UNIVERSITY OF THESSALY

DOCTORAL THESIS

Design space exploration in near-data co-processors for general-purpose acceleration, in high-performance and low-power processing environments

Author:
Athanasios TZIOUVARAS

Supervisors:
Prof. George STAMOULIS
Prof. Nestor EVMORFOPOULOS
Prof. Athanasios LOUKOPOULOS

A thesis submitted in fulfillment of the requirements for the degree of Doctor of Philosophy

in the

Electronics Lab
Department of Electrical and Computer Engineering

May 10, 2021

Declaration of Authorship

I, Athanasios TZIOUVARAS, declare that this thesis titled, "Design space exploration in near-data co-processors for general-purpose acceleration, in high-performance and low-power processing environments" and the work presented in it are my own. I confirm that:

- This work was done wholly or mainly while in candidature for a research degree at this University.
- Where any part of this thesis has previously been submitted for a degree or any other qualification at this University or any other institution, this has been clearly stated.
- Where I have consulted the published work of others, this is always clearly attributed.
- Where I have quoted from the work of others, the source is always given. With the exception of such quotations, this thesis is entirely my own work.
- I have acknowledged all main sources of help.
- Where the thesis is based on work done by myself jointly with others, I have made clear exactly what was done by others and what I have contributed myself.

Signed: _____

Date: _____

"*Fooling around with alternating current (AC) is just a waste of time. Nobody will use it, ever.*"

Thomas Edison

UNIVERSITY OF THESSALY

Abstract

Department of Electrical and Computer Engineering

Doctor of Philosophy

Design space exploration in near-data co-processors for general-purpose acceleration, in high-performance and low-power processing environments

by Athanasios TZIOUVARAS

Modern computer architectures face a performance scaling wall as the throughput and power consumption bottleneck has shifted from the core pipeline towards the DRAM latency and data transfer operations. This phenomenon can be partially attributed to the stop of Dennard's scaling and to the continuous shrinking size of transistors. As a result, the power density of the integrated circuits has increased to a point where most of the cores in a multi-core architecture are forced to operate in near-threshold voltage levels. In order to address such an issue, researchers tend to deviate from the standard Von Neuman architectures towards new computing models. In the last decade there is a resurgence of the NDP paradigm, under which the instructions are executed on the DRAM die instead of the core pipeline. Therefore, the amount of CPU-DRAM transactions is significantly decreased and thus, it positively affects the power dissipation and the achievable throughput of the system. Under this premise, in this dissertation we explore the NDP paradigm for high performance and for low-power computing. Regarding the high performance computing, we propose a novel approach that considers general purpose loop execution. Our design employs an instruction scheduling methodology which issues each individual instruction on a custom integrated circuit acting as loop accelerator that is located on the logic layer of an HMC DRAM. There, instructions are iteratively executed in parallel in a software pipelining fashion, while intermediate results are forwarded through an on-chip interconnection network. Regarding the low-power computing, we develop a novel timing analysis methodology that is based on the premises of STA, specifically for low-power, low-end pipelines. The proposed timing methodology considers the excitation of the timing paths for each instruction supported by the ISA, and calculates the worst-case slack for each individual instruction. As a result, we obtain timing information on an instruction level and we proceed in exploiting such knowledge to adaptively scale the clock frequency according to instruction types that execute in the pipeline at any given time. In the sequel, we employ the aforementioned BTWC methodology to co-design a pipeline from the ground up to support a clock scaling mechanism with cycle-to-cycle granularity. We focus on the general purpose code execution and we implement our design on the logic layer of an HMC DRAM in order to enable near-data execution. We opt to evaluate both the high performance and the low power architectures on post-layout simulations in order to strengthen the validity of our designs. Results indicate a significant performance increase in terms of throughput over the baseline processors while the power consumption levels are critically reduced.

Greek Abstract

Εξερεύνηση σχεδιαστικού χώρου των συν-επεξεργαστών κοντά στην κύρια μνήμη για επιτάχυνση κώδικα γενικού σκοπού, σε περιβάλλοντα υψηλών επιδόσεων και χαμηλής κατανάλωσης ισχύος

Οι σύγχρονες αρχιτεκτονικές υπολογιστών είναι αντιμέτωπες με ένα σοβαρό πρόβλημα που αφορά την κλιμάκωση της απόδοσης τους, καθώς η συμφόρηση της πληροφορίας έχει μετατοπιστεί από τον πυρήνα του επεξεργαστή στην μονάδα της κύριας μνήμης και στις λειτουργίες μεταφοράς δεδομένων. Το φαινόμενο αυτό μπορεί μερικώς να αποδοθεί στο τέλος της ισχύος του νόμου του Dennard και στην διαρκή μείωση του μεγέθους των τρανσίστορς. Ως αποτέλεσμα, η πυκνότητα ισχύος των ολοκληρωμένων κυκλωμάτων έχει αυξηθεί τόσο, ώστε η λειτουργία των πολύ-πυρηνικών επεξεργαστών να επιτελείται σε τάσεις που βρίσκονται κοντά στην τάση κατωφλίου. Για να ξεπεράσουν το πρόβλημα αυτό, οι ερευνητές τείνουν να αποκλίνουν από τις κλασικές αρχιτεκτονικές προσεγγίσεις τύπου Von Neuman και να στρέφουν την προσοχή τους σε νέα μοντέλα επεξεργασίας. Την τελευταία δεκαετία έχει παρατηρηθεί μία αναζωπύρωση του ενδιαφέροντος για το παράδειγμα εκτέλεσης εντολών κοντά στην κύρια μνήμη (NDP) , κατά το οποίο οι εντολές εκτελούνται στο κύκλωμα της κύριας μνήμης αντί του κεντρικού επεξεργαστή. Έτσι, ο αριθμός των λειτουργιών της μεταφοράς δεδομένων μεταξύ της κύριας μνήμης και του επεξεργαστή μειώνεται σημαντικά, κάτι το οποίο επιδρά θετικά στην κατανάλωση ισχύος και την επιτεύξιμη απόδοση του συστήματος. Κινούμενοι προς αυτήν την υπόθεση, στην διατριβή αυτή εξερευνούμε το NDP παράδειγμα για επεξεργαστές υψηλής απόδοσης αλλά και για επεξεργαστές χαμηλούς ισχύος. Όσον αφορά του επεξεργαστές υψηλής απόδοσης, προτείνουμε μία προσέγγιση στην οποία λαμβάνουμε υπ' όψη μας την εκτέλεση βρόγχων γενικού σκοπού. Η αρχιτεκτονική την οποία προτείνουμε κάνει χρήση μίας μεθοδολογίας χρονοδρομολόγησης εντολών, κατά την οποία η κάθε εντολή του βρόγχου εκδίδεται σε ένα ειδικά προσαρμοσμένο ολοκληρωμένο κύκλωμα που έχει τον ρόλο του επιταχυντή της εκτέλεσης του βρόγχου. Το κύκλωμα αυτό τοποθετείται στο λογικό επίπεδο μίας κύριας μνήμης υβριδικού κύβου (HMC). Στο επίπεδο αυτό οι εντολές εκτελούνται επαναληπτικά και παράλληλα, με έναν τρόπο που θυμίζει αυτόν της επικάλυψης λογισμικού, ενώ τα ενδιάμεσα παραγόμενα αποτελέσματα παροχετεύονται δια μέσου ενός δικτύου διασύνδεσης που βρίσκεται πάνω στο ολοκληρωμένο κύκλωμα. Όσον αφορά τις αρχιτεκτονικές χαμηλής κατανάλωσης ισχύος, αναπτύσσουμε μία καινοτόμο μεθοδολογία ανάλυσης χρονισμού, η οποία βασίζεται στις αρχές του STA και προσανατολίζεται συγκεκριμένα προς συστήματα χαμηλών προδιαγραφών και χαμηλής κατανάλωσης ενέργειας. Η μεθοδολογία αυτή λαμβάνει υπ' όψη της την διέγερση των διαδρομών χρονισμού της κάθε εντολής που υποστηρίζεται από το σετ εντολών του επεξεργαστή (ISA) και υπολογίζει την καθυστέρηση της χειρότερης περίπτωσης για την κάθε εντολή ξεχωριστά. Ως αποτέλεσμα, αντλούμε πληροφορίες για την χρονική καθυστέρηση σε επίπεδο εντολής και εκμεταλλευόμαστε την πληροφορία αυτή ώστε να κλιμακώνουμε την συχνότητα του ρολογιού δυναμικά, ανάλογα με τον τύπο εντολής που εκτελείται στο κύκλωμα σε κάθε χρονική στιγμή. Στην συνέχεια χρησιμοποιούμε την μεθοδολογία που περιγράψαμε για να συν-σχεδιάσουμε μία αρχιτεκτονική, με γνώμονα την δυναμική μεταβολή της συχνότητας του ρολογιού του επεξεργαστή η οποία

εκτείνεται στον βαθμό λεπτομέρειας του κύκλου μηχανής. Επικεντρωνόμαστε ξανά στην εκτέλεση κώδικα γενικού σκοπού και υλοποιούμε συνδυαστικά τη αρχιτεκτονική στο λογικό επίπεδο μίας μνήμης τύπου HMC ώστε να καταστήσουμε ικανό το σύστημα μας για εκτέλεση εντολών δίπλα στην μνήμη τυχαίας προσπέλασης. Επιλέγουμε να αξιολογήσουμε τις αρχιτεκτονικές που υλοποιήσαμε (της υψηλής απόδοσης αλλά και της χαμηλής κατανάλωσης ισχύος) σε επίπεδο υλοποίησης ολοκληρωμένου κυκλώματος σύμφωνα με τα πρότυπα της βιομηχανίας ώστε να ενισχύσουμε την εγκυρότητας της μεθοδολογίας μας. Τα αποτελέσματα τα οποία παίρνουμε υποδεικνύουνε μία μεγάλη αύξηση της απόδοσης του συστήματος όσον αφορά την επιτάχυνση της λειτουργίας του σε σύγκριση με την αρχική αρχιτεκτονική, ενώ η κατανάλωση ισχύος πέφτει σε πολύ χαμηλά επίπεδα.

Acknowledgements

I would like to express my sincere gratitude to Prof. George Dimitriu for his continuous support of my Ph.D study and related research, for his motivation and for his intelligent recommendations. His advices, counseling and analytical suggestions constitute for me not only a source of inspiration, but also play a major role on the concepts incorporated within this work. He also assisted me meticulously with the writing process of this dissertation and I think that the realization of this work would be very difficult if not impossible, if it weren't for his help.

I would also like to thank Prof. George Stamoulis for his thoughtful guidance through this period of time. He managed to impart remarkable knowledge and to elaborate on concepts which are more than important for the research process. He has made several scientific remarks and contributions to this thesis for which I am grateful. He also provided substantial supervision during the research and oversaw the writing process.

Besides Prof. George Dimitriu and Prof. George Stamoulis, I would like to thank the rest of my thesis committee: Prof. Nestor Evmorfopoulos, and Prof. Athanasios Loukopoulos, for their insightful comments, encouragement, but also for their remarks which motivated me to broaden my research subject and to explore different perspectives. My sincere thanks also goes to the other committee members for accepting the invitation.

Finally, I thank everyone who assisted, supported and encouraged me during this process of exploration, experimentation and hard work.

Contents

Declaration of Authorship iii

Abstract vii

Greek Abstract ix

Acknowledgements xi

1 Introduction 1
- 1.1 Introduction . . . 1
- 1.2 Contributions . . . 2
- 1.3 Outline . . . 3

2 Near data processing for high performance architectures 5
- 2.1 Introduction . . . 5
- 2.2 Related work . . . 5
- 2.3 Background . . . 7
  - 2.3.1 Hybrid memory cube organization . . . 7
  - 2.3.2 CGRA architecture . . . 8
- 2.4 NDP for general purpose applications . . . 8
  - 2.4.1 CGRA microarchitecture for general purpose instruction execution . . . 9
  - 2.4.2 Loop pipelining for the CGRA microarchitecture . . . 11
  - 2.4.3 Instruction issue for the CGRA . . . 11
  - 2.4.4 Loop execution on the CGRA . . . 13
- 2.5 Implementation . . . 15
  - 2.5.1 System architecture . . . 15
  - 2.5.2 NDP design space exploration and layout . . . 16
- 2.6 Experimental evaluation . . . 20
  - 2.6.1 Normalized speedup . . . 20
  - 2.6.2 Energy reduction . . . 23
  - 2.6.3 Power and area efficiency . . . 24
  - 2.6.4 Speedup improvement per Watt . . . 26
  - 2.6.5 Comparison with related works . . . 26

3 Timing analysis for low power pipelines 29
- 3.1 Introduction . . . 29
- 3.2 Related word . . . 30
- 3.3 Background . . . 32
  - 3.3.1 Static and dynamic timing analysis . . . 32
- 3.4 Timing analysis in processor datapaths . . . 32
  - 3.4.1 The instruction path exhaustive timing analysis concept . . . 32
  - 3.4.2 Dynamic opcode value changes compensation . . . 33
- 3.5 Clock scaling of RISC-V using IPE-STA . . . 35

3.5.1 Adaptive clock scaling in pipelined Processors . . . 35
3.5.2 Scaling clock by opcodes . . . 36
3.5.3 Dynamic Clock Scaling Mechanism . . . 38
3.6 Implementation . . . 40
3.6.1 RISC-V processor parameters . . . 40
3.6.2 CAD toolflow and simulation . . . 41
3.6.3 Clock tree synthesis . . . 42
3.7 Experimental evaluation . . . 43
3.7.1 Normalized speedup . . . 43
3.7.2 Normalized power consumption . . . 44
3.7.3 Overhead of the IPE-STA methodology . . . 45
3.7.4 PVT tolerance considerations . . . 46

4 Near data processing for low power architectures 49
4.1 Introduction . . . 49
4.2 NDP System Architecture . . . 49
4.2.1 Host system architecture . . . 49
4.2.2 PIM core architecture . . . 50
4.3 BTWC-NDP co-design methodology . . . 53
4.3.1 Application of IPE-STA to the PIM core . . . 53
4.3.2 PIM core microarchitecture with adaptive clock scaling . . . 54
4.4 Implementation . . . 56
4.4.1 Design space exploration and parameter considerations . . . 56
4.4.2 CAD toolflow and simulation . . . 57
4.4.3 Adaptive clock scaling with multiple clocks . . . 57
4.4.4 Area and power budget . . . 58
4.5 Experimental evaluation . . . 58
4.5.1 Workload characterization . . . 58
4.5.2 Normalized speedup . . . 59
4.5.3 Normalized energy reduction . . . 61
4.5.4 Energy efficiency . . . 61
4.5.5 Area efficiency . . . 62

5 Conclusions and Future Directions 65
5.1 Conclusions . . . 65
5.2 Future Directions . . . 66

A Relevant Publications 67
Relevant Publications . . . 67

Bibliography 69

List of Figures

2.1 Architecture diagram of HMC DRAM. 8
2.2 The proposed CGRA grid architecture deployed on the logic layer of the HMC DRAM. 9
2.3 PE microarchitecture. 11
2.4 The loop pipelining optimization. 12
2.5 The outcome of the instruction issue process after which each operation is issued on a PE. 14
2.6 Instruction execution instance on CGRA microarchitecture. 15
2.7 System architecture for general purpose NDP. 16
2.8 The speedup improvement of the proposed NDP implementations for each kernel normalized to the host processor execution time. 21
2.9 Host processor-HMC data transfer reduction and its contribution to the normalized speedup of each NDP implementation. 22
2.10 Normalized energy reduction of the NDP methodology. 23
2.11 Energy consumption breakdown of NDP implementations. 24
2.12 Normalized power efficiency of the NDP implementations. 25
2.13 Normalized area efficiency of the NDP implementations. 26
2.14 Speedup per Watt of each NDP implementation. 27

3.1 An instruction execution instance of the Rocket core implementation displaying the minimum operational clock period during each stage. 38
3.2 The clock control unit integrated in the rocket core. 39
3.3 Unstable clock behavior due to subsequent clock selections. 39
3.4 The clock instability compensation technique. 40
3.5 The configuration parameters of both processor implementations. 41
3.6 The CAD toolflow for the IPE-STA methodology. 42
3.7 Normalized throughput improvement and critical instruction appearance rate of the proposed design methodology compared to the corresponding baseline processors. 43
3.8 Normalized power consumption increase of the proposed methodology compared to the baseline processors. 45

4.1 The host system architecture composed by the BOOM core and the PIM preprocessing pipeline. 50
4.2 The PIM core architecture 51
4.3 An instruction execution instance of the PIM core depicting the minimum operational clock period during each stage. 55
4.4 The CCU implemented on the PIM core in the HMC logic layer. 55
4.5 Toolflow of the IPE-STA methodology for the PIM core 57
4.6 Normalized speedup of each PIM core implementation over the baseline RISC-V pipeline. 59
4.7 Impact of different design techniques on each PIM core speedup factor 60

4.8 Normalized energy reduction of each PIM core implementation over the baseline RISC-V pipeline. 61
4.9 Energy efficiency of the PIM core and RISC-V core pipeline implementations 62
4.10 Area efficiency of the PIM core and RISC-V core pipeline implementations . 63

List of Tables

2.1 Key parameters of the host processor die and of the HMC implementations. . 17
2.2 PE and CE post-layout requirements in terms of area, power and latency. . . . 18
2.3 Implementation parameters of 5 different NDP designs. 18
2.4 Workload characterization. 20
2.5 Comparison of the proposed NDP architectures with the current state of the art. 27

3.1 Analysis of the instruction classes of the RISC-V Rocket core architecture. . 37
3.2 The clock periods for critical instructions along with the typical clock period for the Rocket core implementation. 37
3.3 Throughput improvement comparison between Application-adaptive guard-banding and IPE-STA. 44
3.4 Throughput improvement comparison between Blueshift OpenSPARC, Razor and IPE-STA methodology. 44
3.5 The power and area overhead of the clock control and instruction snooping circuits in comparison to RiscV 45
3.6 Area overhead comparison between IPE-STA methodology and the state of the art. 46
3.7 Time requirements of DTA, STA and IPE-STA to complete the timing analysis of RiscV pipeline. 46

4.1 Instruction class IPE-STA analysis of PIM core architecture for different supply voltages 54
4.2 NDP design parameters 56
4.3 Area and power requirements of the RISC-V and PIM core implementations . 58
4.4 Workload characterization 59

List of Abbreviations

BTWC	**B**etter **T**han **W**orst **C**ase
TS	**T**iming **S**peculation
PVT	**P**rocess **V**oltage **T**emperature
ISA	**I**nstruction **S**et **A**rchitecture
STA	**S**tatic **T**iming **A**nalysis
DTA	**D**ynamic **T**iming **A**nalysis
IPE-STA	**I**nstruction **P**ath **E**xhaustive STA
DRAM	**D**ynamic **R**andom **A**ccess **M**emory
PIM	**P**rocess **I**n **M**emory
NDP	**N**ear **D**ata **P**rocessing
TSV	**T**rough **S**ilicon **V**ia
HMC	**H**ybrid **M**emory **C**ube
CGRA	**C**oarse-**g**rained **R**econfigurable **A**rray
PE	**P**rocessing **E**lement
CE	**C**ontrol **E**lement
OoO	**O**ut **o**f **O**rder
SSD	**S**olide **S**tate **D**rive
ILP	**I**nstruction **L**evel **P**arallelism
LP	**L**oop **P**ipelining
RaW	**R**ead **A**fter **W**rite
RF	**R**egister **F**ile
MMU	**M**emory **M**anagement **U**nit
VC	**V**ault **C**ontroller
SoC	**S**ystem-**o**n-**C**hip
IPC	**I**nstruction **P**er **C**lock
HPC	**H**igh **P**erformance **C**omputing

Dedicated to everyone who assisted me in translating this research conception into an engineering realization.

Chapter 1

Introduction

1.1 Introduction

Traditional processors employ the standard Von Neumann architecture which separates the core pipeline from the main memory, while the communication between such entities is handled through an off-chip bus interface. This approach has been adopted by both the academia and industry and it has evolved into a well established paradigm that drives the contemporary design methodologies of integrated circuits. Furthermore, the continuous shrinking of transistor size seems to confirm Moore's law which indicates that the number of transistors in a dense integrated circuit will double about every two years. Moore's Law, coupled with Dennard scaling [1] has led to a stable increase in the processor core count and to proportional performance scaling over the last 20 years. Consequently, the resulting computer architecture paradigm focuses on multi-core or many-core systems, on pipeline optimizations and on parallelism exploitation to drive the evolution of the market processors.

As Dennard's scaling stops mainly due to supply voltage limits and leakage power drain, power densities rapidly increase on the chip and benefits of multi-core scaling begin to abate well before we hit the physical manufacturing limits. Therefore the industry races down the multi-core path and the dark silicon phenomenon [2] emerges. Under the dark silicon effect, the cores in a multi-core processor cannot be functional at the same time due to the lack of energy efficiency and thus, the performance scaling of the CPUs begins to decelerate. As the essential question of "how much more performance can be extracted from the multi-core path in the near future" [2] is formulated, latest research concludes that large performance workload variations are to be expected in present-day computer architectures [3] [4]. Such inherent variations may lead to unpredictable and sub-optimal performance in tightly coupled applications.

Nowadays computer architecture research suggests that the Von Neuman model begins to realize its upper limit in terms of power and throughput, while the performance bottleneck is shifting away from the core pipeline to the DRAM [5]. More specifically, the long DRAM latency and the costly data transfer between the CPU and the DRAM severely affect the energy consumption and throughput of computing systems. In order to alleviate such a scaling wall, researchers have proposed the adoption of the NDP paradigm in which the computations are moved near the DRAM silicon die. NDP promises to decrease the CPU-DRAM transactions and thus, to minimize the energy consumption of the system while also maintaining a very high throughput compared to the core pipeline.

In this work, we explore the premises and prospects of the NDP computing paradigm in both HPC and low power computing domains. To this end, we propose a novel high performance NDP design for general purpose applications that executes instructions on the logic layer of a 3D-stacked DRAM. Under this premise, we deploy a network of functional units within a mesh interconnection network in conjunction with an instruction scheduling methodology which is tailored to minimize the dark silicon phenomenon. To achieve this, we focus our efforts on a high functional unit utilization rate and low power constraints,

both of which diminish the dark silicon effect. The proposed design methodology issues one instruction on each functional unit and propagates the intermediate results through the mesh interconnection network. In this sense, loop execution resembles the software-pipelining paradigm, as each loop instruction executes iteratively in a dedicated functional unit near the DRAM. As a result we manage to obtain a theoretical throughput of one loop iteration per clock cycle while avoiding the costly CPU-DRAM data transactions. We evaluate our methodology by conducting a detailed design space exploration in a post-layout simulation environment using the standard industry CAD toolflow. Results indicate a speedup of 42x and an energy reduction of 22.4x over a high performance OoO CPU.

Considering the NDP for low power computer architectures we shift our focus on low-end small pipelines which are designed for reduced energy consumption. In order to efficiently implement a low-power NDP system we develop a novel timing methodology specifically targeting low-end pipelines and we explore a co-design approach between the proposed technique and the NDP paradigm. The rationale behind the low-end pipeline consideration is supported by the fact that in the Internet of Things era, the low-end processor domination of the embedded market is expected to be further reaffirmed. Then, a question will arise, on whether it is possible to enhance performance of such processors without the cost of high-end architectures. To this end, this part of our work focuses on frequency scaling and the study of related techniques which can boost processor performance with a low cost. Frequency scaling has been recently used in conjunction with the BTWC processor design paradigm. Traditional designs operate under worst-case timing constraints which avert incurring execution errors. However, such constraints impose heavy performance penalties. The BTWC designs are allowed to operate above their critical levels, but expensive error-correction hardware must then be incorporated, to fix any possible errors. Under this premise, we propose a BTWC methodology which enables the processor pipeline to operate at higher clock frequencies compared to the worst-case design approach. We employ a novel timing analysis technique, which calculates the timing requirements of individual processor instructions statically, while also considering the dynamic instruction flow in the processor pipeline. Therefore, using an appropriate circuit that we designed within this work, we are able to selectively increase clock frequency, according to the timing needs of the instructions currently occupying the processor pipeline. In this way, the error-free instruction execution is preserved without requiring any error-correction hardware. In the sequel we apply the proposed timing methodology on a low-end, low-power architecture that is implemented on the logic layer of an HMC DRAM. We design the system pipeline from the ground up to support the frequency scaling mechanism and we make the necessary modifications on the host system to facilitate a pre-processing pipeline that minimizes the power overhead of the NDP processing. Results demonstrate an average speedup factor of 23x with 12x reduction in energy consumption compared to the baseline implementation. We also show that the proposed methodology produces at least 9x times more energy efficient and 24.5x times more area efficient designs and we conclude that it significantly enhances the overall performance of the low-power NDP implementations.

1.2 Contributions

The main contributions of this work to the current state of the art are considered as follows:

- We expand applicability of the NDP paradigm by designing and implementing an HPC NDP system for general purpose loop execution. In particular, we design the control and the processing elements as well as a forwarding unit that speedups the execution of general purpose loop bodies. We employ such a general purpose approach to allow any

type of application to exploit the NDP technology, instead of focusing on an application specific methodology as most of previous works do.

- We provide an NDP instruction issue methodology which issues each loop body instruction on a single functional unit. In this way each functional unit executes one instruction iteratively until the loop execution completes and thus, the NDP throughput is maximized. We also make sure that the whole loop can be mapped on the functional units by optimizing the efficiency of the issue process.
- We propose a novel timing analysis methodology implemented on the circuit level which considers instruction opcodes for performance increase. Previous consideration of opcodes for performance increase has been compiler-only consideration or through the DTA technique.
- Our BTWC methodology accurately identifies the timing requirements of any incoming instructions. Since we are a priory aware of such constraints we do not deploy any error detection or error correction mechanisms and thus, the hardware implementation costs are significantly reduced.
- We co-design and implement an NDP architecture for low-end processors from the ground up, capable of facilitating the aforementioned methodology. We design the NDP architecture considering the processor low-power and low-area requirements, while also providing hardware support for the proposed timing analysis.
- We explore an NDP architecture-oriented approach to the BTWC design paradigm. Our work studies the NDP pipeline architecture to extract timing information based on the ISA of the processor. We consider this approach to have greater applicability, as it can be used on any processor without requiring to adjust or change the premise of our technique.

1.3 Outline

The following chapters of this dissertation are organized as follows. In chapter 2, we present our methodology for the HPC NDP architecture and we provide a detailed design space exploration in order to verify and evaluate the proposed technique. In chapter 3, we introduce the concept of timing analysis for low-end pipelines and we elaborate on the timing analysis approach we employ. In chapter 4 we discuss our co-design perspective between the NDP and the proposed timing analysis technique, and we provide the evaluation of the implemented architectures. Finally, chapter 5 concludes our work.

Chapter 2

Near data processing for high performance architectures

2.1 Introduction

Modern processor throughput encounters a scaling wall as the performance bottleneck has shifted from the core pipeline to the DRAM data transfer operations. To address such issue, researchers proposed the adoption of the NDP paradigm, in which the computations are moved near the DRAM silicon die. Nowadays there is a growing trend towards NDP research that focuses on application specific accelerators, while the general purpose paradigm has received relatively little attention so far. In this chapter we propose an NDP methodology for general purpose applications that targets high performance architectures with complex pipelines and complicated forms of control. Our design is optimized for loop acceleration under the general purpose computing paradigm, in order to cover a wide range of application types. In this sense, we do not employ NDP for application specific execution as previous works do, instead we focus on moving general purpose loop computations to the DRAM from the host system. The proposed architecture consists of a mix of PEs capable of executing simple arithmetic operations, CE that evaluate control statements and a mesh interconnection network that handles the communication between the deployed PEs. Typical CGRAs employ dynamic switching mechanisms that manage the flow of information within the mesh interconnection and thus, enabling the forwarding process between individual PEs. Our design diverges from the standard CGRA approach, as we opt for a scheduling methodology that issues each loop instruction on a single PE, by employing the loop pipelining optimization. Under this premise, the forwarding processes is not dynamically managed, instead it is statically configured to activate the forwarding streams that resolve the instruction data dependencies as they eventuate after the scheduling process completes. As a result, our design leverages the CGRA dataflow execution, but also utilizes a loop acceleration technique to further improve throughput and reduce the energy consumption. In order to verify our methodology we utilize a post-layout netlist of RISC-V OoO BOOM core and a post-layout implementation of a HMC DRAM architecture. Post-layout simulations on the NDP design demonstrate an average speedup factor of 42x, while the energy consumption is reduced by a factor of 22.4x over the host CPU execution.

2.2 Related work

Modern computer architectures are bound by energy constraints that impose heavy penalties on the system's throughput scaling capabilities. Such constraints derive not only from the high energy requirements of temporary storage circuits such as register files and cache memories, but also from the energy cost of data transfer operations to the DRAM, as previous work in [6] demonstrates. In order to alleviate such limitations, researchers have proposed

to execute the computations closer to the data, i.e. to the DRAM. The concept of NDP has been explored in the past, as previous work in [7] shows, but due to technology limitations the applicability of such an approach was limited. Nowadays a resurgence in NDP research is observed mainly due to the appearance of TSV interconnections and 3D stacked memories, which are key enablers for the NDP concept, as mentioned in [7]. Despite the fact that previous work in [8] suggests that the adoption of NDP model can be accomplished without requiring 3D stacked memories, most of the ongoing research focuses on TSV and 3D RAM technologies to deliver near data computations.The NDP promises to alleviate the performance bottleneck imposed by the DRAM bus in modern processors as a previous survey in [9] elaborates and also to increase the performance-to-power ratio of modern processing systems [10]. In [9] researchers argue the existence of a performance wall due to the slow RAM - CPU communication and mention that such a wall leads to performance scaling problems. Although NDP seems a promising solution to the aforementioned scaling problem, there are some challenges that should be addressed, in order for this paradigm to be adopted in the industry. Such challenges include the optimization of NDP architectures, the memory coherence maintenance and the perseverance of the sequential programming model, as previous works in [7] [9] elaborate.

Under this premise, researchers mainly focus on a certain application type to perform NDP optimizations as previous work in [11] demonstrates. In that work authors deploy a near data graph processing accelerator achieving high throughput and reducing the power consumption, while also maintaining memory coherence by defining non-cachable memory spaces. Another work in [12] focuses on bitwise operation execution in memory, while in [13] authors propose an NDP framework for IoT applications. Big data applications are also suitable for NDP due to the increased memory bandwidth requirements they exhibit [14]. Authors in [15] use a HMC architecture to map computing kernels inside the memory network, leveraging in-network computing in data-flow style. CGRAs have also been proposed for the processing elements in NDP systems as in [16]. In that work researchers deploy a network of functional units in a mesh like structure for NDP applications while the proposed solution does not require any micro-architectural changes to the host processor. In other related work [17] authors incorporate heterogeneous reconfigurable logic arrays, which behave like CGRAs in order to improve throughput and reduce the power consumption of target applications. CGRA capabilities are also explored in [18] along with different TSV interconnection networks in order to find the optimal CGRA-TSV combination that leads to the higher speedup improvement. A common target application of CGRAs and NDP is the training and inference of deep neural networks as previous works in [19] and [20] demonstrate. In [19] authors deploy RISC-V cores in a mesh-like structure on the logic layer of an HMC, to provide a scalable architecture for deep neural network training and inference operations. Similarly, in [20] authors explore neural network optimizations to reduce the data transfer rates between the NDP cores and a non-volatile memory system, while also exploiting the parallel execution capabilities of an on-chip processing network to reduce training time. Non-volatile memory structures are considered ideal for hosting NDP processing due to their ability to behave both as storage and as processing units. Previous work in [21] proposes a query acceleration methodology that also reduces the energy consumption per operation on a non-volatile memory. Another work in [22] explores the application of the NDP framework for database operations in SSDs by employing data flow programming features and code reuse optimizations, while in [23] authors employ a near-memory hash table to speed up search operations in databases for big data.

Memory consistency and coherence is a major challenge for NDP systems. While some works declare certain memory spaces as non-cachable, like [11] mentioned above, others propose ways of maintaining memory coherence. In particular, in [24] authors combine a speculative cache coherence and a compressed signature technique, in order to reduce the

overhead of DRAM coherence maintenance with the host system. On the other hand, however, researchers in [25] argue that non-restricted memory regions can often lead to a significant amount of coherence traffic, and thus, a good practice would be to avoid cachable memory blocks.

The concept of programming transparency is explored in [26], where researchers build a GPU framework for NDP that automatically selects the computations to be offloaded to NDP cores, while mapping the instructions on the available processing elements. In this way, programmers are not involved in the offloading procedure and the sequential programming model is maintained. Similarly, in [27] authors exclude the programmer from making decisions on near data execution, instead an automatic mechanism migrates the computations to NDP units, according to the observed levels of data locality. Also in [28] an NDP solution is proposed for heterogeneous systems that provides application transparency, while utilizing the processing power of the SSD devices for NDP operations.

To the best of our knowledge, there is no such a work proposed so far. Previous works in [11] [12] [13] [14] [19] [20] [21] [22] and [23] authors propose NDP frameworks for specific application types, and thus their designs take into account the requirements of the corresponding applications, while our work focuses on general purpose loop acceleration. In [10] [15] [26] [27] [28] [29] and [30] authors propose NDP architectures for general purpose applications but their approach does not include CGRAs, instead they rely on processor pipelines and task-specific accelerators to increase the system performance. In [31] researchers focus on diverse kernel workloads, but their CGRA network design is based on the profiling results of the executing kernels. On the contrary, our work does not require any profiling operation prior to code execution due to the fact that the CGRA is designed for loop acceleration and thus, it can support any issued loop without additional effort. Further, authors in [8] [16] [17] and [18] utilize CGRAs in conjunction with the NDP paradigm but their focus shifts to different aspects of the NDP execution paradigm. Under this premise, previous works lack the application mapping approach or the loop acceleration focus we employ as they do not utilize the CGRA network to execute instructions in an iterative way, i.e. to issue an instruction per processing element. To this end, they rely on run-time reconfiguration techniques [17], memory data buffer exploitation [8] and exploration of TSV interconnection microarchitectures [16] [18] to enable the NDP processing and to increase the throughput of the system.

2.3 Background

2.3.1 Hybrid memory cube organization

HMC architectures are widely adopted by NDP paradigms as previous work in [32] shows, mainly due to the 3D-stacked DRAM layers they employ. An HMC DRAM consists of multiple DRAM layers and achieves much greater internal bandwidth than conventional DRAMs, due to the TSVs, which are vertical links that connect the multiple layers of a DRAM stack together. Figure 2.1 depicts the architecture of a HMC DRAM as described according to the HMC specification in [33]. HMC is organized in vertically structured memory vaults which consist of partitions. Each partition contains a number of banks which store the DRAM data. The lower DRAM layer is reserved for implementing logic and facilitates vault controllers that manage the data transfer process between the corresponding vaults. Vault controllers also internally handle the refresh operations of each vault, removing this responsibility from the host memory controller. In order to support high BW within its vaults, HMC utilizes TSVs that connect vertically the vault layers and redirect data to the vault controllers. In this work we employ the NDP paradigm and thus, we also adopt the aforementioned HMC hierarchy.

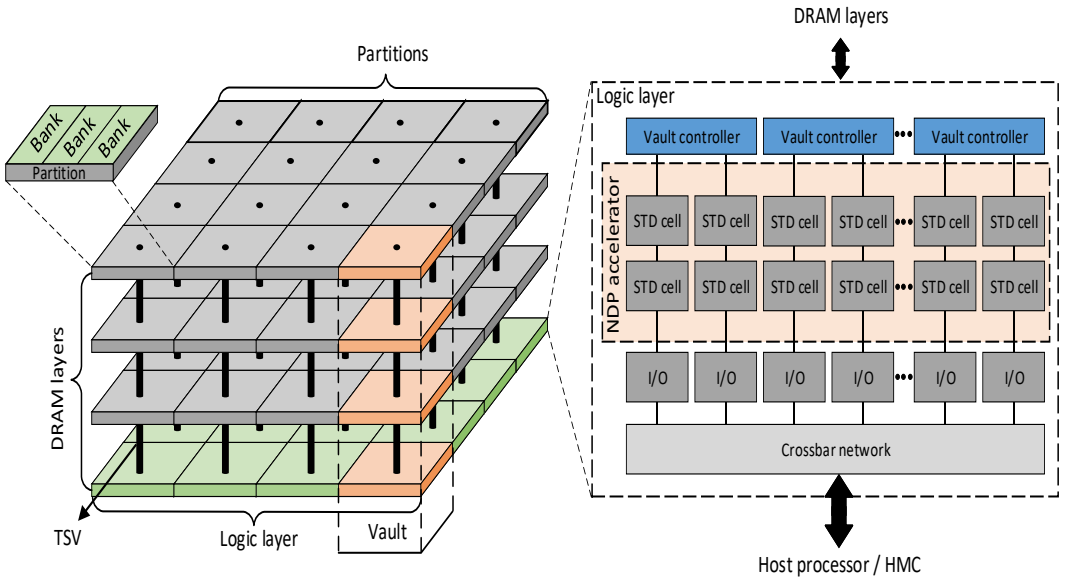

FIGURE 2.1: Architecture diagram of HMC DRAM.

2.3.2 CGRA architecture

CGRAs are frequently employed as hardware accelerators due to the increased performance to area ratio they achieve [34]. A typical CGRA grid consists of a number of PEs arranged in a 2D array structure. The processing elements are designed to execute basic arithmetic or logical operations while their communication is managed by an interconnection network. Previous work in CGRAs usually focuses on targeted applications such as Givens Rotation acceleration [35], convolution neural network applications [36], deep neural networks [19] and wireless telecommunication receiver algorithms [37]. Such designs are proven to be very efficient in terms of performance to area ratio, but their operation is restricted within specific application types. In contrast, CGRA related work that treats them as general purpose architectures often focuses on the instruction mapping procedure of the accelerator [38]. As a result, CGRA architectures tend to exploit local data reuse [39], utilize a local register file of the PEs [40] and organize the instruction mapping process in an efficient way [41].

Despite the fact that such works deal with the routing and resource mapping problems of a CGRA network they employ complex scheduling algorithms which are not suitable to the NDP paradigm due to the restricted area and power requirements of NDP designs. Also when general purpose code execution is considered, the data forwarding and pipeline stalling tasks should be taken into consideration as general purpose applications usually display a great amount of data dependencies. Additionally general purpose code requires a lot of control statements which should be taken into account in order for the CGRA design to be efficient. We consider such requirements critical for designing the CGRA architecture and thus we incorporate them in the design process. Under this premise we design a novel CGRA architecture from the ground up capable of supporting near data general purpose loop execution. To this end, we opt for a static-scheduling static-dataflow (SSD) execution [42] in which each loop instruction is statically scheduled on a corresponding processing element while the execution process leverages the dataflow parallelism of the corresponding loop. We discuss the micro architecture details of such an architecture in the following section.

2.4 NDP for general purpose applications

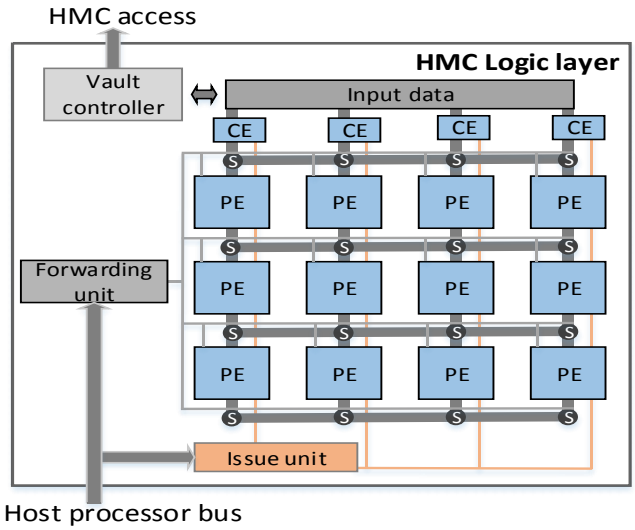

FIGURE 2.2: The proposed CGRA grid architecture deployed on the logic layer of the HMC DRAM.

2.4.1 CGRA microarchitecture for general purpose instruction execution

The proposed CGRA microarchitecture focuses on adopting the general purpose execution paradigm, while also leveraging the dataflow parallelism offered by the CGRAs. Figure 2.2 depicts the architecture of a 4×4 CGRA grid implemented on the logic layer of the HMC DRAM. The array consists of multiple PEs and CE organized in a mesh-like structure, and of an on-chip interconnection network capable of handling the communication among the PEs. Each PE is capable of executing arithmetic or logical operations and utilizes inputs either from the HMC DRAM or from the outputs of other PEs. Each CE is responsible for evaluating control statements that change the execution flow of the instruction sequence. The flow of data between the HMC and the CGRA is controlled by the vault controllers, which execute memory requests heading to the HMC DRAM. In order to manage the data transfer within the CGRA grid, we utilize switches that are designed to redirect data paths to designated PEs and CEs.

We will now discuss the microarchitecture of the units deployed on the CGRA grid. Since this work focuses on loop acceleration for general purpose execution, instructions scheduled on the CGRA units are assumed to be loop instructions. More specifically, the instructions scheduled constitute the pipelined body of a loop and are not changed until all loop iterations are finished. Our detailed model for loop pipelining and acceleration is presented later on.

Issue unit: The issue unit is responsible for the instruction issue process to the PE-CE grid. In order to avoid the power and area costs of a complex instruction issue circuit, we perform the scheduling process on the host processor as part of a pre-processing stage. The outcome of this process is transferred to the NDP architecture and specifically to the issue unit, which propagates the instruction operation codes and inputs to the corresponding PEs.

In the proposed NDP methodology, each PE is assigned multiple instructions but it may execute only one at any given time. In this sense the amount of the instructions that can be executed simultaneously is bound by the total amount of PEs-CEs. We take such a constraint into consideration when designing the issue unit, in order to maximize the amount of issued instructions along the CGRA, while also maximizing exploitation of ILP in instruction execution. Under this premise, we design the unit to schedule instructions, taking into account the control statements that disrupt the sequential code execution, such as if-else branches and nested loop iterations. Such statements always prohibit certain instructions from being executed, after the evaluation of the branch conditions takes place. For example a control

statement enables the execution of the instructions in the if body or in the $else$ body, but does not allow the execution of both if and $else$ body sequences simultaneously. The same rule applies to the nested loop iterations as well. In order to exploit ILP in such cases, we design the issue unit to issue mutually exclusive instructions on the same PEs. Under this premise, a PE may execute an instruction that belongs to an if body, but also the same PE may execute an instruction that belongs to the side of the $else$ body as well. Which instruction is actually executed depends on the evaluation of the branch condition. In this way, we not only allow more parallel instructions to be issued on the grid, but we also decrease the area requirements for our design, as we are able to issue more instructions in the same PE-CE die area.

Forwarding unit: The forwarding unit manages the forwarding and stalling processes by generating the necessary signals that propagate to the corresponding switches and PEs-CEs. After the issue process is complete, the data hazards between the PEs-CEs are known when the loop instructions are assigned to the corresponding PEs-CEs. Thus, the forwarding unit forwards the corresponding data to the appropriate CEs-PEs in order to eliminate data hazards. To accomplish this task, it generates control signals that propagate to the switching elements which act as multiplexors for the incoming data. As instructions and thus data dependencies do not change over consequent loop iterations, this process needs to be conducted at the beginning of the loop execution in order to open the corresponding data forwarding paths. Such data paths may change during the run time only when control statement evaluation results in a redirection of the instruction execution sequence flow. In this case the forwarding unit propagates the corresponding control signals in order to forward the required data to the dependent CEs-PEs. As a result, any unnecessary switching signals are omitted and dynamic power consumption is reduced. The forwarding unit also generates stall signals that freeze the instruction execution on specific PEs-CEs when necessary.

Processing element: The microarchitecture of a PE is depicted in Figure 2.3. Each PE is composed of an ALU or FPU unit capable of executing arithmetic or logical operations and two multiplexors that control the unit's input operands. Such inputs may originate from the DRAM, from other PE outputs or from the output of the same PE, depending on the data dependencies of the executing loop. To resolve such dependencies we utilize the forwarding unit which selects the appropriate inputs for each PE as described above. Each PE is assigned multiple instructions, but it may execute one operation per clock cycle. We modify the PEs so that the output of such an operation is temporarily stored to a fifo queue before propagated to the PE network. In the case of a stall the forwarding unit evokes the write privileges of the ALU/FPU output to the queue and thus, no new entries are stored. Each PE also contains a small operation buffer (OP buffer) for storing the assigned instructions for execution. As described above, each PE is assigned a number of instructions which cannot be executed simultaneously, for example different branches of an if-else statement. Such instructions are propagated through the issue unit to the PE network and are stored in the OP buffer of the corresponding PEs. In the sequel, the CEs control which operation the PEs will execute for each clock cycle via a multiplexor, depending on the evaluation of the branch condition.

Control element: We design the CEs dedicated to the execution of branch related instructions. We opt to implement dedicated units for executing comparison operations due to the high amount of control statements that are usually present within general purpose binaries. As a result, we consider the execution of such instructions in the PEs as inefficient due to the larger amount of power and area requirements of the PEs. To this end, each CE is a small unit capable of comparing two inputs and producing an output that reflects their relative value status. The microarchitecture of CEs is similar to the PEs as depicted in figure 3.3 with two distinct differences. First of all the ALU/FPU is replaced by a small comparator circuit which performs a comparison operation according to the assigned opcode, i.e. "greater than",

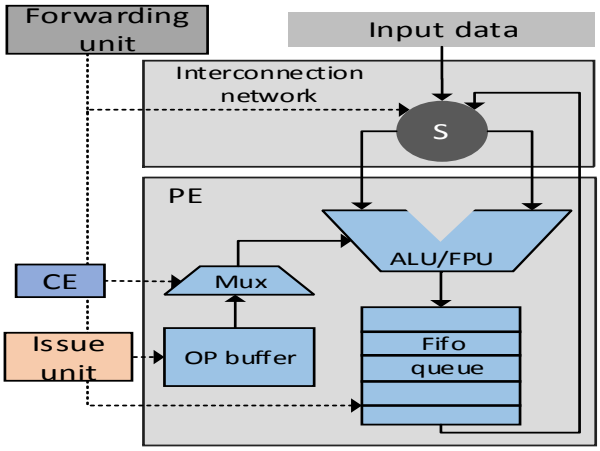

FIGURE 2.3: PE microarchitecture.

"less then" ,"equal" etc. Further, the fifo queue of each CE is smaller than the corresponding fifo queues of the PEs as the outputs of the CEs are not 32-bit or 64-bit wide. Also the comparison results which are stored in the fifo queue are forwarded to PEs, where they act as control signals for the multiplexors, in order to choose which instruction to execute from the OP buffer. As a result, the CE size and power consumption is significantly lower that the PE's and thus, we are able to incorporate more of them in the CGRA implementation.

2.4.2 Loop pipelining for the CGRA microarchitecture

LP of software pipelining is a loop optimization technique that improves the performance of loop execution by overlapping consequent loop iterations and thus, hiding the underlying latency of each iteration. This form of pipelining is widely used as a compiler optimization technique. Previous work in [43] demonstrates that such a technique can also be efficiently mapped on CGRAs resulting in an drastic decrease in loop execution time and power consumption [44]. Previous work in [43] demonstrates that LP can also be efficiently mapped on CGRAs resulting in an drastic decrease in loop execution time and power consumption [45]. LP achieves such results by transforming the loop body in a way that hazards caused by the data dependencies between participating instructions, RaW are reduced. Loop data dependencies come into two forms, intra-loop and inter-loop. Intra-loop are data dependencies that appear within the same loop iteration, while inter-loop dependencies span from one iteration to another. Figure 2.4 depicts an example of software pipelining transformation for a simple loop that contains intra-loop RaW data dependencies. In this example the loop consists of four instructions: a memory operation instruction (*mem*) with 2 cc latency, an addition operation instruction (*add*) with 1cc latency, a multiplication operation instruction (*mull*) with 2cc latency and a subtraction operation instruction (*sub*) with 1cc latency. We use the variable x to denote the dependencies of each instruction within the iteration x. To this end, the *add* instruction depends on the execution of *mem*, the *mull* on the *add* and the *sub* on the *mull* and *add* as they process the same data space. In order to alleviate such dependencies, LP constructs a loop steady state so that it contains instructions from future iterations and may iterate with a reduced number of internal data hazards. The loop steady state is accompanied by a prologue which manages the setup process and an epilogue that waits for the remaining instructions to finish.

2.4.3 Instruction issue for the CGRA

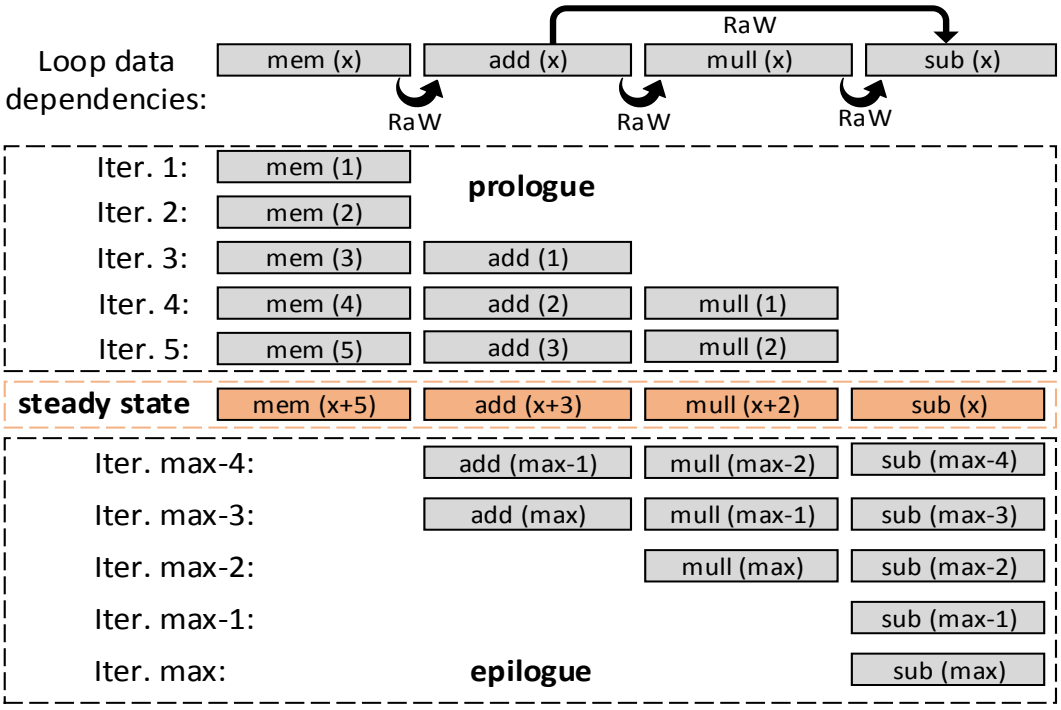

FIGURE 2.4: The loop pipelining optimization.

Algorithm 1 The proposed instruction issue technique for the CGRA.

perform LP
while not converged **do**
  if Instr. No. $<$ Issue slots **then**
    unroll the loop
  else if Instr. No. $>$ Issue slots **then**
    roll back the loop
  else if Instr. No. $==$ Issue slots OR tries $>$ threshold **then**
    converge
  end if
end while
Generate control dependency graph
Form mutually exclusive execution groups
Generate data dependency graph
for mutually exclusive groups **do**
  get mutually exclusive instructions
  map them in PEs OP buffers
end for
for rest of non-control instructions **do**
  get data depended instructions
  issue them in neighboring PEs
end for
for control instructions **do**
  get data depended control instructions
  issue them in neighboring CEs
end for

Although the idea of the LP is not new, the mapping of LP to hardware accelerators is not a trivial process, as previous work in [46] demonstrate. In this work we focus on improving the throughput and the area efficiency of the CGRA grid and thus, we opt for an instruction issue methodology that takes advantage of the LP optimization and the PE-CE network presented in Section 2.4.1. We employ LP as it does not only reduce the RaW data hazards of the loop body, but it also generates a self contained loop steady state which may be efficiently scheduled. The proposed CGRA architecture consists of a grid of processing and control elements, with each PE containing an OP buffer. In this sense, each PE is assigned a number of instructions equal to the OP buffer size, provided that operations can be executed in a mutually exclusive way. Thus, each PE may execute only one instruction at any given time, but it may have been issued more. The total number of instruction issue slots equals to *number of PEs* $\times$ *OP buffer size*.

Algorithm 1 depicts the heuristic method we employ for mapping loop instructions onto the CGRA grid. We first perform LP optimization in order to minimize the data hazards from RaW dependencies, which are very common in general purpose loops. In the sequel, we analyze the number of instructions located in the steady state. If their number is less than the total number of free issue slots, we perform a loop unrolling operation. Loop unrolling is a loop optimization technique, in which the steady state of the loop is increased in size by adding instructions from future loop iterations. In contrast with LP, loop unrolling replicates the loop body and modifies the replicated instructions in order to refer to future iterations. In this way we increase the number of instructions that can be issued to the PEs while also reducing the execution time as we simultaneously execute instructions from future iterations. On the contrary if the amount of the steady state instructions is larger than the free issue slots, the loop cannot be issued and thus we reduce the steady state's size by rolling back the loop to a previous valid state. We repeat this process until the heuristic converges either by equalizing the steady state size with the number of issue slots or by reaching a preset repetition limit. In the sequel we generate the control dependency graph for the steady state of the loop. Such graph classifies the participating instructions into groups that can be executed in a mutually exclusive way. We issue instructions from the same groups, i.e. operations that are not executed in a mutually exclusive way to different CEs-PEs while instructions from different groups are issued on the same PEs provided that the corresponding OP buffers are not full.

The issue process also takes into consideration the data dependencies of the scheduled instructions. To this end, we generate the data dependency graph of the steady state instructions which provides information on the RaW dependencies of each operation. We prioritize the issue of instructions with data dependencies on neighboring CEs-PEs in order to reduce the data routing delays. If some instructions cannot be mapped to neighboring CEs-PEs due to the OP buffers being full, we map them as close as possible with the depended operation. After the PE scheduling completes the control statements of the loop are issued on the CEs . The scheduling procedure of the control instructions follows the greedy method for simplicity reasons. To this end, each control statement is issued on a single CE while taking into account the data dependencies between the data and control instructions in the same way as described above.

2.4.4 Loop execution on the CGRA

By employing the aforementioned methodology we manage to efficiently map the loop instructions on the CGRA network. Figure 2.5 depicts the outcome of the issue process after its application on the instruction sequence used in figure 2.4. The number notated by *i1* refers to the *mem* operation instruction, the *i2* to the *add*, the *i3* to the *mull* and the *i4* to the *sub* instruction. Data dependent instructions are issued on neighboring PEs and the forwarding

unit generates the appropriate control signals, to conduct the data forwarding on the corresponding PEs. Following the data dependencies established in figure 2.4, PE-1 propagates its outputs to PE-2, PE-2 to PE-3 and PE-4, while PE-3 forwards the produced data to PE-4. The rest of the PEs are marked with gray color, since they are not assigned any instructions in this scenario. Under this premise, after the loop execution begins each PE iteratively executes one instruction that belongs in the loop body and thus, the CGRA produces the outputs of one loop iteration per clock cycle when the pipeline is filled.

Figure 2.6 depicts a run time instance of this loop after being issued on the CGRA grid. The data dependency constraints would normally dictate that instruction *i4* should wait for both *i2* and *i3* to finish their execution so that the required inputs for *i4* become available. As a result, *i2* would not execute at the clock cycles 4 and 5 due to the fact that the produced output of the clock cycle 3 would be overwritten. Stalling *i2* for 2cc creates a 2 cycle latency bubble that is propagated throughout the instruction sequence, resulting in throughput decrease and PE under-utilization. This problem is solved as we employ fifo queues capable of storing the outputs of each PE hence, enabling the PEs to continue executing instructions while previously generated outputs are not discarded. As a result, instruction *i2* is never stalled, instead it continues its iterative execution, while its outputs are stored in the PE's fifo. In the sequel, instruction *i4* is forwarded the corresponding results from the *i2* and *i3* fifo queue and thus, eliminating the need to stall the pipeline. We mark the forwarding process with blue arrows that depict the flow of data from the fifo queues to the inputs of the corresponding PEs. The aforementioned technique requires the CGRA pipeline to be full and the intermediate results to be stored in the fifo queues of the PEs. In this sense, after a certain amount of clock cycles, each PE will be able to execute one instruction per clock cycle achieving a theoretical maximum of 1 IPC per PE. Considering that the whole loop body is mapped on the CGRA array with each PE executing one instruction per clock cycle, we are able to execute one loop iteration per clock cycle, after the pipeline is filled.

Such a theoretical throughput estimation is affected by the depth of data dependencies of the loop body and the size of the PE fifo queues. Generally we avoid pipeline stalling when the RaW dependencies appear within a number of instructions equal or less to the amount of fifo slots available. For this reason the LP optimization is employed as discussed in the previous section. as it reduces hazards from the data dependencies in the loop steady state and thus, ensuring that the pipeline stalls are minimized. In order to make sure that any loop can be scheduled on the CGRA, we also equip the forwarding unit with a stalling mechanism

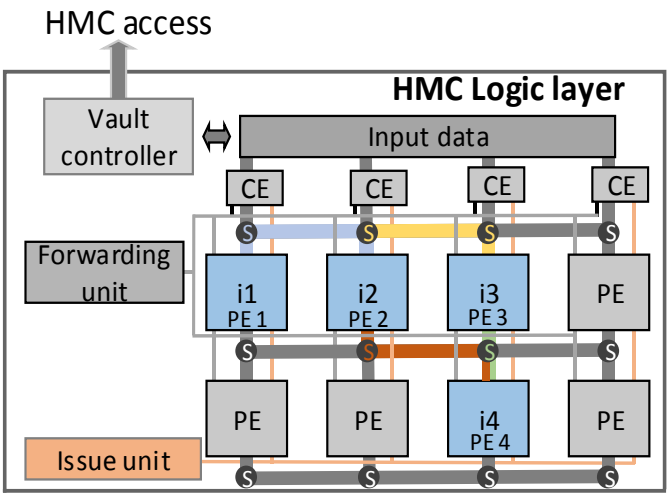

FIGURE 2.5: The outcome of the instruction issue process after which each operation is issued on a PE.

that is enabled when necessary. Nonetheless, we manage to greatly reduce the frequency of such a situation, when using the proposed instruction issue methodology.

2.5 Implementation

2.5.1 System architecture

Figure 2.7 depicts the system architecture we propose for a general purpose NDP methodology. A host system is implemented consisting of a RISC-V CPU core with regular cache hierarchy, RF and a MMU. The host processor die also employs a bus capable of exchanging data with the HMC DRAM. We implement the HMC on a separate die and we design its logic layer to facilitate the NDP accelerator as described in section 2.4. We deploy a SoC mesh network to transfer data between the PE-CE network and the VCs which are responsible for the memory address translation and memory access operations. Each die operates independently from the other, while the communication between them is managed via the MMU and the processor bus.

In order to automate the NDP offloading procedure, we extend the RISC-V instruction set to facilitate the *jalNdp* (jump and link ndp) assembly instruction. This new instruction functions similarly to a *jal* instruction which is used for function calling. The key difference from *jal* is that *jalNdp* initiates the NDP preprocess operation and thus, offloads the instructions located in the function body to the NDP core. For code outside the function invoked through the *jalNdp* instruction, the HMC supports regular memory load/store operations.

In Figure 2.7 we also present a binary execution instance of the H264 encoding protocol. More specifically, the host processor executes the first part of the encoder, namely the part under the *host processor execution* label and above the *jalNdp* instruction. In the sequel, as the *jalNdp* instruction is executed, the NDP preprocess operation is conducted on the host system and accumulates the workload to be dispatched to the NDP core. Such a workload

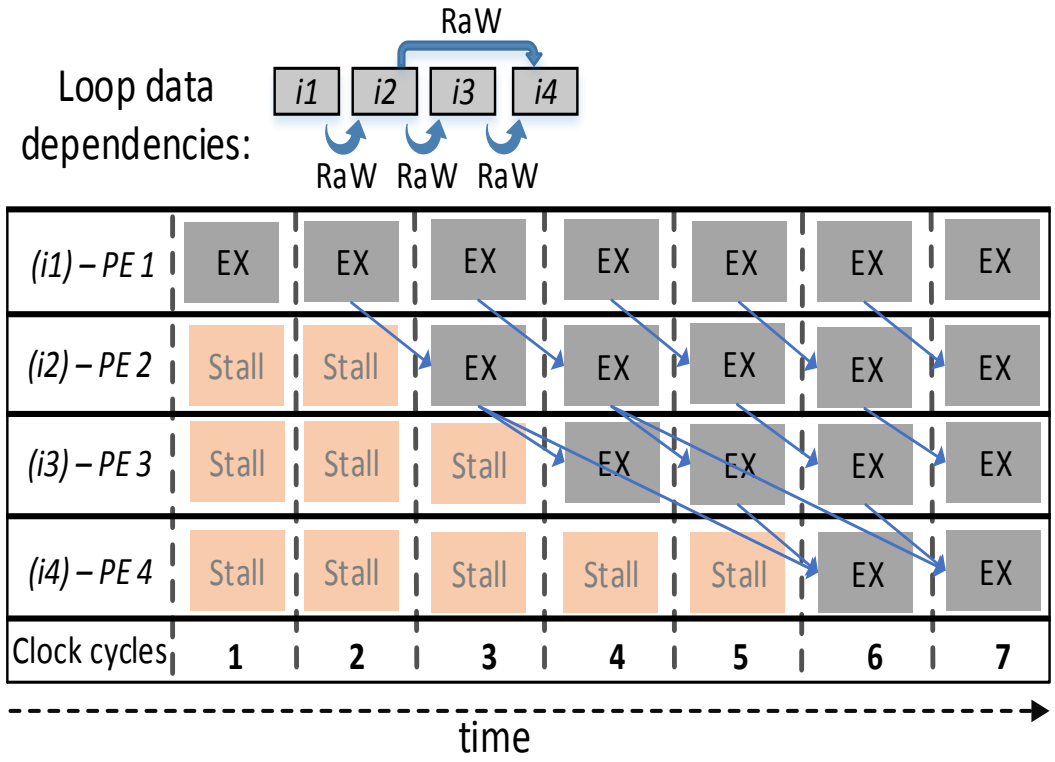

FIGURE 2.6: Instruction execution instance on CGRA microarchitecture.

is composed of decoded instructions with their corresponding operands and data dependencies. Thus the instruction decoding and register fetching stages are executed on the host system. The pre-process operation also generates the data dependency graph and performs the scheduling technique as discussed in section 2.4.3. The resulting outputs include the instructions issued to the PE-CE network and the control signals required for NDP instruction execution. Such data is then transferred to the HMC via the processor bus and the GP I/O links that connect the host processor and HMC dies. On the DRAM side the loop is executed as described in section 2.4.4. In this scenario, the VCs access the memory banks according to the PE-CE load/store requests in order to fetch the required data and deliver them to the CGRA network via the SoC mesh. The switching elements of the SoC interconnect select the VC outputs that fit to each PE-CE memory request and forward the necessary data to the corresponding units. After the loop execution finishes, any outputs needed by the host code are transmitted back to the host system.

In order to maintain the sequential programming model, we stall the host processor pipeline until the NDP execution completes and the results are transferred back to the host die. In this way we enforce that any data dependencies between the executing loop and the rest of the instructions remain unbroken and the loop is executed in-order with the rest of the code. For memory coherence maintenance we consider certain parts of the DRAM uncachable and thus, no information is exchanged between the host cache and the HMC during the NDP execution.

2.5.2 NDP design space exploration and layout

Table 2.1 depicts the host system and HMC design parameters. For the host system we implement a multi-core system that consists of four identical Berkeley Out-of-Order Machine

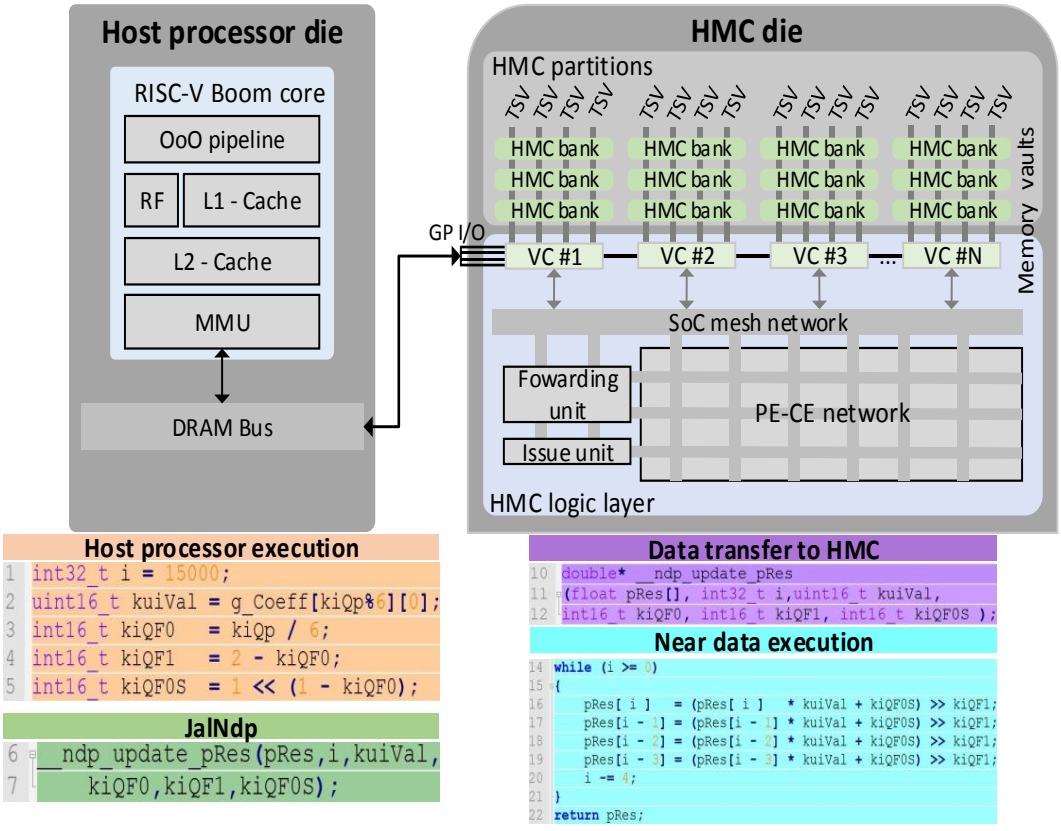

FIGURE 2.7: System architecture for general purpose NDP.

TABLE 2.1: Key parameters of the host processor die and of the HMC implementations.

Host processor	
Core	RiscV Boom OoO, 1 GHz, 64 bit
Amount of Cores	4
Pipeline	10 stages, 4 issue width
L1 cache	32 KB, 8-way, 4 cycle latency
L2 cache	512 KB, 8-way, 12 cycle latency
Branch prediction	gshare, 9-bit history, 512 entries
TLB size	512 entries
HMC 8 GB	
Memory vaults	32
Memory banks	512
Bus Width	128 bits
Timing	tCK = 1.2 ns, tRAS = 24 ns, tRCD = 11 ns, tCAS = 5.5 ns, tWR = 9 ns, tRP = 11 ns
Serial links	480 GBps, 8-cycle latency
BW per vault	16 GB/s
HMC 16 GB	
Memory vaults	64
Memory banks	1024
Bus Width	128 bits
Timing	tCK = 2.5 ns, tRAS = 30 ns, tRCD = 15 ns, tCAS = 7.5 ns, tWR = 10 ns, tRP = 15 ns
Serial links	480 GBps, 10-cycle latency
BW per vault	16 GB/s

(BOOM) [47] cores, which utilizes the RISC-V ISA. BOOM is an open source OoO core that facilitates an 10 stage execution pipeline and offers parameterized synthesis options. We tune such parameters to 32KB L1 and 512 KB L2 cache sizes, gshare branch prediction mechanism and 512 TLB entry size in order to resemble modern process designs. For the HMC implementation we use the openHMC netlist which is a configurable open source HMC architecture [48] developed by the Heidelberg University. We tune the HMC parameters to align its specifications according to the industry standards set by Hybrid Memory Cube Consortium (HMCC) in [33]. Specifically we opt for 8 GB memory size with 32 memory vaults each containing 16 memory banks, resulting in a total of 512 memory banks. We also use a 128-bit memory bus width and GP I/O serial links that transfer data to the host system capable of transmitting 480 GBps within an 8-cycle latency. The maximum bandwidth per vault is 16 GB per second for HMC implementations. We also develop a 16 GB version of the same HMC in order to evaluate our methodology with additional memory implementations.

The physical implementation of the HMC and the host system is carried out by following the CAD toolchain for application specific integrated circuits (ASICs) according to industry standards. To this end, we use verilog HDL to develop the HMC and host system descriptions while the synopsys design compiler is employed for the synthesis operation, gate level and retiming optimizations. In the sequel, we employ the synopsys IC Compiler for place and route, clock tree synthesis and placement operations. For synthesis and physical realization

TABLE 2.2: PE and CE post-layout requirements in terms of area, power and latency.

Functional unit	Area in um^2	Power consumption	Latency
ALU 32/64 bit PE	8335/15471	14/28 mW	1 cc
Mull 32/64 bit PE	15141/28421	23/36 mW	2 cc
Div 32/64 bit PE	18933/34357	65/124 mW	7 cc
FP ALU 32/64 bit PE	6453/11731	28/44 mW	3 cc
FP Mull 32/64 bit PE	6773/13334	33/57 mW	3 cc
FP Div 32/64 bit PE	23858/44836	88/150 mW	8 cc
CE 32/64 bit	2378/4212	3.5/5 mW	1 cc
Issue unit	12556	9 mW	5 cc
Forwarding unit	15436	31 mW	–
Switch	15	1 uW	–

TABLE 2.3: Implementation parameters of 5 different NDP designs.

NDP implementation parameters	NDP-1	NDP-2	NDP-3	NDP-4	NDP-5
HMC size	8 GB	8 GB	16 GB	16 GB	16 GB
Fifo queue size	5 slots	5 slots	10 slots	10 slots	15 slots
Operand width	32 bits	64 bits	32 bits	64 bits	64 bits
ALU PEs	50	25	44	22	21
Mull PEs	28	13	22	12	11
Div PEs	2	2	2	2	2
FP ALU PEs	50	25	43	20	18
FP Mull PEs	28	13	22	12	11
FP Div PEs	2	2	2	2	2
CE	40	40	40	40	35
Number of PEs/CEs	200(20x10)	120(20x6)	175(17x10)	110 (11x10)	100 (10x10)
Interconnection network power	660 mW	720 mW	700 mW	880 mW	1 W
Total power	4.9 W	4.7 W	4.5 W	4.4 W	4.4 W
Total area	1.59 mm^2	1.6 mm^2	1.36 mm^2	1.48 mm^2	1.42 mm^2

we use the 15nm FreePDK library [49] and the post-layout netlist is generated by the IC compiler. In order to verify that our design meets the timing requirements and no timing errors occur we perform static timing analysis with the synopsys primetime tool on the post-layout netlists. Finally the functionality evaluation of the NDP design is conducted by performing gate level simulations on the back-annotated post-layout netlists using the Modelsim tool.

Table 2.2 depicts the post-layout requirements of each PE and CE type for 32 bit and 64 bit implementations in terms of area, power and latency. We present both the integer and floating point (FP) PE versions which are used to compose the PE-CE network. To this end, we implement ALU PE types that are responsible for arithmetic, logical and shift operations, multiplication (Mull) PEs, division (Div) PEs and CEs that evaluate control statements. Table 2.2 also depicts the requirements for the issue unit, forwarding unit and switching elements which are described in Section 2.4. In order to implement the PE-CE network, we also consider the area and the power budget available for the logic layer of the HMC. Previous work in [17] sets the maximum area on the logic layer in the 3D stack at 68 mm^2 and calculates the available power budget at 5W. The functional units are implemented with pipelined execution capabilities, to lower the computation latency and to enable the production of an output per clock cycle (cc).

We conduct a design space exploration of the proposed NDP methodology by implementing five different CGRA designs that are depicted in Table 2.3. Each CGRA implementation consists of a number of PEs, CEs, switching elements, HMC organization and an

interconnection network, as described in Section 2.4. NDP-1 employs 32 bit PEs, an HMC organization of 8 GB and PE fifo queues with 5 slots available for temporary output storage. NDP-2 uses the same configuration options with NDP-1 except for the bit width of the PEs which is tuned to 64 bit. All NDP-3, NDP-4 and NDP-5 utilize 16 GB of HMC DRAM and employ 10 or more PE fifo queue slots. Specifically NDP-3 uses 32 bit PEs with 10 fifo slots available, NDP-4 64 bit PEs with 10 fifo slots and NDP-5 64 bit PEs with 15 fifo slots. The total area and power consumption results of each design also include the requirements of the PE interconnection network and fifo queues, which vary in size according to the number of the CEs and PEs in each design. The core clock frequency is set to 800 MHz in order to balance power requirements and performance goals.

We select the number of PEs-CEs of each CGRA design properly, in order to satisfy the power and area budget as discussed above. The 64 bit designs require significantly more power due to the increase of the functional unit size and thus, a lower number of PEs is selected for such implementations. The number of fifo queue slots also affects the total power consumption, effectively reducing the number of PEs-CEs per design. On the contrary, the memory size of the HMC does not affect the power budget of the logic layer and no adjustments on the PE number are required. By implementing 5 different CGRA designs, we focus on evaluating how the throughput and power efficiency of the NDP methodology is affected by the following parameters:

- **Memory size:** Larger memory organizations tend to display higher latency as depicted in Table 2.1 and thus, data memory load/store operations are completed in longer time intervals.

- **Fifo queue size of each PE:** Larger fifo queues result in more efficient forwarding process and fewer pipeline stalls, but require more power to operate. In this sense, the increase of the fifo size trades pipeline stall time with fewer processing elements.

- **Instruction width:** Smaller instruction widths such as 32bit result in smaller PE size and increase the number of PEs-CEs on the CGRA design. On the contrary, the 64 bit instruction width is widely used in modern microprocessors and thus, we explore its efficiency on the NDP implementations.

We also design and implement two different types of HMC cores to use them for baseline comparison: a simple in-order core and an OoO core in the logic layer of the HMC in order to properly evaluate our methodology in comparison with other NDP systems. The simple in-order core (HMC-DLX) is a 5-stage pipeline implementation of the RISC-V rocket core [105]. It utilizes the RISC-V 64-bit instruction set and facilitates an in-order pipeline with a clock frequency of 800 MHz. It consists of 16 KB L1 instruction and data caches, no L2 cache and the 8GB HMC as described above which is used as the DRAM of the system. The HMC-DLX supports branch prediction and speculative execution using the g-share branch predictor algorithm and a BTB size of 256 entries. For the OoO core (HMC-OoO) we employ a simplified version of the host processor and we utilize the RISC-V BOOM architecture. More specific, the HMC-OoO is a 7-stage pipeline that consists of 8 KB L1 data and instruction cache, of a g-share branch predictor with 256 BTB entries and no L2 cache. It utilizes 64-bit instructions and the clock frequency is set to 800 MHz. We should highlight that the HMC-DLX and the HMC-OoO are not used as host systems; instead we implement such designs on the logic layer of the HMC in order to provide a baseline comparison for our NDP methodology.

2.6 Experimental evaluation

In this section we discuss the experimentation process by which we evaluate our NDP methodology. For this purpose we have opted to use 12 kernels from 8 benchmark suites [50] [51] [52] [53] [54] [55] [56] [57], which are derived from various scientific fields in order to cover a wide range of applications. Table 2.4 depicts the workload characterization of such benchmarks. We profile the binary code of each kernel to obtain the amount of RaW dependencies and the total DRAM memory access requests. We present such characterizations in a qualitative way in the corresponding Table 2.4 rows by using the small(S), medium (M), large(L) and huge(H) annotations. Due to the large benchmark size, the proposed scheduling methodology is not always able to schedule the kernel binary by mapping one instruction in one PE as discussed in Section 2.4. To this end, we apply loop fission transformations as in [58] during the reprocessing stage that split the kernels into groups of smaller loops which can be scheduled according to our technique. The loop fission is automatically performed in the host processor and we include its execution time overhead to the total amount of time a kernel needs to execute. A qualitative depiction of such an overhead is also presented in Table 2.4. We run the aforementioned workloads in all NDP implementations separately and in the host and baseline systems only, in order to compare our findings.

2.6.1 Normalized speedup

Figure 2.8 depicts the speedup of each NDP implementation normalized to the execution time of the host system. To this end we firstly run each kernel on the host processor with no NDP processing taking place. For this purpose we utilize all the available 4 RISC-V BOOM cores by constructing kernel threads which run in parallel in each BOOM core. In the sequel we deploy the proposed NDP implementations along with the host system and we run the kernels again as our methodology dictates. Then we run the same benchmarks on the HMC-DLX and HMC-OoO NDP implementations while employing the same software optimizations (i.e. LP and loop unrolling) we used for the NDP execution. Finally we compare the speedup we obtain for each NDP design over the host-only execution process. Below we discuss the results we obtain after the evaluation process is completed.

Speedup over host only execution: First of all due the the nature of NDP we observe high speedup values as the instruction execution takes place on the DRAM die and thus, the large data movement overhead between the host system and the DRAM is significantly

TABLE 2.4: Workload characterization.

Workload	Kernel	RaW	Mem	Fission
bwaves (Explosion modeling) [50]	K1	H	H	M
cactuBSSN (Physics: relativity) [50]	K2	L	H	L
leela (Monte Carlo tree search) [50]	K3	M	H	L
x264 (video encoding) [50]	K4	S	H	M
K-means (machine learning) [51]	K5	S	S	S
Anisotropic diffusion (image processing) [51]	K6	M	L	M
Feature tracking (computer vision) [52]	K7	S	S	L
Ocean movements (Wave movement) [53]	K8	H	H	S
Linear Regression (ML) [54]	K9	M	H	S
Convolutional Neural Network(ML) [55]	K10	M	H	S
Deep Neural Network (ML) [56]	K11	L	H	M
Shortest Path (graph processing) [57]	K12	L	M	S

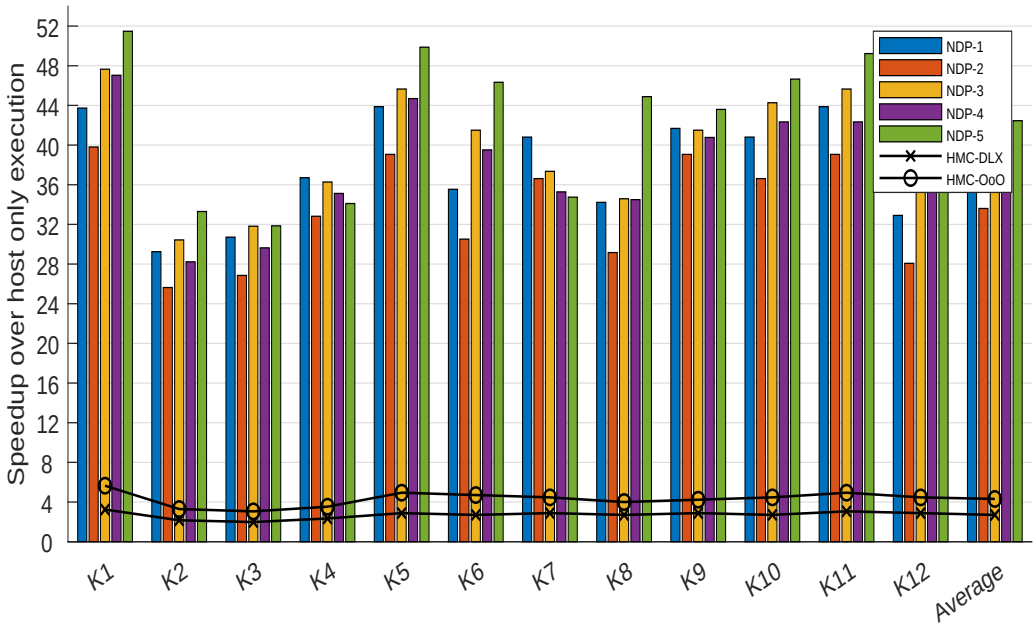

FIGURE 2.8: The speedup improvement of the proposed NDP implementations for each kernel normalized to the host processor execution time.

reduced. Secondly we have designed the CGRAs to optimize general purpose loop execution, hence the execution of the kernel loops is significantly accelerated. The speedup values we obtain for each kernel and NDP implementation depend on various NDP microarchitecture parameters. More specific, 16 GB DRAM designs (NDP-3, NDP-4, NDP-5) come with higher DRAM latency than the corresponding 8GB designs (NDP-1, NDP-2). This phenomenon hinders the execution performance when executing kernels with heavy memory I/O requirements such as K1,K2 and K3 while kernels with lower I/O requirements such as K5 are less affected. Further the amount of PEs on each design plays a major role on the speedup factor. Designs with 64-bit implementations (NDP-2, NDP-4, NDP-5) tend to have lower amounts of PEs due the increased energy dissipation they display. As a result the throughput of 64-bit implementations is hindered as opposed to the throughput of 32-bit designs. The size of the PE fifo queues is also a contributor to the observed speedup. Designs with larger fifos (NDP-4, NDP-5) tend to perform better when they execute kernels with a high amount of RaW dependencies such as K8 and K1. Despite the fact that each NDP design is best suitable for executing kernels with certain workload characteristics, results indicate a 33x to 42x improvement in execution times for NDP implementations. The standard deviation for each kernel speedup is attributed to the combinations of kernel DRAM requirements, size and RaW dependencies that result in a differentiation in execution times. For example K2 displays only a 25x speedup with NDP-2 while K-5 achieves a 44x speedup with NDP-1. Also designs with larger fifo queues such as NDP-3, NDP-4 and NDP-5 tend to perform better even with lower amounts of PEs or with higher memory latency due to the efficient CGRA utilization. For example NDP-4 employs 90 PEs less than NPD-1 and utilizes an HMC configuration with higher latency per operation, but manages to meet the performance of NDP-1 as it utilizes 5 more fifo slots per PE. We conclude that the NDP-5 design achieves 42.4x average speedup factor which is the highest among the other implementations while the NDP-2 achieves 33.6x speedup improvement and thus, being the least performing NDP implementation.

Speedup over the HMC baseline execution: We also compare the performance of the

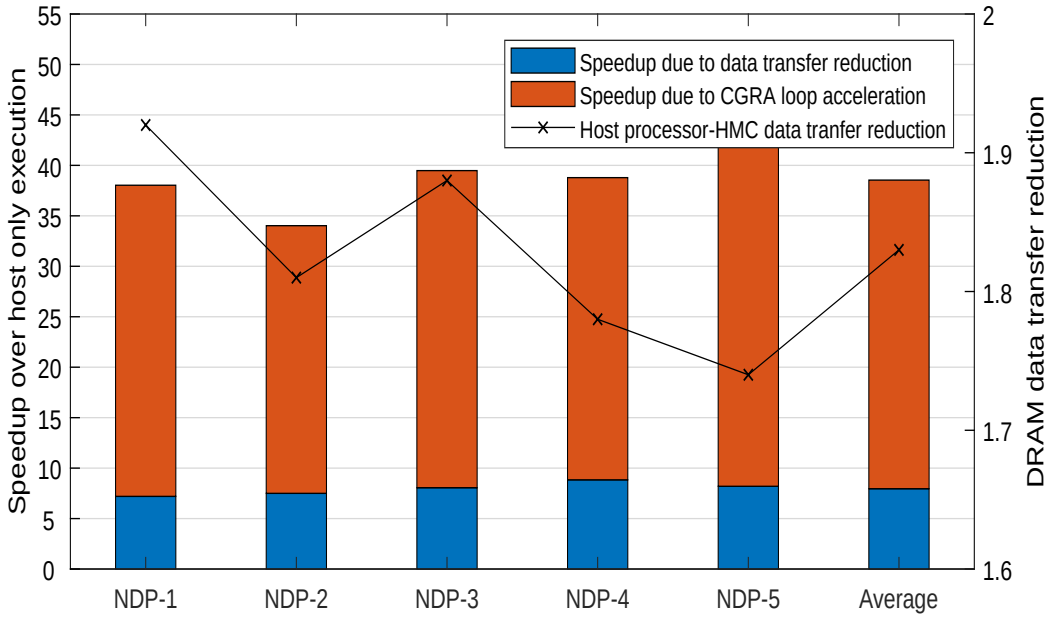

FIGURE 2.9: Host processor-HMC data transfer reduction and its contribution to the normalized speedup of each NDP implementation.

proposed NDP designs with the HMC-DLX and HMC-OoO baseline implementations. The HMC-DLX achieves an average speedup of 2.7x while the HMC-OoO achieves 4.3x speedup over the host-only kernel execution. We expect such a result due to the fact that the host system consists of 4 OoO high-performance cores and thus, the simple HMC-DLX design cannot outperform it by a large margin, despite being deployed near the DRAM. Similarly the HMC-OoO is a less complex and efficient pipeline when compared to the host system, but its performance is adequate to achieve an average speedup of 4.3x when it is deployed on the HMC logic layer. Considering the comparison with the NDP designs (NDP-1, NDP-2, NDP-3, NDP-4 and NDP-5), the proposed NDP architectures outperform the baseline HMC-DLX by an average factor of 14x while the performance increase over the HMC-OoO implementation is 8.8x on average. More specifically, the NDP-5 implementation achieves the best performance over the baseline designs by achieving a speedup factor of 15.7 and 9.88 over the HMC-DLX and HMC-OoO correspondingly. On the contrary, the NDP-2 design depicts the least performance improvement, by achieving 12.2x and 7.82x over the HMC-DLX and HMC-OoO correspondingly.

In Figure 2.9 we depict the contribution of the HMC-host processor data transfer reduction to the overall normalized speedup levels achieved by each NDP implementation. By exploring the impact of the HMC-host processor communication overhead to the speedup improvement we obtain information on the efficiency of the proposed CGRA loop acceleration. We observe that NDP implementations with high amount of PEs such as NDP-1 and NDP-3 reduce the data traffic between the RISC-V BOOM and HMC dies by 90% as the kernels can be efficiently issued on the CGRA. On the other hand NDP implementations with lower amount of PEs require more data transfer operation between the host and DRAM die due to the loop fission procedure which continuously dispatches loops for execution on the HMC logic layer. Further the speedup improvement is also depending on the CGRA microarchitecure parameters such as the PE fifo size and thus, the NDP-5 outperforms the rest of the implementations. Nonetheless, the DRAM data transfer reduction contributes to the overall speedup up to an average 46%, while the rest 54% is attributed to the design of the

CGRA that is implemented on the HMC logic layer.

2.6.2 Energy reduction

Figure 2.10 depicts the reduction of the energy consumption for each kernel execution, normalized to the host only execution. We collect such results by averaging every implementation's normalized energy reduction for each kernel.

Energy reduction over host only execution: We observe an average of 22.4x reduction in energy consumption when executing the aforementioned benchmarks on the NDP designs. NDP achieves significantly faster execution times when compared to the host processor and also reduces the traffic between the DRAM and processor die. As a result the energy requirements of each benchmark are significantly reduced when employing the proposed NDP methodology. Results vary among the executing kernels due to the wide range of requirements of the corresponding workloads. For example K2 achieves 13.5x while K5 achieves 30x reduction in energy consumption due to the fact that the K2 kernel takes longer to execute while also having higher DRAM access energy overheads. The average energy reduction levels are 22.4x compared to the host-only execution, which demonstrates the efficiency of the proposed NDP framework.

Energy reduction over the HMC baseline execution: We also run the aforementioned benchmarks using the HMC-DLX and HMC-OoO implementations so that to compare the proposed NDP methodology with the baseline designs. We observe that the HMC-DLX architecture achieves an average 2.3x reduction in energy consumption over the host system, while the HMC-OoO manages to reduce the energy requirements of each kernel by 4.3x on average. As a result, the proposed NDP implementations perform 5.4x to 10.5x better when compared with the HMC-DLX and 3.66x to 6.2x when compared with the HMC-OoO baseline. On average, our NDP designs achieve 9.5x greater reduction in energy consumption over the HMC-DLX and 5.15x over the HMC-OoO baseline.

Figure 2.11 depicts the breakdown of energy consumption of each NDP implementation. The results are obtained by averaging the energy reduction of each kernel for the corresponding NDP design.

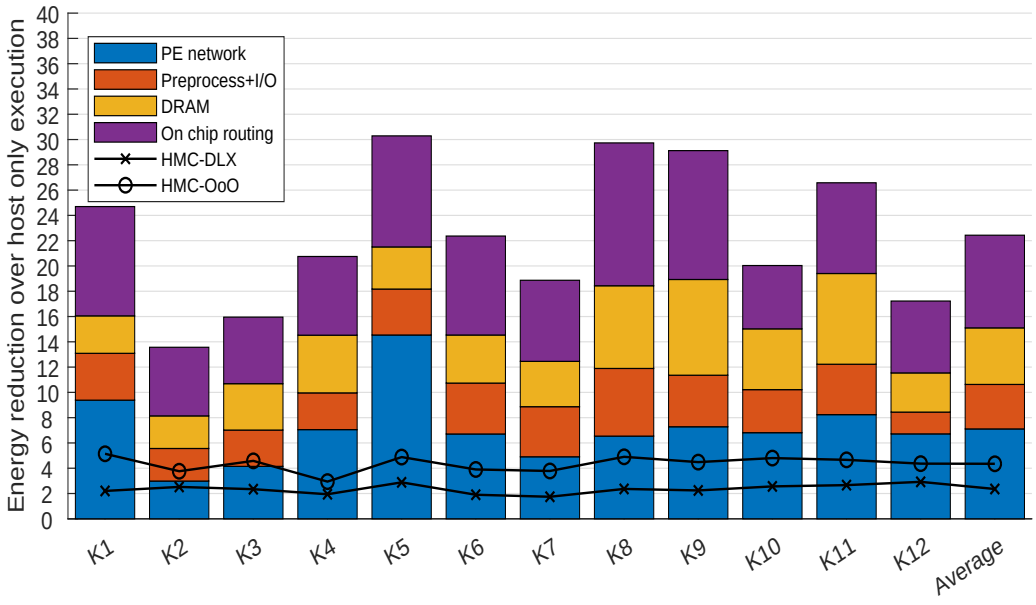

FIGURE 2.10: Normalized energy reduction of the NDP methodology.

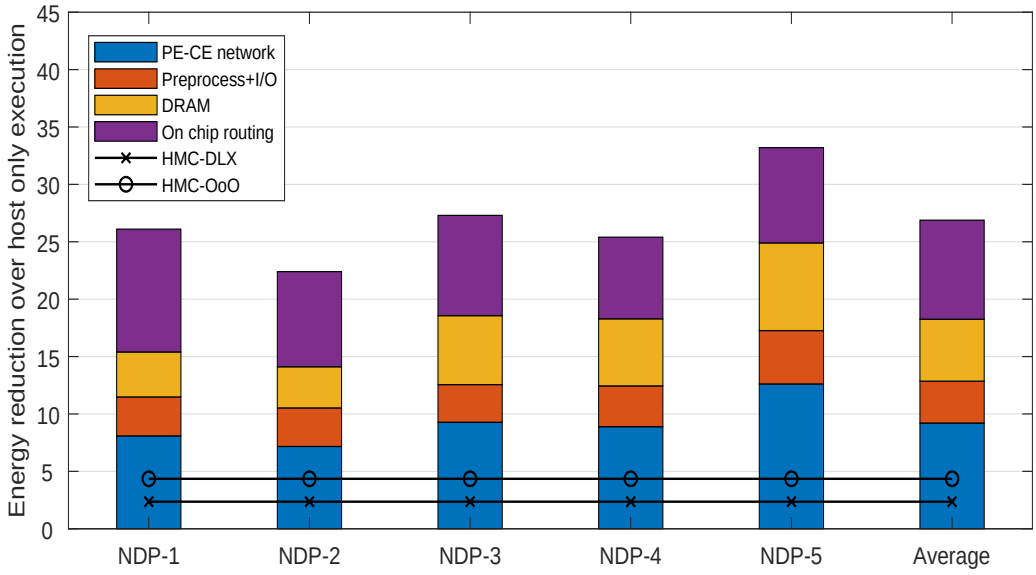

FIGURE 2.11: Energy consumption breakdown of NDP implementations.

Energy reduction over host only execution: We observe that the NDP-5 implementation achieves 33.2x energy reduction normalized to host system which is the highest among NDP designs. On the other hand NDP-2 achieves 22.4x reduction and thus, being the least performing implementation in terms of energy saving. A significant energy overhead of the designs derives from the on-chip routing costs. The SoC mesh interconnection network which connects the PEs-CEs of the CGRA is responsible for data forwarding and consumes 25% (NDP-5) to 41% (NDP-1) of the total energy. Apart from the routing costs, the PE-CE network energy consumption is also significant and increases from 31%(NDP-1) to 38%(NDP-5) as the amount of fifo slots increases too. The preprocess+I/O energy is attributed to the preprocessing that takes place on the host system which includes the loop pipelining, unrolling, fission and scheduling processes. Such an overhead is small but not trivial and contributes around 14% to the NDP energy consumption. Finally the DRAM energy costs come from the amount of memory access requests and the size of the DRAM. Larger DRAMs such as NDP-5 (16 GB) require more energy per operation when compared with smaller DRAM sizes as the one in NDP-1. In conclusion the energy consumption of each NDP implementation is mainly affected by the interconnection network complexity and the amount if fifo slots on each PE. On the other hand the DRAM data transfer energy cost becomes major in 16 GB DRAM implementations while the preprocessing costs remain in relative low levels on each NDP design.

Energy reduction over the HMC baseline execution: We present the average energy reduction achieved by the HMC-DLX and HMC-OoO baseline implementations (i.e. 2.3x and 4.3x respectively) over the host only kernel execution. We depict such values in order to obtain a clear picture of the performance in terms of energy savings of the proposed NDP implementations. Our findings are explained above, when discussing the figure 3.8, while figure 2.1 depicts the average energy reduction of the HMC-DLX and HMC-OoO correspondingly.

2.6.3 Power and area efficiency

Figure 2.12 depicts the power efficiency of the proposed NDP and HMC baseline implementations normalized to the power efficiency of the host processor. We collect such metrics

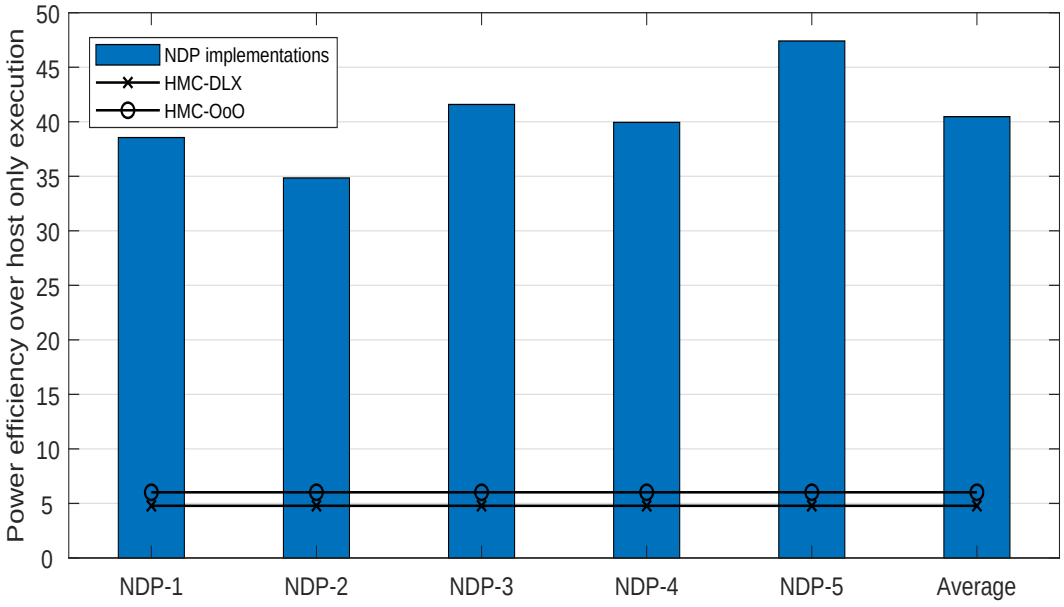

FIGURE 2.12: Normalized power efficiency of the NDP implementations.

by studying the achieved throughput per Watt of the NDP, the HMC baseline and RISCV-V implementations.

Power efficiency over host implementation: Results indicate that NDP implementations are 34x to 47x more power efficient when compared to the host processor. The NDP-5 implementation shows the best power efficiency due to the high speedup ratios it achieves. Such a power efficiency improvement is attributed to the NDP architectures which focus on accelerating general purpose loops, while the host processor under-performs as it is designed to operate under a wider spectrum of applications.

Power efficiency over the HMC baseline implementations: The HMC-DLX baseline implementation achieves an average 4.8x better power efficiency when compared to the host processor, while the HMC-OoO baseline achieves a 6x increase respectively. As a result, the proposed NDP designs depict an average 8.4x and 6.7x improved power efficiency compared with the baseline HMC-DLX and HMC-OoO implementation correspondingly.

Figure 2.13 depicts the area efficiency of the proposed NDP implementations normalized to the area efficiency of the multi-core host processor. Similarly to figure 2.2 we measure the area efficiency as the throughput achieved over the die area of the integrated circuit. We then normalize such measurement to the corresponding area efficiency of the host processor in order to compare our findings.

Area efficiency over the host implementation: Results indicate that the proposed NDP designs are 129x to 179x more area efficient when compared to the host system. Such an improvement is expected due to the small die area of the NDP implementations and the high speedup rates they achieve. We also observe that the NDP-5 is 26% more area efficient when compared to the NDP-1 despite the fact that the speedup improvement of the NDP-5 is 26% compared to NDP-1. Such a difference is attributed to the die area difference of such designs as the NDP-5 is 13% smaller than the NDP-1 implementation, and thus its area efficiency is significantly higher.

Area efficiency over the HMC baseline implementations: The HMC-DLX implementation is 14x times more area efficient compared to the host system, while the HMC-OoO achieves a 18x better area efficiency over the host processor. Further, the proposed NDP implementations (i.e. the NDP-1, NDP-2, NDP-3, NDP-4 and NDP-5) manage to outperform

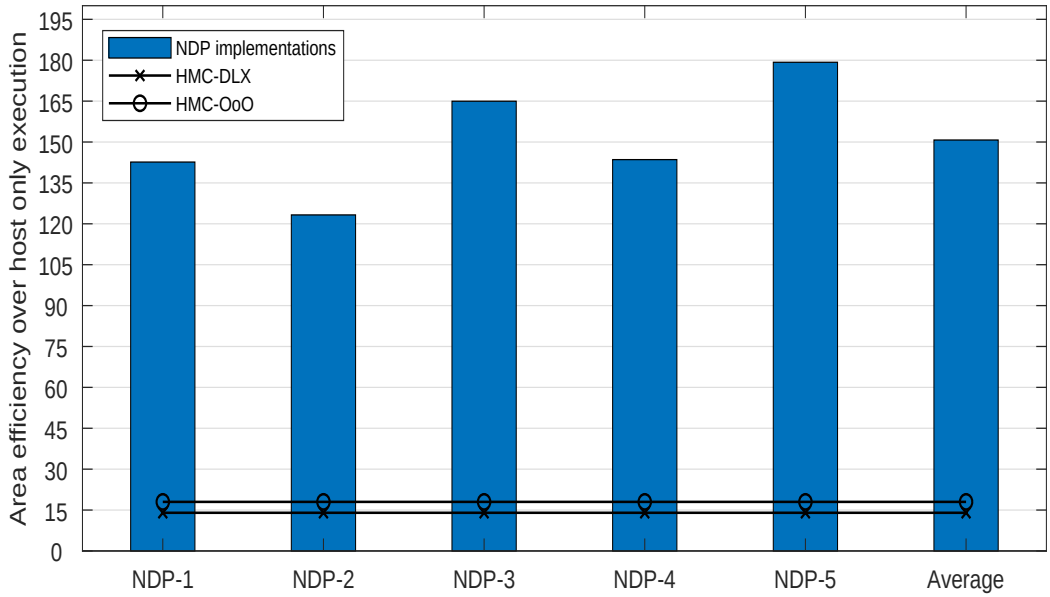

FIGURE 2.13: Normalized area efficiency of the NDP implementations.

the corresponding baseline designs (HMC-DLX and HMC-OoO) by a factor of 10.7x and 8.3x respectively.

2.6.4 Speedup improvement per Watt

Figure 2.14 depicts the speedup improvement of each NDP and baseline implementation for each unit of watt consumed. In this sense, we measures the efficiency of the proposed designs to trade power consumption for speedup improvement.

Speedup improvement per Watt: We observe that the highest ratio is achieved by NDP-5 implementation which improved the speedup by 9.4x per unit of watt consumed. On the other hand, NDP-2 utilizes a unit of watt to increase the speedup by 7x and thus, is evaluated as the least performing design. The average speedup increase per watt is 8.3x which we consider very efficient in terms of trade off benefits.

Speedup improvement per Watt for the HMC baseline implementations: The HMC-DLX baseline pipeline utilizes 1 unit of watt to increase the speedup of the design by a factor of 0.87x, while the HMC-OoO achieves 1.17x speedup over the host system per unit of watt consumed. Thus the proposed NDP methodology outperforms the HMC-DLX baseline by a factor of 9.4x and the HMC-OoO by a factor of 7.1x in terms of speedup per unit of watt consumed.

2.6.5 Comparison with related works

Table 2.5 depicts the normalized speedup and energy reduction levels achieved by the current state-of-the-art works in the NDP literature. In order to compare our work against the other NDP architectures we utilize the mean benchmark results provided by the authors of the corresponding previous works and we compare them with the mean results obtained by our research. We opt to focus on the speedup and reduction in energy consumption measurements, as we believe that they are indicative of the performance of a system. To this end, for each work we use the speedup and energy reduction over the baseline implementations

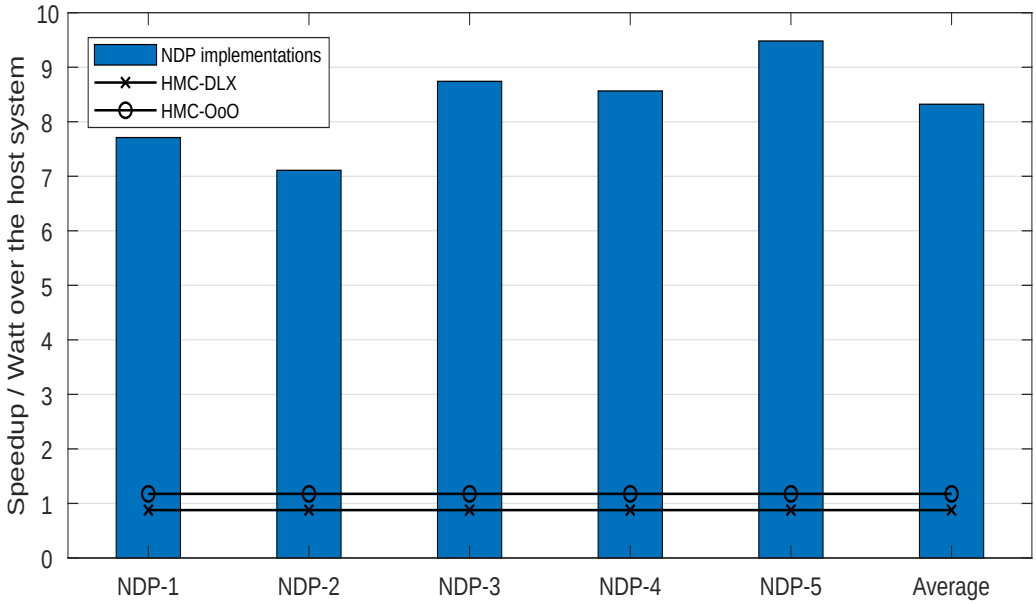

FIGURE 2.14: Speedup per Watt of each NDP implementation.

and not over the host only execution. As a result, we obtain results that correspond to the performance of the NDP designs when compared to other existing NDP implementations.

This comparison demonstrates that the NDP architecture proposed in this work is one of the best performing designs according to the current state of the art. More specifically, previous works in [8] [15] [16] [17] [18] [22] [24] [27] and [31] perform an order of magnitude lower in speedup improvement than the NDP framework proposed in this work. On the contrary, previous work in [11] is closer in terms of speedup with the proposed NDP architecture, but our design manages to outperform it by a factor of 2x in the best case scenario (NDP-5). Considering the energy consumption, we observe that the proposed NDP architecture achieves better energy reduction ratios when compared with the existing literature, even in the worst performing design (NDP-2).

TABLE 2.5: Comparison of the proposed NDP architectures with the current state of the art.

NDP architecture	Normalized speedup over NDP baseline	Normalized energy reduction over NDP baseline	Evaluation
[8] General purpose	3.5x	4x	Spice simulation
[11] Graph Processing	14x	5x	In-house software simulator
[15] Reduction operations	7x	3.5x	Software simulator
[16] General purpose	2.2x	5x	Gem5 simulator
[17] Memory intensive	2.2x	-	ASIC/FPGA
[18] General purpose	3.8x	3.4x	Spice simulation
[22] Big data	6.1x	5x	Device, integrated circuit
[24] PIM cache coherence	2x	2.7x	Software simulator
[27] Graph Processing	1.5x	2x	In-house software simulator
[31] General purpose	3.4x	2x	Software simulator
NDP-1 General purpose	14x - 8.7x	11x - 5.9x	ASIC/gate level
NDP-2 General purpose	12.2x - 7.8x	9.5x - 5.14x	ASIC/gate level
NDP-3 General purpose	14.6x - 9.1x	11.5x - 6.2x	ASIC/gate level
NDP-4 General purpose	14.2x - 8.8x	10.7x - 5.8x	ASIC/gate level
NDP-5 General purpose	15.7x - 9.8x	14x - 7.6x	ASIC/gate level

Chapter 3

Timing analysis for low power pipelines

3.1 Introduction

Traditional microprocessor design ensures an error free instruction execution on general purpose processors. According to the established model, the designer designates the clock frequency and the voltage values of the processor, so that no timing violation of the critical path occurs. Thus, the design revolves around the timing analysis of the worst-case scenario, and the critical path acts as a strict timing threshold, constraining the processor performance.

In contrast to the traditional model, the BTWC paradigm attempts to relax any critical path restrictions through TS, by scaling up and down the processor voltage or clock frequency, allowing timing errors to occur. The resulting errors can then be resolved by an integrated rollback error correction mechanism. Such a paradigm presents many design opportunities for performance enhancement and power reduction.

This work is loosely based on the BTWC design paradigm, primarily focusing on the performance increase of the processor pipeline for low-cost, low power processors. In this chapter, we present the IPE-STA, a methodology that improves performance by executing instructions in varying clock frequencies, according to the opcode of the executing instructions. A proposed timing analysis methodology detects instruction opcodes that may run at high speed, as well as instruction opcodes that must run at low speed. The instruction pipeline is then fed with multiple clock signals, multiplexed in a way that when a critical instruction is decoded, a slower clock is selected for the cycle that exhibits the maximum timing delay, reverting to the previous clock in the following cycle. Thus, the typical clock selected can be faster than the one designated via traditional timing analysis.

Such an approach diverges from classic BTWC design techniques, in that we utilize the knowledge about each individual instruction timing requirements, obtained from our timing analysis methodology. With that knowledge, our model foresees any upcoming timing errors and corrects them a priori. In this way, we eliminate any execution error probability, thus no error recovery mechanism is deployed, as each instruction is certain to meet its timing requirements. Since performance penalty induced by error correction is non-trivial, our more deterministic error detection mechanism avoids the performance implications of traditional BTWC techniques. Furthermore, the developed methodology is architecture independent making it applicable to any given single-issue in-order-execute design, without requiring intrusive changes to its microarchitecture.

The above technique has been implemented and tested on two different post-layout RISC-V Verilog processor implementations, using the SPEC 2017 CPU benchmark suite. The same tests have been applied on the implementations without clock-scaling, and the results obtained show a clear improvement in processor performance, between 12% and 76%, and an average 3.7-fold improvement in performance-to-power ratio, despite the expected increase in power consumption due to high frequency operation.

3.2 Related word

Following decades of standard worst-case processor design methodologies, a considerable amount of research has being conducted in BTWC designs in the last few years. The BTWC paradigm treats the critical path more flexibly than traditional designs [59]. In particular, TS aggressively violates critical path restrictions, allowing and then correcting any resulting errors. Such a technique enables researchers to tamper with energy-performance trade-offs and promises to efficiently increase the performance or lower the power consumption of a circuit [60].

The suggestion to apply TS on processor design has led to the development of Razor [61] [62]. Razor's pipeline employs TS in order to improve the performance-to-power ratio, while also utilizing "shadow latch" circuits, to detect and correct the errors incurred by the alteration of the voltage power. This correction mechanism operates in real time and ensures the error free instruction execution. Another research work focuses on the dynamic frequency scaling of a superscalar processor, supported by error recovery mechanisms to compensate for resulting timing errors [63]. In that work, researchers deploy both local and global error correction mechanisms, which ensure a correct instruction execution when the processor is overclocked at higher frequencies.

BTWC design methodologies have also proven able to address the ever increasing process variation effects or the uncertainty caused by the environment and the fabrication process of the integrated circuits [64]. As a result, a significant amount of research compensates for PVT variations, while exploring possible TS benefits [65]. In such cases, probabilistic methods can be deployed to model the PVT fluctuations [66], while guardbanding has been proposed in one work to safeguard the circuit against timing violations [67]. Another work shows that process variation effects result in pipeline imbalances as long as timing delay is concerned [68]. In order to overcome this problem, a framework has been developed to tighten the timing of the circuit using a time stealing technique equalizing the timing requirements of each pipeline stage. A third work exhibits that the mitigation of PVT effects may be achieved by a properly developed framework [69]. In that work, the researchers manage to model the PVT effects and create a framework that enables error-power and error-frequency trade-offs. Finally, another work demonstrates novel techniques which may be used to design PVT resilient microprocessors [70]. Such techniques include the monitoring of critical paths by sequential circuits that detect timing errors, or the monitoring of each pipeline stage for worst-case delays. In both cases, the designers also propose error recovery mechanisms and exhibit a significant performance increase by utilizing clock frequency scaling.

Previous work has also shown that through a careful application of timing optimizations designers may obtain significant energy-performance trade-offs at a higher architecture level [71]. A marginal cost analysis demonstrates the potential of circuit voltage scaling on architectural level, highlighting the optimal operational point of a target processor. Furthermore, the design process of a microprocessor could also be aligned to facilitate TS friendly microarchitecture adjustments. Specifically the optimization of the most frequently exercised critical paths may result in clock frequency scaling and lower power dissipation on existing TS architectures [72]. Along the same lines, the slack redistribution of the most frequently occurring timing paths of a processor may lead to architectures with lower power consumption and minimum error rate [73]. Another work in [74] proposes a TS cache design which manages to lower the energy consumption of the system while maintaining high cache hit ratios within various cache organizations.

Apart from the architectural point of view, TS techniques have also been proposed on circuit level design. Specifically the proposed DynaTune methodology monitors the dynamic behavior curve of a circuit and performs gate level optimizations to reduce propagation delay [75]. An error detection technique has also been proposed which utilizes multi-level logic

speculation in time domain to make decisions on whether the timing of the circuit should be relaxed or not [76]. Some research also indicates that logic synthesis optimization may result in lower error rates in TS circuits, by using a path delay balancing technique [77]. The goal is to classify high fan-in logic gates together and to optimize their timing balance using a proposed logic synthesis methodology.

BTWC design research is not only focused on architecture or circuit design, but on compiling methods as well. It has been suggested that BTWC logic requires a TS aware compilation method, capable of exploiting the error resilience of the target processor. As a result binary recompilation has been proposed, with respect to the timing speculative processor ISA, to further reduce the error rate of the executable file [78]. Another approach to the problem suggests a convenient ISA design in conjunction with proper compiler optimizations to boost performance by lowering the amount of error rate, while increasing the processor clock frequency [79]. The usage of a critical path profiler has also been explored, resulting in the design of a profile compiler comparing the delays of all instruction paths, while also using a form of value prediction to speed up execution [80].

As the TS paradigm revolves around scaling the clock frequency in real time, research is also focused on clock adaptation techniques. Specifically, previous work in [81] manages to adapt the clock frequency of a POWER 7 processor core by adjusting the voltage values in firmware level. Combined with a critical path monitoring mechanism, researchers achieve voltage scaling when the critical path is not excited while using the available timing margin as a guardband mechanism. Another work utilizes a unary coding scheme to enable the PLL to quickly adapt to the required clock changes in real time [82]. This approach can be applied on a single core clock to enable its dynamic frequency change without imposing significant delays. A research that also underlines the importance of a robust real time clock adaptation scheme is [83]. In this work, researchers manage to deploy a clock adaptation scheme which can reduce the clock frequency in an AMD 28nm microprocessor core improving the power efficiency of the system. This approach utilizes a phase generator which can modify the clock's phase in order to stretch its period. Similarly, in [84] authors employ a dynamic clock adjustment technique on a simple processor pipeline to adjust the clock frequency according to the application type that is being executed on the processor pipeline.

TS designs are often prone to exhibit metastable behavior, resulting in non-deterministic timing phenomena which should be taken into account when designing a TS processor [85] [62]. This issue usually appears when the input data arrives close to a rising clock edge, resulting in the possibility of undetected errors. In the course of minimizing the metastable behavior of timing speculative circuits, design methods have been proposed, utilizing a time borrowing technique alongside of a careful examination of the data path timing, which simplifies the issue of metastability by moving such behavior to the circuit error path only [86]. Despite the progress being made on this issue, a more recent work claims that circuit metastable behavior in TS designs is not yet efficiently addressed [85]. In this regard the mean time between failures of such designs discourages any possible industrial applications.

Another thought provoking aspect in BTWC techniques is the error recovery mechanism and the performance penalty it imposes on the design [87]. Due to that penalty significant effort has been made on the improvement of the deployed prediction mechanisms [88]. While some designs employ statistical methods to successfully detect specific instruction sequences which have pre-analyzed timing requirements [67], others tend to focus on monitoring the critical path excitation by individual instructions [89] [90]. Another approach revolves around the identification of timing critical instructions during runtime, using that knowledge to improve the energy-efficiency ratio of the processor [91]. Finally a study on the CMOS recovery mechanism reveals the impact of the technology on such techniques, as researchers develop a hardware model sufficient to simulate timing speculation designs [92]. The same research also underlines the importance of a fine-grained error recovery mechanism

in BTWC designs. Although the penalty imposed by the execution or by the unsuccessful prediction of a critical instruction is relatively low, results indicate that the performance loss due to error recovery is non-trivial. Moreover, in some extreme cases the design's performance deteriorates to the point that the TS design displays lower throughput than the baseline processor [67].

From all the above reviewed work we have concentrated our interest on the issues of error recovery in the TS design paradigm, as well as on the issues of metastability observed in that paradigm. Our motivation has been to study such issues and come out with a novel technique that exploits TS in a way that any possible speculation errors are detected dynamically and recovered before they appear, thus avoiding the costly error recovery mechanisms, and at the same time eliminating metastability phenomena altogether. In the following sections of this chapter we introduce our opcode-based timing analysis and clock scaling technique for error-free timing speculation in pipelined microprocessors.

3.3 Background

3.3.1 Static and dynamic timing analysis

Timing analysis is a technique traditionally used in order to analyze timing behavior of a digital circuit and establish the optimal clock cycle for that circuit.

Standard STA is performed either at flop to flop or at input to output basis. It calculates the worst-case delay of the analyzed circuit and reports whether setup or hold violations occur under certain design constraints. It can also be used to highlight the critical path of the circuit, which plays an important role on the timing requirements of the design. However, STA is overly pessimistic, since it considers worst-case delay for each path of the design. Such an analysis turns out to be quite inefficient for the BTWC model, as not all paths have their worst-time delay at the same time.

On the other hand, DTA may be used in order to acquire very accurate timing information about a circuit design. It utilizes a set of input vectors as triggers and proceeds to circuit excitation with the selected input values. An exhaustive DTA will display all actually possible timing paths of a given circuit, at a very high time cost though. It eliminates the path pessimism induced by STA as it analyzes the timing requirements of every possible input of the circuit. Although DTA would be more appropriate for BTWC design timing analysis, its time cost renders its usage impossible on large designs, as it trades accuracy for completion time.

3.4 Timing analysis in processor datapaths

3.4.1 The instruction path exhaustive timing analysis concept

In this work we propose the instruction path exhaustive static timing analysis (IPE-STA) technique inspired by the BTWC design approach. More specifically, we develop a timing analysis model, which extracts circuit information that enables us to exploit the timing differences of the processor timing paths. In order to obtain such information, we analyze each individual instruction the processor supports, with respect to its unique timing requirements. To analyze each instruction independently, we isolate from the integrated circuit all the possible paths an instruction may take, while declaring the rest of the paths as false. We repeat this process until we exhaust all the available instructions. Afterwards we perform STA on each separate path group that corresponds to each individual instruction. The results we obtain refer to the worst-case timing requirements of each instruction, instead of referring to the worst-case delay of the processor.

In order to present our technique, we refer to a standard timing analysis tool, with which we conduct timing analysis on the post-layout netlist of a pipelined processor. Our technique is architecture agnostic and thus, it can be applied to any existing processor architectures provided that we have access to their corresponding ISA. Algorithm 2 depicts our initial approach to the problem. First we pick an instruction supported by the processor. We then consult the ISA to identify its opcode, while ignoring any register or data fields facilitated within the instruction word. For that opcode we commence a timing analysis operation, in which the designer may set any of the circuit inputs to constants, and let the tool being utilized perform a flop to flop timing analysis given the fact that some of the circuit inputs are set to a fixed value. Such functions (i.e. "case_analysis") are supported by the current state of the art timing analysis tools such as the synopsys PrimeTime and thus, we consider such an operation standardized. In our case these values represent the current instruction opcode field. The tool propagates any generated signals through the processor pipeline to analyze the timing of each pipeline stage separately while performing STA. As a result we obtain a timing report that displays the timing requirements of each pipeline stage for the fixed instruction opcode, thus for the designated instruction. That report gives the worst-case scenario of the analyzed instruction with respect to timing. In the sequel, we keep record of the slowest pipeline stage timing, before moving on to the next instruction. Finally we pick another instruction and reiterate this process, until all the available instructions are exhausted.

It becomes clear that the timing analysis methodology we propose is a hybrid between STA and DTA. It performs STA for each instruction of the processor ISA, but it further analyzes all possible instructions, giving a DTA flavor to the result. However, it is not a full DTA, since it only varies the opcode field of the instruction word, thus not considering input values neither for any other field of the instruction word nor for any other part of the circuit. In particular, the implementation of the architecture we work on supports 180 instructions, which means that we only need 180 analyses in our method, instead of 264 which would be needed for a full DTA on the variations of the instruction word alone, or many more if we were to consider other circuit inputs as well. On the other hand, our approach is not as pessimistic as classic STA, and with only little higher complexity it can produce designs that exhibit significantly better performance than designs produced by STA. Our second approach presented next can produce even more efficient designs.

3.4.2 Dynamic opcode value changes compensation

Using the proposed technique we manage to analyze the timing requirements of each processor instruction individually. To this end, we are iteratively affixing certain circuit inputs at

Algorithm 2 The proposed timing analysis methodology which studies each instruction as an individual entity.

```
Start
instruction_number == supported_instructions[ISA]
while instruction_number > 0 do
   instruction = next_instruction
   opcode = instruction[opcode]
   stage_delays = STA[netlsit,opcode]
   worst_case[instruction_number] = max[stage_delays]
   instruction_number –
end while
End
```

constant voltage values. Specifically, we are bounding the opcode field bits to static binary values in order to analyze each instruction behavior.

In real time digital circuits though, inputs change in a dynamic way as new values are stored into the pipeline registers at the rising edge of the clock signal, immediately before they are needed and used. As a result, new instructions are fetched for execution on each clock cycle and thus, inputs representing the opcode bits of each instruction are not constrained to fixed voltage values; instead they dynamically change, resulting in an unpredictable transient timing behavior in every pipeline stage after the fetch stage. Such behavior appears at the decode stage due to the opcode bits per se, as well as at all subsequent stages due to the control bits produced by the opcode bits and is propagated to such stages. For this reason, the discrepancy between our initial concept and a real time system behavior in timing deviations should be addressed.

In order to compensate for the dynamic voltage change in the processor pipeline inputs, we employ a modification of the aforementioned timing analysis methodology. Since our focus is now around the timing variance created by the dynamic behavior of the instruction opcode field, we consider the opcode as a bit sequence whose length is defined by the ISA. Consequently we have to take into account every possible value transition which leads to the currently analyzed bit sequence. Normally the amount of all possible combinations grows exponentially with the length of the sequence. We consider this approach unsuitable for our needs as its high time cost makes it impossible for practical application. Furthermore, we aim at the development of a methodology, which can be employed to analyze any ISA, without depending on the instruction opcode length of the design.

The solution we propose to resolve this is based on the observation that in processor architectures not all possible bit transitions lead to valid bit sequences of the opcode field. More specifically, as the instructions succeed one another during the instruction fetch stage, the number of valid opcode bit combinations is constrained by the number of the instructions supported by the ISA. So instead of analyzing the timing delay of each possible opcode bit sequence transition, we focus on the analysis of each possible instruction succession.

Algorithm 3 depicts the proposed solution, which relies on the initial concept as described earlier, augmented with the dynamic value change compensation approach we discussed. In this solution, we analyze each instruction's timing requirements individually as before, but instead of using fixed voltage values to describe the currently examined instruction, we analyze each possible opcode transition that could lead to the opcode bits of the current instruction. Such transitions represent any rising and falling voltage values that could result in that particular bit sequence. The timing analysis of such cases is studied individually, while the timing analysis tool propagates all generated signals through the processor pipeline. We still use the STA operation, as it can be expanded to include rising and falling voltage change. When we complete the timing analysis of an instruction, we save the worst-case delay. Afterwards we proceed with the analysis of the next instruction, until all supported instructions are exhausted.

The method we proposed in this section uses STA to find the worst-case delay path of each individual instruction. But in order to achieve an accurate timing result we utilize an exhaustive iterative analysis resembling more now that of a DTA method. However, even if we restrict value variations within the instruction opcode, with an opcode field of x bits, a standard DTA approach would require $2x$ iterative timing analyses to effectively analyze the timing of the opcode length, as each possible x-bit combination may lead to the required sequence. Instead, using our methodology, the number of iterations needed for each opcode analysis is only equal to the total number of supported ISA instructions.

The proposed timing analysis technique lies somewhere between STA and DTA, closer to STA with respect to complexity, but closer to DTA with respect to output quality. We call this technique Instruction Path Exhaustive Static Timing Analysis (IPE-STA). In the following

Algorithm 3 The timing analysis methodology which compensates for the dynamic voltage change of the opcode field.

```
Start
instruction_number == supported_instructions[ISA]
while instruction_number > 0 do
    instruction = next_instruction
    for all possible opcode transitions do
        opcode = instruction[opcode]
        stage_delays = STA[opcode_transition,netlsit,opcode]
        worst_case[instruction_number] = max[stage_delays]
    end for
    instruction_number –
end while
End
```

section we discuss the application of IPE-STA to adaptively scale the clock frequency of the core pipeline of a processor.

3.5 Clock scaling of RISC-V using IPE-STA

3.5.1 Adaptive clock scaling in pipelined Processors

Adaptive clock scaling is often used for power control in modern processor architectures. Processor cores can be slowed down when not in full use, in an attempt to reduce power consumption and avoid overheating. In some cases, cores can be sped up for a limited time, in order to boost performance for cycle-hungry applications. On the other hand, low-cost processors that are preferred for embedded and low-performance systems can also use clock scaling in order to increase performance, especially if this is achieved in a fairly cheap manner.

Another way to effectively scale up the clock frequency is to deepen the processor pipeline. In this way each stage's latency is reduced and the system may operate lower clock periods. Previous works in [93] and [94] demonstrate that deepening the processor pipeline results in an increase in circuit area and power consumption due to the implementation of additional pipeline registers, Furthermore, deep pipelined processors require more complicated forwarding, control and stalling mechanisms thus, further impairing the design's area and power requirements. Finally authors in [93] and [94] conclude that the increase of the pipeline stage amount does not necessarily result in performance increase as the costs of wrong branch predictions and pipeline flushing become greater.

Our work focuses on low-cost processors which present significantly low area and power requirements as stated in [95] and [96]. Under this premise we do not opt in deepening the pipeline width or enhancing the complexity of the system, instead we focus on increasing the processor throughput while preserving the system microarchitecture as it is. In this section we will present how IPE-STA can be employed for such a purpose.

Clock scaling is often used for power control in modern high-performance processor architectures. Processor cores can be slowed down when not in full use, in an attempt to reduce power consumption and avoid overheating. In some cases, cores can be sped up for a limited time, in order to boost performance for cycle-hungry applications. On the other hand, low-cost processors that are preferred for embedded and low-performance systems can also use clock scaling in order to increase performance, especially if this is achieved in a

fairly cheap manner. In this section we will present how IPE-STA can be employed for such a purpose.

3.5.2 Scaling clock by opcodes

IPE-STA can be used to acquire timing information for each instruction separately and thus, we aim to use such information for adaptive clock scaling based on the instruction opcode. In order to validate our clock-scaling technique, we use a simple 64-bit six-stage pipelined processor architecture. More specific we opt for a a single-issue in-order-execute RISC-V Rocket core implementation [105].

We will refer to this implementation as *baseline processor*. In order to tighten the timing of the processor's functional units we also deploy a second implementation that utilizes pipelined functional units. As a result time consuming operations require more clock cycle to complete but they display lower latency. We will refer to this implementation as *pipelined execution* implementation. We classify the obtained results into 11 instruction classes as shown in table 3.1. Each instruction class contains a group of individual instructions with similar timing requirements. We also pinpoint the slowest pipeline stage in terms of delay, for every class. Finally, we assign *a worst-case delay* value to each class, which is the highest instruction delay in the corresponding group. The classes go as follows:

- The **Logical instruction class** which includes logical operations such as *and*, *ori* and *xor*.
- The **Shift instruction class** which includes shift operations such as *shift left logical* or *shift arithmetic*.
- The **Comparison instruction class** which includes *bit comparison* operations.
- The **Jump instruction class** which includes jump operations such as *jump and link*, *jump register* or *jump*.
- The **Multiplication instruction class** which includes *integer multiplication* operations.
- The **Division instruction class** which includes any *integer division* operations.
- The **Other arithmetic instruction class** which includes all *other integer arithmetic* operations except for multiplication and division, such as addition or subtraction.
- The **Memory access instruction class** which includes any memory access operation such as *load word* or *store byte*.
- The **FP Multiplication instruction class** which includes *floating point multiplication* operations.
- The **FP Division instruction class** which includes *floating point division* operations.
- The **Other FP arithmetic instruction class** which includes all *other floating point arithmetic* operations except for FP multiplication and division, such as FP addition or subtraction.

After studying the aforementioned instruction classes we observe the following:

- Each pipeline stage presents unique timing requirements depending on the instruction being executed.

TABLE 3.1: Analysis of the instruction classes of the RISC-V Rocket core architecture.

Instruction class	Slowest pipeline stage (critical stage)	Baseline worst case delay	Pipelined execution worst case delay
Logical	Execute stage	1.2 ns	1.2 ns
Shift	Execute stage	1.5 ns	1.5 ns
Comparison	Execute stage	1.5 ns	1.5 ns
Jump	Execute stage	1.1 ns	1.1 ns
Multiplication	Execute stage	2.9 ns	1.5 ns
Division	Execute stage	3.3 ns	1.1 ns
Other arithmetic	Execute stage	1.9 ns	1 ns
Memory access	Memory stage	3.9 ns	1.3 ns
FP Division	Execute stage	3.7 ns	1.3 ns
FP multiplication	Execute stage	3.2 ns	1.1 ns
Other FP arithmetic	Execute stage	3.0 ns	1 ns

- Some pipeline stages may produce error free results while utilizing higher clock frequencies than the others.
- The error free instruction execution is preserved if we satisfy the timing requirements of each individual pipeline stage for the executing instruction.

We deduce that we can dynamically adapt the clock period of the processor during the run time in order to achieve higher throughput, while also guarantying the error free instruction execution. To this end, we isolate the critical instructions, i.e. the instructions that constrain the clock frequency and we display the results we obtained for each implementation in table 3.2. We track down all critical instruction classes for our architecture and we assign a minimum operational clock period to each one of them. Due to the prior timing analysis, we are guaranteed that each critical instruction will execute without errors at the designated clock period. We also consider a typical clock period for each design which is suitable for the error free execution of non-critical instructions.

Our design focuses on letting the pipeline operate at high clock frequencies when critical instructions are absent. When a critical instruction is detected, we downscale the clock frequency, as soon as the critical instruction enters the pipeline stage which would otherwise cause a timing error. We refer to such stage as *critical stage*. Figure 3.1 shows an execution instance of a small instruction sequence on the Rocket core Baseline implementation. We track the minimum clock period with respect to the pipeline stages involved and we mark

TABLE 3.2: The clock periods for critical instructions along with the typical clock period for the Rocket core implementation.

Processor implementation	Critical instruction classes	Critical instruction operational clock period	Typical clock period
RISC-V Rocket core baseline implementation	Multiplication, Division, Memory access,FP Division, FP multiplication	4 ns	2 ns
RISC-V Rocket core pipelined execution implementation	Shift, Comparison, Multiplication	1.5ns	1.3ns

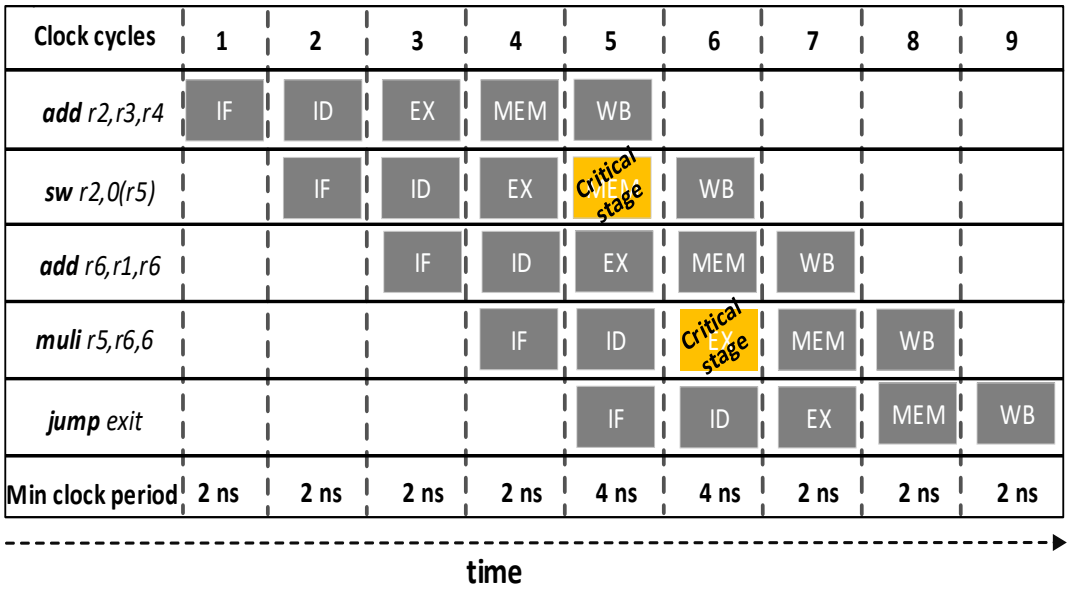

FIGURE 3.1: An instruction execution instance of the Rocket core implementation displaying the minimum operational clock period during each stage.

the critical stages that contribute to frequency downscaling. Under this premise, the critical stage is the slowest pipeline stage in which the clock frequency needs to be adjusted so that no timing errors to occur.In this example, the pipeline under examination may operate at higher clock frequencies during the 1st to 4th and 7th to 9th clock cycle, while the clock frequency must be lower at the 5th and 6th cycle.

3.5.3 Dynamic Clock Scaling Mechanism

We will now present in detail the circuit we designed to carry out the dynamic changes in the clock frequency. The circuit utilizes the information extracted from the timing analysis proposed in section 3.5.2, to decide whether the clock frequency should be changed or not. As the decision making will be occurring in real time, our design needs to employ reliability and speed. Figure 3.2 displays the designed clock control unit, which is responsible for the aforementioned task, deployed on the Rocket core implementation. It consists of an instruction snooping module and a clock selection module. In order to make the design nonintrusive for the processor architecture, we attach the two-module circuit at the side of the fetch and the decode stages of the processor pipeline. Note that the added circuit area shown is not in scale with the pipeline.

Instruction snooping module: To be able to change the clock frequency dynamically, we need information about the class of the instructions that are headed for execution. To this end, we implement an instruction snooping circuit that receives a copy of the instruction word coming out of the instruction cache. This circuit monitors the instructions fetched and tracks down their progress in the pipeline. Moreover, it utilizes lookup tables which contain both the critical instruction opcodes and the critical pipeline stage for each corresponding instruction as calculated in Section 3.5.2. Using this information, the circuit produces a logical output on whether the clock frequency must be changed, driving with that output the clock selection module.

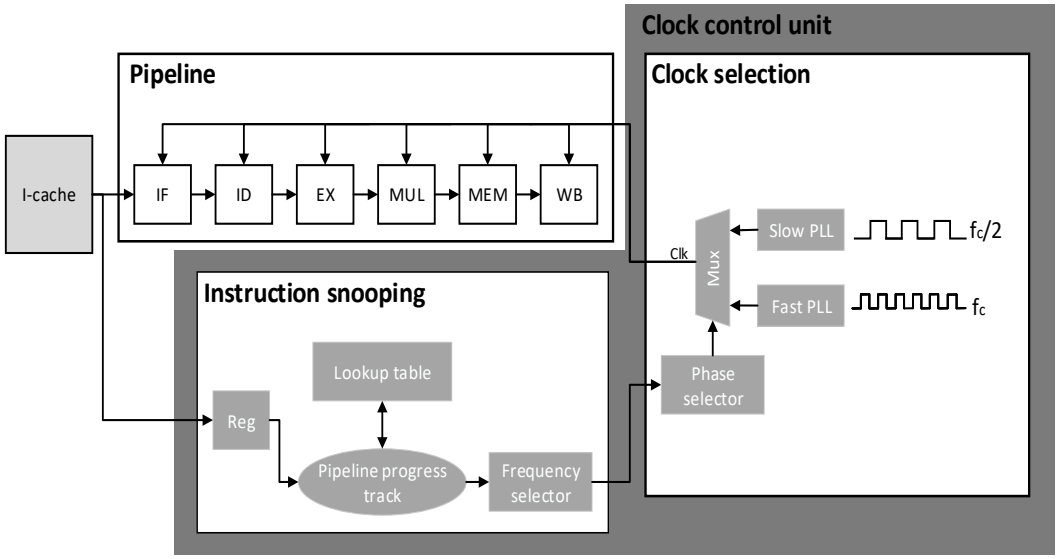

FIGURE 3.2: The clock control unit integrated in the rocket core.

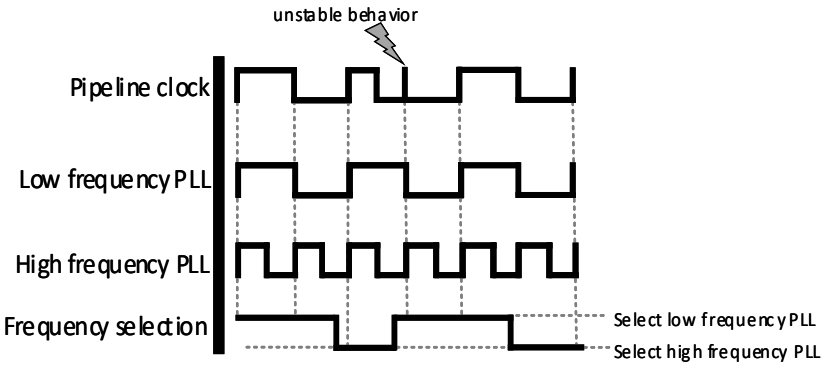

FIGURE 3.3: Unstable clock behavior due to subsequent clock selections.

Clock selection module: This module propagates the appropriate clock pulse selected by the frequency selector mechanism of the instruction snooping module. The clock selection module inputs are the frequency selection signal generated by the instruction snooping module and two PLL signals, one of high and one of low frequency. The frequency selection signal determines which pulse will be selected for the pipeline clock, when the instruction arrives at the critical stage. If frequency is indeed switched to low, the module must revert to the high frequency in the following cycle. The selection circuit is a simple multiplexor with insignificant contribution to the total delay of the module.

In the implementation of the second module, we observed that reverting to the original frequency may result in an unstable pipeline clock behavior, as shown in Figure 3.3. Such a phenomenon exists due to the frequency difference between the clocks and may prove catastrophic for the instruction execution. We address this problem in the way shown in Figure 3.4, by generating another low frequency pulse signal with a 180 degrees phase shift of the original. We also design a phase selector circuit which is responsible for selecting the appropriate phase when necessary. The phase selector is signaled by the frequency selector when a frequency scaling event is about to happen. It then designates the selected PLL phase so that no unstable behavior is exhibited. The complexity of the phase selection mechanism depends on the number of PLLs required.

In the case of the implemented Baseline RISC-V pipeline, where one period is an integer

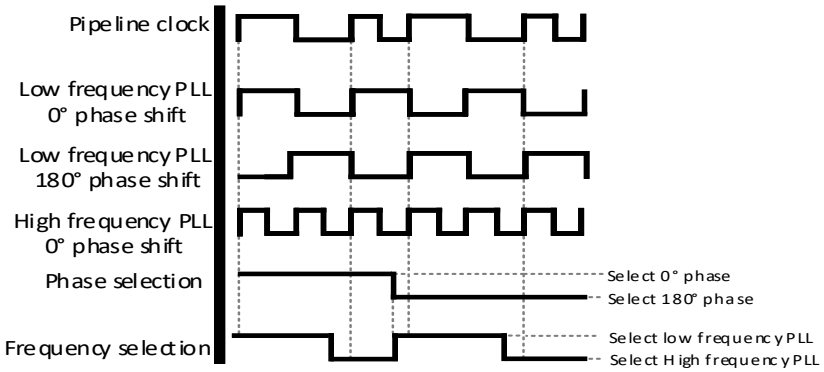

FIGURE 3.4: The clock instability compensation technique.

multiple of the other (2ns and 4ns), the generation of one additional shifted PLL resolves the problem. In other processor implementations though, additional PLLs may be needed, each with a specific phase shift, in order to enable the phase selector to compensate for all possible unstable behaviors of the pipeline clock. We adopt this approach as we acknowledge the need for a robust real time clock scaling mechanism. In contrast with [82] and [83] which manage to change the clock frequency for up to 7.5% of the core clock speed, we require much higher adaptation values. For that reason we do not change the clock frequency directly, instead we pre-generate the number of PLLs required and we proceed in selecting the appropriate candidate each time.

In general, when we have two clocks with a period ratio $m : n$, m and n being mutually primes, we need m PLLs for the phase shifts of the first and n PLLs for the phase shifts of the second clock, giving a total of $m + n$ PLLs. Such a solution to the possible instability problem serves as the most efficient in terms of performance. As we saw earlier, clock frequencies used are the highest possible, with longer clock periods just enough to cover the critical pipeline stage delay. But this choice may result in a large number of PLLs. A cheaper solution would be to always use a slow clock period that is a multiple of the typical period. In this way we would not need that many PLLs, sacrificing performance for simpler implementation. In some cases, like the one examined above, it occurs that the optimal performance solution coincides with the cheapest solution, but this is definitely not the general case though.

3.6 Implementation

3.6.1 RISC-V processor parameters

The parameters of the baseline processor architecture are displayed in figure 3.5. The processor supports in-order instruction issue and execute with 64-bit instruction length. The instructions have an opcode field length of 7 bits. The clock period of the processor implementation is defined by the slowest pipeline stage as described in section 3.5.2. It also employs a BTB of 512 entries using the g-share prediction mechanism. We have incorporated an L1 cache system to the processor; in particular a 4-way associative 16KB i-cache and a 4-way set associative 16KB d-cache with LRU replacement policy. The access time of the data cache is 4 and 7 clock cycles for the baseline and the pipelined execution implementations correspondingly, while the access time for the instruction cache is 1 clock cycle. Finally, the amount of PLLs required for the implementation of the IPE-STA methodology is 4 for the baseline and 18 for the pipelined execution RiscV implementations.

Processor Parameters shared on both implementations	
ISA : Risc V Implementation : Rocket core Instruction width : 64 bits Instruction issue : In order, single issue Pipeline depth : 6 stages	
I-cache	**BTB**
Associativity : 4-way Size: 16 KB Round trip : 1 cycle TLB size : 512 entries	Prediction mechanism : gshare History length: 9 bits Prediction levels : 2 Table entries : 512 entries
Baseline Processor Parameters	**Pipelined Execution Processor Parameters**
D-cache Associativity : 4-way Size: 16 KB Round trip : 4 cycles TLB size : 512 entries **Execute stage:** All operations : 1 cycle **Worst-case clock period :** 4ns	**D-cache** Associativity : 4-way Size: 16 KB Round trip : 7 cycles TLB size : 512 entries **Execute stage:** Multiplication : 2 cycles Division : 3 cycles Other arithmetic : 2 cycles FP Division : 3 cycles FP multiplication : 3 cycles Other FP arithmetic : 3 cycles **Worst-case clock period :** 1.5 ns

FIGURE 3.5: The configuration parameters of both processor implementations.

3.6.2 CAD toolflow and simulation

The experimental methodology used for timing analysis and design layout is described in figure 3.6. After rigorous consideration of many open-source simple processor cores that have been used in architecture-oriented research in the last decade, we have opted for the RISC-V Rocket Core processor implemented in Verilog for our experiments. We created the new version of the processor mentioned in section 3.5, which includes our clock control design, we then compiled and tested the circuit using a number of benchmarks, and produced a final evaluation of our ideas. In the rest of this section we describe all steps in detail. For the front-end design flow we used the *Synopsys Design Compiler* in conjunction with the *NanGate* 45*nm Open Cell Library*, whereas for the back-end place and route process we used the *Synopys IC Compiler*. In the sequel, we employed the *Synopsys PrimeTime* tool to apply the IPE-STA methodology on the generated post-placement netlist, also using the created *spef* files that contain the wire parasitic information. In the sequel, we designed the clock control unit as described in section 3.5.3 and integrated it to the RISC-V Verilog source code. We then followed the same sign-off flow in order to acquire a post-layout netlist, this time with the clock control module embedded in the design. Using the *Synopsys PrimeTime*, we generated the *sdf* file, which describes the cell and interconnection timing delays of the circuit. We subsequently performed a post-layout simulation using *Mentor Graphics Modelsim* with the sdf file to back annotate the design. In order to analyze the performance and power consumption of the system, we selected the *SPEC CPU2017 benchmark* suite and we utilized the RISC-V toolchain to compile each benchmark to generate the required binaries in accordance with the RISC-V architecture. We ran the experiments using the back-annotated netlist and we generated a *saif* file for each benchmark. *Saif* files contain information about the cell's

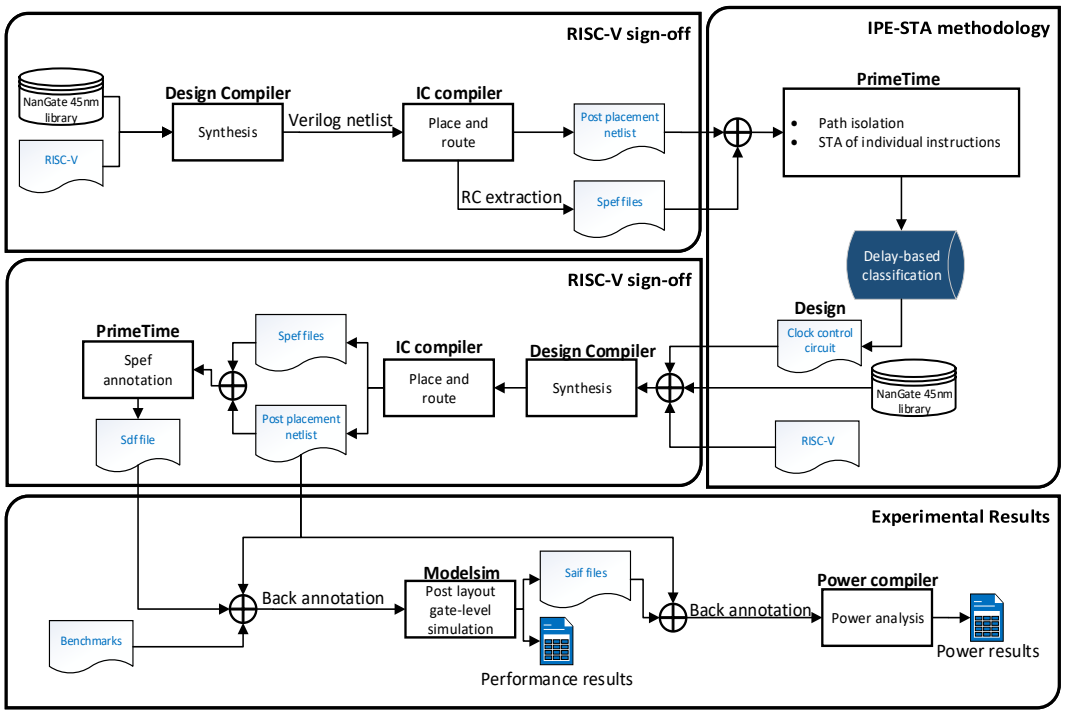

FIGURE 3.6: The CAD toolflow for the IPE-STA methodology.

switching activity and were used to back-annotate the design to generate power reports via *Synopsys Power Compiler*.

3.6.3 Clock tree synthesis

In order for the IPE-STA technique to function properly, a large number of clocks is required and thus, the synthesis and layout processes should be properly conducted to support such functionality. This necessity is derived by the fact that the processor clock is dynamically selected according to the information obtained by the clock control unit and any clock selection decision should be enforced within a very limited amount of time. In order to account for such a delay we design the clock selection unit to generate the outputs that control the clock selection process within the available timing margins. In this sense, we implement a low complexity and low latency control circuit capable of generating outputs with low delay, before the imminent clock pulse. As a result each decision on the dynamic clock frequency change is made within the timing margin available in order to properly distribute the clock pulse throughout the clock network in time. Further, end we opt to synthesize a single clock tree that reaches to every register of the design, as we consider the propagation of all the available PLLs very costly. In this way we avoid the expensive area, routing and energy costs of multiple PLL distribution, but we impose timing skew in the global clock tree network. Also local, cell level clock gating for heavily gated clock networks such as ours, comes with various challenges, as a previous works in [97] and [98] suggest, that are not within the scope of this work. Under this premise, our design utilizes 11 PLLs that are multiplexed into a global clock tree network which propagates the clock pulse to the RISC-V core pipeline. Previous works in [67] [99] and [100] have proven that multiplexing different PLLs is possible through a fast adaptive clocking circuit; while the PLLs are running at independent frequencies. In this scenario we identify two potential problems that may cause catastrophic consequences for the RISC-V pipeline timing, if left unresolved: The clock skew and clock jitter phenomena. Considering the clock skew of each PLL, we implement the clock control unit at the

base of the synthesized clock tree and thus, the globally synthesized clock tree drives the PLL by the clock control unit. As a result, we ensure that every PLL presents the same skew as we utilize one clock network for clock propagation in each RISC-V implementation. In order to account for the clock jitter phenomenon, we design the RISC-V implementations to be able to operate under a clock uncertainty of 10% by relaxing the timing requirements of the execute stage by the corresponding amount of time. As a result, our design may operate normally with out errors under a clock jitter of 10%. Generally, the synthesis and layout operations for designs with many PLLs are strenuous processes that require a lot of fine tuning and testing. That being said, a detailed synthesis and implementation discussion of clock tree networks are outside of the scope of this work.

3.7 Experimental evaluation

3.7.1 Normalized speedup

We have run the Spec2017 CPU benchmarks on the baseline processor implementation, on the pipelined execution implementation and on their corresponding BTWC versions. We present the normalized instruction throughput improvements we obtained according to our experiments in Figure 3.7 where results indicate an average performance increase in instruction throughput of 1.6 and 1.3 correspondingly. In the same figure we also present the appearance rate of critical instructions for each processor implementation. Further result analysis discloses the following information:

Firstly, designs with relaxed timing constrains benefit more from the IPE-STA methodology when compared with designs that display tighter timing requirements. This behavior is expected as the IPE-STA methodology exploits timing differences between individual processor operations. As a result, the more relaxed the system timing, the higher performance increase is achieved. Secondly, frequent appearance of critical instructions throttles the system's performance as the design is forced to operate under the worst-case clock period. As a result benchmarks that display low critical instruction appearance rates, also display higher throughput increase. In order to further assess the effectiveness of IPE-STA methodology we

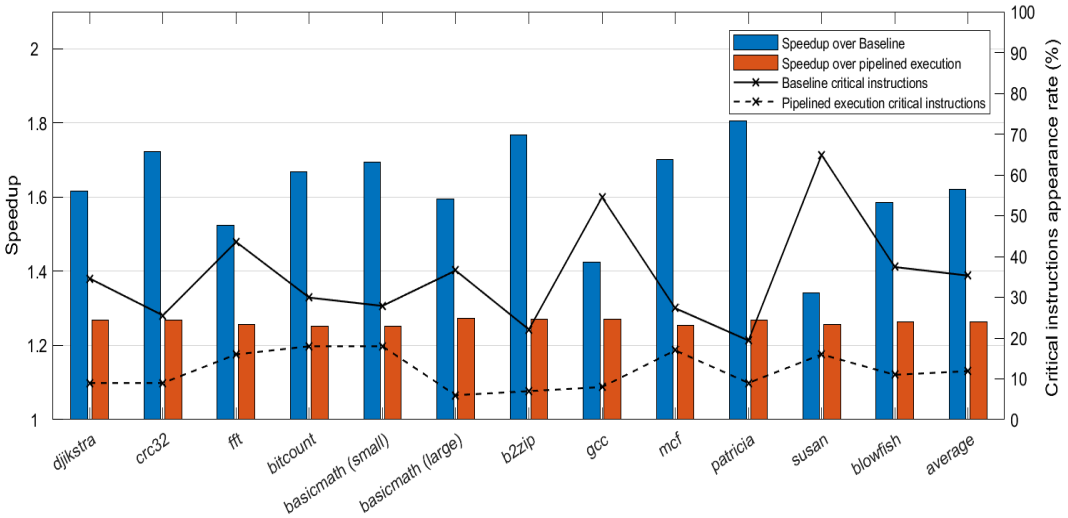

FIGURE 3.7: Normalized throughput improvement and critical instruction appearance rate of the proposed design methodology compared to the corresponding baseline processors.

compare the obtained throughput results with other state of the art timing speculation techniques. Tables 3.3 and 3.4 below demonstrate the outcomes of such comparison. Table 3.3 depicts the normalized throughput improvement of the application-adaptive guardbanding technique proposed by A. Rahimi et. Al. in [67]. We compare the IPE-STA methodology of both RiscV implementations with the best and worst performance-wise design corners explored by [67]. The application-adaptive guardbanding technique outperforms our methodology when it comes to the best-case design corners, but IPE-STA proves to be more efferent in terms of performance in worst-case design corners. Table 3.4 below displays the IPE-STA results in conjunction with Blueshift optimization as described by B. Greskamp et. Al in [72]. In this work the proposed TS methodology is applied on both Razor [61] and OpenSPARC T1 processors displaying significant performance improvements. According to Table 3.4 IPE-STA design approach achieves better throughput improvements if compared with Razor or OpenSPARC T1 processor when the baseline RiscV is considered. In contrast, the pipelined execution RiscV design is slightly behind the Razor processor in terms of performance, while it still surpasses the OpenSPARC T1 with the Blueshift design paradigm. IPE-STA comparison with state of the art TS methodologies highlights the competitive edge of our methodology as its performance is measured on average the same level if not above, compared to other TS approaches.

3.7.2 Normalized power consumption

Due to clock frequency scaling, our design often tends to operate at higher frequencies. As higher frequencies are more power hungry, we expect a higher power usage compared to the baseline processor. To verify that assumption, we measure the power consumption of the BTWC design and we compare it to its relevant baseline processors in Figure 3.8. Results show that the power consumption increase is higher for the benchmarks that present more opportunities for aggressive frequency scaling. Specifically, an increase of 4% to 36% in power consumption is observed, depending on each benchmark's capacity for frequency scaling. Nevertheless, by dividing the performance improvement over the power increase for

TABLE 3.3: Throughput improvement comparison between Application-adaptive guardbanding and IPE-STA.

Benchmark	**Adaptive Guardbanding Best/Worst design** corner [68]	**IPE-STA Baseline /IPE-STA Pipelined**
djikstra	1.87 / 1.36	1.61 / 1.28
patricia	1.89 / 1.38	1.8 / 1.26
susan	1.81 / 1.58	1.4 / 1.25
blowfish	1.84 / 1.35	1.6 / 1.25
Average	1.88 / 1.25	1.6 / 1.26

TABLE 3.4: Throughput improvement comparison between Blueshift OpenSPARC, Razor and IPE-STA methodology.

Benchmark	**Blueshift OpenSPARC [72]**	**Blueshift Razor [61]**	**IPE-STA Baseline /IPE-STA Pipelined**
b2zip	1.18	1.37	1.8 / 1.28
gcc	1.25	1.39	1.42 / 1.29
mcf	1.04	1.05	1.7 / 1.25
Average	1.15	1.27	1.64 / 1.273

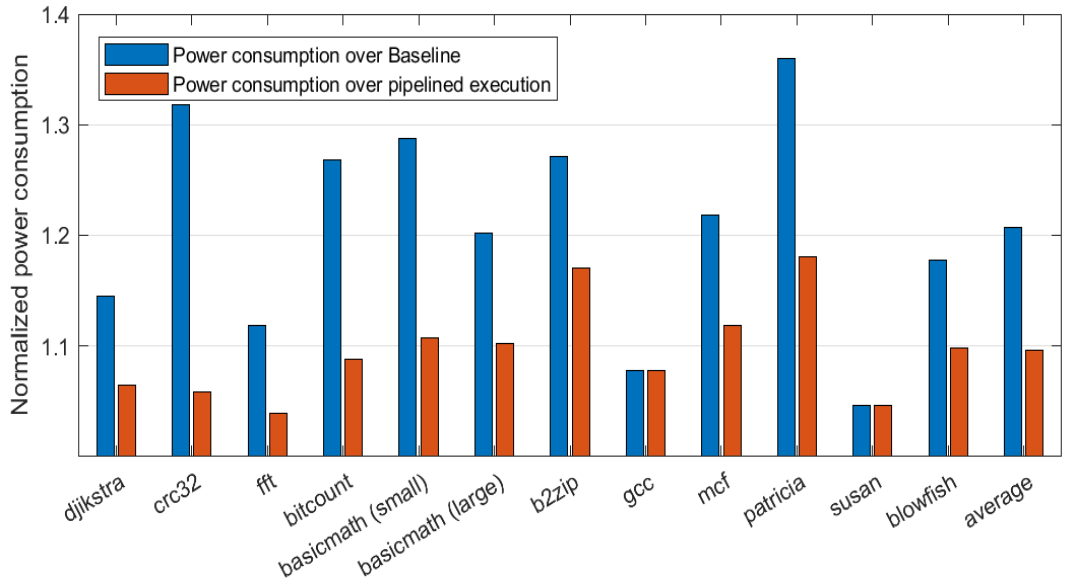

FIGURE 3.8: Normalized power consumption increase of the proposed methodology compared to the baseline processors.

each benchmark, we get an average of 3.7 improvement rate for the performance-to-power ratio, which is a quite significant overall improvement that we observe with our technique.

3.7.3 Overhead of the IPE-STA methodology

In order to properly evaluate the overhead of the IPE-STA methodology we measure both the IPE-STA design costs in terms of area and power complexity and the IPE-STA analysis cost in terms of time requirements. Regarding the design costs of the IPE-STA, we quantify the overhead in power and area of the post-layout implementation of the clock control circuit and instruction snooping modules. Such modules are the essential components of the IPE-STA design and are used as described in section 3.5. Table 3.5 lists the power and area requirements of the aforementioned modules, along with the power and area requirements of the baseline processors. We observe that the power overhead of the control unit is less than 2% of the total average power consumption of the baseline pipelines, while the area overhead almost 0.001%. Further, Table 3.6 depicts the area overhead comparison between the state of the art methodologies and the IPE-STA approach which is proposed in this work. We observe that the IPE-STA methodology achieves the least area overhead compared with the rest and thus, we conclude that the IPE-STA is well suited for low-end, low power pipelined processors.

Regarding the time requirements of the IPE-STA, we measure the amount of iterations and the amount of time required for the IPE-STA analysis to complete. The results we obtain are depicted in Table 3.7 in comparison with the standard STA and DTA methodologies. We

TABLE 3.5: The power and area overhead of the clock control and instruction snooping circuits in comparison to RiscV .

Implementation	Average power	Area
RiscV Baseline	$65.67mW$	$0.24mm^2$
RiscV Pipelined execution	$149.04mW$	$0.55mm^2$
Clock control and instruction snooping	$98.21uW$	$321um^2$

TABLE 3.6: Area overhead comparison between IPE-STA methodology and the state of the art.

Benchmark	Area overhead
iRazor [62]	13.6%
Application-adaptive guardbanding [68]	0.022%
TS Cache [74]	1.8%
Active management [82]	0.12%
DynOR [84]	$5-13\%$
Optimal In Situ Monitoring [89]	3.15%
Razor-Lite [110]	4.42%
Bubble Razor [111]	21%
IPE-STA (this work)	0.001%

define the iteration count as the amount of times the corresponding timing methodology in invoked in order to sufficiently cover the timing paths of the pipeline under examination. To this end, the DTA methodology examines every possible bit-transition and thus, it requires $2^{instruction_length}$ iterations (2^{64} for the 64-bit RiscV Rocket core implementations). On the other hand, the STA methodology examines the worst case scenario only, for each instruction path and thus, the required timing iterations are analogous to the complexity of the design. The IPE-STA methodology resides in between the DTA and STA approaches as the amount of the required iterations is depended on the bit length of the opcode field of the ISA, as discussed in section 3. As a result, the IPE-STA requires 2^7 iterations in order to properly analyze the timing paths of the RiscV Rocket core pipeline, as the opcode field of the rocket core ISA is 7 bits. In order to evaluate the time requirements of each methodology, we run the DTA, STA and IPE-STA approaches on the same RiscV rocket core pipeline using an Intel i7 coffee lake processor with 6 cores and 16 GB of DDR4 DRAM. Results indicate that the STA analysis finishes in a 20 second period of time while the IPE-STA methodology requires 5 minutes. In contrast, the DTA requires over 100 hours to finish and thus, it is considered time costly for timing analysis in processor pipelines. We conclude that the IPE-STA methodology manages to efficiently manage the trade-offs between STA and DTA as it provides detailed timing reports for each ISA-supported instruction while also requiring a reasonable time to finish.

3.7.4 PVT tolerance considerations

In order to evaluate our design we utilize a single design corner that operates in 0°and 0.72 V. We select the aforementioned corner as the 0.72 V is considered low power pipeline operation and thus, it stays within the scope of this work. We also set a clock uncertainty of 10% to compensate for the process variation effect which may induce clock jitter and clock uncertainty to the integrated circuit. Evaluating the proposed methodology with a full range

TABLE 3.7: Time requirements of DTA, STA and IPE-STA to complete the timing analysis of RiscV pipeline.

Implementation	Iterations Theoretical	Iterations (RiscV)	Time (RiscV)
DTA	$2^{instruction_length}$	2^{64}	Over 100 hours
STA	Supported instructions	215	20 seconds
IPE-STA (this work)	2^{opcode_length}	2^7	5 minutes

of dynamic variations as well as static process parameters variations is possible but the IPE-STA analysis should be conducted independently for each individual design corner. A higher voltage than 0.72 volts would result in shorter delay instruction paths, while lower operating temperatures would lead to higher delays in the low-voltage region of 0.72 volts as previous work in [101] demonstrates. To this end, the IPE-STA analysis should be conducted for each design corner in order to extract the exact timing information for the corresponding operating Voltage and Temperature values. On the other hand, the process variation effect can be emulated by setting clock jitter and clock delay values, similarly to our approach. The methodology of the IPE-STA does not require any modifications in order to function properly within different PVT effects and thus, it can produce accurate timing results given the exact operating condition of the integrated circuit.

Chapter 4

Near data processing for low power architectures

4.1 Introduction

In this chapter we discuss the application of the IPE-STA methodology, as described in chapter 3, on the domain of low-power NDP computer architectures. To this end, we employ a BTWC-NDP co-design approach to develop a low-power, low-end pipeline from the ground up to facilitate the IPE-STA methodology. We employ such an approach considering the architectural requirements of low-power pipelines in terms of area, energy consumption and logic complexity. To this end, we implement a low-end pipeline on the logic layer of an HMC DRAM and we conduct a design space exploration in order to evaluate the applicability of the IPE-STA methodology to NDP designs. We should note that in this work we also consider the power and area constrains of the logic layer of the HMC as defined by HMC consortium [33]. In order to evaluate our methodology we opt for the same baseline processor as the one used in chapter 2 and thus, we are able to draw solid conclusions on the efficiency of the proposed co-design approach when compared with the high performance computing designs.

4.2 NDP System Architecture

The proposed BTWC-NDP co-design is consisted of a host system and of an HMC DRAM. For the host system we implement a RISC-V BOOM processor core which facilitates the core pipeline of the host system processor; whereas the HMC DRAM provides both DRAM functionalities for the host system and also facilitates the PIM core responsible for the NDP. The PIM core is where the processing-in-memory takes place and thus, it also facilitates the co-design approach we propose in this work.

4.2.1 Host system architecture

The host system architecture is depicted in Figure 4.1. We employ a high performance RISC-V BOOM [47] out of order (OoO) core that supports 4-wide instruction issue width.

The RISC-V BOOM architecture is a synthesizable and parameterizable open-source RISC-V 9-stage pipeline that includes the following stages: Instruction fetch (IF), branch prediction (BP),instruction decode (DEC), reorder buffer update, instruction dispatch, instruction issue, register file read, execute pipeline and memory access.

We extend the RISC-V ISA in order to include the necessary functionalities to support the NDP processing paradigm. To this end we implement *jump-and-link-PIM* (JalPim), which is an instruction that behaves as the original jump-and-link (Jal) instruction and thus, it triggers a function call. A key difference between Jal and JalPim is that the former initiates a function call that executes on the RISC-V core pipeline while the latter triggers a function

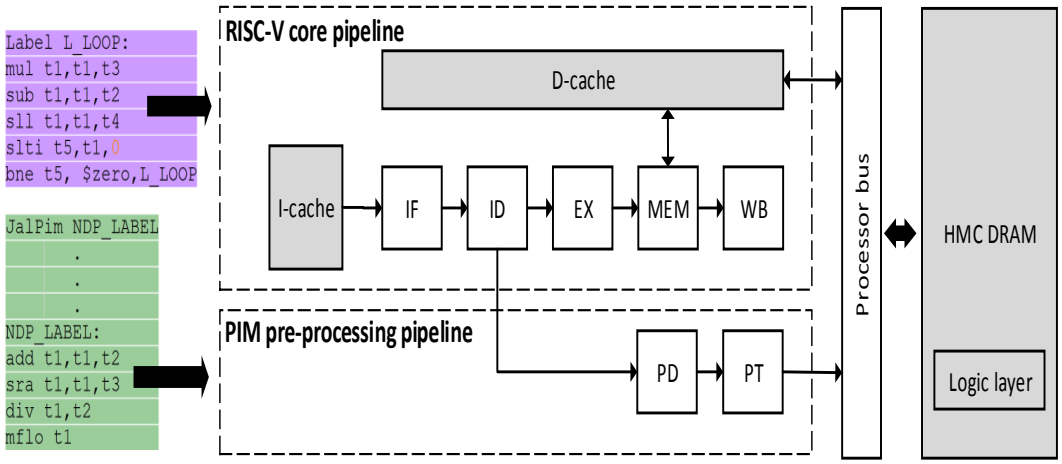

FIGURE 4.1: The host system architecture composed by the BOOM core and the PIM pre-processing pipeline.

call that executes on the PIM core. When the JalPim is evoked, the ID stage of the BOOM core pipeline propagates a stall signal to the rest of the RISC-V pipeline to disable further instruction execution. Then, the PIM pre-processing pipeline is enabled, which is responsible for pre-processing the instructions of the JalPim function before dispatching them to the PIM core. We design the PIM pre-processing pipeline to include the following stages:

PIM-Decode (PD): The PD decodes the instructions of the JalPim function and generates the necessary ALU/FPU signals for the instruction execution on the PIM core. We perform the PIM instruction decoding process on the core pipeline instead of the PIM core in order to reduce the power consumption and the area requirements of the PIM core as the area and power budget is limited on the HMC logic layer [17] [33].

PIM-Transfer (PT): The PT stage transfers the decoded instructions to the PIM core for execution. The PIM core is located at the logic layer of the DRAM and thus, the PT utilizes the processor bus to transmit the corresponding instructions to the PIM core. In order to speed up the instruction transfer operation, the processor bus is dedicated to the PT transfer once the PT stage commences and thus, no other DRAM access operations are allowed.

We employ the PIM pre-processing pipeline and the JalPIM instruction to maintain interoperability between the RISC-V BOOM core pipeline and the PIM core. In this sense, the PIM pre-processing pipeline acts as an abstraction layer that hides the underlying complexity of the PIM core from the user. Also, the PIM core supports the RISC-V ISA without any additional extensions and users are capable of using the RISC-V instructions for NDP processing without requiring any further ISA extension.

4.2.2 PIM core architecture

Figure 4.2 depicts the architecture of the PIM core which is implemented on the logic layer of the HMC DRAM and is responsible for executing the instructions of the JalPIM function call. Under this premise, we deploy a simple, low power RISC-V superscalar pipeline capable of fulfilling the power and area requirements imposed by the HMC consortium [33]. We opt for a lightweight RISC-V pipeline design, to stay in the low power domain, to ensure that our design is well within the power and the area budget of the logic layer of the HMC and also to achieve interoperability with the host system. In order to balance the power to performance trade offs we employ data forwarding functionalities and a 2 issue-width depth while we

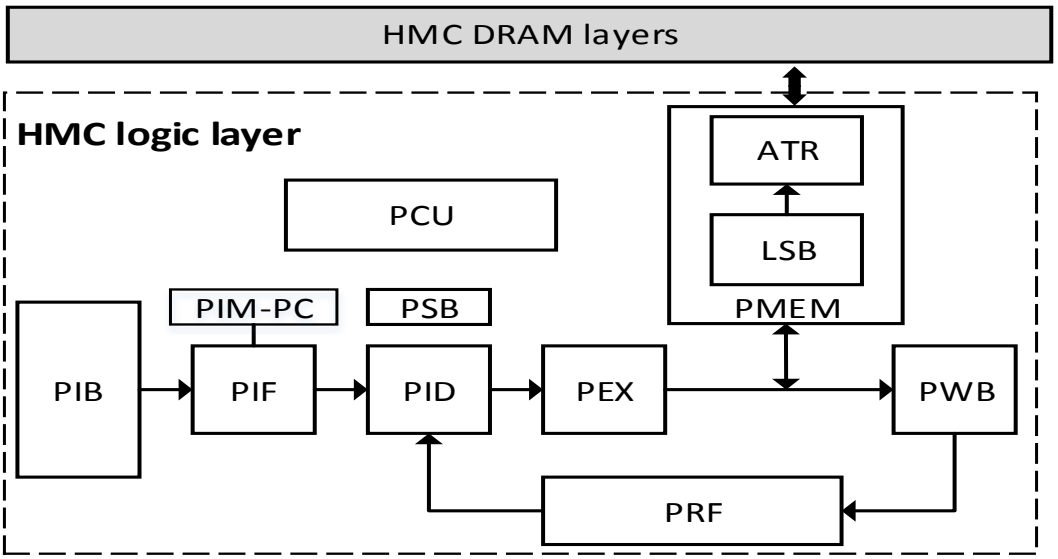

FIGURE 4.2: The PIM core architecture

omit complex performance optimizations such as speculative execution, branch prediction or larger pipeline depth. Below we discuss the details of the pipeline stages of the PIM core.

PIM Instruction Buffer (PIB): PIB is a simple buffer that stores the instructions which are headed for execution at the PIM core. Such instructions are already decoded by the PD stage of the PIM pre-processing pipeline of the host system and they are dispatched to the PIM core via the PT stage thought the processor bus, as described above. We opt for a simple PIB design to avoid complex structures such as an I-cache, that require more power and area to operate.

PIM Instruction fetch (PIF): The PIM instruction fetch stage utilizes a PIM program counter (PIM-PC) to fetch the next two instructions from the PIB. The instructions are then headed to the PID stage. The PIF is also charged with updating the PIM-PC value accordingly, to refer to the next two instructions for the following clock cycle fetch operation.

PIM Register file (PRF): The PIM core employs a register file module of 32 registers. Although the PRF increases the power consumption on the PIM core, we consider such module integral to the PIM architecture for preserving the interoperability with the RISC-V core pipeline. To this end, registers used by the RISC-V workloads are mapped on the PRF without requiring any further modifications. Also the PIM core does not employ any D-cache hierarchy due to the cache large power and area requirements that impose a problem to the limited area and power budget of the HMC logic layer and thus, instruction outputs would be otherwise stored to the DRAM if the PRF is not employed. In this sense, PRF also decreases the data traffic to the DRAM by storing the results of the executing instructions.

PIM Instruction Dispatch (PID): The PID stage reads the required registers from the PRF, performs sign extension and dispatches the decoded instructions fetched by the PIF stage to the PEX or PMEM stages along with their corresponding operands.

PIM Execute (PEX): The PEX stage is responsible for the instruction execution according to the operands received by the PID stage. PEX supports the logical, shift, branch and arithmetic operations of the RISC-V ISA.

PIM Memory access (PMEM): PMEM is a two-stage operation, coordinating the HMC DRAM access, i.e. the load and store operations. To this end, PMEM utilizes a load/store buffer (LSB) that temporarily queues the DRAM read and write requests. Such requests are stored to the buffer until they are handled by the HMC DRAM. In the sequel the buffer dispatches the DRAM requests to the address translation (ATR) stage which translates the

Algorithm 4 PCU adaption mechanism.

```
while always do
  Check the amount of pending DRAM requests
  if Pending percentage < HML Threshold then
    Enter Regular mode
  else if Pending percentage >= HML Threshold then
    Enter HML mode
    Save the pipeline state to the PSB
    Check the PIB for instructions without dependencies
    if no instructions remain then
      Load the pipeline state from the PSB
      Stall the pipeline
    end if
  end if
end while
```

request addresses to physical memory address. The resulting requests are sent to the HMC vault controllers which access the HMC DRAM layers and handle the read/write operations. In order to avoid complex cache coherence mechanisms, we consider the DRAM regions accessed by the PIM core uncachable as suggested in [11] and [25].

PIM Write back (PWB): The PWB stage writes back the PEX or PMEM results to the corresponding registers in the PRF.

PIM Control unit (PCU): The PCU coordinates the PIM pipeline by selecting the operating mode under which the PIM core functions. We design the PIM core to be able to function in two operating modes: regular and high memory load (HML). Previous work in [102] [103] demonstrate that the HMC DRAM read/write delay increases as the percentage of pending read/write requests increases too. To this end a read/write request may take from some ns up to several μs [103] depending on the amount of pending DRAM operations. The purpose of the HML mode is to hide the underlying delay of the HMC DRAM by enabling the pipeline to utilize the idle clock cycles during which the PIM core awaits the DRAM responses.

Algorithm 4 depicts how the PCU adapts to the amount of pending DRAM requests by choosing the PIM core operating mode. Under this premise, the PCU checks the percentage of pending HMC DRAM requests at every clock cycle. If this value is less than a pre-defined threshold, then the PIM core enters the regular operating mode. In regular mode the PIM core executes the instructions stored at the PIB while the PCU is responsible for identifying the dependencies between the instructions fetched by the PIF circuit and the instructions that execute in the PEX and PMEM stages of the PIM pipeline. The PCU resolves the instruction dependencies by either forwarding or by pipeline stalling mechanisms. On the contrary, if the percentage of pending DRAM requests surpasses the threshold, the PCU enters the HML operating mode. In HML the PCU saves the state of the PIM pipeline, i.e. the PIM-PC value and the outputs of each pipeline register, to the PIM state buffer (PSB). The PSB size requirements are trivial due to the small size of the PIM pipeline and thus, the PSB does not impose significant area or power overheads on our design. In the sequel the PCU searches the PIB for any instructions that can be executed without requiring any inputs from the pending DRAM requests or from the instructions that are saved to the PSB. The PIF changes the PIM-PC value accordingly and proceeds into fetching the aforementioned instructions, if any. Thus the PIM pipeline may continue the instruction execution process until the DRAM requests are handled. If the data independent instructions of the PSB are exhausted and the instruction execution cannot proceed, the PCU restores the PIM pipeline to its initial state by

loading the corresponding data from the PSB and then stalls instruction execution until the HMC pending operations complete and thus, lowers the power consumption of the pipeline.

4.3 BTWC-NDP co-design methodology

4.3.1 Application of IPE-STA to the PIM core

Since the IPE-STA methodology which is described in chapter 3 can provide timing information for each pipeline instruction separately, we use such information for a clock scaling methodology based on the instruction opcode. To this end, we apply the IPE-STA methodology to the post-layout implementation of the PIM core architecture described in Section 4.2. We opt to tighten the timing of the PIM core's functional units by also deploying a modified PIM core that utilizes pipelined functional units. As a result, time consuming operations require more clock cycles to complete, but they display lower latency per pipeline stage. In order to evaluate our design in different supply voltage options, we also implement three different power supply configurations for the PIM core, i.e. $0.72V$, $0.81V$ and $0.99V$. More details on the implementation decisions are discussed in the next section.

The obtained results by the application of IPE-STA has led to the classification of instructions into 11 classes, similarly with the chapter 3:

- The **Logical instruction class** which includes logical operations such as *and, ori and xor*.
- The **Shift instruction class** which includes shift operations such as *shift left logical* or *shift arithmetic*.
- The **Comparison instruction class** which includes *bit comparison operations*.
- The **Jump instruction class** which includes jump operations such as *jump register or jump*.
- The **Multiplication instruction class** which includes *integer multiplication* operations.
- The **Division instruction class** which includes any *integer division* operations.
- The **Other arithmetic instruction class** which includes all other integer arithmetic operations except for multiplication and division, such as *addition or subtraction*.
- The **Memory access instruction class** which includes any memory access operation such as *load word or store byte*.
- The **FP Multiplication instruction class** which includes *floating point multiplication* operations.
- The **FP Division instruction class** which includes *floating point division* operations.
- The **Other FP arithmetic instruction class** which includes all other floating point arithmetic operations except for FP multiplication and division, such as *FP addition or subtraction*.

Each PIM core instruction class contains a group of individual instructions that exhibit similar timing requirements. Table 4.1 shows the timing results of the analysis on all variations of functional unit implementation and supply voltage. In particular, we track down the slowest pipeline stage of each class in both non-pipelined execution (NPE) and pipelined

execution (PE) PIM core designs. Finally, we assign a "worst-case delay" value to each class, which is the highest instruction delay in the corresponding group.

By studying the classified instruction classes we observe that pipeline stage presents unique timing requirements depending on the instruction being executed. We should also note that the slowest pipeline stage is the execute stage as previous work in [67] also highlights. Further, some pipeline stages may produce error free execution results while utilizing higher clock frequencies than the others. Such a free instruction execution is guaranteed if we satisfy the timing requirements of each individual pipeline stage for the corresponding executing instruction.

By using the aforementioned information we can dynamically increase the clock frequency of the PIM core at run time in order to achieve higher throughput, while maintaining the error free instruction execution. To this end,we track down all instruction classes of the PIM core and we assign a minimum operational clock period to each one of them according to their timing requirements. In this sense, our approach treats the instruction execution sequence as consecutive opcode alterations and thus, we adapt the clock frequency to meet the timing requirements of each executing instruction. When an instruction with large path delay is met, we scale down the clock frequency, as soon as the instruction reaches the pipeline stage which would otherwise cause a timing error. Figure 4.3 shows an execution instance of five instructions on the PIM core $0.99V$ NPE implementation. We track the maximum delay of each clock cycle with regard to the pipeline stages involved, and we mark the pipeline stages that contribute to frequency down-scaling. In this instance, the pipeline under examination may function at lower clock periods during certain clock cycles according to the worst-case delay of each active pipeline stage. In this sense, the operational clock period is 1.1 ns for the 3rd cycle, 2.7ns for the 4rth and 1.5ns for the 5th. We mark with black color the PIM core stages with the highest stage delays and thus, force the clock to adapt to their timing requirements in order to ensure error-free instruction execution.

4.3.2 PIM core microarchitecture with adaptive clock scaling

In order for the PIM core to support the adaptive clock scaling mechanism that is driven by the IPE-STA technique, we design a clock control unit (CCU) that carries out the dynamic changes in the clock frequency. This unit utilizes the information extracted from the IPE-STA, to decide whether the clock frequency should be adapted or not. Figure 4.4 depicts the CCU, which is responsible for the aforementioned task, deployed on the HMC logic layer. It consists of an instruction class snooping module, a decision logic circuit, a lookup table and a number of PLLs.

TABLE 4.1: Instruction class IPE-STA analysis of PIM core architecture for different supply voltages

Instruction class	**Slowest pipeline stage**	$0.72V$ **NPE - PE**	$0.81V$ **NPE - PE**	$0.99V$ **NPE - PE**
Logical	PEX	1.6 ns - 0.8 ns	1.4 ns - 0.7 ns	1 ns - 0.5 ns
Shift	PEX	1.8 ns - 0.9 ns	1.5 ns - 0.8 ns	1.2 ns - 0.6 ns
Comparison	PEX	1.5 ns - 0.8 ns	1.3 ns - 0.7 ns	1.1 ns - 0.6 ns
Jump	PEX	1.2 ns - 0.6 ns	1 ns - 0.5 ns	0.8 ns - 0.4 ns
Multiplication	PEX	3.8 ns - 1.9 ns	3.2 ns - 1.6 ns	2.7 ns - 1.4 ns
Division	PEX	5 ns - 2.5 ns	4.2 ns - 2.1 ns	3.3 ns - 1.7 ns
Other arithmetic	PEX	1.8 ns - 0.9 ns	1.5 ns - 0.8 ns	1.1 ns - 0.6 ns
Memory access	ATR	2.2 ns - 1.1 ns	1.9 ns - 1 ns	1.5 ns - 0.8 ns
FP Division	PEX	4.7 ns - 2.4 ns	3.7 ns - 1.3 ns	3 ns - 1.5 ns
FP multiplication	PEX	3.7 ns - 1.9 ns	3.2 ns - 1.6 ns	2.2 ns - 1.1 ns
Other FP arithmetic	PEX	2.5 ns - 1.3 ns	2 ns - 1 ns	1.3 ns - 0.7 ns

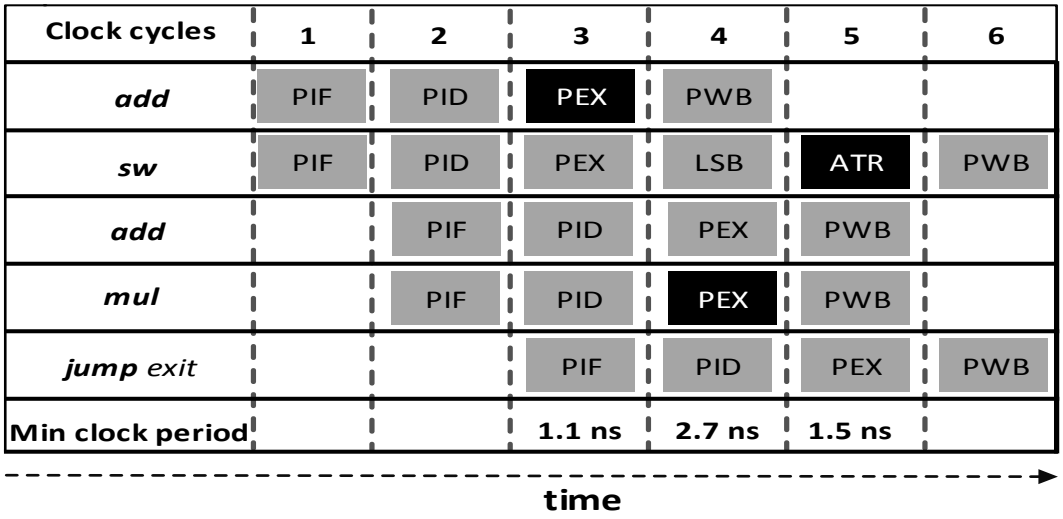

FIGURE 4.3: An instruction execution instance of the PIM core depicting the minimum operational clock period during each stage.

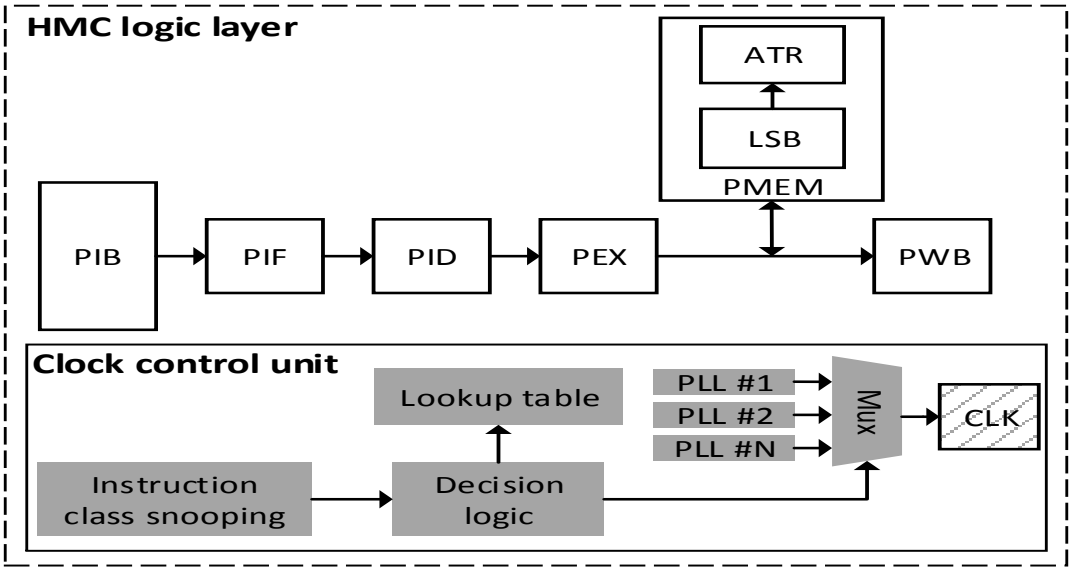

FIGURE 4.4: The CCU implemented on the PIM core in the HMC logic layer.

Instruction class snooping: The instruction class snooping circuit monitors the PIM core pipeline and tracks down each pipeline stage separately. In order to boost the circuit's ability to identify the executing instruction classes, we also modify the PD stage of the PIM pre-processing pipeline of the host system. More specific, we enable the PD stage to not only decode the fetched instructions but also to classify them into the instruction classes mentioned in Section 4.3. There are 11 existing instruction classes, and thus 4 additional bits are required for each decoded instruction in order to represent its class. The information about the class that each instruction belongs to, is forwarded through the PT stage of the PIM pre-processing pipeline to the PIM core. There, such information is stored along with the instruction operands at the PIB and is forwarded on each pipeline register during the instruction execution process. Under this premise, the instruction class snooping circuit may quickly identify the instruction class that currently exists within each PIM core pipeline stage.

Decision logic: The decision logic circuit utilizes the information about the identified

instruction classes provided by the snooping module and designates the clock period of the PIM core for the next clock cycle. In order to facilitate such functionality, the decision logic uses lookup tables that withhold the timing requirements of each instruction class according to the IPE-STA analysis we proposed in chapter 3. The decision logic utilizes such information and generates a control signal to a multiplexer that selects the appropriate PLL for the current clock cycle. Each PLL is running at independent frequencies, and a multiplexer quickly switches between them in a single cycle, resulting in ultra-fast frequency changes as in [67].

4.4 Implementation

4.4.1 Design space exploration and parameter considerations

Table 4.2 depicts the design parameters for the RISC-V BOOM core pipeline, the PIM core and the HMC implementations. For the PIM core we employ a superscalar architecture with an issue width of 2 and we implement both integer and floating point arithmetic. More specifically, we deploy 2 ALUs, 2 multiplication and 1 divider circuits for both integer and floating point operations, resulting in a total of 10 functional units. We also implement 6 PIM designs that operate in various supply voltages ($0.72V$, $0.81V$ and $0.99V$) and facilitate pipelined (PE) and non-pipelined (NPE) functional units, as described in the previous section. We use the following notation for referencing the PIM designs: PIM-1 ($0.72V$ NPE), PIM-2 ($0.72V$ PE), PIM-3 ($0.81V$ NPE), PIM-4 ($0.81V$ PE), PIM-5 ($0.99V$ NPE) and PIM-6 ($0.99V$ PE). We set the PCU HML threshold, to 53% due to the fact that beyond this limit the HMC DRAM latency dramatically increases [102].In order to maximize the RISC-V BOOM core pipeline performance,we set its clock frequency to 800 MHz. On the other hand, the PIM cores operate under an adaptive clock frequency of range 200MHz - 1.5GHz, depending on

TABLE 4.2: NDP design parameters

RISC-V Rocket pipeline		**PIM core**	
ISA	RISC-V	ISA	RISC-V
Pipeline width	4	Pipeline width	2
Pipeline depth	5	Pipeline depth	6(MEM), 4 (OTHER)
Instruction width	64-bits	Instruction width	64-bits
I-cache	4-way, 8 KB	HML threshold	53%
D-cache	4-way, 8 KB	PIB size	512
Branch prediction	g-share	LSB size	128
BTB size	256 entries	PSB size	10
TLB size	512 entries	PRF size	20
Clocks	1	Clocks	11
Clock frequency	200 MHz	Clock frequency	200 MHz-1.5GHz
Supply voltage	$0.72V$	Supply voltage	$0.72V$, $0.81V$, $0.99V$
		HMC DRAM	
	Organization	2 GB, 4 layers	
	Bus Width	128 bits	
	DRAM timing	tCK=1.5ns, tRAS=12ns	
		tRCD=8ns tCAS=4ns	
	Serial links	160 GBps,6-cycle latency	

the executing instruction timing requirements and the supply voltage of the PIM implementation. In this sense, the RISC-V BOOM host system utilizes one clock to operate, while the PIM cores require an amount of clocks equal to the amount of the instruction classes, i.e. 11.

4.4.2 CAD toolflow and simulation

The design toolflow used for the IPE-STA application on the PIM core is described in Figure 4.5. We use Verilog HDL to design the PIM core, the Synopsys Design Compiler with NanGate 15nm Open Cell Library [104] for the PIM core logic synthesis and the Synopys IC Compiler for place and route. In the sequel we apply the IPE-STA technique on the post-layout netlist of the PIM core, as described in section and 4.3. For that reason we evoke the *case_analysis* option of the Synopsys PrimeTime tool under which we can set circuit inputs at fixed voltage values and then perform STA, given the affixed inputs. After the IPE-STA completes, we utilize the resulting instruction classes to design the CCU as described in Section 4.3. We then integrate the CCU into the PIM CORE design and we place the PIM core pipeline on the logic layer of the HMC DRAM. Then we repeat the sign-off process to obtain the post-layout version of the NDP design and we use the back annotated netlist to evaluate our methodology. The evaluation is conducted by performing gate-level simulations using the back annotated netlist with the ModelSim tool.

4.4.3 Adaptive clock scaling with multiple clocks

In order for the IPE-STA technique to function properly, a large number of clocks is required and thus the synthesis and layout processes should be properly conducted to support such functionality. Previous works in [67] [99] and [100] have proven that multiplexing different PLLs is possible through a fast adaptive clocking circuit; while the PLLs are running at independent frequencies. Under this premise, our design utilizes 11 PLLs that are multiplexed into a global clock tree network which propagates the clock pulse to the PIM core pipeline. In this scenario we identify two potential problems that may cause catastrophic consequences for the PIM pipeline timing, if left unresolved: The clock skew and clock jitter phenomena.

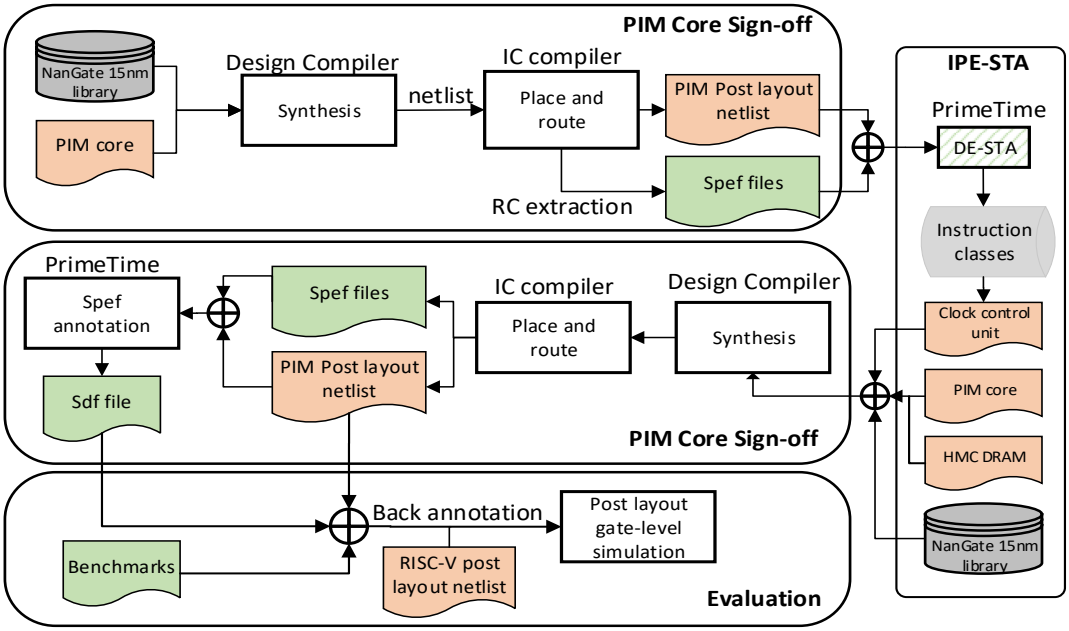

FIGURE 4.5: Toolflow of the IPE-STA methodology for the PIM core

Considering the clock skew of each PLL, we implement the clock control unit at the base of the synthesized clock tree and thus, the globally synthesized clock tree drives the PLL by the clock control unit. As a result, we ensure that every PLL presents the same skew as we utilize one clock network for clock propagation in each PIM implementation. In order to account for the clock jitter phenomenon, we design the PIM implementations to be able to operate under a clock uncertainty of 10% by relaxing the timing requirements of the execute stage by the corresponding amount of time. As a result, our design may operate normally with out errors under a clock jitter of 10%. Generally, the synthesis and layout operations for designs with many PLLs are strenuous processes that require a lot of fine tuning and testing. That being said, a detailed synthesis and implementation discussion of clock tree networks are outside of the scope of this work.

4.4.4 Area and power budget

NDP design constrains emerge from the limited area and power budget of the HMC logic layer. Table 4.3 depicts the area and power requirements of the RISC-V BOOM pipeline and PIM core implementations. More specifically,our design utilizes a 1 mm^2 RISC-V BOOM core pipeline and a 0.004 mm^2 RISC-V pre-processing pipeline as a host processor with a combined power consumption of 0.2mW. We should note that the RISC-V PIM pre-processing pipeline imposes less than a 0.3% area and 1% power overhead to the RISC-V BOOM core pipeline. For the PIM core area and power limitations, the HMC memory consortium [33] specifies that the maximum power consumption of the HMC logic is 5W while the maximum area budget is 7mm^2 and thus, our design operates well within the required budget. Further, the CCU of the PIM implementations imposes less that 1% power and 0.1% area overhead to the PIM cores. We should note that the small PIM core size is attributed to the lack of instruction and data caches and to the lack of completed OoO mechanisms. Table 4.3 also depicts the operating clock frequencies of each PIM core implementation which are designated by the IPE-STA analysis and by the supply voltage of each design.

4.5 Experimental evaluation

4.5.1 Workload characterization

We evaluate our design using several benchmarks suitable for a wide range of applications. Table 4.4 depicts the benchmarks used for the evaluation process. Under this premise, we use the specCPU 2017 suite [50], the Google's Inception V3 deep neural network training [106], machine learning and I/O benchmarks [107] [108] and big data benchmarks [109]. We run

TABLE 4.3: Area and power requirements of the RISC-V and PIM core implementations

Implementation	Area	Power	Clock (MHz)
RISC-V BOOM core pipeline	1 mm^2	0.2 W	1000
RISC-V pre-processing	0.0004 mm^2	10 uW	1000
CCU	51 μm^2	12 uW	-
PIM-1	0.03 mm^2	2 mW - 8 mW	200 MHz - 750 MHz
PIM-2	0.04 mm^2	4 mW - 11 mW	400 MHz - 750 MHz
PIM-3	0.03 mm^2	2.5 mW - 12 mW	240 MHz - 900 MHz
PIM-4	0.04 mm^2	5.5 mW - 14 mW	470 MHz - 1 GHz
PIM-5	0.03 mm^2	3 mW - 18 mW	300 MHz - 1.2 GHz
PIM-6	0.04 mm^2	7 mW - 23 mW	660 MHz - 1.5 GHz

TABLE 4.4: Workload characterization

Benchmark	Kernel	Type	PIM execution
Leela [50]	K1	AI: Monte Carlo tree search	75%
x264 [50]	K2	Video compression	80%
xz [50]	K3	General data compression	72%
Inception v3 [106]	K4	DNN training	78%
XML Parsing [107]	K5	Parser	73%
Azure Table Lookup [107]	K6	I/O	75%
K-means [108]	K7	Machine learning	76%
PageRank [108]	K8	Web search engine	74%
FFT [109]	K9	Signal processing	77%
Connected Component [109]	K10	Graph processing	73%

each benchmark kernel on both RISC-V BOOM core pipeline and PIM implementations on order to compare our findings. We also note the percentage of the total instructions which is executed at the PIM cores. Due to the fact that we are unable to map every benchmark instruction at the PIM pipelines, we employ the near data execution paradigm on large loops with a high iteration count. To this end, we dispatch the loops that compose the benchmark binaries to the PIM cores in a serialized fashion and we collect the results back to the BOOM core pipeline after the loop execution completes.

4.5.2 Normalized speedup

Figure 4.6 depicts the normalized speedup of each kernel execution on the PIM core implementations over the RISC-V core execution only. In order to properly measure each kernel execution time, we also include the PIM pre-processing pipeline time cost for each PIM implementation. We observe that the kernel speedup factors depend on the characteristics of each workload and on the corresponding PIM core implementation parameters. In this sense,

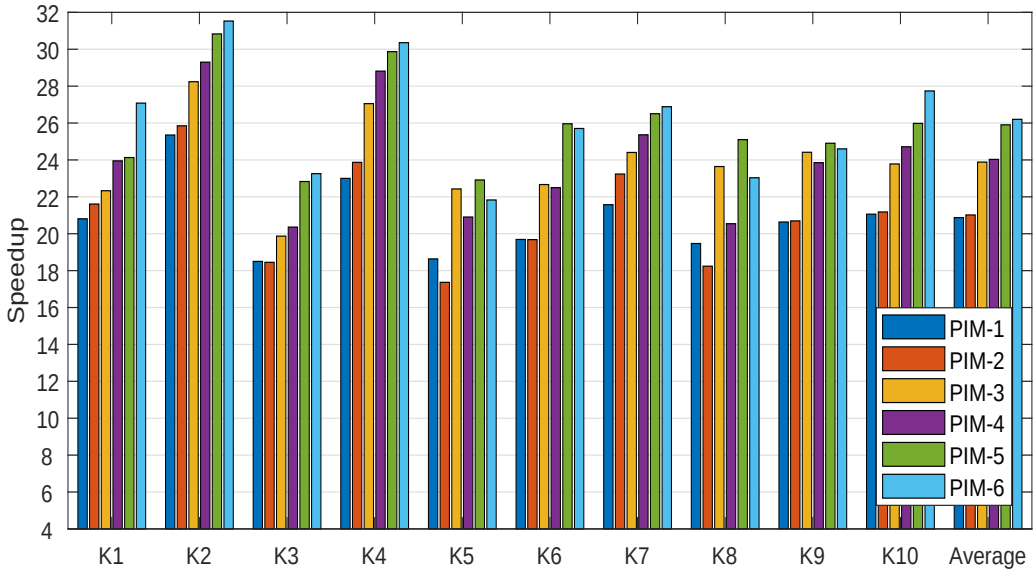

FIGURE 4.6: Normalized speedup of each PIM core implementation over the baseline RISC-V pipeline.

the NDP execution seems to benefit more the kernels with large memory overheads such as K2,K4,K7 and K10. This behavior is expected due to the fact that PIM execution does not require intensive host processor-DRAM communication and thus, the processor bus bottleneck is eliminated. For memory intensive workloads, NDP execution achieves 21x - 30x speedup, depending on the requirements of each kernel. Also the kernel performance is affected by its instruction types, as the adaptive clocking mechanism identifies the instruction classes and scales the clock frequency correspondingly. To this end, PIM designs with greater deviation in path slack such as PIM-1, PIM-3 and PIM-5 match the speedup values of the pipelined designs as the CCU compensates for such variations in timing paths. Further, PIM cores with higher voltage supply such as PIM-5 and PIM-6 outperform the rest of the implementations as they provide more opportunities for aggressive clock adaptation. The average speedup factors for each PIM implementation range from 20.8x to 26x and demonstrate the efficiency of the PIM designs in terms of performance.

Figure 4.7 depicts the impact of each design technique to the overall speedup of each PIM implementation. To this end, we execute each kernel under different PIM core configurations and then, we average the benchmark speedup factors of each PIM implementation. We observe that the near data execution process contributes 57% to the overall speedup while the adaptive clock scaling technique, which is enabled by the IPE-STA technique, contributes 30% on average. Further, the PIM pre-processing pipeline manages to improve the workload execution time by 13%. We should note though, that the purpose of PIM pre-processing is not to speedup execution time; instead it contributes to lowering the complexity and power consumption by eliminating the need of complex hardware structures on the PIM cores. We deduce that designs with relaxed timing constrains such as PIM-1, PIM-3 and PIM-5 benefit more from the IPE-STA methodology when compared with designs that display tighter timing requirements such as PIM-2, PIM-4 and PIM-6. This behavior is expected as the IPE-STA exploits the timing differences between individual PIM operations. As a result, the more relaxed the system timing, the better performance increase is observed.

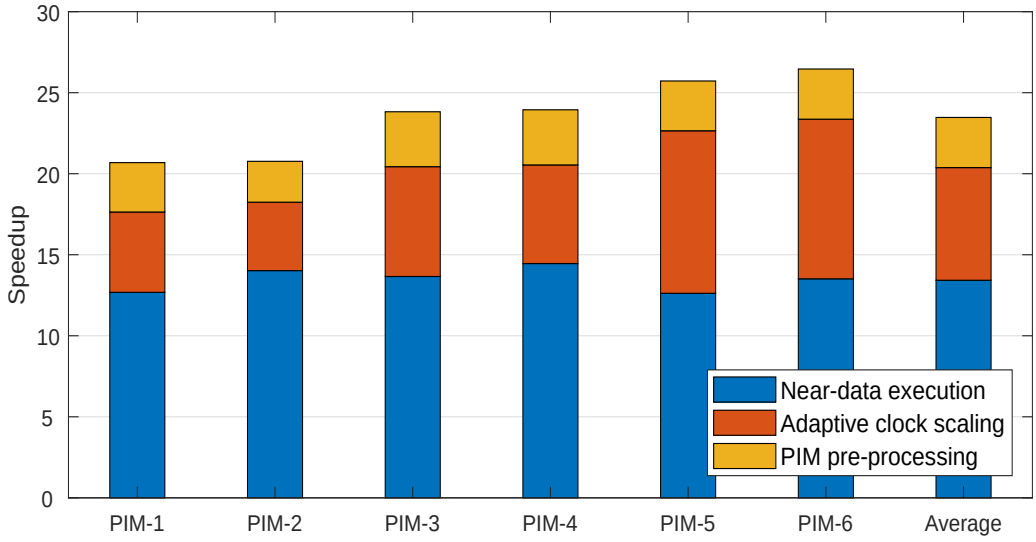

FIGURE 4.7: Impact of different design techniques on each PIM core speedup factor

4.5.3 Normalized energy reduction

Figure 4.8 depicts the normalized energy reduction of each kernel execution on the PIM core implementations over the RISC-V BOOM core execution only. We observe that PIM core execution manages to reduce the energy consumption of the system for 9x to 15.3x on average. Such a high reduction in energy costs derives from both NDP and IPE-STA application on PIM-cores. As the kernel execution is conducted on the HMC logical layer, the large energy overhead of data transmission between the host system and the DRAM is eliminated. Also, the energy consumption is affected by the power consumption of the PIM cores and by the benchmark execution time. To this end, the IPE-STA manages to overclock the PIM cores and thus, reducing the execution time of each benchmark while also increasing the power consumption. As the benchmark execution is accelerated by an average factor of 23x and the power consumption is increased by a factor of 3x-8x, the benefits of accelerated execution outweigh the power costs and thus, the energy consumption is significantly reduced despite the increase in power. Moreover the energy reduction of each core implementation differs due to the supply voltage variations, clock scaling opportunities and architectural parameters. More specifically, PIM cores with lower supply voltage such as PIM-1 and PIM-2 tend to reduce the energy consumption by a larger factor, when compared to PIM cores with higher supply voltage such as PIM-5 and PIM-6. Further, designs with pipelined execution units such as PIM-2, PIM-3 and PIM-6 tend to consume more energy than the NPE PIM cores due to the power overhead of the additional pipeline registers. Finally, high frequency operating PIM cores such as PIM-5 and PIM-6 depict larger speedup factors and thus, reduce the amount of time required for kernel execution, but they also consume significantly more energy when compared to PIM cores with relatively slower clocks, such as PIM-1, PIM-2, PIM-3 and PIM-4.

4.5.4 Energy efficiency

Figure 4.9 depicts the energy efficiency in terms of Gops/Watt for the PIM core and RISC-V implementations. We observe that the PIM implementations achieve an average 202 Gops/W

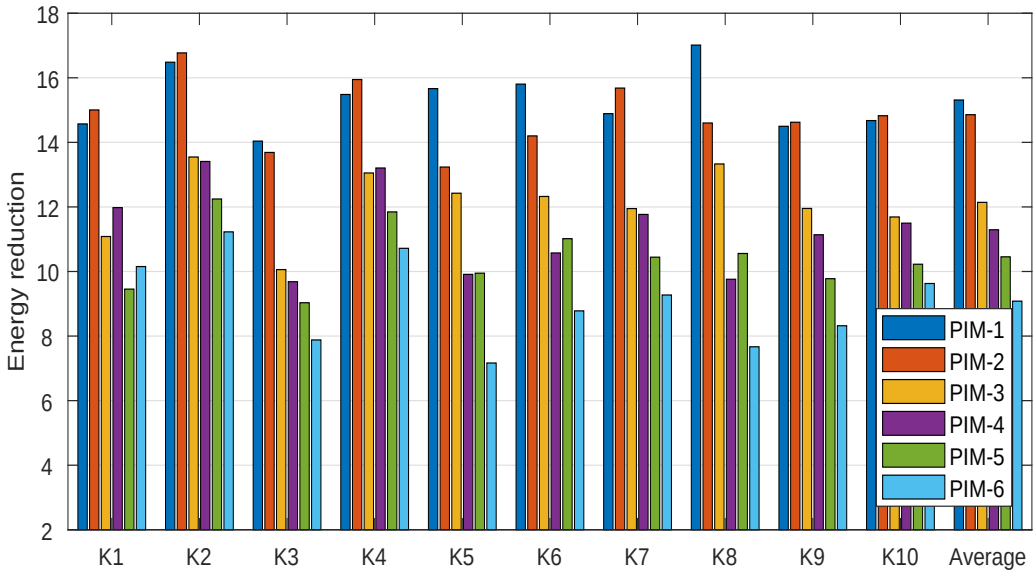

FIGURE 4.8: Normalized energy reduction of each PIM core implementation over the baseline RISC-V pipeline.

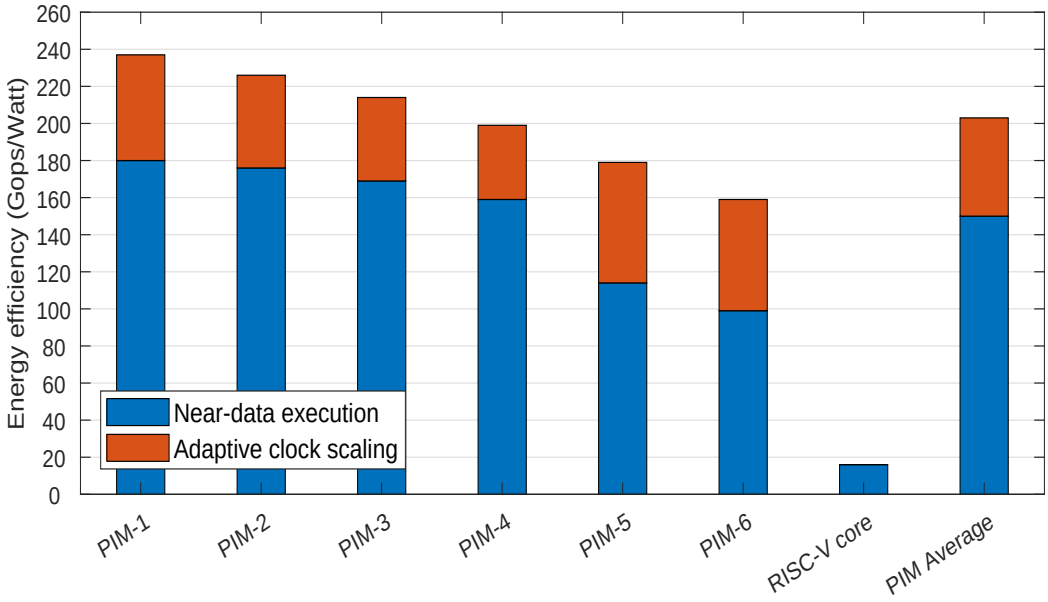

FIGURE 4.9: Energy efficiency of the PIM core and RISC-V core pipeline implementations

and are 10x-14x times more energy efficient, compared to the RISC-V core pipeline. Also the energy efficiency of PIM designs drops as the supply voltage increases, but even high voltage implementations such as PIM-6 manage to maintain a 160 Gops/W energy efficiency, which we consider very high. Further, PE PIM designs tend to be less energy efficient when compared to the corresponding NPE implementations as the extra pipeline registers tighten the circuit timing and thus, the IPE-STA technique has less opportunities for aggressively increasing their performance. The average contribution of the IPE-STA technique to the overall energy efficiency of the NDP designs is measured at 33% and varies according to the timing opportunities for each PIM implementation. As a result, designs with larger timing margins tend to benefit more from the adaptive clock scaling technique, while designs with tighter timing depict lower but noticeable energy efficiency improvement.

4.5.5 Area efficiency

Figure 4.10 depicts the area efficiency in terms of Gops/mm^2 for the PIM core and RISC-V implementations correspondingly. The PIM cores manage to outperform the RISC-V BOOM pipeline by 24.5x times, achieving an average area efficiency of 54 Gops/mm^2. Such an area efficiency escalates as the voltage supply increases, due to the fact that higher supply voltages do not pose additional area requirements; instead they only contribute to the overall execution speedup. As a result, the IPE-STA manages to aggressive over-scale the clock frequency while the design area requirements remain the same. Further, NPE PIM cores such as PIM-1, PIM-3 and PIM-5 are more area efficient compared to the corresponding PE PIM implementations. Such behavior is attributed to the fact that while the PE implementations require more area to operate, their performance increase due to the pipelined execution does not compensate for such area requirements. Further, the IPE-STA methodology contributes 44% in the total area efficiency of the NDP designs, while the rest 56% comes from the NDP execution paradigm. We deduce that both adaptive clock scaling mechanism and the NDP are critical to the overall performance of the PIM cores.

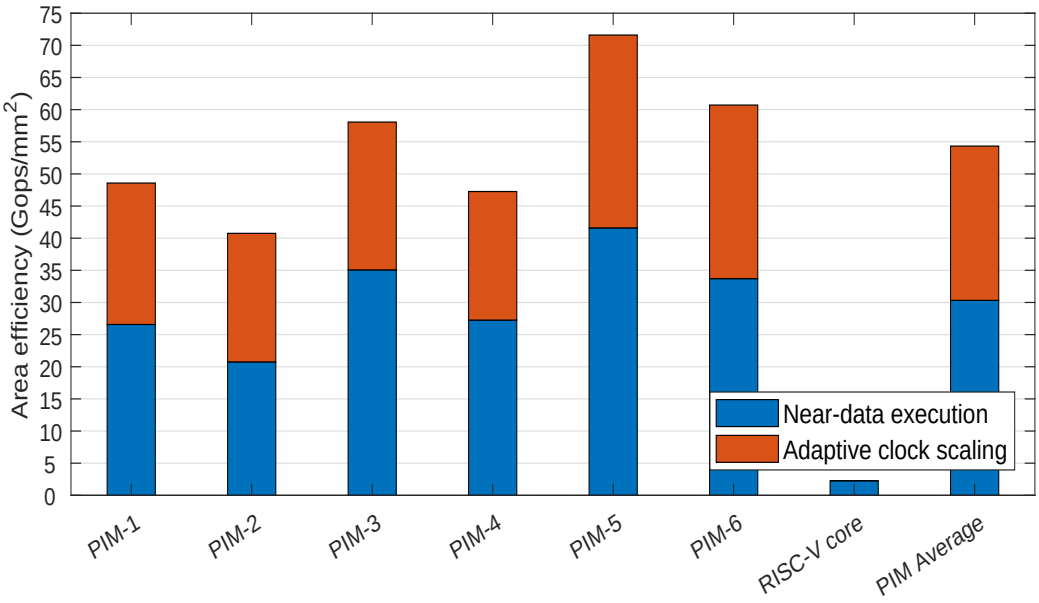

FIGURE 4.10: Area efficiency of the PIM core and RISC-V core pipeline implementations

Chapter 5

Conclusions and Future Directions

5.1 Conclusions

In this dissertation we explored the NDP paradigm under the premise of general purpose code execution. To this end, we employed the NDP model for HPC and for low-end, low-power computer architectures. Our work considers the architectural characteristics of each domain separately and we develop an approach tailored for the requirements of each case. In this sense, we propose and evaluate the following design methodologies:

- **An NDP methodology for general purpose applications within the HPC domain**. Therefore, we design a CGRA from the ground up to accelerate instruction loop execution by leveraging the dataflow paradigm, while addressing the data dependencies that hamper the achievable throughput. Our design utilizes a number of PEs and CEs that are deployed on the logic layer of an HMC DRAM die. In order to minimize pipeline stalls, each PE is assigned a fifo queue capable of storing its temporary outputs before they are forwarded to the next PE. The CGRA instruction issue process is conducted after applying the LP optimization, to reduce hazards from intra-loop data dependencies. This process issues a number of mutually exclusive instructions on a single PE-CE and thus, each processing element is charged with executing one instruction at any given clock cycle. As a result, the CEs-PEs iteratively execute a single instruction with each execution belonging to a subsequent loop iteration. The NDP methodology is evaluated on post-layout simulations using benchmark kernels that cover a wide range of applications. Results indicate a 42.4x speedup improvement and 22.4x reduction in energy consumption normalized to the host processor execution. Further comparison with related NDP methodologies highlights the effectiveness of the proposed methodology, as it can be classified among the state-of-the-art NDP techniques in terms of both speedup and energy reduction.

- **The IPE-STA methodology**, which shifts the focus from a general critical path analysis to the less constrained analysis of paths that are actually followed by individual instructions. To this end, we design and implement a circuit capable of identifying the timing requirements of any incoming instruction and selecting the appropriate pipeline clock out of a number of deployed PLLs. Thus we are able to scale up the clock frequency beyond its worst-case operational limit. We evaluate our methodology using a RiscV processor architecture which presents differences in pipeline stage timing. Results demonstrate an average performance increase of 1.62x, as well as a 3 to 4-fold improvement in performance-to-power ratio, compared to the baseline processor.

- **An BTWC-NDP co-design approach** for low-power, low-end pipelines through the IPE-STA methodology. The proposed concept includes an NDP system architecture approach combined with a novel timing analysis technique which is inspired by the BTWC paradigm, called IPE-STA. We employ the HMC DRAM which is a 3D stacked

DRAM that enables NDP by facilitating logic cells in its lower DRAM layer. To this end, we design and implement six different PIM core pipelines based on the RISC-V BOOM processor by taking into account the HMC requirements while respecting the low power design constrains. In the sequel, we proceed to analyzing each PIM core timing requirements by employing the IPE-STA technique. IPE-STA allows us to obtain timing information about each instruction's worst case delay, instead of analyzing the worst case delay of the circuit critical path. We use such information to design and implement a clock control unit capable of identifying the timing requirements of any incoming PIM instruction and select the appropriate clock frequency so that no timing violations occur. This adaptive clock scaling mechanism is implemented on the PIM core designs and is supported by a PIM pre-processing pipeline which is deployed on the host system architecture. We evaluate our methodology in post-layout simulations of the implemented PIM cores by utilizing general purpose workloads from a great variety of application fields. Results indicate a significant speedup rate of 23x on average, and a average energy reduction factor of 12x over the RISC-V BOOM execution. The proposed PIM core implementations also demonstrate a 12x better energy and 24.5x better area efficiency compared to the RISC-V host system.

5.2 Future Directions

In the future, we plan to extend the research presented in this dissertation towards the following directions:

- **Develop a BTWC methodology for HPC applications**. We plan on further developing the proposed BTWC methodology in order to include the HPC paradigm. To this end, we will extend the IPE-STA methodology to be fully applicable on more tightened-timing designs while also lowering the amount of clocks required for the IPE-STA to work.

- **Further improve the proposed CGRA implementation for general purpose loop execution.** We plan on expanding our design to include several micro-architecture configurations that will contribute on a significant performance increase. Moreover, we will develop a more sophisticated instruction scheduling methodology that will efficiently utilize the CGRA topology. We will also conduct a detailed design space exploration on cache utilization on the logic layer of the HMC so that to reduce the total amount of DRAM access and scale down the energy consumption of the system.

- **Consider exploiting the bank-level parallelism of the HMC** memory, instead of relying in serialized memory access. The HMC can greatly benefit from the bank-level parallelism instead of the regular spatial data-level parallelism of the standard memory modules. As a result, the design that is implemented on the logic layer of the HMC will take into account potential data movements between the HMC banks in order to maximize the amount of parallelized HMC access operations.

- **Design and implement an IPE-STA extention for OoO cores.** In this work we focus our efforts on creating a timing analysis approach for low-end, low-power computer architectures. Although such an approach contributes to great performance improvements on the pipelines that is applied to, a question arises on how the IPE-STA can be used for high performance OoO cores. To this end, we plan on expanding the current state of the IPE-STA to integrate the HPC premise and to make it interoperable with more complex designs.

Appendix A

Relevant Publications

Conference publications:

- A. Tziouvaras, G. Dimitriou, M. Dossis and G. Stamoulis, "Instruction-Based Timing Analysis in Pipelined Processors," 2019 4th South-East Europe Design Automation, Computer Engineering, Computer Networks and Social Media Conference (SEEDA-CECNSM), Piraeus, Greece, 2019, pp. 1-6.
- A. Tziouvaras, G. Dimitriou, M. Dossis and G. Stamoulis, "Instruction-Flow-Based Timing Analysis in Pipelined Processors," 2019 Panhellenic Conference on Electronics & Telecommunications (PACET), Volos, Greece, 2019, pp. 1-6.
- A. Tziouvaras, G. Dimitriou, M. Dossis and G. Stamoulis, "Adaptive Operation-Based ALU and FPU Clocking," 2020 9th International Conference on Modern Circuits and Systems Technologies (MOCAST), Bremen, Germany, 2020, pp. 1-4.
- A. Tziouvaras, G. Dimitriou,F. Foukalas and G. Stamoulis, "Low power general purpose loop acceleration for NDP applications," in PCI 2020

Journal publications:

- A. Tziouvaras, G. Dimitriou, M. Dossis, G. Stamoulis "Frequency Scaling for High Performance of Low-End Pipelined Processors", Advances in Science, Technology and Engineering Systems Journal, vol. 6, no. 2, pp. 763-775 (2021).
- A. Tziouvaras, G. Dimitriou, F. Foukalas and G, Stamoulis, "A near-data processing architecture for accelerating loop execution in memory intensive workloads," in ACM Transactions on Computer Systems (TOCS) (Under review)
- A. Tziouvaras, G. Dimitriou and G, Stamoulis, "Low power near-data code execution leveraging opcode-based timing analysis," in ACM Transactions on Computer Systems (TOCS) (Under review)

Bibliography

[1] R. H. Dennard, F. H. Gaensslen, V. L. Rideout, E. Bassous, and A. R. LeBlanc. Design of ion-implanted mosfet's with very small physical dimensions. IEEE Journal of Solid-State Circuits, 9, October 1974.

[2] H. Esmaeilzadeh, E. Blem, R. S. Amant, K. Sankaralingam and D. Burger, "Dark silicon and the end of multicore scaling," 2011 38th Annual International Symposium on Computer Architecture (ISCA), San Jose, CA, USA, 2011, pp. 365-376.

[3] A.Marathe, Y. Zhang, G. Blanks, N. Kumbhare, G. Abdulla, and B. Rountree. "An empirical survey of performance and energy efficiency variation on Intel processors," in Proceedings of the 5th International Workshop on Energy Efficient Supercomputing (E2SC'17). Association for Computing Machinery, New York, NY, USA, Article 9, 1–8. Nov. 2017.

[4] B.Acun, P. Miller, and L.t V. Kale. "Variation Among Processors Under Turbo Boost in HPC Systems," in Proceedings of the International Conference on Supercomputing (ICS '16). Association for Computing Machinery, New York, NY, USA, Article 6, 1–12, June 2016.

[5] W. Shin et al., "DRAM-Latency Optimization Inspired by Relationship between Row-Access Time and Refresh Timing," in IEEE Transactions on Computers, vol. 65, no. 10, pp. 3027-3040,Oct. 2016.

[6] S. Borkar, "Role of Interconnects in the Future of Computing," in Journal of Lightwave Technology, vol. 31, no. 24, pp. 3927-3933, Dec.15, 2013.

[7] R. Balasubramonian et al., "Near-Data Processing: Insights from a MICRO-46 Workshop," in IEEE Micro, vol. 34, no. 4, pp. 36-42, July-Aug. 2014.

[8] H. Asghari-Moghaddam, Y. H. Son, J. H. Ahn and N. S. Kim, "Chameleon: Versatile and practical near-DRAM acceleration architecture for large memory systems," 2016 49th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO), Taipei, 2016, pp. 1-13.

[9] S. Patrick , B. Rainer and B. Mladen, "Data-Centric Computing Frontiers: A Survey On Processing-In-Memory". in Proceedings of the Second International Symposium on Memory Systems, pp 295-308, 2016.

[10] Marko Scrbak, Mahzabeen Islam, Krishna M. Kavi, Mike Ignatowski, and Nuwan Jayasena. 2017. Exploring the Processing-in-Memory design space. J. Syst. Archit. 75, C (April 2017), 59–67.

[11] J. Ahn, S. Hong, S. Yoo, O. Mutlu and K. Choi, "A scalable processing-in-memory accelerator for parallel graph processing," 2015 ACM/IEEE 42nd Annual International Symposium on Computer Architecture (ISCA), Portland, OR, 2015, pp. 105-117.

[12] V. Seshadri, et al., "Fast bulk bitwise AND and OR in DRAM," IEEE Comput. Archit. Lett., vol. 14, no. 2, pp. 127–131, Jul–Dec. 2015.

[13] X. Yang,Y. Hou and H. He, "A Processing-in-Memory Architecture Programming Paradigm for Wireless Internet-of-Things Applications.",in Sensors, vol. 19, Jan. 2019.

[14] H. Zhang, G. Chen, B. C. Ooi, K. Tan and M. Zhang, "In-Memory Big Data Management and Processing: A Survey," in IEEE Transactions on Knowledge and Data Engineering, vol. 27, no. 7, pp. 1920-1948, 1 July 2015.

[15] J. Huang et al., "Active-Routing: Compute on the Way for Near-Data Processing," 2019 IEEE International Symposium on High Performance Computer Architecture (HPCA), Washington, DC, USA, 2019, pp. 674-686.

[16] A. Farmahini-Farahani, J. H. Ahn, K. Morrow and N. S. Kim, "NDA: Near-DRAM acceleration architecture leveraging commodity DRAM devices and standard memory modules," 2015 IEEE 21st International Symposium on High Performance Computer Architecture (HPCA), Burlingame, CA, 2015, pp. 283-295.

[17] M. Gao and C. Kozyrakis, “HRL: Efficient and Flexible Reconfigurable Logic for Near-Data Processing,” in HPCA, 2016.

[18] H. Asghari-Moghaddam, A. Farmahini-Farahani, K. Morrow, J. H. Ahn, and N.S. Kim, “Near-DRAM acceleration with single-ISA heterogeneous processing in standard memory modules,” IEEE Micro, vol. 36, 2016.

[19] F. Schuiki, M. Schaffner, F. K. Gürkaynak and L. Benini, "A Scalable Near-Memory Architecture for Training Deep Neural Networks on Large In-Memory Datasets," in IEEE Transactions on Computers, vol. 68, no. 4, pp. 484-497, 1 April 2019.

[20] S. Gupta, M. Imani, H. Kaur and T. S. Rosing, "NNPIM: A Processing In-Memory Architecture for Neural Network Acceleration," in IEEE Transactions on Computers, vol. 68, no. 9, pp. 1325-1337, 1 Sept. 2019.

[21] M. Imani, S. Gupta, S. Sharma and T. S. Rosing, "NVQuery: Efficient Query Processing in Nonvolatile Memory," in IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems, vol. 38, no. 4, pp. 628-639, April 2019.

[22] B. Gu et al., "Biscuit: A Framework for Near-Data Processing of Big Data Workloads," 2016 ACM/IEEE 43rd Annual International Symposium on Computer Architecture (ISCA), Seoul, 2016, pp. 153-165.

[23] S. Lloyd and M. Gokhale, “Near memory key/value lookup acceleration,” in ACM International Symposium on Memory Systems. ACM, 2017.

[24] A. Boroumand et al., "LazyPIM: An Efficient Cache Coherence Mechanism for Processing-in-Memory," in IEEE Computer Architecture Letters, vol. 16, no. 1, pp. 46-50, 1 Jan.-June 2017.

[25] G. Singh et al., "A Review of Near-Memory Computing Architectures: Opportunities and Challenges," 2018 21st Euromicro Conference on Digital System Design (DSD), Prague, 2018, pp. 608-617.

[26] K. Hsieh, et al., “Transparent Offloading and Mapping (TOM): Enabling programmer-transparent near-data processing in GPU systems,” in Proc. Int. Symp. Comput. Archit., 2016.

[27] J. Ahn, S. Yoo, O. Mutlu and K. Choi, "PIM-enabled instructions: A low-overhead, locality-aware processing-in-memory architecture," 2015 ACM/IEEE 42nd Annual International Symposium on Computer Architecture (ISCA), Portland, OR, 2015, pp. 336-348.

[28] N. S. Kim and P. Mehra, "[INVITED] Practical Near-Data Processing to Evolve Memory and Storage Devices into Mainstream Heterogeneous Computing Systems," 2019 56th ACM/IEEE Design Automation Conference (DAC), Las Vegas, NV, USA, 2019, pp. 1-4.

[29] B. Akin, F. Franchetti, and J. C. Hoe, “Data Reorganization in Memory Using 3D-Stacked DRAM,” in ISCA, 2015.

[30] F. Devaux, "The true Processing In Memory accelerator," 2019 IEEE Hot Chips 31 Symposium (HCS), Cupertino, CA, USA, 2019, pp. 1-24.

[31] H. Lim and G. Park," Triple Engine Processor (TEP): A Heterogeneous Near-Memory Processor for Diverse Kernel Operations," In ACM Trans. Archit. Code Optim. 14, 4, Article 49 (December 2017), 25 pages.

[32] S. Ghose, K. Hsieh, A. Boroumand, R. Ausavarungnirun and Onur Mutlu, "Enabling the Adoption of Processing-in-Memory: Challenges, Mechanisms, Future Research Directions" in ArXiv, 2018, abs/1802.00320.

[33] Hybrid Memory Cube Consortium (HMCC), “Hybrid memory cube specification 2.1.” 2016. [Online]. Available: http://www.hybridmemorycube. org/files/SiteDownloads/HMC-30G-VSR_HMCC_Specification_Rev2.1_20151105.pdf.

[34] M. Wijtvliet, L. Waeijen and H. Corporaal, "Coarse grained reconfigurable architectures in the past 25 years: Overview and classification," 2016 International Conference on Embedded Computer Systems: Architectures, Modeling and Simulation (SAMOS), Agios Konstantinos, 2016, pp. 235-244.

[35] Z. E. Rákossy, F. Merchant, A. Acosta-Aponte, S. K. Nandy and A. Chattopadhyay, "Efficient and scalable CGRA-based implementation of Column-wise Givens Rotation," 2014 IEEE 25th International Conference on Application-Specific Systems, Architectures and Processors, Zurich, 2014, pp. 188-189.

[36] M. Tanomoto, S. Takamaeda-Yamazaki, J. Yao and Y. Nakashima, "A CGRA-Based Approach for Accelerating Convolutional Neural Networks," 2015 IEEE 9th International Symposium on Embedded Multicore/Many-core Systems-on-Chip, Turin, 2015, pp. 73-80.

[37] X. Chen, A. Minwegen, S. B. Hussain, A. Chattopadhyay, G. Ascheid and R. Leupers, "Flexible, Efficient Multimode MIMO Detection by Using Reconfigurable ASIP," in IEEE Transactions on Very Large Scale Integration (VLSI) Systems, vol. 23, no. 10, pp. 2173-2186, Oct. 2015.

[38] S. Yin, D. Liu, L. Sun, L. Liu and S. Wei, "DFGNet: Mapping dataflow graph onto CGRA by a deep learning approach," 2017 IEEE International Symposium on Circuits and Systems (ISCAS), Baltimore, MD, 2017, pp. 1-4.

[39] Y. Kim, J. Lee, A. Shrivastava, J. W. Yoon, D. Cho and Y. Paek, "High Throughput Data Mapping for Coarse-Grained Reconfigurable Architectures," in IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems, vol. 30, no. 11, pp. 1599-1609, Nov. 2011.

[40] M. Hamzeh, A. Shrivastava and S. Vrudhula, "REGIMap: Register-aware application mapping on Coarse-Grained Reconfigurable Architectures (CGRAs)," 2013 50th ACM/EDAC/IEEE Design Automation Conference (DAC), Austin, TX, 2013, pp. 1-10.

[41] Y. Kim, R. N. Mahapatra and K. Choi, "Design Space Exploration for Efficient Resource Utilization in Coarse-Grained Reconfigurable Architecture," in IEEE Transactions on Very Large Scale Integration (VLSI) Systems, vol. 18, no. 10, pp. 1471-1482, Oct. 2010.

[42] L. Liu, J. Zhu, Z. Li, Y. Lu, Y. Deng, J. Han, S. Yin, and S. Wei,"A Survey of Coarse-Grained Reconfigurable Architecture and Design: Taxonomy, Challenges, and Applications," In ACM Comput. Surv. 52, 6, Article 118 (January 2020), 39 pages.

[43] S. Yin, D. Liu, L. Liu, S. Wei and Y. Guo, "Joint affine transformation and loop pipelining for mapping nested loop on CGRAs," 2015 Design, Automation Test in Europe Conference Exhibition (DATE), Grenoble, 2015, pp. 115-120.

[44] B. Xu, S. Yin, L. Liu and S. Wei, "Low-power loop pipelining mapping onto CGRA utilizing variable dual VDD," 2014 IEEE 57th International Midwest Symposium on Circuits and Systems (MWSCAS), College Station, TX, 2014, pp. 242-245.

[45] S. Yin, D. Liu, Y. Peng, L. Liu and S. Wei, "Improving Nested Loop Pipelining on Coarse-Grained Reconfigurable Architectures," in IEEE Transactions on Very Large Scale Integration (VLSI) Systems, vol. 24, no. 2, pp. 507-520, Feb. 2016.

[46] M. Dossis and G. Dimitriou, "Resolving Loop Pipelining Issues in the CCC High-level Synthesis E-CAD Framework," 2018 41st International Conference on Telecommunications and Signal Processing (TSP), Athens, 2018, pp. 1-4.

[47] C. Celio, P. Chiu, B. Nikolic, D.A Patterson, K.Asanović, " BOOM v2: an open-source out-of-order RISC-V core", Technical report in EECS Department, University of California, Berkeley, 2017.

[48] J. Schmidt and U. Bruning, "openHMC - a configurable open-source hybrid memory cube controller," 2015 International Conference on ReConFigurable Computing and FPGAs (ReConFig), Mexico City, 2015, pp. 1-6.

[49] M. Martins, J. Maick Matos, R. P. Ribas, A. Reis, G. Schlinker, L. Rech, and J. Michelsen, "Open Cell Library in 15nm FreePDK Technology". In Proceedings of the 2015 Symposium on International Symposium on Physical Design (ISPD '15). ACM, New York, NY, USA, pp. 171–178, 2015.

[50] James Bucek, Klaus-Dieter Lange, and Jóakim v. Kistowski. 2018. SPEC CPU2017: Next-Generation Compute Benchmark. In Companion of the 2018 ACM/SPEC International Conference on Performance Engineering (ICPE '18). Association for Computing Machinery, New York, NY, USA, 41–42.

[51] S. Che et al., "Rodinia: A benchmark suite for heterogeneous computing," 2009 IEEE International Symposium on Workload Characterization (IISWC), Austin, TX, 2009, pp. 44-54.

[52] S. K. Venkata et al., "SD-VBS: The San Diego Vision Benchmark Suite," 2009 IEEE International Symposium on Workload Characterization (IISWC), Austin, TX, 2009, pp. 55-64.

[53] S. C. Woo, M. Ohara, E. Torrie, J. P. Singh and A. Gupta, "The SPLASH-2 programs: characterization and methodological considerations," Proceedings 22nd Annual International Symposium on Computer Architecture, Santa Margherita Ligure, Italy, 1995, pp. 24-36.

[54] J. Dean and S. Ghemawat, "MapReduce: Simplified Data Processing on Large Clusters," in OSDI-6, pp. 10–10, 2004.

[55] J. Dean, G. S. Corrado, R. Monga, K. Chen, M. Devin, Q. V. Le, M. Z. Mao, M. Ranzato, A. Senior, P. Tucker, K. Yang, and A. Y. Ng, "Large Scale Distributed Deep Networks," in NIPS, pp. 1223–1231, 2012.

[56] T. Chilimbi, Y. Suzue, J. Apacible, and K. Kalyanaraman, "Project Adam: Building an Efficient and Scalable Deep Learning Training System," in OSDI-11, pp. 571–582, 2014.

[57] J. E. Gonzalez, Y. Low, H. Gu, D. Bickson, and C. Guestrin, "Power-Graph: Distributed Graph-parallel Computation on Natural Graphs," in OSDI-10, pp. 17–30, 2012.

[58] J.Lin, X.Tian and J Ng, "Mis-speculation-Driven Compiler Framework for Aggressive Loop Automatic Parallelization," 2013 IEEE International Symposium on Parallel and Distributed Processing, Workshops and Phd Forum, Cambridge, MA, 2013, pp. 1159-1168.

[59] T. Austin and V. Bertacco, "Deployment of better than worst-case design: solutions and needs," in Proceedings of the International Conference on Computer Design, Oct. 2005.

[60] R. Ye, F. Yuan, J. Zhang, and Q. Xu, "On the premises and prospects of timing speculation," in Proceedings of the Design, Automation and Test in Europe Conference and Exhibition (DATE), March 2015.

[61] D. Ernst, N. S. Kim, S. Das, S. Pant, R. Rao, T. Pham, C. Ziesler, D. Blaauw, T. Austin, K. Flautner, and T. Mudge, "Razor: a low-power pipeline based on circuit-level timing speculation," in Proceedings of the 36th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO), Dec. 2003.

[62] Y. Zhang, M. Khayatzadeh, K. Yang, M. Saligane, N. Pinckney, M. Alioto, D. Blaauw and D. Sylvester, "iRazor: Current-Based Error Detection and Correction Scheme for PVT Variation in 40-nm ARM Cortex-R4 Processor," in IEEE Journal of Solid-State Circuits, vol. 53, no. 2, pp. 619-631, Feb. 2018.

[63] V. Subramanian, M. Bezdek, N. D. Avirneni, and A. Somani, "Superscalar processor performance enhancement through reliable dynamic clock frequency tuning," in Proceedings of the 37th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN), June 2007.

[64] T. Austin, V. Bertacco, D. Blaauw, and T. Mudge, "Opportunities and challenges for better than worst-case design," in Proceedings of the ASP-DAC, Jan. 2005.

[65] C. -C. Wang, K. -Y. Chao, S. Sampath and P. Suresh, "Anti-PVT-Variation Low-Power Time-to-Digital Converter Design Using 90-nm CMOS Process," in IEEE Transactions on Very Large Scale Integration (VLSI) Systems, vol. 28, no. 9, pp. 2069-2073, Sept. 2020.

[66] B. Poudel and A. Munir, "Design and Evaluation of a PVT Variation-Resistant TRNG Circuit," 2018 IEEE 36th International Conference on Computer Design (ICCD), Orlando, FL, USA, pp. 514-521, 2018.

[67] A. Rahimi, L. Benini, and R. K. Gupta, "Application-adaptive guardbanding to mitigate static and dynamic variability," IEEE Transactions on Computers, vol. 63, pp. 2160–2173, Sept. 2014.

[68] A. Tiwari, S. R. Sarangi, and J. Torrellas, "Recycle: pipeline adaptation to tolerate process variation," in Proceedings of the 34th Annual International Symposium on Computer Architecture (ISCA), June 2007.

[69] S. Sarangi, B. Greskamp, A. Tiwari, and J. Torrellas, "Eval: Utilizing processors with variation-induced timing errors," in Proceedings of the 41st IEEE/ACM International Symposium on Microarchitecture, Nov. 2008.

[70] K. A. Bowman, J. W. Tschanz, S.-L. L. Lu, P. A. Aseron, M. M. Khellah, A. Raychowdhury, B. M. Geuskens, C. Tokunaga, C. B. Wilkerson, T. Karnik, and V. K. De, "A 45 nm resilient microprocessor core for dynamic variation tolerance," IEEE Journal of Solid-State Circuits, vol. 46, pp. 194–208, Jan. 2011.

[71] O. Azizi, A. Mahesri, B. C. Lee, S. J. Patel, and M. Horowitz, “Energy-performance tradeoffs in processor architecture and circuit design: A marginal cost analysis,” in Proceedings of the 37th Annual International Symposium on Computer Architecture (ISCA), June 2010.

[72] B. Greskamp, L. Wan, U. R. Karpuzcu, J. J. Cook, J. Torrellas, D. Chen, and C. Zilles, “Blueshift: Designing processors for timing speculation from the ground up,” in Proceedings of the 15th International Symposium on High-Performance Computer Architecture (HPCA), March 2009.

[73] A. B. Kahng, S. Kang, R. Kumar, and J. Sartori, “Designing a processor from the ground up to allow voltage/reliability tradeoffs,” in Proceedings of the 16th International Symposium on High-Performance Computer Architecture (HPCA), April 2010.

[74] S. Shen et al., "TS Cache: A Fast Cache With Timing-Speculation Mechanism Under Low Supply Voltages," in IEEE Transactions on Very Large Scale Integration (VLSI) Systems, vol. 28, no. 1, pp. 252-262, Jan. 2020.

[75] L. Wan and D. Chen, “Dynatune: circuit-level optimization for timing speculation considering dynamic path behavior,” in Proceedings of the International Conference on Computer-Aided Design (ICCAD), Dec. 2009.

[76] Y. Sun, Y. Liu, X. Wang, H. Xu, and H. Yang, “Design methodology of multistage time-domain logic speculation circuits,” in Proceedings of the IEEE International Symposium of Circuits and Systems (ISCAS), May 2011.

[77] Y. Liu, R. Ye, F. Yuan, R. Kumar, and Q. Xu, “On logic synthesis for timing speculation,” in Proceedings of the IEEE/ACM International Conference on Computer-Aided Design (ICCAD), Nov. 2012.

[78] J. Sartori and R. Kumar, “A case for timing error resilience-aware compilation,” in Proceedings of the 7th Workshop on Silicon Errors in Logic - System Effects (SELSE), March 2011.

[79] G. Hoang, R. B. Findler, and R. Joseph, “Exploring circuit timing-aware languages and compilation,” in Proceedings of the 16th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS), March 2011.

[80] D. M. Tullsen and B. Calder, “Computing along the critical path,” tech. rep., University of California San Diego (UCSD), Oct. 1998.

[81] C. R. Lefurgy, Alan J. Drake, Michael S. Floyd, Malcolm S. Allen-Ware, Bishop Brock, Jose A. Tierno and John B. Carter, "Active management of timing guardband to save energy in POWER7," 2011 44th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO), Porto Alegre, 2011, pp. 1-11.

[82] T. Hashimoto. Yukihito Kawabe, Michiharu Kara, Yasushi Kakimura, Kunihiko Tajiri, Shinichiro Shirota, Ryuichi Nishiyama, Hitoshi Sakurai, Hiroshi Okano, Yasumoto Tomita, Sugio Satoh and Hideo Yamashita, "An adaptive clocking control circuit with 7.5% frequency gain for SPARC processors," 2017 Symposium on VLSI Technology, Kyoto, 2017, pp. C112-C113.

[83] A. Grenat, S. Pant, R. Rachala and S. Naffziger, "Adaptive clocking system for improved power efficiency in a 28nm x86-64 microprocessor," 2014 IEEE International Solid-State Circuits Conference Digest of Technical Papers (ISSCC), San Francisco, CA, 2014, pp. 106-107.

[84] J. Constantin, A. Bonetti, A. Teman, C. Müller, L. Schmid and A. Burg, "DynOR: A 32-bit microprocessor in 28 nm FD-SOI with cycle-by-cycle dynamic clock adjustment," ESSCIRC Conference 2016: 42nd European Solid-State Circuits Conference, Lausanne, Switzerland, pp. 261-264, 2016.

[85] S. Beer, M. Cannizzaro, J. Cortadella, R. Ginosar, and L. Lavagno, “Metastability in better-than-worst-case designs,” in Proceedings of the 20th IEEE International Symposium on Asynchronous Circuits and Systems, May 2014.

[86] K. A. Bowman, J. W. Tschanz, N. S. Kim, J. C. Lee, C. B. Wilkerson, S.-L. L. Lu, T. Karnik, and V. K. De, “Energy-efficient and metastability-immune resilient circuits for dynamic variation tolerance,” IEEE Journal of Solid-State Circuits, vol. 44, pp. 49–63, Jan. 2009.

[87] X. Wang and W. H. Robinson, "Error Estimation and Error Reduction With Input-Vector Profiling for Timing Speculation in Digital Circuits," in IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems, vol. 38, no. 2, pp. 385-389, Feb. 2019.

[88] Z. Li, T. Zhu, Z. Chen, J. Meng, X. Xiang and X. Yan, "Eliminating Timing Errors Through Collaborative Design to Maximize the Throughput," in IEEE Transactions on Very Large Scale Integration (VLSI) Systems, vol. 25, no. 2, pp. 670-682, Feb. 2017.

[89] H. Ahmadi Balef, H. Fatemi, K. Goossens and J. Pineda De Gyvez, "Timing Speculation With Optimal In Situ Monitoring Placement and Within-Cycle Error Prevention," in IEEE Transactions on Very Large Scale Integration (VLSI) Systems, vol. 27, no. 5, pp. 1206-1217, May 2019.

[90] E. Tune, D. Liang, D. M. Tullsen, and B. Calder, “Dynamic prediction of critical path instructions,” in Proceedings of the 7th International Symposium on High-Performance Computer Architecture (HPCA), Jan 2001.

[91] J. Xin and R. Joseph, “Identifying and predicting timing-critical instructions to boost timing speculation,” in Proceedings of the 44th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO), Dec. 2011.

[92] M. de Kruijf, S. Nomura, and K. Sankaralingam, “A unified model for timing speculation: Evaluating the impact of technology scaling, cmos design style, and fault recovery mechanism,” in Proceedings of the 40th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN), June 2010.

[93] H. Y. Cheah, S. A. Fahmy and N. Kapre, "Analysis and optimization of a deeply pipelined FPGA soft processor," 2014 International Conference on Field-Programmable Technology (FPT), Shanghai, 2014, pp. 235-238.

[94] A. Hartstein and T. R. Puzak, "The optimum pipeline depth for a microprocessor," Proceedings 29th Annual International Symposium on Computer Architecture, Anchorage, AK, USA, 2002, pp. 7-13.

[95] V. Agarwal, R. A. Patil and A. B. Patki, "Architectural Considerations for Next Generation IoT Processors," in IEEE Systems Journal, vol. 13, no. 3, pp. 2906-2917, Sept. 2019.

[96] M. Tariq Banday. 2018. A study of current trends in the design of processors for the internet of things. In Proceedings of the 2nd International Conference on Future Networks and Distributed Systems (ICFNDS ’18). Association for Computing Machinery, New York, NY, USA, Article 21, 1–10.

[97] W.Liu, E. Salman, C. Sitik, and B. Taskin. 2015. Clock Skew Scheduling in the Presence of Heavily Gated Clock Networks. In Proceedings of the 25th edition on Great Lakes Symposium on VLSI (GLSVLSI ’15). Association for Computing Machinery, New York, NY, USA, 283–288.

[98] C.Chang, S.Huang, Y.Ho, J. Lin, H. Wang and Y. Lu, "Type-matching clock tree for zero skew clock gating," 2008 45th ACM/IEEE Design Automation Conference, Anaheim, CA, 2008, pp. 714-719.

[99] J. Tschanz et al., "Adaptive Frequency and Biasing Techniques for Tolerance to Dynamic Temperature-Voltage Variations and Aging," 2007 IEEE International Solid-State Circuits Conference. Digest of Technical Papers, San Francisco, CA, 2007, pp. 292-604.

[100] B. A. Floyd, "Sub-Integer Frequency Synthesis Using Phase-Rotating Frequency Dividers," in IEEE Transactions on Circuits and Systems I: Regular Papers, vol. 55, no. 7, pp. 1823-1833, Aug. 2008.

[101] R. Kumar and V. Kursun, “Reversed temperature-dependent propagation delay characteristics in nanometer CMOS circuits,” IEEE Trans. Circuits Syst., vol. 53, no. 10, pp. 1078–1082, Oct. 2006.

[102] J. Schmidt, H. Fröning, and U. Brüning. Exploring Time and Energy for Complex Accesses to a Hybrid Memory Cube. In Proceedings of the Second International Symposium on Memory Systems (MEMSYS ’16). Association for Computing Machinery, New York, NY, USA, 142–150, 2016.

[103] R. Hadidi et al., "Performance Implications of NoCs on 3D-Stacked Memories: Insights from the Hybrid Memory Cube," 2018 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS), Belfast, UK, 2018, pp. 99-108.

[104] M. Martins, J. M. Matos, R. P. Ribas, A. Reis, G. Schlinker, L. Rech, and J. Michelsen. 2015. Open Cell Library in 15nm FreePDK Technology. In Proceedings of the 2015 Symposium on International Symposium on Physical Design (ISPD ’15). Association for Computing Machinery, New York, NY, USA, 171–178.

[105] K. Asanović, R. Avižienis, J. Bachrach, S. Beamer, D. Biancolin, C.Celio, H. Cook, P. Dabbelt, J. Hauser, A. Izraelevitz, S. Karandikar, B. Keller, Do. Kim, J. Koenig, Y. Lee, E. Love, M. Maas, A. Magyar, H. Mao, M. Moreto, A. Ou, D. Patterson, B. Richards, C. Schmidt, S. Twigg, H. Vo, and A. Waterman, "The Rocket Chip Generator," Technical Report UCB/EECS-2016-17, EECS Department, University of California, Berkeley, April 2016.

[106] C. Szegedy, V. Vanhoucke, S. Ioffe, J. Shlens and Z. Wojna, "Rethinking the Inception Architecture for Computer Vision," 2016 IEEE Conference on Computer Vision and Pattern Recognition (CVPR), Las Vegas, NV, 2016, pp. 2818-2826.

[107] A. Shukla, S. Chaturvedi and Y. Simmhan, "RIoTBench: A Real-time IoT Benchmark for Distributed Stream Processing Platforms", In ArXiv :https://arxiv.org/abs/1701.08530, Jan. 2017.

[108] S. Huang, J. Huang, J. Dai, T. Xie and B. Huang, "The HiBench benchmark suite: Characterization of the MapReduce-based data analysis," 2010 IEEE 26th International Conference on Data Engineering Workshops (ICDEW 2010), Long Beach, CA, 2010, pp. 41-51.

[109] L. Wang et al., "BigDataBench: A big data benchmark suite from internet services," 2014 IEEE 20th International Symposium on High Performance Computer Architecture (HPCA), Orlando, FL, 2014, pp. 488-499.

[110] I. Kwon, S. Kim, D. Fick, M. Kim, Y. Chen and D. Sylvester, "Razor-Lite: A Light-Weight Register for Error Detection by Observing Virtual Supply Rails," in IEEE Journal of Solid-State Circuits, vol. 49, no. 9, pp. 2054-2066, Sept. 2014.

[111] M. Fojtik et al., "Bubble Razor: An architecture-independent approach to timing-error detection and correction," 2012 IEEE International Solid-State Circuits Conference, San Francisco, CA, USA, 2012, pp. 488-490.