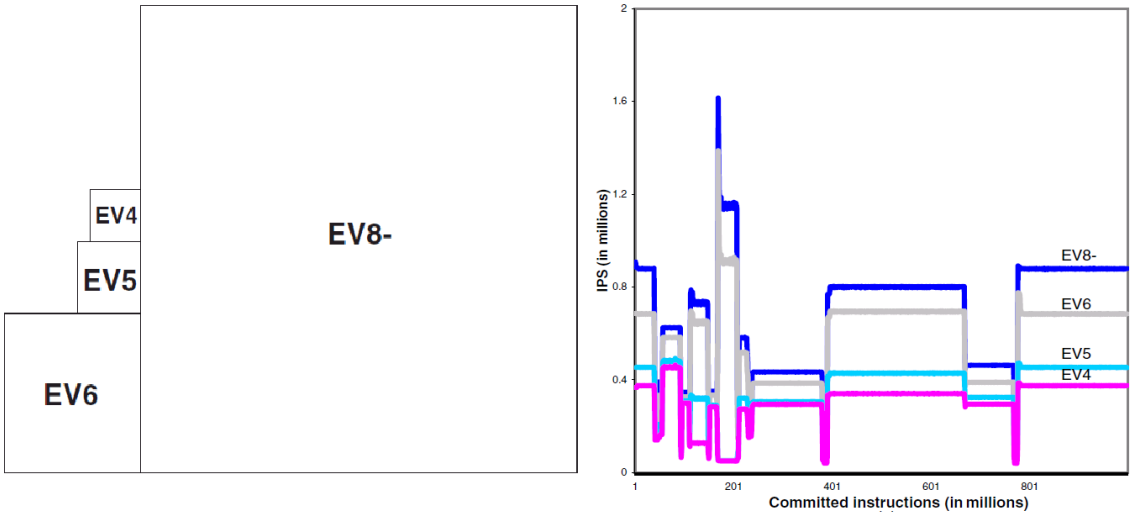
Initially, to support the applications' demands for high latency, high throughput, and devices' physical limitations, homogeneous multicore architectures were extensively employed (ADEGBIJA et al., 2018). Multicore architectures tackle the problem of per-

formance not only by providing means for TLP exploitation but also by exploiting ILP with superscalar and Out-of-Order (OoO) processors. However, when a homogeneous multicore system faces applications with diverse behavior, it bears the power inefficiency of executing every application (even the ones with low ILP opportunities) on processors optimized for low latency.

Only recently, seeking to improve energy efficiency, the paradigm has shifted to heterogeneous architectures that can better accommodate applications' diverse workload. Asymmetric, or heterogeneous systems, benefit from the differences among applications' needs, or even among phases of an application, to mix into the same system, computing resources with varying processing capabilities (consequently, with different area and power costs).

Kumar et al. (2003) is one of the first works to propose a heterogeneous system with a homogeneous instruction set architecture (ISA), enabling the transparent migration of tasks between cores. Four cores made up the heterogeneous system, each with distinct performance and computing capability. They were retrieved from the Alpha processor roadmap (two in-order and two out-of-order cores), namely Alpha 21064 (EV4), Alpha 21164 (EV5), Alpha 21264 (EV6), and a single-thread Alpha 21464 (EV8-). On the left side of Figure 2.9, the relative sizes of the four heterogeneous cores used by the authors is given. On its right side, Figure 2.9 presents the cores' performance (measured as committed instructions per second - IPS) when executing the *applu* benchmark from SPEC2000, evidencing the difference in performance and area size among the cores.

Figure 2.9: Relative Alpha cores sizes (a) and their performance, given by committed instructions per second, for the *applu* application (b).



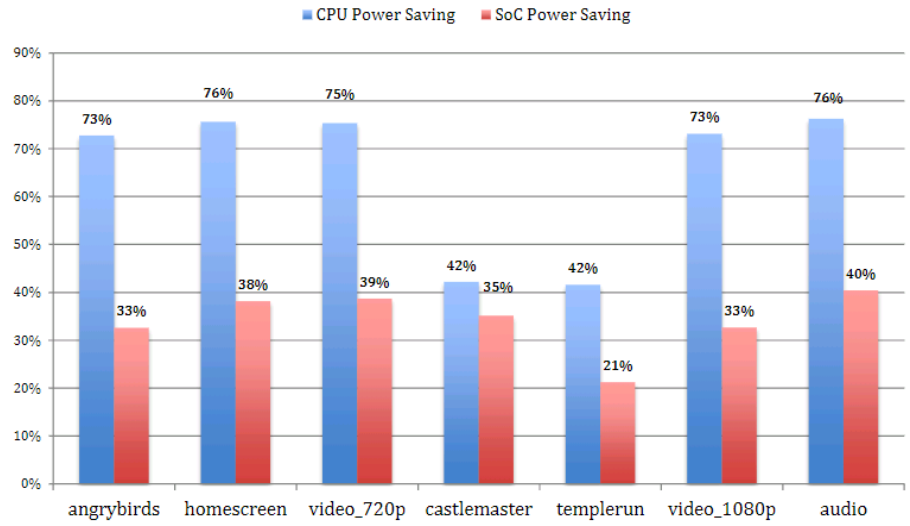
(a) (b)

Source: Adapted from (KUMAR et al., 2003).

Kumar et al. (2003) used two different objective functions that performed the online match of applications to cores. The first one targeted energy efficiency with a tight performance loss (up to 10% of the EV8- performance). When executed for 14 SPEC2000 benchmarks, an average reduction of 39% on energy consumption was observed with a small 3% performance degradation for the energy-targeting objective function. The second objective function was designed to optimize for energy-delay product with a looser performance constraints (up to 50% of the EV8- performance). With the energy-delay objective function, an average of nearly 3× improvement on energy consumption was observed with a more substantial performance penalty of 22%.

The big.LITTLE is a heterogeneous architecture with homogeneous ISA that became popular in industry (ARM, 2013). It is composed of two superscalar processors with distinct power requirements that are capable of exploring different levels of instruction parallelism. Big.LITTLE employs an online scheduler that is responsible for profiling the running applications and perform the switching of the cores to best accommodate the application needs. The efficiency achieved by the big.LITTLE system is shown in Figure 2.10. From it, we may also see the potential gains in migrating application between energy-efficient and performance-powerful cores in accordance with the application needs.

Figure 2.10: big.LITTLE (Cortex-A17 and Cortex-A7) power savings (y axis) when compared to a homogeneous system composed of only big processors (Cortex-A15) for a set of applications (x axis).



Source: (ARM, 2013).

The performance of heterogeneous systems is dependent on factors that range from software compatibility to task scheduling (MITTAL, 2016; AMALARETHINAM; JOSPHIN, 2015). Single-ISA systems, like the big.LITTLE, tackle on the problem of

software compatibility since the same code can run on any of the system's core. However, scheduling tasks appropriately has proved itself to be of a great challenge for computer architects.

The first approach to schedule applications in a heterogeneous system consists of statically scheduling tasks (e.g., Oh e Ha (1996)). However, a static schedule is not able to adapt to changes in the input set and becomes impractical due to its complexity as the number of applications rises. The second approach is the scheduling performed dynamically (e.g., (BECCHI; CROWLEY, 2006; CRAEYNEST et al., 2013)). The first issue in dynamic schedulers regards on how the applications are profiled. It is essential to the scheduler to acquire knowledge of the application's behavior so that it can match it to the appropriate core. One way to profile the application is through estimations made about the performance of each core type. Despite the fact that these estimations do not require the application to actually run on the cores, this approach is error-prone, and the scheduler becomes specific to the hardware it was initially designed for. Another way to profile applications is to set applications to run on each core type and profile their behavior. Nonetheless, it incurs in the obvious issue of scalability since it may present high overheads due to profiling that can take most of the lifetime of a short-lived thread.

Furthermore, there is a cost caused by task migration. For example, Pricopi et al. (2013) found that the migration latency in ARM's big.LITTLE architecture is 3.75ms when migrating a thread from the big (Cortex-A15) core to the little (Cortex-A7) and 2.10ms when migrating from the little to the big core. Also, there are overheads originating from cache warm-up and flushing and other state variables, restricting the scheduling of tasks in a finer grain.

2.3.2 Reconfigurable Architectures

Usually, the execution of algorithms can be based on two main methods. In the first method, ASIC devices are tailored for specific computations as they rely on lengthy and costly design flows. By customizing the hardware, designers can achieve high levels of performance and energy efficiency. However, a solution based on ASIC devices presents a serious drawback: they cannot be changed after fabrication. This implies that whenever the algorithm or computation, which the device was built to perform, requires modifications, the substitution of all deployed devices must be done. The second method involves software-programmed processors that can execute an endless number of algo-

rithms since they support a defined set of generic instructions for execution. Nonetheless, the flexibility brought by software comes with a highly complex hardware infrastructure, incurring a significant penalty on the performance and energy efficiency of GPPs.

Reconfigurable Architectures (RAs) offer a compromise between efficiency (as found in ASIC devices) and the flexibility of software-programmed processors (COMPTON; HAUCK, 2002; BECK; CARRO, 2010). RAs (or reconfigurable accelerators) contain an array of computational elements that are interconnected by reconfigurable fabrics. As a result, RAs can potentially achieve performance higher than solutions based only on GPP while still providing some flexibility. Systems that couple RAs to GPPs in order to achieve some acceleration are called Reconfigurable-Instruction-Set Processors (RISP) (BARAT; LAUWEREINS, 2000).

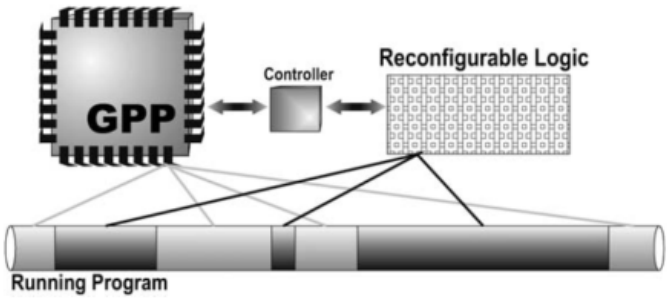
As shown in Figure 2.11(a), instructions can be executed either by the GPP or by the reconfigurable logic in a RISP. A hardware controller arbitrates what traces of instructions are offloaded as well as performs communication and synchronization tasks. Generally, the first step required for code execution in reconfigurable logic is the identification of traces of instructions (step 1 in Figure 2.11(b)) with high acceleration opportunities (hot spots) in the application. After hot spots are flagged, their instructions are transformed (or translated) to execution on the reconfigurable logic (step 2). Once the hot spot is transformed, it is ready for execution. To that end, the accelerator fabric is configured accordingly to the configuration at hand (step 3), and its input data is fetched (step 4). Now, the configuration is ready to be executed (step 5). Finally, the results are written back to the GPP (step 6).

2.3.2.1 Classification

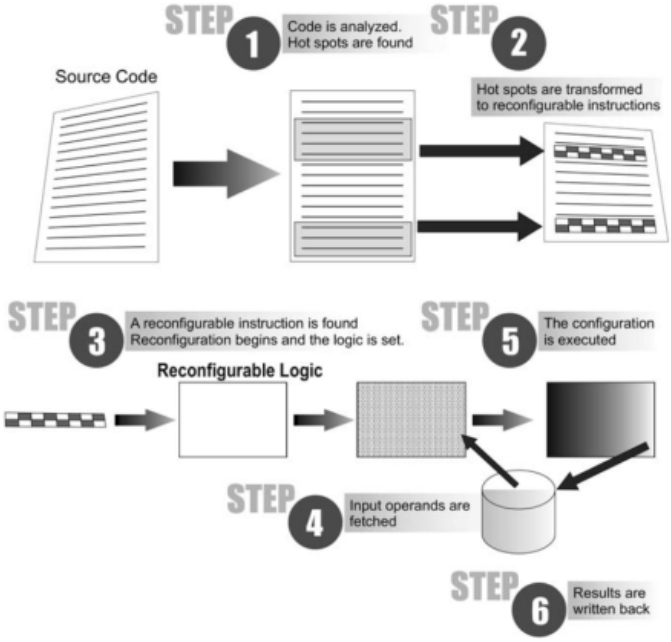
In the field of reconfigurable computing, there is no consensus for the classification of reconfigurable architectures (BARAT; LAUWEREINS, 2000; COMPTON; HAUCK, 2002; THEODORIDIS; SOUDRIS; VASSILIADIS, 2007; WIJTVLIET; WAEIJEN; CORPORAAL, 2016; LIU et al., 2019). Inevitably, the classification proposed in (BECK; LISBôA; CARRO, 2013) will be adopted in this work. According to the authors, reconfigurable systems can be classified regarding *code analysis and transformation*, *coupling*, *granularity*, and *reconfigurability*.

Code Analysis and Transformation. This subject concerns the search and transformation of *hot spots* in the application code for execution in the reconfigurable fabric (steps 1 and 2 in Figure 2.11). The code analysis can be done at the executable or source

Figure 2.11: A RISP overview.
(a) Basic principle.



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



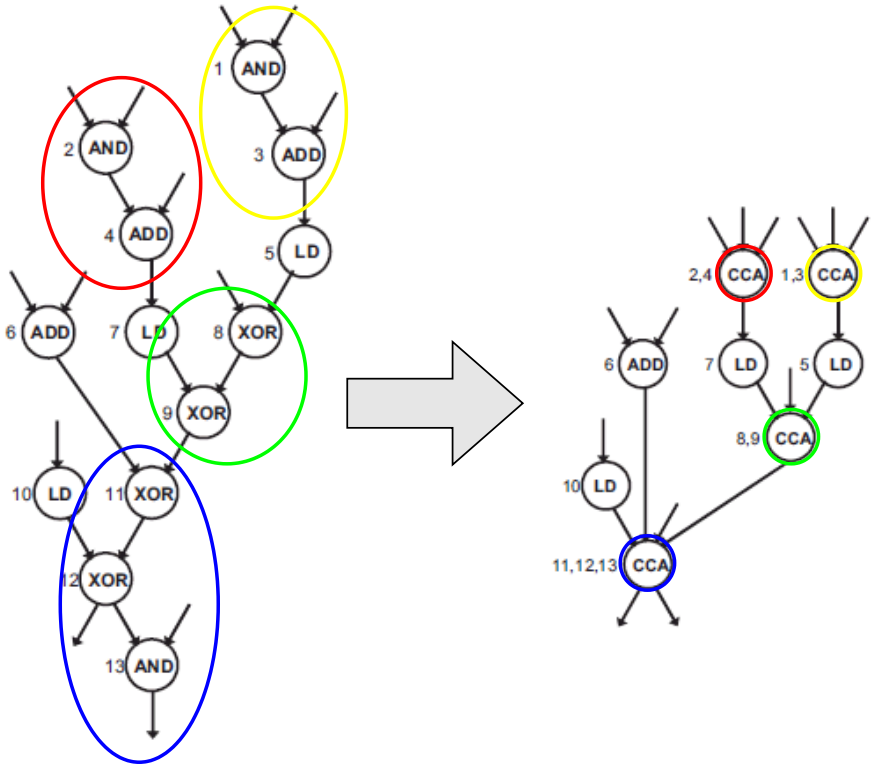
Source: (BECK; CARRO, 2010)

code, or even analyzing execution traces of applications. At any level, hot spots can be automatically identified by specific tools or manually by the designer.

Once these hot spots are flagged, a transformation takes place for replacing them with reconfigurable instructions. Code transformation is usually highly dependent on the system it was designed for. When transforming a code region, issues like communication between host and reconfigurable fabric, reconfiguration overheads, memory accesses, and writing-back of results have to be addressed. For automated processes, some kind of Data Flow Graph (DFG) analyzer is customarily employed. For instance, the DFG at the left side of Figure 2.16 represents the application's code instructions (nodes) and their dependencies (edges). After the DFG is analyzed, instructions that can be grouped are identified and transformed into reconfigurable instructions, resulting in the DFG at the

right side of Figure 2.16.

Figure 2.12: Original DFG (left) and transformed DFG (right).



Source: adapted from (CLARK et al., 2004).

Analysis and transformation can be further divided into *static* and *dynamic*. In static approaches, all work is done at compile time. It makes the analysis and transformation simpler to implement since it can be based on code annotation and modifications to the assembler, for example. However, some critical information is left aside when static methods are used. Only dynamic methods, which have access to information such as loop bounds and input size, can have a better assessment of an application's hot spots. Besides, dynamic analysis and transformation do not require re-compilation, and can naturally adapt to the new, or already-deployed, workloads.

An important aspect that may help with the adoption of reconfigurable architectures is binary compatibility. As can be seen in the history of commercial systems, maintaining compatibility with legacy systems is essential to the success in the deployment of new systems and architectures. In the context of dynamic code analysis and transformation, binary translation is a technique that enables transparent portability of code between different Instruction Set Architectures (ISA) (ALTMAN; KAELI; SHEFFER, 2000; ALTMAN et al., 2001). For example, when applied to a RISP machine, binary translators are capable of porting the code form the host processor to the reconfigurable fabric in a dynamic and transparent manner. When referring to binary translators in this

work, it will be assumed a dynamic translator. According to Altman, Kaeli e Sheffer (2000), binary translators can be divided into three types: emulators, static translators, and dynamic translators. The letter besides the translation of the code, saves the resulting code for further reuse (reducing the translations overheads).

Coupling. Reconfigurable units can be coupled to the main processor in different ways. Generally, the reconfigurable unit can be attached to the system through a secondary I/O bus, as a co-processor, or as a functional unit (inside the main processor). With the exception of the allocation as a functional unit, which is also called tightly coupling, the other two coupling types are also said to be loosely coupled. Naturally, the coupling has an impact on the performance and complexity of the reconfigurable hardware. For instance, loosely coupled accelerators are not constrained, as tightly coupled accelerators, in their area. However, for loosely coupled accelerators, communication overheads may become a significant impediment for achieving higher performance levels. For example, access to the register file, and other modules of the processor microarchitecture, are only granted to tightly coupled reconfigurable units (sometimes called Reconfigurable Functional Units, or RFU).

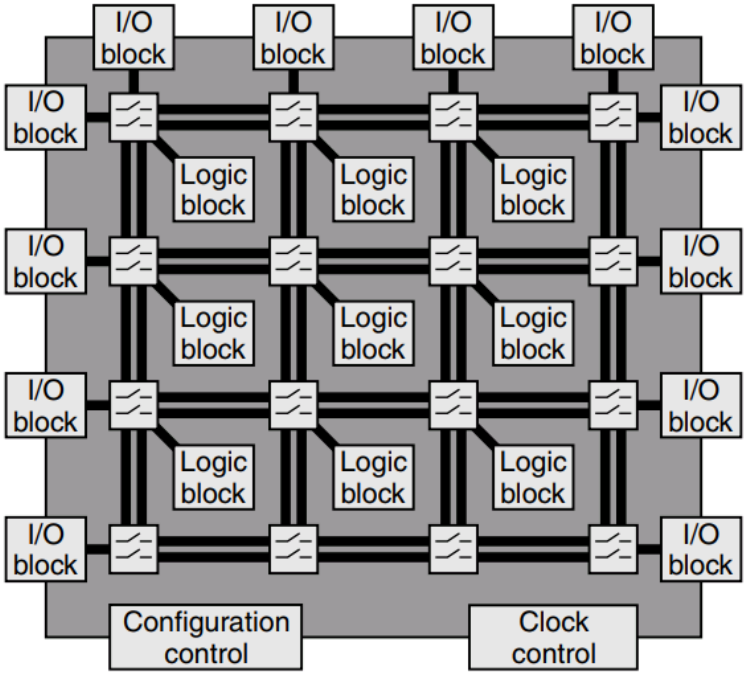
Granularity. The level that data is manipulated by the reconfigurable unit gives the granularity of the architecture.

Fine-grained reconfigurable architectures manipulate data at the lowest levels (i.e., at the bit-level), which is the reason behind the high flexibility. In such systems, the set of Processing Elements (PEs) implement one bit functions, and are interconnected by some network. A well-known example of fine-grained reconfigurable architecture is the Field Programmable Gate Arrays or FPGAs. The principle behind FPGAs is a generic circuitry that can be configured to execute any function for any specific application (digital functions). To achieve its programmability, the FPGA distributes several logic blocks across a programmable interconnection fabric surrounded by input and output blocks for interfacing the FPGA core with external devices.

Figure 2.13 shows the arrangement of the basic blocks inside an FPGA. In current FPGAs, several lookup tables (LUTs) are grouped into larger *logic blocks*, also called Configurable Logic Blocks (CLBs). These modules provide faster internal connections between LUTs than those provided by connections across the FPGA network. Also, they may house flip-flops, shift registers, distributed RAMs, multiplexers, and arithmetic operators. The CLBs are disposed of in a mesh-like programmable network. Recent FPGAs have been using SRAM to program the interconnection and logic. In SRAM-based

FPGAs, the logic function of a block or the state of interconnection is controlled by programmed SRAM cells (BAILEY, 2007).

Figure 2.13: The basic architecture of an FPGA.



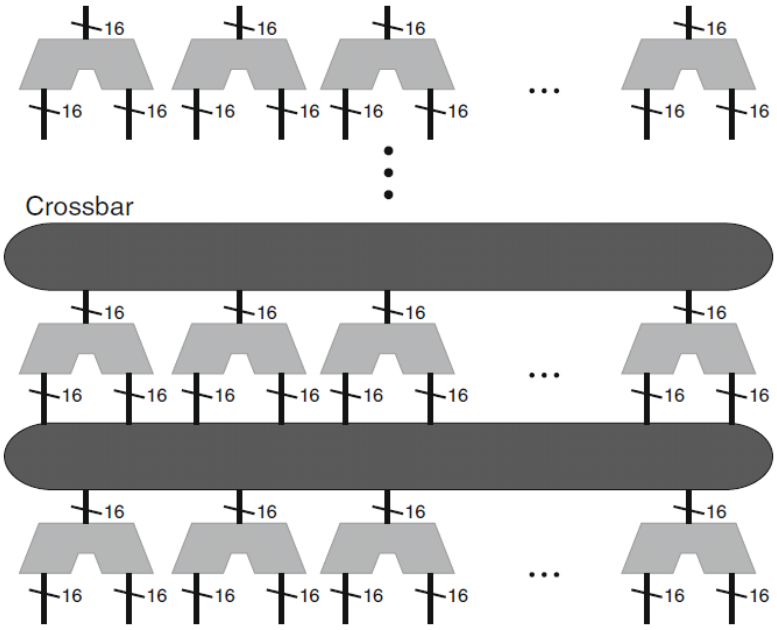
Source: (BAILEY, 2011)

Coarse-grained reconfigurable architectures, on the other hand, make use of PEs that implement operations at word-level and configurable interconnections that gives some degree of flexibility to the system. Usually, Coarse Grained Reconfigurable Architectures (CGRAs) are implemented using off-the-shelf functional units and multiplexers as well as crossbars for interconnection. Figure 2.14 depicts an example of such architectures. It implements 16-bit long functional units interconnected by a set of crossbars.

When comparing fine and coarse granularities, the authors in (THEODORIDIS; SOUDRIS; VASSILIADIS, 2007) point out some aspects that favor coarse-grained reconfigurable architectures:

- *Small configuration contexts*. Due to the coarser configuration grain, CGRAs require less context to configure the PEs and interconnection.
- *Reduced configuration time*. For the same reason that makes the size of the configuration context smaller in CGRAs, the time required for configuring the fabric is also reduced.
- *Reduced context memory size*. Since the configuration contexts are smaller when compared to RA of finer grains, the memory used for storing configurations is also smaller.

Figure 2.14: A coarse-grained reconfigurable array of functional units.



Source: (BECK; LISBôA; CARRO, 2013)

- *High performance and low power dissipation*. Hard-wired functional units tend to perform better than the same functions implemented with programmable logic at bit-level (e.g., LUTs). Also, many CGRA implementations do not use memory elements (flip-flops) inside the reconfigurable array. By being fully combinational, a huge source of power dissipation is eliminated.
- *Silicon area efficiency and reduced routing overhead*. When supporting fine-grained configuration, architectures are bound to use large interconnection networks, with high overheads, that cannot be optimized in the way networks that are required to communicate at coarser grains are.

Overall, CGRAs are favored for the acceleration of applications with a large number of hot spots. With multiple hot spots, the aspects mentioned above, like configuration context size and reconfiguration time become decisive in the system performance. Conversely, for applications that have few hot spots and do not require frequent reconfiguration, fine-grained reconfigurable architectures might present better results.

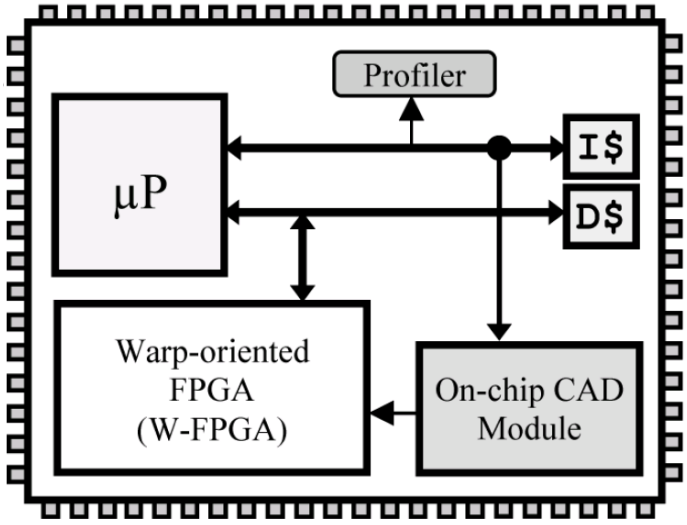
Reconfigurability. Reconfigurable architectures are differentiated accordingly to their capacity of reconfiguration. Architectures, where the configuration is possible only at startup, are not considered reconfigurable. Hence, only those architectures capable of adapting their fabric during the execution of an application, at runtime, are considered reconfigurable architectures.

2.3.2.2 Implementations

This subsection will first present the main works on general *dynamic* reconfigurable architectures. Later, it will also be shown works that apply resource or power management techniques to reconfigurable architectures.

The **Warp Processor** is one of the first attempts to unify techniques of dynamic optimization with reconfigurable computing (LYSECKY; VAHID, 2004; LYSECKY; STITT; VAHID, 2006; LYSECKY; VAHID, 2009). The proposed system consists of a System on Chip (SoC) that integrates a microprocessor, instruction and data caches, an embedded FPGA, a hardware-implemented profiler, and another microprocessor dedicated to the execution of a simplified CAD tool (Figure 2.15). The execution flow of the Warp Processor can be decomposed into five steps: *(i)* the application first executes on software only (over the main processor); *(ii)* the profiler determines the hot spots by listening to the addresses being requested to the instruction memory; *(iii)* the on-chip CAD synthesizes the circuit from the selected hot spots; *(iv)* the circuit is programmed in the FPGA and the hot spots in the application's binary code are replaced with instructions for partitioned execution; *(v)* application's hot spots execute in the FPGA.

Figure 2.15: Overview of the Warp Processor.

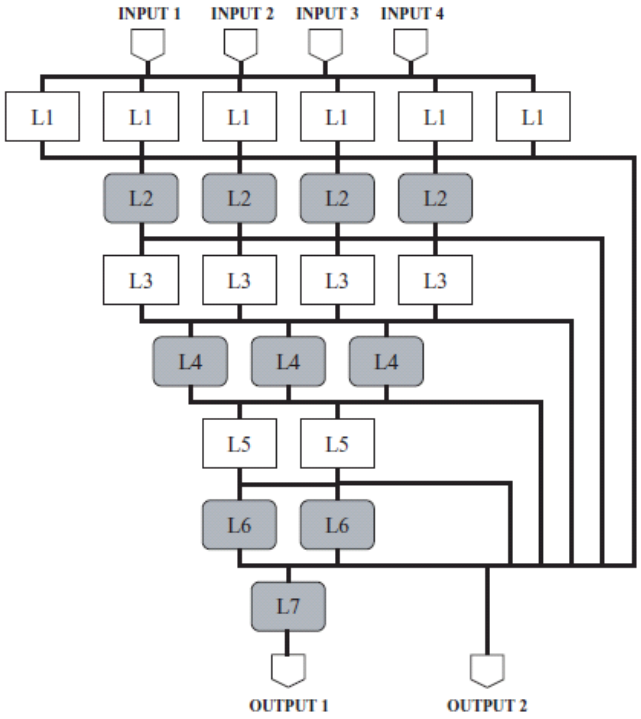


Source: (LYSECKY; STITT; VAHID, 2006)

The **CCA** (Configurable Compute Accelerator) architecture was proposed in (CLARK et al., 2004). The architecture tightly couples a CGRA to an ARM processor. The reconfigurable array is equipped with two types of functional units: one that is capable of performing both logical and addition/subtraction operations on 32-bit words, and another only capable of the logical and/or/xor/not operations. Also, the functional units in the same array level execute in parallel, and instructions with dependence inside the hot spot

can be mapped to subsequent levels of the array. As can be seen in Figure 2.16, the reconfigurable array was designed following a triangular shape. The triangular shape was selected after an empirical study conducted by the authors. In this experiment, they found that the average shape of configurations being mapped to the array followed a triangular shape (i.e., mapping more instructions in the initial levels). In its dynamic operation, the CCA uses a trace cache to profile the retired instructions. Based on that trace, it is possible to build the DFG.

Figure 2.16: Schematic of the CCA's reconfigurable array.



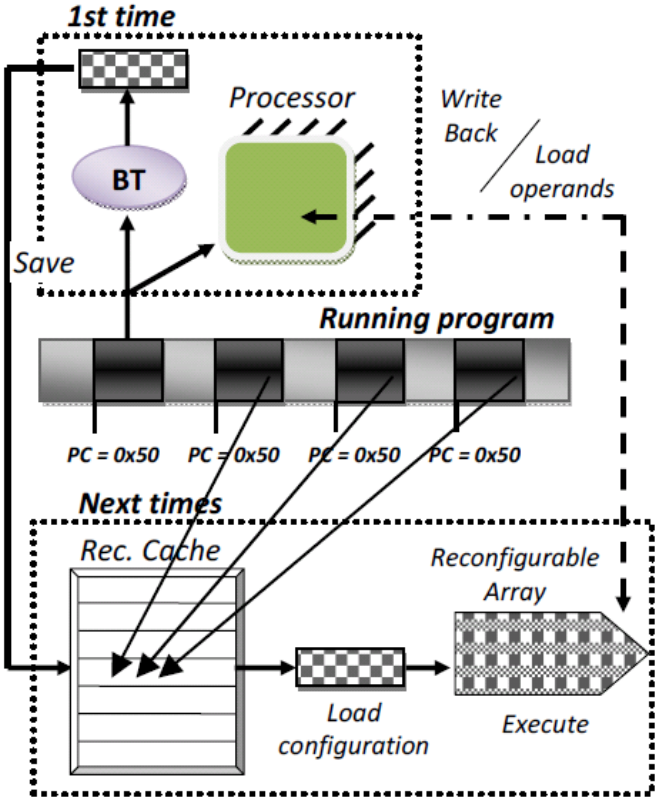
Source: (CLARK et al., 2004)

In the **RISPP** (Rotating Instruction Set Processing Platform) architecture (BAUER et al., 2007), an FPGA is used for the acceleration of hot spots. Bauer et al. proposed the concept of Atoms and Molecules (that implement the so-called Special Instructions, based on the hot spots synthesized at compile time). These components can be pre-allocated in the FPGA, so a runtime manager can decide whether to compute a hot spot in the FPGA or leave the execution for the main processor. Also, the authors point out that partial reconfiguration could be used, so the system's adaptability could be even increased.

The **DIM** (Dynamic Instruction Merging) reconfigurable system (BECK; CARRO, 2007; BECK et al., 2008; BECK; CARRO, 2009) is proposed by coupling a CGRA, with configurations generated by a hardware-implemented binary translator, to a MIPS processor. The configurations are extracted from the application's hot spots by a fully transparent

and dynamic binary translation mechanism when they are first executed. The hot spots, or basic blocks, already translated are saved to a cache where they are indexed by their first instruction program counter (PC), so future fetching of the configurations are possible. Figure 2.17 details the approach proposed for execution in the reconfigurable system.

Figure 2.17: The DIM reconfigurable system.

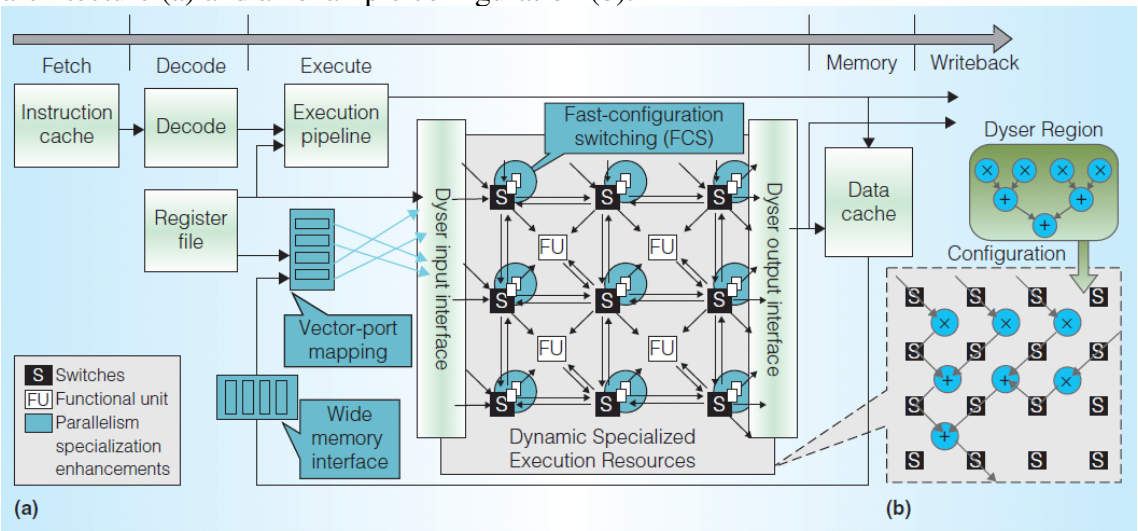


Source: (BECK; CARRO, 2007)

Govindaraju et al. (2012) presented the **DySER** (Dynamically Specialized Execution Resources) architecture. It consists of a coarse-grained reconfigurable fabric of heterogeneous functional units, which, as claimed by the authors, entitles *functionality specialization* to the architecture. On the other hand, *parallelism specialization* is granted to architectures that employ homogeneous resources disposed on wide and independent interconnects (such as vector processors and GPUs). Govindaraju et al. suggest that the DySER is a step towards the unification of functionality and parallelism specialization. To attend the specialization, the DySER's compiler synthesizes data paths in the reconfigurable fabric that are specific to each application's phase. Whereas, the parallelization is achieved by employing vectorization techniques only possible via the reconfigurable array.

The system overview is shown in Figure 2.18. The reconfigurable fabric is tightly coupled to a general-purpose processor that acts as a load/store unit feeding the DySER.

Figure 2.18: Overview of the DySER (Dynamically Specialized Execution Resources) architecture (a) and an example configuration (b).



Source: (GOVINDARAJU et al., 2012)

As for the reconfigurable fabric, the heterogeneous functional units are connected by simple switches. The interconnection is implemented in a credit-based technique forming a circuit-switched network. Data propagates only between neighboring functional units. Also, the ISA was extended with five instructions for configuring the DySER execution units and interfacing with the register file and external memory. Later, the **DORA** (Dynamic Optimizer for Reconfigurable Architectures) system extended DySER by replacing the offline generation of configurations with a dynamic binary translation mechanism (WATKINS; NOWATZKI; CARNO, 2016).

The work in (LIU et al., 2015) couples a reconfigurable fabric to an OoO processor. **DynaSpAM** (Dynamic Spatial Architecture Mapping) leverages some already existing structures (branch predictor and issue unit) that enable out-of-order instruction execution in the hardware dynamically generating configurations. Specifically, as the issue unit schedules instructions to the OoO functional units, DynaSpAM simultaneously maps the instruction to an available reconfigurable PE along with the necessary routing. Once the configuration is generated, it is saved in a configuration cache for future reuse. When compared to a traditional 8-issue OoO processor, DynaSpAM achieves a geomean speedup of $1.42\times$ and a geomean energy reduction of 23.9%.

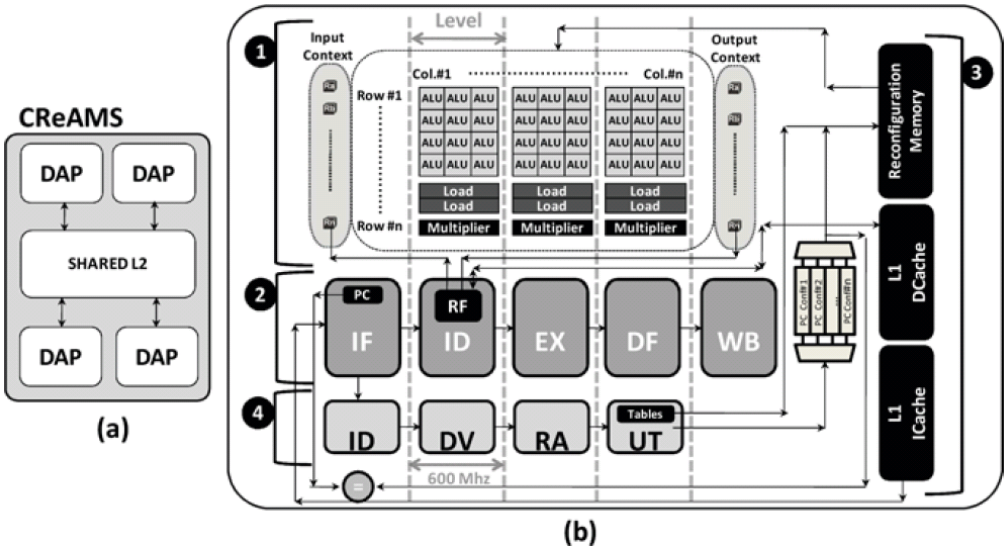
As became clear in studies like (WALL, 1991) and (OLUKOTUN et al., 1996), instruction level parallelism (or ILP) alone cannot scale performance gains indefinitely. Also, the software heterogeneity, as shown in Section 2.1, present in modern workloads, aggravates the problem even more. The following works approach the thread level par-

allelism (TLP), via multicore implementations, in combination with the fine-grained ILP exploration enabled by their reconfigurable fabrics.

Watkins e Albonesi (2010) presented the **ReMAP** (Reconfigurable Multicore Acceleration and Parallelization), a general-purpose fine-grained reconfigurable architecture that executes both sequential and parallel workloads. To enable the acceleration, the system requires an offline mapping procedure that may allocate the reconfigurable fabric to single or multiple threads. The architecture supports fine-grained point-to-point communication for the executing threads, enabling pipeline parallelization and barrier synchronization.

Rutzig, Beck e Carro (2011) proposed the **CReAMS** (Custom Reconfigurable Arrays for Multiprocessor System) architecture. It is composed of multiple Dynamic Adaptive Processors (DAP), consisting of a coarse-grained reconfigurable path connected to the Dynamic Detection Hardware (DDH), a binary translator, that transparently configures the reconfigurable datapath with instructions fetched by the main, SparcV8, processor. Since multiple DAPs are instantiated together sharing an L2 cache (as detailed in Figure 2.19), the CReAMS system is capable of exploring both thread level and instruction level parallelisms.

Figure 2.19: The Custom Reconfigurable Arrays for Multiprocessor System (or CReAMS) in (a) and its Dynamic Adaptive Processor (DAP) in (b).



Source: (RUTZIG; BECK; CARRO, 2011)

The **HARTMP** (Reconfigurable and Transparent Multicore Processing) is a work that followed the CReAMS architecture by providing a heterogeneous reconfigurable multicore organization (SOUZA et al., 2016). As explained earlier in Subsection 2.3.1, the use of heterogeneous organizations present an energy-efficient alternative to homoge-

neous architectures that cannot match their resources to the diverse applications' needs. As the DAPs in the HARTMP have different performance potentials, it is necessary the use of a predictive scheduler to match the threads' demands to the accelerator.

Having presented some of the mains works on dynamic reconfigurable architectures implemented in both single and multicore systems, we can now focus on the works that involve techniques targeting either power-saving or resource management to increase aspects like performance and power or energy efficiency of reconfigurable architectures.

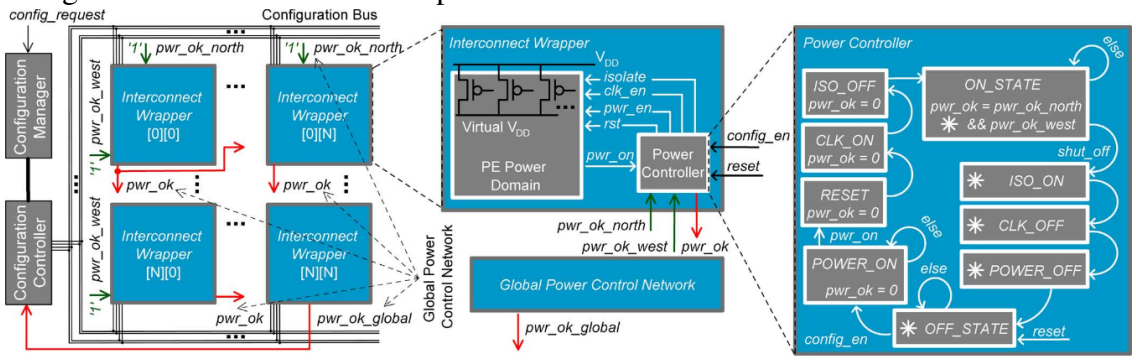
The studies in (LAMBRECHTS; RAGHAVAN; JAYAPALA, 2005) and (LAMBRECHTS; RAGHAVAN; JAYAPALA, 2005) motivated the authors of the ADRES reconfigurable system (MEI et al., 2003) to propose a new architecture, based on the previous one, but featuring architectural changes for increasing performance and energy efficiency. The ***Enhanced*** **ADRES** (BOUWENS et al., 2008) includes operand isolation and clock gating optimizations, so an energy reduction of 50% was achieved when executing classical digital signal processing algorithms. The ADRES framework relies on code analysis and transformations made at compile time for mapping basic blocks to a tightly coupled CGRA. Specifically, clock gates were automatically added to the CGRA's register file by CAD tools, reducing the power dissipated by the registers from 50 to 80%.

One of the first attempts to reduce leakage power in CGRA architectures was proposed by Saito et al. (2008). The **MuCCRA-2.32b** extends its predecessor architecture, the MuCCRA (AMANO et al., 2007), by providing a fine-grained power gating mechanism. Only targeting the PE's Arithmetic Logic Unit (ALU) and Shift and Mask Unit (SFU) execution units, the sleep controls are concatenated into the CGRA configuration word. As configurations are generated statically, it is possible to know in advance which execution units will be idle for each configuration. Also, two control modes are proposed: the *pair* control mode and the *unit individual* control mode. When running with *pair* control mode, the MuCCRA-2.32b powers-off both PE's units (ALU and SFU). In contrast, configurations can select individual units to power gate when the *unit individual* mode is active. The proposed method achieves reductions of up to 48% in leakage power.

Kissler et al. (2011) proposed a scalable many-domain power gating scheme for CGRA architectures depicted in Figure 2.20. The architecture implements power gating at the PE level (resulting in 24 independent power domains) controlled by a distributed power control network. Particularly, the status signals of all PEs propagate from west to east and north to south across the neighboring PEs up to an AND-tree that generates the *pwr_ok_global* signal. This signal informs the application whether the CGRA is ready

to execute configurations. Another advantage of propagating the status signal among the PEs is that it inherently power-up the PEs in a daisy-chain fashion. The authors also make use of EDA design tools for automatic insertion of hierarchical clock gating. However, the whole process relies on a static scheme. The programmer is responsible for assigning the PEs to a specific application, which will cause the controller to power gate the unassigned PEs. Hence, power, throughput, and latency trade-offs have to be assessed by the programmer at design time. Overall, a 60% reduction in the leakage power is reported for the edge detection application, which assigned a 2x2 subarray (from a total of 24 PEs).

Figure 2.20: Schematics of the power control network and controller state machine.

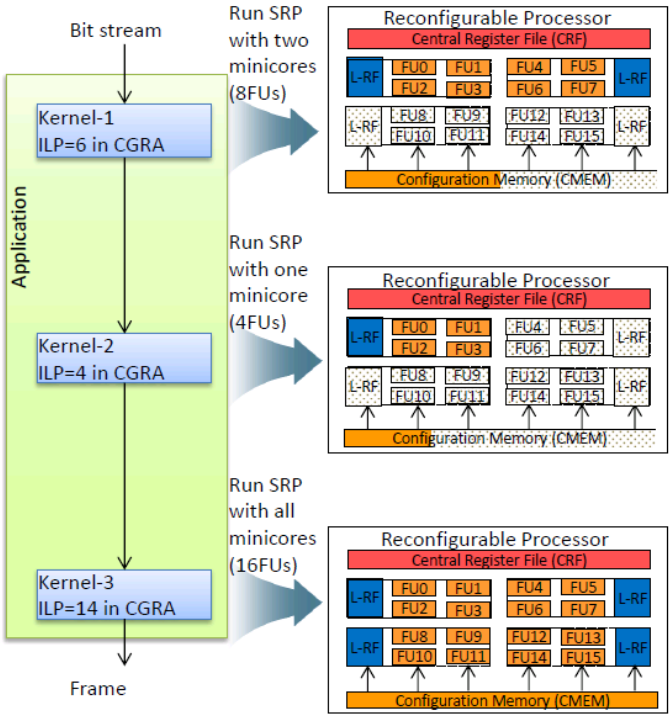


Source: (KISSLER et al., 2011)

In (MINISKAR et al., 2016), an architecture, based on the Samsung Reconfigurable Processor (SRP), makes use of a code annotation mechanism to power gate functional units in the reconfigurable array. Hence, it is in charge of the programmer to identify which parts of the program have low ILP that would lead to low utilization of the reconfigurable fabric in case no power gate is performed. For example, in Figure 2.21 an application is manually profiled by the programmer who identified three distinct kernels. After the programmer finds the ILP of each kernel, program directives in the application source code can be used to "instruct" the power gating. Also, the architecture can operate on both Very Large Instruction Word (VLIW) mode, in which maximum savings of 33% are achieved in power dissipation, or CGRA mode, where savings of up to 56% are reported.

There is also the work by Akbari et al. (2018) that aims reductions in the power dissipation of a CGRA system by using approximate functional units. The authors argue that the same class of applications that are the most amenable to CGRA execution, such as multimedia and digital signal processing, are also inherently error-resilient. Based on an energy consumption breakdown of an exact PE, the authors propose the replacement

Figure 2.21: ILP-oriented power gate of the reconfigurable array.



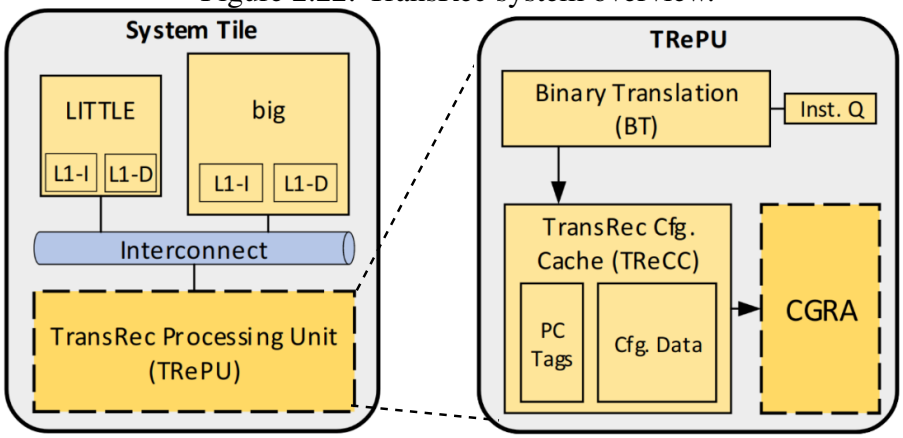
Source: (MINISKAR et al., 2016)

by their approximated counterparts, of the two most consuming operations, multiplication (corresponding to 43% of the PE's total energy consumption) and addition (16% of the total PE's energy). PEs in the **PX-CGRA** (Polymorphic Approximate CGRA) architecture are made up of mixed-accuracy ALUs connected to a switch box. The ALUs of varying accuracy can be allocated as independent units in the CGRA configuration words. However, the configuration of the approximate CGRA is not transparent to the user, who has to program the CGRA accuracy in the application code explicitly, and evaluate possible issues caused by approximation. Overall, the PX-CGRA achieves up to 45% improvement in energy-efficiency when compared with their fully exact counterpart (with a 35% output quality degradation).

The **TransRec** was proposed in (BRANDALERO et al., 2019). The system couples the acceleration provided by a CGRA to big.LITTLE-like architecture. In it, both *big* and *LITTLE* processors have access to the reconfigurable fabric through a shared bus. As can be seen in Figure 2.22, the TransRec Processing Unit (TRePU) is composed of three main modules: the CGRA, responsible for executing the reconfigurable datapaths, the Binary Translator, in charge of transparently transforming the instructions arriving through the *Inst. Queue* into CGRA configurations that are saved in the third module, the configuration cache, allowing further reuse whenever an already translated basic block reappears for execution. Additionally, Brandalero et al. (2019) make use of the DVFS technique to

improve the system's energy efficiency further. It achieves an energy efficiency of $1.59\times$ better when compared to execution on the *big* core, and performance improvements of $2.28\times$ and $1.32\times$ when compared to execution on the *LITTLE* and *big* core, respectively.

Figure 2.22: TransRec system overview.



Source: (BRANDALERO et al., 2019)

Das, Martin e Coussy (2019) propose CGRA acceleration for ultra low-power environments. The approach taken by the authors, however, targets power savings at the algorithm level where smaller CGRA configurations should be generated during code analysis and transformation. A reduction in the configuration size means that reconfigurable units require smaller caches. Hence, the authors tackle the reduction of an important source of power dissipation of dynamic reconfigurable systems. As the mapping of basic blocks to the reconfigurable fabric is performed by an offline tool, an exhaustive search can look among multiple mapping options for the one mapping that executes the specific basic block requiring the smallest amount of CGRA resources. Das, Martin e Coussy (2019) report $2.3\times$ gains in energy consumption while requiring $2\times$ less configuration memory when compared to traditional mappings.

2.4 Contributions to the State-of-the-Art

Following works like (LYSECKY; VAHID, 2004; BAUER et al., 2007), and (GOVINDARAJU et al., 2012), the architecture proposed here involves the use of a reconfigurable fabric dynamically configured for offloading execution from the main processor. However, those works have the significant drawback of requiring offline code generation for execution on the reconfigurable fabric. To overcome this issue, authors in (BECK et al., 2008; LIU et al., 2015) and (WATKINS; NOWATZKI; CARNO, 2016), have pro-

posed the use of dynamic mechanisms for generating reconfigurable instructions at runtime. Specifically, this work leverages the approach in (BECK et al., 2008), which uses a hardware-implemented binary translator responsible for the online transformation of the instructions in the application's code to configuration words enabling execution on the CGRA.

In the context of multicore reconfigurable architectures, this work differs from (RUTZIG; BECK; CARRO, 2011) that employs a set of CGRAs with homogeneous organizations that may lead to inefficiency due to applications with low or unbalanced ILP. On the other hand, a multicore with a heterogeneous set of CGRAs was already proposed in (SOUZA et al., 2016). However, such organization incurs on costs related to task scheduling and migration, which are avoided by our dynamic adaptation scheme, as will be detailed in the next chapter. Furthermore, many works have proposed improvements to the power or energy efficiency of CGRAs. Orthogonal to this work, the automatic insertion of clock gating by EDA tools can be used to take reductions in dynamic power dissipation even further, as done in (BOUWENS et al., 2008) and (KISSLER et al., 2011). On the works that make use of power gating, the authors in (AMANO et al., 2007) also employ a fine-grained power gating mechanism. However, their approach requires including in the configuration word information about which of the functional units must be power gated - information that is gathered offline by the programmer. The approach taken in this work explores a power gating mechanism similar to the one proposed in (KISSLER et al., 2011), which implements a scalable many-domain power gate across a CGRA. Nevertheless, the adaptation of the reconfigurable fabric to the application needs to be carried out manually by the programmer via code annotation. Code annotation-based power gating is also performed in (MINISKAR et al., 2016).

Regarding other orthogonal techniques for reducing power and energy consumption presented in this chapter, approximate functional units could be integrated into CGRA devices, as performed in (AKBARI et al., 2018). Additionally, more advanced techniques like DVFS, in addition to power gating, can also be used to reduce the power dissipated by CGRA's functional units, as shown in (BRANDALERO et al., 2019). Finally, the memory used for storing configuration words that is present in most CGRA works can also be the target of improvement as described in (DAS; MARTIN; COUSSY, 2019).

In summary, our work is the first one to incorporate dynamic and transparent adaptation of the reconfigurable fabric resources for keeping power dissipation under edge-tolerable levels. As will be explained later, the proposed approach is based on online

monitoring of the applications' performance requirements to match it to the CGRA resources. Moreover, the proposed approach is extended to a multicore architecture, where it is used to manage resources of multiple reconfigurable fabrics simultaneously.

3 A RESOURCE-AWARE MULTICORE CGRA ARCHITECTURE

This chapter presents the resource-aware multicore CGRA architecture proposed in this work. It is a multicore architecture, where each processor is coupled to a dedicated reconfigurable unit for accelerating data-dependent sequences of instructions, as further detailed in Section 3.1. The architecture's primary goal is to leverage the thread-level parallelism provided by the multiple cores and the elevated instruction-level parallelism achieved by reconfigurable accelerators. In order to reduce the power dissipation of the reconfigurable accelerators, a resource management technique that performs online profiling of the executing applications and the state of the system's multiple reconfigurable fabrics was developed. By selectively using power gate to adapt the available resources of each accelerator to the applications at hand, the efficiency of a heterogeneous organization is achieved without incurring costs, such as task migration. The proposed architecture is fully transparent to the user since it extends existing works on dynamic reconfigurable architectures based on the use of transparent hardware-implemented binary translation modules.

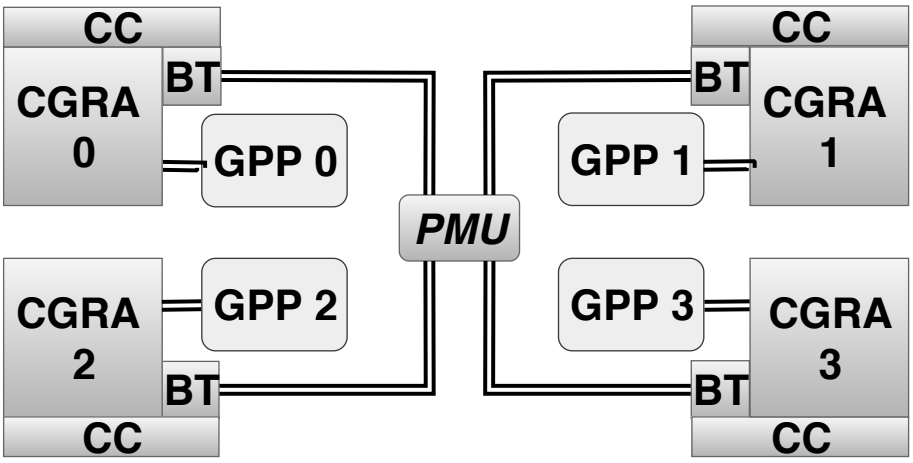
Since the proposed architecture extends the works in (BECK et al., 2008) and (BRANDALERO; BECK, 2017). The description will focus on the novel architectural elements, while modules that were implemented in the original works will only be briefly described.

3.1 System Overview

Figure 3.1 gives an overview of the proposed architecture. It consists in a set of General-Purpose Processors (GPP) tightly coupled to reconfigurable units, each including a Coarse-Grained Reconfigurable Architecture (CGRA), a special cache dedicated for storing the CGRA configurations called Configuration Cache (CC), and a Binary Translator (BT). At the center, a Power Management Unit (PMU) is connected to all reconfigurable units.

Generally speaking, the execution flow involving the execution of a *single* thread on the GPP and reconfigurable fabric can be decomposed into five steps. To support the explanation, a system's tile (GPP plus reconfigurable unit) is detailed in Figure 3.2. For the sake of simplicity, despite the multicore architecture, only a single tile is used for explanation.

Figure 3.1: Multicore CGRA Architecture.



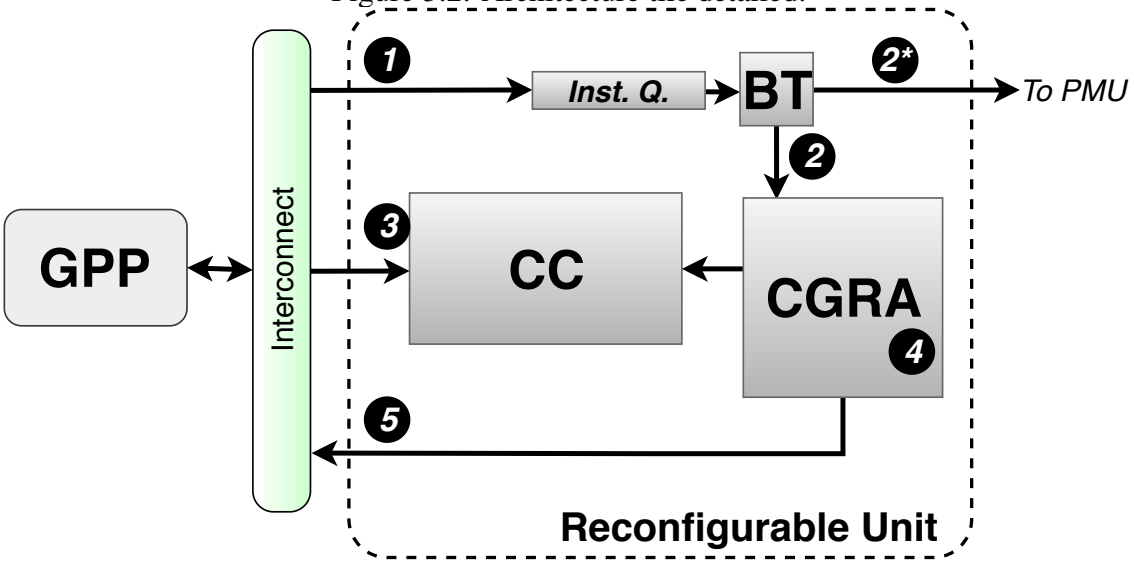
CGRA - Coarse-Grained Reconfigurable Architecture
BT - Binary Translator
CC - Configuration Cache
PMU - Power Management Unit

Source: the author

First-time instructions are always executed by the core as in a regular execution. However, these instructions are simultaneously forwarded to the the Instruction Queue (*Inst. Q.* in Figure 3.2), which acts as an interface between core and BT module. Then, the incoming instructions are fed to the BT and the translation step takes place (❶ in Figure 3.2). During the first step, the BT is producing configurations, from the instructions in the Instruction Queue, for future execution in the CGRA. Step ❷ consists in saving the configurations produced by the BT in the CC. Configurations are indexed by the Program Counter (PC) of their first instruction. Later, during program execution, the GPP fetch unit performs a lookup for the PC in the CC. If the lookup returns a match (there is a PC tag in the CC equal to the fetched PC), a configuration for executing the application's current basic block is loaded from the CC (Step ❸). Now, the CGRA can be configured and execution is offloaded to the reconfigurable fabric (Step ❹). Finally, step ❺ regards the results that are written back to the GPP. Besides the execution flow described above, data about the applications' performance on the CGRA is continuously sent to the PMU module (e.g., configurations' ILP). The steps from 1 to 5 just described were proposed in (BECK et al., 2008), and are used in other recent works such as (SOUZA et al., 2016) and (BRANDALERO et al., 2019). Moreover, the second step was extended to support the proposed resource management scheme (indicated as ❷* in Figure 3.2). Now, this step is also responsible for calculating configurations' ILP after each translation is fin-

ished. As will be further discussed in Section 3.4, this data will help the PMU to perform adjustments to all system's reconfigurable units accordingly to the applications at hand.

Figure 3.2: Architecture tile detailed.



Source: the author

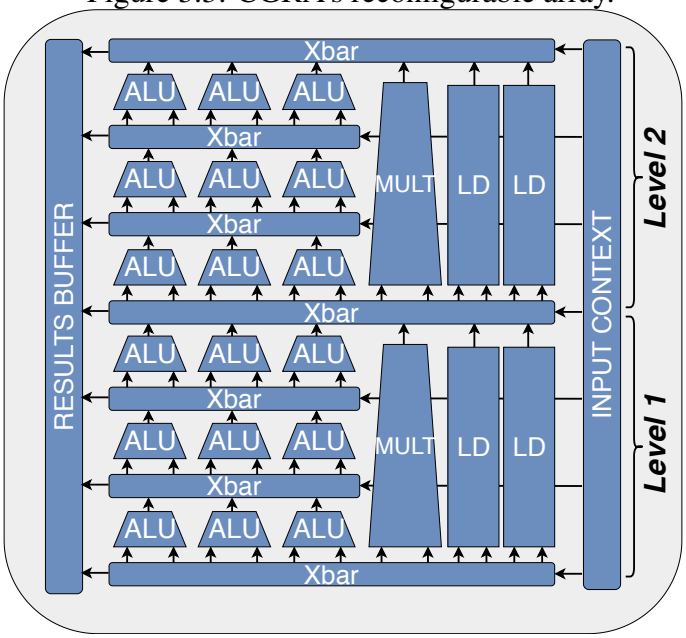
The following sections will detail the functioning of the architecture's main components. First, Section 3.2 presents the CGRA used as case-study and the configuration cache. Next, Section 3.3 explains how the system achieves transparent acceleration of basic blocks over the CGRAs' reconfigurable fabric. Finally, Section 3.4 presents the novel strategies implemented for dynamically and transparently adapting the reconfigurable resources to the applications' requirements.

3.2 CGRA

To facilitate the description of the reconfigurable fabric, we depict a two-level CGRA, smaller than the one used in this work, in Figure 3.3. The reconfigurable array is divided into rows and levels of Functional Units (FUs), which may implement integer ALU, integer multiplication (MULT), or memory operations (LD, for instance). The main characteristic of the employed CGRA is that it has no state-holding elements (i. e., there are no flip-flops between FUs). Data propagates vertically through the FUs (from row to row). FUs residing in the same row execute in parallel. A sequence of rows where the propagation delay equals the main processor cycle is called a level - therefore, it is possible to fit one level within a processor equivalent cycle without affecting its critical path. For example, the CGRA in Figure 3.3 is capable of executing up to 9 integer ALU, one

multiplication, and two memory loads, totalling a maximum execution of 12 instructions per cycle.

Figure 3.3: CGRA's reconfigurable array.



Source: the author

Rows of FUs are connected via crossbars (Xbars in Figure 3.3). Input data are fed to all rows through the *input context*. Results are sent to the *result buffer*, which leads to the main processor's register file. Not depicted in the Figure 3.3, for the sake of clarity, the multiplexers at the input and output of all FUs are configured by the CGRA configuration word. Input multiplexers select which words from the previous crossbar will feed the FU. As for the output multiplexers, they select whether the word passed to the next crossbar will be the FU's output value or a value bypassed from a previous layer's input context.

In a broader context, many CGRA organizations are possible implementing. For example, in (BECK et al., 2008), three CGRA configurations of 24, 48, and 150 rows are explored. Each row includes either 8 ALUs (in the CGRAs with 24 and 48 rows) or 12 ALUs (for the 150-rows CGRA), one or two multipliers, or two to six load/store units. In another work, Rutzig, Beck e Carro (2011) use reconfigurable arrays of 24 rows, where each row is composed of 6 ALUs, 4 load/store, and 2 multipliers. In (BRANDALERO; BECK, 2017), experiments are carried out with CGRAs of 15, 30, and 60 levels of ALUs, multipliers, and load/store units.

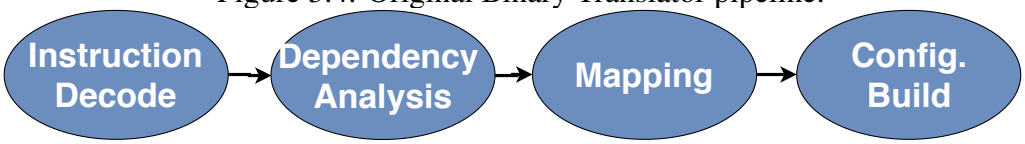
3.3 Binary Translator

A key component of the proposed architecture is the Binary Translator (BT). The use of a hardware-implemented BT enables the architecture to perform automatic and transparent *selection* and *mapping* of an applications' basic blocks. As explained in Subsection 2.3.2.1, the selection, or code analysis, consists of choosing the code regions suitable for acceleration. Ideal regions provide easily predictable branches, few true data dependencies among instructions, and a reduced number of memory operations. On the other hand, the mapping, or code transformation, lies on configuring the reconfigurable fabric to execute the selected code region.

3.3.1 Original Binary Translation Module

As the GPP executes an application, a Binary Translator (BT - Figure 3.1) connected to it is concurrently analyzing the fetched instructions to find basic blocks suitable for translation and acceleration on the CGRA. Figure 3.5 presents the Binary Translator's four pipeline stages used for the translation.

Figure 3.4: Original Binary Translator pipeline.

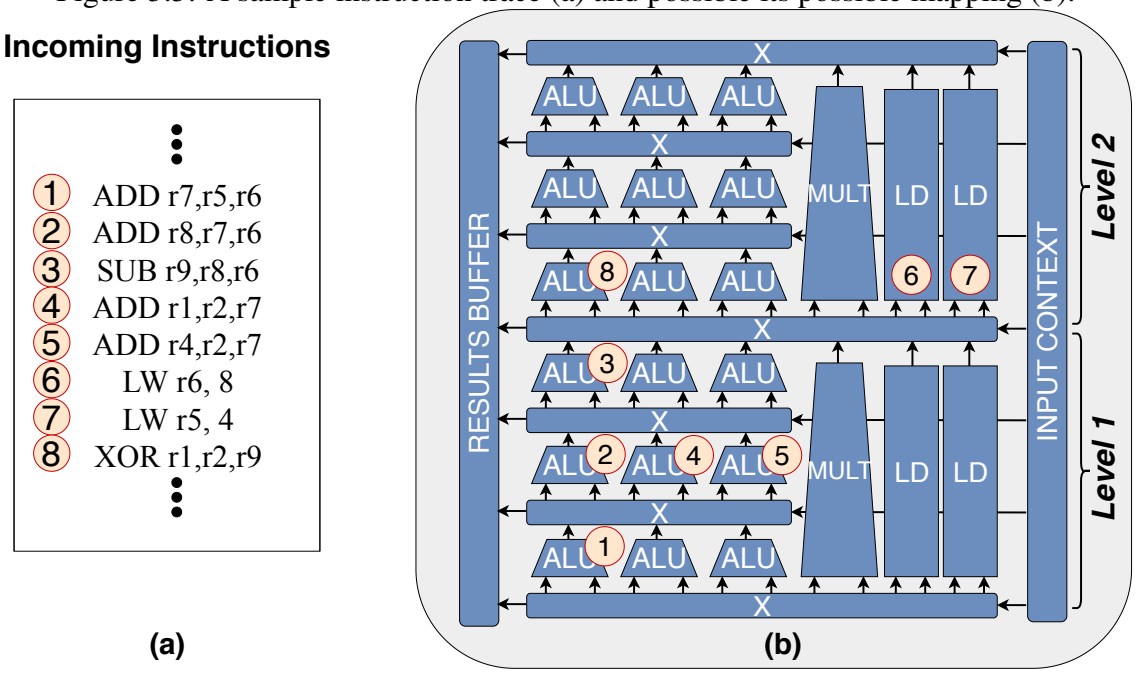


Source: the author

As adapted from (BECK; RUTZIG; CARRO, 2014), the pipeline starts with the **Instruction Decoding** stage, where it reads the incoming instructions from the processor fetch stage and decodes these instructions before storing them in a queue, where the second stage can further analyze data hazards. Next, the **Dependency Analysis** assures that the input data will be ready at the FU input by detecting data hazards. Moreover, in this stage, subsequent basic blocks and multiple branches can be speculated to enable their mapping into a single configuration. The Binary Translator speculation helps to improve the CGRA utilization of upper levels and ILP exploitation across control boundaries. The Dependency Analysis stage results in a table (initial mapping) that gives the lowest level each instruction can be mapped to. The **Mapping** stage attempts to find, for each instruction in the BB, an available FU from the lowest possible level for allocation onwards. This stage builds a bitmap, assigning an (x,y) coordinate to all instructions in the BB. From

there, the Mapping allocates FUs to instructions as well as configures the multiplexers by setting the FUs' inputs and outputs to their correct datapaths. Finally, the **Configuration Build** stage stores all the configuration context (FUs and multiplexers control signals) into the Configuration Cache. Thus, when a BB re-appears for execution, its respective configuration can be loaded from the Configuration Cache (CC - Figure 3.1), and the CGRA is configured for correct execution. As an example, Figure 3.3(b) shows numbers next to some FUs to indicate the resulting mapping from the translation of a basic block shown in Figure 3.3(a).

Figure 3.5: A sample instruction trace (a) and possible its possible mapping (b).



Source: the author

Besides the basic functionality implemented in the BT pipeline described above, there are few other points that deserve better clarification.

Immediate Values are handled by the BT with the help of a dedicated table. At translation time, these values are detected and, at runtime, these immediate values are loaded to their respective FUs by using the input context (see Figure 3.3).

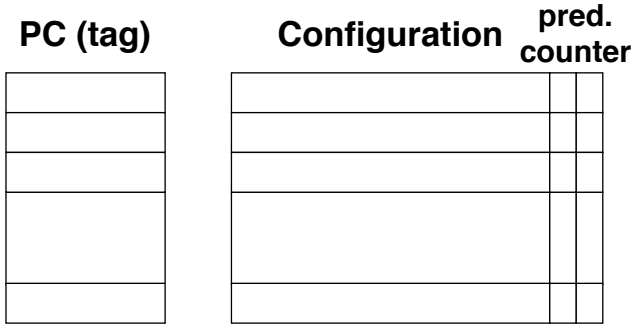
Memory Accesses are mostly handled as ordinary operations (e.g., ALU) by the BT. However, few modifications are necessary. For FUs implementing load operations, there are two input multiplexers (as in ordinary ALUs) responsible for feeding the base address and offset values. For FUs implementing stores, three input multiplexers are in order: two for address (base plus offset) and one for the value. During translation, an additional counter is used to keep track of the stores and avoid them of being reordered

w.r.t. previous loads.

Speculative Execution is also performed by the BT. Speculation happens "naturally" as subsequent basic blocks are translated into the same configuration (i.e., after a branch operation is found, BT assumes one of the possible outcomes to continue translation). Later, during execution, a mechanism confirms the outcome and the validity of the configuration from the branch forward. This mechanism consists of an auxiliary table that enables a comparison between the predicted (stored in the table) and actual (executed) outcome. If a lookup to this *branch table* returns a match, the branch is successful, and the execution may continue in the CGRA. Although, if a mismatch occurs, the last instruction prior to the branch is committed, and the remaining execution of that configuration is aborted from the CGRA.

Moreover, all configurations generated by the Binary Translator module are saved in the **Configuration Cache**. Each basic block originating a configuration is addressed by the PC of its first instruction. The configuration cache is organized into two array blocks (Figure 3.6): one for PC tag and another for their corresponding data (CGRA configuration). Also, an entry can be erased due to end of its lifetime. Each entry line in the Configuration Cache has a 2-bit counter incremented whenever a misspeculation happens (*pred. counter* in Figure 3.6). When the counter saturates, the entry is erased.

Figure 3.6: Configuration Cache structure.



Source: the author

As briefly discussed in Section 3.1, the BT includes an interface with the Power Management Unit (PMU). The additional steps for implementing this *enhancement* to BT are described below.

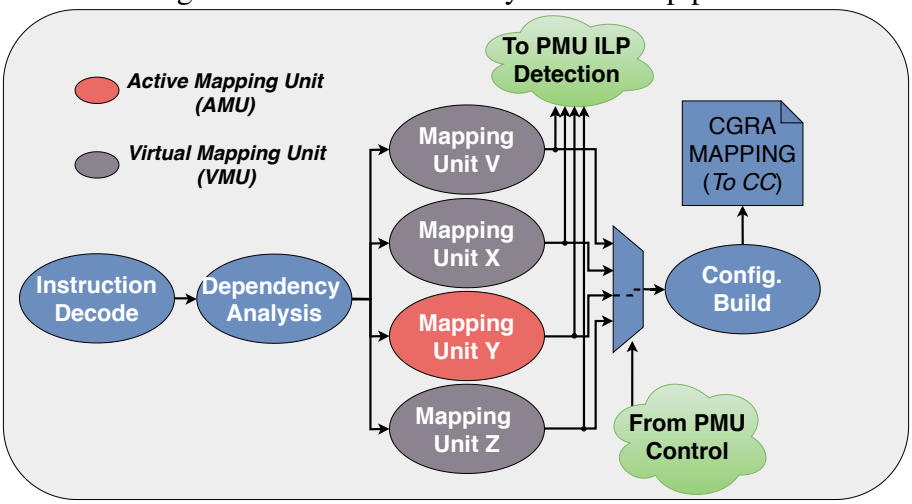
3.3.2 Enhanced Binary Translation

The Binary Translator (BT) was enhanced to interface with the module responsible for power management and enable translation when parts of the reconfigurable fabric

are power gated. Essentially, multiple mappings are concurrently being generated during translation. Each mapping matches a reconfigurable fabric of different size (accomplished via power gating of CGRA resources). Also, as will be detailed below, the ILP provided by the multiple mappings are used for evaluating the CGRA performance under multiple sizes.

As seen in Figure 3.7, the enhanced binary translator keeps the same **Instruction Decode** and **Dependency Analysis** steps as the original BT approach. The main changes were proposed in the **Mapping** and **Configuration Build** steps, described in the following subsections.

Figure 3.7: Enhanced Binary Translator pipeline.



Source: the author

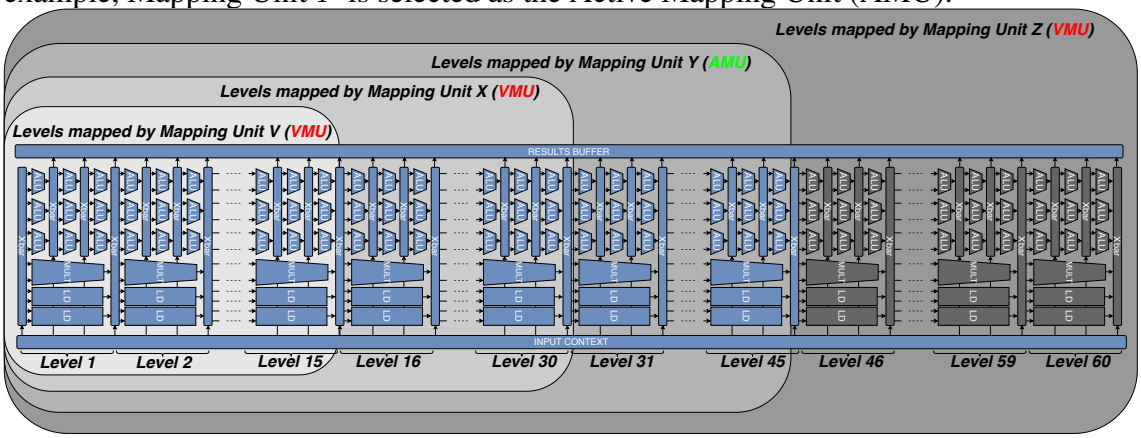
3.3.2.1 Mapping Step

The third stage in the enhanced BT (in charge of the instruction mapping) is now composed of four Mapping Units, as observed in Figure 3.7. It was empirically found that dividing the CGRA into four power domains brings the best trade-off between hardware complexity, energy consumption, and timing overheads. The main goal of replicating the mapping stage is to generate, concurrently, configurations for CGRAs under all possible power gate scenarios. That is, each tile (CGRA plus GPP), composing the multicore system, has a private BT with multiple mapping units translating basic blocks for only the tile's CGRA.

Power domains, which are defined at design time, comprise groups of four CGRA levels corresponding to the Mapping Units V, X, Y, and Z. Four Mapping Units give rise to four possible CGRA array sizes, which are given by the four power gate sleep controls.

For example, in Figure 3.8, a CGRA of 60 levels is depicted. It, then, corresponds to the array sizes of 15, 30, 45, and 60 levels. The figure also shows how those CGRA levels are divided in respect with the Mapping Units. Precisely, Mapping Unit V covers the first 15 levels of the reconfigurable fabric. Whereas Mapping Unit X is responsible for mapping the CGRA from levels 1 to 30. Following the pattern, Mapping Unit Y and Z assume a 45-level and 60-level CGRA, respectively.

Figure 3.8: Example of a 60-level CGRA with its respective four mapping units. In the example, Mapping Unit Y is selected as the Active Mapping Unit (AMU).



Source: the author

With each system's tile being able to execute CGRAs in any one of four possible sizes, the PMU can measure the ILP of the application as well as how efficiently the resources of each CGRA are being used and then cooperatively decide to which ones it will apply power gate. Let us assume that during the execution of a certain application, the Mapping Units Y and Z are providing the same average ILP, meaning that the BT is not able to leverage the extra levels provided in Z with more instructions. Hence, it is possible to make the CGRA size compatible to the mappings in Y by turning-off the underutilized levels (as done in the example of Figure 3.8, where the CGRA levels under mapping of Mapping Unit Z are power gated). Hence, power is saved and efficiency is increased since no performance penalty is incurred when shutting off the underutilized levels.

It is important to note that only configurations from one Mapping Unit are saved in the Configuration Cache. We call the Mapping Unit generating the configurations being sent to the Configuration Cache the *Active Mapping Unit* (AMU), while the remaining ones are called *Virtual Mapping Units* or VMU (Figure 3.7). These units are only used for ILP monitoring and their configurations are discarded afterward. Another relevant point to make is that replicating mapping units adds low complexity to the hardware module

and small costs in power dissipation (as will be shown in Section 4). According to Figure 3.7, the selection of the Active Mapping Unit is performed by the PMU module based on the ILP averaged over configurations produced by each Mapping Unit. As will be better detailed in Section 3.4, a new Active Mapping Unit is selected whenever the PMU detects a new program phase. In the example of Figure 3.8, Mapping Unit Y is selected as the AMU. As we can see, levels that are not under Mapping Unit Y coverage are power gated. Hence, the CGRA is running with 45 levels. However, the remaining Mapping Unit Z, a VMU in this case, keeps translating the application's basic blocks so the PMU module can monitor the performance delivered by the Mapping Unit Z (measured as ILP sent after each mapping is finished.

Furthermore, the reason for adding multiple Mapping Units to the Binary Translator pipeline instead of simply clipping a configuration into smaller configurations is as follows: the array size targeted by the mapping algorithm influences which basic blocks will compose a configuration during the runtime analysis of the application and, consequently, the acceleration delivered by the CGRA. In other words, changing the size of the array would influence where a configuration would begin and end, so it is not possible to infer, from configurations built for a larger array, how configurations of a smaller CGRA size would behave.

3.3.2.2 Configuration Build Step

Because a configuration entry is composed of control signals for all FUs and multiplexers in the array, Mapping Units targeting arrays of different sizes create configuration entries of different lengths. In other words, a Mapping Unit mapping 15 levels needs to map fewer FUs and multiplexers than a Mapping Unit that is mapping 60 levels, for example. Hence, a 60-level configuration entry is longer than a 15-level one. Consequently, the Binary Translator must be able to adapt configurations from the Configuration Cache (which has fixed size) to the current array size (which may vary in size depending on the Active Mapping Unit).

To overcome the compatibility problem caused by mixing configurations from different Mapping Units, the Binary Translator **Configuration Build** stage physically configures the storing and loading of configurations in the Configuration Cache considering the worst case in terms of entry size, which are the ones mapping to the maximum array size (in number of levels). In this respect, before storing a configuration in the Configuration Cache, the Configuration Build stage matches the length of configurations created by

the Mapping Units (of any array size) to the Configuration Cache entry by filling the configuration parts corresponding to the levels that were not mapped. Additionally, a word indicating the number of mapped levels was added to all configurations. Consequently, the Binary Translator is able to adapt configurations from any Mapping Unit to the actual length of a Configuration Cache entry (process illustrated in Figure 3.9(a)).

On the other hand, a similar problem arises when configurations are loaded from the Configuration Cache. If an instruction that is fetched by the GPP produces a hit in the Configuration Cache (i.e., its PC corresponds to an already translated basic block and there is a configuration stored in the Configuration Cache), the current CGRA size may not match the configuration size. Particularly, the issue can be broken into two cases:

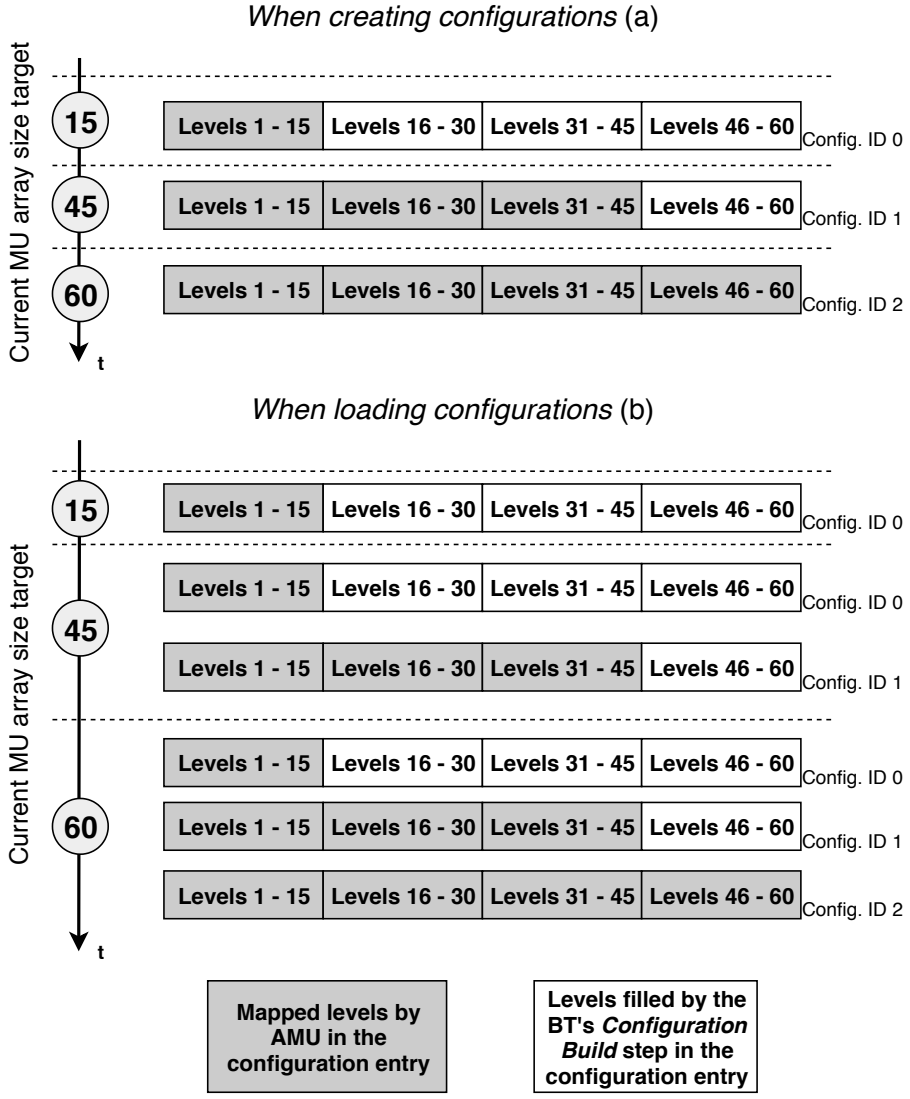
1. The Binary Translator loads from Configuration Cache a configuration that maps more levels than what is currently available in the array (e.g., CGRA is running with 15 levels and a configuration mapping 60 levels is loaded from Configuration Cache). Then, the configuration is labeled by the Binary Translator as invalid and a new translation, which maps the current array size, takes place. As in a Configuration Cache miss, the execution of the basic block is left to the GPP. The process causes this particular configuration entry to be erased from the Configuration Cache.
2. The Binary Translator loads from Configuration Cache a configuration that maps fewer levels than what is currently available in the CGRA (e.g., CGRA is running with 60 levels and a configuration previously mapped for 15 levels is loaded from Configuration Cache). Hence, due to the adaptation performed by the Configuration Build stage, the configuration can be transparently loaded. This process is depicted in Figure 3.9(b).

3.4 PMU

The Power Management Unit (PMU - Figure 3.1) dynamically monitors each CGRA coupled to each core in the system to assess its utilization, measuring the workload being executed by all concurrent applications. Depending on the applications' potential ILP of each hardware tile (CGRA plus GPP), the PMU may adapt the CGRAs to the workload at hand.

Since performing power gating implies on long interval cycles for charging and discharging circuits and additional energy consumption (HU et al., 2004), we group en-

Figure 3.9: Creation and loading of configurations with varying mapping units and array sizes in a 60-level CGRA. When creating configurations, the Binary Translator matches the configuration length from the Active Mapping Unit (black boxes) with the Configuration Cache entry length by filling the configuration (white boxes). When loading configurations for execution, they are already adapted to bigger array sizes.

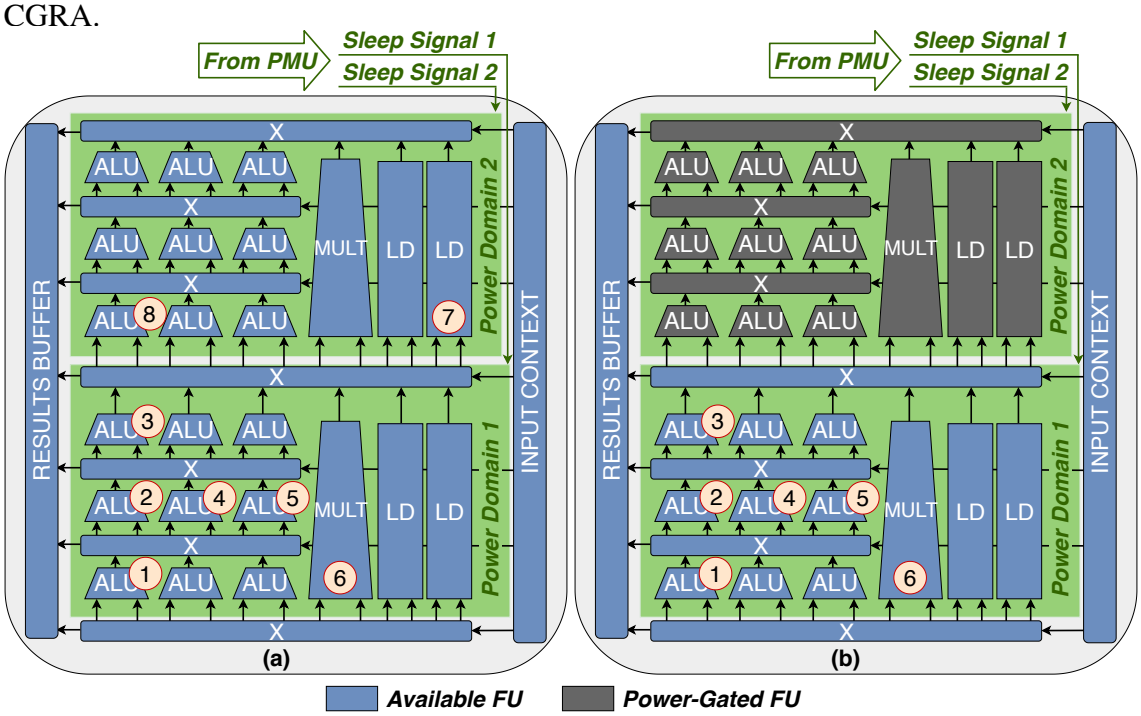


Source: the author

tire CGRA levels into power domains rather than power gate at the FU granularity. As stated earlier in Subsection 3.3.2, the CGRA was divided into four power domains. The PMU controls the sleep transistors responsible for activating and deactivating the power domains (therefore increasing or decreasing the number of available resources), as indicated in Figure 3.10.

Let us assume a straightforward example of a CGRA with only two levels and two power domains when the same instruction mapping is loaded (Figure 3.10). In the example, we have the CGRA in two states, fully operational (a) and partially power gated (b). The PMU increases energy efficiency by avoiding configurations like the one pre-

Figure 3.10: Example of CGRA mappings for fully (a) and partially (b) functioning CGRA.



Source: the author

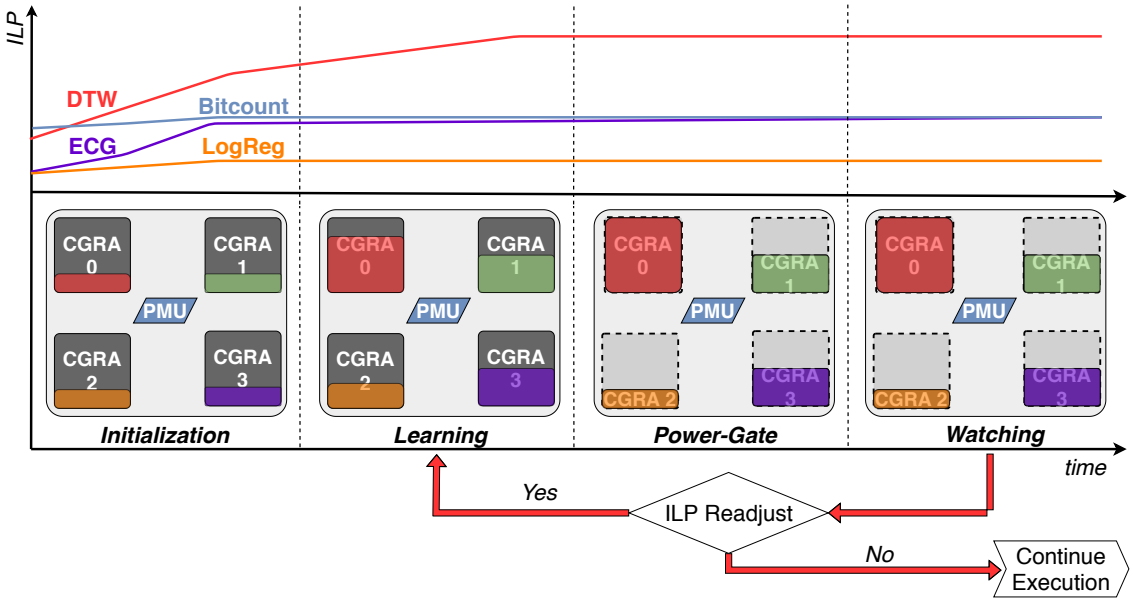
sented in Figure 3.10(a) - which results in low resource utilization in the upper levels. In this one, the array takes two clock cycles after reconfiguration to execute eight instructions. Our strategy lies in pruning the levels with low utilization from the configurations. Therefore, not executing all possible instructions of a configuration in the CGRA raises the opportunity to power gate the under-utilized levels, as shown in Figure 3.10(b).

3.4.1 PMU Phases

The PMU algorithm is divided into four phases: Initialization, Learning, Power Gate, and Watching (Figure 3.11).

1. **Initialization Phase**. All CGRAs start fully powered-on, but, in order to avoid *configuration thrashing* in the future (i.e., when a partially power gated CGRA attempts to fetch a configuration for a full powered-on CGRA), all Binary Translators start execution limited to the reduced amount of CGRA resources (done by selecting the Active Mapping Unit to the smallest number of levels - Mapping Unit V). Concurrently, the PMU starts the supervision of the four CGRA instances (from the four tiles). Meaning that it starts collecting ILP data from *active* and *virtual* Mapping

Figure 3.11: Power Management Unit functionality with sample applications.



Source: the author

Units from all Binary Translators.

2. **Learning Phase**. In this phase, the PMU cooperatively adjusts the number of resources of each CGRA with respect to both the ILP information provided by the applications' behavior and the total system power dissipation (*Learning* in Figure 3.11). After configurations from the Active and Virtual Mapping Units have their mapping finished, each Binary Translator sends the resulting ILP to the PMU. Effectively, the PMU has access to the ILP produced by each Mapping Unit of each tile in the architecture. With that information at hand, the PMU can determine the precise amount of CGRA resources demanded by all running applications. The PMU always prioritizes the application that delivers the highest ILP to its CGRA at the expense of reducing the number of activated levels from the remaining CGRAs.
3. **Power Gate Phase**. Next, parts of the CGRA resources are deactivated by the PMU through power gating (light gray shaded over *Power Gate* in Figure 3.11). In the example, the CGRA accelerating the application with the lowest ILP (LogReg) has most of its resources power gated. As a consequence, the system can employ the power that would be dissipated by the under-utilized CGRA to other CGRAs running applications with higher ILP that are more suitable for acceleration.
4. **Watching Phase**. Even though the power gating phase has passed and some CGRAs have part of their resources turned-off, the PMU keeps monitoring the ILP data from all Binary Translators in the *Watching Phase*. Consequently, the PMU can adjust

the CGRAs resource whenever it detects a change in the applications' ILP (e.g., due to distinct program phases). In case of detecting significant losses in the CGRA performance, the PMU returns to the *Learning Phase* to adjust the number of resources to each CGRA to new applications or program phases. Specifically, a value of 20% was adopted to flag a new program phase, and, consequently, the selection of the new BT's Active Mapping Unit. Also during this phase, the power gating *break-even* point is observed to assure that power dissipation is not increased. For that, a sequence of at least five CGRA configurations is monitored before performing any power gating, which gives an interval of hundreds of cycles between phases (according to experiments performed with the benchmark described in Chapter 4) - safely above the break-even point suggested in (HU et al., 2004).

4 EVALUATION

Initially, this Chapter describes the framework used to develop and evaluate the proposed architecture. Next, two evaluation scenarios are proposed that attest to the feasibility of using power gating for increasing the energy efficiency of CGRAs. In the first experiment, the PMU controller is evaluated after insertion to an architecture featuring a single GPP coupled to a relatively large reconfigurable array (60 levels). Next, a second experiment is proposed where an evaluation of the resource-aware multicore is performed against a modern multicore system of four OoO 2-issue BOOM processors. Also, in the second experiment the proposed approach is compared to multicore architectures with homogeneous and heterogeneous reconfigurable organizations.

4.1 Tools

This section presents the tools used during the development and evaluation of the proposed architecture. In this work, a cycle-accurate performance simulator was used in combination with data from hardware synthesis of real processor designs.

4.1.1 Gem5

Simulations using the Gem5 cycle-accurate simulator (BINKERT et al., 2011) were carried out for extracting performance results. When modeling an architecture in Gem5, it is possible to perform rapid design space exploration with accurate microarchitectural models, enabling shorter development cycles. Gem5 includes a wide range of CPU models and ISAs such as x86, ARM, ALPHA, SPARC, and RISC-V. It also implements detailed and flexible cache coherent mechanisms and interconnection models. The flexibility in Gem5 comes at a simulation speed/accuracy trade-off. It can be broken into three principal axes:

- **CPU Model**. Gem5 provides four CPU models, where each model gives a distinct speed and accuracy combination. *AtomicSimple* performs the fastest simulations since it runs as an instruction-level simulator with no microarchitectural timing. *TimingSimple* adds to the previous model memory access latency. *MinorCPU* models an in-order processor. It simulates a microarchitecture that can be configured to

simulate any number of pipeline stages and issue widths. It also supports branch prediction. The *O3CPU* models the microarchitecture of an out-of-order superscalar processor. This model simulates instruction dependence, functional units, and memory latency. It also allows the configuration of processor resources like load/store queue and reorder buffer that even enables Simultaneous Multithreading (SMT) execution.

- **System Mode**. Each previously described CPU model can simulate under two system modes: System call Emulation (SE) or Full System (FS). In SE mode, there is no need to model devices or an operating system since a simplified library emulates most of the system calls. On the other hand, when running in FS mode, both user-level and kernel-level code are executed by Gem5, which enables simulation of a complete system including operating system and devices.
- **Memory System**. The Gem5 simulator inherited from its predecessors two memory system models. The *Classic* model (from M5) provides fast and easy memory simulation. The second model, *Ruby* (from GEMS), implements a more complex memory system simulator. It can accurately simulate a wide range of cache coherence protocols (e.g., SLICC). Additionally, Ruby can implement any network topology composed of point-to-point links. Once the topology is declared (via a simple Python script), Ruby can simulate the network under two modes: Simple and Garnet. The former models latency of link and router and bandwidth of network links. While the latter implements the router microarchitecture in detail, including resource contention and flow control.

The Gem5 framework used leveraged an existing implementation that has coupled the Binary Translator (BT) to the O3CPU and TimingSimple CPU models used in the (BRANDALERO; BECK, 2017) and (BRANDALERO et al., 2019) works, for instance. The BT algorithm (as described in Subsection 3.3.1) was added as a pipeline stage after the CPU instruction commit to generate CGRA configurations. These configurations are then saved to a high-level Configuration Cache. The Gem5 was adapted to consider the CGRA execution when producing the simulation statistics. Precisely, whenever a configuration executes causing no misspeculation or cache misses, the timing information taken by the instruction trace executing in the CPU is replaced by the timing information in CGRA execution.

For this work, a few modifications to the Gem5 framework were introduced. First, the original BT implementation in the Gem5 code was modified to include the changes

detailed in Subsection 3.3.2. The modifications consisted of the replication of the third translation stage, where multiple Mapping Units were added with predefined array sizes for mapping (corresponding to CGRA power domains). Later, the Configuration Build stage was modified to include the selection of the Active Mapping Unit (for storing configurations in the Configuration Cache) and the new interface with the PMU module.

Second, the PMU module was implemented as a Gem5 module connected to the BT via *buffers*. The module implements a version of the algorithm explained in Section 3.4. The final changes to the Gem5 simulation framework were to enable multicore simulations in SE mode (currently, the only mode allowed). To that end, all simulation scripts were firstly adapted to lunch simulation of multiple CPUs. Then, the instantiation of BTs and Configuration Caches were also adapted to enable their functioning with multiple CPUs. Finally, the PMU module that was first implemented for interfacing with a single BT was extended to work with multiple BTs.

4.1.2 CACTI

CACTI (BALASUBRAMONIAN et al., 2017) is an analytical tool that takes as input user-specified parameters like line size, associativity, number of sets and ports, and optimization targets that can be latency, area, or power. Then, CACTI estimates the results of latency, area, energy per access, and leakage power. In its latest major review, CACTI provides results for 90nm, 65nm, 45nm, and 32nm technology nodes.

Moreover, an extension of CACTI, the FinCACTI (SHAFAEI et al., 2014), was proposed to include more recent technology nodes, modeling caches and memories based on FinFET devices. FinCACTI is used in this work to estimate area and power of Configuration Cache implementations.

4.1.3 Rocket Chip Generator

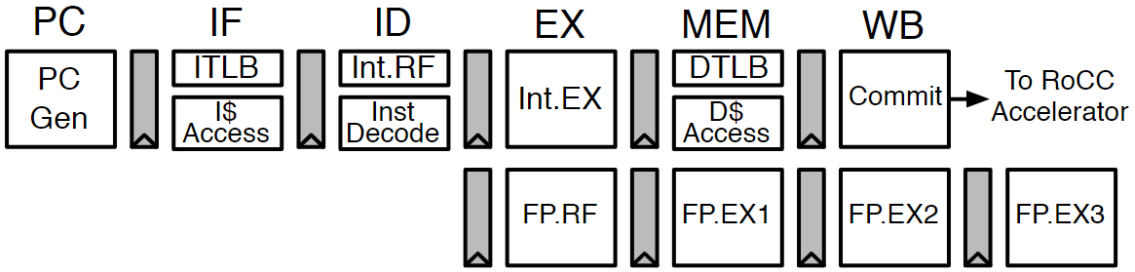
For power and area results of the main processor, the Rocket Chip Generator [1] was used. It is an open-source framework for generating parameterizable RISC-V processors (ASANOVIć et al., 2016). It is implemented in the Chisel language (BACHRACH et al., 2012), which is a hardware construction language that provides a high level of abstraction

[1] https://github.com/chipsalliance/rocket-chip

by enabling object orientation, type inferencing, and parameterized types, for instance. From a hardware module described in Chisel, it is possible to generate a high-speed C++ simulator for validation or a low-level Verilog design that works as input for traditional FPGA and ASIC design flows.

The Rocket Chip Generator was first proposed for generating the *Rocket Core*, an in-order 5-stage scalar processor with branch prediction that implements the RV32G and RV64G ISAs. The Rocket Core also includes a Memory Management Unit (MMU) that supports page-based virtual memory and may include a floating-point functional unit. Figure 4.1 details the Rocket Core pipeline.

Figure 4.1: The Rocket Core pipeline.

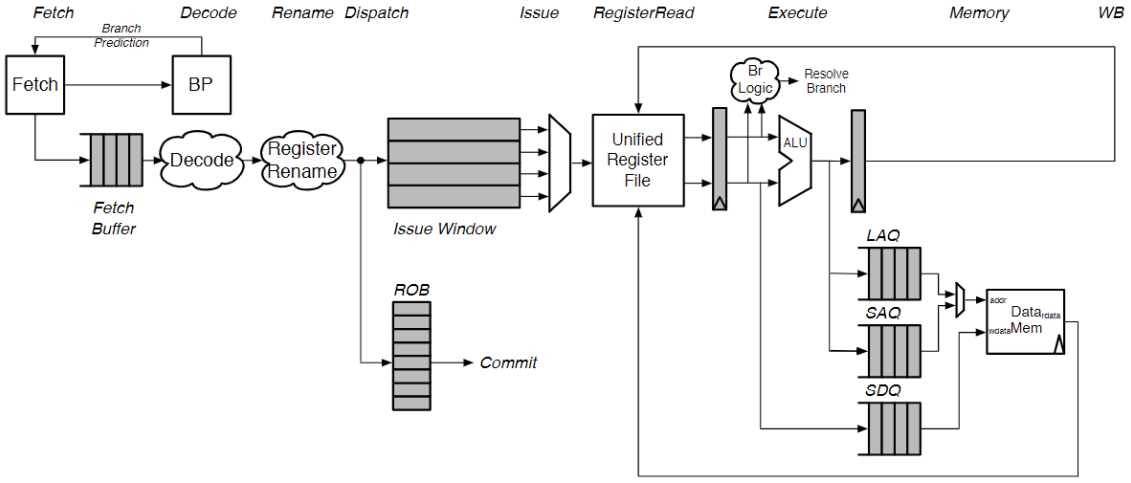


Source: (ASANOVIć et al., 2016).

Later, an out-of-order superscalar core was incorporated into the generator. The *BOOM Core* (Berkeley Out-of-Order Machine) implements the RV64G (CELIO; PATTERSON; ASANOVIć, 2015). BOOM supports full branch prediction with branch target buffer (BTB), branch history table (BHT), and a return address stack (RAS). The load/store unit allows for out-of-order loads and forward of stores to dependent loads. Figure 4.2 shows the six pipeline stages: Fetch, Decode/Rename/Dispatch, Issue/RegisterRead, Execute, Memory, and Writeback. Chisel enables us to easily parameterize the BOOM Core according to fetch, decode, and issue widths as well as customization of the functional units mix.

In this work, experiments were carried out using both Rocket and BOOM (2-issue configured) cores. As the Rocket Chip Generator offers a full design flow, performance results could also be extracted from it through RTL simulations, instead of Gem5 simulations. However, it would require re-implementation, in the generator's language Chisel, of the full reconfigurable unit (including Binary Translator and its interface with the CPU, the reconfigurable array and its interface, and the Configuration Cache). Additionally, it is known that RTL simulations involve long simulation times, restricting the benchmark to a more simple set of applications and input sets, and incurring in longer design cycles

Figure 4.2: The BOOM Core pipeline.



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

than what is possible with Gem5.

4.1.4 Logic Synthesis

The hardware modules implemented in register transfer level (RTL) were synthesized using the Cadence Genus tool. Additionally, the 15nm predictive FinFET standard cell library by Silvaco was used in the logic synthesis (MARTINS et al., 2015) for power and area characterization. It is important to note that, for the power analysis, the power gated parts of the CGRA are assumed to consume no power.

4.2 Single-core Scenario

4.2.1 Methodology

Aiming to evaluate the resource management scheme in a single-core environment, in terms of utilization rate, performance, power, and energy, the baseline architecture has the same configuration without the ability to power gate any CGRA resource. The setup used for this scenario is presented in Table 4.1. The setup is based on the architecture proposed in (BRANDALERO; BECK, 2017). Precisely, the configuration cache and CGRA size, as well as for selecting the CGRA functional units setup are kept from the work by Brandalero e Beck (2017).

Table 4.1: Evaluation Setup for the single-core scenario.

GPP	Processor	RISC-V Rocket (In-order) 1.6GHz
	L1 Cache	64KB DCache, 32KB ICache
CGRA	Level	12 Integer ALUs, 2 Multipliers (3 cycles, pipelined), 2 Read Ports (3 cycles), 1 Write Port (3 cycles)
	Configuration Cache	32KB
PMU	Power Domains (#levels) for Mappings Units from V to Z	15, 30, 45, 60

Source: the author.

Applications from domains usually present in IoT devices, namely machine learning, image/signal processing, and security, were selected as benchmark from (LECUN et al., 1998; GUTHAUS et al., 2001; FRITTS et al., 2009; REAGEN et al., 2014; POUCHET, 2019). More details about the selected applications are given in Table 4.2.

Table 4.2: Benchmark summary for the single-core scenario.

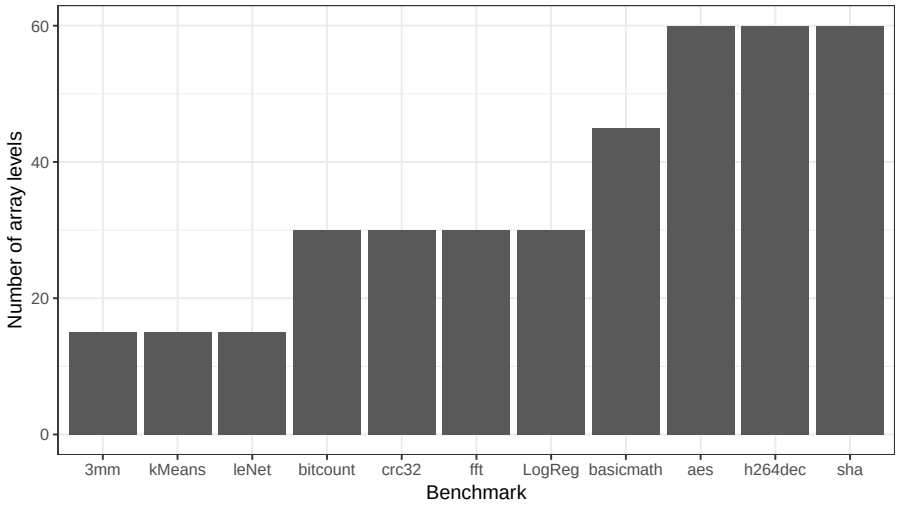
Benchmark	**Description**	**Data Size**
3mm (POUCHET, 2019)	Matrix Multiplication	3x (128x128)
K-Means	Machine Learning	100 points, 10 iterations
LeNet (LECUN et al., 1998)	Machine Learning	1 image
FFT (GUTHAUS et al., 2001)	Signal Processing	4 waves, 4096 points
Bit Count (GUTHAUS et al., 2001)	Bit Manipulation	75000 iterations
CRC32 (GUTHAUS et al., 2001)	Security	10M Bytes
LogReg	Logistic Regression	500 epochs, 10 train points, 10 test points
BasicMath (GUTHAUS et al., 2001)	Automotive	MiBench's small
AES (REAGEN et al., 2014)	Security	2M Bytes
H264Dec (FRITTS et al., 2009)	Video Decoder	904K Bytes
SHA (GUTHAUS et al., 2001)	Security	2M Bytes

Source: the author.

4.2.2 Results

CGRA Usage. The first analysis regards the CGRA resource usage when a diverse set of applications is adopted. Figure 4.3 attest for the necessity of having a dynamic allocation scheme that takes into account the application running when targeting efficiency. Since the PMU module optimizes CGRA size (given by the number of enabled levels) at the program phase granularity (not application), it is shown the array size used for the application's program phase with the longest duration. For example, applications like 3mm, K-Means, and LeNet spent most of their time using 15 levels only while other applications namely SHA, H264Dec, and AES end up using all the levels available by the CGRA for most of their execution time. Essentially, in a system that aims to, simultaneously, meet performance and energy savings for a variety of applications, the ability to adapt the available resources to the current application is paramount.

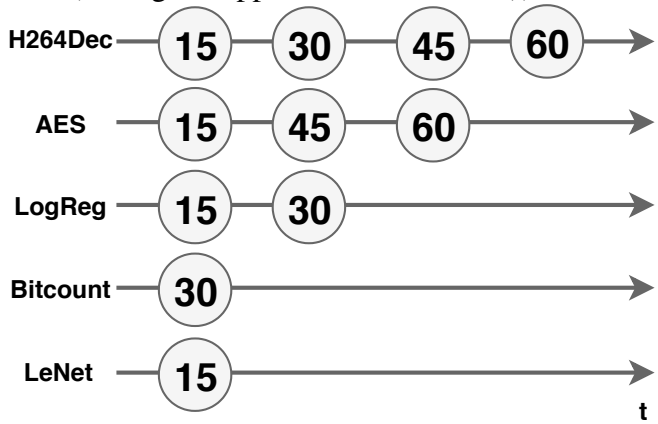
Figure 4.3: The CGRA array size for the longest application program phase.



Source: the author.

Additionally, adapting CGRA resources is also relevant during the execution of a single application. Shifts in program phase (SHERWOOD et al., 2002) provide opportunities for power savings if one can power-gate under-utilized levels of CGRA during the execution of a single application. To illustrate how the PMU module adapts the CGRA size according to the application program phase, Figure 4.4 shows the program phases of five applications. Each circle represents a program phase detected by the PMU. Inside a circle is the CGRA number of levels picked by the PMU. Each CGRA size is loading configurations from a particular mapping unit, where the Mapping V is used for generating configurations for the array of 15 levels, Mapping X for 30 levels, Mapping Y for 45 levels, and Mapping Z for 60 levels.

Figure 4.4: The current CGRA size, in number of levels, attributed by the PMU to each program phase (circles) along the application execution (t).



Source: the author.

From Figure 4.4 is also possible to notice a remarkable difference in the applications behavior. The homogeneity in program phases positively contributes to the overall

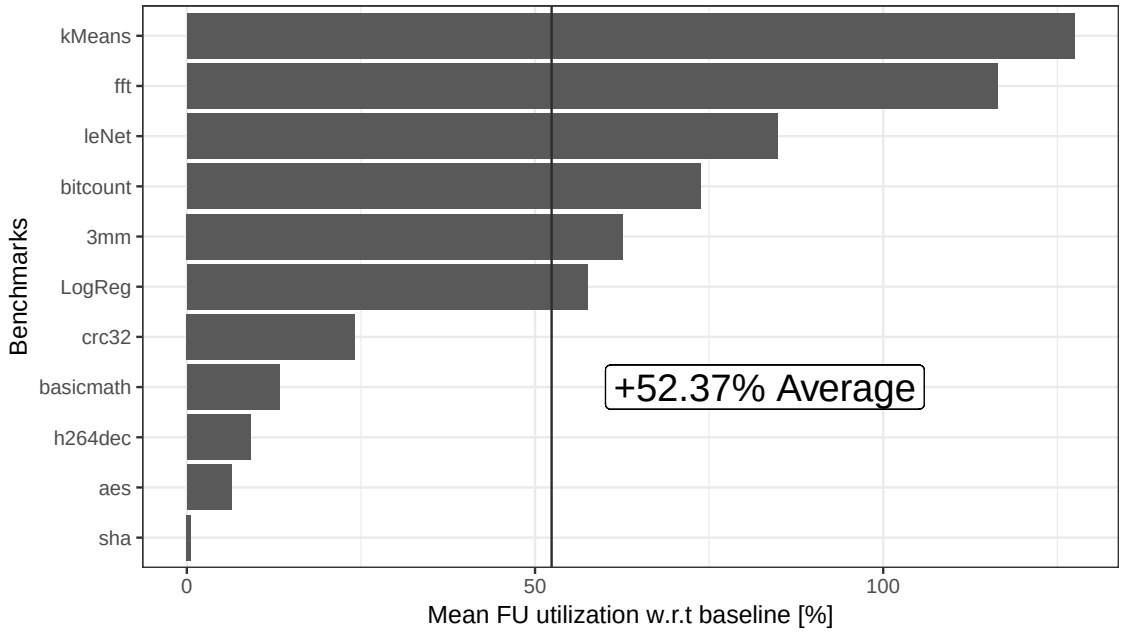
system performance. For instance, applications such as Bitcount and LeNet have a single program phase. Consequently, PMU needs to perform power gating only once during the execution of those applications - implicating in smaller performance overhead and higher gains in power dissipation when compared to applications with highly heterogeneous program phases. Applications like H264Dec and AES possess four and three program phases, respectively. For applications with a heterogeneous behavior, the PMU will be more frequently interrupted by the detection of new program phases, hence, performing more power gating and switches between Mapping Units.

Utilization Rate. One important aspect to investigate is the number of powered functional units (i.e. not power-gated) that are actually performing computation. We have called *utilization rate* the ratio between mapped and total number of FUs. To achieve high levels of energy efficiency, the mapping process has to produce configurations with high utilization rates, which may also be interpreted as high levels of instructions per cycle (IPC). Figure 4.5 shows the FU utilization rates of the proposed architecture over the baseline architecture. As expected, when the PMU converges to smaller array, higher utilization rates are shown. For instance, the PMU improves the utilization of FUs more than $2\times$ when considering the K-Means and FFT applications. On the other hand, for applications that have long program phases requiring all 60 levels (SHA, AES and H264Dec), small improvements in utilization rate are observed. BasicMath also provides small improvements in utilization rate (13.7%), since it has three very distinct program phases. Considering all benchmarks, the increasing on utilization rate is 52.37%, on average.

Performance Evaluation. Figure 4.6 shows the performance of the proposed approach normalized to the baseline architecture, where positive values mean an increase and negative values mean a reduction in the execution time of each application (distributed along the y-axis). As explained in the Chapter 3.2, the more opportunities to power gate the application has, more overhead will be added, given by the frequently changing of the Binary Translator's Active Mapping Unit. 3mm, LogReg and SHA suffer from that behavior, which affects their performance in 10%, 3.3% and 3.2%, respectively.

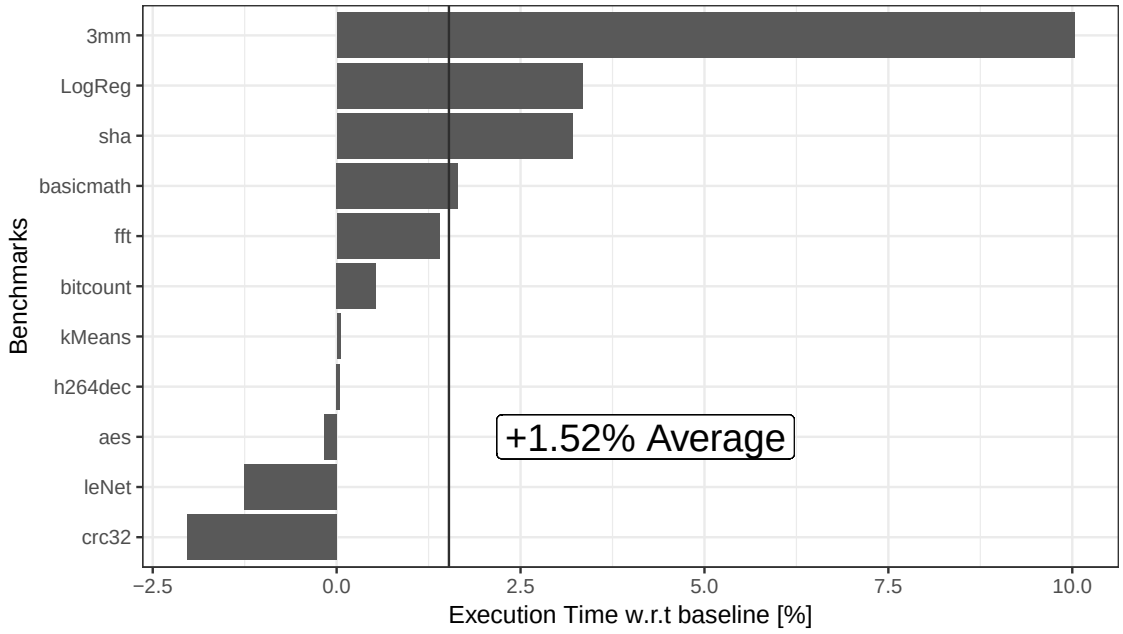
However, applications that rapidly converge and stabilize their execution into a reduced number of levels show performance improvements since fewer CGRA levels are propagated in the Binary Translator configurations than in the baseline architecture. For instance, the proposed architecture even improves the performance of the CRC application in 2%, for which the PMU converged to 30 levels (refer to Figure 4.3) - producing levels of ILP higher than the produced by the baseline system with 60 levels.

Figure 4.5: Mean FU utilization rate after adding the proposed resource management scheme to the CGRA.



Source: the author.

Figure 4.6: Performance after adding the proposed resource management scheme to the CGRA.

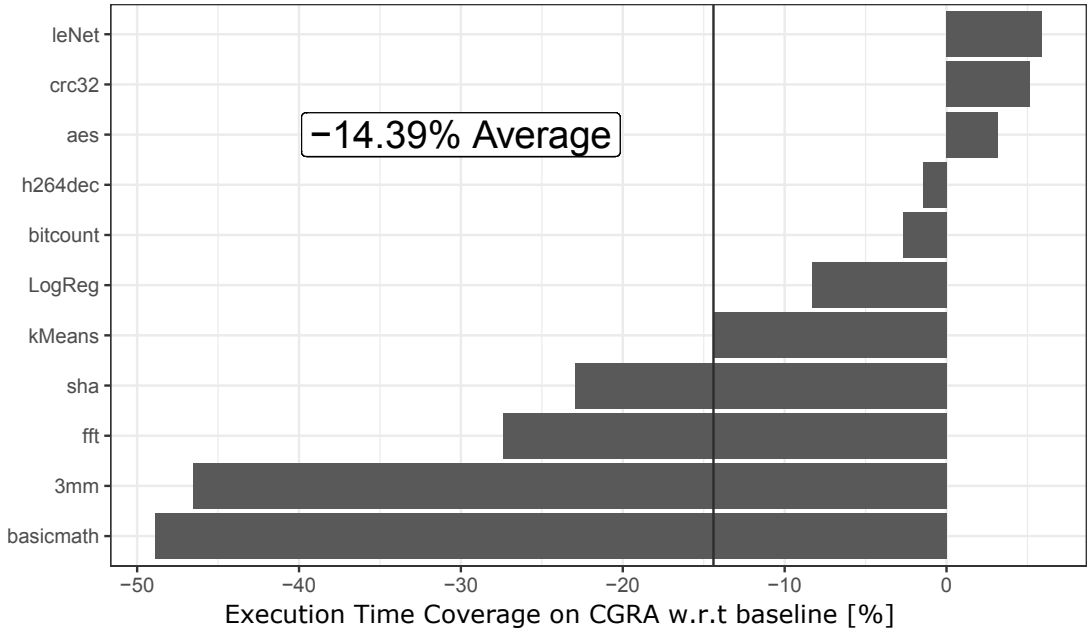


Source: the author.

Additionally, we also can assess the difference in performance between baseline and proposed system by looking at what fraction of execution time an application spent in the CGRA during its complete execution. We call *Execution Time Coverage* the fraction of time the application executed in the reconfigurable fabric. Naturally, when more

instructions execute in the reconfigurable fabric, greater acceleration levels are expected. Figure 4.7 depicts the Execution Time Coverage of the proposed architecture w.r.t baseline. It is clear that only for those applications with marginal speedups, more time was spent with CGRA execution. Precisely, for Lenet, CRC, and AES applications, an improvement of 5.92%, 5.13%, and 3.19%, respectively, is observed. As for the remaining applications, all experienced some reduction in the execution time spent over the CGRA - with direct impacts on performance (as seen in Figure 4.6). For Basicmath and 3mm, the PMU caused the biggest impacts on Execution Time Coverage with reductions of 48.9% and 46.53%, respectively.

Figure 4.7: Execution Time Coverage w.r.t baseline.

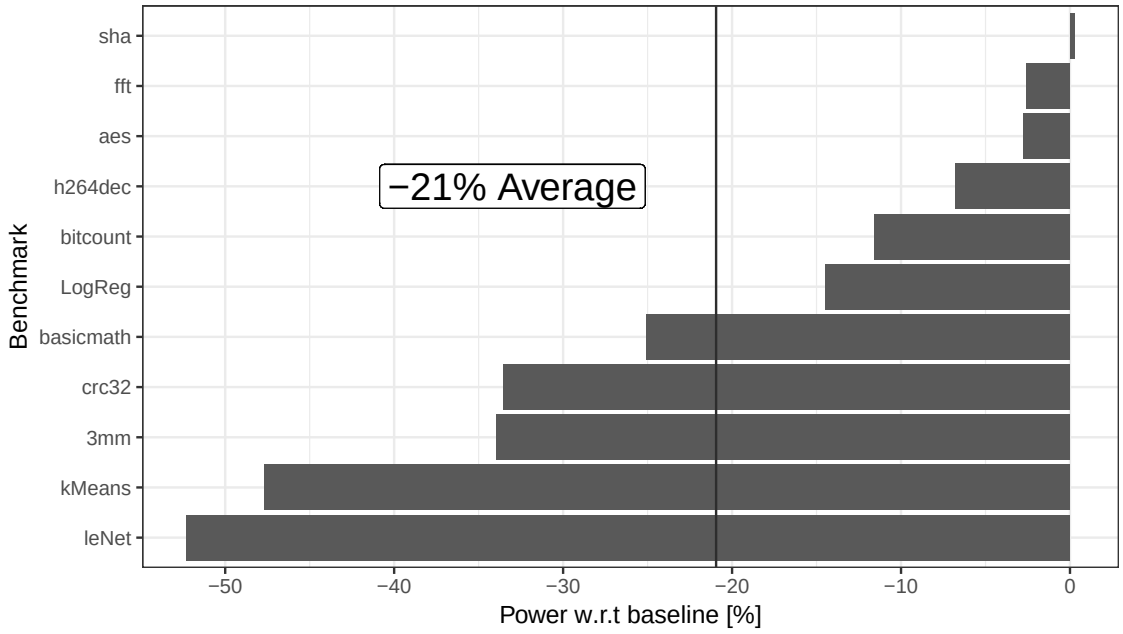


Source: the author.

Power Evaluation. Figure 4.8 shows the savings in power dissipation when using the PMU over the baseline architecture. Negative values mean a reduction, while positive values mean an increase in power dissipation. In addition, the power results consider both dynamic and static power.

As it can be noticed in Figure 4.8, the PMU provides power savings in most applications. LeNet (-52.3%), K-Means (-47.7%) and 3mm (-34%) are most benefited from PMU since they rapidly converge to 15 levels, allowing the system to power-gate 45 CGRA levels early in the program execution. In addition, such an applications have regular program phases (i.e., with few shifts in CGRA usage) requiring fewer power gating executions.

Figure 4.8: Power dissipation after adding the proposed resource management scheme to the CGRA.



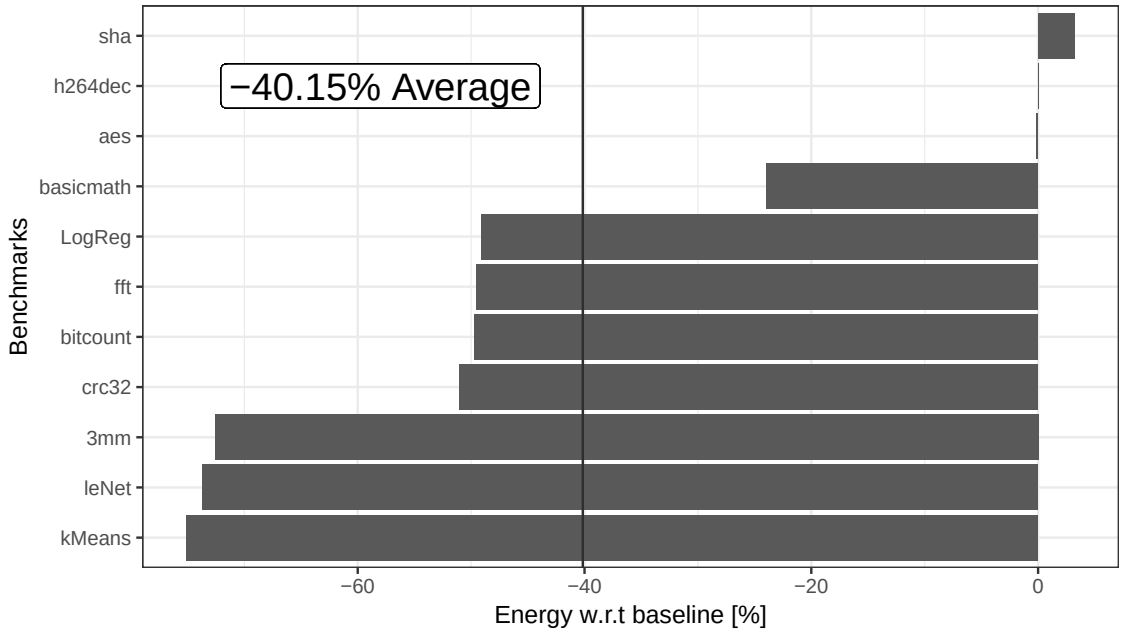
Source: the author.

On the other hand, as there is no room for power gating in SHA (refer to Figure 4.3), a small power overhead of 0.26% is observed in the SHA application execution. Conclusively, even though the enhanced Binary Translator consumes 29% more power than the baseline Binary Translator, the proposed approach results in an average decrease of 21% in the overall system power dissipation since the BT represents only a small portion of the total power.

Energy Evaluation. To evaluate energy consumption, we have set the applications to continually execute for twenty seconds on both proposed and baseline systems. For all applications, but SHA, the PMU achieves energy savings since there are more power savings (refer to Figure 4.8) than performance penalties (refer to Figure 4.6). For instance, as K-means rapidly converges to 15 levels, an energy saving of 75% is achieved by avoiding power dissipation of unused functional units. On the other hand, even in applications, such as SHA, that do not provide any room for power gating, the proposed system spends, at most, 3.2% more energy. Summarizing, the proposed system provides a transparent power gating approach based on application phase prediction that reduces, on average, 40.15% of the energy consumption with an negligible (1.52%) performance penalty.

Key Findings. In this experiment, it became clear that applications' requirements for CGRA resources are diverse when one considers a heterogeneous and complex set of

Figure 4.9: Energy Reduction after adding the proposed resource management scheme to the CGRA.



Source: the author.

applications. It was shown that to overcome the underutilization caused by applications with low levels of ILP, a simple but yet efficient power gating technique can be used. Precisely, with an average increase of the CGRA utilization rate of 52.37%, the average energy consumption could be reduced in 40.15% at marginal costs in performance (1.52% average).

4.3 Multicore Scenario

4.3.1 Methodology

In the multicore scenario, full usage of the PMU is required. With multiple tiles under coordination of the PMU, a full system view is possible by the manager module.

The evaluated system configuration (shown in Table 4.3) was designed to function under the power envelope of a state-of-the-art edge platform. For that, the edge-tailored Stitch (TAN et al., 2018), designed to meet the power requirements of wearable devices, was modeled with synthesis data of RISC-V Rocket processors, which gives the 140mW upper-bound on the total power dissipation. The *baseline* used for comparison is composed of a quad-core Out-of-Order (OoO) 2-issue BOOM processors (CELIO; PATTER-

SON; ASANOVIć, 2015) (running at 2GHz), *with area footprint equivalent to system of Table 4.3*.

We also compare the proposed with other two systems: the same quad-core RISC-V Rocket system used, but coupled to four *homogeneous* CGRAs; and coupled to four *heterogeneous* CGRAs. While the homogeneous system is configured with 24-levels CGRAs (same size as a fully active CGRAs in Table 4.3), the heterogeneous configuration was selected by taking the average occupation on each CGRA for the evaluated applications and respective scenarios' allocation. The resulted heterogeneous configuration is composed of 6, 18, 18, and 24-level CGRAs. By comparing to the homogeneous version, which is basically a version of same architecture without resource management that can achieve the maximum possible performance, we can evaluate what is the price to pay in performance for achieving the aimed energy gains. The comparison with the heterogeneous version is also interesting: in this case, we are assessing how important the adaptability provided by the resource management is, since the heterogeneous version has the same amount of hardware available in Table 4.3, but is fixed throughout execution. An additional performance evaluation against the state-of-the-art **Stitch** is provided (TAN et al., 2018).

Table 4.3: Evaluation Setup for the multi-core scenario.

4xGPP	Processor	RISC-V Rocket (in-order, single-issue) 2GHz
	Cache Size	64KB DCache, 32KB ICache
4xCGRA	CGRA level	12 Integer ALUs, 2 Multipliers (3 cycles, pipelined), 1 Read Port (3 cycles), 1 Write Port (3 cycles)
	Configuration Cache Size	32KB
PMU	Power Domains (#levels)	6, 12, 18, 24

Source: the author.

A set of edge representative benchmarks were evaluated, which include applications from four application domains: Machine Learning, Security, Image and Signal Processing, and Miscellaneous (TAN et al., 2016; POUCHET, 2019; GUTHAUS et al., 2001). We propose four evaluation scenarios where each one (from A to D) is composed of four applications, every one of which was randomly selected from a different domain (Table 4.4).

Table 4.4: Evaluation scenarios for the multi-core scenario.

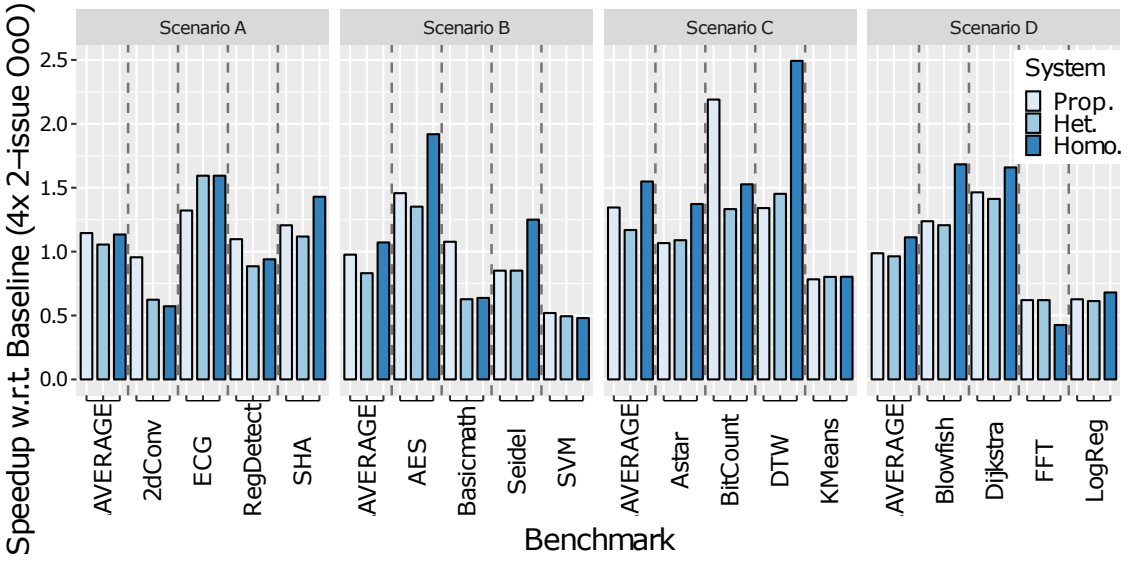
Scenario	Machine Learning	Security	Image and Signal Processing	Misc.
A	2D Convolution	SHA	Reg-detect	ECG
B	SVM	AES	Seidel	Basicmath
C	KMeans	Bitcount	DTW	A*
D	Logistic Regression	Blowfish	FFT	Dijkstra

Source: the author.

4.3.2 Results

Performance Evaluation. Figure 4.10 shows the speedup when executing the evaluation scenarios in the proposed, heterogeneous, and homogeneous systems (presented as distinct bar colors). Applications composing each of the four scenarios are grouped with their average speedup. Considering all evaluation scenarios, the average speedup over the *baseline* system is 1.11x. While the systems with *homogeneous* and *heterogeneous* CGRAs reported average speedups of 1.22x and 1.01x, respectively.

Figure 4.10: Speedup w.r.t. baseline for the proposed, homogeneous, and heterogeneous systems (higher is better).



Source: the author.

From the results, it is clear that even compared to a system with powerful GPPs, CGRAs still provide performance benefits for some applications. Also, Figure 4.10 shows that applications' ILP causes significant impacts on the performance attainable by CGRAs. For instance, applications with high levels of ILP like BitCount, AES, and ECG present significant speedups over the *baseline* system (2.19x, 1.46x, and 1.32x, respectively) while running over the proposed system. On the other hand, it provides smaller levels of acceleration for other applications, which do not have enough ILP to use the CGRA resources better. For example, SVM and LogReg are not able to leverage the highly parallel CGRA's fabric due to their low ILP and highly memory-dependent codes, resulting in faster execution times when running on OoO 2-issue processors (speedups of 0.52x and 0.63x, respectively).

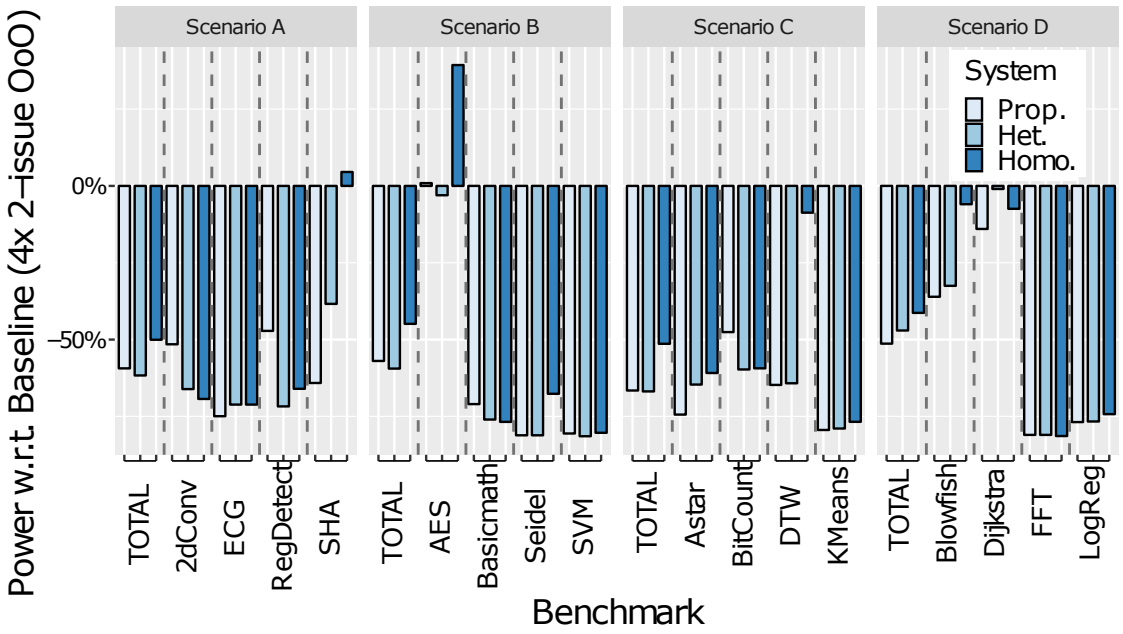
The proposed system even outperforms its *homogeneous* and *heterogeneous* coun-

terparts for some applications, showing the little impact the PMU has on performance. The effect of applications obtaining better performance results when parts of the CGRA are power gated is mainly caused by the Binary Translator algorithm producing configurations with higher ILP for smaller CGRAs (as previously discussed in Subsection 3.3). For instance, the BitCount application, when executing in the proposed system, achieves a speedup 30% greater than the provided by the *homogeneous* system and 39% greater than the provided by the *heterogeneous* one. This effect is also notable for the BasicMath application that improved the speedup, over the *homogeneous* system, in 40% and 41% over the *heterogeneous*.

Power Dissipation. Figure 4.11 displays the evaluation scenarios, their applications, and their total power dissipation w.r.t. the *baseline* (a quad-core OoO 2-issue BOOM processors). Results are given for the proposed, *homogeneous*, and *heterogeneous* systems, where lower bars represent savings in power dissipation over the *baseline* system. Overall, the proposed system can keep up with similar saves in power dissipation to what is achieved with the *heterogeneous* system. Averaging the results across all application scenarios and comparing it to the baseline, the proposed system saves 58.98% over the total system power dissipation. Whereas the system with *heterogeneous* CGRAs decreases the total power dissipation in 59.24%. On the other hand, when the system with four *homogeneous* CGRAs is compared to the *baseline*, a reduction of 47.64% in the total power dissipation is observed. The increase in power dissipation, w.r.t the *baseline*, for the AES and SHA applications, is caused by high CGRA utilization rates provided by the applications' high levels of ILP (also observed in terms of elevated speedups in Figure 4.10).

As presented earlier, the savings in power dissipation, when compared to systems with no online management, come at little impact on performance since the power management leverages the applications that make poor usage of the CGRA to let other applications to allocate more resources cooperatively. Precisely, by employing the proposed resource management technique 44.79%, on average, of the power dissipated by CGRAs is saved when compared to the *homogeneous* system and -0.55% when compared to the system with *heterogeneous* CGRAs. In Figure 4.12, we focus our analysis on the power dissipated by the *homogeneous* and *heterogeneous* CGRAs w.r.t. proposed's CGRAs. From Figure 4.12, the gains arising from dynamic adaptability of CGRAs become clearer. While keeping power dissipation at levels similar to the *heterogeneous* architecture, the proposed system obtains power savings ranging from 26.89% to 78.12%

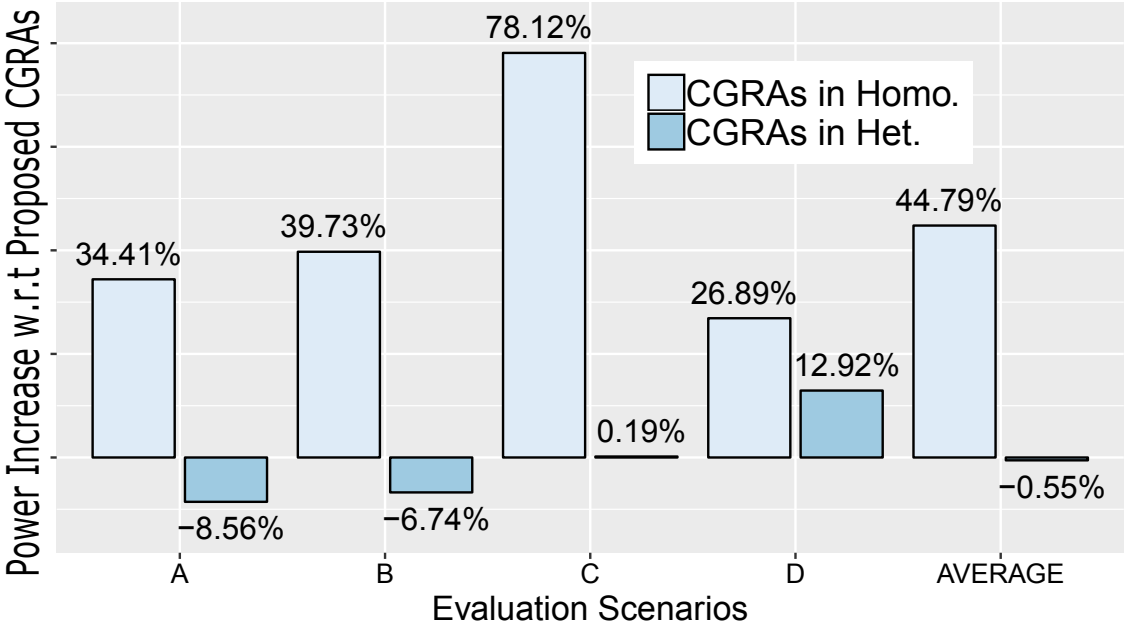
Figure 4.11: Power dissipation w.r.t. baseline for the proposed, homogeneous, and heterogeneous systems (lower is better).



Source: the author.

when compared to the *homogeneous* system.

Figure 4.12: CGRAs' power dissipation in homogeneous and heterogeneous systems w.r.t. the CGRAs in proposed system.

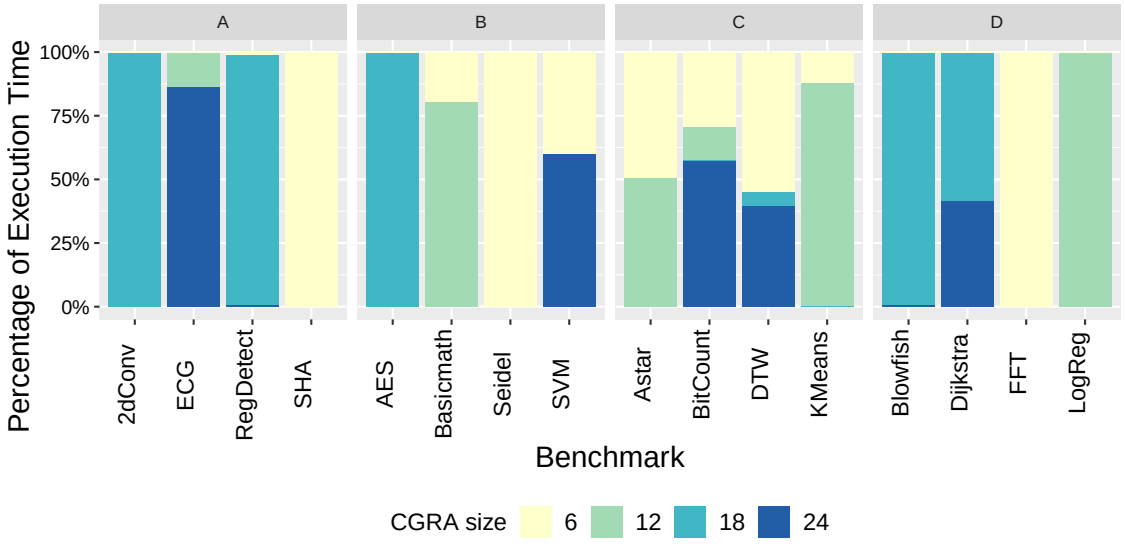


Source: the author.

CGRA usage. Here, an analysis of the CGRA utilization is proposed. In Figure 4.13, each bar corresponds to the full execution breakdown of the CGRA for each

application regarding the available power domains (6, 12, 18, and 24 levels - See Table 4.3), while running over the proposed system. For example, for the SVM application, in 60.23% of the execution time, the PMU selected the Mapping Unit Z (running with the full 24 CGRA levels). In contrast, for the remaining 39.77% of the execution, the PMU selected execution over six levels (Mapping Unit X). Additionally, in other applications, it is possible to note a more steady behavior. For instance, applications like 2dConv (18 levels), FFT (6 levels), and LogReg (12 levels) have spent most of their execution time with the same CGRA size (99.78%, 99.98%, and 99.93%, respectively). Also, Figure 4.13 may help in understanding the gains in power dissipation. For example, the SHA application, which produced over 50% in power savings, has spent most of its execution with the only six levels powered-on. Naturally, as more CGRA levels can be powered-off by the PMU module, more power is saved.

Figure 4.13: Percentage of the execution time spent with each CGRA size.

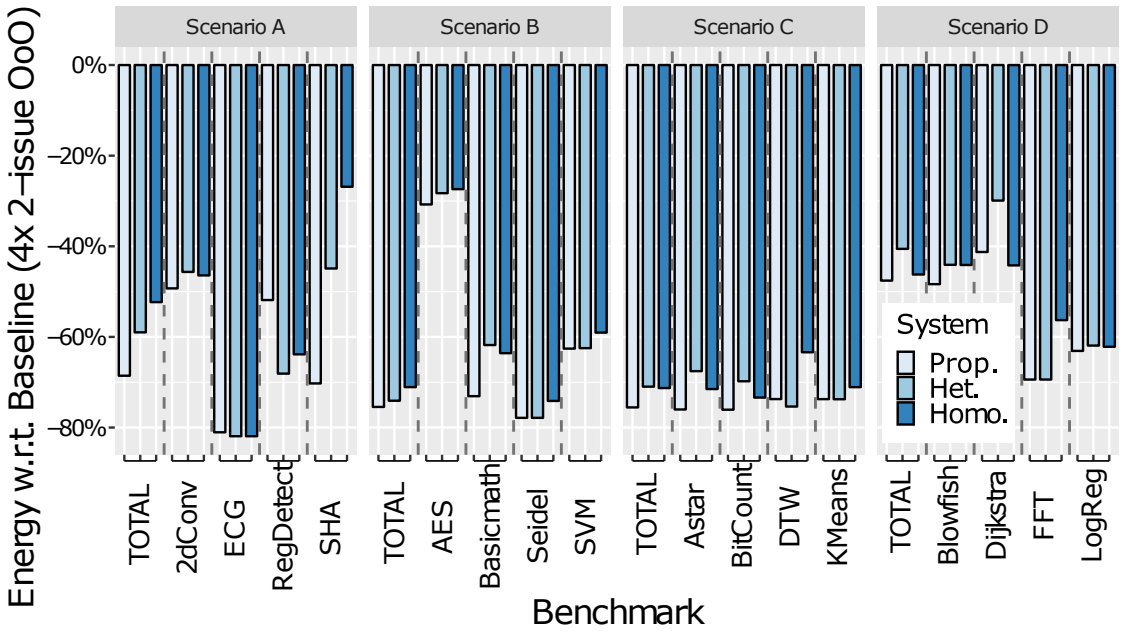


Source: the author

Energy Evaluation. As expected by the results in power dissipation and the acceleration provided by the CGRAs, all the proposed scenarios present reductions in energy consumption. Figure 4.14 presents the energy consumed by the proposed, *heterogeneous*, and *homogeneous* systems w.r.t. the *baseline*. Averaging the savings in energy consumption over all application scenarios, the proposed, *heterogeneous*, and *homogeneous* systems achieved savings of 63.65%, 60.17%, and 58% respectively.

Additionally, we present the energy savings when one considers only the CGRAs. Figure 4.15 displays the increase in energy consumption for the *homogeneous* and *heterogeneous* systems when compared to the proposed system for the four evaluation scenarios

Figure 4.14: Energy w.r.t. baseline for the proposed, homogeneous, and heterogeneous systems (lower is better).



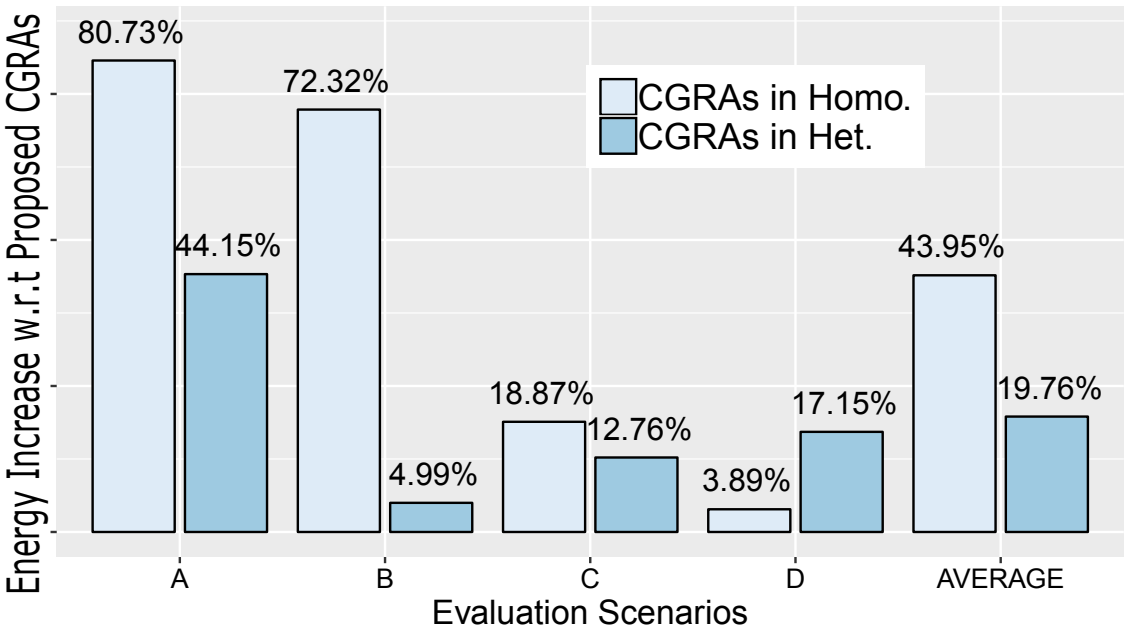
Source: the author.

and their average. Resulting from the savings in power dissipation and ability to adapt to the applications' ILP requirements, architecture employing resource management is more energy efficient than both *homogeneous* and *heterogeneous* systems in all evaluation scenarios. The greater savings, ranging from 18.87% to 80.73% are observed in the comparison with the *homogeneous* CGRAs. As for the *heterogeneous* system, savings from 4.99% to 44.15% are observed. On average, the *homogeneous* and *heterogeneous* systems consume 43.95% and 19.76% more energy than the proposed system, respectively.

Energy-Delay Product Evaluation. We provide a final comparison of the proposed system with its *heterogeneous* and *homogeneous* counterparts. Figure 4.16 displays the energy-delay product (EDP) of the CGRAs in the *heterogeneous* and *homogeneous* systems w.r.t. the managed ones. Given that the average EDP increases 42.18% for the *homogeneous* and 47.48% for the *heterogeneous* CGRAs, it is clear that by dynamically adapting the CGRAs resources to the applications' needs, the proposed system can effectively provide the energy efficiency of heterogeneous architectures and the performance of homogeneous architectures.

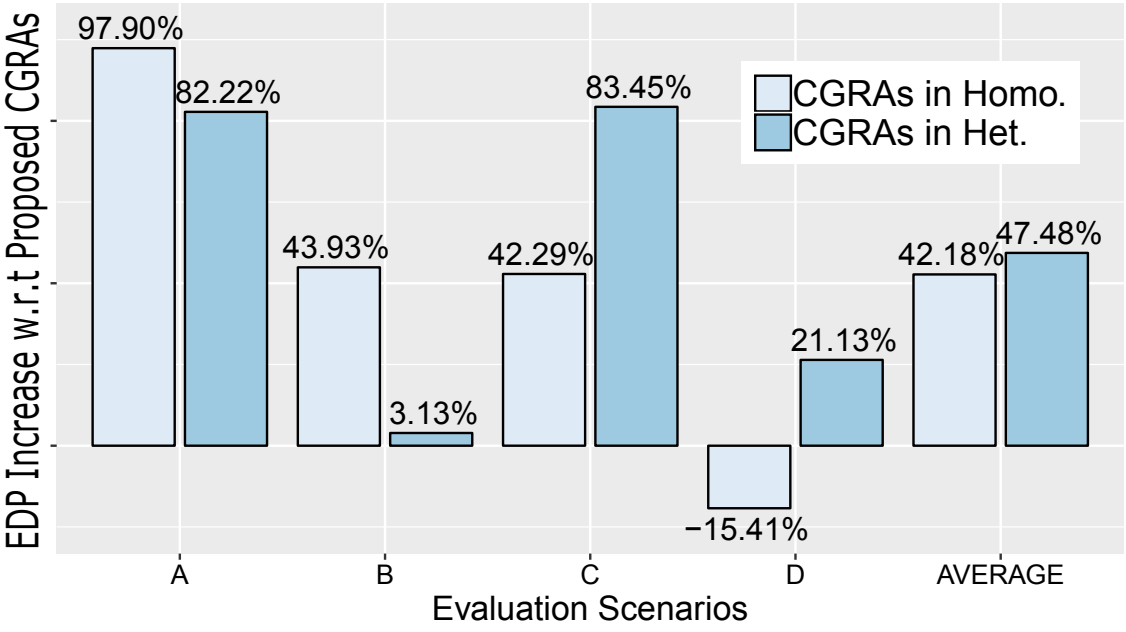
Power Management Area Overhead. After synthesizing the proposed system, it was observed a small area overhead due to the addition of the hardware responsible for the power management scheme since it mainly arises from the PMU module and the BT

Figure 4.15: CGRA's energy consumption in the homogeneous and heterogeneous systems w.r.t. the CGRAs in the proposed system.



Source: the author.

Figure 4.16: CGRA's EDP in the homogeneous and heterogeneous systems w.r.t. the CGRAs in the proposed system.



Source: the author.

replicated mapping units. Specifically, the addition of the PMU incurs in a 2.14% of area overhead.

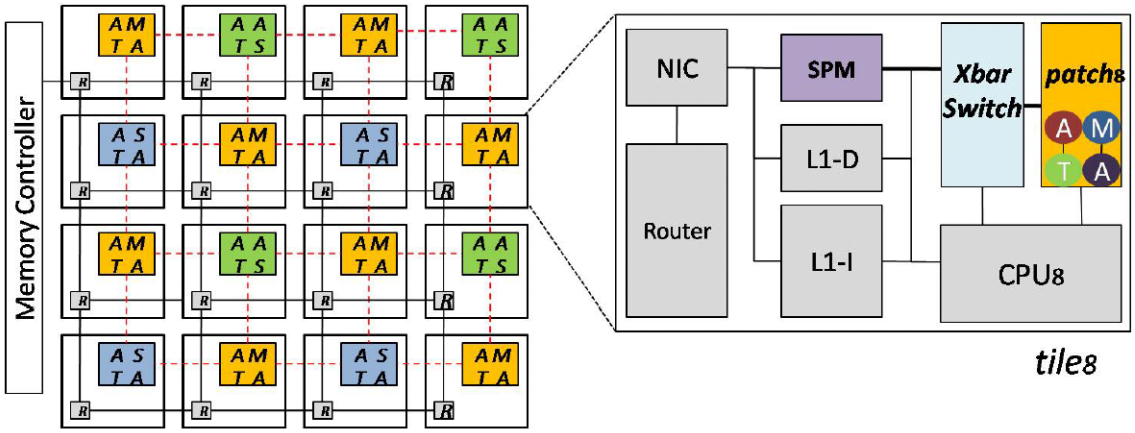
Key Findings. Overall, the proposed system is capable of dynamically and transparently adapting the CGRAs to the applications' workloads while applying power gating

for limiting their power dissipation. Therefore, we bring the power efficiency nature of heterogeneous systems without demanding complex schedulers at performance levels equivalent to what is achieved by homogeneous systems. Specifically, the main results show an average reduction in the power dissipated by the CGRAs of 44.79% when compared to a homogeneous architecture. When compared to a heterogeneous system, the average energy consumed by the CGRAs is reduced in 19.76%. As for EDP, significant gains are reported over both homogeneous (42.18%) and heterogeneous (47.18%) systems.

4.3.3 Comparison with State-of-the-art

To validate this proposal under edge performance constraints, we first present the architecture, which targets the same domain, that is used as benchmark throughout this experiment. The Stitch Architecture (TAN et al., 2018) proposes a set of reconfigurable Instruction Set Extension (ISE) accelerators tightly coupled to ARM in-order processors in a many-core system (Figure 4.17). Its predecessor, the Locus Architecture (TAN et al., 2016), was also designed to work under Edge-tolerable power envelopes. The reconfiguration, in both Locus and Stitch architectures, comes from the instructions added to the ISA that can set the extra functional units called *patches* (in the case of Stitch), which are coupled to the CPUs.

Figure 4.17: Overview of the Stitch architecture.



Source: (TAN et al., 2018).

Precisely, the Stitch's patches can be of one out of three types: AT-MA (consisting of ALUs connected to a local memory access and a multiplier connected to another ALU), AT-SA (consisting of ALUs connected to a local memory access and a shifter connected

to another ALU), and AT-AS (consisting of ALUs connected to a local memory access and an ALU connected to one shifter). There are in total 16 ARM Amber Cores [2] running at 200MHz and interconnected by a 2-D mesh Network-on-Chip (NoC). The patches have a dedicated bufferless NoC with clockless repeaters. The reason for interconnecting multiple patches is that processors not using their "close" patches may yield their resources to form bigger accelerators for other processors running performance-demanding tasks. In other words, the system's reconfigurable units can be *patched* together forming more capable accelerators. However, all patching process is done offline since the ISE approach naturally requires compilation of the application's code. Also, the disposition of patch types in the system is left in charge of the user that has to identify at design time the mix of patches that best match the kernels in the workload.

Comparison with the Stitch architecture. We evaluate the attainable performance by comparing the proposed architecture to Stitch, using the same baseline in (TAN et al., 2018) (an edge-popular GPP-only baseline (Qualcomm, 2019)) under the applications available in (TAN et al., 2016). The quad-core ARM Cortex-A7 (at 1.6GHz) baseline was modeled in the Gem5 simulator.

In Table 4.5, the performance in terms of applications' throughput normalized to the quad-core ARM Cortex-A7 baseline is presented along with the result reported in (TAN et al., 2018). As can be seen, the proposed system, configured to work *under the same power envelope* of its counterpart (140mW upper-bound), achieves higher throughput than the reported in (TAN et al., 2018) when compared to the same baseline. Additionally, the approach proposed comes with the advantage of being fully automated and transparent to the programmer.

Table 4.5: Throughput w.r.t quad-core ARM Cortex A7.

System	Homogeneous	Heterogeneous	Proposed	Stitch (TAN et al., 2018)
Throughput	1.84x	1.40x	1.77x	1.65x

Source: the author

[2] https://opencores.org/projects/amber

5 CONCLUSION

This work presented a resource-aware multicore CGRA architecture for edge applications. Extending on the works by (BECK; RUTZIG; CARRO, 2014) and (BRANDALERO; BECK, 2017), the primary motivation was to improve the energy efficiency of those architectures. Generally speaking, this work is based on the observation that CGRA systems may face significant variations in the utilization of their reconfigurable fabrics when a broad spectrum of applications is faced. As shown in Chapter 4, control flow applications that have low ILP cannot fully explore the benefits of larger CGRAs. At the same time, other applications in the same working set may achieve great acceleration thanks to their higher levels of intrinsic parallelism. In previous works, a careful design space exploration was required to size the CGRA to the workload correctly. However, in modern IoT and Edge environments acquiring knowledge of the workload beforehand can be a difficult task. Furthermore, when considering a multicore scenario, this problem aggravates. Designing the CGRAs as a homogeneous or heterogeneous set of reconfigurable accelerators has a significant impact on metrics like performance, power, and energy.

With the scenario above discussed in mind, it was clear that a dynamic approach to manage CGRA resources was crucial. Thus, this work approaches the problem by introducing changes to the Binary Translator mechanism used in previous works (as described in Chapter 3) and proposing a module to orchestrate the selective power gating of portions of the reconfigurable fabric. Specifically, the changes to the Binary Translator module and the new PMU module aim to provide an online scheme for monitoring performance and utilization of the system' CGRAs. With that information at hand, it is possible to evaluate what parts of the reconfigurable fabric can be powered-off with no significant harm to performance.

In the first experiment, this work explored the use of power gating and the proposed resource management technique in a single-core scenario. This scenario consisted of an in-order Rocket processor coupled to a larger array (compared to the second experiment) in which average savings of energy consumption of 40% were achieved. The increased efficiency is mainly due to a rise in the utilization rate of the CGRA. Specifically, the online and dynamic adaptation of the reconfigurable fabric, at a program phase granularity, granted an average increase of over 50% in the utilization rate for a benchmark composed of a diverse and modern set of applications.

Later, in a second experiment, a multicore scenario was proposed. It enabled a

more complex evaluation of the benefits of dynamically adapting the reconfigurable fabric of a CGRA to the applications at hand. By comparing the proposed approach to a homogeneous architecture (which could achieve higher results in performance) and to a heterogeneous architecture (which could provide a more energy-efficient architecture), it was possible to attest that the online adaptation presented in this work brings together the desired aspects of both approaches. Notably, when focusing on an analysis of the CGRAs only, the reconfigurable fabrics under resource management achieved average reductions in EDP of over 40% when compared to either homogeneous and heterogeneous systems. When compared to homogeneous architectures, these results are mainly due to improvements in power dissipation (44.79% average) at marginal performance penalties. When compared to heterogeneous architectures, the advantages of matching the applications' performance requirements to the CGRAs in a dynamic fashion becomes evident as better performance results are achieved at similar levels of power dissipation, producing improvements in energy consumption of 19.76% (on average).

5.1 Future Work

The work developed for this dissertation opened up many other research opportunities. First, there are still other possible benefits of applying power gate to CGRA resources that can be evaluated. For example, power gating functional units may impact the system's reliability due to beneficial effects on negative bias temperature instability (NBTI). Alternatively, the approach taken to produce CGRA configurations for an array with dynamic size can be employed to continue execution in case that faulty functional units have to be avoided. Furthermore, the resource management technique may include additional, and orthogonal, optimization targets beyond the applications' requirements. For example, the system may enter a low-power phase. And, independently from applications' performance, power gate portions of the CGRA leveraging the mechanisms proposed in this work. Also, power management might be extended to other parts of the architecture like the configuration cache. In the example of a low-power phase, parts of the configuration cache may also be power gated, increasing the margins for power savings.

In the context of multicore CGRA architectures, the adaptability added to the Binary Translator module may be enhanced to enable a shared CGRA. In such a system, a central reconfigurable fabric is shared by the multiple cores. Here, many implemen-

tations paths are worthwhile pursuing. For instance, the architecture may encompass a set of regular dedicated Binary Translators or a central, more complex, shared module responsible for translating basic blocks from all cores. The same idea goes to the configuration cache. As applications may migrate between cores, it will be possible to share the CGRA context (entries of the configuration cache) between cores - in case of a centralized configuration cache. It would also be possible to dedicate smaller and distributed configuration caches to their respective cores, avoiding the necessary hardware controllers for managing a shared cache. A work on shared CGRA may also include exploration of resource managing policies for load balancing, performance, energy, or temperature, for example. Here, several methods are possible. They may include static allocation of sections of the reconfigurable array for specific cores or policies based on dynamic evaluation of applications' array usage.

5.2 Publications

In consequence of the work developed on the course of the Master's program, the following publications have been made.

- G. Korol, M. Jordan, M. Brandalero, M. B. Rutzig, A. C. S. Beck. "Power-Aware Phase Oriented Reconfigurable Architecture". 26th IEEE International Conference on Electronics Circuits and Systems (ICECS). Genova, 2019.
- G. Korol, M. Jordan, R. S. Silva, M. Pereira, M. Brandalero, M. B. Rutzig, A. C. S. Beck. "A Runtime Power-Aware Phase Predictor for CGRAs". International Conference on Reconfigurable Computing and FPGAs (ReConFig). Cancun, 2019.
- G. Korol, M. Jordan, M. Brandalero, M. Hübner, M. B. Rutzig, A. C. S. Beck. "MCEA: A Resource-Aware Multicore CGRA Architecture for the Edge". Field-Programmable Logic and Applications (FPL). Gothenburg, 2020.

Additionally, as a Master's student, the author collaborated in one more publication:

- J. Schwarzrock, M. Jordan, G. Korol, C. C. Oliveira, A. F. Lorenzon, A. C. S. Beck. "On the influence of Data Migration in Dynamic Thread Management of Parallel Applications". Brazilian Symposium on Computing System Engineering (SBESC). Natal, 2019.

REFERENCES

ADEGBIJA, T. et al. Microprocessor optimizations for the internet of things: A survey. **IEEE Trans. on CAD of Integrated Circuits and Systems**, v. 37, n. 1, p. 7–20, 2018.

AGARWAL, A. et al. Leakage power analysis and reduction: models, estimation and tools. **IEE Proceedings - Computers and Digital Techniques**, v. 152, n. 3, p. 353–368, May 2005.

AKBARI, O. et al. PX-CGRA: polymorphic approximate coarse-grained reconfigurable architecture. In: **2018 Design, Automation & Test in Europe Conference & Exhibition, DATE 2018, Dresden, Germany, March 19-23, 2018**. [S.l.: s.n.], 2018. p. 413–418.

ALTMAN, E. R. et al. Advances and future challenges in binary translation and optimization. **Proceedings of the IEEE**, v. 89, n. 11, p. 1710–1722, Nov 2001.

ALTMAN, E. R.; KAELI, D.; SHEFFER, Y. Welcome to the opportunities of binary translation. **Computer**, v. 33, n. 3, p. 40–45, March 2000.

AMALARETHINAM, D. I. G.; JOSPHIN, A. M. Article: Dynamic task scheduling methods in heterogeneous systems: A survey. **International Journal of Computer Applications**, v. 110, n. 6, p. 12–18, January 2015.

AMANO, H. et al. Muccra chips: Configurable dynamically-reconfigurable processors. In: **2007 IEEE Asian Solid-State Circuits Conference**. [S.l.: s.n.], 2007. p. 384–387.

ARM. **big.LITTLE Technology: The Future of Mobile**. 2013. <https://www.arm.com/files/pdf/big_LITTLE_Technology_the_Futue_of_Mobile.pdf>.

ASANOVIć, K. et al. **The Rocket Chip Generator**. [S.l.], 2016.

BACHRACH, J. et al. Chisel: Constructing hardware in a scala embedded language. In: **Proceedings of the 49th Annual Design Automation Conference**. New York, NY, USA: Association for Computing Machinery, 2012. (DAC '12), p. 1216–1225. ISBN 9781450311991.

BAILEY, D. G. **Introduction to Reconfigurable Computing: Architectures, Algorithms, and Applications**. [S.l.]: Springer, 2007.

BAILEY, D. G. **Design for embedded image processing on FPGAs**. [S.l.]: Wiley-IEEE Press, 2011.

BALASUBRAMONIAN, R. et al. CACTI 7: New Tools for Interconnect Exploration in Innovative Off-Chip Memories. **ACM Transactions on Architecture and Code Optimization**, Association for Computing Machinery, New York, NY, USA, v. 14, n. 2, jun. 2017.

BARAT, F.; LAUWEREINS, R. Reconfigurable Instruction Set Processors: A Survey. In: **Proceedings of the 11th IEEE International Workshop on Rapid System Prototyping (RSP 2000), Paris, France, June 21-23, 2000**. [S.l.: s.n.], 2000. p. 168–173.

BARROSO, L. A.; HÖLZLE, U. The case for energy-proportional computing. **IEEE Computer**, v. 40, n. 12, p. 33–37, 2007.

BAUER, L. et al. RISPP: rotating instruction set processing platform. In: **Proceedings of the 44th Design Automation Conference, DAC 2007, San Diego, CA, USA, June 4-8, 2007**. [S.l.: s.n.], 2007. p. 791–796.

BAUER, S. et al. FPGA-GPU architecture for kernel SVM pedestrian detection. In: **2010 IEEE Computer Society Conference on Computer Vision and Pattern Recognition - Workshops**. [S.l.: s.n.], 2010. p. 61–68.

BECCHI, M.; CROWLEY, P. Dynamic thread assignment on heterogeneous multiprocessor architectures. In: **Proceedings of the Third Conference on Computing Frontiers, 2006, Ischia, Italy, May 3-5, 2006**. [S.l.: s.n.], 2006. p. 29–40.

BECK, A. C. S.; CARRO, L. Transparent acceleration of data dependent instructions for general purpose processors. In: **IFIP VLSI-SoC 2007, IFIP WG 10.5 International Conference on Very Large Scale Integration of System-on-Chip, Atlanta, GA, USA, 15-17 October 2007**. [S.l.: s.n.], 2007. p. 66–71.

BECK, A. C. S.; CARRO, L. Reconfigurable acceleration with binary compatibility for general purpose processors. In: ____. **VLSI-SoC: Advanced Topics on Systems on a Chip: A Selection of Extended Versions of the Best Papers of the Fourteenth International Conference on Very Large Scale Integration of System on Chip (VLSI-SoC2007), October 15-17, 2007, Atlanta, USA**. Boston, MA: Springer US, 2009. p. 1–16.

BECK, A. C. S.; CARRO, L. **Dynamic Reconfigurable Architectures and Transparent Optimization Techniques - Automatic Acceleration of Software Execution**. [S.l.]: Springer, 2010.

BECK, A. C. S.; LISBôA, C. A. L.; CARRO, L. **Adaptable Embedded Systems**. [S.l.]: Springer, New York, NY, 2013.

BECK, A. C. S.; RUTZIG, M. B.; CARRO, L. A transparent and adaptive reconfigurable system. **Microprocessors and Microsystems - Embedded Hardware Design**, v. 38, n. 5, p. 509–524, 2014.

BECK, A. C. S. et al. Transparent reconfigurable acceleration for heterogeneous embedded applications. In: **Design, Automation and Test in Europe, DATE 2008, Munich, Germany, March 10-14, 2008**. [S.l.: s.n.], 2008. p. 1208–1213.

BINKERT, N. L. et al. The gem5 simulator. **SIGARCH Computer Architecture News**, Association for Computing Machinery, v. 39, n. 2, p. 1–7, 2011.

BLAKE, G. et al. Evolution of thread-level parallelism in desktop applications. In: **37th International Symposium on Computer Architecture (ISCA 2010), June 19-23, 2010, Saint-Malo, France**. [S.l.: s.n.], 2010. p. 302–313.

BORKAR, S.; CHIEN, A. A. The future of microprocessors. **Commun. ACM**, v. 54, n. 5, p. 67–77, 2011.

BOUWENS, F. et al. Architecture enhancements for the ADRES coarse-grained reconfigurable array. In: **High Performance Embedded Architectures and Compilers, Third International Conference, HiPEAC 2008, Göteborg, Sweden, January 27-29, 2008, Proceedings**. [s.n.], 2008. p. 66–81. Disponível em: <https://doi.org/10.1007/978-3-540-77560-7_6>.

BRANDALERO, M.; BECK, A. C. S. A mechanism for energy-efficient reuse of decoding and scheduling of x86 instruction streams. In: **Design, Automation & Test in Europe Conference & Exhibition, DATE 2017, Lausanne, Switzerland, March 27-31, 2017**. [S.l.: s.n.], 2017. p. 1468–1473.

BRANDALERO, M. et al. Transrec: Improving adaptability in single-isa heterogeneous systems with transparent and reconfigurable acceleration. In: **Design, Automation & Test in Europe Conference & Exhibition, DATE 2019, Florence, Italy, March 25-29, 2019**. [S.l.: s.n.], 2019. p. 582–585.

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.

CHADHA, R.; BHASKER, J. **Architectural Techniques for Low Power**. [S.l.]: Springer New York, 2012. 93–111 p.

CHOI, R. et al. Fabrication of high quality ultra-thin hfo/sub 2/ gate dielectric mosfets using deuterium anneal. In: **Digest. International Electron Devices Meeting,**. [S.l.: s.n.], 2002. p. 613–616. ISSN null.

CHUNG, E. S. et al. Single-chip heterogeneous computing: Does the future include custom logic, fpgas, and gpgpus? In: **43rd Annual IEEE/ACM International Symposium on Microarchitecture, MICRO 2010, 4-8 December 2010, Atlanta, Georgia, USA**. [S.l.: s.n.], 2010. p. 225–236.

CLABES, J. et al. Design and implementation of the power5 microprocessor. In: **Proceedings. 41st Design Automation Conference, 2004.** [S.l.: s.n.], 2004. p. 670–672.

CLARK, L. T. et al. An embedded 32-b microprocessor core for low-power and high-performance applications. **IEEE Journal of Solid-State Circuits**, v. 36, n. 11, p. 1599–1608, Nov 2001.

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)**. [S.l.: s.n.], 2004. p. 30–40.

COMPTON, K.; HAUCK, S. Reconfigurable computing: a survey of systems and software. **ACM Comput. Surv.**, v. 34, n. 2, p. 171–210, 2002.

CONG, J. et al. Accelerator-rich architectures: Opportunities and progresses. In: **The 51st Annual Design Automation Conference 2014, DAC '14, San Francisco, CA, USA, June 1-5, 2014**. [S.l.: s.n.], 2014. p. 180:1–180:6.

CRAEYNEST, K. V. et al. Fairness-aware scheduling on single-isa heterogeneous multi-cores. In: **Proceedings of the 22nd International Conference on Parallel**

Architectures and Compilation Techniques, Edinburgh, United Kingdom, September 7-11, 2013. [S.l.: s.n.], 2013. p. 177–187.

DAS, S.; MARTIN, K. J. M.; COUSSY, P. Context-memory aware mapping for energy efficient acceleration with cgras. In: **Design, Automation & Test in Europe Conference & Exhibition, DATE 2019, Florence, Italy, March 25-29, 2019**. [S.l.: s.n.], 2019. p. 336–341.

DENNARD, R. H. 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.

FAN, T. et al. Energy aware edge computing: A survey. In: **High-Performance Computing Applications in Numerical Simulation and Edge Computing**. Singapore: Springer Singapore, 2019. p. 79–91.

FLAMM, K. Has moore's law been repealed? an economist's perspective. **Computing in Science Engineering**, v. 19, n. 2, p. 29–40, Mar 2017.

FRITTS, J. E. et al. Mediabench II video: Expediting the next generation of video systems research. **Microprocessors and Microsystems - Embedded Hardware Design**, v. 33, n. 4, p. 301–318, 2009.

GOVINDARAJU, V. et al. DySER: Unifying Functionality and Parallelism Specialization for Energy-Efficient Computing. **IEEE Micro**, v. 32, n. 5, p. 38–51, 2012.

GUPTA, S. et al. Bundled execution of recurring traces for energy-efficient general purpose processing. In: **44rd Annual IEEE/ACM International Symposium on Microarchitecture, MICRO 2011, Porto Alegre, Brazil, December 3-7, 2011**. [S.l.: s.n.], 2011. p. 12–23.

GUTHAUS, M. R. et al. MiBench: A free, commercially representative embedded benchmark suite. In: **WWC**. [S.l.: s.n.], 2001. p. 3–14.

HAMEED, R. et al. Understanding sources of inefficiency in general-purpose chips. In: **37th International Symposium on Computer Architecture (ISCA 2010), June 19-23, 2010, Saint-Malo, France**. [S.l.: s.n.], 2010. p. 37–47.

HARRIS, J. Direct-coupled transistor logic circuitry in digital computers. In: **1956 IEEE International Solid-State Circuits Conference. Digest of Technical Papers**. [S.l.: s.n.], 1956. p. 9–9. ISSN null.

HENNESSY, D. A. P. J. L. **Computer Architecture, Sixth Edition: A Quantitative Approach**. 6. ed. [S.l.]: Morgan Kaufmann, 2017. (The Morgan Kaufmann Series in Computer Architecture and Design).

HENNING, J. L. SPEC CPU2000: Measuring CPU Performance in the New Millennium. **Computer**, v. 33, n. 7, p. 28–35, jul. 2000. ISSN 0018-9162.

HU, Z. et al. Microarchitectural techniques for power gating of execution units. In: **Proceedings of the 2004 International Symposium on Low Power Electronics and Design, 2004, Newport Beach, California, USA, August 9-11, 2004**. [S.l.: s.n.], 2004. p. 32–37.

HWU, W.; PATEL, S. J. Accelerator architectures A ten-year retrospective. **IEEE Micro**, v. 38, n. 6, p. 56–62, 2018.

INTEL. **SA-1100 Microprocessor**. [S.l.], 1998.

KHAN, M. U. K.; SHAFIQUE, M.; HENKEL, J. Power-efficient accelerator allocation in adaptive dark silicon many-core systems. In: **Proceedings of the 2015 Design, Automation & Test in Europe Conference & Exhibition, DATE 2015, Grenoble, France, March 9-13, 2015**. [S.l.: s.n.], 2015. p. 916–919.

KISSLER, D. et al. Scalable Many-Domain Power Gating in Coarse-Grained Reconfigurable Processor Arrays. **Embedded Systems Letters**, v. 3, n. 2, p. 58–61, 2011.

KOÇBERBER, Y. O. et al. Meet the walkers: accelerating index traversals for in-memory databases. In: **The 46th Annual IEEE/ACM International Symposium on Microarchitecture, MICRO-46, Davis, CA, USA, December 7-11, 2013**. [S.l.: s.n.], 2013. p. 468–479.

KULKARNI, A. et al. An energy-efficient programmable manycore accelerator for personalized biomedical applications. **IEEE Transactions on Very Large Scale Integration (VLSI) Systems**, v. 26, n. 1, p. 96–109, Jan 2018.

KUMAR, R. et al. Single-isa heterogeneous multi-core architectures: The potential for processor power reduction. In: **Proceedings of the 36th Annual International Symposium on Microarchitecture, San Diego, CA, USA, December 3-5, 2003**. [S.l.: s.n.], 2003. p. 81–92.

LAMBRECHTS, A.; RAGHAVAN, P.; JAYAPALA, M. Energy - aware interconnect - exploration of coarse grained reconfigurable processors. In: **WASP 4th Workshop on Application Specific Processors**. [S.l.: s.n.], 2005.

LECUN, Y. et al. Gradient-based learning applied to document recognition. **Proceedings of the IEEE**, Nov 1998.

LEVERICH, J. et al. Power management of datacenter workloads using per-core power gating. **Computer Architecture Letters**, v. 8, n. 2, p. 48–51, 2009.

LI, Y. et al. A 34-fps 698-gop/s/w binarized deep neural network-based natural scene text interpretation accelerator for mobile edge computing. **IEEE Transactions on Industrial Electronics**, v. 66, n. 9, p. 7407–7416, Sep. 2019. ISSN 1557-9948.

LIU, F. et al. Dynaspam: dynamic spatial architecture mapping using out of order instruction schedules. In: **Proceedings of the 42nd Annual International Symposium on Computer Architecture, Portland, OR, USA, June 13-17, 2015**. [S.l.: s.n.], 2015. p. 541–553.

LIU, L. et al. A survey of coarse-grained reconfigurable architecture and design: Taxonomy, challenges, and applications. **ACM Comput. Surv.**, Association for Computing Machinery, New York, NY, USA, v. 52, n. 6, out. 2019.

LIU, Z.; KURSUN, V. Leakage power characteristics of dynamic circuits in nanometer CMOS technologies. **IEEE Trans. on Circuits and Systems**, v. 53-II, n. 8, p. 692–696, 2006.

LYSECKY, R.; STITT, G.; VAHID, F. Warp processors. **ACM Transaction on Design of Automated Electroninc Systems**, Association for Computing Machinery, New York, NY, USA, v. 11, n. 3, p. 659–681, jul. 2006.

LYSECKY, R.; VAHID, F. A configurable logic architecture for dynamic hardware/software partitioning. In: **Proceedings Design, Automation and Test in Europe Conference and Exhibition**. [S.l.: s.n.], 2004. v. 1, p. 480–485 Vol.1.

LYSECKY, R.; VAHID, F. Design and implementation of a microblaze-based warp processor. **ACM Trans. Embed. Comput. Syst.**, Association for Computing Machinery, New York, NY, USA, v. 8, n. 3, abr. 2009.

MARTINS, M. G. A. et al. Open cell library in 15nm freepdk technology. In: **Proceedings of the 2015 Symposium on International Symposium on Physical Design, ISPD 2015, Monterey, CA, USA, March 29 - April 1, 2015**. [S.l.: s.n.], 2015. p. 171–178.

MEI, B. et al. ADRES: an architecture with tightly coupled VLIW processor and coarse-grained reconfigurable matrix. In: **Field Programmable Logic and Application, 13th International Conference, FPL 2003, Lisbon, Portugal, September 1-3, 2003, Proceedings**. [S.l.: s.n.], 2003. p. 61–70.

MINISKAR, N. R. et al. Intra mode power saving methodology for cgra-based reconfigurable processor architectures. In: **IEEE International Symposium on Circuits and Systems, ISCAS 2016, Montréal, QC, Canada, May 22-25, 2016**. [S.l.: s.n.], 2016. p. 714–717.

MITTAL, S. A survey of techniques for architecting and managing asymmetric multicore processors. **ACM Comput. Surv.**, v. 48, n. 3, p. 45:1–45:38, 2016.

MOHAMMADI, M. et al. Deep learning for iot big data and streaming analytics: A survey. **IEEE Communications Surveys and Tutorials**, v. 20, n. 4, p. 2923–2960, 2018.

MOORE, G. E. Cramming more components onto integrated circuits, reprinted from electronics, volume 38, number 8, april 19, 1965, pp.114 ff. **IEEE Solid-State Circuits Society Newsletter**, v. 11, n. 3, p. 33–35, Sep. 2006.

MÜCK, T.; SARMA, S.; DUTT, N. D. Run-dmc: Runtime dynamic heterogeneous multicore performance and power estimation for energy efficiency. In: **2015 International Conference on Hardware/Software Codesign and System Synthesis, CODES+ISSS 2015, Amsterdam, Netherlands, October 4-9, 2015**. [S.l.: s.n.], 2015. p. 173–182.

MUKHOPADHYAY, S. et al. Leakage in nanometer scale cmos circuits. In: **2003 International Symposium on VLSI Technology, Systems and Applications. Proceedings of Technical Papers. (IEEE Cat. No.03TH8672)**. [S.l.: s.n.], 2003. p. 307–312.

OH, H.; HA, S. A static scheduling heuristic for heterogeneous processors. In: **Euro-Par '96 Parallel Processing, Second International Euro-Par Conference, Lyon, France, August 26-29, 1996, Proceedings, Volume II**. [S.l.: s.n.], 1996. p. 573–577.

OLUKOTUN, K.; HAMMOND, L. The future of microprocessors. **ACM Queue**, v. 3, n. 7, p. 26–29, 2005.

OLUKOTUN, K. et al. The case for a single-chip multiprocessor. In: **ASPLOS-VII Proceedings - Seventh International Conference on Architectural Support for Programming Languages and Operating Systems, Cambridge, Massachusetts, USA, October 1-5, 1996**. [S.l.: s.n.], 1996. p. 2–11.

PADMANABHA, S. et al. Trace based phase prediction for tightly-coupled heterogeneous cores. In: **The 46th Annual IEEE/ACM International Symposium on Microarchitecture, MICRO-46, Davis, CA, USA, December 7-11, 2013**. [S.l.: s.n.], 2013. p. 445–456.

PAGANI, S. et al. Energy efficiency for clustered heterogeneous multicores. **IEEE Trans. Parallel Distrib. Syst.**, v. 28, n. 5, p. 1315–1330, 2017.

PAL, A. **Low-Power VLSI Circuits and Systems**. [S.l.]: Springer Publishing Company, Incorporated, 2014. ISBN 8132219368.

PANDA, P. R. et al. **Power-Efficient System Design**. 1st. ed. [S.l.]: Springer Publishing Company, Incorporated, 2010. ISBN 1441963871.

PATEL, S. J.; HWU, W. W. Guest editors' introduction: Accelerator architectures. **IEEE Micro**, v. 28, n. 4, p. 4–12, 2008.

POLLACK, F. J. New microarchitecture challenges in the coming generations of cmos process technologies (keynote address)(abstract only). In: **Proceedings of the 32nd Annual ACM/IEEE International Symposium on Microarchitecture**. USA: IEEE Computer Society, 1999. (MICRO 32), p. 2. ISBN 076950437X.

POUCHET, L.-N. **PolyBench/C: the Polyhedral Benchmark suite**. 2019. Http://web.cse.ohio-state.edu/ pouchet.2/software/polybench/.

PRICOPI, M. et al. Power-performance modeling on asymmetric multi-cores. In: **International Conference on Compilers, Architecture and Synthesis for Embedded Systems, CASES 2013, Montreal, QC, Canada, September 29 - October 4, 2013**. [S.l.: s.n.], 2013. p. 15:1–15:10.

Qualcomm. **Qualcomm Snapdragon Wear 20100: Wearables Processor**. 2019. Https://www.qualcomm.com/media/documents/files/snapdragon-wear-2100-processor-product-brief.pdf.

RABAEY, J. M.; CHANDRAKASAN, A.; NIKOLIC, B. **Digital integrated circuits- A design perspective**. 2ed. ed. [S.l.]: Prentice Hall, 2004.

RABAEY, J. M.; PEDRAM, M.; LANDMAN, P. E. **Low Power Design Methodologies**. 1st.. ed. [S.l.]: Springer US, 1996.

RANGAN, K. K.; WEI, G.; BROOKS, D. M. Thread motion: fine-grained power management for multi-core systems. In: **36th International Symposium on Computer Architecture (ISCA 2009), June 20-24, 2009, Austin, TX, USA**. [S.l.: s.n.], 2009. p. 302–313.

REAGEN, B. et al. MachSuite: Benchmarks for accelerator design and customized architectures. In: **IISWC**. [S.l.: s.n.], 2014. p. 110–119.

RUTZIG, M. B.; BECK, A. C. S.; CARRO, L. Creams: An embedded multiprocessor platform. In: **Reconfigurable Computing: Architectures, Tools and Applications - 7th International Symposium, ARC 2011, Belfast, UK, March 23-25, 2011. Proceedings**. [S.l.: s.n.], 2011. p. 118–124.

SADASIVAM, S. K. et al. IBM power9 processor architecture. **IEEE Micro**, v. 37, n. 2, p. 40–51, 2017.

SAITO, Y. et al. Leakage power reduction for coarse grained dynamically reconfigurable processor arrays with fine grained power gating technique. In: **2008 International Conference on Field-Programmable Technology, FPT 2008, Taipei, Taiwan, December 7-10, 2008**. [S.l.: s.n.], 2008. p. 329–332.

SHAFAEI, A. et al. Fincacti: Architectural analysis and modeling of caches with deeply-scaled finfet devices. In: **IEEE Computer Society Annual Symposium on VLSI, ISVLSI 2014, Tampa, FL, USA, July 9-11, 2014**. [S.l.: s.n.], 2014. p. 290–295.

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, 2017.

SHAFIQUE, M. et al. Dark silicon as a challenge for hardware/software co-design. In: **CODES+ISSS**. [S.l.: s.n.], 2014. p. 13:1–13:10.

SHAFIQUE, M. et al. An overview of next-generation architectures for machine learning: Roadmap, opportunities and challenges in the iot era. In: **2018 Design, Automation & Test in Europe Conference & Exhibition, DATE 2018, Dresden, Germany, March 19-23, 2018**. [S.l.: s.n.], 2018. p. 827–832.

SHERWOOD, T. et al. Automatically characterizing large scale program behavior. In: **ASPLOS-X**. [S.l.: s.n.], 2002. p. 45–57.

SHI, W. et al. Edge computing: Vision and challenges. **IEEE Internet of Things Journal**, v. 3, n. 5, p. 637–646, 2016.

SHI, W.; DUSTDAR, S. The Promise of Edge Computing. **IEEE Computer**, v. 49, n. 5, p. 78–81, 2016.

SHIMA, M.; FAGGIN, F.; MAZOR, S. An n-channel 8-bit single chip microprocessor. In: **1974 IEEE International Solid-State Circuits Conference. Digest of Technical Papers**. [S.l.: s.n.], 1974. XVII, p. 56–57.

SOUZA, J. D. et al. A reconfigurable heterogeneous multicore with a homogeneous ISA. In: **2016 Design, Automation & Test in Europe Conference & Exhibition, DATE 2016, Dresden, Germany, March 14-18, 2016**. [S.l.: s.n.], 2016. p. 1598–1603.

TAN, C. et al. Stitch: Fusible heterogeneous accelerators enmeshed with many-core architecture for wearables. In: **45th ACM/IEEE Annual International Symposium on Computer Architecture, ISCA 2018, Los Angeles, CA, USA, June 1-6, 2018**. [S.l.: s.n.], 2018.

TAN, C. et al. Locus: Low-power customizable many-core architecture for wearables. In: **2016 International Conference on Compilers, Architectures and Synthesis for Embedded Systems, CASES 2016, Pittsburgh, Pennsylvania, USA, October 1-7, 2016**. [S.l.: s.n.], 2016.

THEODORIDIS, G.; SOUDRIS, D.; VASSILIADIS, S. **A Survey of Coarse-Grain Reconfigurable Architectures and Cad Tools"**. [S.l.]: Springer Netherlands, 2007. 89–149 p.

VEENDRICK, H. J. M. Short-circuit dissipation of static cmos circuitry and its impact on the design of buffer circuits. **IEEE Journal of Solid-State Circuits**, v. 19, n. 4, p. 468–473, Aug 1984.

WALL, D. W. Limits of instruction-level parallelism. In: **ASPLOS-IV Proceedings - Forth International Conference on Architectural Support for Programming Languages and Operating Systems, Santa Clara, California, USA, April 8-11, 1991**. [S.l.: s.n.], 1991. p. 176–188.

WANLASS, F.; SAH, C. Nanowatt logic using field-effect metal-oxide semiconductor triodes. In: **1963 IEEE International Solid-State Circuits Conference. Digest of Technical Papers**. [S.l.: s.n.], 1963. VI, p. 32–33.

WATKINS, M. A.; ALBONESI, D. H. ReMAP: A Reconfigurable Heterogeneous Multicore Architecture. In: **MICRO**. [S.l.: s.n.], 2010. p. 497–508.

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 2016, Barcelona, Spain, March 12-16, 2016**. [S.l.: s.n.], 2016. p. 138–150.

WEI, L. et al. Design and optimization of low voltage high performance dual threshold cmos circuits. In: **Proceedings 1998 Design and Automation Conference. 35th DAC.** [S.l.: s.n.], 1998. p. 489–494.

WIJTVLIET, M.; WAEIJEN, L.; CORPORAAL, H. Coarse grained reconfigurable architectures in the past 25 years: Overview and classification. In: **International Conference on Embedded Computer Systems: Architectures, Modeling and Simulation, SAMOS 2016, Agios Konstantinos, Samos Island, Greece, July 17-21, 2016**. [S.l.: s.n.], 2016. p. 235–244.

WU, Q. et al. Dynamo: Facebook's data center-wide power management system. In: **2016 ACM/IEEE 43rd Annual International Symposium on Computer Architecture (ISCA)**. [S.l.: s.n.], 2016. p. 469–480.

WU, Q.; PEDRAM, M.; WU, X. Clock-gating and its application to low power design of sequential circuits. **IEEE Transactions on Circuits and Systems I: Fundamental Theory and Applications**, v. 47, n. 3, p. 415–420, March 2000.

WUNDERLICH, R. E. et al. SMARTS: accelerating microarchitecture simulation via rigorous statistical sampling. In: **30th International Symposium on Computer Architecture (ISCA 2003), 9-11 June 2003, San Diego, California, USA**. [S.l.: s.n.], 2003. p. 84–95.

XU, B.; ALBONESI, D. H. Methodology for the analysis of dynamic application parallelism and its application to reconfigurable computing. In: **Proceedings of the Society of Photo-Optical Instrumentation Engineer (SPIE), Volume 3844, p. 78-86 (1999).** [S.l.: s.n.], 1999. v. 3844, p. 78–86.