Towards High-Level Leakage Power Reduction Techniques for FPGAs

by

Rehan Ahmed

M.Sc. Electrical Engineering, Technical University of Munich, Germany, 2009
B.Sc. (Hons) Computer Engineering, Univ. of Eng. and Tech., Pakistan, 2004

A THESIS SUBMITTED IN PARTIAL FULFILLMENT OF
THE REQUIREMENTS FOR THE DEGREE OF

DOCTOR OF PHILOSOPHY

in

THE COLLEGE OF GRADUATE STUDIES

(Electrical Engineering)

THE UNIVERSITY OF BRITISH COLUMBIA

(Okanagan)

July 2015

Abstract

A field-programmable gate array (FPGA) is an integrated circuit (IC) which can be configured to implement any digital circuit. The speed and power-efficiency of an FPGA is better than general-purpose-processors (GPPs) and the flexibility, time-to-market, and low-volume costs of an FPGA are better than application-specific-integrated circuits (ASICs). As FPGAs are implemented using more advanced programming technologies, power, specifically leakage power, has become a first-class concern for many FPGA applications. Towards that end, the work in this thesis proposes software optimizations inside a high-level computer-aided-design (CAD) tool chain and a modeling technique that increases the effectiveness and accessibility of low-power architectural features, such as Dynamic Power Gating and Dynamic Partial Reconfiguration, in reducing the overall static leakage power in an FPGA.

Three research contributions are presented. The first contribution is a high-level synthesis (HLS) based design methodology that targets an FPGA with dynamic power gating support. The proposed methodology automatically finds the coarse-grained (accelerator-level) power gating opportunities in a design expressed in *C* language and exploits the power-gating feature of the FPGA to minimize the static power dissipation.

The second contribution demonstrates the impact of performing power gating at a finer granularity in which individual sub-accelerators within a parent accelerator are power-gated. Results reveal that for some applications, this finer granularity results in more effective power gating, thus providing more static power savings than just turning off the whole accelerator.

Finally, the third contribution presents a model which aids in designing an FPGA-based dynamic partial reconfigurable system in which parts of the FPGA's functionality can be modified at run-time. The time-multiplexing of chip resources enables the use of a smaller FPGA device that helps in lowering the static power dissipation. A modeling approach is proposed that predicts the performance trends and bottlenecks in a reconfigurable datapath, without implementing the system on an FPGA device. This early prediction increases the ability to effectively explore the design space and experiment with various system parameters in less time.

Preface

The research contributions presented in this thesis have been published in various FPGA related research conferences [AWHK15, ABW+14, AH11] and in a book chapter [AH13]. The proposed high-level synthesis (HLS) based methodology and results have been published in paper [ABW+14] and are presented in Chapter 3 of this thesis. The hierarchical dynamic power gating approach in FPGAs, presented in Chapter 4, has been published in paper [AWHK15]. Finally, the model-based performance estimation technique for dynamic partial reconfigurable FPGAs, which is the topic of Chapter 5, was first published in paper [AH11] and later invited for a book chapter published in [AH13].

I would like to recognize my co-author, Assem.A.M.Bsoul, in paper [ABW+14] for providing support in using his proposed dynamic power-gated FPGA architecture [Bso].

The research work presented in paper [AH11] and book chapter [AH13] was carried out under the supervision of my research co-advisor, Dr. Peter Hallschmid from Blackcomb Design Automation, who also provided editorial support for these manuscripts.

For all the work in this thesis, I conducted the research, designed and performed the experiments and prepared the publications under the close supervision and guidance of my research advisor, Prof. Steve Wilton, who also provided editorial support for all of my manuscripts.

List of my Publications:

[ABW+14] Rehan Ahmed, Assem A.M. Bsoul, Steven J.E Wilton, Peter Hallschmid, and Richard Klukas. High-level Synthesis-based Design Methodology for Dynamic Power-Gated FPGAs. In *24th International Conference on Field Programmable Logic and Applications, FPL 2014, Munich, Germany, 2-4 September, 2014*, pages 1–4, 2014. → pages iii, 4, 46, 125

[AH11] Rehan Ahmed and Peter Hallschmid. Modeling and Evaluation of Dynamic Partial Reconfigurable Datapaths for FPGA-

Based Systems Using Stochastic Networks. In *Proceedings of the 2011 21st International Conference on Field Programmable Logic and Applications*, FPL '11, pages 70–75, Washington, DC, USA, 2011. IEEE Computer Society. → pages iii, 125

[AH13] Rehan Ahmed and Peter Hallschmid. Model-based Performance Evaluation of Dynamic Partial Reconfigurable Datapaths for FPGA-based Systems. In *Embedded Systems Design with FPGA*, chapter 5, pages pp 101–123. Springer, 2013. → pages iii, 125

[AWHK15] Rehan Ahmed, Steven J.E Wilton, Peter Hallschmid, and Richard Klukas. Hierarchical Dynamic Power Gating in FPGAs. In *The 11th International Symposium on Applied Reconfigurable Computing, ARC 2015, Bochum, Germany, 13-17 April, 2015*, pages 1–8, 2015. → pages iii, 125

Table of Contents

List of Tables

List of Figures

Glossary of Notation

ASIC	Application-Specific Integrated Circuit
CAD	Computer-Aided Design
CB	Connection Box
DPG	Dynamic Power Gating
DPR	Dynamic Partial Reconfiguration
GPP	General Purpose Processor
HLS	High-level Synthesis
IP	Intellectual Property
LC	Logic Cluster
LE	Logic Element
HDL	Hardware Description Language
IC	Integrated Circuit
LUT	Look-Up Table
PGR	Power-Gated Region
RTL	Register-Transfer Level
SB	Switch Box
SoC	System-on-Chip
FPGA	Field Programmable Gate Array

Acknowledgements

In the name of Allah, The Most Gracious and The Most Merciful.
All praises, glory and gratitude to Almighty Allah (Subhanahu Wa Taala) who gave me courage, strength and patience to successfully complete this thesis work.

This work would not have been completed without the help of many great people. First and foremost, I would like to thank my research mentors, Prof. Steve Wilton and Dr. Peter Hallschmid, for the incredible opportunity and their technical advice, moral support and financial support throughout my Phd studies. They have gone far beyond the call of duty as a supervisor.

I would like to extend my gratitude to my supervisory committee members, Dr. Nicholas Swart, Dr. Wilson Eberle and Dr. Richard Klukas, for serving on my committee and providing useful feedback to improve my work.

Although the names are far too numerous to list here, I thank all the members of the System-on-Chip lab at UBC, my friends at EECE department and especially my group mates for their valuable technical discussions, support and feedback. Thanks for making my time so enjoyable at UBC!

I would like to express my deepest gratitude to my father, mother and the rest of my family members for their tremendous love and constant support throughout my life.

Last, but certainly not least, my heartfelt thanks to my loving wife and my children who have been extremely patient and supportive through the tough times of my Phd studies. Thanks for making this goal happen!

Chapter 1

Introduction

1.1 Research Context

A field programmable gate array (FPGA) is an integrated circuit (IC) in which the functionality can be (re)configured in the field after manufacturing - the user can implement virtually any digital circuit on an FPGA. Since their inception in 1985 [Wik], FPGAs have grown enormously, both in terms of capacity and market, and have seen a radical shift in their role, from a simple glue-logic device to a platform that can now provide 95% throughput improvements compared to pure software implementation [PCC+14].

In the market, FPGAs have continued to make gains over *application specific integrated circuits (ASICs)*, in part, due to the rising cost of fabrication and have proven to be a promising candidate for hardware specialization, providing 3 to 28X improvements in efficiency relative to *general-purpose processors (GPPs)* [CMHM10]. Relative to ASICs and GPPs, FPGAs represent a compromise in which their speed and power-efficiency is better than GPPs [KMN02] and their flexibility, time-to-market, and low-volume costs are better than ASICs [KR06].

The programmability of an FPGA is achieved through a grid of programmable logic blocks which are interconnected through programmable routing resources. Usually SRAM cells are used to control the pass transistors, multiplexers and various other resources that provide configurability. These SRAM cells add a significant area overhead. [YR04] reports that more than 40% of a logic-block area is due to SRAM cells that are used to implement look-up tables (LUTs). An FPGA, because it uses a large number of transistors per logic function to provide the programmability, pays a very high price in terms of power consumption and area-efficiency. Compared to an ASIC, an FPGA implementation typically dissipates 7-14x more power, has a 20-35x larger area overhead and is 3-4x more slower [KR07].

The continuous shrinking of the CMOS process from micrometer to nanometer range has already challenged the FPGA industry with deep submicron effects such as *leakage*. The MOS gate oxide thickness has already reduced to an extent that carriers *tunnel* through the oxide, causing *gate*

leakage. The lower threshold voltage keeps the transistor *ON* even when it is supposed to be in an *OFF* state, causing *sub-threshold leakage*. Together, these leakage paths contribute significantly to the static power dissipation of a MOS gate. That said, the leakage power of each transistor multiplied by the millions of transistors in an FGPA chip increases the dominance of static leakage power in commercial FPGAs built on 28nm (and below) technology node [JHH, Kol, Altd]. This highlights the dominance of leakage power dissipation in FPGAs; reducing leakage power will be an important design objective in future FPGAs.

Power optimization, in general, is a holistic process and requires strategies at various levels in a design process, as depicted in Figure 1.1. These strategies range from the metal-level enhancements in the process technology, innovative low-power techniques at FPGA architectural level and software optimizations inside CAD tool chain. Specific techniques that have been proposed include:

– **Enhancements in the process technology:** These techniques focus on making better transistors by enhancing and improving the process technology that could possibly prevent the leakage in transistors. For example, TSMC has started using high-K metal gate (HKMG) process in its 28nm technology node which is highly energy-efficient [TSM]. Xilinx and Altera have both embraced TSMC's 28nm low power high-k metal gate (HPL) technology in their state-of-the-art

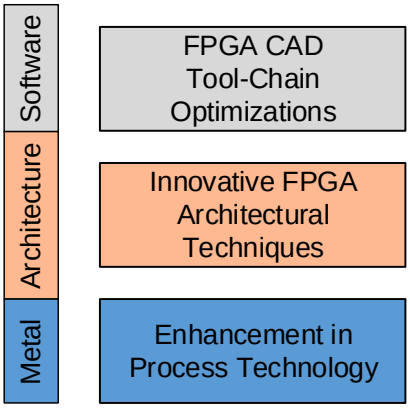


Figure 1.1: An Abstract View of Power Optimization Strategies at Various Levels in a Design Process

FPGAs [JHH, Kol, Altd]. Xilinx reports a saving of 50% static power just by moving to this next generation enhanced process [JHH]

- **Innovative low-power architectural techniques:** These techniques focus on different aspects of FPGA architecture to reduce the static dissipation. These techniques range from employing high-Vt transistors [alt08, Kle09], clock-gating, dynamic voltage scaling [CTL+05, NY13, NNY12], dynamic partial reconfiguration [JHH, Altd] and power-gating parts of the chip [BW10, BW12b, BW12a].
- **Software optimizations inside CAD tool-chain:** These techniques make the CAD tool chain more power-aware. For example, power-aware optimizations in the low-level CAD algorithms such as mapping, placement and routing are proposed by [LW03, VRD+08, DCW09, AN02]. The decisions at these stages are usually steered with the help of abstract or detailed power models [LCHC03, PWY05, AN04].

The work in this thesis proposes software optimizations inside a high-level CAD tool chain and a modeling technique that increases the effectiveness of low-power architectural features, such as Dynamic Power Gating and Dynamic Partial Reconfiguration, in reducing the overall static leakage power in an FPGA.

1.2 Research Motivation

As discussed in the previous section, much of the power dissipation is due to static leakage power, which is dissipated when the chip is powered, regardless of the activity of the device. Leakage power is becoming more pronounced in larger FPGAs which are in excess of a billion transistors; static power increases with the size of the device because each transistor has a leakage component which increases as the size is scaled up.

As FPGAs grow in size, a portion of an FPGA resources remains under-utilized if a design could not completely fill the device. As a result there are *used* and *unused* parts of an FPGA that contribute to the leakage dissipation: the unused parts always remain powered-on and consume significant leakage power and the used (occupied) logic resources, in fact, are only operated for a brief burst of activities and remain *idle* otherwise during which again significant leakage power is dissipated.

Two FPGA architectural techniques, namely *Dynamic Power-Gating (DPG)* and *Dynamic Partial Reconfiguration (DPR)*, have been shown to effectively reduce the static power dissipation. Results in [ABW+14] show that up to 96% reduction in static energy is achieved for individual accelerators by turning them off during their idle periods using dynamic power-gating technique. Likewise, Xilinx reports that upto 80% of static power savings can be obtained by swapping several parts in a logic design using dynamic partial reconfiguration [JHH]. Below, we briefly outline the operation of each technique:

- **Dynamic Power-Gating (DPG):** The basic idea of power gating is to control the supply voltage or ground of the circuit through a sleep transistor. In doing so, the leakage current of the whole circuit is limited by the sleep transistor which significantly reduces the leakage power dissipated. Figure 1.2 depicts an abstract illustration of an FPGA-based dynamic power gating. In DPG technique, parts of the FPGA chip can be turned off *dynamically* at run-time, depending on the behaviour of the program [BW10]; regions of the FPGA logic and routing fabric (called *power-gated regions (PGR)*) can be turned off under the control of an on-chip power-state controller (PSC). The addition of sleep transistors incurs area overhead which can be amortized by choosing the optimum granularity for basic power-gated unit in the architecture. This technique is especially effective for designs in which blocks have long idle times. If used correctly, DPG has the potential to significantly reduce the leakage power of many designs.

  Commercial FPGA vendors provide techniques to turn off unused device primitives such as Block RAMs, DSP Blocks, Clock Managers and I/O pins at compile time but do not support power-gating of the logic blocks at run-time.

- **Dynamic Partial Reconfiguration (DPR):** In DPR, parts of the FPGA's functionality can be modified at run-time. Figure 1.3 illustrates the concept of DPR. Computational tasks (hardware modules) not required to operate simultaneously can be assigned to a single region, usually known as reconfiguration region, and are loaded into FPGA at run-time as needed [LBM+06]. In doing so, the FPGA functionality can be adapted to different phases of execution in order to improve speed, chip density, and energy consumption [LPF10]. The time-multiplexing of chip resources leads to higher implementation density thus leading to reduced per part costs and enables the

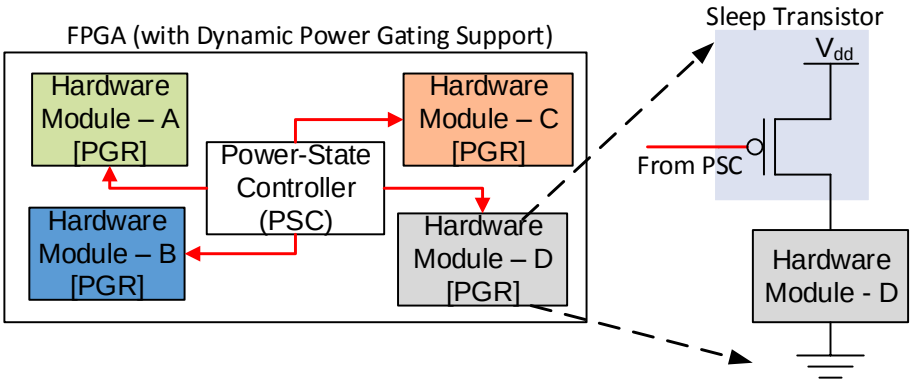


Figure 1.2: An Abstract Illustration of FPGA-based Dynamic Power Gating (DPG). [PGR = Power-Gated Region]

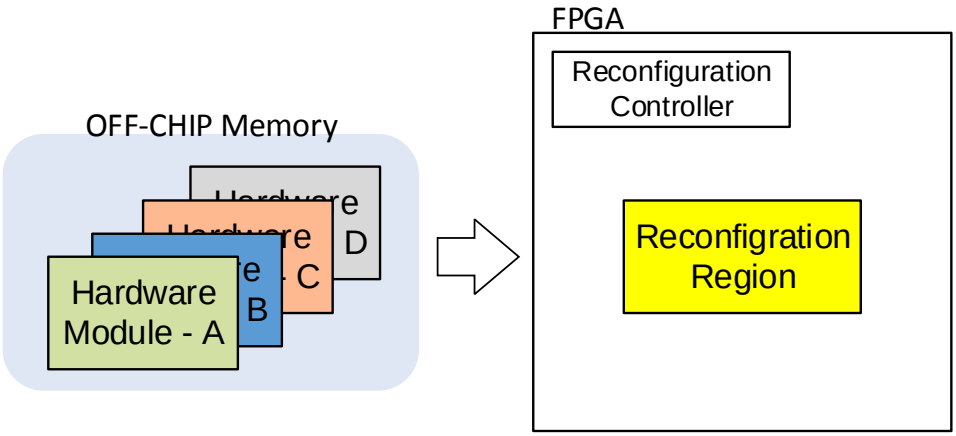


Figure 1.3: An Abstract Illustration of FPGA-based Dynamic Partial Reconfiguration (DPR)

use of a smaller FPGA device that helps in lowering the static power dissipation.

These two techniques are most effective in reducing the leakage dissipation, but making use of them is a daunting task. Consider the case of dynamic power gating: given an application, it is quite challenging to identify *when* and *how long* a certain portion in the design is going to remain idle and whether it is profitable to power-gate it during that period of time; power gating feature incurs an extra energy overhead in switching between

sleep and *active* modes and the logic resources should only be power-gated if it will be idle long enough to compensate for this energy penalty. The effectiveness of dynamic power gating, therefore, depends on the designer's ability to detect and exploit power gating opportunities in the design. In ASIC designs, where power-gating is common, power-gating opportunities are identified manually and typically described in a standard format, such as the Unified Power Format (UPF) or Common Power Format (CPF) [CB]. For FPGAs, many designers may not be willing or able to invest the effort into this manual identification, thus reducing the effectiveness of power gating. Automatically identifying significant power-gating opportunities from a netlist or RTL design remains a difficult challenge.

Similarly, to perform dynamic partial reconfiguration, a partial bitstream is loaded into FPGA's configuration memory from an off-chip memory at run-time. The primary limitation of this technique is the overhead associated with the reconfiguration process and its effect on overall system performance. This overhead can be significant especially for fine-grained architectures in which programming bitstreams are long. Therefore, it becomes incumbent on the designer to assess the reconfiguration overhead before decomposing a large design into mutual exclusive parts such that the overall impact of reconfiguration does not affect the performance when the design is moved to a smaller foot print device; a smaller device means less area which translates into less leakage power. Current design practices allow this overhead to be measured but only after the system has already been built. Predicting reconfiguration performance early is difficult, in part, due to non-deterministic factors such as the sharing of the bus structures with traffic not related to the reconfiguration process.

Termed as the *productivity gap*, an important trend is the growing gap between system complexity and the productivity of engineers to design systems within the required time-to-market [TCL09, Pla08]. The downside of DPG and DPR techniques is that they add an additional level of design complexity and present a formidable challenge to a designer. This further contributes to the productivity gap and does not allow the designer to exploit these techniques to their full potential.

1.3 Thesis Contributions

The work in this thesis focuses on the following two goals:

1. Reducing the design complexity for Dynamic Power-Gated (DPG) FPGAs such that the designer is insulated from low-level decisions and

the overall effectiveness of the technique is increased.

2. Performance estimation of FPGA-based Dynamic Partial Reconfigurable (DPR) systems without requiring the user to actually build the DPR system.

Towards these goals, this thesis makes the following three contributions.

1. High-level Synthesis Compiler-Assisted Design Methodology for Dynamic Power-Gated FPGAs
2. Hierarchical Dynamic Power-Gating in FPGAs
3. Model-based Performance Evaluation of Dynamic Partial Reconfigurable Systems

The first two contributions are geared towards the first goal and the third contribution is towards the second goal. An overview of these contributions is given in the following sections.

1.3.1 HLS Compiler-Assisted Design Methodology for Dynamic Power-Gated FPGAs

The first contribution of this thesis is a *Design Methodology* that targets an FPGA supporting dynamic power gating. The proposed methodology is based on a high-level synthesis compiler framework that automatically detects the power-gating opportunities in a design expressed in C language, and turns-off accelerators when it is deemed profitable.

Dynamic Power-Gating (DPG), in which parts of the FPGA logic fabric are powered-down at run-time, is a promising technique to reduce the static power. Adoption of such emerging dynamic power gating feature in an FPGA remains challenging as the current tool chains to program the FPGA do not support this type of power gating. Moreover, manually identifying profitable power-gating opportunities in an application requires significant design expertise and is time consuming. Although techniques have been proposed for automatically identifying some power gating opportunities from a netlist or dataflow graph, these techniques typically focus on *fine-grained* power-gating opportunities which are as short as several cycles [BKBB05, UO06, IHK11, CXS04].

Realistic DPG architectures, however, have significant overhead when parts of the chip are turned off and on; powering up internal capacitances requires significant power, and the need to throttle power gating circuitry to

avoid in-rush current problems adds delay overhead [BW12b, BW14]. Thus, these architectures are not well-suited to take advantage of the fine-grained power-gating opportunities uncovered by these previous CAD techniques. To take advantage of power-gating technique in FPGAs, therefore, it is necessary to uncover much more *coarse-grained* opportunities – regions of the circuit that can be powered down for significant clock cycles. Intuitively, these regions do exist in a circuit, but they are difficult to detect in an automatic way and therefore finding them remains a difficult challenge.

The first contribution, discussed in Chapter 3, proposes a high-level synthesis-based design framework that finds coarse-grained power-gating opportunities in a design expressed in C language and exploit the power-gating feature of the FPGA to minimize the static power dissipation. This framework is applied to the set of CHStone benchmark suite to demonstrate that power-gating opportunities for hardware accelerators can be identified in an automatic way. Results show a reduction of 14%–61% in static energy for individual accelerators using dynamic power-gating technique. To the best of author's knowledge, this is the first HLS-based framework that automates the design for dynamic power-gated FPGAs.

1.3.2 Hierarchical Dynamic Power-Gating in FPGAs

In the first contribution, described in previous Section 1.3.1, the granularity of power gating is fixed at the *accelerator* level. An entire accelerator is turned off or on as a unit. For very large accelerators, it may be profitable to power gate at a finer granularity to save more static dissipation. If an accelerator has many phases, each of which is implemented by a separate logic circuit (which we call a *sub-accelerator*), then it may be desirable to turn off individual sub-accelerators when other sub-accelerators are running. The second contribution of this thesis, discussed in Chapter 4, shows the impact of performing power-gating at a finer granularity and reveals that for some applications, this finer-granularity results in a more effective power gating, thus providing more static power savings than just turning off the whole accelerator.

Results on CHStone benchmarks show that hierarchical power-gating can save 8%–25% of static energy when the parent and descendant accelerators are power-gated independently such that the parent accelerator is power-gated while the sub-accelerator runs.

1.3.3 Model-based Performance Evaluation of Dynamic Partial Reconfigurable Systems

The third and final contribution, discussed in Chapter 5, is a flexible approach for modeling the dynamic partial reconfiguration (DPR) datapath based on *Queueing Theory* to estimate performance trends and bottlenecks of the reconfiguration process while considering the impact of shared resources on it, from an abstract queueing-network model.

Dynamic partial-reconfigurable (DPR) FPGAs have the property that all or part of their functionality can be time-multiplexed at run-time. In doing so, a large design can be fit into a smaller device instead which reduces the number of transistors used in a design and thus reduces the static power dissipation.

DPR is achieved by dynamically transferring partial configuration bitstreams from off-chip memory to FPGA configuration memory via a specialized *datapath*. The performance of this datapath can have a significant impact on the overall system performance and should be considered early in the design cycle. Unfortunately, performance measures for such systems can typically be determined only after development. Such measures are heavily dependent upon the detailed characteristics of the datapath and on the particular workload imposed on the system during measurement and thus can only be used to make predictions for systems similar to that used for initial measurements.

In this contribution, we outline an approach to model the DPR datapath early in the design cycle using Queueing Theory. This modeling approach is essential for experimenting with system parameters without implementing and running them on a FPGA device. Two case studies are provided to demonstrate the usefulness and flexibility of the modeling scheme.

1.4 Thesis Organization

The organization of the remainder of this thesis is depicted in Figure 1.4. Chapter 2 reviews several topics which forms the background material relevant to the research in this thesis: it reviews the FPGA architecture and conventional CAD tool flow, high-level synthesis (HLS) based FPGA designing, includes a discussion on static and dynamic power consumption in CMOS circuits in general and in FPGAs in particular. It also reviews the fundamentals of Queueing Theory which forms the basis for Chapter 5. Finally, it covers related work on HLS-based design methodology for dynamic power-gated architectures and model-based performance evaluation

of dynamic partial reconfigurable systems. The main research contributions, discussed in Section 1.3, are separated into:

– HLS-based methodology for dynamic power-gated (DPG) FPGA. Chapter 3 and 4 presents the main contributions in this leg.

– Modeling of Dynamic Partial Reconfigurable (DPR) FPGA which is presented in Chapter 5.

Each of the above contribution is presented at its own - the described technique and experimental results are presented together in its own chapter.

Finally, Chapter 6 presents concluding remarks and suggestions for future work.

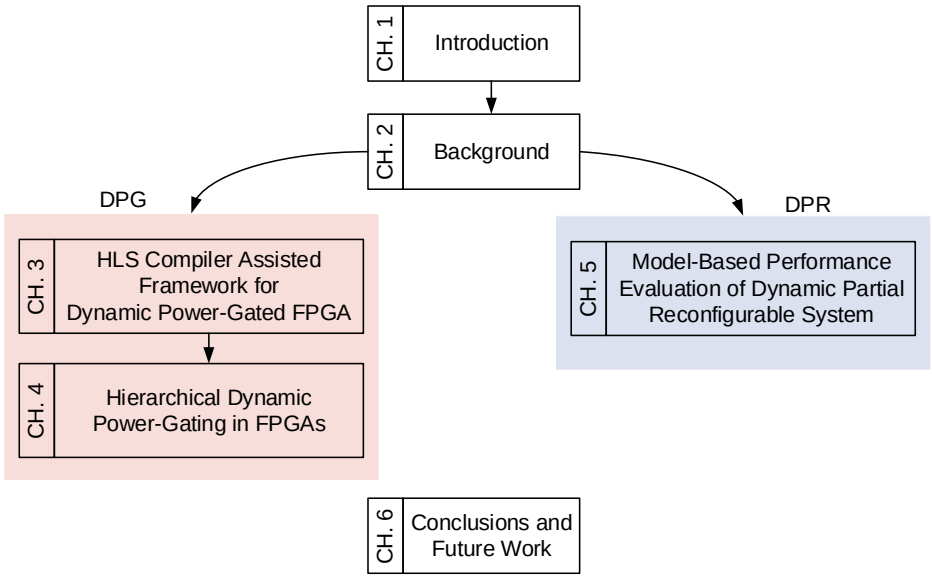


Figure 1.4: Thesis Organization

Chapter 2

Background and Related Work

This chapter presents the background and related work material that forms the basis for the research presented in the later chapters. Section 2.1 provides a brief review of FPGA architecture and the associated CAD flow. Section 2.2 then introduces the emerging high-level synthesis (HLS) based FPGA design methodology that forms the basis for research presented in Chapter 3 and 4. Section 2.3 reviews power dissipation in CMOS circuits, covering both dynamic, as well as static power. Section 2.4 examines sources of power dissipation in an FPGA and review various mitigating techniques. Section 2.5 presents the key concepts in Queueing Theory as it forms the basis of research presented in Chapter 5. Finally, Section 2.6 provides an overview of the related work in context of the contributions of this thesis.

2.1 Field Programmable Gate Array (FPGA): A Brief Review

Embedded-systems are special-purpose designs intended for the computational demand for a specific application or application domain. Such systems have to obey specific design constraints, for example power consumption, speed, chip-area and time-to-market etc. The choice of implementation technology for an embedded system is usually based on the degree of *flexibility* and *performance* required.

Application-Specific Integrated Circuits (ASICs) are integrated circuits designed to implement a particular application with specific performance requirements. They are well-suited for low-power, high-density, and high-speed applications. However, ASICs require a time consuming and costly design process along with expensive manufacturing processes which can only be justified when high volumes are targeted.

In contrast, an application can be implemented on a General-Purpose-Processor (GPP) that can execute a broad set of applications with reason-

ably high performance. For a particular application, however, they perform an order of magnitude worse than ASIC with respect to both speed and power-efficiency. A significant reason for the performance limitation of GPPs is the sequential nature by which operations are performed.

A Field-Programmable-Gate-Array (FPGA) is an integrated circuit designed to be (re)configured in the field after manufacturing [HD07] and is a promising candidate for hardware specialization, providing 3 to 28X improvements in efficiency relative to general-purpose processors [CMHM10]. Relative to ASICs and GPPs, FPGAs represent a compromise in which their speed and power-efficiency is better than GPPs [KMN02] and their flexibility, time-to-market, and low-volume costs are better than ASICs [KR06].

In the market, FPGAs have continued to make gains on ASICs, in part, due to the rising cost of fabrication. Additionally, vendors have developed ways of closing the performance gap between FPGAs and ASICs. Modern FPGA platforms integrate a reconfigurable fabric with general-purpose processors and dedicated hardware blocks on a single FPGA chip [Xil08, Xil12b, Xil12a, Alt15, Alt14]. Since such platforms can be used to build arbitrary hardware by changing the hardware configuration at design-time, the compute intensive parts of the application are offloaded to dedicated hardware blocks and less intense parts remain on the general-purpose processor. In this way, the architecture can adapt to the application, combining hardware performance with software flexibility.

2.1.1 FPGA Architecture

The general structure of an FPGA architecture is depicted in Figure 2.1. It is composed of three main types of resources, namely:

- Programmable logic blocks
- Routing Resources
- Input/Output (I/O) Interfacing Pins

2.1.1.1 Programmable Logic Block

A logic block is a basic component that can be programmed with any kind of logic function. These logic blocks are organized in a two-dimensional array and are interconnected with programmable routing channels. Thus, a complete digital circuit is implemented by programming each of these logic blocks with parts of the logic functions making up the complete circuit.

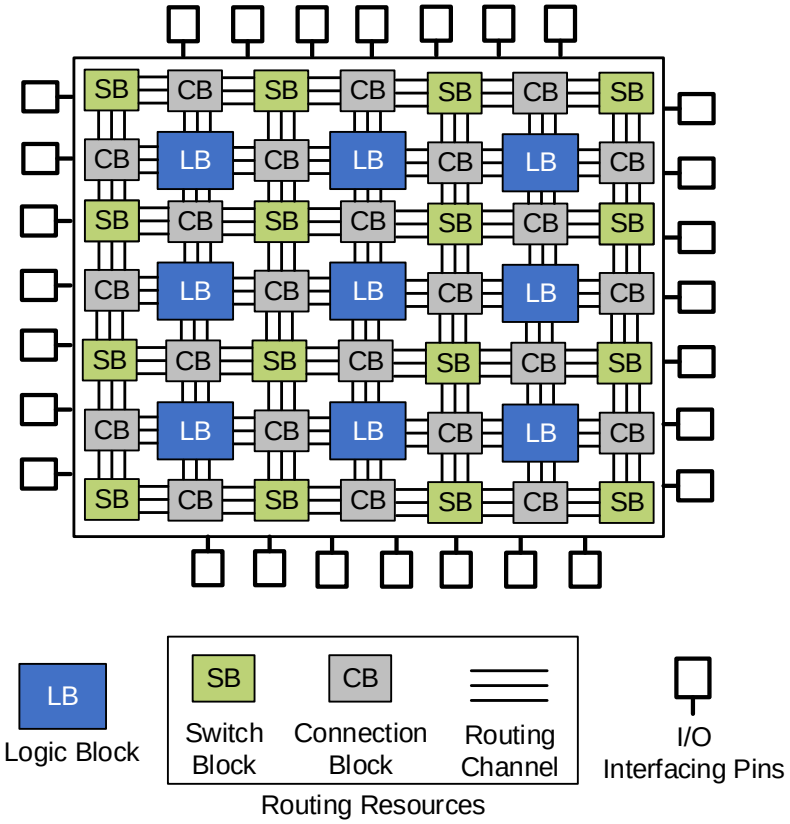


Figure 2.1: An Abstract view of Island-Style FPGA Architecture

Internally, a logic block is implemented as a programmable *Look-up Table (LUT)*. This programmability is achieved by using memory cells to implement a particular function. The number of memory cells in a LUT is defined by it's number of inputs. An n-input LUT requires 2^n memory cells and a 2^n-input multiplexer which forwards the contents of one of the memory cells to the output of LUT. Thus, an n-input LUT is capable of implementing any logic function defined by n variables. For example, Figure 2.2 shows a 3-input LUT implementing an arbitrary function $f = XYZ + \bar{X}\bar{Y}\bar{Z}$.

Besides a LUT, a logic block also has a flip-flop which stores the output from the LUT. A multiplexer provides a mechanism to programmably connect either the LUT output or the flip-flop output to the logic block output, as depicted in Figure 2.2.

Modern FPGAs combine multiple logic blocks together to form a *logic-*

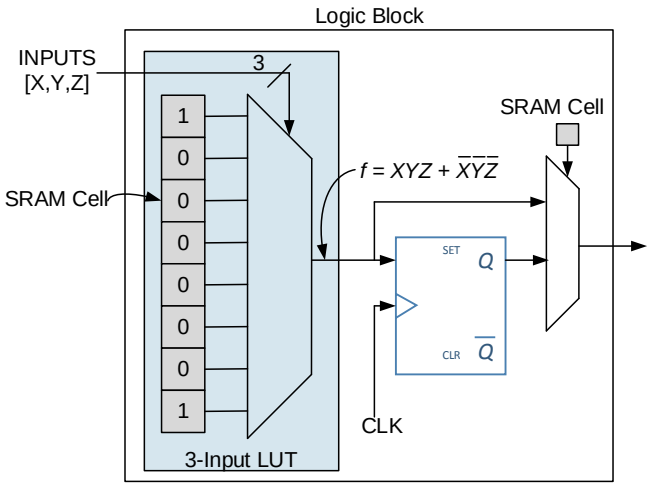


Figure 2.2: An Abstract view of Logic Block Architecture having 3-input LUT

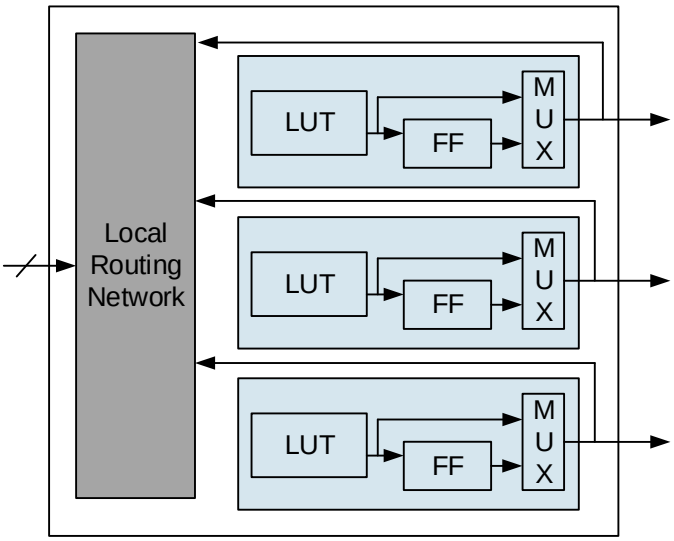


Figure 2.3: An Abstract view of Logic Cluster Architecture

cluster. The logic blocks in a cluster are interconnected through a local routing network, as depicted in Figure 2.3. The logic-clusters are called by various names by different vendors, for example, Altera calls it Logic Array Block (LAB)[Arca] and Xilinx refers to it as a Configurable Logic Block (CLB) [Arcb].

2.1.1.2 Routing Resources

The logic resources in an FPGA are connected to each other and to input/output interfacing blocks through programmable routing resources. The flexible nature of the intra-chip routing resources provide the ability to implement any kind of user circuit with varying routing demands.

The routing resources consist of interconnection wires and programmable *switch-blocks (SBs) & connection-blocks (CBs)*. The interconnection wires are arranged as vertical and horizontal channels on all four sides of the logic resources. There is a switch-block at every intersection of horizontal and vertical routing channel and provides connectivity between the incident wires through programmable switches. The input and output pins of the logic blocks are connected to the routing channels through programmable connection-block. Figure 2.1 shows an abstract arrangement of switch blocks and connection blocks in an island-style FPGA.

A routing channel is composed of wires. The number of wires contained in a routing channel is called the *channel width*, which is denoted by W. In a switch block, the number of wires to which each incident wire can connect is called the *switch block flexibility*, which is denoted by F_s. Similarly, the number of wires in each routing channel to which the pins of an adjacent logic block can connect is called as *connection block flexibility*, which is denoted by F_c. Another important parameter in routing is the *wire length*, denoted by L, which is the number of logic blocks a wire spans before terminating at a switch block. Modern FPGAs use a combination of long and short length wires to balance area, delay and more importantly flexibility of the routing network.

2.1.1.3 I/O Interfacing Pins

The input/output (I/O) interfacing pins allow an FPGA to communicate with the peripheral circuitry to which it is attached. The I/O pins in a modern commercial FPGA support a wide variety of communication and voltage standards that enable it to connect to a multitude of devices without requiring any additional intermediate interfaces [Xilb, Alte].

2.1.1.4 Embedded Hard Blocks

In addition to the three basic resources described previously, modern commercial FPGAs also contain hard logic blocks which provide improved performance and area compared to their counterparts implemented otherwise using soft-logic. These hard blocks include dedicated multipliers, adders, on-chip memories and hard-processors [Xil12a].

2.1.2 CAD for FPGAs

The FPGA design flow starts with the design describing the functionality in a textual language. Conventionally, the design intent is expressed using hardware description languages (HDLs) such as Verilog and VHDL, which are around for many years now. However, more abstract methods for expressing the design in high-level languages, such as C/C++ [Xila, Alta, Exp, Cad, Cat] and BlueSpec [Blu], are emerging. Regardless of the design entry method, the underlying steps in a typical CAD tool-chain for FPGA remains the same. Below, the steps involved in a typical FPGA CAD tool chain are briefly reviewed.

Figure 2.4 shows the various stages that make up the FPGA CAD flow. The logic *synthesis* is the first stage in the CAD flow during which an abstract description of the circuit, typically expressed in a hardware description language (HDL), is transformed into a netlist of *technology-independent* logic gates. In doing so, the synthesizer generates an optimized netlist that represents the circuit behavior using as few gates as possible. Commercial FPGA vendors provide proprietary logic synthesizers with their CAD suite.

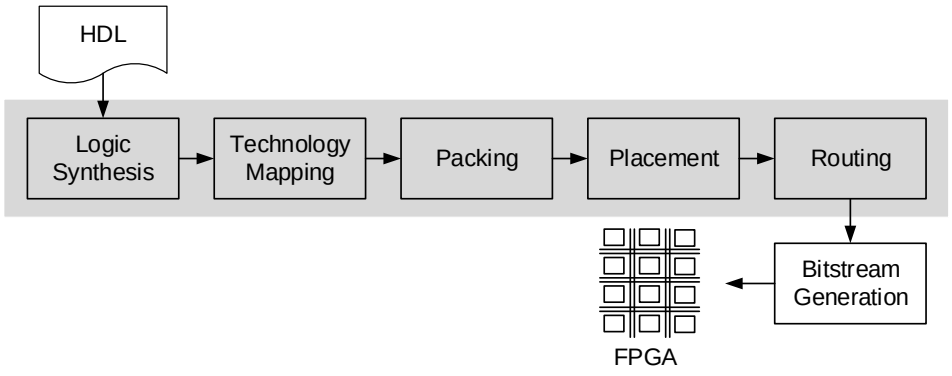


Figure 2.4: FPGA CAD Flow

For example, Xilinx provides *Xilinx Synthesis Technology (XST)* synthesizer [Synb] with their Vivado design suite [Sui] and Altera has *Integrated Synthesis* in their Quartus CAD suite [Altc]. Additionally, both companies provide interfaces with other synthesis tools from third parties such as Synplify [Syna] by Synopsys and Leonardo Spectrum [Gra] by Mentor Graphics. The most common synthesis tool in academic research is Odin-II [JKGS10].

The technology-independent gate netlist is then mapped to n-input LUTs available in the target architecture. This is known as *technology-dependent* mapping. The technology mapping tool decomposes the logic truth table into multiple n input logic tables. Commercial CAD suites have their own proprietary technology mapping softwares. In academic research, the ABC [SG] technology-mapping tool is most common.

In the *packing* stage, the group of connected LUTs are packed together into logic-clusters. In doing so, the packing algorithm attempts to fill the logic clusters to their capacity in order to reduce the number of physical resources used. Next, in the *placement* stage, the resources required by a circuit are assigned physical locations in an FPGA. The goal is to place the connected logic-blocks and/or logic-clusters close together so that the length of the required wiring can be reduced. Placement results in a detailed floor-plan of the circuit in which the packed netlist has been locked to the physical FPGA locations. In the final stage of *routing*, the physical locations are wired together using available routing channels in the device. A router determines which switch-blocks and connection-blocks should be turned-on for various input/output connections between logic-blocks.

After the complete circuit has been placed and routed, a bitstream file - that represents the functionality described in HDL - is generated and downloaded to the FPGA.

2.2 High-level Synthesis (HLS) based FPGA Designing

High-level Synthesis (HLS) raises the level of design expression from conventional HDL to a high-level software-like specification. The designer specifies the application behaviour using *un-timed* C/C++ constructs and the HLS tool generates a *cycle-accurate* hardware architecture from it. The generated architecture is described at the register-transfer-level (RTL) using a hardware description language (HDL) and contains a datapath and controller. The HLS generated RTL is then synthesized using conventional CAD suite, as described in Section 2.1.2, which then generates a bitstream

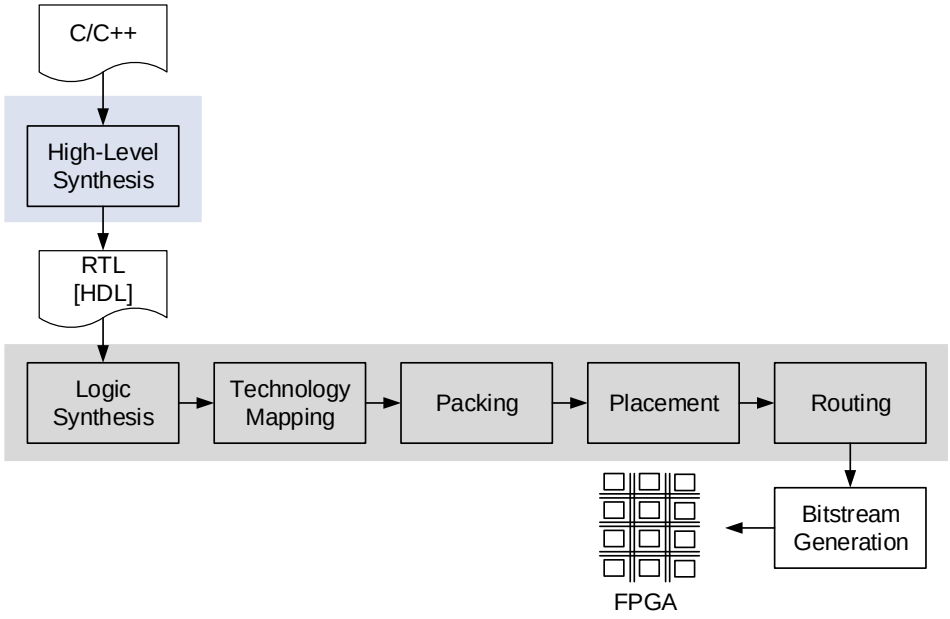


Figure 2.5: An Overview of High-Level Synthesis Flow Targeting an FPGA

for the target FPGA. An abstract HLS flow targeting a FPGA is shown in Figure 2.5. The work in this thesis is heavily based on HLS; therefore the key concepts in a HLS flow are briefly reviewed below.

2.2.1 HLS Design Steps: Key Concepts

High-level Synthesis (HLS) is divided into five basic steps [CGMT09], as illustrated in Figure 2.6. These steps are briefly discussed below:

1. *Compilation:* This step starts with the transformation of input specifications into a representation that exhibits the data and control dependencies among the operations. Usually a control data flow graph (CDFG) is used for such representation. A CDFG is a directed graph in which the nodes represent the basic-blocks and the edges represent the flow of operations.

2. *Allocation:* The allocation step determines the number and type of hardware components available in the target architecture that are required by the nodes in a CDFG. Hardware components include resources such as functional units to execute operations, connectivity re-

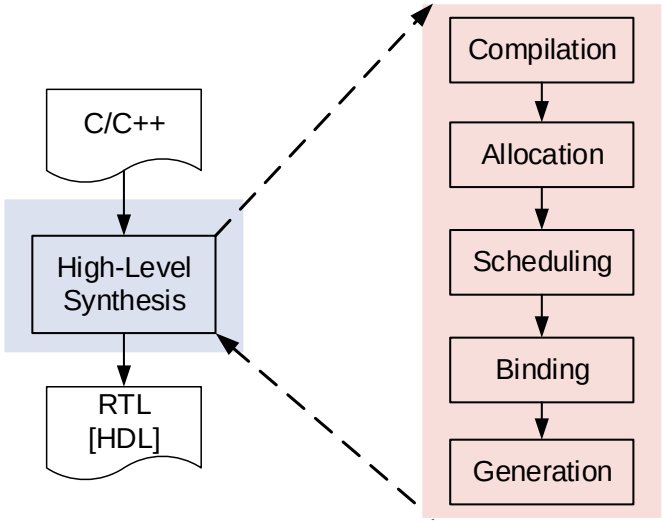


Figure 2.6: High-level Synthesis (HLS) Basic Steps

sources such as bus for linking various functional units and storage resources for saving results. Together these hardware components make the final datapath. The properties of these hardware components, such as number of pipeline stages in a functional unit and estimated unit delay/power, become the architecture constraints for scheduling and binding steps.

3. *Scheduling:* This step schedules the operations into clock cycles while satisfying all data and control dependencies. A single operation can be scheduled into a single or multiple clock cycles depending on the operation type. For instance, division is a multi-cycle operation.

   The output result from an operation can be directly fed as an input to another operation. This is known as *operation chaining* and is used wherever possible to save extra clock cycles required for reading the operands from their sources. The output of the scheduling step is a finite state machine (FSM) that controls the datapath.

4. *Binding:* The binding step binds the datapath schedule to hardware resources. Every operation is bound to one of the functional units and the inter-communication between various functional and storage units is bound to a bus. Multiple non-overlapping operations can be assigned to a same functional unit.

5. *Generation:* This step generates the cycle-accurate RTL architecture of the datapath that is fully compliant to the decisions made in the preceding steps of allocation, scheduling and binding. Besides datapath, the RTL of the controller FSM is also generated in this step. The RTL model is usually expressed in HDL which then is synthesized and mapped to a target FPGA by a conventional FPGA CAD suite, as discussed in Section 2.1.2 and shown in Figure 2.5.

2.2.2 LegUp: An Academic Open-Source HLS Framework

The work in this thesis is based on the *LegUp* High-Level Synthesis Framework, an open-source high-level synthesis suite [CCF+13]. LegUp automatically generates an SoC consisting of a MIPS processor and one or more accelerators from an application design expressed in *C*. The application is first profiled on a hardware profiler to identify the functions in a program that would benefit from hardware implementation. Based on this information, the user then annotates each identified function to be synthesized into the *hardware accelerator*. LegUp framework generates the datapath and a controller for the hardware accelerators. Each accelerated function in the original code is replaced with a call to its hardware accelerated version. The portions of the application that are not selected for acceleration are mapped to the MIPS processor and run as software. These accelerators and a MIPS processor are then combined using an Avalon fabric, creating an accelerated version of the original C code. Speedups of up to 12.9x are reported in [CBA13]. The user also has the option to exclude the processor from the final implementation; in that case the entire application is mapped to hardware.

The internals of LegUp tool flow are described below.

2.2.2.1 LegUp Tool Flow

The LegUp framework leverage the *LLVM* (low-level virtual machine) compiler infrastructure [LA04] as a front-end which is used for *parsing C* code and performing optimizations such as dead-code elimination and false data dependency removal, to name a few. The hardware synthesis steps, such as scheduling and binding, are performed in a back-end compiler pass within the LLVM infrastructure.

The input *C* program is transformed to an *intermediate representation (IR)* which is a machine-independent assembly language. A series of optimizations passes are performed on the IR before hardware synthesis com-

mences.

Once the IR has been optimized, the allocation step begins in which the amount of hardware required for implementation is assessed. The target FPGA device characteristics, specified in a TCL configuration file, are read at this stage for allocation.

In the next step, scheduling assigns each operation to a state in FSM which controls the datapath. LegUp uses SDC scheduling algorithm [CZ06]. The branches and jump instructions are used to determine the next state transition.

Next during binding step, the operators are bound to functional units and variables are assigned to registers. If more than one operator is assigned to a same functional unit, then the access is arbitrated using a multiplexer. The binding problem is solved using a weighted bipartite matching heuristic [HCLH90]. Finally, the Verilog RTL specifying the datapath architecture and its controlling state machine is generated, while satisfying all the control and data dependencies

Although HLS-generated designs for FPGAs represent only a subset of all FPGA design efforts; the size of this subset is growing. FPGA vendors have invested heavily in HLS-based methodologies. Commercial HLS tools such as Xilinx Vivado HLS [Xila], Altera OpenCL [Alta], eXCite from Y Explorations [Exp], Cadence Cynthesize [Cad] and Calypto Catapult [Cat] are gaining traction. They all are driven by the desire to improve designer productivity and open the door to many more designers than ever before.

2.3 Power Dissipation in CMOS Circuits

The innovation and enhancements in the fabrication technology has already enabled the integration of billions of transistors on a single die. As the transistors have shrunk in size, the amounts of current leaking through each transistor in an OFF state has become more pronounced: the reduced MOS gate oxide thickness allows the carriers *tunnel* through the oxide, causing *gate leakage* and the lower threshold voltage keeps the transistor *ON* even when it is supposed to be in an *OFF* state, causing *sub-threshold leakage*. Together, these leakage paths contribute significantly to the static power dissipation of a MOS gate. This leakage through each gate, when multiplied by billions of transistors on a chip, leads to a noticeable leakage power consumption. Moreover, the switching of these billions of transistors has significantly increased the operating power as well [WH].

In this section, we briefly review some basic definitions and discuss the

CMOS dynamic and static power consumption.

2.3.1 Basic Definitions

– *Instantaneous Power:* The instantaneous power, $P_{inst}(t)$, delivered by the power supply is the product of the voltage source, V_{dd}, and the supply current drawn by the circuit at time instant t. Mathematically, it is expressed as:

$$P_{inst}(t) = V_{dd}I(t) \tag{2.1}$$

As the voltage drop across a circuit element depends on the current passing through it, the instantaneous power consumed by the element is the product of instantaneous voltage across the element and the instantaneous current passing through it. Mathematically, it is expressed as:

$$P_{element}(t) = V_{element}(t)I_{element}(t) \tag{2.2}$$

– *Energy:* The total energy consumed during time interval T is the integral of instantaneous power. It is expressed as:

$$E = \int_0^T V_{dd}I(t)dt \tag{2.3}$$

– *Average Power:* The amount of energy per unit time is the average power. So over some interval T, it is expressed as:

$$P_{avg} = \frac{1}{T}\int_0^T V_{dd}I(t)dt \tag{2.4}$$

2.3.2 Dynamic Power

Dynamic Power is dissipated when the circuit is doing useful work. There are two main contributing factors to dynamic power: *Gate Switching* and *Short-circuit Current*, as discussed below:

1. **Gate Switching:**
   In order to understand the gate switching effect, let's consider a single CMOS inverter gate as shown in Figure 2.7.

   Following (2.3), the total energy drawn from the power supply can be given as:

$$\begin{aligned} E &= V_{dd} \int_0^T I(t)dt \\ &= CV_{dd}^2 \end{aligned} \tag{2.5}$$

   For input transition of $1 \rightarrow 0$, the PMOS transistor turns on which charges the load capacitance C_L. Out of the total CV_{dd}^2 energy drawn from the power supply, only half is stored in C_L while the rest is dissipated in the PMOS transistor as heat. For input transition of $0 \rightarrow 1$, NMOS transistor turns on which provides a discharging path to the capacitor. Therefore, the stored energy in the capacitor, $\frac{1}{2}CV_{dd}^2$, is dissipated as heat energy across NMOS transistor.

   As the load capacitance is switched between V_{dd} and GND with an activity factor α times the switching frequency f, the average switching power in a CMOS gate is given as:

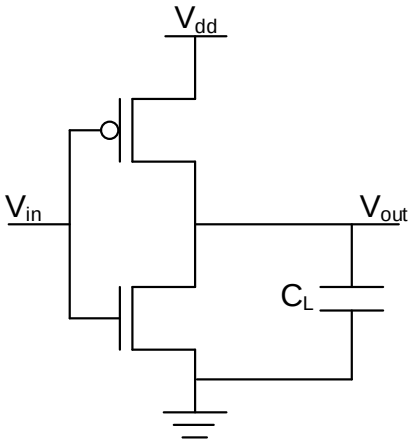


Figure 2.7: CMOS Logic Inverter

$$P_{switching} = C_L V_{dd}^2 \alpha f \tag{2.6}$$

As the digital circuits are built using CMOS gates which contain pull-up (PMOS) and pull-down (NMOS) transistor network, the basic inverter analysis can be extended to estimate the dynamic power in more complex gates.

2. **Short-Circuit Current:**
   During the input transition, there is a brief moment when both PMOS and NMOS transistors are partially turned on at the same time. This causes a direct path between V_{dd} and GND. As a result, *short-circuit* current flows which draws additional energy from the power supply.

2.3.3 Static Power

The static power is dissipated when the circuit is in a quiescent state i.e when there is no switching activity. A transistor leaks some amount of current even when it is in *OFF* state. There are three main types of leakage paths in a transistor, as shown in Figure 2.8. These leakage paths are discussed below:

1. **Subthreshold Leakage Current - I_{sub} :** Every MOS-transistor has a threshold voltage, V_t. In an *ON* state, current flows from source

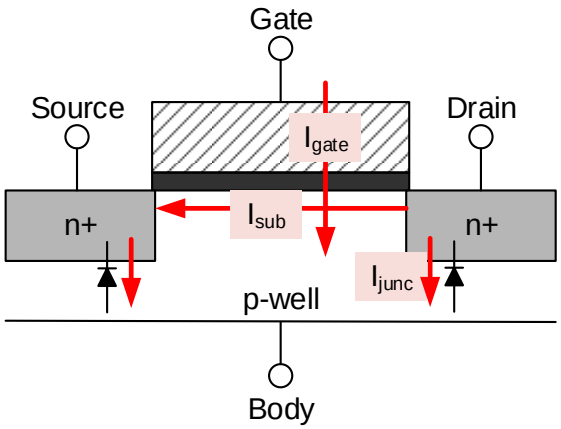


Figure 2.8: Illustration of Leakage Current Paths in an NMOS

to drain when gate to source voltage, V_{gs}, is greater than the threshold voltage. One would assume that this current would stop *ideally* when V_{gs} is less than V_t i.e when the transistor is supposed to be in *OFF* state. But in real transistor, there is always a leakage current from drain to source even when $V_{gs} < V_t$. This leakage is known as *subthreshold leakage* and has become more pronounced in $65nm$ and lower technology nodes.

Subthreshold leakage increases exponentially as threshold voltage is reduced. Further, it also increases exponentially with temperature [WH].

2. **Gate Leakage Current - I_{gate} :** In modern technology nodes, the gate oxide thickness has reduced to such an extent, typically thinner than 15-20A, that carriers *tunnel* through the oxide when the gate voltage is applied. This tunneling effect results in a leakage current from gate to body. Gate leakage has become comparable to subthreshold leakage in $45nm$ technology node and would be worrisome as CMOS is further scaled down [WH].

3. **Junction Leakage Current - I_{junc} :** The junction between diffusions (source/drain) and the substrate form a diode i.e $p-n$ junction. When the transistor is in the *OFF* state, these diodes become reverse-biased but still conduct a small amount of leakage current.

The subthreshold and gate leakage are the most dominant leakage current in modern process technologies due to ever decreasing threshold voltage and oxide thickness.

2.4 Power Dissipation in FPGAs

FPGAs can be programmed to become any arbitrary digital circuit at design-time. This architectural flexibility is mainly achieved by the programmable nature of the logic blocks and routing resources which add significant area overhead. [YR04] reports that more than 40% of a logic-block area is due to SRAM cells that are used to implement LUTs. Because FPGA use a large number of transistors per logic function to provide the programmability, they have a high power consumption. Compared to an ASIC, an FPGA implementation typically dissipates 14x more power [KR07].

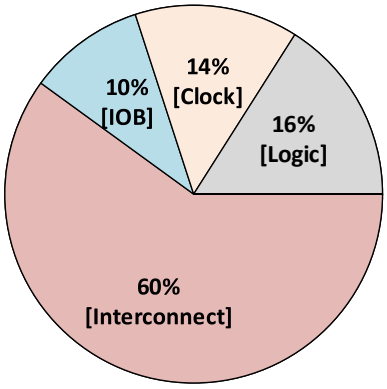


Figure 2.9: Breakdown of Dynamic Power Dissipation in a Xilinx Virtex-II FPGA [SKB02]

2.4.1 Dynamic Power Sources

Several studies have analyzed the dynamic power consumption in an FPGA[SKB02, LCHC03, TKR+06]. [SKB02] measured the dynamic power in a Xilinx Virtex-II FPGA. His results, shown in Figure 2.9, show that the routing resources account for 60% of the total dynamic power of the device. Similarly, [TKR+06] reports that interconnect resources consume 62% of the dynamic power in a Xilinx Spartan-3 FPGA.

Routing resources are most power inefficient because of their composition; the wires in a channel are pre-fabricated wires with configurable switches providing the connectivity between horizontal and vertical junctions. These wires and switches act as high capacitive load that needs to be charged and then discharged in each switching activity. [LLH+05] reports the various power components for global interconnects used in a *bigkey* circuit. His results, shown in Figure 2.10, show that dynamic power alone accounts for 57% of the total global interconnect power, of which the short-circuit power is dominant.

2.4.2 Static Power Sources

In modern process technologies, much of the power dissipation is due to static leakage power, which is dissipated when the chip is in a quiescent state. Leakage power has already become equal to that of dynamic power in 28nm commercial FPGAs [JHH]. [LCHC03] shows that, for cluster size of 12 and LUT size of 6, leakage power accounts for 52% of the total FPGA power.

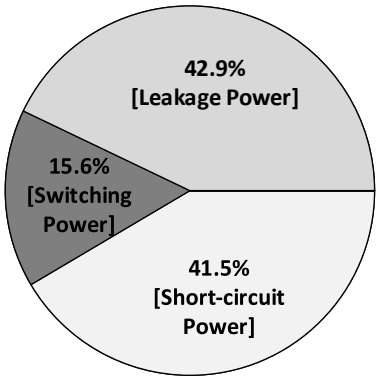


Figure 2.10: Breakdown of Global Interconnect Power in *bigkey* Circuit [LLH+05]

Moreover, on average, FPGAs dissipate 5.4x more static power as compared to their ASIC counterparts under worst-case operating conditions [KR06].

Tuan et al. [TKR+06] measures the leakage power in a Xilinx Spartan-3 device. Their results, given in Figure 2.11, show that 44% of the leakage power is dissipated in configuration memory cells. These cells are only used once per FPGA re-configuration cycle. Hence they are not performance critical and can be made using high V_t to reduce the static leakage power. For rest of the FPGA components, however, low V_t cells are used for performance

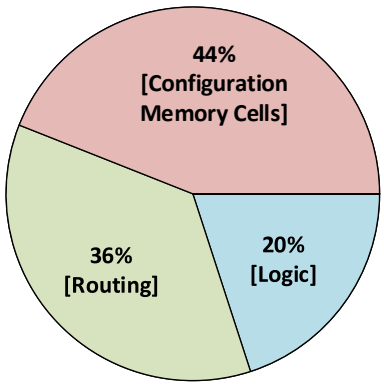


Figure 2.11: Break-down of Static Power Dissipation in Spartan-3 FPGA [TKR+06]

reasons which account for significant portion of leakage power.

Static power increases with the size of the device because each transistor has a leakage component which accumulates as the size is scaled up. Leakage is becoming more pronounced in larger FPGAs which contain in excess of one billion transistors. Additionally, a significant portion of these resources remain *underutilized* due to lack of applications that could completely fill an FPGA device. As a result, there are *unused* parts which always remain turned-on and consume significant leakage power. Commercial vendors are providing techniques to turn-off unused device primitives such as Block RAMs, DSP Blocks, Clock Managers and I/O pins at compile time but do not support power-gating of the unused logic-blocks. Moreover, the *used* logic resources in an application often operate for a brief burst of activity and remain *idle* otherwise during which leakage power is dissipated.

Going forward, while the refinements in the process technology will play a role in mitigating the static power, it is prudent to innovate in low-power FPGA architectures and design methodologies to maintain performance scaling.

2.4.3 Power Reduction Techniques in FPGAs

Power optimization in FPGAs requires techniques at various levels in a design process. These techniques range from enhancements in the process technology, innovative low-power techniques at the FPGA architectural level and software optimizations inside the CAD tool chain.

2.4.3.1 Dynamic Power Reduction

Voltage Scaling: The supply voltage V_{dd} is a quadratic term in switching power equation (2.6). Therefore, reducing V_{dd} is the most effective way to reduce dynamic power dissipation. For example, Xilinx 7-series FPGAs can operate on two core voltages: 1.0V and 0.9V. By operating a 7-series device core at 0.9V, up to 20% reduction in dynamic power is obtained [JHH].

[CTL+05] demonstrate the ability to dynamically scale the operating voltage in a Xilinx Virtex FPGA which leads to up to 54% of dynamic power reduction. Similarly, [NY13, NNY12] reports savings up to 85% by operating a Xilinx Virtex-5 FPGA at 0.58V as compared to operating at the nominal voltage of $1\,V$.

Reducing the voltage, however, increases the circuit delay. Therefore, the operating voltage should only be lowered to a point which maintain the reliable circuit operation. That is why, all dynamic/adaptive voltage scaling

techniques operate in a close-loop fashion with constant circuit performance monitoring [CTL+05, NY13, NNY12].

Clock Gating: In clock-gating, a clock signal is *ANDED* with an enable signal to disable the clock to the logic circuit. This prevents switching and thus reduces the activity factor in power equation (2.6). Significant amount of dynamic power is dissipated in FPGA clock trees. Therefore, commercial FPGAs by Xilinx and Altera provide mechanisms for static and dynamic clock gating at various levels of granularity in a clock spine [Kol, JHH, Altd]. Xilinx reports savings of 30-80% dynamic power in clock network depending on the enable rate [JHH].

The ability to perform clock-gating at run-time depends on the designer's ability to detect and exploit these clock-gating opportunities in a design. Recently, Xilinx has introduced an *intelligent clock-gating* support in its Vivado [Sui] tool-chain which automatically finds the fine-grained clock-gating opportunities in a design. It does that by finding the source registers that do not contribute to the output logic, in a place and routed design, and disables them during those idle clock cycles [JHH].

2.4.3.2 Static Power Reduction

Significant work has been done developing low-power FPGA architectures and CAD innovations [BW10, LLHC04, LL08, HA10]. Techniques to reduce static power have been proposed, including employing high-Vt transistors [alt08, Kle09]

Process Technology Enhancement: Significant leakage power can be controlled by fabricating better transistors; this is made possible through enhancements in the process technology [JHH, Kol, Altd]. For example, Xilinx reports a saving of 50% static power by just moving to next generation of an enhanced process [JHH].

Turning-off FPGA Primitives: Commercial vendors provide techniques to turn-off unused device primitives such as Block RAMs, DSP Blocks, Clock Managers and I/O pins at compile time. Block RAMs alone consume 30% of the total leakage power which is addressed by disabling power to the unused BRAMs in a design.

Dynamic Partial Reconfiguration of FPGA: Dynamic Partial Reconfiguration (DPR) is the ability to time-multiplex the FPGA resources at

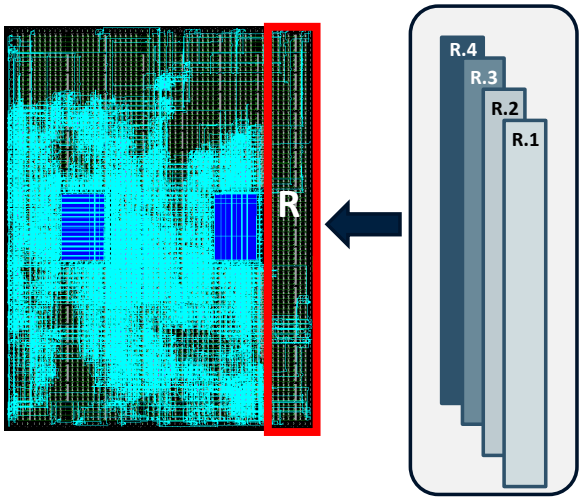


Figure 2.12: Illustration of Dynamic Partial Reconfiguration (DPR) in a FPGA

run-time. This is achieved by identifying the mutual exclusive parts in a design and loading them at run-time as needed. In doing so, the design is divided into two distinct regions, namely: *static* and *reconfigurable* region. The functionality in the static region does not change over time. The reconfigurable region acts as a placeholder which can be programmed with different functionality at run-time. In Figure 2.12, the region marked with a red boundary is designated as a reconfigurable region, R, with all possible design permutations for this region stored in an external memory. At run-time, design permutations stored in the memory are loaded into the reconfigurable region with the help of a controller residing in the static region.

DPR enables the use of a smaller FPGA device which directly lowers the static power consumption and reduces the per unit cost. DPR has the benefit of reducing static as well as dynamic power. The accelerators in an FPGA design may only operate for a brief burst of activities and remain idle otherwise. Whenever an accelerator is in idle state, its design partition can be reconfigured with a blank bitstream putting it in low dynamic power mode. Additionally, the ability to run parts of the design independently reduces the device density and requires a smaller device to implement a design which reduces the static power dissipation. Up to 80% of static power savings can be obtained by swapping several parts in a logic design [JHH].

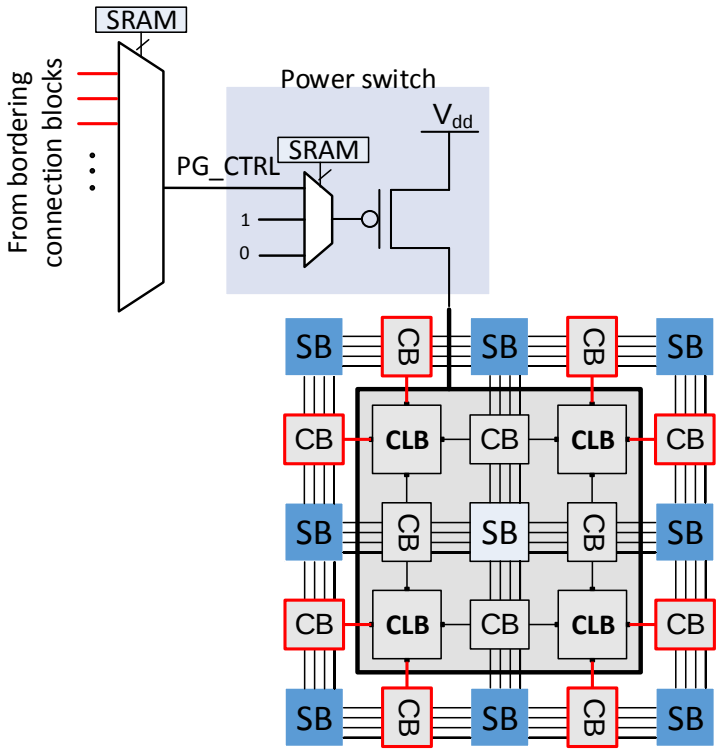


Figure 2.13: An Example Power-Gated Region

Dynamic Power-Gated (DPG) FPGA: One of the most effective techniques to reduce the static leakage power dissipation in an FPGA is to turn off (*power gate*) part of the design when it is idle. While power-gating is commonly used in ASIC designs, it has not find its way to commercial FPGAs as of yet. Part of the reason is the overhead of power-gating circuitry and the performance penalty. A recent academic work has presented low-overhead techniques for dynamic power gating (DPG) [BW10, BW12b, BW12a], in which parts of the FPGA can be turned off at run time, under control of a power controller. As we target this academic FPGA architecture to demonstrate the techniques presented in this thesis, we briefly review its architecture below.

DPG Architecture: The DPG architecture from [BW10] is a typical island-style FPGA in which the logic and routing fabric have been aug-

mented with header switches and associated control logic that allow regions of the chip to be selectively powered-down, under the control of signals from elsewhere on the chip (typically from a power-state controller). Each of these regions, called as power-gated regions (PGR), consists of a small number of CLBs, as shown in Figure 2.13 (the output pins of CLBs are not shown in the figure). In this figure, the power state for all grey-colored blocks can be controlled using the same power control signal that is routed from one of the bordering routing channels through one of the input pins to the CLBs. The power gating multiplexers selects which of the inputs to the CLBs will be used as the control signal. The connection block (CB) that is used to route the control signal is kept in an *always on* state. A Power-Gated Region (PGR) is the basic unit of power-gating which is turned on or off as a unit. The flip-flops within each logic element are not turned off as this allows for a more rapid power-up sequence since state is retained.

As described in [BW10], attention must be paid to the power state of the routing fabric. Although it would be possible to turn off many of the routing drivers adjacent to a logic block when the logic block is powered down, it is important to maintain power for the signal that will eventually turn the region back on. It is also important to maintain power for signals that pass by this logic block, connecting signals from un-related parts of the logic that have not been powered down. To accommodate this, the architecture contains extra control logic to selectively power down switch blocks that are *not* used for one any of these signals. If a switch block is used for a power gating signal or a signal that must retain power, the switch block is not powered down; these switch blocks are termed as *always-on* switch blocks. Switch blocks are large, therefore each switch block is divided into a number of *partitions* where each partition can be selectively powered down, reducing the power penalty of these always-on switch blocks. Since switch blocks constitute a significant part of an FPGA, the number of switch blocks that must be kept "always-on" has a significant impact on the power reduction possible from the device.

2.5 Queueing Theory

Queues are formed whenever there is a competition for limited resources. Often this happens in real-world systems when there is more demand for service than there is capacity for service. Due to this discrepancy, the customers for a service entering the system form *queues*. Queueing theory provides a set of tools for analyzing such systems to predict system performance under

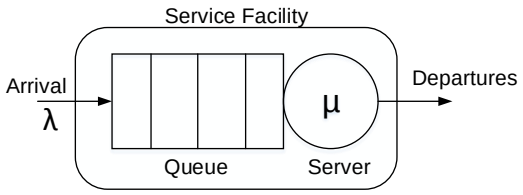


Figure 2.14: Single-Node Queuing System

varying circumstances.

2.5.1 Fundamental Concepts

In its simplest form, a queuing model consists of a single service facility containing both a queue and a server, as shown in Figure 2.14. Customers arriving at the input of the service facility can be served immediately or must wait in the queue until the server is free. After being served, customers leave the service facility. This type of arrangement is also known as a *single-node* queueing system. The term *customer* refers to the physical entity in the modeled system which arrives for service and leaves the system after being served. In our context of DPR system, the term customer refers to partial bit-stream data packets.

The queueing system, such as shown in Figure 2.14, is generally described by arrival rate of the customers at the queue input, the service rate at which the customers are serviced at the server and their scheduling discipline. The time between successive customer arrivals, also known as inter-arrival time, is stochastic in nature and therefore described by a probability distribution. The reciprocal of the inter-arrival time is the arrival rate, denoted by λ. Similarly, the service time, denoted by μ, is modeled by a distribution function. The scheduling discipline refers to the protocol with which customers in the queue are serviced. For example, some common use scheduling options include first-come-first-served (FCFS) and last-come-first-served (LCFS).

The above mentioned attributes describing the queueing system are expressed in a shorthand notation, also known as Kendall's notation, as: $A/B/m/S$, where A represents the inter-arrival times distribution, B indicates service time distribution function, m denotes the number of servers and S indicates the queue discipline.

2.5.1.1 System Performance Measures

Once a queueing system has been defined, it is analyzed mathematically to determine various performance measures. Common performance measures include, waiting time of the customer in a queue or in a system, system throughput, system utilization and queue length. These results are obtained when the system is in *steady-state* condition, which is when all the transients have passed and the performance values become independent of time. For example, if the number of customers in the system at time t is given as $N(t)$, then the probability of having n customers in the system is given as $p_n(t) = Pr\{N(t) = n\}$ while in steady-state the same measure is given as $p_n = Pr\{N = n\}$. Note that in steady-state there is no timing notion, which implies that the rate of customers entering the system must equal the rate at which customers leave the system. Such system is said to be in steady-state equilibrium.

The probability of the number of customers in the system, p_n, has to be calculated first in order to obtain various system performance values.

Calculating Probability of the Number of Customers p_n: A system can be represented by a set of states with state transitions for which states are defined by the number of customers in the system. The system can occupy only one state at any given time and will evolve by moving from state to state with some probability. The set of all states defines the *state space* of the system. If the future evolution of the system depends only on its current state and not on its past history, then the system is said to be *memoryless*. Such a system is said to have the *markov property* and can be represented by a *Markov Chain*. The state space corresponding to the single-node system in Figure 2.14 can be represented by the Markov Chain shown in Figure 2.15.

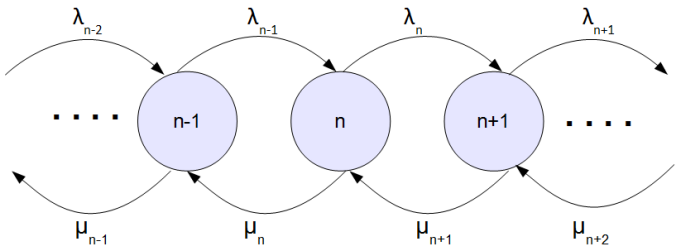


Figure 2.15: Birth-Death Markov Chain

The Markov Chain shown in Figure 2.15 is a *birth-death* process for which an arrival at a node increases the number in the queue by one and a departure decreases it by one. Once the state space model of a system has been specified, a set of *flow-balance equations* can be determined based on the balance of flow of incoming and outgoing customers for each state of the system. For example, the flow-balance equation corresponding to the state n can be written as:

$$(\lambda_n + \mu_n)\ p_n = \lambda_{n-1}\ p_{n-1} + \mu_{n+1} p_{n+1}, (n \geq 1) \tag{2.7}$$

where λ_n and μ_n are the arrival and departure rates in state n, respectively. Solving (2.7) for p_n yields (2.8):

$$p_n = p_0 \prod_{i=1}^{n} \frac{\lambda_{i-1}}{\mu_i} \tag{2.8}$$

The flow-balance equations can be used to solve for the probability of the system being in a state, p_n, which can then be used to calculate system measures such as the expected value of the mean number of customers in the system as:

$$L = E\left[N\right] = \sum_{n=0}^{\infty} n p_n, \tag{2.9}$$

2.5.2 Queueing Network

In order to capture the behavior of the real-world problems, queuing models can consist of multiple queueing nodes connected in a networked fashion, as show in Figure 2.16. This arrangement of nodes, where customers require service at more than one service station, is referred to as a *queuing network* or a *multiple-node* system.

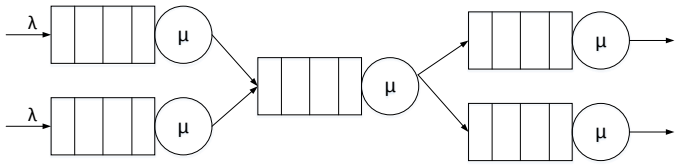


Figure 2.16: Example Queueing Network

A queueing network is called *open* when customers can arrive from outside and can similarly exit the network. Customers can arrive from outside the network to any node and depart from the network from any node. In contrast, a queuing network is said to be *closed* when customers can neither enter nor leave the network. The number of customers in a closed network is constant. There are many other variations of networks, the details of which are not discussed here.

An important measure of queueing networks is the joint-probability, $p_{n_1,n_2,\ldots,n_k}$, which is the probability that a network of k nodes is in the state, $\{n_1, n_2, \ldots, n_k\}$. This joint probability is used to calculate various performance measures of a queueing network. One approach of calculating the joint probability of a queueing network is to generate its underlying state space and solve the flow-balance equations, as is done in case of single-queue system discussed in Section 2.5.1.1. As the complexity of a network increases, the ease at which this probability can be solved grows correspondingly. Jackson [Jac57] solved such complex queueing networks and presented *product-form* solutions.

2.5.2.1 Jackson Network

Jackson showed that the joint-probability can be expressed as the product of its single-node station marginal-probabilities [Jac57] i.e the joint probability of the queue sizes in the network is a product of the probabilities of queue sizes in individual service centers. In doing so, the solution can be obtained without having the state-space of the queueing network in question. Therefore, the joint-probability is expressed as:

$$p_{n_1,n_2,\ldots,n_k} = \frac{1}{G}\rho_1^{n_1}\rho_2^{n_2}\cdots\rho_k^{n_k}, \tag{2.10}$$

where

$$G = \sum_{n_1+n_2+\cdots+n_k=N} \rho_1^{n_1}\rho_2^{n_2}\cdots\rho_k^{n_k}, \tag{2.11}$$

is a normalizing constant equal to 1 representing the sum of all possible joint-probabilities. From this, performance measures similar to Equation 2.9 can be used to calculate performance numbers for the entire queueing network system.

One limitation of Jackson networks is that all customers belong to the single class and therefore have the same routing, arrival-rate, and service-rate characteristics. This is often a limiting factor when modeling real-life

systems for which customers have different behaviors. For example, in our context of DPR systems, we would like to differentiate between PR and non-PR traffic in the system.

2.5.2.2 BCMP Network

BCMP networks are an extension of Jackson networks but with the additional feature that several classes of traffic can co-exist in the network each with their own statistical characteristics [BCMP75]. We use BCMP networks to correctly model the various phases and types of traffic in the DPR system.

For BCMP Networks, it is required that all network nodes belong to one of the following four types:

1. *Type-1*: *First come first serve (FCFS)* node. All classes must have the same service time.

2. *Type-2*: *Processor sharing (PS)* node. Classes can have different services times but order is not maintained. Instead, customers are processed in time slices weighted by their total service time.

3. *Type-3*: *Infinite Server (IS)* node. An infinite number of servers allows all customers to be serviced immediately.

4. *Type-4*: *Last come first serve (LCFS)* node with pre-emptive resume. Preempted customers are resumed without loss.

As will be discussed later in the chapter, this restriction is a concern only when we solve the network analytically.

Assuming there are N nodes and C classes in the queuing network, the joint-probability for the BCMP queueing network is given as,

$$p[S = (y_1, y_2, ...y_N)] = \frac{1}{G} d(S) \prod_{i=1}^{N} F_i(y_i) \tag{2.12}$$

where,

– $S = (y_1, y_2, ..., y_N)$ is the aggregated state of an N node network with y_i denoting the state of the ith node. $y_i = (y_{i,1}, y_{i,2}, ..., y_{i,C})$ is a vector of length C with $y_{i,j}$ representing the number of class-j customers at node i,

- G is a normalization constant that insures the sum of probabilities is equal to 1,
- $d(S)$ is a function of all queue arrival-times. All networks used in this chapter are closed chains and thus this function becomes 1.
- $F_i(y_i)$ is a function to each node type.

A more detailed description of Equation 2.12 can be found in [BCMP75]

2.5.3 Non-Product Form Queueing Networks

The product-form solution can only be achieved under specific assumptions which may make the modeled system over-simplified. Most practical problems may lead to queueing networks which may not result in product-form networks. Such non-product form networks cannot be solved analytically. Various approximation methods exist for such networks. Alternatively, discrete-event simulation can also be performed for quick analysis.

In order to capture some advanced aspects of the DPR system and model them correctly, we had to use few queueing primitives which resulted in a non-product form queueing network. We solve these queueing networks using simulations to obtain various performance measures.

2.6 Related Work

This section provides an overview of the related work in context of the contributions of this thesis. Section 2.6.1 review the work related to Chapter 3 and 4. Section 2.6.2 provide the related work in context of Chapter 6.

2.6.1 High-level Synthesis-based Design Methodology for Dynamic Power-Gated FPGAs

Dynamic power-gating of logic blocks in an FPGA is an emerging concept and still requires few years before we expect it to appear in the commercial FPGA devices. It is to be noted that dynamic power gating of logic-blocks is different from power-gating of variety of on-chip FPGA primitives such as BRAMs, PLLs, DCMs, I/O transceivers and I/O pins as supported by Xilinx and Altera. It is easier to turn-off such resources when they are known to be never used in a design. The design tools can detect such situation at compile time and power-gate the *un-used* resources. However, the active switching of logic blocks (containing the user circuit) requires knowledge of

the application schedule *a-priori* so that all the power-gating opportunities can be identified.

Power-gating has been extensively applied in ASIC and Microprocessor architectures and correspondingly there is a lot of work on techniques that find the *fine-grained* power-gating opportunities for such architectures [HBS+04, LBBS09, KKRS12, RRK09, ZKV+03, RPOG02, YCCY11, DKA+02, SSCS10, ADSN06, BJ12]. Similar techniques have been proposed for FPGAs at various levels of design abstraction, as reviewed below.

Bharadwaj et al. [BKBB05] use Directed Acyclic Graphs (DAGs) of an application to reorganize the operations into different states such that they are mutual exclusive and satisfy data dependencies. In doing so, a temporal partition is created and a state-machine cycles through sequence of states, turning on operations in one state and turning off the rest for every clock cycle.

Usami et al. [UO06] uses the enable signals in a gated clock design to partition the circuit into domains. These domains are then power-gated to save leakage power.

Ishihara et al. [IHK11] analyze the activity of a power-gated domain by comparing the phases of the input data with that of the output to switch between on and off state. Choi et al. [CXS04] utilizes the activation probability of a given circuit to determine power gating.

Hosseinabady et al. [HN14] demonstrate an approach for run-time power-gating in Xilinx hybrid ARM-FPGA device. The Xilinx ZYNQ device is powered-down to emulate power-gating. On wake-up, the full device is reconfigured before resuming functionality. However, finding the profitable power-gating opportunities is left to the user as a manual task.

All these techniques typically identify very *fine-grained* power-gating opportunities, as short as several cycles. Realistic DPG architectures, however, have significant overhead when parts of the chip are turned off and on; powering up internal capacitances requires significant power, and the need to throttle power gating circuitry to avoid in-rush current problems adds delay overhead [BW12b, BW14]. Thus, these architectures are not well-suited to take advantage of the fine-grained power-gating opportunities uncovered by these previous CAD techniques.

To take advantage of power-gating technique in FPGAs, therefore, it is necessary to uncover much more *coarse-grained* opportunities – regions of the circuit that can be powered down for many clock cycles. Intuitively, these regions do exist in a circuit, but they are difficult to detect in an automatic way. In most cases, the high-level behaviour of the circuit will depend heavily on the temporal behaviour of input signals, and this is not

encoded in the DFG, netlist, or any other form of circuit specification.

In this thesis work, we propose identifying power-gating opportunities at a higher level. In particular, we focus on SoC designs that are created using a High-Level Synthesis (HLS) methodology, and identify opportunities during HLS.

2.6.2 Model-based Performance Estimation of Dynamic Reconfigurable FPGAs

Various aspects of dynamic partial reconfigurable (DPR) systems have been researched in the past. In this section, we review the proposals focusing on towards the modeling and performance estimation strategies for DPR systems which is the focus of our work.

Papadimitrou et al. [PDH11] provide an extensive survey of recent works that measure reconfiguration time for real-world platforms. This survey compares the reconfiguration time of several platforms as a function of the type of bitstream storage (BRAM, SRAM, DDR, or DDR2), the internal configuration access port (ICAP) width and operating frequency, the bitstream size, and the type of controller, whether it be vendor provided or custom-built. The expected reconfiguration time is calculated based on the time spent in the different phases of the reconfiguration process. The limitations of this approach were explored by Gries et al. [GVPR04] and Galdino et al. [GPLR08] who showed that such predictions can be in error by one to two orders of magnitude.

Claus et al. [CZS+08] provide a set of equations to calculate the expected reconfiguration time and throughput. These equations are based on a value called the *busy-factor* of the ICAP, which is the percentage time that the ICAP is busy and not able to receive new bitstream data. Unfortunately, calculating the busy-factor requires that the system be built first. Further, this approach only works well for datapaths in which the ICAP is the bottleneck. An additional limitation of the approach is that it does not account for the effects of traffic not related to the partial reconfiguration (PR) process on PR datapath performance.

The reconfiguration cost model proposed by Papadimitriou et al. [PDH11] is the first cost model based on a theoretical analysis of the different phases of dynamic PR. The PR process is divided into phases and the total reconfiguration time is calculated by adding together the time spent in each phase. Because software time-stamps are used to make measurements, additional contributions to the measurement beyond that of the bus transfer are not included such as the time spent by the PR controller to initiate the transfer

and the impact of non-PR traffic. Because these additional contributions can be significant and are implementation-specific, phase measurements are only suitable for use with systems that are incrementally different from the measured system.

Hsiung et al. [HLL08] developed a SystemC model of a simple system consisting of a processor, bus, memory and reconfigurable region and used it to evaluate system performance and to identify bottlenecks. This work was restricted by the limited availability of SystemC functions to model the PR process and considered only an oversimplified form of the PR datapath.

Y.Qu et al. [QTSN07] propose a SystemC extension that works with the SystemC model of the existing DPR. The approach assumes the design starts with ANSI-C code. Based on the code, the time-resource estimates of application function blocks are generated. From this, the candidate blocks, also called as DRCF (dynamically reconfigurable fabric), that are suitable for partially reconfigurable hardware are identified. The design is then divided into static and dynamic region manually but the DRCF are identified automatically. The SystemC models are then used to estimate the reconfiguration time and the results simulates the reduction of configuration overhead.

In [GS03] the authors present *Algorithm Architecture Adequation (AAA)* methodology for rapid prototyping of embedded systems. The framework is not geared towards the DPR systems specifically but is extended by others for such use. AAA methodology aims at finding the best matching between an algorithm and an architecture through adequation step. The architecture is based on graph theory where the graph vertices denotes the architecture resources. For example, BUS/MUX/DEMUX vertices represent the communication infrastructure in the graph. The application is modeled using a hyper-graph which is partially ordered by data-dependences. The application is mapped to the architecture and performance is predicted for two metrics: execution duration of the task and memory space required to store and execute the whole application. The framework concepts are implemented in SynDEx tool-chain [GLS99].

Berthelot et al. [BN06] extends the above mentioned AAA methodology and SynDEx tool-chain for DPR system design. They manually specify the reconfigurable tasks in the application graph. On the architecture side, the reconfigurable parts are considered as vertices in the architecture graph. An internal media allows data exchanges between the parts. The scheduling is based on prefetching cost of the configurations. In the tool-chain the macro-codes are generated for run-time reconfigurable components. The configuration manager is then automatically generated by interpreting the

operation sequences expressed in the macro-code.

Boden et al. [BFR+08] presents a framework which aims at reducing the reconfiguration time by extracting the temporal reusable modules (TRMs). These modules are then reused by several sequentially executed tasks.

Conger et al. [CGRG08] presents DPR floorplanning framework. Their framework provides support in the floorplaning of the system with respect to some PR modules constraints. Further the design space exploration results are limited to floorplaning.

Duhem et al. [DML11] presents a SystemC methodology to evaluate the scheduling strategies for hardware task management in a dynamically reconfigurable system. This simulation is performed after the HDL synthesis step to incorporate the resource usage information. The design is explored with different configurations of the Reconfigurable Zones (RZ). The aim is to provide the designer with a suitable combination of RZs that will meet the application timing constraints once implemented on FPGA. The design space is explored based on various Reconfigurable Zones by feeding the simulator with a pre-defined pool of RZs.

We believe that the most important disadvantage of these approaches is the fact that these systems must be built first before they could be measured for performance estimates. This is counter to the motivation of our work which is to predict performance trends early in the design cycle and without the need to implement the system. Further, we model the impact of non-PR traffic on various performance estimates which is not considered by any of the above mentioned work.

2.7 Summary

High-level synthesis (HLS) methodologies are increasingly being used to raise the abstraction level of a design and improve designer productivity. In such a methodology, a designer specifies a design using a language such as C, and the tool generates a circuit. Importantly, the overall architecture of the resulting digital system is determined by the high-level synthesis compiler.

As FPGAs are migrated to advanced processing nodes, static (leakage) power has become a first-class concern for many applications. In modern process technologies, much of the power dissipation is due to static leakage power, which is dissipated when the chip is powered-on, regardless of the activity of the device. Techniques to reduce static power have been proposed, including employing high-Vt transistors [alt08, Kle09], dynamic partial reconfiguration and power-gating parts of the chip. One of the most

effective techniques to reduce the power dissipation in an FPGA is to turn off (power gate) part of the design when it is idle. A recent work has presented techniques for dynamic power gating (DPG) [BW10], in which parts of the FPGA can be turned off at run time, under control of an on-state power controller. If used correctly, DPG has the potential to significantly reduce the leakage power of many designs.

Queueing Theory is a promising tool for the performance evaluation of dynamically reconfigurable systems. It helps system designers to make informed decisions early in the design process thus avoiding the time and costs associated with building candidate DPR systems.

Chapter 3

High-Level Synthesis-based Design Methodology for Dynamic Power-Gated FPGAs

3.1 Context

There is a growing trend towards implementing multiple custom accelerators on an FPGA to speed-up the critical portions of a design. Depending on the application, these accelerators may only operate for a brief burst of activity and remain idle otherwise during which significant static leakage power is dissipated. Static power dissipation has become a first-class concern for many applications. One of the most effective techniques to reduce the static dissipation in an FPGA is to turn off (power-gate) part of the design when it is idle. However, given an application it is quite challenging to identify *when* and *how long* a certain accelerator in the design is going to remain idle and whether it is profitable to power-gate it during that period of time – Power gating feature incurs an extra energy overhead in switching between *sleep* and *active* modes and the accelerator should only be power-gated if it will be idle long enough to compensate for this energy penalty. The effectiveness of dynamic power gating, therefore, depends on the designer's ability to detect and exploit power gating opportunities in the design.

Automatically identifying power gating opportunities as part of a FPGA CAD tool is difficult. Although it is possible to examine the dataflow graph or netlist of an application to try to find parts of the chip that are not needed for periods of time, these techniques typically identify very *fine-grained* power gating opportunities, as short as several cycles [BKBB05, UO06, IHK11, CXS04]. Realistic DPG architectures, however, have significant overhead when parts of the chip are turned off and on; powering up

internal capacitances requires significant power, and the need to throttle power gating circuitry to avoid in-rush current problem adds delay overhead [BW12b, BW14]. Thus, these architectures are not well-suited to take advantage of the fine-grained power gating opportunities uncovered by these previous CAD techniques.

To take advantage of power gating technique in FPGAs, therefore, it is necessary to uncover much more *coarse-grained* opportunities – regions of the circuit that can be powered down for a significant number of clock cycles. Intuitively, these regions do exist in a circuit, but they are difficult to detect in an automatic way. In most cases, the high-level behaviour of the circuit will depend heavily on the temporal behaviour of input signals, and this is not encoded in the data-flow-graph (DFG), netlist, or any other form of circuit specification.

Identifying power gating opportunities can be much easier, however, if the design is created at a higher level. In particular, high-level synthesis (HLS) methodologies are increasingly being used to raise the abstraction level of a design from RTL to behavioural level. In addition to the benefits of improved designer productivity, faster time to market and increased automation, expressing the design at a higher level provides significantly larger power reduction opportunities as shown in several studies [Rab09, RJD98, CB95]. Figure 3.1 depicts the trends in uncovering the power reduction opportunities available at various levels of design abstrac-

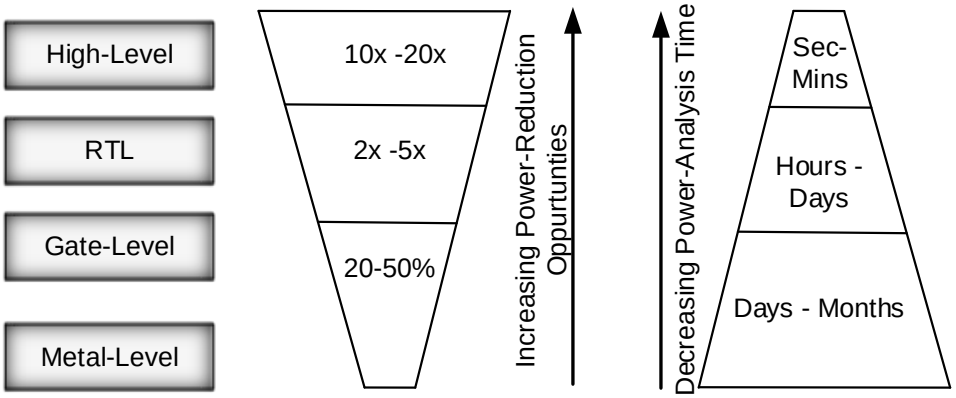


Figure 3.1: Power Reduction Opportunities and Power Analysis Time at Various Levels of Design Abstraction from [RJD98]

tion and the time it takes to analyze them. The highest-level of design abstraction, potentially the HLS level, offers the most power saving opportunities that can be analyzed within minutes [RJD98].

This chapter describes:

1. An HLS based design methodology for dynamic power-gated FPGAs that,
   - automatically identifies profitable power gating opportunities,
   - automatically extracts a power-state controller to dynamically marshal the identified power gating opportunities,
   - maps the application to the power-gated FPGA architecture.

   A computer-aided design (CAD) framework that demonstrates the proposed methodology is discussed in Section 3.2 and has been published in [ABW+14].

2. The quantification of the possible energy savings achieved through power gating in real-world applications. Section 3.3 discusses the experimental results which are published in [ABW+14] .

3.2 Proposed Methodology

We focus on system-on-chip (SoC) designs that are created using a high-level synthesis (HLS) methodology, and identify opportunities during HLS. In such a methodology, a designer specifies a design using a high-level language such as C, and the tool generates a circuit. Importantly, the overall architecture of the resulting digital system is determined by the high-level synthesis compiler. The compiler not only knows about the structure of the circuit, but also its temporal behaviour which can be used to identify power gating opportunities automatically.

A typical system-on-chip (SoC) design created using an HLS tool, such as [CCF+13], contains a processor with one or more hardware accelerators used to speed up critical portions of the algorithm. An accelerator may have several phases, each of which is accelerated by a separate logic circuit, which we call as a sub-accelerator. In such a multi-accelerator SoC, there are two types of power gating opportunities:

1. **Accelerator-level Opportunities:** Accelerator-level opportunities occur when an entire accelerator (including all the sub-accelerators of

this accelerator) is predicted to be idle for a period of time. An entire accelerator is turned off or on as a unit.

2. **Intra-accelerator Opportunities:** Intra-accelerator opportunities occur when parts of an accelerator are predicted to be idle for a period of time.

In this chapter, we consider power gating at the *accelerator*-level. Accelerators are well-suited as a unit of granularity for power gating as they are typically large enough that the benefit of power gating outweigh the overhead in architectures such as the one presented in [BW10]. The identification and evaluation of intra-accelerator opportunities are discussed in Chapter 4.

In the proposed methodology, the schedule generated from a high-level synthesis tool is used to determine the idle periods of individual hardware accelerators in a synthesized system. The predicted length of these idle periods is used to determine whether the power saved by power gating the accelerator is more than the overhead of turning the accelerator off and then on again at the end of the idle period. Based on this knowledge, individual accelerators can then be power-gated when it is deemed profitable. A CAD framework, presented in Section 3.2.2, implements the proposed methodology.

3.2.1 Context

Although our work will apply to any FPGA which provides dynamic power gating control, and any high-level synthesis tool, we describe it in the context of the Dynamic Power-Gated FPGA architecture from [BW10] and the LegUp high-level synthesis tool from [CCF+13], which were discussed in Section 2.4.3.2 and Section 2.2 respectively.

3.2.2 Design Tool Framework

The design tool framework, depicted in Figure 3.2, is composed of the following three phases:

1. HLS Guided Phase
2. Pruning Phase
3. Mapping Phase

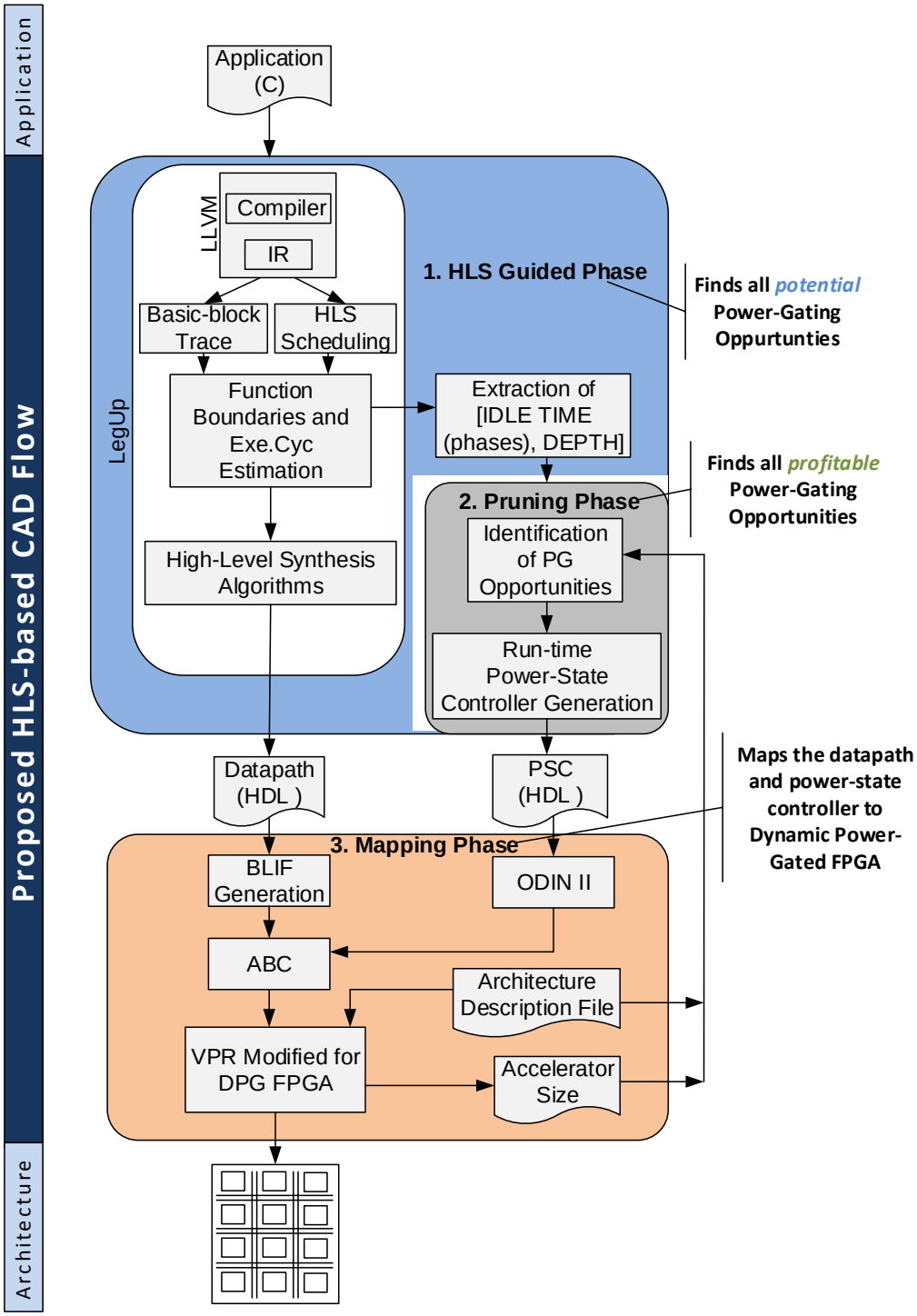


Figure 3.2: Design Tool Framework

The flow starts from finding all the potential power gating opportunities in an application in the first phase. It then selects the profitable opportunities in the second stage and maps the generated datapath and controller to the target dynamic power-gated FPGA in the third phase. The three phases of the proposed design approach are described below along with a description of how the techniques were instantiated in the *LegUp* framework for the architecture from [BW10].

3.2.2.1 HLS Guided Phase

Each accelerator typically has *active* and *idle* periods. The active periods represent the execution periods of an accelerator and the idle periods are potential power gating opportunities. The goal of this phase is to identify all the idle periods across all the accelerators in a design. The overall flow of the HLS guided phase, shown in Figure 3.2, is as follows.

The input *C* program is transformed to an *intermediate representation (IR)* (which is a machine-independent assembly language) by the LLVM front end *clang*. Each function in the program is decomposed into basic blocks by the compiler at this stage. A basic block is the portion of the program with a single entry and a single exit point. It is to be noted that LegUp converts functions in a program to hardware accelerators. A series of optimization passes are performed on the IR before hardware synthesis commences. Once the IR has been optimized, the allocation step begins in which the amount of hardware required for implementation is assessed. The target FPGA device characteristics, specified in a TCL configuration file, are read at this stage for allocation. The next step is scheduling, which assigns each LLVM instruction inside a basic block to a hardware clock cycle and generates a schedule for the entire program. In order to estimate the length of each execution period, we keep track of the basic blocks within a function and record all the function calls as the program runs inside HLS framework. We then combine this program trace information with the scheduling (which contains the number of clock cycles for each basic block) to estimate the length of each execution period in an accelerator. We also record the start and end event of each invocation in each accelerator, and use this information to calculate the length of the idle periods in each accelerator. This way we are able to profile the C program within the HLS framework and identify all the idle periods along with their duration.

The idle periods length is then passed to the pruning phase which determines the profitable idle periods for which power gating will potentially save leakage power.

3.2.2.2 Pruning Phase

The previous phase produces a list of all potential power gating opportunities. However, many of these may not be profitable. Powering down (and later powering up) a power-gated region incurs both energy and delay overhead. When powering up a region, each node in the circuit must be charged through the finite header transistors and power distribution framework of the chip, and this consumes energy. Further, the speed at which internal nodes are re-powered must be controlled to ensure that the power droop on the voltage lines is less than an allowable threshold [BW12b]. Thus, for each power gating opportunity, we must determine whether a power gating event should be generated. This phase makes this determination and then generates a power-state controller that sequence these power gating events at run-time.

In general, an accelerator should be turned off when idle if the power saved by power gating the accelerator is more than the overhead of turning the accelerator off and then on again at the end of the idle period. To make this determination, we evaluate both the size of the accelerator and the idle time which together indicate how much leakage might be saved if this region is turned off. An accelerator is turned off in its idle period if energy with power gating, E_{PG}, is less than the energy with no power gating, E_{NPG}, during that period:

$$E_{PG} < E_{NPG} \tag{3.1}$$

In Equation (3.1), the total energy consumed by an accelerator by power gating it during its idle period, E_{PG}, is given by Equation (3.2),

$$E_{PG} = E_{turn-off} + E_{sleep} + E_{turn-on} \tag{3.2}$$

where $E_{turn-off}$ and $E_{turn-on}$ is the energy required to turn-off and later turn on an accelerator respectively; for simplicity, we combine these energies into switching energy E_{switch}. E_{sleep} represents the static leakage energy of the accelerator when power-gated during its idle time. These energies can be calculated as follows:

In Equation (3.2), E_{switch} is calculated using Equation (3.3),

$$\begin{aligned} E_{switch} =& NumPGR \times (P_{switchPGR} \times T_{switchPGR}) + \\ & NumSBs \times (P_{switchSB} \times T_{switchSB}) \end{aligned} \tag{3.3}$$

where in Equation (3.3), $NumPGR$ and $NumSBs$ is the number of PGRs and SBs, respectively, occupied by the accelerator. We find this by perform-

ing an initial mapping of the accelerator to the fabric, however, estimation techniques could also be used. $P_{switchPGR}$ and $T_{switchPGR}$ is the power and time required for a PGR to enter and exit power gating mode. Likewise, $P_{switchSB}$ and $T_{switchSB}$ is the power and time required for an SB to enter and exit power gating. The switching parameters are fixed architecture parameters (defined in an architecture description file).
In Equation (3.2), E_{sleep} is calculated by Equation (3.4),

$$\begin{aligned} E_{sleep} =& NumPGR \times P_{PGR-leakage-OFF} \times (T_{IDLE} \times DF) + \\ & NumSBs \times P_{SB-leakage-OFF} \times (T_{IDLE} \times DF) \end{aligned} \tag{3.4}$$

In Equation (3.4), $P_{PGR-leakage-OFF}$ and $P_{SB-leakage-OFF}$ is the static leakage power of a power-gated region and switch block respectively in the off state. Both are fixed architecture parameters. T_{IDLE} is the idle time of an accelerator which is extracted from the schedule, as discussed in Sec. 3.2.2.1. The addition of the power gating support in an architecture incurs an area and delay overhead which degrades the performance. In order to account for this performance degradation, the execution and idle time period should be increased by a *Degradation Factor (DF)* when calculating the accelerator energy in a power gating mode. The value of the degradation factor depends on the power-gated architecture design.

The energy in non power gating mode but on a power gating architecture, E_{NPG}, is given in Equation (3.5),

$$\begin{aligned} E_{NPG} =& NumPGR \times P_{PGR-leakage-ON} \times (T_{IDLE} \times DF) + \\ & NumSBs \times P_{SB-leakage-ON} \times (T_{IDLE} \times DF) \end{aligned} \tag{3.5}$$

where $P_{PGR-leakage-ON}$ and $P_{SB-leakage-ON}$ is the ON leakage power dissipated by the PGR and SB respectively and is defined in the architecture description file. Note that there is no performance degradation factor used in calculating E_{NPG} if the accelerator were to execute on an architecture without power gating support.

The decision whether or not to power gate in a particular idle phase is then made using Equation (3.1). Once the profitable power gating opportunities have been identified, a power-state controller (PSC) which controls the sleep signal of the accelerator at run-time is extracted from the accelerator's schedule. The output of the pruning phase is the RTL of the PSC.

3.2.2.3 Mapping Phase

In the mapping phase, the entire system – the datapath and power-state controller (PSC) – is mapped to the target dynamic power-gated FPGA architecture. The detailed steps inside the mapping phase are depicted in Figure 3.3. The mapping phase is based on several open-source academic tools. Quartus II [Altb] is used to generate *BLIF* files from the Verilog HDL produced by LegUp [CCF+13], *ODIN-II* [JKGS10] for elaboration of power state controller, *ABC* for logic synthesis and a modified version (5.0) of the VPR FPGA tool [BRM99] to target the dynamic power-gated FPGA architecture.

The datapath, containing the accelerators and the datapath controller, is

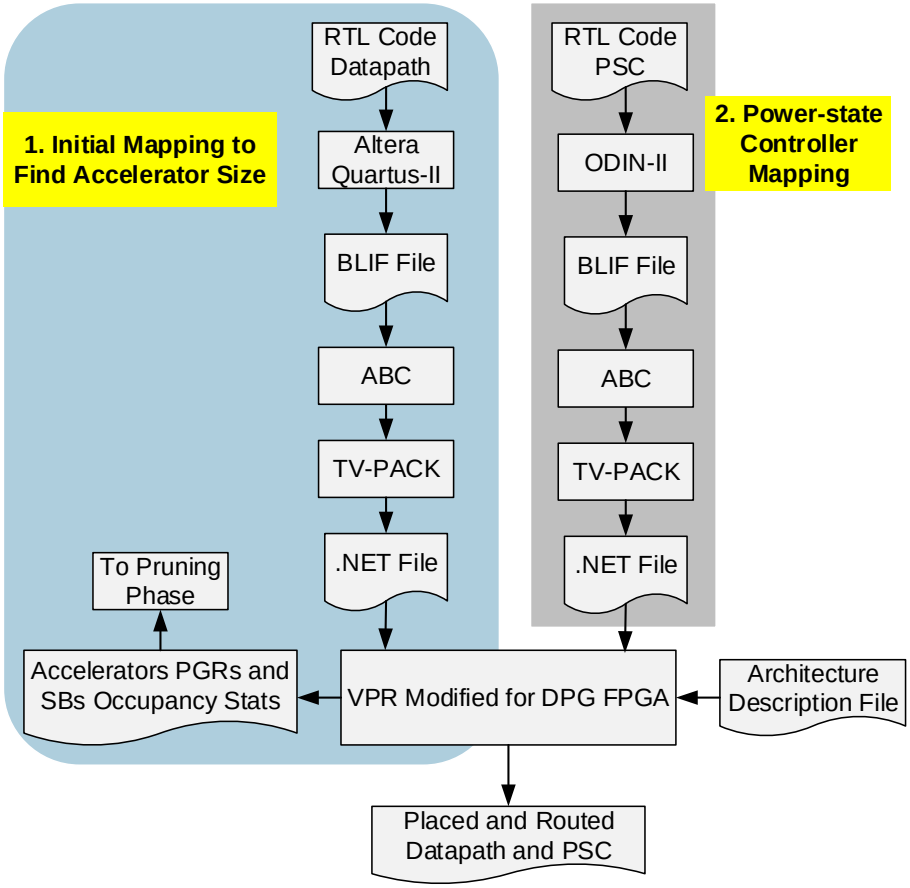


Figure 3.3: Detailed Steps inside Mapping Phase

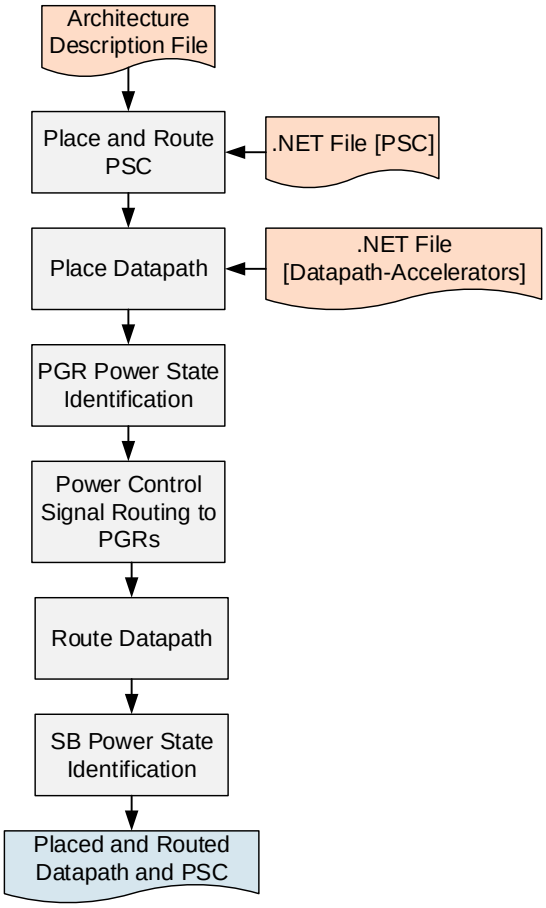


Figure 3.4: Detailed Steps inside VPR Modified for Dynamic Power-Gated FPGA from [Bso]

mapped first to find the size of all the accelerators in a design. As discussed earlier, this information is then fed into the pruning phase which determines the profitable power gating opportunities as discussed in Section 3.2.2.2. Once all the profitable power gating events have been identified, the RTL of the power-state controller is generated which is then mapped in the second round. The control signals coming out from the power-state controller are connected to the respective accelerators inside modified VPR flow as discussed below.

Figure 3.4 shows the implementation steps taken inside the modified version of VPR from [Bso]. In Step 1, the power state controller (PSC) is

placed and its internal connections are routed on the FPGA resources. The FPGA resources that are used by the PSC are locked, and their power state is set to *always-on*. In Step 2, placement of the accelerator is performed. In Step 3, each PGR occupied by the accelerator is examined to determine its power state. Based on the parts of the accelerator mapped to the logic cells in a PGR, one of the following power states is assigned to each PGR:

- **Dynamically-controlled PGR:** if only one power-gated module is mapped to the PGR's logic-clusters.
- **Always-off PGR:** if all of the PGR's logic-clusters are empty.
- **Always-on PGR:** all other cases.

In Step 4, the nets of the power control signals are routed. Each switch-block (SB) that is used to route these signals is set as *always-on* to ensure that the power control signals are available all the time. In Step 5, the connections in the circuit netlist are routed on the available FPGA resources. Although the power control signals are routed before the circuit's nets, this has negligible effect on routability and performance of the circuit because only a small fraction of the routing resources are used to route the control signals. Finally, in Step 6, the power state for each of the switch blocks is determined as follows:

- **Dynamically-controlled SB:** if the PGR is dynamically-controlled and only one power-gated module is routed through the SB.
- **Always-off SB:** if the SB is not used to route signals.
- **Always-on SB:** all other cases.

Ideally, we would like to be able to dynamically turn off all SB partitions occupied by the power-gated modules. However, due to the complexity of the routing topology of multiple accelerator circuits, some SBs are required to be always-on, as explained in [BW12a].

Figure 3.5 shows the floorplan of an advanced encryption standard (AES) benchmark obtained by applying the mapping steps described above. AES has three accelerators and a datapath controller. The logic blocks belonging to an accelerator are positioned close to each other by the VPR placement algorithm [Bso]. The dashed box represents the always-on power-gated region. The solid line box represents the power-gated region that can be potentially turned-off at run-time to save leakage power.

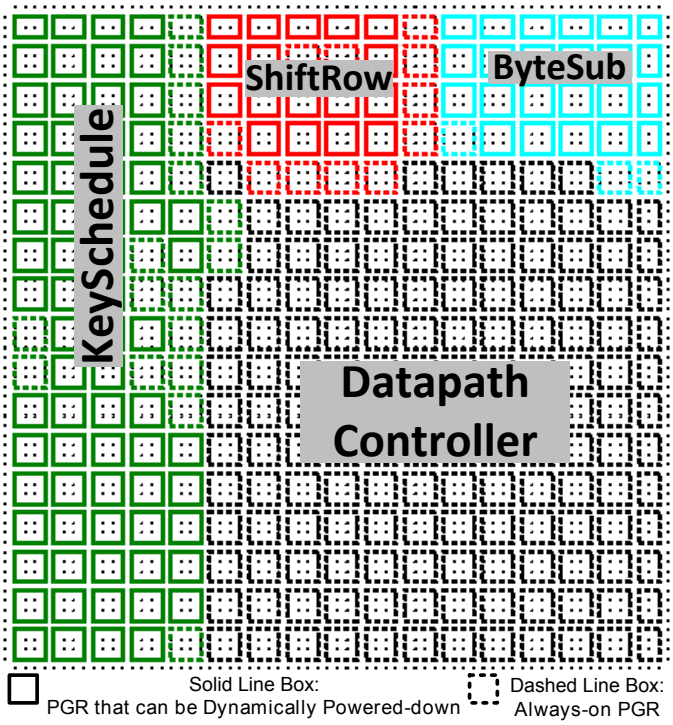


Figure 3.5: Floorplan of AES Benchmark using VPR in the Mapping-Phase 3.2.2.3: Showing the PGR occupancy of three accelerators and the datapath controller

Since we have not fabricated an FPGA with the power gating support, we do not generate a bitstream. However, the mapping described in this section allows us to estimate the power consumption of the resulting design in order to evaluate the effectiveness of our techniques, as will be described in the experimental results Section 3.3.

The first two phases of the proposed design framework, which identifies the profitable power gating opportunities at HLS level, can be seamlessly integrated with any other FPGA CAD tool-chain to target a real FPGA device.

3.3 Experimental Results

3.3.1 Experimental Setup

To quantify the impact of our proposal, we use the CHStone benchmarks suite [HTH+08]. The CHStone benchmarks suite is developed for C-based high-level synthesis (HLS) and represents diverse real-world applications from media, encryption and arithmetic domains. The C-based source of each benchmark is provided as input to the LegUp HLS framework which performs the standard compiler optimization passes on the code before HLS commences. In our experiments, we turn on all optimizations which passes the -O3 flag to clang to enable all LLVM optimizations. We also disable function inlining to maximize functions which could be implemented as hardware accelerators.

LegUp synthesizes the functions in the optimized version of the code to hardware accelerators. Using our CAD framework, we model the impact of power gating of these hardware accelerators and quantify the energy savings. Five of the twelve CHStone benchmarks have sufficient idle time in their schedules to warrant power gating; we present results only for these five.

Table 3.1 shows the architecture specifications of the target dynamic power-gated FPGA from [BW10] and Table 3.2 shows its HSPICE simulated parameters used in the experiments. These architecture and simulation parameters are stored in an architecture description file.

Table 3.1: Architectural Parameters of the Target Dynamic Power-Gated FPGA from [BW10]

Architecture Parameters		
PGR_W	Power-gated Region Width	4
PGR_H	Power-gated Region Height	4
W	Channel Width	120
N	Cluster Size	6

3.3.2 Power-Gating Potential and Savings

Both the size of the accelerator and its idle period duration determine how much leakage power could be saved if the accelerator is power-gated during idle period. To estimate these savings, we first find the number

Table 3.2: HSPICE Simulated Parameters of the Target Dynamic Power-Gated FPGA from [BW10]. [PGR = Power-Gated Region, SB = Switch-Block]

	Simulated Parameters	**PGR**	**SB**
$P_{1\rightarrow0}$	Power to Turn-Off [Watt]	1.22E-04	1.45E-05
$P_{0\rightarrow1}$	Power to Turn-On [Watt]	5.98E-04	5.53E-05
$T_{1\rightarrow0}$	Time to Turn-Off [sec]	6.59E-09	4.12E-10
$T_{0\rightarrow1}$	Time to Turn-On [sec]	7.14E-09	4.46E-10
$P_{ON-leak}$	On Leakage Power [Watt]	6.70E-05	5.36E-06
$P_{OFF-leak}$	Off Leakage Power [Watt]	2.03E-06	2.93E-07

of dynamically controlled (DC) power-gated regions (PGRs) and switch-blocks (SBs) for each accelerator in a benchmark by performing the mapping as discussed in Section 3.2.2.3. This accelerator size information is then combined with its schedule to find the profitable power gating opportunities using the criteria and equations discussed in Section 3.2.2.2.

In order to quantify the impact of power gating, we estimate the following three energies for each accelerator in the benchmark. The nomenclature used to differentiate energies in various modes and architectures is *Energy_Mode_Architecture*, where mode can be power-gated (PG) or non-power-gated (NPG) mode. Similarly, architecture can be a dynamic-power-gated (DPG) or a non-power-gated (NPG) architecture.

E_PG_DPG: This represents the total leakage energy consumed by an accelerator in power gating mode, in which the accelerator is turned off during its idle period if it is deemed profitable, on an architecture supporting dynamic power gating. The ON-leak energy dissipated in all execution periods and OFF-leak energy in all idle periods are added together to estimate total *E_PG_DPG*, as shown in Figure 3.6 and given in Equation (3.6).

$$E\_PG\_DPG = \sum_{Exe\_Period}^{n} \sum_{i=1} (Exe\_ON\_Leak\_DPG)_i + \sum_{Idle\_Period}^{n} \sum_{i=1} (Idle\_OFF\_Leak\_DPG)_i \tag{3.6}$$

The DPG architecture suffers a 10% performance degradation due to the presence of power gating circuitry; to account for this, the clock period of the operating clock is increased by this amount when calculating *E_PG_DPG*.

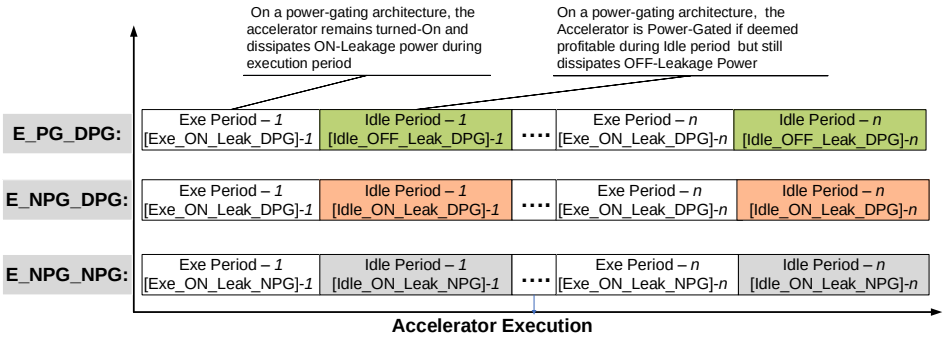


Figure 3.6: Illustration of Various Energies Calculated for Comparison. [PG = Power-Gated, NPG = Non-Power-Gated, DPG = Dynamic Power-Gated]

E_NPG_DPG: This represents the total leakage energy consumed by an accelerator if it were to run in a non power gating mode; keeping the accelerator powered-on during idle period, on an architecture supporting dynamic power gating. E-NPG-DPG is given as Equation (3.7).

$$E\_NPG\_DPG = \sum_{Exe\_Period}^{n} \sum_{i=1} (Exe\_ON\_Leak\_DPG)_i + \sum_{Idle\_Period}^{n} \sum_{i=1} (Idle\_ON\_\_DPG)_i \tag{3.7}$$

Comparing E_PG_DPG to E_NPG_DPG gives an indication of the effectiveness of our power gating algorithm.

E_NPG_NPG: This represents the total leakage energy consumed by an accelerator if it were to run on a non power-gated FPGA architecture, keeping the accelerator powered-on during idle periods. E_NPG_NPG is given as Equation (3.8).

$$E\_NPG\_NPG = \sum_{Exe\_Period}^{n} \sum_{i=1} (Exe\_ON\_Leak\_NPG)_i + \sum_{Idle\_Period}^{n} \sum_{i=1} (Idle\_ON\_Leak\_NPG)_i \quad (3.8)$$

Comparing E_PG_DPG to E_NPG_NPG gives an indication of the effectiveness of power gating approach.

The LegUp scheduler performs *As-Soon-As-Possible (ASAP)* scheduling. This default scheduling policy has been used in the experiments. All the benchmark circuits in LegUp framework have been functionally verified in hardware targeting a Cyclone-II FPGA on Altera DE2 board by running them at 66Mhz clock. Therefore, by default, the scheduler uses a 66Mhz clock period constraint while scheduling and chaining the operations. We assume the same clock period for calculating various metrics such as wake-up cycles or break-even cycles as will be discussed next.

Of the circuits in the CHStone suite, we present power gating potential and savings results for the five benchmarks below.

3.3.2.1 ADPCM

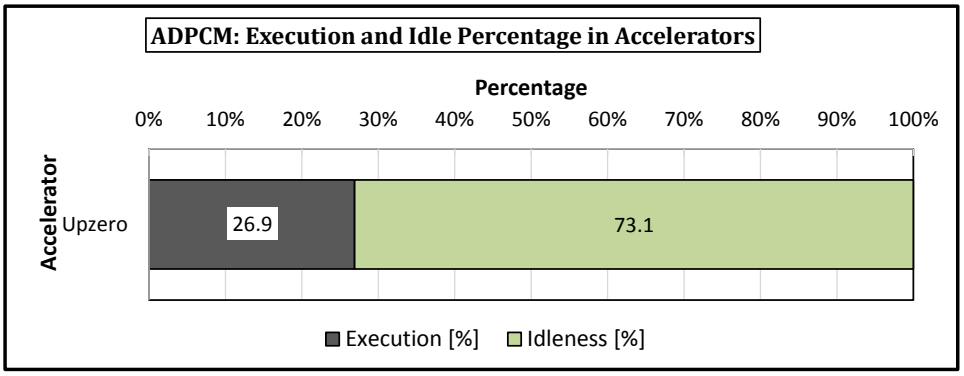
We start with a relatively simple benchmark, *ADPCM* from the media class, which has only one accelerator, *UpZero*. The size and power gating overhead of *UpZero* is reported in Table 3.3. The rows labeled *Percentage of DC PGRs* and *Percentage of DC SBs* show the percentage of power-gated regions (PGRs) and switch-blocks (SBs) respectively that can be *dynamically controlled (DC)*; typically not all PGRs and switch blocks in an accelerator can be turned off, since routing switches within a switch block or other parts of a PGR must remain active to implement parts of the circuit that can not be turned off as determined by the mapping phase discussed in Section 3.2.2.3. The row labeled *Wake-Up Cycles* is the number of cycles to transfer from sleep to normal execution mode and is used to quantify the impact of power gating on execution length. The row labled *Break Even Time* shows the accelerator's energy-break even time which is converted into energy-break even cycles (EBC) assuming a 66Mhz clock. In Table 3.3, all the energies are expressed in *Joules* and all the times are expressed in *Seconds*.

Table 3.3: ADPCM Benchmark Accelerator: Size and Power-Gating Overhead. [PGR=Power Gated Region, SB=Switch Block, DC=Dynamically Controlled]

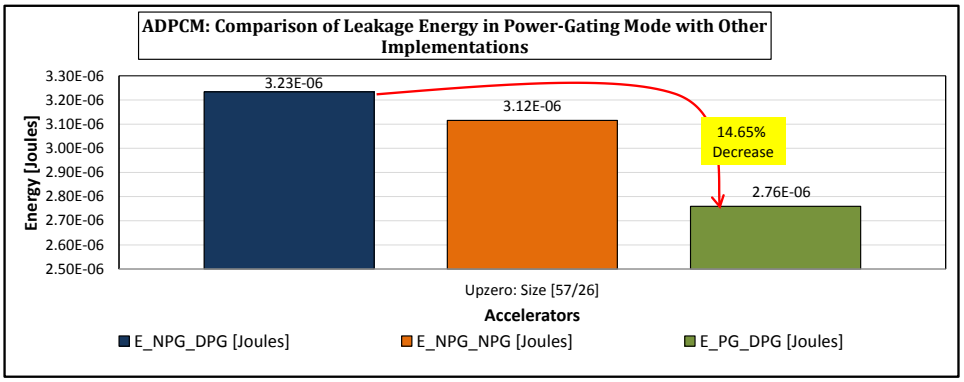
		ADPCM: Accelerator
		UpZero
Size	*Total PGRs*	57
	Percentage of DC PGRs [%]	45.61
	Total SBs	934
	Percentage of DC SBs [%]	3.00
Overhead	*Time to Turn Off*	6.59E-09
	Time to Turn On	9.28E-09
	Wake-Up Cycles	1
	Energy to Turn Off	2.10E-11
	Energy to Turn On	1.12E-10
	Switching Energy	1.33E-10
	Break Even Cycles	5
	Break Even Time	7.25E-08

Figure 3.7(a) shows the execution and idle time percentages in the *UpZero* accelerator which remains idle for 73.1% of the execution time. The idle time presents a potential power gating opportunity. Figure 3.7(b) plots the various estimated energies to gauge the impact of power gating. The X-axis of Figure 3.7(b) shows the name of the accelerator with its size; first number indicates the total number of PGRs and the second number indicates the number of dynamically controlled PGRs. As discussed above, E_PG_DPG represents the total leakage energy consumed by the *UpZero* accelerator in power gating mode, in which it is turned off during its idle period if it is deemed profitable, on an architecture supporting dynamic power gating. In doing so, the *UpZero* accelerator dissipates 14.65% less leakage energy compared to the case in which it is not power-gated during its idle periods. Similarly, E_PG_DPG is 11.43% smaller compared to E_NPG_NPG which highlights the advantage of having a power gating feature in an FPGA architecture.

As can be observed in Figure 3.7(b), E_NPG_DPG which is the energy



(a) ADPCM: Execution and Idle Percentage in *Upzero* Accelerator.



(b) ADPCM: Comparison of Leakage Energy in Power-Gating Mode on an Architecture supporting Dynamic Power-Gating, E_PG_DPG, with other Implementations

Figure 3.7: ADPCM Benchmark

consumed by *UpZero* in non power gating mode on an architecture supporting dynamic power gating is higher compared to E_NPG_NPG, the case if *UpZero* were to run on a non power-gated FPGA architecture. This is due to the design of dynamic power gating architecture which suffers a 10% performance degradation due to the presence of power gating circuitry.

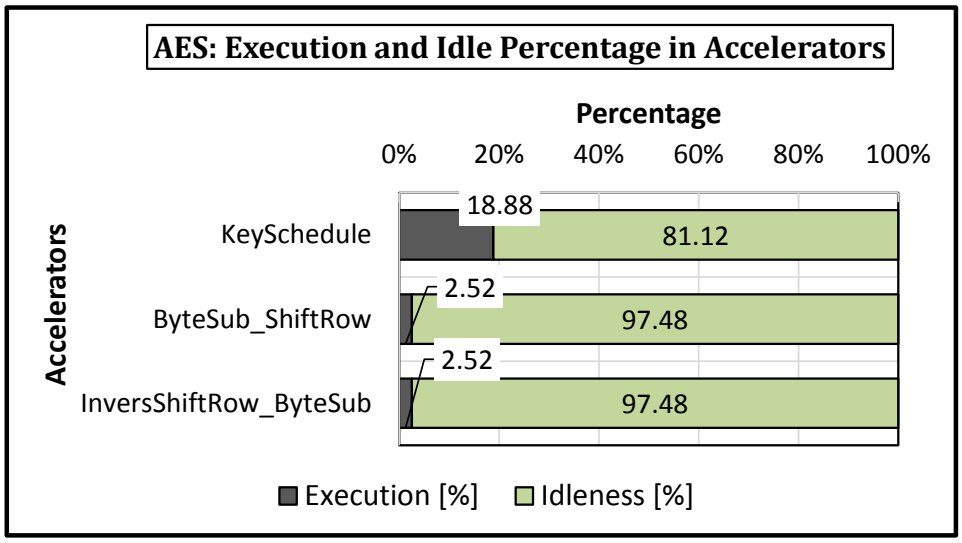
3.3.2.2 AES

The AES benchmark is from the encryption class and has three accelerators: *KeySchedule*, *ByteSub_ShiftRow* and *InversShiftRow_ByteSub*. The size and power gating overhead of these accelerators is reported in Table 3.4. The rows labeled *Percentage of DC PGRs* and *Percentage of DC SBs* show the percentage of power-gated regions (PGRs) and switch-blocks (SBs) respectively that can be *dynamically controlled (DC)*. Of all the accelerators in AES, *KeySchedule* is the largest and has the highest overhead.

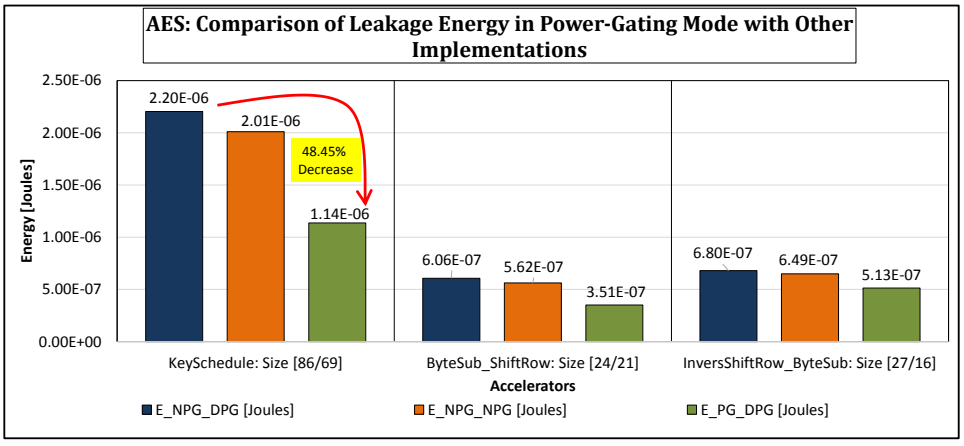
Figure 3.8(a) shows an ample amount of idleness available in all the three accelerators of the AES benchmark. Figure 3.8(b) plots the various estimated energies. The X-axis of Figure 3.8(b) shows the name of the accelerator with its size; first number indicates the total number of PGRs and the second number indicates the number of dynamically controlled PGRs. As can be seen in the graph of Figure 3.8(b), the E_PG_DPG in *KeySchedule* accelerator is decreased by 48.45% compared to E_NPG_DPG. Similarly, the E_PG_DPG implementation of *ByteSub_ShiftRow* and *InversShiftRow_ByteSub*

Table 3.4: AES Benchmark Accelerators: Size and Overhead.

		AES Accelerators		
		KeySchedule	**ByteSub-ShiftRow**	**InversShift-Row_ByteSub**
Size	*Total PGRs*	86	24	27
	Percentage of DC PGRs [%]	80.23	87.50	59.26
	Total SBs	1500	414	459
	Percentage of DC SBs [%]	50.33	15.46	2.61
Overhead	*Time to Turn Off*	6.59E-09	6.59E-09	6.59E-09
	Time to Turn On	2.46E-08	7.34E-09	5.62E-09
	Wake-Up Cycles	2	1	1
	Energy to Turn Off	5.97E-11	1.68E-11	1.27E-11
	Energy to Turn On	3.13E-10	8.94E-11	6.75E-11
	Switching Energy	3.73E-10	1.06E-10	8.02E-11
	Break Even Cycles	3	5	5
	Break Even Time	4.49E-08	6.40E-08	7.41E-08



(a) AES: Execution and Idle Percentage in Various Accelerators.



(b) AES: Comparison of Leakage Energy in Power-Gating Mode on an Architecture supporting Dynamic Power-Gating, E_PG_DPG, with other Implementations.

Figure 3.8: AES Benchmark

accelerators are 42.16% and 24.5% less than their E_NPG_DPG counterparts. As before, the E_NPG_DPG in all the three accelerators is slightly higher than E_NPG_NPG due to power gating overhead.

3.3.2.3 Motion

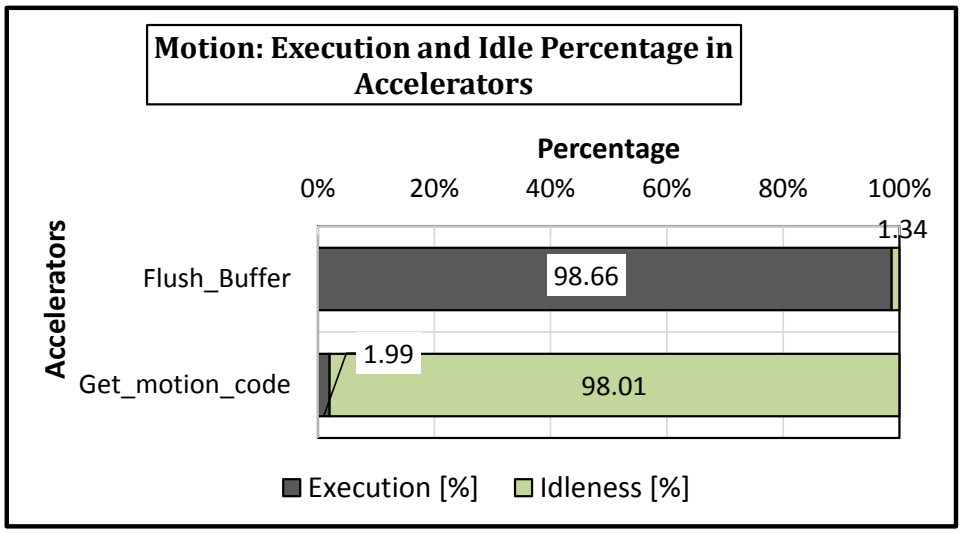
Motion benchmark is from the media class and has two accelerators: *Flush_Buffer* and *Get_Motion_Code*. The size and power gating overhead of these accelerators is reported in Table 3.5.

The graph in Figure 3.9(a) shows that the *Flush_Buffer* accelerator executes 98.66% of the time, leaving little power gating opportunities. Correspondingly, the energy in the power gating mode, E_PG_DPG, is increased by 3.64% due to power gating overhead which could not be amortized due to limited idleness.

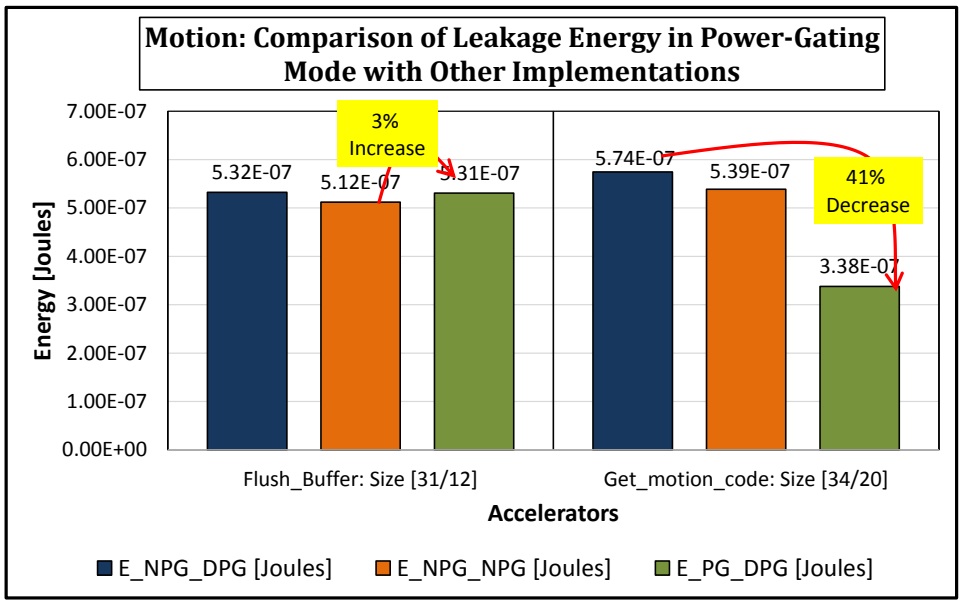
On the other hand, the *Get_Motion_Code* accelerator remains idle for a significant amount of execution time as shown in Figure 3.9(a). Power-gating the *Get_Motion_Code* accelerator during its idle periods yields 41% savings compared to the non power gating mode on a dynamic power-gated architecture.

Table 3.5: Motion Benchmark Accelerators: Size and Overhead.

		Motion Accelerators	
		Flush_Buffer	**Get_motion_code**
Size	*Total PGRs*	31	34
	Percentage of DC PGRs [%]	38.71	58.82
	Total SBs	545	584
	Percentage of DC SBs [%]	17.25	33.05
Overhead	*Time to Turn Off*	6.59E-09	6.59E-09
	Time to Turn On	4.28E-09	7.05E-09
	Wake-Up Cycles	1	1
	Energy to Turn Off	1.02E-11	1.70E-11
	Energy to Turn On	5.35E-11	8.91E-11
	Switching Energy	6.37E-11	1.06E-10
	Break Even Cycles	4	4
	Break Even Time	5.08E-08	4.69E-08



(a) Motion: Execution and Idle Percentage in Various Accelerators.



(b) Motion: Comparison of Leakage Energy in Power-Gating Mode on an Architecture supporting Dynamic Power-Gating, E_PG_DPG, with other Implementations.

Figure 3.9: Motion Benchmark

3.3.2.4 SHA

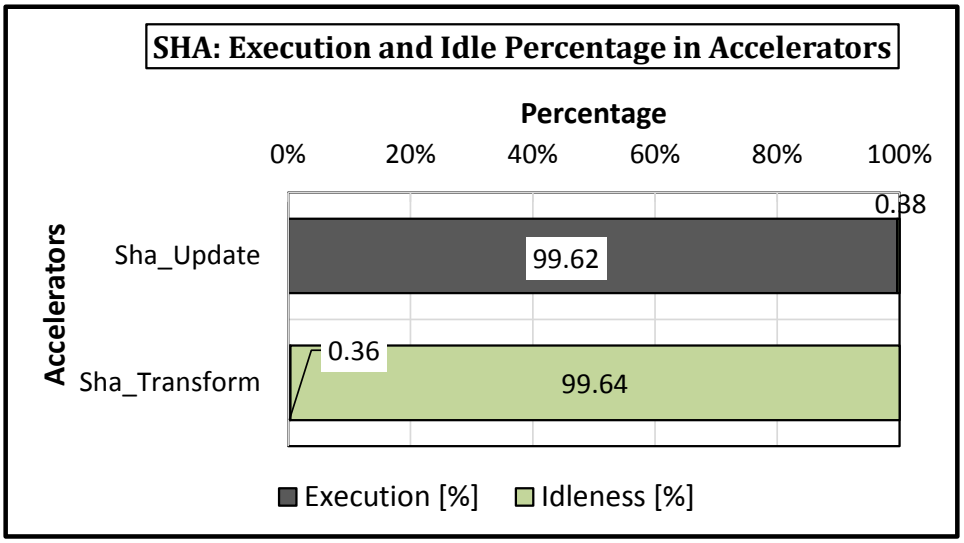
The SHA benchmark is from the encryption class and has two accelerators: *Sha_Update* and *Sha_Transform*. The size and power gating overhead of these accelerators is reported in Table 3.6. The results from this benchmark have the same trends as were seen in the Motion benchmark results. The *Sha_Update* accelerator is only idle for 0.38% of the execution time as depicted in Figure 3.10(a). Correspondingly, the E_PG_DPG, that is the energy in the power gating mode on a dynamic power-gated architecture, is increased by 10% due to power gating overhead. On the other hand, the *Sha_Transform* accelerator remains idle for 99.64% of the execution time as shown in Figure 3.9(a). Correspondingly, its E_PG_DPG is decreased by 52% due to ample power gating as compared to E_NPG_DPG.

3.3.2.5 JPEG

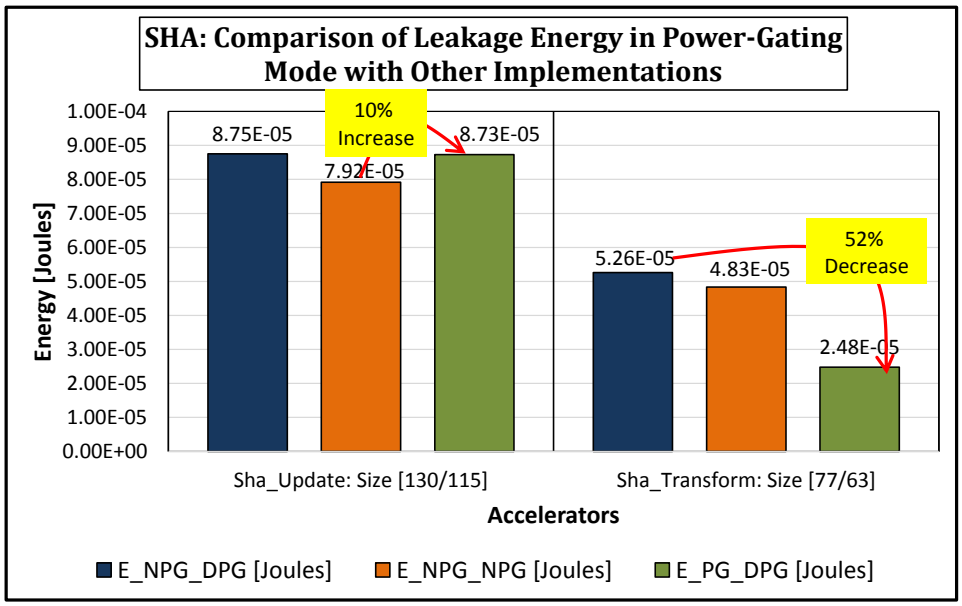
The JPEG benchmark is from the media class and has three accelerators: *Write4Block*, *Decode_Block* and *Huff_make_dhuff_tb*. The size and power gating overhead of these accelerators is reported in Table 3.7. These

Table 3.6: SHA Benchmark Accelerators: Size and Overhead.

		SHA Accelerators	
		Sha_update	**Sha_transform**
Size	*Total PGRs*	130	77
	Percentage of DC PGRs [%]	88.46	81.81
	Total SBs	2157	1313
	Percentage of DC SBs [%]	50.02	36.40
Overhead	*Time to Turn Off*	6.59E-09	6.59E-09
	Time to Turn On	4.10E-08	2.25E-08
	Wake-Up Cycles	3	2
	Energy to Turn Off	9.85E-11	5.33E-11
	Energy to Turn On	5.17E-10	2.81E-10
	Switching Energy	6.16E-10	3.34E-10
	Break Even Cycles	4	4
	Break Even Time	4.76E-08	5.13E-08



(a) SHA: Execution and Idle Percentage in Various Accelerators.



(b) SHA: Comparison of Leakage Energy in Power-Gating Mode on an Architecture supporting Dynamic Power-Gating, E_PG_DPG, with other Implementations.

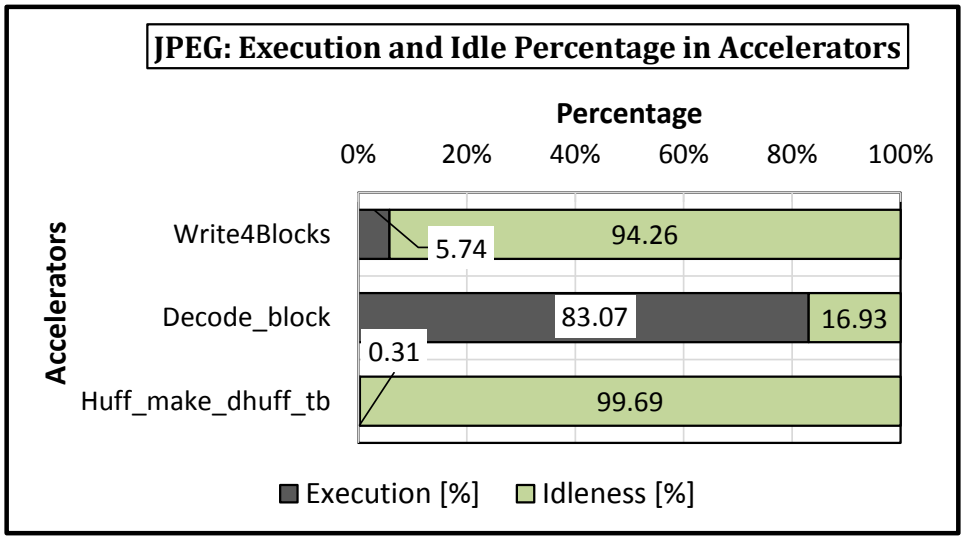
Figure 3.10: SHA Benchmark

accelerators are larger compared to the accelerators in the other benchmarks.

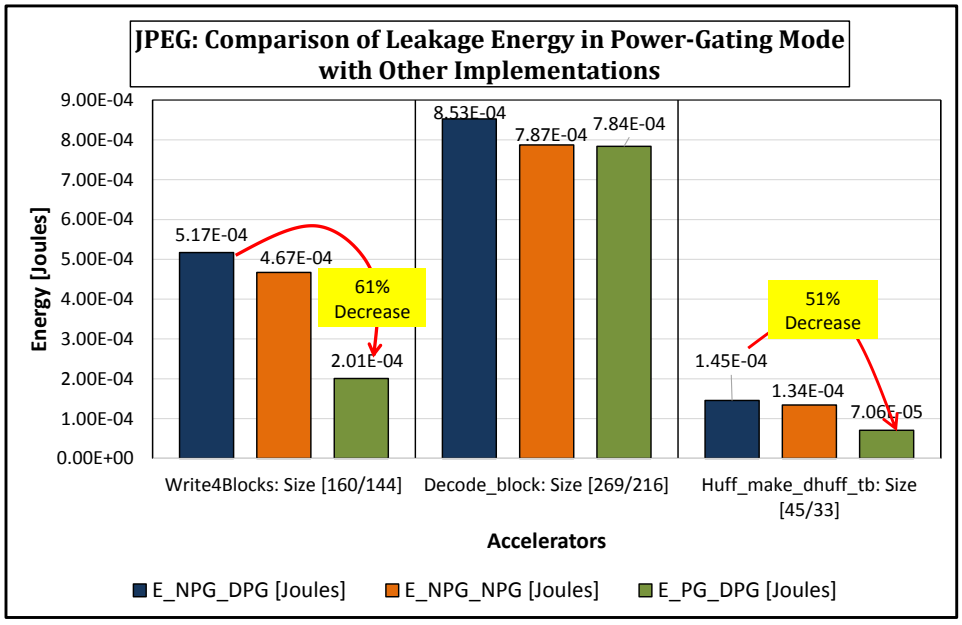
Figure 3.11(a) shows significant idleness available in the *Write4Block* and *Huff_make_dhuff_tb* accelerators which leads to leakage savings of 61% and 51% respectively when compared with E_NPG_DPG. Similarly, the E_PG_DPG, that is the energy in the power gating mode on a dynamic power-gated architecture, in *Write4Block* and *Huff_make_dhuff_tb* accelerators is decreased by 57% and 47% respectively compared to E_NPG_NPG, that is the energy in the non power gating mode on a non power-gated architecture. The *Decode_Block* accelerator saves 8% leakage energy as compared to its non power gating counterpart. The smaller savings are due to less idleness, 16.93%, available in the *Decode_Block* accelerator. Naturally, the E_NPG_DPG in all the three accelerators is more than E_NPG_NPG due to the power gating overhead.

Table 3.7: JPEG Benchmark: Size and Overhead.

		JPEG Accelerators		
		Write4-Block	**Decode-Block**	**Huff_make_dhuff_tb**
Size	*Total PGRs*	160	269	45
	Percentage of DC PGRs [%]	90	80.30	73.33
	Total SBs	2681	4358	753
	Percentage of DC SBs [%]	51.81	26.53	41.43
Overhead	*Time to Turn Off*	6.59E-09	6.59E-09	6.59E-09
	Time to Turn On	5.14E-08	7.71E-08	1.14E-08
	Wake-Up Cycles	4	6	1
	Energy to Turn Off	1.24E-10	1.80E-10	2.75E-11
	Energy to Turn On	6.49E-10	9.51E-10	1.45E-10
	Switching Energy	7.72E-10	1.13E-09	1.72E-10
	Break Even Cycles	4	4	4
	Break Even Time	4.71E-08	5.68E-08	4.70E-08



(a) JPEG: Execution and Idle Percentage in Various Accelerators.



(b) JPEG: Comparison of Leakage Energy in Power-Gating Mode on an Architecture supporting Dynamic Power-Gating, E_PG_DPG, with other Implementations.

Figure 3.11: JPEG Benchmark

3.3.3 Power-Gating Overhead

Every time an accelerator transitions between the *sleep* and *active* modes, it incurs two major penalties: switching overhead and execution overhead. These overheads are discussed in the following sections.

3.3.3.1 Switching Energy Overhead

Every time an accelerator enters and then exits a *sleep* mode, there is a switching energy penalty; it takes some energy to turn off an accelerator and back on at the end of the idle period. An accelerator should only be power gated if it will be idle long enough to compensate for this penalty. The energy break-even time is the minimum idle time for which an accelerator should be power gated. The switching overhead increases with the accelerator size as depicted in Figure 3.12. The X-axis of Figure 3.12 shows the overhead energy to turn on (E_{TurnOn}) and turn off ($E_{TurnOff}$) an accelerator. The Y-axis shows the sizes of various accelerators, expressed in terms of the number of power-gated regions that can be power-gated. As can be expected, the E_{TOn} is greater than E_{TOff} due to the fact that during wake up of an accelerator significant amount of power is required to charge internal capacitances which adds delay.

The leakage saving results presented in the previous section take into account this switching overhead.

3.3.3.2 Impact of Power-Gating on Execution Length

Each power gating event incurs extra cycles to transition between *sleep* and *active* modes which impacts the execution length of the program. Figure 3.13 shows the percentage increase in the execution length for various accelerators. The X-axis of the Figure 3.13 shows accelerators in various benchmarks and Y-axis is the percentage increase in their execution length. The numbers presented in the graph assume that the accelerator starts to wake up at the end of the idle period i.e when the start event for the next execution period is detected; when this occurs the accelerator waits for the duration of wake-up cycles before resuming execution. Therefore, the percentage increase in the graph shows the worst case scenario. It is possible to hide the wake-up latency inside the idle period, such that the accelerator is ready to execute again at the end of its idle period, to reduce the impact on execution length.

As can be seen from the graph, power gating increases the run-time of *Upzero* accelerator by 0.46%; this is due to the large number of power gating

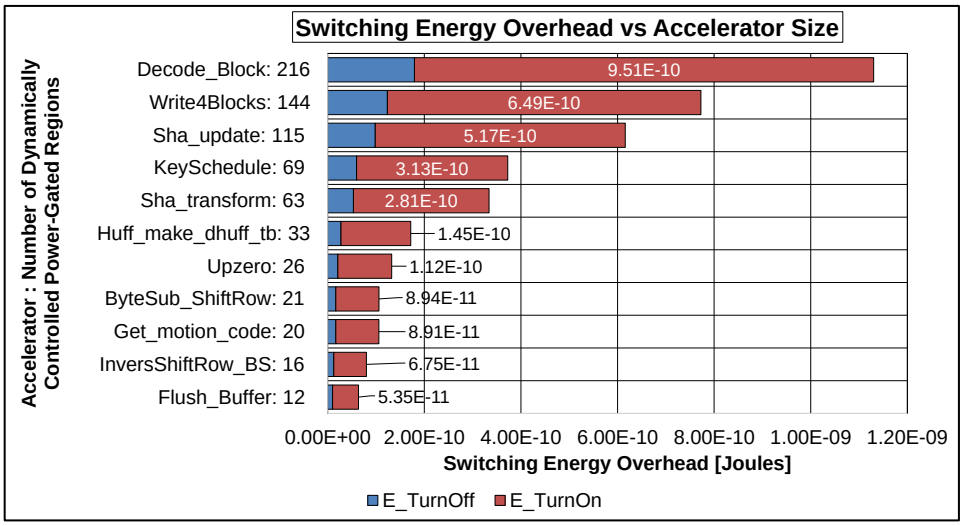


Figure 3.12: Number of PGRs in an Accelerator that can be Dynamically Turned-off and their Switching Energy Overhead

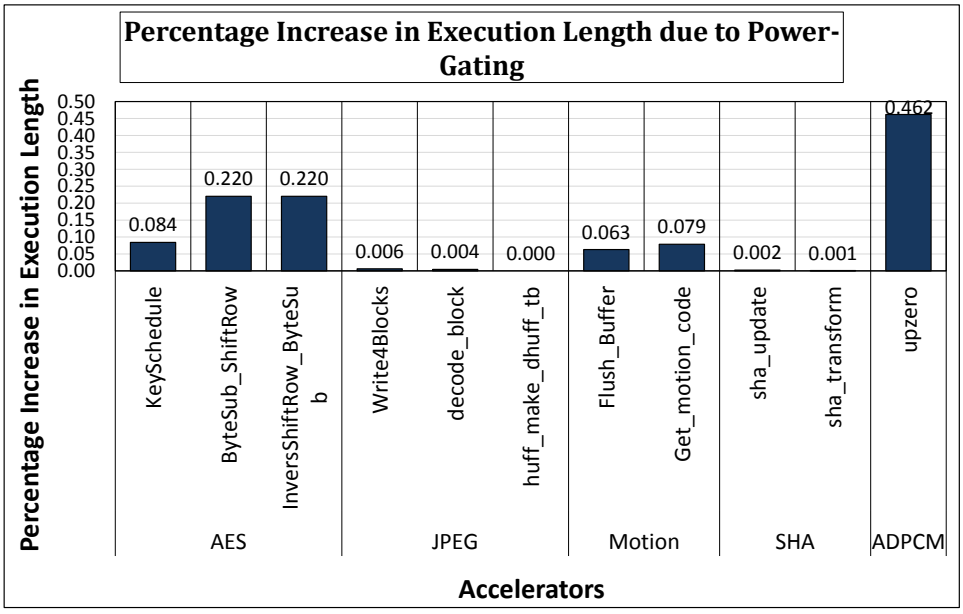


Figure 3.13: Worst-Case Percentage Increase in Execution Length due to Power-Gating Overhead

events, which are 100 in number in *Upzero* accelerator.

3.4 Summary

In this chapter, we described a high-level synthesis (HLS) based design methodology that uncovers the power gating opportunities that may exist in an application. We consider accelerator-level power gating in which an entire accelerator (including all its sub-accelerators) are power gated as a single unit. The execution schedule generated by the high-level synthesis tool is used to determine the idle periods of individual hardware accelerators in a synthesized system. The predicted length of these idle periods is used to determine whether the power saved by power-gating the accelerator is more than the overhead of turning the accelerator off and then on again at the end of the idle period. Based on this knowledge, individual accelerators can then be power-gated when it is deemed profitable.

The achievable leakage savings is influenced by the amount of idleness in the accelerator, which depends on the nature of the input vectors. In addition, the leakage savings is influenced by the size of the portion of the accelerator that can be turned-off at run-time. The design of the dynamic power gating architecture, inherently suffers from performance degradation due to the presence of power gating circuitry. Moreover, power gating incurs an extra energy overhead in switching between the *sleep* and *active* modes; an accelerator should only be power-gated if it will be idle long enough to compensate for this energy penalty. Results on the CHStone benchmark suite show that power gating can reduce the leakage power by 14% to 61% in individual accelerators.

Chapter 4

Hierarchical Dynamic Power-Gating in FPGAs

4.1 Context

A typical system-on-chip (SoC) design created using a high-level synthesis (HLS) tool, such as [CCF+13], contains a processor with one or more hardware *accelerators* used to speed up critical portions of the algorithm. An accelerator may have several phases, each of which is implemented by a separate logic circuit, which we call as a *sub-accelerator*. In Chapter 3, we presented a methodology in which the schedule from a high-level synthesis tool is used to determine the idle periods of individual hardware accelerators in a synthesized system. The predicted length of these idle periods is used to determine whether the power saved by power gating the accelerator is more than the overhead of turning the accelerator off and then on again at the end of the idle period. Based on this knowledge, individual accelerators are then power-gated when it is deemed profitable. In doing so, however, we fixed the granularity of power gating at the accelerator level; an entire accelerator is turned off or on as a unit. For very large accelerators, it may be profitable to power gate at a finer granularity – it may be desirable to turn off individual sub-accelerators while parent or other sub-accelerators are running to gain more leakage savings than power gating an entire accelerator alone. We refer to these intra-accelerator level power gating opportunities as *hierarchical power gating* opportunities.

In this chapter, we present an HLS compiler-assisted framework that automatically detects the *hierarchical* power gating opportunities from an application expressed in C language. A study is presented in which the granularity of power gating is varied to quantify the benefits of making power gating decisions for each sub-accelerator separately, rather than simply at the accelerator level. We show that for some applications, this finer granularity results in more effective power gating, providing more power savings than the previous technique; results on CHStone benchmarks show that hi-

erarchical power gating can save 8% to 25% of static energy when the parent and descendant accelerators are power-gated independently.

A concern that arises as the granularity of power gating decreases is that the power gating decisions may become more sub-optimal i.e. when determining whether an idle period is long enough to make power gating worthwhile, our approach assumes a static schedule constructed using a single set of input vectors. If these input vectors change, the idle times experienced during the run of the application may deviate from the predicted idle times, potentially leading to sub-optimal power gating decisions. For the accelerator-level power gating considered in Chapter 3, the idle periods are long enough that this is not likely to be a concern. Finer-grained power gating, however, implies shorter idle periods, meaning the optimal power gating decisions may be more sensitive to changes in the inputs. In this chapter, we investigate whether this is an issue as the granularity of power gating decreases.

Thus, this chapter presents three contributions:

1. We enhance the design framework presented in Chapter 3 to support hierarchical power gating. Section 4.2 presents our compiler-assisted framework for automatically generating hierarchical power gating decisions.

2. We study the impact of reducing the power gating granularity to consider power gating the sub-accelerators. Section 4.3 quantifies the impact of hierarchical power gating.

3. We investigate whether the compiler-assisted power gating approach used in Chapter 3 is sufficient as the granularity of power gating decreases. Section 4.4 presents our findings across a large number of input patterns.

4.2 Hierarchical Dynamic Power Gating

4.2.1 Context

Although our work will apply to any FPGA which provides dynamic power gating control and any high-level synthesis tool, we describe it in the context of the Dynamic Power-Gated FPGA architecture from [BW10] and the LegUp high-level synthesis tool from [CCF+13]; the background on each was presented in Section 2.4.3.2 and Section 2.2 respectively.

4.2.2 Design Framework

The design framework presented in the previous chapter considers accelerator level power gating opportunities. We enhance the same framework to identify the hierarchical power gating opportunities during HLS. In doing so, we consider not only each function in the C code, but their sub-functions as well. LegUp converts each function into a hardware accelerator. If a function has sub-functions, they get converted into a sub-accelerator. In the context of hierarchical power gating, we refer to the top accelerator in the hierarchy as a *parent accelerator* and the accelerators called by this top accelerator as *child accelerators*. The hierarchical design framework presented in this section considers turning off each of these hardware units separately.

The various phases of the framework are discussed below.

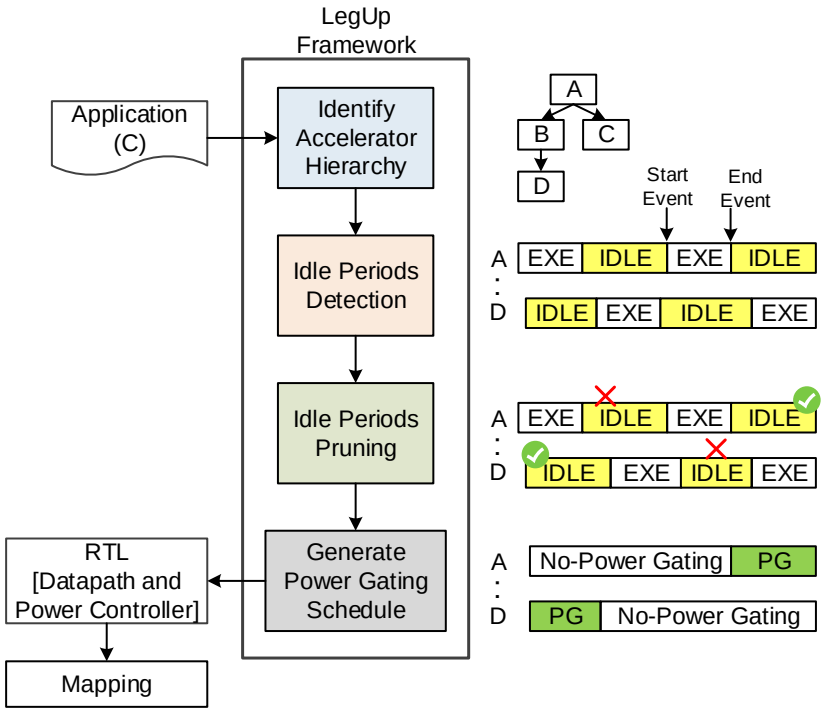


Figure 4.1: HLS Compiler Assisted Hierarchical Power-Gating Framework

4.2.2.1 Identifying Accelerator Hierarchy

The flow starts from identifying all the parent and child accelerators in the design. The input to this phase is a *C* program which gets converted to an *intermediate representation (IR)* by the LLVM front end. Each function in the program, along with its descendant functions, gets decomposed into basic blocks. The LegUp framework emulates the program execution using the *GXemul* program and keeps track of the basic block execution. We intercept this basic block trace information and build a hierarchical call tree which aids in identifying all the parents and their child functions in the hierarchy. These parent and child functions later get synthesized into hardware accelerators.

4.2.2.2 Idle Periods Detection

Each accelerator (parent or child) typically has active and idle periods. The idle periods are potential power gating opportunities. The goal of this phase is to estimate the lengths of all the idle periods that may exist in parent and child accelerators. For this, the basic block trace information from the previous stage is combined with the scheduled lengths of the basic blocks (obtained from the HLS scheduler) to estimate the idle cycles between each invocation of the parent or child accelerator.

Note that the actual duration of the idle period will typically vary from run to run as input patterns change. In Section 4.4, we evaluate the impact of changing inputs on the number of idle periods and their duration.

4.2.2.3 Idle Periods Pruning

The previous phase produces a list of all potential power gating opportunities. However, many of these may not be profitable; powering down (and later powering up) a power gated region incurs both energy and delay overhead. Thus, for each power gating opportunity, we must determine whether a power gating event should be generated.

In general, an accelerator (parent or child) should be turned off when idle if the power saved by power gating the accelerator is more than the overhead of turning the accelerator off and then on again at the end of the idle period. To make this determination, we first find the accelerator's energy break-even time – *the minimum idle time at which the leakage-savings compensate the energy penalty for mode transition.* If the estimated idle time duration, as determined in Section 4.2.2.2, is more than break-even time, we generate a power gating event.

Below, we briefly describe how we find an accelerator's break-even time.

Determining Accelerator's Energy Break-Even Time: Every time an accelerator transitions between the *sleep* and *execution* modes, there is an energy penalty. An accelerator should only be power gated if it will be idle long enough to compensate for this penalty. The energy break-even time is the minimum idle time for which an accelerator should be power gated, and can be calculated as:

$$T_{break-even} = \frac{(P_{1\rightarrow0} \times T_{1\rightarrow0}) + (P_{0\rightarrow1} \times T_{0\rightarrow1})}{P_{ON-leak} - P_{OFF-leak}} \tag{4.1}$$

where $P_{1\rightarrow0}$, $T_{1\rightarrow0}$, is the power and time required to enter power-saving mode. Similarly, $P_{0\rightarrow1}$ and $T_{0\rightarrow1}$ is the power and time required to exit the power-saving mode. $P_{ON-leak}$ and $P_{OFF-leak}$ is the leakage power dissipated during turned-on and turned-off state respectively. The numerator in equation (4.1) represents the switching energy overhead which is denoted by E_{switch} and the denominator represents the power saved which is denoted as P_{saved}.

In context of our target DPG architecture, the power-gated region (PGR) is the basic unit of granularity. Thus, an accelerator occupies an integral number of PGRs and switch-blocks (SBs). Therefore, the switching energy and power-saved can be expressed as,

$$\begin{aligned} E_{switch} = & NumPGR \times (P_{switchPGR} \times T_{switchPGR}) + \\ & NumSBs \times (P_{switchSB} \times T_{switchSB}) \end{aligned} \tag{4.2}$$

$$\begin{aligned} P_{saved} = & NumPGR \times (P_{PGR-on-leak} - P_{PGR-off-leak}) + \\ & NumSBs \times (P_{SB-on-leak} - P_{SB-off-leak}) \end{aligned} \tag{4.3}$$

where in (4.2) and (4.3), $NumPGR$ and $NumSBs$ is the number of PGRs and SBs, respectively, occupied by the accelerator. We find this by performing an initial mapping of the accelerator to the fabric as discussed previously in Section 3.2, however, estimation techniques could also be used.

4.2.2.4 Power-gating Schedule Generation

Once all the profitable power gating opportunities across all the hierarchical accelerators have been identified, a schedule with these decisions is generated. The power gating schedule records the conditions (start and end events) that trigger a particular idle period in an accelerator. Each time

these conditions are met at run-time, power gating is triggered which would put the accelerator in sleep mode.

4.2.2.5 Mapping

Hierarchical power gating requires parent and child accelerators to be treated as independent units for power gating. In doing so, the parent and each child accelerator is assigned to its own power-gated region. This requires the mapping tool to distinguish the logic resources of a child accelerator from its parent such that they could be assigned to independent power-gated regions.

As described earlier, LegUp converts each C-function into a hardware accelerator. If a function has a sub-function, it gets converted into a sub-accelerator. In doing so, the sub-accelerator appears as a hardware module present inside its parent body in the resulting circuit, as shown in Figure 4.2(a). If such a combined circuit (in which the child is a part of the parent body) is synthesized, it would be difficult to distinguish the child's logic resources distinctively from its parent. The problem is that during synthesis the tool assigns arbitrary names to the logic resources of the child accelerator, making it impossible to identify the child's logic resources in the resulting netlist. In order to keep track of the child's logic resources, we remove the child accelerator from its parent body in the RTL. This leaves just the parent accelerator without any sub-accelerator in it. In doing so, both the parent and child accelerators become independent of each other,

(a) An illustration of Combined Parent and Child accelerators in the circuit; Child appears as part of the Parent

(b) An illustration of Separate Parent and Child accelerators in the circuit; Child is independent of the Parent

Figure 4.2: An illustration of Combined and Separate Parent/Child Accelerators

as depicted in Figure 4.2(b).

After separating the parent and child accelerators at the RTL-level, both the parent and child accelerators are synthesized independently and two separate BLIF files (one for each parent and child accelerator) are generated, as depicted in the mapping flow in Figure 4.3. This allows us to identify the parent and child accelerators independently and aid the rest of the tool-chain in properly assigning the parent and child accelerators to their own power-gated regions.

It is to be noted that the separate synthesis of the parent and child accelerators will optimize the logic differently as compared to the combined case where both parent are child accelerators are synthesized together. This may result in variation in the area occupancy for the two cases. As a result, the total number of dynamically controlled PGRs and always-on PGRs will also differ which will impact the leakage savings results.

The implementation steps taken inside the modified version of VPR stays the same as described in Section 3.2.2.3.

4.3 Experimental Results

4.3.1 Experimental Setup

We use the CHStone benchmarks suite developed for C-based high-level synthesis (HLS) [HTH+08]; these benchmarks represent diverse real-world application domains. The C-based source of each benchmark was provided as input to the Legup HLS framework, augmented with our tool flow, shown in Figure 4.1.

Table 4.1 shows the architecture specifications of the target dynamic power-gated FPGA. The HSPICE simulated architecture parameter values, given in Table 4.2, allows us to quantify the impact of hierarchical power

Table 4.1: Architectural Parameters of the Target Dynamic Power-Gated FPGA from [BW10]

Architecture Parameters		
PGR_W	Power-gated Region Width	4
PGR_H	Power-gated Region Height	4
W	Channel Width	120
N	Cluster Size	6

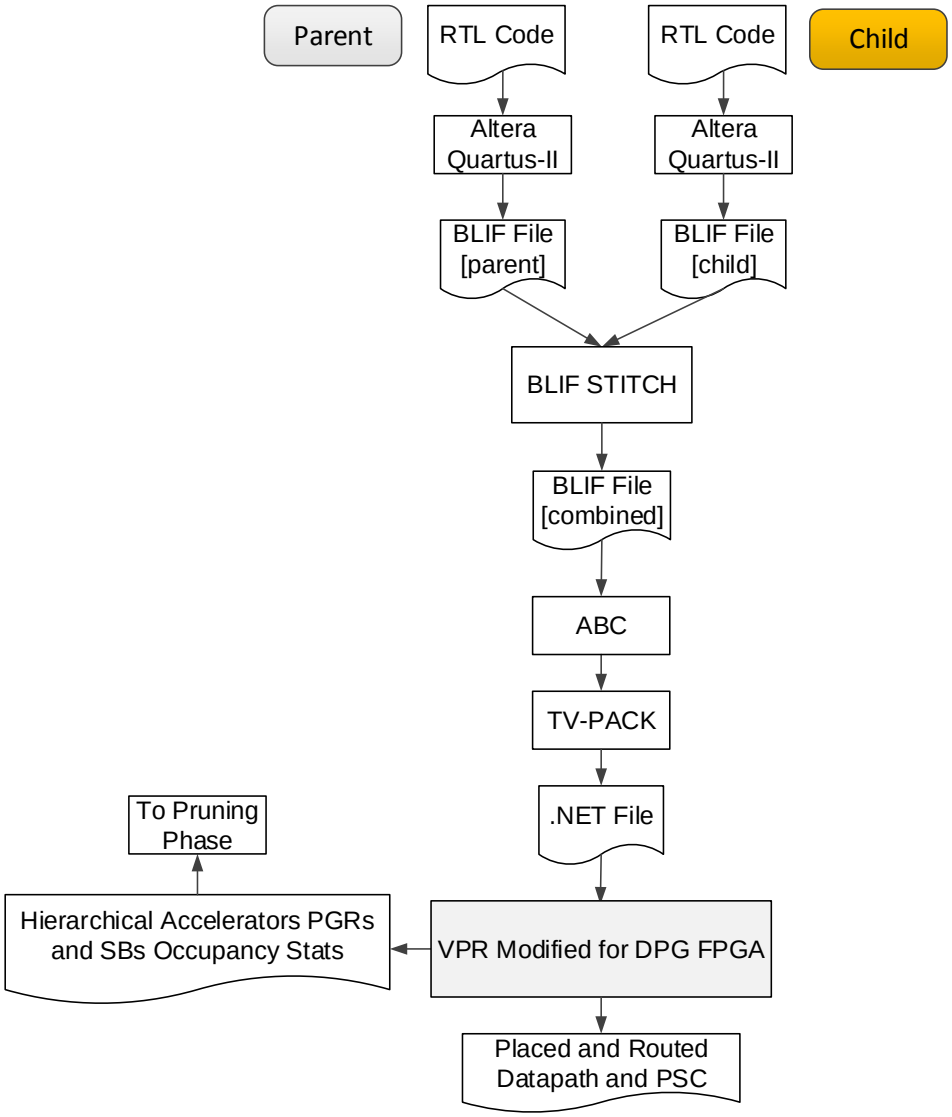


Figure 4.3: Steps inside Mapping of Hierarchical Power Gated Accelerators

gating. The values presented in Tables 4.1 and 4.2 are the same as used in the experiments of Chapter 3.

In LegUp framework, we enable all LLVM optimizations and disable function inlining to maximize the number of functions which could be implemented as hierarchical hardware accelerators. The default LegUp scheduling

Table 4.2: HSPICE Simulated Parameters of the Target Dynamic Power-Gated FPGA from [BW10]. [PGR = Power-Gated Region, SB = Switch-Block]

	Simulated Parameters	**PGR**	**SB**
$P_{1\rightarrow0}$	Power to Turn-Off [Watt]	1.22E-04	1.45E-05
$P_{0\rightarrow1}$	Power to Turn-On [Watt]	5.98E-04	5.53E-05
$T_{1\rightarrow0}$	Time to Turn-Off [sec]	6.59E-09	4.12E-10
$T_{0\rightarrow1}$	Time to Turn-On [sec]	7.14E-09	4.46E-10
$P_{ON-leak}$	On Leakgae Power [Watt]	6.70E-05	5.36E-06
$P_{OFF-leak}$	Off Leakgae Power [Watt]	2.03E-06	2.93E-07

policy, *As-Soon-As-Possible (ASAP)*, has been used in the experiments. We assume a 66Mhz clock period for calculating various metrics such as wake-up cycles and break-even cycles.

4.3.2 Hierarchical Power-Gating Evaluation

In order to quantify the impact of hierarchical power gating, we propose the following power gating policies and apply to the CHStone benchmarks to compare their impact:

Policy - P1: Accelerator-Level Power-Gating: In this policy, the entire accelerator, including all the sub-accelerators of this accelerator, is considered as one power gating unit. The sub-accelerators have no separate power gating control; that means whenever the parent accelerator is active at the end of its idle period, all its sub-accelerators in the hierarchy become active as well. This approach is equivalent to the accelerator-level power gating approach presented in Chapter 3 and serves as a reference.

Policy - P2: Parent Runs while Child Runs: This approach represents intra-accelerator power gating in which the parent and child accelerators are treated as independent units for power gating i.e. each hierarchical accelerator can be controlled independently. In this policy, the parent accelerator remains active and does not sleep after initiating the sub-accelerator. The sub-accelerator is power-gated, however, if only the parent is required to be active.

Policy - P3: Parent is Power-Gated while Child Runs: This approach also represents intra-accelerator level power gating. In this approach, we power-gate the parent accelerator after it has started a sub-accelerator. Once the sub-accelerator has finished processing, the parent accelerator is woken up.

The proposed power gating policies are evaluated based on the total leakage energy consumed by an accelerator, denoted as *E_PG*. The leakage energy in all the execution and idle periods is added together to estimate *E_PG*. The DPG architecture in [BW10] suffers a 10% performance degradation due to the presence of power gating circuitry. To account for this, we increase the idle and execution time periods by this degradation factor when calculating *E_PG*.

Of the benchmarks in CHStone suite, three have sub-accelerator level hierarchy, so we focus on those three benchmarks and present the impact of hierarchical power gating below.

4.3.2.1 SHA

The hierarchical call tree of the SHA benchmark is shown in Figure 4.4. There are two parent-level and one child-level accelerators. We evaluate the above mentioned power gating policies for the *Sha_Update (SU)* accelerator as it has one sub-accelerator – *Sha_Transform*. Note that, *Sha_Tansform* appears as both parent and child accelerator, and is instantiated twice when the hardware is generated. We distinguish the child-level accelerator by appending its parent name before it; in this case it would be *Sha_Update_Sha_Tansform (SU_ST)*.

The sizes of various SHA accelerators in various power gating policies are

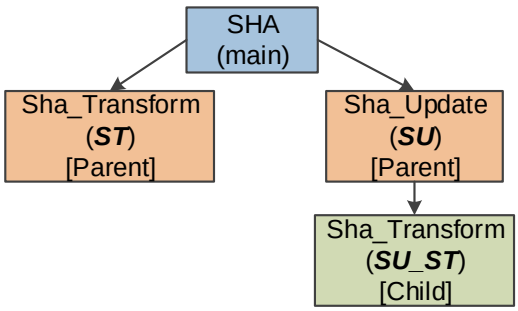


Figure 4.4: SHA Hierarchical Call Tree

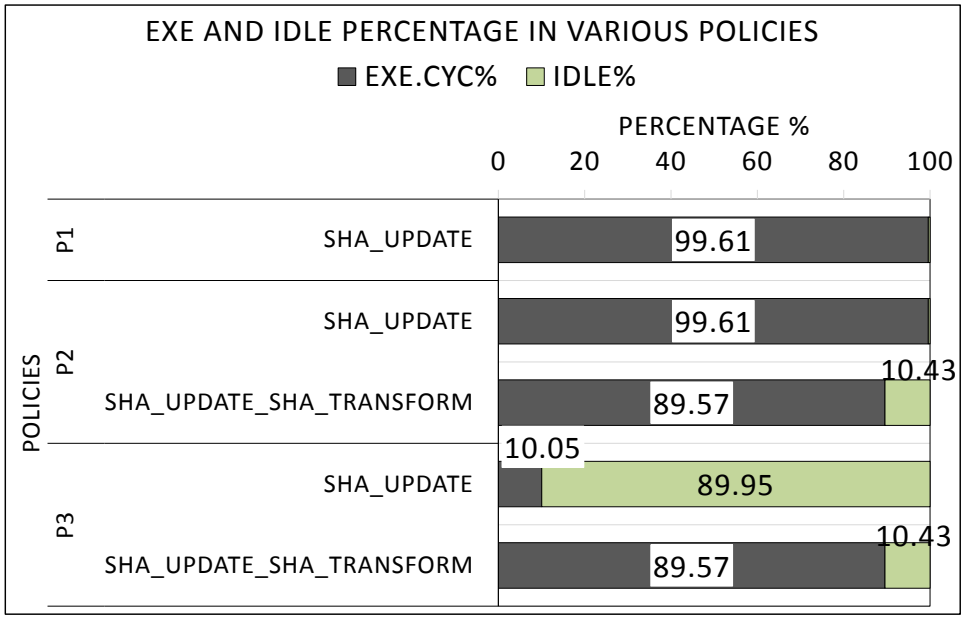
Table 4.3: SHA Occupancy Statistics. (DTF=Dynamically Turned off, AO=Always On)

Policy	Accelerator	Total PGRs	PGRs (DTF)	PGRs (AO)	EBT (sec)	EBC	W.Cyc
P1	SU	130	115	15	4.76E-08	4	3
P2 and P3	SU	52	38	14	4.79E-08	4	1
	SU_ST	75	68	7	4.95E-08	4	2

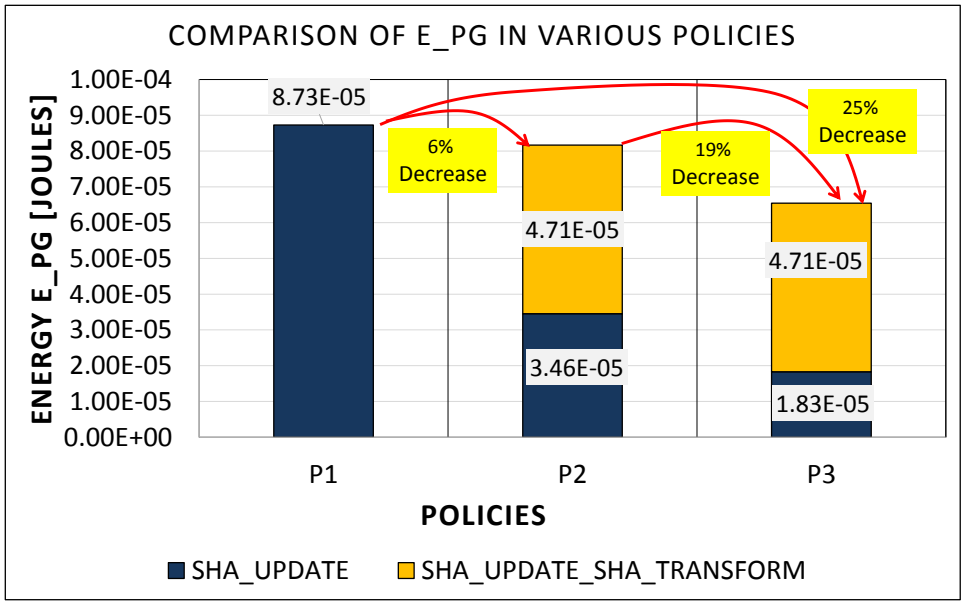
reported in Table 4.3. The column labeled *(Total PGRs)* shows the total number of power-gated regions occupied by an accelerator. The column labeled *(PGRs DTF)* shows the number of power-gated regions (PGRs) in an accelerator that can be *dynamically turned off (DTF)*; typically not all PGRs and switch blocks can be turned off even when an accelerator is idle, since routing switches within a switch block or other parts of a PGR must remain active to implement parts of the circuit that have not been turned off. The column labeled *PGRs AO* shows the *always-on* power-gated regions in each accelerator. The sixth column shows the accelerator's energy break-even time (EBT) which is converted into energy break-even cycles (EBC) assuming a 66Mhz clock period. The last column is the number of cycles to transfer from sleep to normal execution mode, denoted as *wake-up cycles (W.Cyc)*, and is used to quantify the impact of hierarchical power gating on execution length.

Figure 4.5(a) shows the execution and idle time percentages for the parent *Sha_Update (SU)* and child *Sha_Transform* accelerators across the three power gating policies. Correspondingly, Figure 4.5(b) shows the value of E_PG in each power gating policy. In calculating E_PG, the accelerator is turned off during its idle period if deemed profitable.

As can be observed in Figure 4.5(a), policy P2 in which both the parent (SU) and child (SU_ST) accelerators are treated as independent units for power gating, reveals 10% idle time in the child accelerator which was hidden in policy P1. Power gating the child accelerator during 10% of its idle time reduces the net E_PG in P2 by 6% when compared with E_PG in P1, as can be seen in Figure 4.5(b). Next, power gating the parent accelerator (SU) when the child (SU_ST) is executing (policy P3) provides 89% additional idle time in the parent accelerator, as depicted in Figure 4.5(a). Turning off the parent (SU) during this time reduces the leakage in the parent; compare the magnitude of E_PG in *Sha_Update (SU)* in policy P2 and P3 as shown



(a) Percentage of Execution and Idle Times in Various Power Gating Policies



(b) Comparison of Total Leakage Energy Consumed in Various Power Gating Policies

Figure 4.5: SHA Benchmark

in the graph of Figure 4.5(b) – the leakage energy of Sha_Update is reduced in policy P3. This reduction lowers the net E_PG in policy P3 by 19% when compared with P2. This shows the effectiveness of power gating the parent when the child is executing for most of the time. Overall, E_PG in policy $P3$ is reduced by 25% in comparison to policy $P1$. This comparison suggests that it is best to treat the parent and child as independent accelerators which, if are power-gated during their respective idle times can reduce the leakage energy significantly. Therefore, in case of SHA benchmark, policy $P3$ remains best overall.

4.3.2.2 Motion:

The hierarchical call tree of the Motion benchmark is shown in Figure 4.6 and its accelerators sizes in various power gating policies are given in Table 4.4. There are two parent-level and one child-level accelerators. We evaluate the power gating policies for the *Get_Motion_Code (GMC)* accelerator as it has one sub-accelerator, *Flush_Buffer*, which appears as both

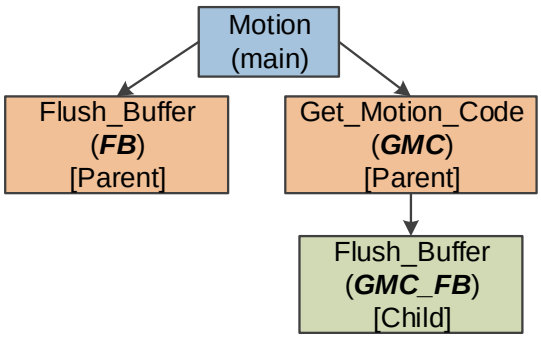


Figure 4.6: Motion Hierarchical Call Tree

Table 4.4: Motion Occupancy Statistics. (DTF=Dynamically Turned off, AO=Always On)

Policy	Accelerator	Total PGRs	PGRs (DTF)	PGRs (AO)	EBT (sec)	EBC	W.Cyc
P1	GMC	34	20	14	4.69E-08	4	1
P2 and P3	GMC	6	1	5	6.83E-08	5	1
	GMC_FB	34	18	16	4.93E-08	4	1

parent and child accelerator and is instantiated twice when the hardware is generated. We distinguish the child-level accelerator by appending its parent name before it; in this case it would be *Flush_Buffer (GMC_FB)*.

Figure 4.7(a) shows the execution and idle time percentages for the parent (GMC) and child (GMC_FB) accelerators across all the three policies and reveals significant idle time in both parent and child accelerators. However, the E_PG in policy P2 is increased by 39% when compared with E_PG in policy P1. This is due to an increase in the area in policy P2 – the individual synthesis of the parent and child accelerators increases the net PGRs (total PGRs in both parent and child accelerators) in policy P2 by 17% when compared with total PGRs in policy P1. As a result, the total always-on PGRs in policy P2 increases by 50% which increases E_PG in policy P2.

In policy P3, the low execution time of the child does not provide any additional parent-level power gating opportunities and therefore the overall E_PG in policy $P2$ and $P3$ remains the same.

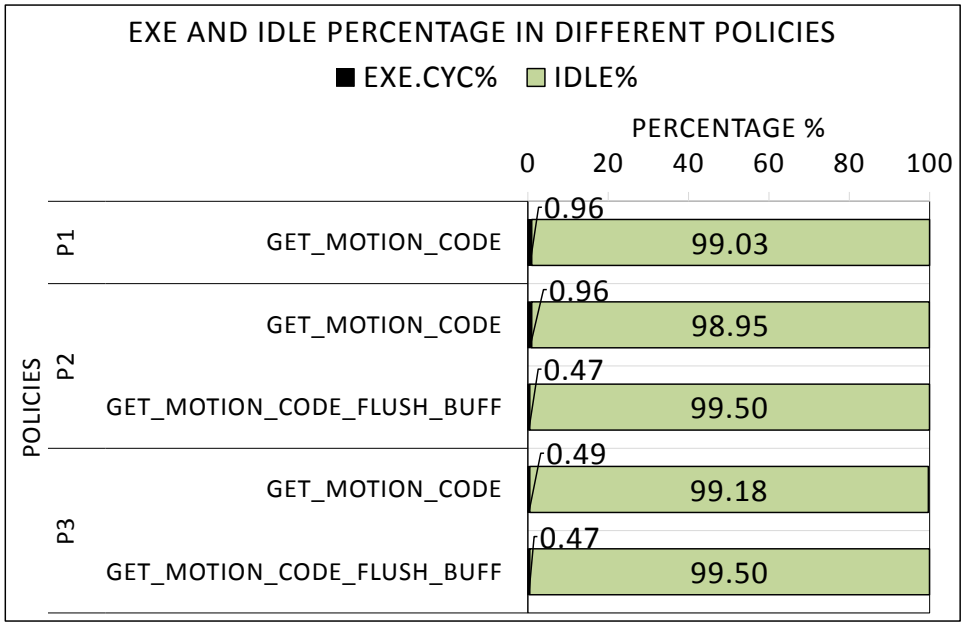
4.3.2.3 JPEG:

The hierarchical call tree of the JPEG benchmark is shown in Figure 4.8. There are three parent-level and two child-level accelerators. We evaluate the power gating policies for the *Decode_Block (DB)* accelerator as it has two descendants. The occupancy statistics of various JPEG accelerators is reported in Table 4.5.

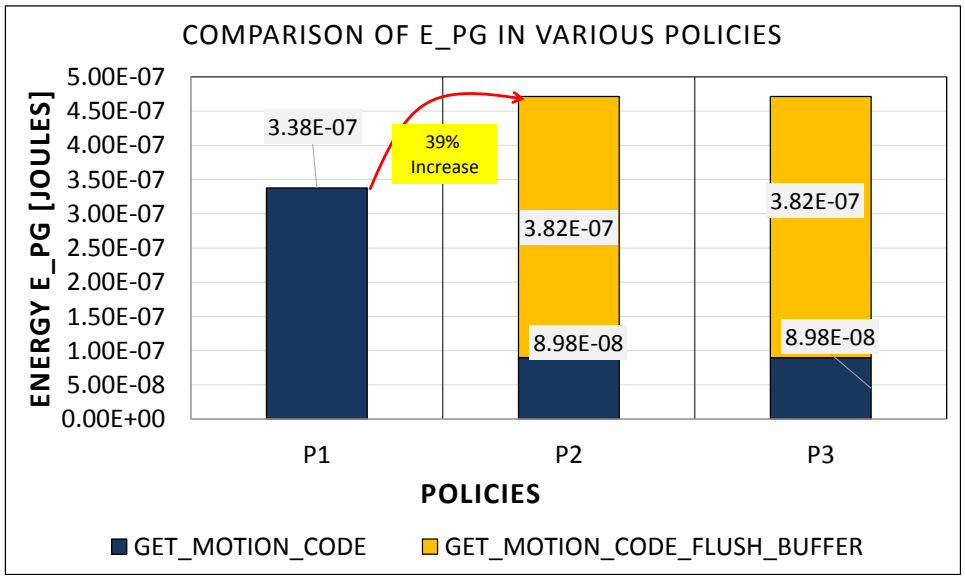
Figure 4.9(a) shows the execution and idle time percentages in parent (Decode_Block (DB)) and child (Decode_Huffman–DB_DH and Buf_Getv–DB_BGv) accelerators in the three power gating policies. The *Decode_Block (DB)* appears as a single accelerator in policy $P1$, as the parent and its descendants are treated as one combined hardware unit for power gating, while its descendants are visible in the other two policies.

Table 4.5: JPEG Occupancy Statistics. (DTF=Dynamically Turned off, AO=Always On)

Policy	Accelerator	Total PGRs	PGRs (DTF)	PGRs (AO)	EBT (sec)	EBC	W.Cyc
P1	DB	269	216	53	5.68E-08	4	6
P2 and P3	DB	222	155	67	6.01E-08	4	4
	DB_DH	31	21	10	5.91E-08	4	1
	DB_BGv	25	19	6	6.50E-08	5	1



(a) Percentage of Execution and Idle Times in Various Power Gating Policies



(b) Comparison of Total Leakage Energy Consumed in Various Power Gating Policies

Figure 4.7: Motion Benchmark

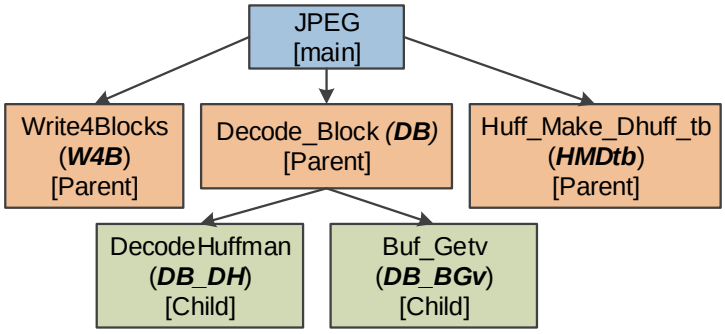


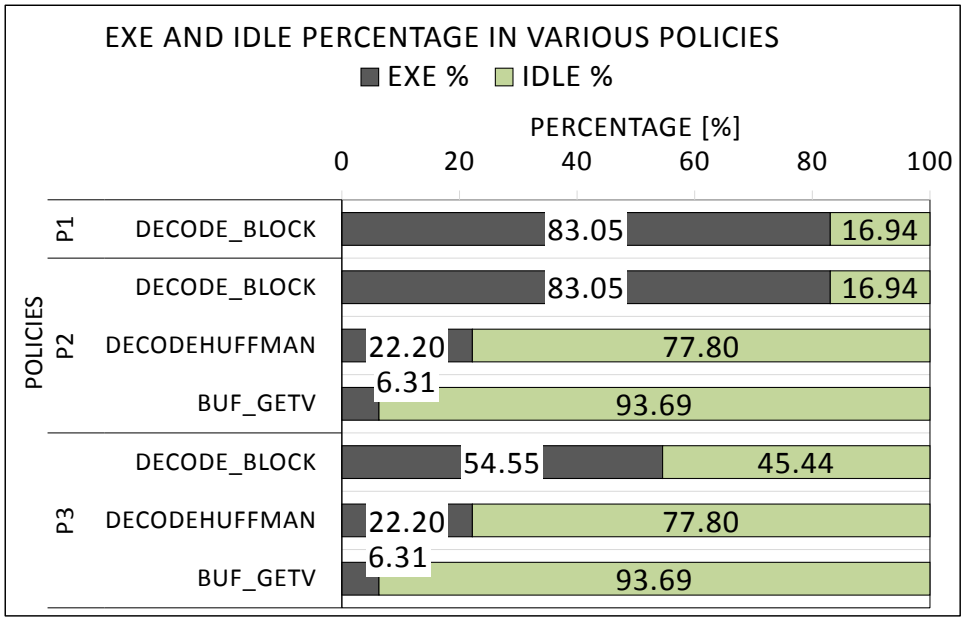
Figure 4.8: JPEG Hierarchical Call Tree

The graph in Figure 4.9(b) shows the E_PG comparison across various power gating policies. The E_PG for the parent and child accelerators are added together in policy $P2$ and $P3$. As can be observed from the graph in Figure 4.9(b), E_PG in $P2$ remains the same as E_PG in $P1$. This is due to the increased number of always-on PGRs in P2 which does not improve the E_PG in $P2$. However, power gating the parent accelerator while its descendants are running in $P3$ reduces the leakage energy in the parent (Decode_Block (DB)) accelerator; observe the reduction in the bar height of DB in policy P3 as compared to P2. This reduction in parent-level leakage energy reduces E_PG in P3 by 8% when compared with P2. Overall, E_PG in $P3$ is 8% less than E_PG in $P1$. This shows the effectiveness of having independent power gating control of parent and child accelerator which enable intra-accelerator level power gating opportunities to save additional leakage power.

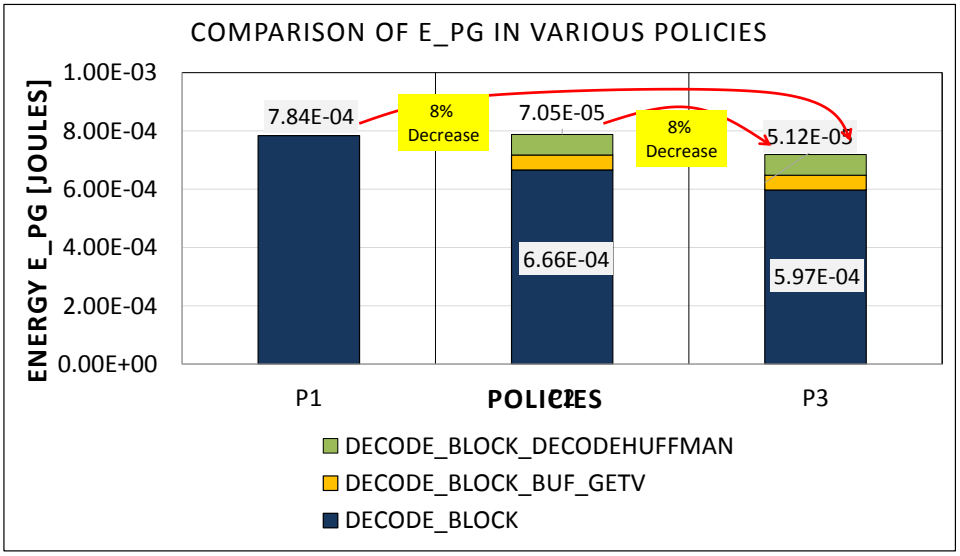
4.3.3 Impact of Power-Gating on Execution Length

Each power gating event incurs extra cycles to transition between *sleep* and *active* modes. The number of cycles to transfer from sleep to active mode, denoted as *wake-up cycles* (W.Cyc) in the previous tables, represent the overhead and determines the impact on execution length; powering up internal capacitances requires significant power and the need to throttle power gating circuitry to avoid inrush current problems adds delay overhead.

Figure 4.10 shows the percentage increase in run-time for various accelerators due to power gating overhead in policy P1, P2 and P3. Policy P3, in which power gating of the parent while the child runs, introduces more idle periods for the parent which increase the overhead; power gating the parent



(a) Percentage of Execution and Idle Times in Various Power Gating Policies



(b) Comparison of Total Leakage Energy Consumed in Various Power Gating Policies

Figure 4.9: JPEG Benchmark

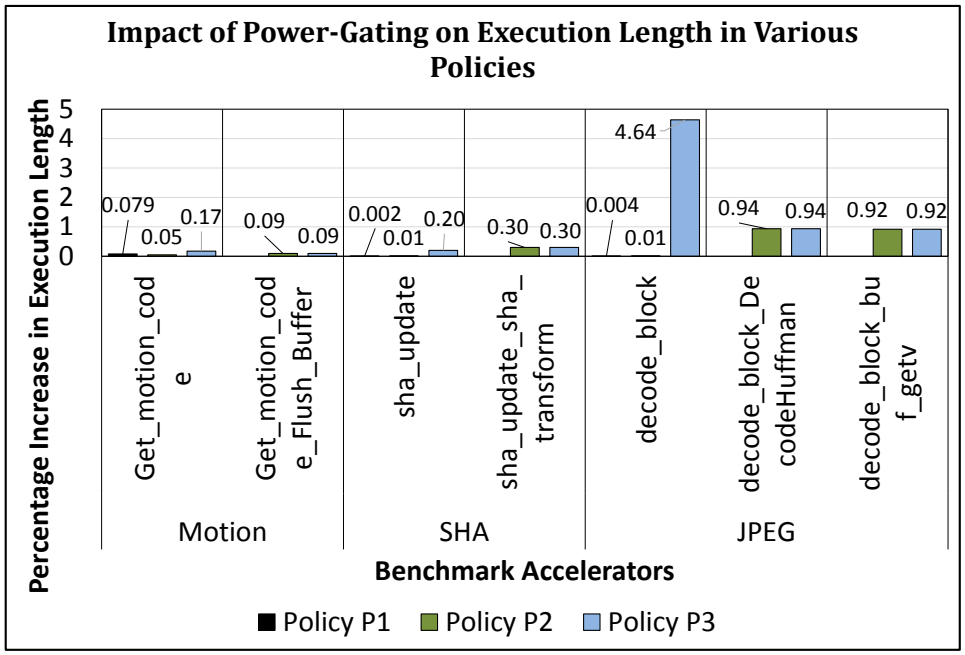


Figure 4.10: Impact of Power-Gating on Execution Length in Various Power Gating Policies

accelerator during each of its idle period incurs extra clock cycles to enter the sleep mode and then wake-up from it. As a result, the execution length of the parent accelerator increases more in P3 than in P2. As shown in Figure 4.10, the run-time of the *decode-block* accelerator in policy P3, which is a parent accelerator in JPEG, is increased by 4.6% due to the large number of calls to the two child accelerators. In comparison, policy P1 has the lowest impact on the execution length because in this policy an entire accelerator, including all the sub-accelerators of this accelerator, is considered one power gating unit; the sub-accelerators have no separate power gating control.

4.4 Impact of Input Patterns on Static Power-Gating Decisions

As discussed in Section 4.2.2.4, a single set of input vectors is used when constructing the static power gating schedule. When the application is run, it may experience different input patterns than those assumed during scheduling. This may cause changes in the duration of idle peri-

ods of accelerators and sub-accelerators. This may mean that power gating decisions made during scheduling are sub-optimal; if the idle period of an accelerator is much smaller than predicted, it may be that more energy is wasted turning off and on the region than is saved while power-gated. If the idle period of an accelerator is longer than predicted, it may be that the scheduler decides that power gating is not worthwhile, when in actuality, it would have been. To investigate this concern, we vary the input vectors ten thousand times and observe the impact on the number and duration of idle periods in an accelerator. In particular, we are interested in whether the change in input can reduce the idle period duration below the energy break-even cycles leading to a sub-optimal decision.

We performed the experiment across nine accelerators in three of the CHStone benchmarks circuits. For each accelerator, we varied the input pattern by randomly selecting values for each input independently. In doing so, the allowable range of values for each input was determined by examining the program carefully. Ten thousand input sets are generated, and for each, the total number and duration of idle periods were recorded.

We found that in eight (out of nine) accelerators, there was no variation in the number of idle periods and the duration of each idle period remained the same. As a result, for these accelerators, the static power gating scheduler makes the optimal decisions.

The ninth accelerator, *Get_Motion_Code (GMC)* in the Motion benchmark, exhibited variation in the duration of idle periods. This particular accelerator has three idle periods and the static power gating schedule determines that power gating is appropriate for all of them. Changing the input vector ten thousand times reveals the variation as illustrated in the histograms of Figure 4.11. The X-axis of each histogram represents the length of idle period expressed in number of clock cycles and the Y-axis is the number of input vectors that led to that number of idle cycles. The first idle period does not show any variation in the idle period length across all the ten thousand input variations. However, the second and third idle periods do show variation. In both cases, the periods were either 5 cycles long or 100 cycles long, depending on one of the input signals. For period 2, about 40% of input samples led to an idle period of 5 cycles. In this accelerator, the energy break-even point is exactly 5 cycles. As a result, the static scheduler always assumes that the accelerator should sleep for this idle period. If the period happens to be 5 cycles, the overall cost and benefit of power gating are equal, but if the idle period is 100 cycles, significant power savings are obtained by power gating. In either case, however, the decision to power gate during this idle period is optimal.

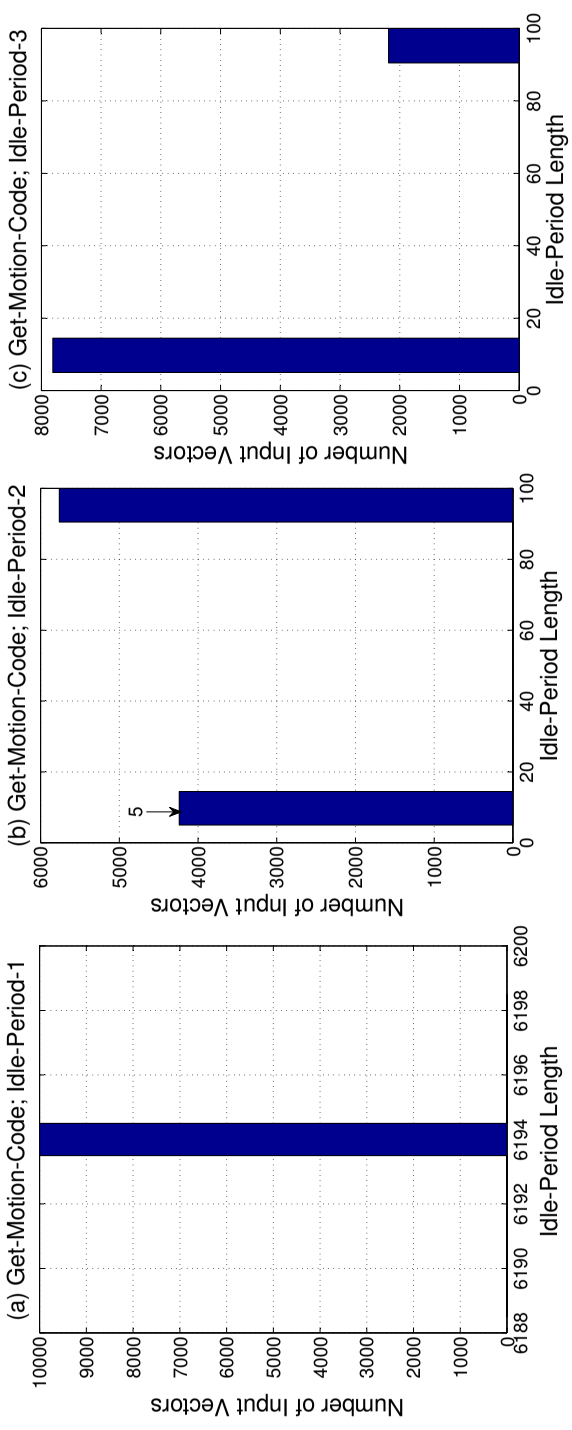


Figure 4.11: Histograms showing the Variation in the Idle-Period Duration of Get-Motion-Code Accelerator (a) Idle-Period-1 (b) Idle-Period-2 (c) Idle-Period-3

This situation is the same for the third idle period, in which 20% of input vectors lead to long idle periods. In general, it is possible that the minimum idle period duration goes below energy break-even point, however, in none of our experiments did this occur.

4.5 Summary

Dynamic power gating has been shown to reduce the FPGA static leakage power significantly. This chapter demonstrates the potential of hierarchical power gating to save static leakage power. Unlike the approach taken in Chapter 3, which considers turning off entire accelerators when they are not required, the hierarchical power gating technique is more fine-grained, in that it allows turning off a portion of an accelerator when other parts of an accelerator are running.

We show that fine-grained control over the sub-accelerators allow them to sleep when only the parent context is required. Similarly, power gating the parent accelerator when the child is running creates more power saving opportunities but increases the program run time. Results on CHStone benchmarks show that hierarchical power-gating can save 8%–25% of static energy when the parent and descendant accelerators are power-gated independently such that the parent accelerator is power-gated while the sub-accelerator runs.

Reducing the granularity, however, reduces the duration of idle periods. We show that the compiler-assisted power gating decisions remain optimal as long as the idle period experienced at run-time is greater than the break-even point. The static power-gating predictions can be greatly improved if the application workload is known at compile-time. Adapting the power-gating decisions with varying workload at run-time, however, would require a dynamic power-gating predictor which is an interesting area for future investigation.

Chapter 5

Model-Based Performance Estimation of Dynamic Partial Reconfigurable FPGAs

5.1 Context

Dynamic Partial Reconfiguration (DPR) is the ability to time-multiplex the FPGA resources at run-time. This enables the use of a smaller FPGA device which directly reduces the per unit cost and lowers the power consumption. DPR has the benefit of reducing static as well as dynamic power. The accelerators in an FPGA design may only operate for a brief burst of activities and remain idle otherwise; whenever an accelerator is in idle state, its design partition can be reconfigured with a blank bitstream - putting it in low dynamic power mode. Additionally, the ability to run parts of the design independently reduces the device density and requires a smaller device to implement a design which reduces the static power dissipation.

Dynamic partial reconfiguration is achieved by writing partial configuration bit-streams into an FPGA's configuration memory. The partial bitstreams are typically stored in an off-chip memory repository and are loaded into FPGA as needed. The primary limitation of DPR is the overhead associated with the reconfiguration process and its impact on the overall system performance. This PR overhead can be significant for fine-grained architectures for which programming bit-streams are long.

Current design practices allow PR overhead to be measured only after the system has already been built thus limiting the number of potential architectures that can be explored due to time and cost constraints. Predicting reconfiguration performance early is difficult, in part, due to non-deterministic factors such as the sharing of the bus structures with traffic not related to the reconfiguration process. This type of traffic, which we

call *non-PR traffic*, is typically due to memory accesses initiated by on-chip processors or other peripherals (IPs) attached to the bus.

In this chapter, we describe a method of modeling the reconfiguration process using well-established tools from Queueing Theory. By modeling the reconfiguration datapath using Queueing Theory, performance measures can be estimated early in the design cycle for a wide-variety of architectures with non-deterministic elements. This approach has many advantages over current approaches in which hardware measurements are made after the system has been built. Reconfiguration performance is heavily dependent on the detailed characteristics of the datapath and on the particular workload imposed on the system during measurement and thus can only be used to make projections for systems similar to that used for initial measurements. A flexible modeling method that works for a wide-range of DPR architectures allows us to explore a large design space early in the design flow ultimately leading to a better implementation.

A set of guidelines is proposed for mapping any reconfiguration datapath to a multi-class queueing network such that the various phases of the reconfiguration traffic (PR traffic) and non-PR traffic can be modeled simultaneously using multiple classes of traffic. Thus, each class can be assigned different queueing and routing properties, allowing for the correct modeling of shared bus resources. Once defined, this model can then be used to generate performance estimates such as the reconfiguration throughput, resource utilization and memory requirements. This can be accomplished by solving the network using either an analytic or simulation-based approach, each of which has its advantages and disadvantages.

Performance estimates generated from the model can be used in the design process in several ways: bottlenecks can be identified; performance trends can be generated to relate system performance to hardware parameters; and the effects of non-PR traffic on PR traffic and vice-versa can be quantified. The advantage of using queueing networks over other modeling techniques is that Queuing Theory has a wealth of analytic and simulation-based approaches for solving a wide variety of modeling features.

The remainder of the chapter is organized as follows: Section 2.5 provides a brief review of Queueing Theory concepts which are important to understand the material presented in rest of the sections of this chapter. Section 5.2 discusses the DPR features to be modeled and describes how Queueing Theory can be used to model reconfiguration datapaths. It also provides schemes for mapping the DPR datapath to Queueing Network model, for both simulation and analytical performance analysis. Finally, Section 5.3 provides two case studies to demonstrate the effectiveness of the proposed

modeling approach.

5.2 Modeling DPR with Queueing Networks

In this section, we describe our approach to model the reconfiguration datapath using Queueing Networks. First, in Section 5.2.1, we list the datapath features we model thus defining the inclusivity of the model. Second, in Section 5.2.3, we outline the principles from Queuing Theory that are used in the proposed modeling approach. Third, in Sections 5.2.4 and 5.2.5, we provide a set of guidelines for mapping reconfiguration datapaths to queueing networks.

5.2.1 What Needs to be Modeled in a DPR Datapath?

We consider a generalized datapath as shown in Figure 5.1 in which a partial reconfiguration consists of the transfer of partial bit-streams from external memory to FPGA configuration memory through a series of zero or more intermediate memories. Transfers between memories can occur over dedicated interconnect or over buses shared with other traffic not related to partial reconfiguration data, which we call as non-PR traffic. We refer to the transfer between any two intermediate memories as a *phase*.

Assuming this generalized datapath, a number of datapath components and features must be modeled. The following is a list of these components and features with a brief description:

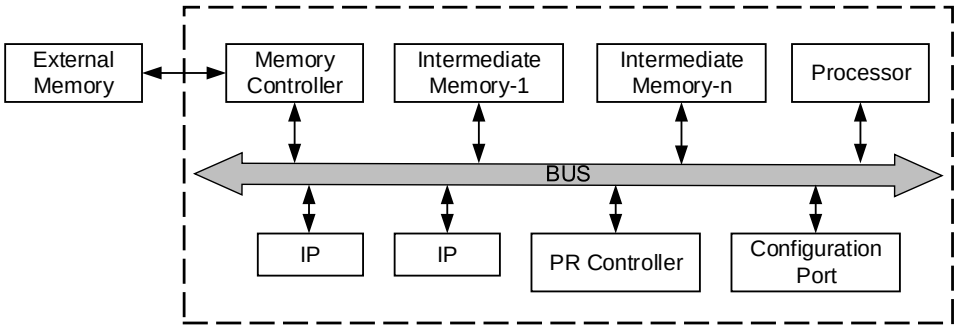


Figure 5.1: A generalized datapath for the dynamic partial reconfiguration of an FPGA-based system

1. **Architecture Components:**

   (a) *Processor:* In a typical DPR system, software running on a processor core initiates the reconfiguration process. The processor may be involved in one more phases of the reconfiguration by controlling the transfer of bit-stream data on a word-by-word basis or via a DMA transfer. If, at any point, control is handed over to a custom reconfiguration controller, the controller ends the reconfiguration by informing the processor with an interrupt.

   (b) *Bus:* Bus is a shared resource that is used for inter-chip communications to perform dynamic partial reconfigurations and by other peripherals for inter-chip communications. From the perspective of the reconfiguration process, the processor's use of the bus is a source of non-determinism. Reconfiguration time is directly affected by bus access patterns initiated by the processor.

   (c) *Memory:* Memories are broadly divided into off-chip and on-chip memories. PR traffic (partial bitstream) is typically stored in off-chip memory, the type of which varies from compact flash (CF) card to DDR memory. On-chip memory in the form of block RAM (BRAM) is used to cache PR traffic before it is transferred to configuration memory.

   (d) *Configuration Port:* The configuration port acts as a gateway through which partial bitstreams are transferred to FPGA configuration memory. The width and operating frequency of the port varies from vendor to vendor and from device to device.

2. **Operating Features:**

   (a) *Reconfiguration Phases:* The reconfiguration process typically has more than one phase. We define a phase as a transfer of bitstream data across a dedicated or shared routing resource ending at a memory. Phases can be sequential such that an entire block of data must complete a phase before the subsequent phase begins, or phases can be pipelined such that a phase can simultaneously operate on the same block of data as the previous phase. This is accomplished using a FIFO.

   (b) *Size of Transfers:* PR traffic through the bus can either be transferred word-by-word or in burst mode.

(c) *Bus Arbitration:* Bus arbitration should be modeled to enforce fairness in the use of a shared bus. Typically, a round-robin schedule is used.

3. **Non-determinism:**

(a) *Non-PR Traffic:* There can be random events in the system such as traffic coming from different Intellectual Property (IP) blocks with non-deterministic start and end times. This traffic competes for shared resources such as the bus with PR traffic.

(b) *Configuration Port:* The configuration port for Xilinx devices is known as ICAP port. The transfers across the ICAP can only be initiated when *ICAP Busy* signal is not active. The behavior of this signal is stochastic in nature.

5.2.2 What Do We Want to Estimate from Queueing Model?

The most important performance metric for a reconfiguration datapath is the *reconfiguration time* which is generally considered to be the time needed to transfer a partial bitstream from off-chip memory to the internal FPGA configuration memory. In the early stages of system development, it is not only important to predict the reconfiguration time of the system but also its variability, especially for safety-critical systems. The primary source of variability comes from non-deterministic components involved in the reconfiguration process. Examples of non-deterministic components includes the ICAP port and buses shared with non-deterministic non-PR traffic.

In addition to reconfiguration time, another important metric includes the *reconfiguration throughput* of a particular stage, or phase, of the datapath. The inverse of this is the estimated time spent in a phase - the sum of all phases corresponds to the time spent per byte to transfer a bitstream from external memory to configuration memory. *Memory utilization* is also important because it directly corresponds to the expected size requirements of memory. Further, utilization of various hardware components can be useful for identifying the bottlenecks of the system.

There are several factors that influence the aforementioned metrics in a hardware implementation. These include the speed and type of external memory, the speed of the memory controller, the type of ICAP controller and the way it is interfaced with the bus. Depending on these choices, reconfiguration time can vary significantly across implementations.

5.2.3 Application of Queueing Networks to DPR Modeling

In order to model different types of traffic in the DPR datapath, as explained in Section 5.2.1, the BCMP network Type-1 node, as discussed in Section 2.5.2.2, has to support different queue service time per customer class. This feature is needed to model the various types of traffic passing through a shared resource, where each arriving customer has different service times depending on its class. Further, to accurately model bus level transfers and different phases of the system a fork-join pair is required. The purpose of fork nodes is to divide incoming queueing customers into one or more sub-customers. All sub-customers generated by a fork must be routed downstream into a corresponding join node. Therefore, all sub-customers must wait until all "sibling" sub-customers have arrived, after which they recombine into the original customer.

To correctly model the datapath, two of the above-mentioned features are required which are incompatible with the BCMP theorem. Thus, they no longer have a product-form expression for the joint-probability and are thus not easily solved analytically. The first incompatibility is that fork-join nodes are used to model the word-by-word transfers of data across the bus. The second is in the use of non-compliant Type-1 queueing nodes for which each arriving customer has different service times depending on its class.

One solution for solving a non-product-form network is to use approximation methods. The disadvantage of this is that it cannot be used to model multi-class networks and it is very difficult to find approximations methods that work with both fork-join nodes and non-compliant Type-1 nodes simultaneously. A second approach is to reduce the accuracy of the model through the introduction of assumptions such that the network becomes simpler and therefore BCMP-compliant which can be solved analytically. The third approach is to use a simulation-based approach and to collect performance statistics rather than to calculate them analytically.

The primary advantage of simulation-based solutions is that all queueing features are supported. This includes fork-join pairs, phase barriers, and finite-capacity regions. Additional benefits to a simulation-based approach include the availability of transient analysis and what-if analysis techniques. The disadvantage to simulation-based network solvers is poor run-time complexity; however, this is not an issue due to the relatively small sizes of the networks needed for modeling DPR datapaths. While simulations can provide a solution in cases where an analytical solution is not possible, there are situations in which an analytical solution is required. One such situation is when an analytic solution is required in order to be incorporated into

high-level design space exploration tools for quick estimates.

We provide mapping guidelines for both simulation- and analytical-based approaches. The choice of modeling scheme depends on the complexity of the network and intended use of the result. Section 5.2.4 presents the mapping scheme for simulation-based solutions followed by the mapping scheme for analytical-based solutions in Section 5.2.5.

5.2.4 Mapping Scheme for Simulation-based Solutions

For simulation-based solutions, the DPR datapath is mapped to queueing network using various queueing primitives, as shown in Figure 5.2. Each primitive models a different aspect of the datapath.

Buses and memories used for the reconfiguration process are modeled with single-node queues. The partial bitstream is represented by queueing customers that pass through a sequence of single-node queues that together represent the DPR datapath. The partial bitstream is divided into smaller byte or word sized blocks (i.e. sub-customers) depending on the nature of the transfer. Single-node queues are connected with directed edges that dictate the flow of bitstream data through hardware resources thus defining the *reconfiguration datapath*. Customers are categorized into classes to model different types of traffic whether they be different phases of the PR process or traffic not directly related to the PR process such as the fetching of instructions by the processor core. By using customer classes, we can assign different service rates and different routing policies for each type of traffic passing through shared resources.

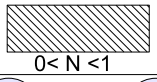
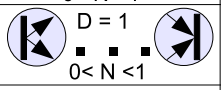
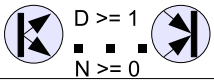
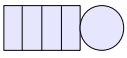
PR Feature to be Modeled	Mapped To Queueing Primitive	
	Component	Name
Arbitration for Shared Resource		Finite-Capacity Region
Phase Barrier		Fork-Join
Bus Transfer		Fork-Join
Non-deterministic resource (Bus, ICAP, Traffic Source)		Single-Node Queue

Figure 5.2: Mapping of PR features to queueing primitives

5.2.4.1 Separation of Concerns

To capture the behavior of a DPR datapath for quantitative performance measures, aspects of the overall reconfigurable system not related to the reconfiguration process are not included in the queueing model. The first implication of this is the time period considered when modeling a PR. We model the beginning of a PR process as the point at which the bitstream data is first transferred out of external memory to the FPGA. All other activities driving this process such as the initiation of the transfer by the processor are not considered. We model the end of the PR process as the point at which the last byte of the partial bitstream has been transferred to configuration memory. The queueing network models only the transfer of bitstream data between this start and end point. *Closed networks* are used as a convenient way to model the assumption that new transfers start only when old transfers are finished.

All aspects of the architecture not directly relevant to the PR process are removed from the model. The only exception to this is the inclusion of all non-PR traffic that shares bus resources with PR traffic. The set of generalized rules for mapping any reconfigurable datapath to queueing network is described below.

5.2.4.2 Modeling Non-deterministic Resources

Components of the PR process that are non-deterministic in nature are modeled as *single-node queues* as shown in the mapping table provided in Figure 5.2. Depending on the nature of the hardware components of the datapath being modeled, this can include shared buses, the configuration port (i.e. ICAP port), the source nodes for PR and non-PR traffic, memories, and custom IP cores. We assume that the arrival of PR and non-PR traffic is non-deterministic. The ICAP port is modeled as a queue with exponential service distribution and with an average service time corresponding to the measured speed of the ICAP in our experiments.

With respect to the shared bus, the service time is different for each class of customer (i.e. each type of PR and/or non-PR traffic) and is dependent on the number of words per transfer and the speeds of the devices involved in the transfer. Arrivals to the bus are dictated by source queues used for each type of traffic; in case of PR traffic, this is provided by the queue representing external memory.

Although other distributions are possible, all queues are modeled in this chapter using an exponential service time distribution which is the most

commonly used in Queueing Theory. It corresponds to a process by which the arrival times (service times) are uniformly random and the number of arrivals that occur per interval of equal width is identically distributed.

5.2.4.3 Modeling Bus Traffic

The transfer of a block of the configuration bitstream from one memory to another corresponds to a *PR phase*. As shown in Figure 5.2, bus transfers are modeled using fork-join pairs with a forking degree of $D \geq 1$ and a capacity $N \geq 0$. D represents the degree to which a block is divided for the transfer. If, for example, a 2048-byte block is to be transferred over a bus in 32-bit words, a 512-way fork is needed. After passing through the bus, the 32-bit words are reassembled to form the 2048-byte block via a corresponding join node. By breaking blocks into words as such, we can accurately model the interlaced flow of traffic through the bus.

To model bus arbitration, a *finite-capacity region (FCR)* is used as listed in Figure 5.2. The number of customers, N, allowed into an FCR is limited to a maximum, $1 \leq N \leq N_{max}$. When a queue representing a shared bus is placed in the FCR, customers of different classes are allowed into the queue by a round-robin arbitration. This is required to prevent a large number of customers spawned from a fork from filling the queue thus preventing customers of a different class from having interleaved access to the bus.

For block transfers, it may be required that a new PR phase not start until the entire block has completed the previous phase. Join nodes inherently solve this problem by enforcing a synchronization mechanism in which all sub-customers generated from a single fork must arrive at the join node before the join is completed. If this behavior is not needed, then the join node must be placed after the ICAP node when the complete transfer is finished.

Phases may be composed of more than one "sub-phase". For some datapaths, it may be required that the number of blocks that can be "processed" by a series of phases is limited. To model this, the *phase barrier* shown in Figure 5.2 is used which is a fork-join pair with a forking degree of $D = 1$ and a finite capacity of $0 \leq N \leq 1$.

5.2.4.4 Modeling Pipelining

Phase pipelining is modeled by populating the queueing network with more than one customer such that each customer represents a portion of the bitstream. The number of customers in the system represents the number of

bitstream blocks that can be processed by different phases of the pipeline at the same time. The maximum number of customers in the system is limited by the maximum number of phases in the pipeline.

5.2.5 Mapping Scheme for Analytic-based Solutions

The philosophy behind the analytic solution method is to represent the system using mathematical equation or set of equations from which certain measures can be deduced. The solution of these equations can be a *closed-form solution* or can be obtained through algorithms from numerical analysis techniques. The closed-form solution is bounded by basic operations which make the solution computationally less complex and hence faster to calculate. In many situations where it is not possible to obtain a closed-form solution formula, the solution is often approximated using numerical methods which usually have higher computational complexity.

In the work discussed in this chapter, we focus on analytical models with product-form solutions. To achieve this, we model the reconfigurable system using a queueing network which can be solved using the BCMP theorem, as explained in Section 2.5.2.2, thus yielding a product-form solution.

In Section 5.2.4, we introduced a generalized mapping scheme for simulation-based solutions. In doing so, we introduced the use of several modeling features such as fork-join pairs in order to accurately model bus traffic. Some of these queueing features are not compatible with product-form theorems such as BCMP and consequently approximation methods must be used. If, on the other hand, a product-form solution is required then the only option is to simplify the network by removing features incompatible with BCMP. As a result, the model loses the ability to capture the real-world behavior of the system and hence a difference might exist between the simulation-based and analytical-based results.

To fit the proposed modeling scheme to BCMP, we make the following simplifications:

1. We eliminate the use of fork-join nodes. Thus, we no longer model word-level bus interactions by splitting partial bitstreams into words. Instead, we model the transfer of large blocks across the bus by increasing their queue service time. This can potentially skew the results when one type of traffic uses burst mode transfers. Another implication of removing fork-join pairs is that the interaction between sequential phases cannot be modeled well. In addition to fork-join pairs being useful to break reconfiguration bitstreams into smaller parts, it is also

useful for synchronizing packets. All sub-packets created by a fork get blocked at the corresponding join until all sibling sub-packets arrive; this provides a useful way of differentiating between two sequential phases such that a set of packets are transferred to memory in the first phase and, upon completion, the subsequent phase begins.

2. We eliminate the use of finite-capacity constructs which are used to limit the number of queueing customers allowed within a set of one or more queues. These regions are used for our simulation-based approach in two ways. First, we use finite-capacity fork-join pairs to limit the number of packets that can enter a sequence of phases and to then synchronize them. Second, we limit the number of packets allowed to queue at a bus thus enforcing bus arbitration.

3. For BCMP type-1 nodes, all classes sharing a queue must have the same service time. The implication is that we will not be able to assign different service times for different traffic types on a shared resource such as a bus. In order to assign different service times to PR and non-PR traffic at a bus node, we assume that bus nodes use a Processor Sharing service strategy (i.e BCMP type-2 node).

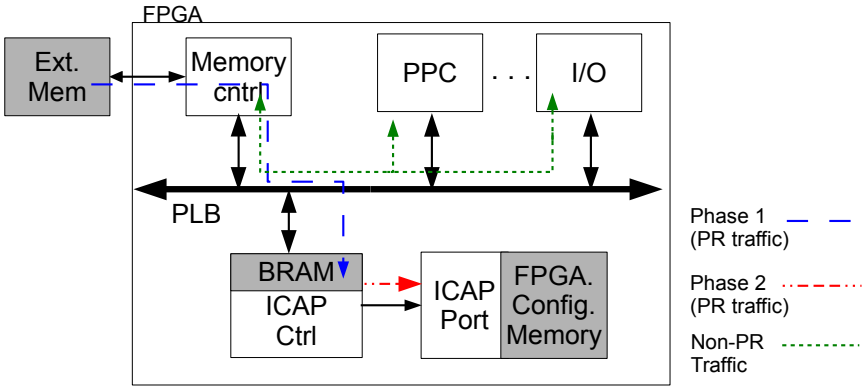
5.3 Experimental Results

5.3.1 Case-Study 1: Two-Phase Datapath with Custom Reconfiguration Controller

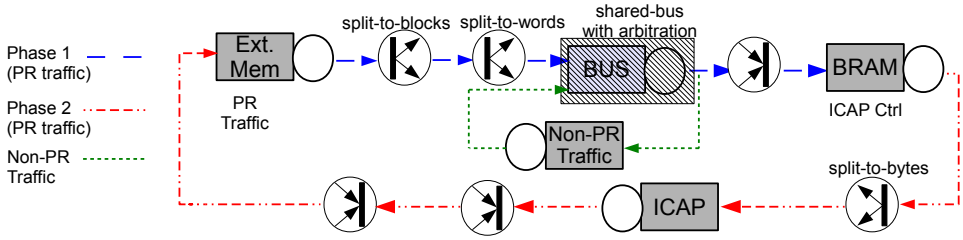
The DPR datapath of our first case study is shown in Figure 5.3(a). The architecture used in this example is typically used to lower reconfiguration overhead through the use of a custom reconfiguration-port controller implemented as an IP. The partial bitstream is fetched from external memory and is buffered in a FIFO from where it is fed into FPGA configuration memory, whenever the ICAP port is not busy. The two phases of the datapath are pipelined thus further improving throughput. Below we explain the experimental setup and results achieved through actual measurements and model-based estimates.

5.3.1.1 Experimental Setup

In the example system in Figure 5.3(a), the partial bitstream is transferred from external DDR memory to configuration memory in two distinct



(a) Case-Study 1: Dynamic Partial Reconfigurable Datapath.



(b) Queueing Network Model of the above shown DPR Datapath.

Figure 5.3: Case-Study 1: Two-Phase Datapath with Custom Reconfiguration Controller.

but overlapped phases. The only involvement of the PowerPC (PPC) processor core during these phases is to initiate the transfer, while the rest of the transfer is handled by the custom ICAP port controller. We call this system a two-phase datapath with custom ICAP port controller.

The two phases of the reconfiguration process for this platform are as follows:

- **First phase:** a 128-byte block is transferred from DDR Memory to BRAM via the PLB in burst mode.
- **Second phase:** data is transferred byte-by-byte from BRAM to configuration memory via the ICAP port in 8-bit packets.

Phases 1 and 2 are pipelined such that when the bitstream is being written into the FPGA configuration memory, new bitstream data is fetched from external memory and placed after the previously read data in BRAM. This process is controlled by a custom ICAP controller which transfers partial bitstream data from external DDR memory to the FPGAs configuration memory using DMA transfers. The system is implemented on a Xilinx Virtex-II Pro device thus allowing us to validate the estimated results with measured results.

5.3.1.2 Mapping to a Queueing Network

In order to obtain estimates of the various performance metrics for the example datapath in Figure 5.3(a), we map it to a queueing network model, shown in Figure 5.3(b), using the guidelines provided in Section 5.2.4.

There are five queues in the queueing network representing the shared PLB bus, the ICAP port, BRAM, and source nodes for both PR and non-PR traffic. The source node for PR traffic represents DDR memory in hardware and its service rate models the initiation of transfers as dictated by the PR controller. The source node for non-PR traffic models sources for all traffic not related to PR such as memory reads and writes involving the PowerPC, PowerPC memory, and other IPs attached to the bus. Both phases are bounded by memory-transfer primitives (i.e. fork-join nodes) to convert 128-byte blocks to words and words to bytes, respectively.

Customers are divided into three classes namely *Phase-1* customers, *Phase-2* customers and *Non-PR traffic* customers. Customers belonging to phase-1 and phase-2 represent the PR traffic. Phase-1 customers originate from the external memory node and reach the BRAM via the bus. After BRAM, phase-1 customers switch their class to phase-2 and are finally transferred to the ICAP. The non-PR traffic customers originate from the non-PR source node and move towards the bus. The bus is shared between PR and non-PR traffic in an interlaced fashion thus modeling the impact of non-PR traffic on PR traffic.

It is important to note that the Non-PR traffic can be generated by different hardware nodes in the system i.e the processor and/or several IPs attached to the bus. We represent all the possible sources of Non-PR traffic with only one Non-PR source node. This simplifies the model but still captures the impact of Non-PR traffic on PR-traffic. Further, one always has the flexibility to add as many Non-PR nodes as needed to correctly model the Non-PR traffic workload characteristics.

Table 5.1 provides queue service time for each node in the network model,

Table 5.1: Different traffic types present at different hardware (H/W) nodes and the corresponding queue service times.

Node Traffic Type	**H/W Node Specs.**	**Service Time** (usec)
PR (Phase-1) at Bus	Bus Freq/Width: 100 Mhz/64 bits Mem Cycles: 30 Bus cycles/128-byte	0.3
Non-PR (Phase-2) at Bus	Bus Freq/Width: 100 Mhz/64 bits Processing: 1 bus cycle/8-bytes	0.01
PR (Phase-1) at ICAP	ICAP Freq/Width: 100 Mhz/8 bits No of cycles/byte: 1.099	0.01099
Non-PR (Phase-2) at PPC	PPC Freq: 300 Mhz 30% memory operations 0.1% Miss rate	0.0256

which is derived from the hardware specifications used in the actual DPR system. The service time of external memory is set to zero as the station acts as a source that immediately provides bitstream blocks once the previous block has exited the system. The service time for PR traffic at the bus node is 0.3μ, which is calculated from the 100Mhz operating frequency of the PLB bus and assuming that it takes 30 bus clock cycles to read 16 (64-bit)words from DDR Memory. The non-PR traffic for this system is generated by the PowerPC and takes one bus cycle per transfer. The PowerPC operates at 300 Mhz, has 30% memory operations, and has a 10% miss rate for both instruction and data caches. Thus, the service rate for the non-PR queue is 0.0256μ sec. The ICAP is 8-bits in width and operates at a frequency of $100MHz$. Ideally, the partial bitstream should be transferred at $100MB/s$ but due to the non-deterministic nature of the ICAP-BUSY signal, the transfer rate is reduced to $91.6MB/s$. Thus, the service time for the ICAP queue is 0.0109μ sec.

5.3.1.3 Model-Based Simulation Results

We simulated the queueing network using open source Java Modeling Tool (JMT) Queueing Network Simulator [BCS09]. In particular, JMT was modified to include multi-class switching. Simulations were conducted on a 2.2GHz Intel Core2 Duo T7500 with a typical run-time of 2 minutes for each network simulation.

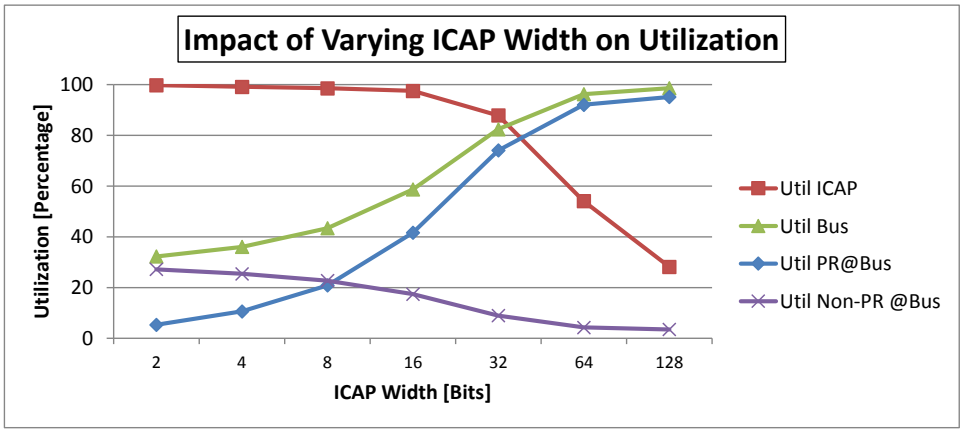
In our experiments, the ICAP width and external memory speed were varied to represent variety of candidate architectures. Performance measures for utilization and throughput metrics are provided.

Impact of Varying ICAP Width on Utilization and Throughput: Results were generated for the expected utilization and throughput of queues in the network and are shown in Figure 5.4. The plot in Figure 5.4(a), which depicts the impact of varying ICAP width on utilization of various queues, the X-axis represents the ICAP width expressed in bits and the Y-axis represents the queue utilization expressed in percentage. The plot in Figure 5.4(b) depicts the impact of varying ICAP width on throughput of partial reconfiguration and non partial reconfiguration traffic, denoted as TP PR and TP Non-PR respectively. The X-axis of plot 5.4(b) represents the ICAP width expressed in bits and the y-axis represents the throughput in Bytes/s.

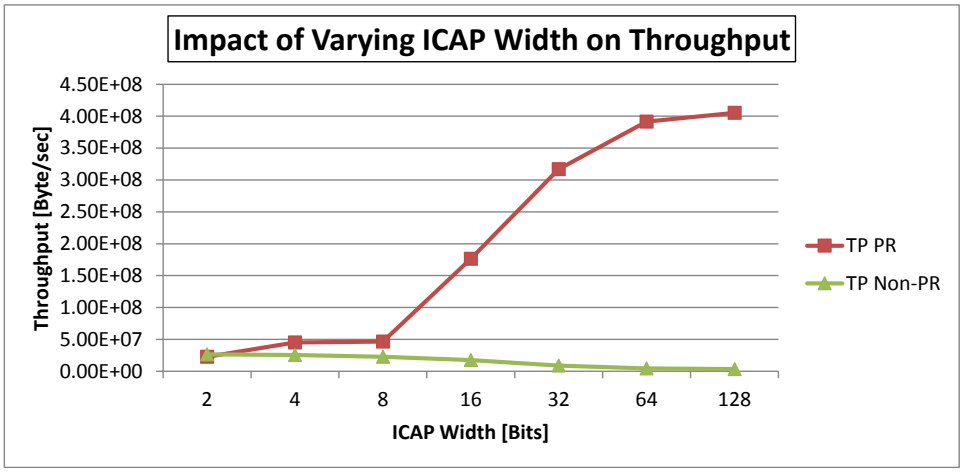
As can be seen in plot 5.4(a), as the size of the ICAP port is increased from 2 bits to 128 bits, the utilization of the ICAP due to PR traffic drops from 99.8% to 28.1%. As the width of the ICAP increases, it can transfer more data per unit time thus reducing its utilization. The reverse trend can be observed for the utilization of the shared bus due to PR traffic which increased from 5.3% to 95.1% over the same range. This demonstrates that improvements in ICAP speed places increased pressure on the bus thus making the bus the new bottleneck, affecting overall throughput. As shown in Figure 5.4(b), overall PR throughput increases with increased ICAP width until 32 bits after which it begins to saturate due to the high utilization of the bus.

For small ICAP widths, utilization of the bus was roughly equal for both PR and non-PR traffic. As PR traffic was increased due to larger ICAP widths, PR requests for the bus increased thus reducing the availability of the bus for non-PR traffic. As a consequence, the utilization of the bus due to non-PR traffic reduced from 27.2% to 3.5% and its throughput correspondingly suffered. These results clearly demonstrate the relationship between PR and non-PR traffic in a system with shared resources and supports the need to consider non-PR traffic early when designing such a system.

Impact of Varying External Memory Speed on Utilization and Throughput: Results were generated for the expected utilization and throughput of the ICAP and the bus as a function of external memory



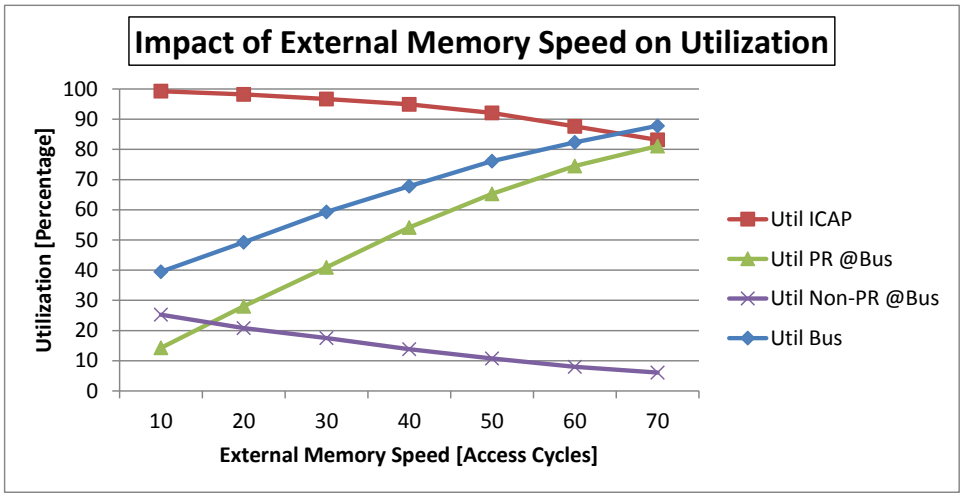
(a) Case-Study 1: Impact of Varying ICAP Width on Utilization.



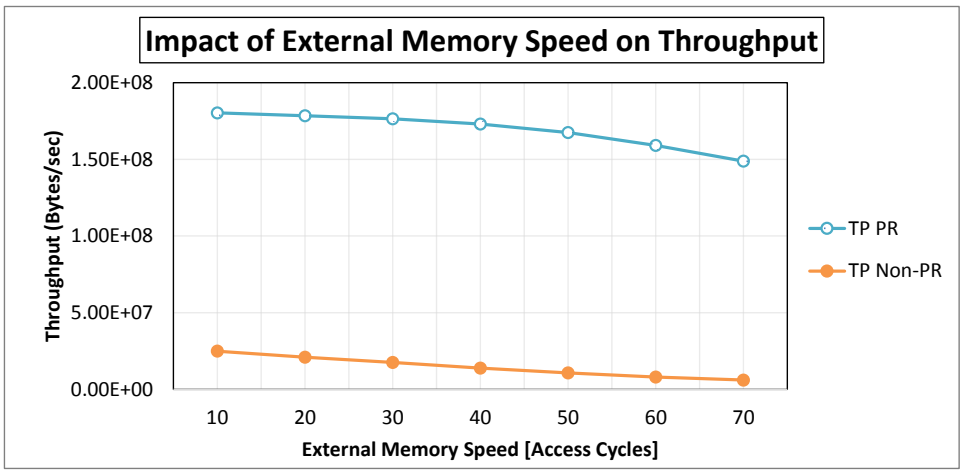
(b) Case-Study 1: Impact of Varying ICAP Width on Throughput.

Figure 5.4: Case-Study 1 Results: Impact of Varying ICAP Width on Utilization and Throughput.

speed as shown in Figure 5.5. The plot in Figure 5.5(a), which depicts the impact of varying external memory speed on utilization of various queues, the X-axis represents the external memory speed expressed in access cycles and the Y-axis represents the queue utilization expressed in percentage.



(a) Case-Study 1: Impact of External Memory Speed on Utilization.



(b) Case-Study 1: Impact of External Memory Speed on Throughput.

Figure 5.5: Case-Study 1 Results: Impact of Varying External Memory Speed on Utilization and Throughput.

The plot in Figure 5.5(b) depicts the impact of varying external memory speed on throughput of partial reconfiguration and non partial reconfigura-

tion traffic, denoted as TP PR and TP Non-PR respectively. The X-axis of plot 5.5(b) represents the memory speed expressed in access cycles and the Y-axis represents the throughput in Bytes/s. Memory speed was varied by changing the number of bus cycles needed to perform a memory read over the PLB. The default number of bus cycles was 30 which corresponds to DDR memory. Experiments were conducted over the range of 10 to 70 cycles representing a broad range of memory speeds for an ICAP width of 32-bits.

As the number of bus cycles was increased, thus representing slower memories, the utilization of the bus due to PR traffic increased from 39.5% to 87.8%, as show in plot 5.5(a). At the same time, the utilization of the bus due to non-PR traffic was reduced. Longer PR transfers due to slower memory resulted in higher utilization of the bus due to PR traffic and thus non-PR had less utilization of the bus. Longer bus transfers also resulted in fewer transfers of PR traffic across the ICAP and thus its utilization went down. Overall, the throughput of both PR and non-PR were negatively affected by slower external memory.

Expected BRAM Requirements The average number of customers (bytes) for the ICAP queue is shown in Figure 5.6 which provides insight into the BRAM needs of the system. As the the size of the ICAP was varied from 2 bits to 128 bits, the average number of customers followed an exponential decay from approximately 3860-bytes to 2-bytes. These results predict that a BRAM size of 512 bytes would be more than sufficient for the

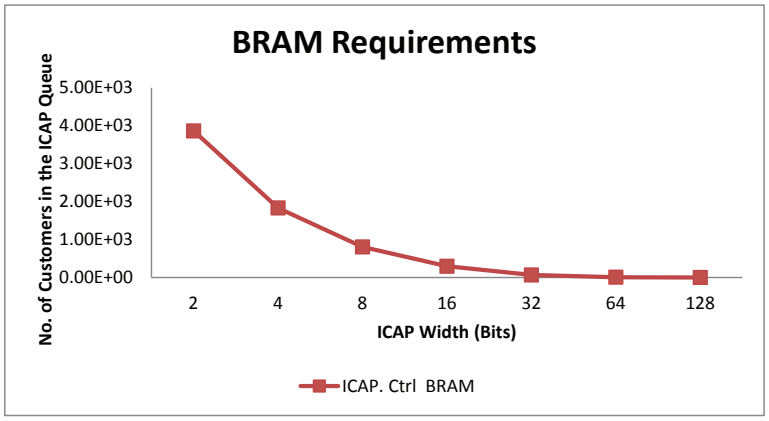


Figure 5.6: The effect of *ICAP width* on the expected BRAM size requirements.

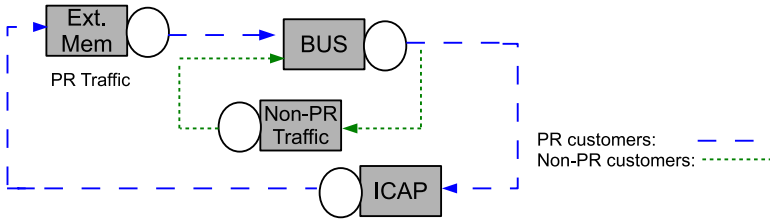


Figure 5.7: Queuing Network model of the example datapath.

8-bit port. Results of this type demonstrate how the proposed model could be used to determine memory requirements of a system before it is built.

5.3.1.4 Model-Based Analytical Results

In this section, we present results for an analytic solution to the datapath shown in Figure 5.4(a). Because this platform is relatively simple, it is well-suited to be force-fit to a BCMP network. This is true because it has only two simple phases for PR traffic. The difference in transfer time of the two phases can be approximated using queue service times.

To illustrate the analytical approach, we map a simplified version of the datapath shown in Figure 5.4(a), following the mapping guidelines provided in Section 5.2.5. The resultant BCMP queueing network is shown in Figure 5.7. This model consists of four nodes and two job classes. One job class is used for PR traffic and the second for non-PR traffic. All queues use FCFS queue service strategies.

This network of queues can be solved analytically using the BCMP theorem, which states that the probability of a particular steady network state, is given by Equation 2.12, which represents the closed-form solution. Upon solving, different performance measures such as throughput and utilization of different nodes in the queueing network can be deduced.

For the network shown in figure 5.7 which is a closed network, the term $d(S)$ in Equation 2.12 equates to one and G represents the normalization constant. The calculation of this normalization constant, G, requires that all states of the network be considered. A more tractable approach makes use of an iterative algorithm that does not need to consider the states of the queueing network when computing the normalization constant. A commonly used algorithm of this type is the *mean value analysis (MVA)* which gives the mean values of the performance measures directly without computing the normalization constant [PKT90].

Impact of ICAP Width on Throughput and Utilization: We apply the MVA algorithm along with Equation 2.12 to determine PR and non-PR traffic throughput and utilization. Figure 5.8 shows the analytical results compared with the simulation based solutions for varying ICAP-width from 8-bits to 128-bits. The solid lines represent the ICAP utilization while the dashed-lines represents the throughput for the (A)nalytical and (S)imulation-based results. It is evident from the graph that the results found through an analytical approach closely match those generated using a simulation-based approach. Thus, the assumptions made for the analytic mapping scheme for our example system did not adversely affect estimation accuracy.

Model Validation In Table 5.2, physical measurements were compared against those estimated using both the simulation-based and analytic-based approaches for a bitstream of size 150 KB. Results showed that both approaches were able to predict throughput with 0.3% error. This result helps to validate that the proposed model provides a reasonable estimate of PR throughput compared to hardware.

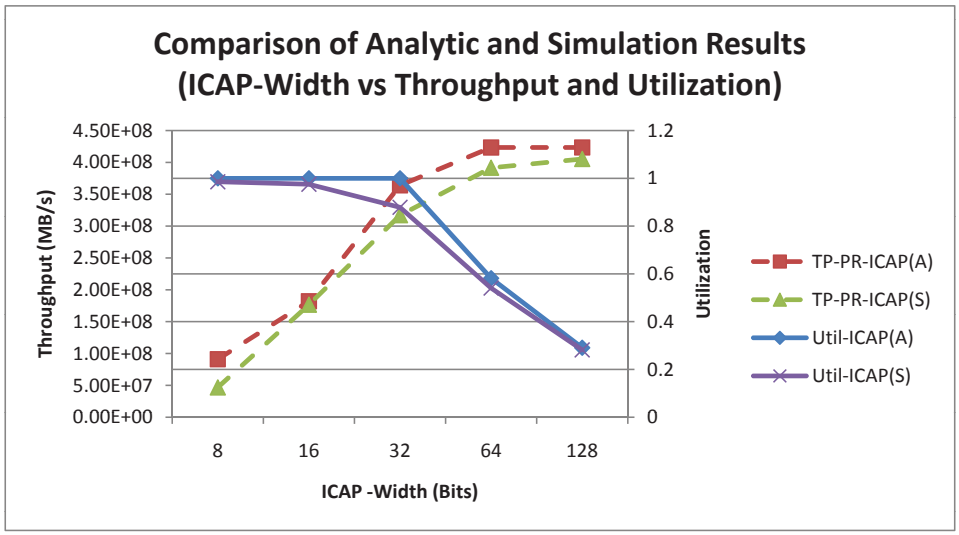


Figure 5.8: A comparison of analytic and simulation-based utilization and throughput results as a function of *ICAP width*.

Table 5.2: Comparison of measured, estimated and analytical results for an ICAP width of 8-bits.

	Measured	Estimated	Analytical
TP-PR (MB/s)	90	89.7	91
RT (ms) (BS Size:150 KB)	1.66	1.67	1.64

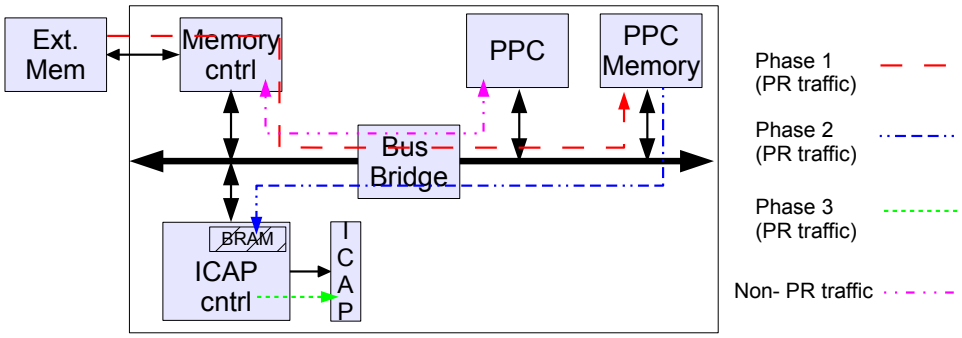
5.3.2 Case-Study 2: Three-Phase Datapath with PowerPC Controller

5.3.2.1 Experimental Setup

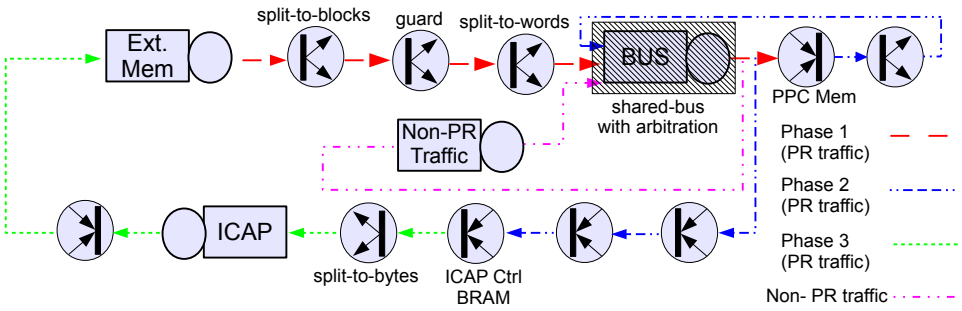
In this example, an FPGA-based system with three-phase partial reconfigurability is modeled as shown in Figure 5.9(a). This system is similar to that used by [PDH11] which is built upon Xilinx's Virtex-II Pro. The reconfiguration process is controlled by the on-chip PowerPC which transfers partial bitstream data from external memory to the FPGAs configuration memory in three phases. In the first phase, a 512-byte block is transfered word-by-word to the PowerPC's memory via the PLB. In the second phase, the 512-byte block is transfered word-by-word from the PowerPC's local memory to the local memory of the ICAP controller via the PLB. In the third phase, byte transfers are made directly to the ICAP port.

Figure 5.9(b) is an illustration of the queueing network constructed from the system in Figure 5.9(a). There are four queues representing the non-deterministically shared PLB, the ICAP port, non-PR PowerPC traffic, and the external memory. All three phases are bounded by fork-join pairs in order to properly divide the 512-blocks into the correct sizes needed for each transfer. PowerPC traffic is modeled by a single queue feeding customers into the shared bus. The PowerPC and ICAP memories can be modeled by join nodes since their services times are assumed to be zero.

Table 5.3 includes system specifications and the calculated queue service times parameters. The operating frequency of PLB bus is 100Mhz. It is assumed that the data from the compact flash (CF) card is transferred to the PowerPC memory one word at a time, with each word 4-bytes long. Assuming that it takes 11 bus clock cycles to read 4-bytes of data from the external CF card, the corresponding service time for PR traffic at bus node is calculated accordingly. After class switching, PR Phase-1 customers switch class to become PR Phase-2 customers and then arrive at the shared



(a) Case-Study 2: Dynamic Partial Reconfigurable Datapath.



(b) Queueing Network Model of the above shown DPR Datapath.

Figure 5.9: Case-Study 2: Three-Phase Datapath with PowerPC Controller.

bus for service. They are then serviced in one bus clock cycle. The traffic generated by the PowerPC is the non-PR traffic which is assumed to take one bus clock cycle to be transfered across the bus. The PowerPC itself is assumed to operate at 300 Mhz with 30% memory operations and a 10% miss rate. The ICAP is assumed to be 8-bits bits in width and to operate at 100Mhz. To account for the stochastic nature of the ICAP-BUSY signal, it is assumed that it takes approximately 2-cycles to transfer one byte. This assumption makes sure that ICAP throughput is less than ideal.

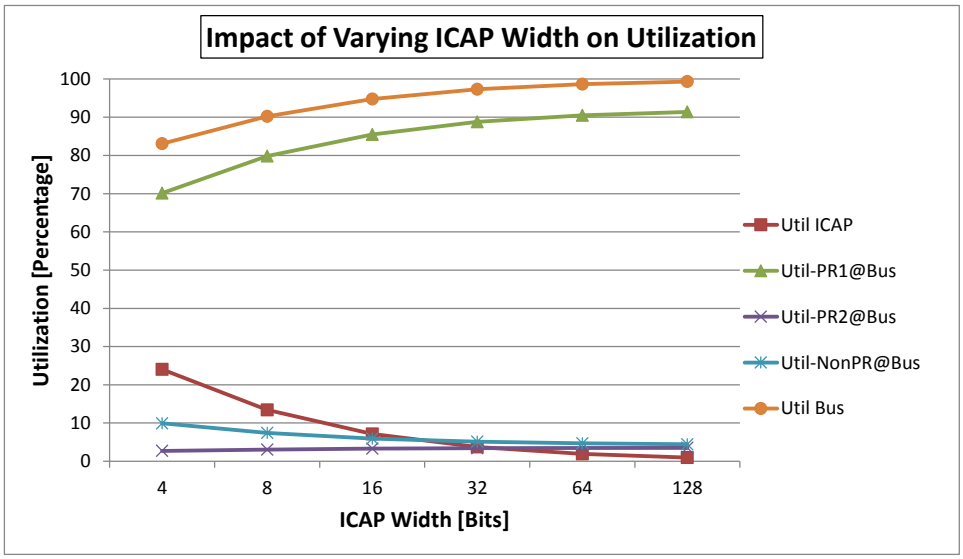
Table 5.3: Different traffic types present at different hardware (H/W) nodes and the corresponding queue service times.

Node Traffic Type	**H/W Node Specs.**	**Service Time** (usec)
PR (Phase-1) at Bus	Bus Freq/Width: 100 Mhz/32-bits Mem Cycles: 26 Bus cycles/4-bytes	0.26
PR (Phase-2) at Bus	Bus Freq/Width: 100 Mhz/32-bits Bus Cycles: 1 Bus cycle/4-bytes	0.01
Non-PR (Phase-2) at Bus	Bus Freq/Width: 100 Mhz/32-bits Processing: 1 bus cycle/8-bytes	0.01
PR (Phase-1) at ICAP	ICAP Freq/Width: 100 Mhz/8-bits No of cycles/byte: 1.099	0.01099
Non-PR (Phase-2) at PPC	PPC Freq: 300 Mhz 30% memory operations 0.1% Miss rate	0.0256

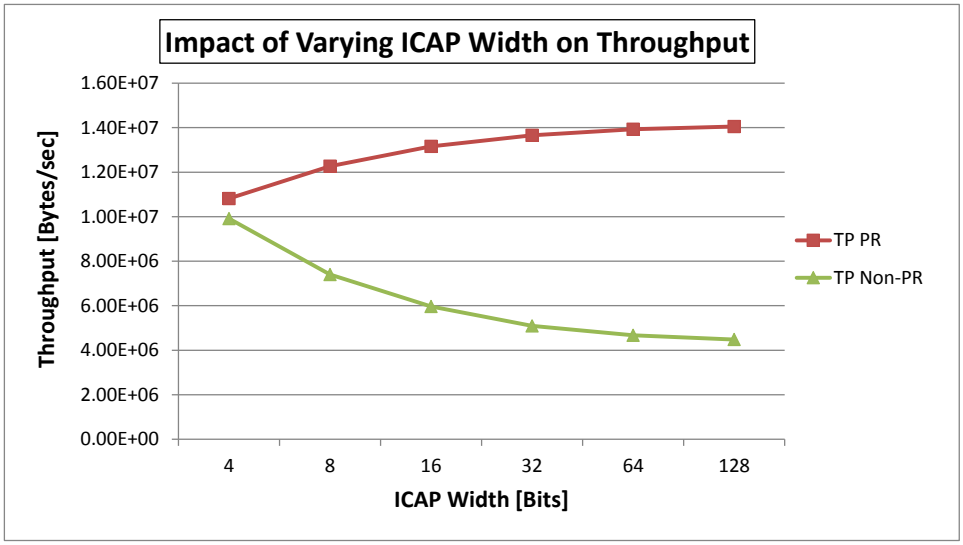
5.3.2.2 Impact of Varying ICAP Width on Utilization and Throughput:

We varied the ICAP-width from 4-bit to 128-bit and generated the expected utilization and throughput of various nodes in the network. The plot in Figure 5.10(a) depicts the impact of varying ICAP width on utilization of various queues, the X-axis represents the ICAP width expressed in bits and the Y-axis represents the queue utilization expressed in percentage. The plot in Figure 5.10(b) depicts the impact of varying ICAP width on throughput of partial reconfiguration and non partial reconfiguration traffic, denoted as TP PR and TP Non-PR respectively. The X-axis of plot 5.10(b) represents the ICAP width expressed in bits and the y-axis represents the throughput in Bytes/s.

As the size of the ICAP port is increased from 4-bit to 128 bit, the utilization of the ICAP due to PR traffic dropped from 24.4% to 0.09%, as shown in Figure 5.10(a). As the width of the ICAP increased, it can transfer more data per unit time thus reducing its utilization. The reverse trend can be observed for the utilization of the shared bus due to PR traffic which increases from 83% to 99% over the same range. Throughout the range of ICAP widths used, the bus is more utilized than the ICAP, making it the bottleneck. Further, increases in ICAP speed puts pressure on the bus by



(a) Case-Study 2: Impact of Varying ICAP Width on Utilization.



(b) Case-Study 2: Impact of Varying ICAP Width on Throughput.

Figure 5.10: Case-Study 2 Results:Impact of Varying ICAP Width on Utilization and Throughput.

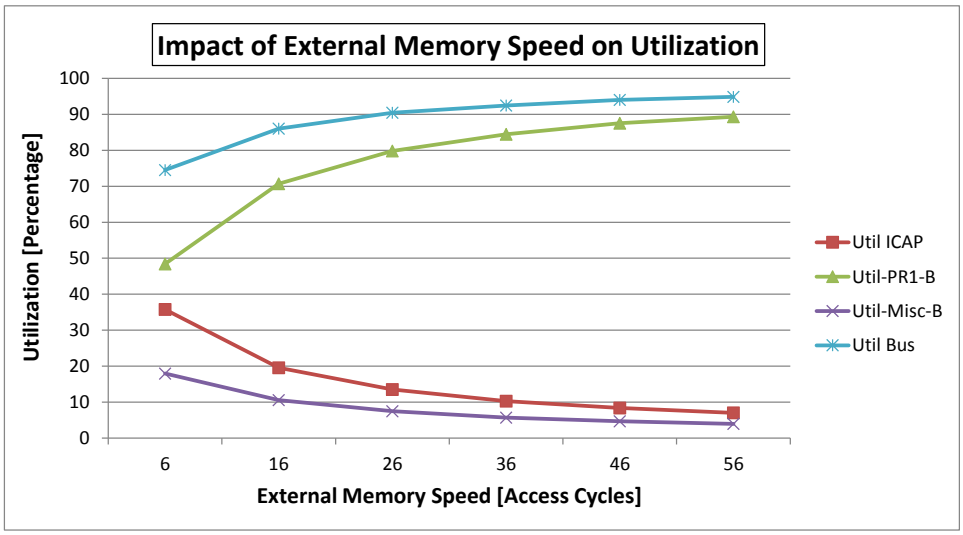
increasing its utilization. As shown in Figure 5.10(b), the overall throughput improves with increased ICAP width until 32 bits after which it begins to saturate due to the high utilization of the bus.

For small ICAP widths, utilization of the bus is roughly equal for both PR and non-PR traffic. As PR traffic is increase due to larger ICAP widths, PR requests for the bus increases thus reducing the availability of the bus for non-PR traffic. As a consequence, the utilization of the bus due to non-PR traffic reduces from 27.2% to 3.5% and it's throughput correspondingly suffers. These results clearly demonstrate the relationship between PR and non-PR traffic in a system with shared resources and supports the need to consider non-PR traffic when designing such a system.

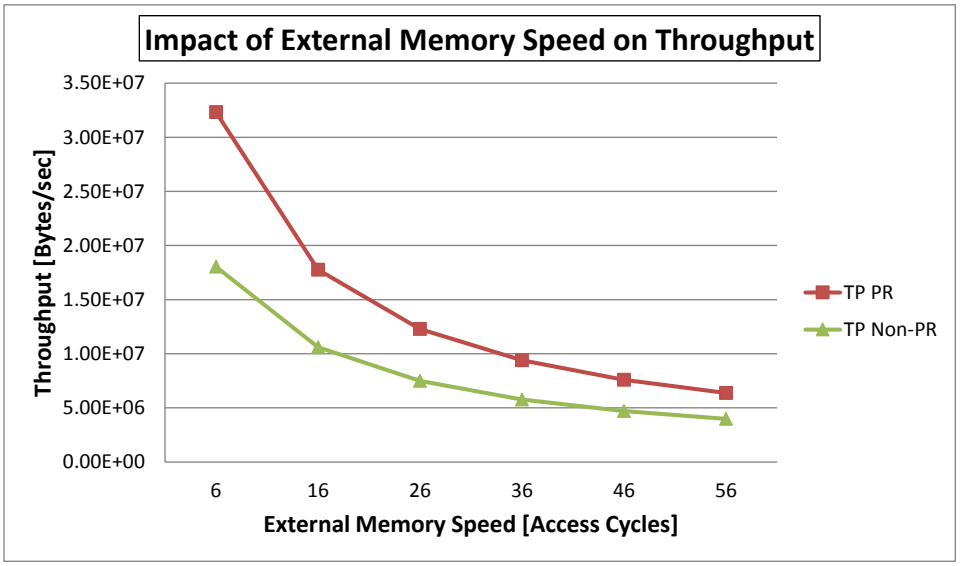
5.3.2.3 Impact of External Memory Speed on Utilization and Throughput:

Results were generated for the expected utilization and throughput of the ICAP and the bus as a function of external memory speed as shown in Figure 5.11. The plot in Figure 5.11(a), which depicts the impact of varying external memory speed on utilization of various queues, the X-axis represents the external memory speed expressed in access cycles and the Y-axis represents the queue utilization expressed in percentage. The plot in Figure 5.11(b) depicts the impact of varying external memory speed on throughput of partial reconfiguration and non partial reconfiguration traffic, denoted as TP PR and TP Non-PR respectively. The X-axis of plot 5.11(b) represents the memory speed expressed in access cycles and the Y-axis represents the throughput in Bytes/s.

Memory speed was varied by changing the number of bus cycles needed to perform a memory read over the PLB. The default number of bus cycles was 26 which corresponds to compact-flash(CF) memory. Experiments were conducted over the range 6 to 56 external memory cycles representing a broad range of memory speeds for an ICAP width of 8-bits. As the number of bus cycles increased (i.e. representing slower memories), the utilization of the bus due to PR traffic increases from 74% to 94%. As the external memory became slower, a larger share of the bus time was spent on transfers from the external memory to the ICAP controller's BRAM. At the same time, the utilization of the bus due to non-PR traffic was reduced. Longer PR transfers due to slower memory resulted in higher utilization of the bus and thus non-PR had less utilization of the bus. Longer bus transfers also resulted in fewer transfers of PR traffic across the ICAP and thus its utilization went down. Thus, slower memories not only slow down the PR



(a) Case-Study 2: Impact of External Memory Speed on Utilization.



(b) Case-Study 2: Impact of External Memory Speed on Throughput.

Figure 5.11: Case-Study 2 Results: Impact of Varying External Memory Speed on Utilization and Throughput.

process but all other traffic sharing on the bus as shown in Figure 5.11(b).

5.3.2.4 Expected BRAM Requirements

The average number of bytes residing in the ICAP queue at any given time is shown in Figure 5.12. This plot provides insight into the requirements for BRAM inside the ICAP controller. As the the size of the ICAP was varied from 4 bits to 128 bits, the average number of customers decayed from 486.61 bytes to 0.62 bytes. These results predict that a BRAM size of 128 bytes would be more than sufficient for such a reconfigurable system when the ICAP is 8-bit wide.

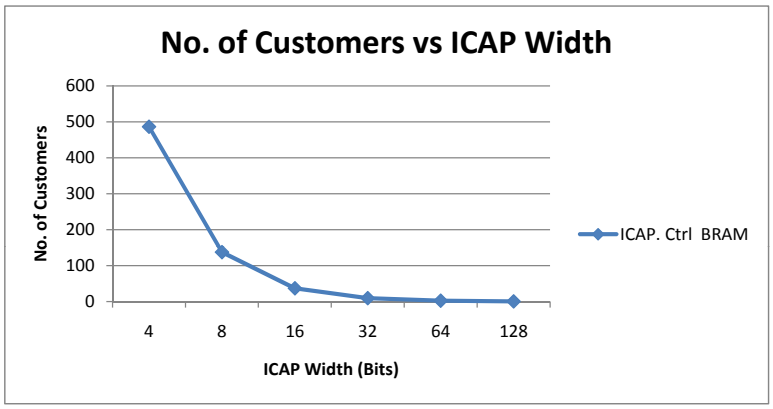


Figure 5.12: Impact of ICAP Width on the expected BRAM size requirements.

5.3.3 Comparative Results Discussion: Case Study-1 vs Case Study-2

5.3.3.1 Comparing Increasing ICAP-Width Results

Several trends can be noticed when comparing the results from architecture of platform-1 as shown in Figure 5.4 and example platform-2 as shown in Figure 5.10. In both cases, an increase in ICAP width reduces ICAP utilization thus putting increased pressure on the bus. The difference between the architectures, however, is that platform-1 has a more dramatic response with the utilization of the bus shifting from 5.3% to 95.1% whereas the utilization of the bus is less pronounced for Platform-2 with an increase from 83% to 99%. For platform-2, the bus is the bottleneck for the system and

has a large influence on throughput for all ICAP widths. This is likely due to the fact that the bitstream has to pass through the bus for two phases before reaching the ICAP. These phases occur in sequence and thus do not compete for the PLB; however, each phase must still compete with non-PR traffic. In contrast, in platform-1, PR traffic requires only one pass through the PLB making it more balanced with the ICAP in terms of throughput. In fact, the bus and the ICAP are roughly balanced when the ICAP has a width of 64-bits. Because platform-2 has two phases involving the bus, the overall throughput of the system is significantly worse than platform-1. For an ICAP width of 8-bits, the throughput is an order of magnitude faster for platform-1.

Although platform-1 outperforms platform-2 with respect to PR throughput, this is in partly due to the assumption made in the model with respect to the intermediate storage for platform-2. In phase-1 of this platform, partial bitstreams are transfered to processor memory via the PLB. In the next phase, the partial bitstream is transfered from processor memory to BRAM via the PLB. The assumption made is that each new bitstream loaded into configuration memory originates in external memory. A more efficient approach would have partial bitstreams cached in the processor memory. Thus, a percentage of bitstreams would originate at the processor memory rather than external memory. Depending on the cache hit rate, this approach would improve the performance of platform-2.

For both platforms, results demonstrated the impact of non-PR traffic on PR traffic thus supporting the need to consider non-PR traffic early in the design cycle. Comparatively speaking, non-PR had a larger share of bus utilization for platform-2 versus platform-1. This is likely because platform-1 has its PR traffic transfered across the PLB in burst mode in 128-byte bursts.

5.3.3.2 Comparing Varying External Memory Speed Results

The utilization and throughput were measured as a function of the speed of external memory for both of the example platforms. For both platforms, the bus utilization approached 100% as the required number of memory cycles increased. Further, utilization of the bus due to non-PR traffic decreases as the bus becomes more occupied with the larger transfers of PR traffic from external memory. Similar to the ICAP width results, the relative difference between the throughput of PR versus non-PR traffic is significantly greater for platform-1. This is due to the fact that phase-1 bus transfers for platform-1 operate in burst mode and thus it's bus is dominated by PR

traffic.

5.3.3.3 Comparing BRAM Requirements Results

Performance trends for the expected number of customers in the BRAM queue for platform-1 and platform-2 are provided in Figures 5.6 and 5.12 respectively. These results are used to estimate the BRAM memory requirements for both platforms. The overall shape of the trend lines are similar; however, the expected BRAM requirements for platform-1 was an order of magnitude larger than that for platform-2. The bus serves as a significant bottleneck for platform-2 and thus the ICAP utilization is an order of magnitude less than that in platform-1. The lower the utilization of the BRAM, the fewer the average number of customers in the queue. Thus, platform-1 requires less BRAM to accommodate fewer customers.

5.4 Summary

In this chapter, we described a method for modeling the dynamic partial reconfiguration (DPR) of FPGA-based systems using multi-class Queuing Networks. We provided a generalized scheme for mapping hardware components of the reconfiguration datapath to queueing primitives. We provided an example system which we modeled and solved using simulation- and analytic-based approaches. Results were provided for system throughput and utilization as a function of ICAP width and external memory speed. These results were used to identify bottlenecks and optimize the system for both reconfiguration time and cost in the presence of non-PR traffic.

The results demonstrate the usefulness of the proposed model for the early design of PR systems. We were able to evaluate two different base systems under a variety of ICAP widths and external memory speeds. From these results, one could decide whether or not the potential gain in PR throughput is worth the extra costs associated with an architecture that supports a wider ICAP port. Although PR throughput increases with ICAP width, non-PR throughput decreases. Thus, this design decision would need to be made based on the criticality of PR versus non-PR traffic. An increase in external memory speed benefits both PR and non-PR throughput, so the results could be used to decide whether or not the increased performance is worth the costs associated with faster memory. Results indicating the average number of customers in the ICAP queue could be used to allocate the size of BRAM needed to implement the ICAP buffer.

The proposed modeling approach is a promising tool for the design and evaluation of dynamically reconfigurable systems. It helps system designers to make informed decisions early in the design process thus avoiding the time and costs associated with building candidate systems. Further, the ease at which candidate architectures can be evaluated allows for a broader exploration of the design space and results in a faster and lower-risk design process.

Chapter 6

Conclusions and Future Work

6.1 Thesis Summary

A Field-Programmable-Gate-Array (FPGA) is an integrated circuit designed to be (re)configured in the field after manufacturing. Compared to application-specific-integrated circuits (ASICs) and general-purpose-processors (GPPs), FPGAs represent a compromise in which their speed and power-efficiency is better than GPPs [KMN02] and their flexibility, time-to-market, and low-volume costs are better than ASICs [KR06]. An FPGA uses a large number of transistors per logic function to provide the programmability and therefore pays a very high price in terms of power consumption and area-efficiency. The continuous shrinking of CMOS transistors is making matters worse by drastically increasing the leakage power dissipation in FPGAs.

Power optimization, in general, is a holistic process and requires strategies at various levels in the design process. These strategies range from metal-level enhancements in the process technology, innovative low-power techniques at the FPGA architecture level to software optimizations inside the CAD tool chain.

Towards that end, the work in this thesis has contributed a new high-level synthesis (HLS) based software power optimization methodology and a modeling technique that increased the effectiveness of low-power architectural features, such as Dynamic Power Gating and Dynamic Partial Reconfiguration, in reducing the overall static leakage power in an FPGA.

Chapter 3 introduced a high-level synthesis based design methodology that targets an FPGA supporting dynamic power gating. The proposed methodology is based on a high-level synthesis compiler framework that automatically finds the coarse-grained (accelerator-level) power gating opportunities in a design expressed in the C language and exploits the power-gating feature of the FPGA to minimize the static power dissipation. In doing so, the designer is insulated from the burden of detecting profitable

power gating opportunities. A computer-aided design (CAD) framework that demonstrated the proposed methodology was also discussed. This framework was applied to the set of CHStone benchmarks to demonstrate the effectiveness of the proposed approach. Results showed a reduction of 14% to 61% in static energy for individual accelerators using the dynamic power-gating technique. The proposed technique has been published in [ABW+14]. To the best of the author's knowledge, this is the first HLS-based framework that automates the design for dynamic power-gated FPGAs.

Chapter 4 looked at the intra-accelerator-level power gating opportunities – turning off individual sub-accelerators within a parent accelerator, which we refer to as hierarchical power gating. The idea behind hierarchical power gating is that for very large accelerators with several phases implemented by sub-accelerators, turning off individual sub-accelerators while parent or other sub-accelerators are running could result in more leakage savings. Chapter 4 showed that fine-grained control over the sub-accelerators allows them to sleep when only the parent context is required. Similarly, power gating the parent accelerator when the child is running creates more power saving opportunities but increases the program run time. An experimental study on CHStone benchmarks showed that hierarchical power-gating saves 8%–25% of static energy when the parent and descendant accelerators are power-gated independently such that the parent accelerator is power-gated while the sub-accelerator runs. This work has been published in [AWHK15].

Finally, Chapter 5 presented a modeling approach to predict the performance trends and bottlenecks in an FPGA-based dynamic partial reconfigurable system. The modeling approach is based on Queueing Theory in which a reconfigurable datapath is mapped to a multi-class queueing network model. An important aspect of the proposed modeling approach is the ability to predict several performance parameters, such as reconfiguration time, without implementing and running the system on an FPGA device. This early prediction has several uses; for example, it increases the ability to explore the design space and experiment with various system parameters in less time. It could also be used for early power estimation; the predicted reconfiguration time combined with the accelerator sizes estimates the energy dissipated. Two case studies were provided to demonstrate the usefulness and flexibility of the modeling scheme. Results were provided for system throughput and utilization as a function of ICAP width and external memory speed. These results were used to identify bottlenecks and optimize the system for both reconfiguration time and cost in the presence of non-PR traffic. This work has been published in [AH11] and [AH13].

6.1.1 Research Impact

It is expected that the results of this research will dramatically improve the *productivity* of engineers designing FPGA-based low power systems. In particular, the design complexity of dynamic power gating and dynamic partial reconfiguration techniques will be substantially reduced which will allow the designers to exploit these techniques to their full potential for increased leakage savings.

Software engineers with less expertise in programming FPGAs will be benefited at large. The electronics industry will gain the understanding of how to engineer complex low-power FPGA systems in less time. Hence, the design revision time and the time-to-market will be significantly reduced. This will enable cheaper embedded computing systems for use in medical, communications and consumer electronics.

Further, the proposed CAD tool-chain and the modeling technique will help in designing *environment friendly* systems by allowing the use of low-power FPGAs in a convenient manner. The use of low-power FPGAs will reduce the heat generated in a system, reducing the environmental footprint of embedded systems. Furthermore, it will become possible to bring low-power reconfigurable technology to hand-held devices, which remains a long cherished goal of the mobile industry.

6.2 Open Issues and Future Work

6.2.1 Fine Grained Power Gating Opportunities and FPGA Architectural Support

The work in this thesis investigated the impact of performing power gating at two levels of granularity: accelerator-level and intra-accelerator-level. The intra-accelerator-level approach considered immediate sub-accelerators in the hierarchy. A direction for future work is to extend the hierarchical power gating framework to consider power gating at the intra-sub-accelerator-level which may include sub-accelerators within a sub-accelerator or individual basic blocks in a sub-accelerator. It will be interesting to find the optimum power gating granularity, both at the application and architecture level, that leads to maximum leakage savings. It might be easier to find more fine grained power gating opportunities within a sub-accelerator in an application but it might be harder to find the required support in the FPGA architecture; realistic dynamic power gating architectures have significant overhead when parts of the chip are turned off and on. Therefore,

a challenge will be to innovate in architecture that supports power gating, allowing more fine grained opportunities that may exist in an application.

6.2.2 Adjusting Application Schedule to Increase Idleness

The proposed methodology for dynamic power gating uses default LegUp scheduling algorithm – as-soon-as-possible (ASAP) – to find the lengths of idle periods. The choice of the scheduling algorithm can impact the idle periods of accelerators. Going forward, it would be interesting to schedule the application such that more idle periods are generated during which power gating could be potentially applied.

6.2.3 Run-time Power Gating Algorithm

In the proposed methodology, the schedule from a high-level synthesis tool is used to determine the idle periods of individual hardware accelerators (or sub-accelerators) in a synthesized system. The predicted length of these idle periods is used to determine whether the power saved by power-gating the accelerator is more than the overhead of turning the accelerator off and then on again at the end of the idle period. Based on this knowledge, individual accelerators can then be power-gated when it is deemed profitable. In doing so, the workload of the application is examined at compile time for static power gating decisions. Moving forward, it will be interesting to adapt the power gating decisions with varying workloads at run-time. This would require a dynamic power gating predictor which is able to predict the length of an upcoming idle period in advance and then decide whether to power gate.

6.2.4 Relating Software Granularity to Power-Saving Techniques Continuum: A Holistic Design Framework

The work in this thesis focused on dynamic power gating (DPG) and dynamic partial reconfiguration (DPR) techniques for leakage reduction and proposed design methodologies which increased the accessibility of these individual techniques to the end user. In doing so, both techniques were considered independently.

As discussed earlier in Section 2.4.3, clock gating has been shown as an effective technique to reduce dynamic power. The ability to perform clock-gating at run-time, however, depends on the designer's ability to detect and exploit these clock-gating opportunities in a design. The techniques for finding the power gating opportunities through a high-level synthesis

(HLS) framework (Chapter 3 and 4) can be seamlessly targeted towards the dynamic clock gating (DCG) technique which is a proven method to reduce dynamic power.

Moving forward, Figure 6.1 depicts our long-term vision of a holistic design framework in which parts of an application can exploit various power-saving features in an architecture to reduce power. Consider the three mentioned power-saving techniques – DPG, DPR and DCG – as power-saving options available on a continuum in an architecture, where each technique is uniquely characterized by its overhead along with various other parameters. On the application side, consider a software granularity (decomposed from

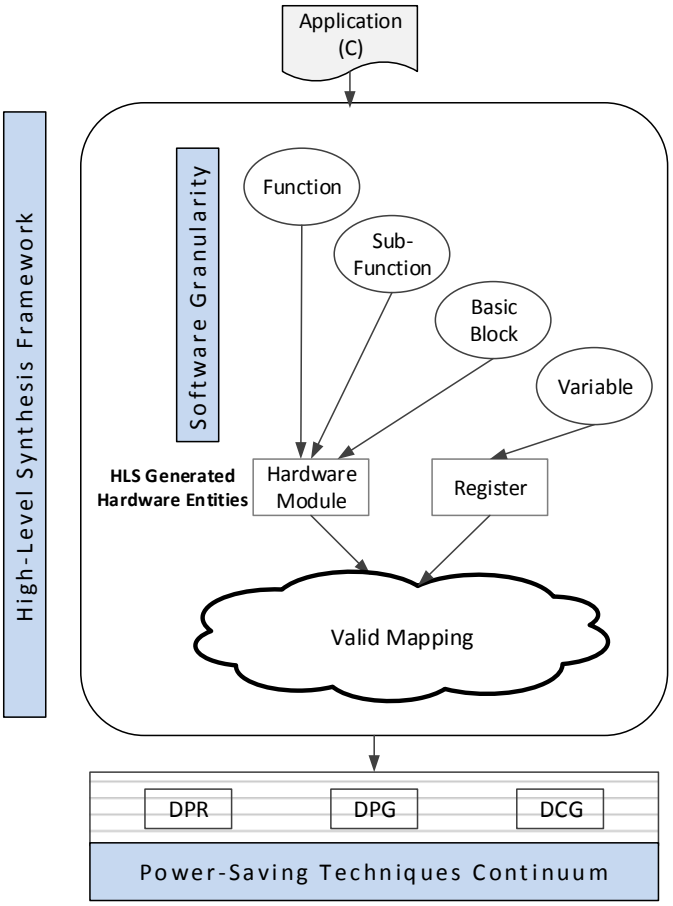


Figure 6.1: An Abstract Vision: Relating Software Granularity to Power-Saving Techniques Continuum - A Holistic Design Framework

the C program) in the form of *functions*, *sub-functions*, *basic blocks* and *variables*. A direction for future work is to add intelligence to the high-level synthesis algorithms such that the tool is able to find the best matching power-saving technique on the continuum for a given level of software granularity. For example, a function (synthesized to a hardware accelerator by HLS) may not be able to take advantage of power gating feature because it is not profitable to do so due to switching overhead but the basic block inside that function (or sub-function) can potentially take advantage of clock gating technique due to low switching overhead. In doing so, wherever possible, each level in the software granularity can potentially exploit one of the power-saving techniques on the continuum. This would allow various parts in a program to take advantage of various power-saving techniques, rather than just one level of granularity being mapped to a single power-saving technique as we described in Chapter 3 and 4. So under the hood, as the application runs in real-time, a combination of various power-saving techniques is applied which holistically will reduce the power.

6.3 Final Remarks

We anticipate that the complexity of applications from various domains will only increase in future and porting such applications to FPGAs will remain challenging in terms of design and power optimization. We believe that increased automation, right from the prototyping to deployment, is one of several methods to cope with the design complexity and power optimization should be a core part of such design automation. This thesis is a first step towards that vision.

Bibliography

[ABW+14] Rehan Ahmed, Assem A.M. Bsoul, Steven J.E Wilton, Peter Hallschmid, and Richard Klukas. High-level Synthesis-based Design Methodology for Dynamic Power-Gated FPGAs. In *24th International Conference on Field Programmable Logic and Applications, FPL 2014, Munich, Germany, 2-4 September, 2014*, pages 1–4, 2014. → pages iii, 4, 46, 125

[ADSN06] K. Agarwal, H. Deogun, D. Sylvester, and K. Nowka. Power gating with multiple sleep modes. In *Quality Electronic Design, 2006. ISQED '06. 7th International Symposium on*, pages 5 pp.–637, March 2006. → pages 39

[AH11] Rehan Ahmed and Peter Hallschmid. Modeling and Evaluation of Dynamic Partial Reconfigurable Datapaths for FPGA-Based Systems Using Stochastic Networks. In *Proceedings of the 2011 21st International Conference on Field Programmable Logic and Applications*, FPL '11, pages 70–75, Washington, DC, USA, 2011. IEEE Computer Society. → pages iii, 125

[AH13] Rehan Ahmed and Peter Hallschmid. Model-based Performance Evaluation of Dynamic Partial Reconfigurable Datapaths for FPGA-based Systems. In *Embedded Systems Design with FPGA*, chapter 5, pages pp 101–123. Springer, 2013. → pages iii, 125

[Alta] Altera. [link]. → pages 16, 21

[Altb] Altera. [link]. → pages 52

[Altc] Altera. *Advanced Synthesis Cookbook*. → pages 17

[Altd] Altera. *Reducing Power Consumption and Increasing Bandwidth on 28-nm FPGAs*. → pages 2, 3, 29

[Alte] Altera. *Using Selectable I/O Standards in Cyclone Devices.* → pages 15

[alt08] 40-nm FPGA Power Management and Advantages. Altera, Corp. white paper WP-01059-1.2 (v1.2), August 2008. → pages 3, 29, 42

[Alt14] Altera. *"Stratix V Device Overview"*. Altera Inc., 101 Innovation Drive, San Jose, CA 95134, 2015.01.15 edition, April 2014. → pages 12

[Alt15] Altera. *"Cyclone IV Device Handbook"*. Altera Inc., 101 Innovation Drive, San Jose, CA 95134, volume 1 edition, 2015. → pages 12

[AN02] J.H. Anderson and F.N. Najm. Power-aware technology mapping for LUT-based FPGAs. In *Field-Programmable Technology, 2002. (FPT). Proceedings. 2002 IEEE International Conference on*, pages 211–218, Dec 2002. → pages 3

[AN04] J.H. Anderson and F.N. Najm. Power estimation techniques for FPGAs. *Very Large Scale Integration (VLSI) Systems, IEEE Transactions on*, 12(10):1015–1027, Oct 2004. → pages 3

[Arca] Altera ALM Architecture. [link]. → pages 15

[Arcb] Xilinx CLB Architecture. [link]. → pages 15

[AWHK15] Rehan Ahmed, Steven J.E Wilton, Peter Hallschmid, and Richard Klukas. Hierarchical Dynamic Power Gating in FPGAs. In *The 11th International Symposium on Applied Reconfigurable Computing, ARC 2015, Bochum, Germany, 13-17 April, 2015*, pages 1–8, 2015. → pages iii, 125

[BCMP75] Forest Baskett, K. Mani Chandy, Richard R. Muntz, and Fernando G. Palacios. Open, Closed, and Mixed Networks of Queues with Different Classes of Customers. *J. ACM*, 22:248–260, April 1975. → pages 37, 38

[BCS09] Marco Bertoli, Giuliano Casale, and Giuseppe Serazzi. JMT: performance engineering tools for system modeling. *SIGMETRICS Perform. Eval. Rev.*, 36(4):10–15, 2009. → pages 107

[BFR+08] Maik Boden, Thomas Fiebig, Markus Reiband, Peter Reichel, and Steffen Rülke. GePaRD - A High-Level Generation Flow for Partially Reconfigurable Designs. In *Proceedings of the 2008 IEEE Computer Society Annual Symposium on VLSI*, ISVLSI '08, pages 298–303, Washington, DC, USA, 2008. IEEE Computer Society. → pages 42

[BJ12] William Lloyd Bircher and Lizy John. Predictive Power Management for Multi-core Processors. In *Proceedings of the 2010 International Conference on Computer Architecture*, ISCA'10, pages 243–255, Berlin, Heidelberg, 2012. Springer-Verlag. → pages 39

[BKBB05] R.P. Bharadwaj, R. Konar, P.T. Balsara, and D. Bhatia. Exploiting temporal idleness to reduce leakage power in programmable architectures. In *Design Automation Conference, 2005. Proceedings of the ASP-DAC 2005. Asia and South Pacific*, volume 1, pages 651–656 Vol. 1, Jan 2005. → pages 7, 39, 44

[Blu] Bluespec. [link]. → pages 16

[BN06] Florent Berthelot and Fabienne Nouvel. Partial and Dynamic Reconfiguration of FPGAs: a top down design methodology for an automatic implementation. *VLSI, IEEE Computer Society Annual Symposium on*, 0:436–437, 2006. → pages 41

[BRM99] Vaughn Betz, Jonathan Rose, and Alexander Marquardt, editors. *Architecture and CAD for Deep-Submicron FPGAs*. Kluwer Academic Publishers, Norwell, MA, USA, 1999. → pages 52

[Bso] Assem A.M. Bsoul. *FPGA architectures and CAD algorithms with dynamic power gating support*. PhD thesis, The University of British Columbia. → pages iii, x, 53, 54

[BW10] A.A.M. Bsoul and S. J E Wilton. An FPGA architecture supporting dynamically controlled power gating. In *Field-Programmable Technology (FPT), 2010 International Conference on*, pages 1–8, 2010. → pages viii, 3, 4, 29, 31, 32, 43, 47, 49, 56, 57, 74, 79, 81, 82

[BW12a] A.A.M. Bsoul and S.J.E. Wilton. An FPGA with power-gated switch blocks. In *Field-Programmable Technology (FPT), 2012 International Conference on*, pages 87–94, Dec 2012. → pages 3, 31, 54

[BW12b] Assem A.M. Bsoul and Steven J.E. Wilton. A Configurable Architecture to Limit Wakeup Current in Dynamically-controlled Power-gated FPGAs. In *Proceedings of the ACM/SIGDA International Symposium on Field Programmable Gate Arrays*, FPGA '12, pages 245–254, New York, NY, USA, 2012. ACM. → pages 3, 8, 31, 39, 45, 50

[BW14] Assem A. M. Bsoul and Steven J. E. Wilton. A Configurable Architecture to Limit Inrush Current in Power-Gated Reconfigurable Devices. *J. Low Power Electronics*, 10(1):1–15, 2014. → pages 8, 39, 45

[Cad] Cadence. [link]. → pages 16, 21

[Cat] Catapult. [link]. → pages 16, 21

[CB] Rakesh Chadha and J. Bhasker. *An ASIC Low Power Primer Analysis*. → pages 6

[CB95] Anantha P. Chandrakasan and Robert W. Brodersen. *Low Power Digital CMOS Design*. Kluwer Academic Publishers, Norwell, MA, USA, 1995. → pages 45

[CBA13] Jongsok Choi, S. Brown, and J. Anderson. From software threads to parallel hardware in high-level synthesis for FPGAs. In *Field-Programmable Technology (FPT), 2013 International Conference on*, pages 270–277, Dec 2013. → pages 20

[CCF+13] A. Canis, Jongsok Choi, B. Fort, Ruolong Lian, Qijing Huang, N. Calagar, M. Gort, Jia Jun Qin, M. Aldham, T. Czajkowski, S. Brown, and J. Anderson. From software to accelerators with LegUp high-level synthesis. In *Compilers, Architecture and Synthesis for Embedded Systems (CASES), 2013 International Conference on*, pages 1–9, Sept 2013. → pages 20, 46, 47, 52, 73, 74

[CGMT09] P. Coussy, D.D. Gajski, M. Meredith, and A. Takach. An Introduction to High-Level Synthesis. *Design Test of Computers, IEEE*, 26(4):8–17, July 2009. → pages 18

[CGRG08] Chris Conger, Ann Gordon-Ross, and Alan D. George. Design Framework for Partial Run-Time FPGA Reconfiguration. In Plaks [Pla08], pages 122–128. → pages 42

[CMHM10] E.S. Chung, P.A. Milder, J.C. Hoe, and Ken Mai. Single-Chip Heterogeneous Computing: Does the Future Include Custom Logic, FPGAs, and GPGPUs? In *Microarchitecture (MICRO), 2010 43rd Annual IEEE/ACM International Symposium on*, pages 225–236, 2010. → pages 1, 12

[CTL+05] C.T. Chow, L.S.M. Tsui, P.H.W. Leong, W. Luk, and S.J.E. Wilton. Dynamic voltage scaling for commercial FPGAs. In *Field-Programmable Technology, 2005. Proceedings. 2005 IEEE International Conference on*, pages 173–180, Dec 2005. → pages 3, 28, 29

[CXS04] Jin-Hyeok Choi, Yingxue Xu, and T. Sakurai. Statistical leakage current reduction in high-leakage environments using locality of block activation in time domain. *Solid-State Circuits, IEEE Journal of*, 39(9):1497–1503, Sept 2004. → pages 7, 39, 44

[CZ06] J. Cong and Zhiru Zhang. An efficient and versatile scheduling algorithm based on SDC formulation. In *Design Automation Conference, 2006 43rd ACM/IEEE*, pages 433–438, 2006. → pages 21

[CZS+08] Christopher Claus, Bin Zhang, Walter Stechele, Lars Braun, Michael Hübner, and Jürgen Becker. A multi-platform controller allowing for maximum Dynamic Partial Reconfiguration throughput. In *FPL*, pages 535–538, 2008. → pages 40

[DCW09] Quang Dinh, Deming Chen, and Martin D.F. Wong. A Routing Approach to Reduce Glitches in Low Power FPGAs. In *Proceedings of the 2009 International Symposium on Physical Design*, ISPD '09, pages 99–106, New York, NY, USA, 2009. ACM. → pages 3

[DKA+02] S. Dropsho, V. Kursun, D.H. Albonesi, S. Dwarkadas, and E.G. Friedman. Managing static leakage energy in microprocessor functional units. In *Microarchitecture, 2002. (MICRO-35). Proceedings. 35th Annual IEEE/ACM International Symposium on*, pages 321–332, 2002. → pages 39

[DML11] François Duhem, Fabrice Muller, and Philippe Lorenzini. Methodology for designing partially reconfigurable systems using transaction-level modeling. In *DASIP*, pages 316–322, 2011. → pages 42

[Exp] Y Explorations. [link]. → pages 16, 21

[GLS99] T. Grandpierre, C. Lavarenne, and Y. Sorel. Optimized rapid prototyping for real-time embedded heterogeneous multiprocessors. In *Proceedings of the seventh international workshop on Hardware/software codesign*, CODES '99, pages 74–78, New York, NY, USA, 1999. ACM. → pages 41

[GPLR08] Juan Galindo, Eric Peskin, Brad Larson, and Gene Roylance. Leveraging Firmware in Multichip Systems to Maximize FPGA Resources: An Application of Self-Partial Reconfiguration. *Reconfigurable Computing and FPGAs, International Conference on*, 0:139–144, 2008. → pages 40

[Gra] LenardoSpectrum Mentor Graphnics. [link]. → pages 17

[GS03] Thierry Grandpierre and Yves Sorel. From Algorithm and Architecture Specifications to Automatic Generation of Distributed Real-Time Executives: a Seamless Flow of Graphs Transformations. In *Proceedings of the First ACM and IEEE International Conference on Formal Methods and Models for Co-Design*, MEMOCODE '03, pages 123–, Washington, DC, USA, 2003. IEEE Computer Society. → pages 41

[GVPR04] Björn Griese, Erik Vonnahme, Mario Porrmann, and Ulrich Rückert. Hardware Support for Dynamic Reconfiguration in Reconfigurable SoC Architectures. In *FPL*, pages 842–846, 2004. → pages 40

[HA10] Hassan Hassan and Mohab Anis. *Low-Power Design of Nanometer FPGAs: Architecture and EDA*. Morgan Kaufmann Publishers, 2010. → pages 29

[HBS+04] Zhigang Hu, Alper Buyuktosunoglu, Viji Srinivasan, Victor Zyuban, Hans Jacobson, and Pradip Bose. Microarchitectural Techniques for Power Gating of Execution Units. In *Proceedings*

of the 2004 International Symposium on Low Power Electronics and Design, ISLPED '04, pages 32–37, New York, NY, USA, 2004. ACM. → pages 39

[HCLH90] Chu-Yi Huang, Yen-Shen Chen, Youn-Long Lin, and Yu-Chin Hsu. Data path allocation based on bipartite weighted matching. In *Design Automation Conference, 1990. Proceedings., 27th ACM/IEEE*, pages 499–504, Jun 1990. → pages 21

[HD07] Scott Hauck and Andre DeHon. *Reconfigurable Computing: The Theory and Practice of FPGA-Based Computation*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2007. → pages 12

[HLL08] Pao-Ann Hsiung, Chao-Sheng Lin, and Chih-Feng Liao. Perfecto: A systemc-based design-space exploration framework for dynamically reconfigurable architectures. *ACM Trans. Reconfigurable Technol. Syst.*, 1:17:1–17:30, September 2008. → pages 41

[HN14] Mohammad Hosseinabady and Jose Luis Nunez-Yanez. Runtime power gating in hybrid ARM-FPGA devices. In *24th International Conference on Field Programmable Logic and Applications, FPL 2014, Munich, Germany, 2-4 September, 2014*, pages 1–6, 2014. → pages 39

[HTH+08] Y. Hara, H. Tomiyama, S. Honda, H. Takada, and K. Ishii. CHStone: A benchmark program suite for practical C-based high-level synthesis. In *Circuits and Systems, 2008. ISCAS 2008. IEEE International Symposium on*, pages 1192–1195, May 2008. → pages 56, 79

[IHK11] S. Ishihara, M. Hariyama, and M. Kameyama. A Low-Power FPGA Based on Autonomous Fine-Grain Power Gating. *Very Large Scale Integration (VLSI) Systems, IEEE Transactions on*, 19(8):1394–1406, Aug 2011. → pages 7, 39, 44

[Jac57] James R. Jackson. Networks of Waiting Lines. *Operations Research*, 5(4):pp. 518–521, 1957. → pages 36

[JHH] Matt Klein Jameel Hussein and Michael Hart. *Lowering Power at 28 nm with Xilinx 7 Series Devices*. → pages 2, 3, 4, 26, 28, 29, 30

[JKGS10] Peter Jamieson, Kenneth B. Kent, Farnaz Gharibian, and Lesley Shannon. Odin II - An Open-source Verilog HDL Synthesis tool for CAD Research. In *Proceedings of the IEEE Symposium on Field-Programmable Custom Computing Machines*, pages 149–156, 2010. → pages 17, 52

[KKRS12] Andrew B. Kahng, Seokhyeong Kang, Tajana Rosing, and Richard Strong. TAP: Token-based Adaptive Power Gating. In *Proceedings of the 2012 ACM/IEEE International Symposium on Low Power Electronics and Design*, ISLPED '12, pages 203–208, New York, NY, USA, 2012. ACM. → pages 39

[Kle09] Matt Klein. Power Consumption at 40 and 45 nm. Xilinx, Inc. white paper WP298 (v1.0), April 2009. → pages 3, 29, 42

[KMN02] K. Keutzer, S. Malik, and A.R. Newton. From ASIC to ASIP: the next design discontinuity. In *Computer Design: VLSI in Computers and Processors, 2002. Proceedings. 2002 IEEE International Conference on*, pages 84 – 90, 2002. → pages 1, 12, 124

[Kol] Srinivasa Kolluri. *Power Reduction in Next-Generation Ultra-Scale Architecture*. → pages 2, 3, 29

[KR06] Ian Kuon and Jonathan Rose. Measuring the gap between FPGAs and ASICs. In *Proceedings of the 2006 ACM/SIGDA 14th international symposium on Field programmable gate arrays*, FPGA '06, pages 21–30, New York, NY, USA, 2006. ACM. → pages 1, 12, 27, 124

[KR07] Ian Kuon and J. Rose. Measuring the Gap Between FPGAs and ASICs. *Computer-Aided Design of Integrated Circuits and Systems, IEEE Transactions on*, 26(2):203–215, 2007. → pages 1, 25

[LA04] Chris Lattner and Vikram Adve. LLVM: A Compilation Framework for Lifelong Program Analysis & Transformation. In *Proceedings of the 2004 International Symposium on Code Generation and Optimization (CGO'04)*, Palo Alto, California, Mar 2004. → pages 20

[LBBS09] Anita Lungu, Pradip Bose, Alper Buyuktosunoglu, and Daniel J. Sorin. Dynamic Power Gating with Quality Guarantees. In *Proceedings of the 2009 ACM/IEEE International Symposium on Low Power Electronics and Design*, ISLPED '09, pages 377–382, New York, NY, USA, 2009. ACM. → pages 39

[LBM+06] P. Lysaght, B. Blodget, J. Mason, J. Young, and B. Bridgford. Invited Paper: Enhanced Architectures, Design Methodologies and CAD Tools for Dynamic Reconfiguration of Xilinx FPGAs. In *Field Programmable Logic and Applications, 2006. FPL '06. International Conference on*, pages 1 –6, aug. 2006. → pages 4

[LCHC03] Fei Li, Deming Chen, Lei He, and Jason Cong. Architecture Evaluation for Power-efficient FPGAs. In *Proceedings of the 2003 ACM/SIGDA Eleventh International Symposium on Field Programmable Gate Arrays*, FPGA '03, pages 175–184, New York, NY, USA, 2003. ACM. → pages 3, 26

[LL08] Julien Lamoureux and Wayne Luk. An Overview of Low-Power Techniques for Field-Programmable Gate Arrays. In *Proceedings of the 2008 NASA/ESA Conference on Adaptive Hardware and Systems*, pages 338–345, 2008. → pages 29

[LLH+05] Fei Li, Yan Lin, Lei He, Deming Chen, and J. Cong. Power modeling and characteristics of field programmable gate arrays. *Computer-Aided Design of Integrated Circuits and Systems, IEEE Transactions on*, 24(11):1712–1724, Nov 2005. → pages ix, 26, 27

[LLHC04] Fei Li, Yan Lin, Lei He, and Jason Cong. Low-power FPGA Using Pre-defined dual-Vdd/dual-Vt Fabrics. In *Proceedings of the 2004 ACM/SIGDA 12th International Symposium on Field Programmable Gate Arrays*, FPGA '04, pages 42–50, New York, NY, USA, 2004. ACM. → pages 29

[LPF10] Shaoshan Liu, Richard Neil Pittman, and Alessandro Forin. Energy reduction with run-time partial reconfiguration (abstract only). In *Proceedings of the 18th annual ACM/SIGDA international symposium on Field programmable gate arrays*, FPGA '10, pages 292–292, New York, NY, USA, 2010. ACM. → pages 4

[LW03] Julien Lamoureux and Steven J. E. Wilton. On the Interaction Between Power-Aware FPGA CAD Algorithms. In *Proceedings of the 2003 IEEE/ACM International Conference on Computer-aided Design*, ICCAD '03, pages 701–, Washington, DC, USA, 2003. IEEE Computer Society. → pages 3

[NNY12] Atukem Nabina and Jose Luis Nunez-Yanez. Adaptive Voltage Scaling in a Dynamically Reconfigurable FPGA-Based Platform. *ACM Trans. Reconfigurable Technol. Syst.*, 5(4):20:1–20:22, December 2012. → pages 3, 28, 29

[NY13] Jose Nunez-Yanez. Energy Proportional Computing in Commercial FPGAs with Adaptive Voltage Scaling. In *Proceedings of the 10th FPGAworld Conference*, FPGAworld '13, pages 6:1–6:5, New York, NY, USA, 2013. ACM. → pages 3, 28, 29

[PCC+14] Andrew Putnam, Adrian Caulfield, Eric Chung, Derek Chiou, Kypros Constantinides, John Demme, Hadi Esmaeilzadeh, Jeremy Fowers, Gopi Prashanth Gopal, Jan Gray, Michael Haselman, Scott Hauck, Stephen Heil, Amir Hormati, Joo-Young Kim, Sitaram Lanka, Jim Larus, Eric Peterson, Simon Pope, Aaron Smith, Jason Thong, Phillip Yi Xiao, and Doug Burger. A Reconfigurable Fabric for Accelerating Large-Scale Datacenter Services. In *41st Annual International Symposium on Computer Architecture (ISCA)*, June 2014. → pages 1

[PDH11] Kyprianos Papadimitriou, Apostolos Dollas, and Scott Hauck. Performance of Partial Reconfiguration in FPGA Systems: A Survey and a Cost Model. *ACM Trans. Reconfigurable Technol. Syst.*, 4(4):36:1–36:24, December 2011. → pages 40, 114

[PKT90] K. R. Pattipati, M. M. Kostreva, and J. L. Teele. Approximate mean value analysis algorithms for queuing networks: existence, uniqueness, and convergence results. *J. ACM*, 37:643–673, July 1990. → pages 112

[Pla08] Toomas P. Plaks, editor. *Proceedings of the 2008 International Conference on Engineering of Reconfigurable Systems & Algorithms, ERSA 2008, Las Vegas, Nevada, USA, July 14-17, 2008*. CSREA Press, 2008. → pages 6, 134

[PWY05] Kara K. W. Poon, Steven J. E. Wilton, and Andy Yan. A Detailed Power Model for Field-programmable Gate Arrays. *ACM*

Trans. Des. Autom. Electron. Syst., 10(2):279–302, April 2005. → pages 3

[QTSN07] Yang Qu, Kari Tiensyrjä, Juha-Pekka Soininen, and Jari Nurmi. System-Level Design for Partially Reconfigurable Hardware. In *ISCAS*, pages 2738–2741, 2007. → pages 41

[Rab09] Jan Rabaey. *Low Power Design Essentials*. Springer Publishing Company, Incorporated, 1st edition, 2009. → pages 45

[RJD98] Anand Raghunathan, Niraj K. Jha, and Sujit Dey. *High-Level Power Analysis and Optimization*. Kluwer Academic Publishers, Norwell, MA, USA, 1998. → pages x, 45, 46

[RPOG02] Siddharth Rele, Santosh Pande, Soner Önder, and Rajiv Gupta. Optimizing Static Power Dissipation by Functional Units in Superscalar Processors. In *Proceedings of the 11th International Conference on Compiler Construction*, CC '02, pages 261–275, London, UK, UK, 2002. Springer-Verlag. → pages 39

[RRK09] S. Roy, N. Ranganathan, and S. Katkoori. A Framework for Power-Gating Functional Units in Embedded Microprocessors. *Very Large Scale Integration (VLSI) Systems, IEEE Transactions on*, 17(11):1640–1649, Nov 2009. → pages 39

[SG] Berkeley Logic Synthesis and Verification Group. ABC A System for Sequential Synthesis and Verification. → pages 17

[SKB02] Li Shang, Alireza S. Kaviani, and Kusuma Bathala. Dynamic Power Consumption in Virtex™-II FPGA Family. In *Proceedings of the 2002 ACM/SIGDA Tenth International Symposium on Field-programmable Gate Arrays*, FPGA '02, pages 157–164, New York, NY, USA, 2002. ACM. → pages ix, 26

[SSCS10] Youngsoo Shin, Jun Seomun, Kyu-Myung Choi, and Takayasu Sakurai. Power Gating: Circuits, Design Methodologies, and Best Practice for Standard-cell VLSI Designs. *ACM Trans. Des. Autom. Electron. Syst.*, 15(4):28:1–28:37, October 2010. → pages 39

[Sui] Xilinx Vivado Desing Suite. [link]. → pages 17, 29

[Syna] Synopsys Synplify Pro Logic Synthesis. [link]. → pages 17

[Synb] Xilinx XST Synthesis. [link]. → pages 17

[TCL09] D.B. Thomas, J. Coutinho, and W. Luk. Reconfigurable computing: Productivity and performance. In *Signals, Systems and Computers, 2009 Conference Record of the Forty-Third Asilomar Conference on*, pages 685 –689, nov. 2009. → pages 6

[TKR+06] Tim Tuan, Sean Kao, Arif Rahman, Satyaki Das, and Steve Trimberger. A 90Nm Low-power FPGA for Battery-powered Applications. In *Proceedings of the 2006 ACM/SIGDA 14th International Symposium on Field Programmable Gate Arrays*, FPGA '06, pages 3–11, New York, NY, USA, 2006. ACM. → pages ix, 26, 27

[TSM] TSMC. TSMC: The 28nm Process Family. → pages 2

[UO06] K. Usami and N. Ohkubo. A Design Approach for Fine-grained Run-Time Power Gating using Locally Extracted Sleep Signals. In *Computer Design, 2006. ICCD 2006. International Conference on*, pages 155–161, Oct 2006. → pages 7, 39, 44

[VRD+08] K. Vorwerk, M. Raman, J. Dunoyer, Yaun chung Hsu, A. Kundu, and A. Kennings. A technique for minimizing power during FPGA placement. In *Field Programmable Logic and Applications, 2008. FPL 2008. International Conference on*, pages 233–238, Sept 2008. → pages 3

[WH] Neil H. E. Weste and David Money Harris. *CMOS VLSI Design A Circuits and Systems Perspective*. 4th edition. → pages 21, 25

[Wik] Wikipedia. Field-programmable gate array (FPGA) Wiki. → pages 1

[Xila] Xilinx. [link]. → pages 16, 21

[Xilb] Xilinx. *7 Series FPGAs SelectIO Resources User Guide*. → pages 15

[Xil08] Xilinx. *"Virtex-4 FPGA User Guide"*. Xilinx Inc., 2100 Logic Drive, San Jose CA 95124, v2.6 edition, December 1 2008. → pages 12

[Xil12a] Xilinx. *"THE FIRST GENERATION OF EXTENSIBLE PROCESSING PLATFORMS: A NEW LEVEL OF PERFORMANCE, FLEXIBILITY AND SCALABILITY"*. Xilinx Inc., 2100 Logic Drive, San Jose CA 95124, v5.4 edition, March 16 2012. → pages 12, 16

[Xil12b] Xilinx. *"Virtex-5 FPGA User Guide"*. Xilinx Inc., 2100 Logic Drive, San Jose CA 95124, v5.4 edition, March 16 2012. → pages 12

[YCCY11] Chang-Ching Yeh, Kuei-Chung Chang, Tien-Fu Chen, and Chingwei Yeh. Maintaining Performance on Power Gating of Microprocessor Functional Units by Using a Predictive Prewakeup Strategy. *ACM Trans. Archit. Code Optim.*, 8(3):16:1–16:27, October 2011. → pages 39

[YR04] A.G. Ye and J. Rose. Using multi-bit logic blocks and automated packing to improve field-programmable gate array density for implementing datapath circuits. In *Field-Programmable Technology, 2004. Proceedings. 2004 IEEE International Conference on*, pages 129–136, Dec 2004. → pages 1, 25

[ZKV+03] W. Zhang, M. Kandemir, N. Vijaykrishnan, M.J. Irwin, and V. De. Compiler support for reducing leakage energy consumption. In *Design, Automation and Test in Europe Conference and Exhibition, 2003*, pages 1146–1147, 2003. → pages 39