Energy Optimization for Many-Core Platforms under Process, Voltage and Temperature Variations

by

SOHAIB MAJZOUB

M.A.Sc, Swiss Federal Institute of Technology, 2004
M.Eng., American University of Beirut, 2003
B.Eng., Beirut Arab University, 2000

A THESIS SUBMITTED IN PARTIAL FULFILMENT OF THE REQUIREMENTS OF THE DEGREE OF

DOCTOR OF PHILOSOPHY

in

The Faculty of Graduate Studies

(Electrical and Computer Engineering)

THE UNIVERSITY OF BRITISH COLUMBIA

(VANCOUVER)

August 2010

© Sohaib Majzoub, 2010
Abstract

Many-core architectures are the most recent shift in multi-processor design. This processor design paradigm replaces large monolithic processing units by thousands of simple processing elements on a single chip. With such a large number of processing units, it promises significant throughput advantage over traditional multi-core platforms. Furthermore, it enables better localized control of power consumption and thermal gradients. This is achieved by selective reduction of the core’s supply voltage, or by switching some of the cores off to reduce power consumption and heat dissipation.

This dissertation proposes an energy optimization flow to implement applications on many-core architectures taking into account the impact of Process Voltage and Temperature (PVT) variations. The flow utilizes multi-supply voltage techniques, namely voltage island design, to reduce power consumption in the implementation. We propose a novel approach to voltage island formation, called Voltage Island Clouds, that reduces the impact of on-chip or intra-die PVT variations. The islands are created by balancing their shape constraints imposed by intra- and inter-island communication with the desire to limit the spatial extent of each island to minimize PVT impact. We propose an algorithm to build islands for Static Voltage Scaling (SVS) and Multiple Voltage Scaling (MVS) design approaches.

The optimization initially allows for a large number of islands, each with its unique voltage level. Next, the number of the islands is reduced to a small practical number, e.g., four voltages, using our new efficient voltage selection approach, called the Removal Cost Method (RCM), that provides near optimal results with more than a 10X speedup compared to the best-known previous methods. Finally, we present an evaluation platform considering pre- and post-fabrication PVT scenarios where multiple applications with hundreds to thousands of tasks are mapped onto many-core platforms with hundreds to thousands of cores to evaluate the proposed techniques. Results show that the geometric average energy savings for 33 test cases using the proposed methods is between 8-30% better than previous methods.
# Table of Contents

**Abstract** .................................................................................................................................................. II

**Table of Contents** .................................................................................................................................. III

**List of Figures** ......................................................................................................................................... VIII

**List of Tables** ......................................................................................................................................... XI

**List of Abbreviations** ............................................................................................................................ XII

**Acknowledgments** ................................................................................................................................. XIII

**Chapter 1: Introduction** .......................................................................................................................... 1

1.1. Motivation ............................................................................................................................................. 1

1.2. Research Challenges ............................................................................................................................ 3

1.3. Research Goals ..................................................................................................................................... 8

1.4. Thesis Organization .............................................................................................................................. 9

**Chapter 2: Background** .......................................................................................................................... 10

2.1. Chapter Overview ................................................................................................................................ 10

2.2. Many-Core Architectures .................................................................................................................. 11

2.3. The Impact of Variations on Many-Core Architectures ........................................................................ 15

2.3.1. Process, Voltage, and Temperature (PVT) Variations ................................................................. 15

2.3.2. Prior Work on Mitigating PVT Impact on Multi-Core Designs ...................................................... 24
CHAPTER 3: ENERGY OPTIMIZATION FLOW .......................................................... 39

3.1. Chapter Overview .................................................................................. 39

3.2. Platform Characteristics ....................................................................... 40

3.3. Application Representation .................................................................. 41

3.4. Overview of the Optimization Flow ...................................................... 42

3.5. Initial Voltage Scaling .......................................................................... 44

3.6. Initial Voltage Island Formation .............................................................. 47

3.7. Task Scheduling .................................................................................... 50

3.8. Voltage Scaling and Selection ................................................................. 55

3.9. Voltage Island Cloud Formation ............................................................ 55

3.10. Routing Optimization using Genetic Algorithms .............................. 57
CHAPTER 5: VOLTAGE ISLAND FORMATION .............................................................. 92

5.1. Chapter Overview ....................................................................................... 92

5.2. Voltage Island Design Methodology ........................................................ 93

5.3. Motivation for a New Voltage Island Formation Approach ..................... 94

5.4. Problem Formulation .................................................................................. 97

5.5. Voltage Island Seeding .............................................................................. 99

5.6. Communication and PVT Weighting Model .............................................. 101

5.7. Seed Weight Calculation .......................................................................... 102

5.7.1. SW Calculation for Static Voltage Scaling (SVS) ................................ 102

5.7.2. SW Calculation for Multiple Voltage Scaling (MVS) ....................... 103

5.8. Application Partitioning .......................................................................... 105

5.9. Voltage Island Cloud Placement Algorithm .......................................... 108

5.10. Experimental Platform .......................................................................... 112

5.11. Results and Analysis ............................................................................. 115

5.11.1. Voltage Island Shape ........................................................................ 115

5.11.2. Core-Speed Correlation Impact ....................................................... 119

5.11.3. Increased Process Variation Level ..................................................... 120

5.11.4. Final Energy Savings after Voltage Selection ................................ 122

5.12. Chapter Summary .................................................................................. 125

CHAPTER 6: CONCLUSIONS AND FUTURE WORK .............................................. 126
6.1. Research Context ................................................................. 126

6.2. Research Contributions .......................................................... 127

6.3. Research Observations and Limitations...................................... 129

6.4. Future Work ........................................................................... 130

   6.4.1. Near-Term Future Work ................................................... 130

   6.4.2. Long-Term Future Work ................................................... 131

REFERENCES .................................................................................. 133
## List of Figures

Figure 2.1: Many-core with 64 cores and its Network-on-Chip [2] ......................................................... 12
Figure 2.2: 167-processor many-core [44] .................................................................................................. 13
Figure 2.3: Example of \( V_i \) systematic variation .................................................................................. 19
Figure 2.4: Probability distribution of normalized frequencies in a chip with respect to \( V_i \)'s \( \sigma_{\text{total}}/\mu \) ratio, nominal \( V_i = 400 \text{mV}, T = 25^\circ \text{C} \) .......................................................................................................................................... 20
Figure 2.5: A 30x30 many-core with the normalized core frequency due to within-die process variation... 21
Figure 2.6: Example of resistor network used to emulate the power grid .................................................. 23
Figure 2.7: Core modeled as current source ............................................................................................... 24
Figure 2.8: Voltage interpolation to compensate process variation effects ............................................... 25
Figure 2.9: Multi-\( V_{dd} \) Design Flow ........................................................................................................... 30
Figure 2.10: Voltage scaling to eliminate idle time ..................................................................................... 31
Figure 2.11: Power Management Unit to handle DVFS ............................................................................... 34
Figure 2.12: Different power modes in MVS design .................................................................................. 36
Figure 3.1: A core, its communication switch/router and buses ................................................................. 40
Figure 3.2: A Directed Acyclic Graph (DAG) ............................................................................................ 41
Figure 3.3: Improved Energy Optimization Flow ....................................................................................... 43
Figure 3.4: Voltage scaling algorithm ....................................................................................................... 46
Figure 3.5: Initial voltage island placement algorithm ............................................................................... 49
Figure 3.6: Illustration of the initial voltage island placement with initial seeding ................................... 49
Figure 3.7: Example of RAW weighting model for root tasks scheduling .................................................. 52
Figure 3.8: Task scheduling algorithm ...................................................................................................... 54
Figure 3.9: Routing path optimization using GA, example of how GA would solve conflicting routes ....... 58
Figure 3.10: Legal versus Illegal Routing ................................................................................................. 59
Figure 3.11: Pseudo-code for crossover and mutation ............................................................................... 60
Figure 3.12: Delay and power numbers are updated depending on PVT values ...................................... 64
Figure 3.13: Pre- and Post-Fabrication experimental setup ...................................................................... 65
Figure 3.14: Energy savings improvement compared to previous technique, (a) geometric average of 30 randomly generated TGs, (b) SMS (c) RCP (d) SPEC95, fpppp-kernel. ................................................................. 66
Figure 3.15: Energy-Delay-Product savings improvements, (a) geometric average of 30 randomly generated TGs (b) SMS (c) RCP, and (d) SPEC95, fpppp-kernel. ................................................................. 67
Figure 4.1: A task-graph scheduled on 9 cores showing the task’s slack times. ......................................................... 71
Figure 4.2: Voltage scaling to eliminate slack time ........................................................................................................... 72
Figure 4.3: Four Voltage Islands to be merged to two islands. ...................................................................................... 73
Figure 4.4: RCM Algorithm. ........................................................................................................................................ 81
Figure 4.5: Energy Optimization Flow. ........................................................................................................................ 84
Figure 4.6: Application parallelism versus Communication. .......................................................................................... 85
Figure 4.7: Cores’ voltage distribution before VSP for selected benchmarks with (a) high parallelism and low number of communication edges, and (b) low parallelism and high number of communication edges. 86
Figure 4.8: Energy increase for 25 benchmarks (RCM versus pre-selected voltages), due to selecting (a) two (b) three and (c) four out of 18 voltages. ................................................................................. 87
Figure 4.9: Runtime of DP and RCM. ........................................................................................................................... 90
Figure 5.1: Many-core (a) circular vs. rectangular shapes, (b) cloud-based Voltage Islands. ................................. 96
Figure 5.2: Example of the undirected graph of voltage islands (VI) to be placed. ................................................. 98
Figure 5.3: Seeding hierarchy used in the Voltage Island Clouds (VIC) placement .................................................. 99
Figure 5.4: Illustration of two islands and four applications in the case of (a) SVS and (b) MVS voltage island design approaches ........................................................................................................ 106
Figure 5.5: Platform partitioning algorithm for MVS approach. ................................................................................ 107
Figure 5.6: An illustration of platform partitioning for four applications. .............................................................. 108
Figure 5.7: Pseudo-code of Voltage Island Cloud placement algorithm. ................................................................. 109
Figure 5.8: Cloud-based voltage islands (a) before and, (b) after island merging, on 30x30 many-core, implemented using SVS, with $SW = \infty$ ........................................................................................................ 110
Figure 5.9: Cloud-based voltage islands (a) before and, (b) after island merging, on 30x30 many-core, implemented using MVS, $SW = mean (\Sigma CW)$. ................................................................. 111
Figure 5.10: Energy Optimization Flow. ................................................................................................................... 113
Figure 5.11: Selected task-graphs characteristics, number of communication edges versus (a) Parallelism, and (b) Critical path length.

Figure 5.12: Normalized energy consumption relative to SVS-RVI, versus (a) Communication edges, and (b) Parallelism.

Figure 5.13: Energy saving over SVS-RVI, $\sigma/\mu=0.12$; $\varphi=0.5$, (a) geometric mean of 30 randomly generated task-graphs (b) SMS (c) RCP, and (d) SPEC95, fppp-kernel.

Figure 5.14: Energy saving over SVS-RVI, $\sigma/\mu = 0.12$, $\varphi=0.1$, (a) geometric mean of 30 randomly generated task-graphs (b) SMS (c) RCP, and (d) SPEC95, fppp-kernel.

Figure 5.15: Energy savings over SVS-RVI, $\sigma/\mu=0.18$, $\varphi=0.5$, (a) geometric mean of 30 randomly generated task-graphs (b) SMS (c) RCP, and (d) SPEC95, fppp-kernel.

Figure 5.16: Energy saving after Voltage Selection, $\sigma/\mu = 0.12$, and $\varphi=0.5$; (a) geometric mean of 30 randomly generated (b) SMS (c) RCP and (d) SPEC95, fppp-kernel.
List of Tables

Table 3.1: Test Cases used in the Experiments [132]................................................................. 65
Table 4.1: Energy Cost Matrix..................................................................................................... 73
Table 4.2: Voltage selection using dynamic programming.......................................................... 75
Table 4.3: Voltage selection using RCM...................................................................................... 82
Table 4.4: Characteristics of the Benchmarks used [132].............................................................. 85
Table 4.5: Percentage energy increase (geometric mean of 30 benchmarks) when 2, 3, and 4 are selected out of 18 Vdds using RCM......................................................................................... 88
Table 4.6: Percentage Energy Increase when selecting 4 out of 18 voltages (geometric mean of 30 benchmarks) for different methods................................................................................................................. 89
Table 5.1: Example of CAW Calculation........................................................................................ 104
Table 5.2: Characteristics of the applications used in the Experiments [132].............................. 114
Table 5.3: Runtime Geometric Mean of the 33 Test Cases [132].................................................. 123
# List of Abbreviations

<table>
<thead>
<tr>
<th>Abbreviation</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>AEST</td>
<td>Absolute Earliest Starting Time</td>
</tr>
<tr>
<td>ALAP</td>
<td>As Late As Possible</td>
</tr>
<tr>
<td>ALST</td>
<td>Absolute Latest Starting Time</td>
</tr>
<tr>
<td>ASAP</td>
<td>As Soon As Possible</td>
</tr>
<tr>
<td>AVS</td>
<td>Adaptive Voltage Scaling</td>
</tr>
<tr>
<td>CAD</td>
<td>Computer Aided Design</td>
</tr>
<tr>
<td>CAW</td>
<td>Core Assignment Weight</td>
</tr>
<tr>
<td>CMOS</td>
<td>Complementary Metal Oxide Semiconductors</td>
</tr>
<tr>
<td>CW</td>
<td>Communication Weight</td>
</tr>
<tr>
<td>CU</td>
<td>Computing Unit</td>
</tr>
<tr>
<td>DSP</td>
<td>Digital Signal Processing</td>
</tr>
<tr>
<td>DVS</td>
<td>Dynamic Voltage Scaling</td>
</tr>
<tr>
<td>DVFS</td>
<td>Dynamic Voltage and Frequency Scaling</td>
</tr>
<tr>
<td>EDP</td>
<td>Energy Delay Product</td>
</tr>
<tr>
<td>FFT</td>
<td>Fast Fourier Transform</td>
</tr>
<tr>
<td>FIFO</td>
<td>First-In-First-Out</td>
</tr>
<tr>
<td>GPGPU</td>
<td>General Purpose Graphics Processing Unit</td>
</tr>
<tr>
<td>MPSoC</td>
<td>Multi Processor System on Chip</td>
</tr>
<tr>
<td>MVS</td>
<td>Multiple Voltage Scaling</td>
</tr>
<tr>
<td>NoC</td>
<td>Network on Chip</td>
</tr>
<tr>
<td>PVT</td>
<td>Process, Voltage, and Temperature</td>
</tr>
<tr>
<td>RAW</td>
<td>Root Task Assignment Weight</td>
</tr>
<tr>
<td>RC</td>
<td>Routing Cost</td>
</tr>
<tr>
<td>RCM</td>
<td>Removal Cost Method</td>
</tr>
<tr>
<td>RCP</td>
<td>Robot Control Program</td>
</tr>
<tr>
<td>RISC</td>
<td>Reduced Instruction Set Computing</td>
</tr>
<tr>
<td>RVI</td>
<td>Rectangular Voltage Island</td>
</tr>
<tr>
<td>SMS</td>
<td>Sparse Matrix Solver</td>
</tr>
<tr>
<td>SVS</td>
<td>Static Voltage Scaling</td>
</tr>
<tr>
<td>SW</td>
<td>Seed Weight</td>
</tr>
<tr>
<td>TAW</td>
<td>Task Assignment Weight</td>
</tr>
<tr>
<td>VI</td>
<td>Voltage Island</td>
</tr>
<tr>
<td>VIC</td>
<td>Voltage Island Cloud</td>
</tr>
<tr>
<td>VLSI</td>
<td>Very Large Scale Integrated Circuits</td>
</tr>
<tr>
<td>VPU</td>
<td>Vector Processing Unit</td>
</tr>
<tr>
<td>VSP</td>
<td>Voltage Selection Problem</td>
</tr>
<tr>
<td>WID</td>
<td>Within-Die</td>
</tr>
</tbody>
</table>
Acknowledgments

I would like to express my gratitude to my academic supervisors, Prof. Resve Saleh, Prof. Rabab Ward, and Prof. Steve Wilton. Prof. Saleh has gone far beyond the call of duty as a supervisor. His passion, dedication, and wisdom have mentored me throughout my degree. Prof. Ward has been incredible mentor, providing me with the opportunity, infinite support, and guidance during my PhD years. Prof. Wilton’s technical insight, patience, and positive outlook were very helpful to me during the difficult times of my degree.

I am also grateful to the other professors in the System-on-Chip Lab for providing the excellent opportunity and support. I would especially like to thank Dr. André Ivanov, Dr. Shahriar Mirabbasi, Dr. Guy Lemieux, and Dr. Alan Hu.

I am also very thankful to Dr Roberto Rosales for his help, guidance, and support during my PhD years. I would also thank Roozbeh Mehrabadi for his technical support.

I would like to also thank past and present SoC mates who made the PhD years an enriching experience. I especially thank Peter Hallschmid, Marwa Hamour, Shahrzad Jalali Mazlouman, Xiongfei Meng, Shaohua Yuan, Usman Ahmed, Uthman Al-Saiari, Rod Foist, Neda Nouri, and Dipanjan Sengupta.

Last but not least, I owe all my accomplishments to my family. Their tremendous love, unconditional support, and constant encouragement kept me going during the toughest times.
Chapter 1: Introduction

1.1. Motivation

A Multiprocessor System-on-Chip (MPSoC) is an integrated circuit composed of multiple processing elements (cores) that are fabricated together on a single chip to implement the functions of a complete electronic system [1]. An MPSoC can be homogeneous, sometimes referred to as a Chip Multiprocessor (CMP), in which all cores are identical, or heterogeneous, in which the processing components are different types of cores selected based on the target application domain. The heterogeneous components can be signal and video processing blocks, arithmetic logic units, multipliers, configurable blocks, memory blocks, and other specialized blocks.

The cores in these multi-core systems typically communicate with each other via a network of busses. In the case of systems with a few cores, a traffic management unit is used to control the utilization of busses among the cores. On the other hand, systems with a large number of cores have a more sophisticated network with routers or switches called a Network-on-Chip (NoC), also sometimes referred to as a communication fabric. Normally, the NoC handles communication between cores as well as input/output traffic, and data transfer to and from the on-chip memory [1].

MPSoCs are suitable for applications with high computational concurrency and real-time requirements; examples of these applications include video, audio, and any other form of telecommunications with an incoming data stream. For instance, video encoding requires multiple processing stages running simultaneously, such as prediction, transformation and quantization, and coding, among other operations. The prediction step
uses redundancies in the spatial and temporal data in video frames to reduce the data size and encode the useful information only. Transformation and quantization steps utilize mathematical techniques to transform and express the data to further compact the frames. Finally, the coding step uses compression techniques to produce the final compressed bit stream.

Since video frames enter at a high rate, e.g., 30 to 100 frames per second, and given the amount of computation that has to be performed for each frame, the incoming data frames are best handled in parallel. In an MPSoC, each stage is performed by a specific group of cores, each of which processes the incoming frames and then delivers the result to another group of cores to carry on the next task [1].

Many-Core platforms are the latest step in the evolution of MPSoC architectures. This new paradigm replaces the few complex cores in a multi-core platform by a larger number of smaller and simpler cores. The number of cores in a many-core architecture ranges from hundreds to thousands of cores on a single chip. It has been proposed that many-core architectures will provide a considerable increase in computing performance. These systems will require the development of new operating systems, software development tools to exploit the available parallelism in the applications, software debugging and evaluation tools, multiple application scheduling mechanisms, and hardware/software support for thermal and power management. These are key directions for research in this field and are all currently ongoing. However, the use of such architectures will also require an evolution in optimization techniques and design tools to provide a high performance within limited power budgets [2][3][4][5], and this is the main focus of the thesis.
1.2. Research Challenges

This major shift towards having chips with thousands of small cores comes as a result of two important and inevitable multi-core design obstacles. First, technology scaling is no longer sufficient to increase system performance. The increase in performance due to transistor scaling was used to boost the overall performance of a whole system. However, improved transistor performance does not translate into improved processor speed even at today’s scaling rate [2][6]. In effect, increasing processor frequency with each technology generation is no longer the primary method for improving processor performance. Increasing parallelism, on the other hand, is becoming the primary method to improve system throughput [4]. Second, as the trend in chip fabrication is expected to reach 100 billion transistors per 300mm$^2$ by the year 2015 [2], circuits are expected to suffer from high thermal and power density levels per unit area. Solutions to these two problems are beyond the capabilities of current battery and cooling technologies. As a result, the diminishing performance gain of scaling technology combined with the thermal and power consumption problems of large chips are creating new barriers for designing large complex processors on a single die [2][5].

A many-core system is better able to handle these two problems. Since overall performance is no longer improving as technology scales down, the new trend is to double the throughput by doubling the number of cores on a chip with each silicon generation [2]. By doing so, a many-core system can deliver a higher computing throughput than a multi-core for the same die area.

Many-core architectures also have better localized control over thermal and power consumption problems. Each core can be operated at its optimized
voltage/frequency or switched off completely to reduce the power consumption [2][7]. In addition, smaller cores have less thermal density per unit area compared to larger ones. Task scheduling can be carried out so that the workloads are distributed more evenly across the chip. Any hot cores can be switched off and their tasks can migrate to cooler ones to avoid reliability and leakage problems due to temperature increase. This allows a better balance of the temperature gradients across the chip [8].

Although changing course from multi-core into many-core is a necessity, this change requires development of new design automation and optimization techniques [2][4][9][10][11]. Due to the large number of cores expected to be used in a many-core architecture, traditional multi-core tools and techniques may not be sufficient [2][3][4]. Therefore, it is essential to address the power/performance design challenges for many-core architectures and develop the necessary optimization techniques to overcome them.

The most prominent problem to be solved is that of the power consumption [2][3][7][12]. Solving the power consumption problem of such massively parallel systems is vital in the current era of portable computing. The conventional perception that power is free but transistors are expensive is no longer valid in the new era of portable computing systems. In fact, it is quite the opposite; power is expensive but transistors are cheap [4]. Thus, having large number of cores in the design is only feasible as long as the power is kept to manageable levels.

Other problems such as routing optimization techniques need to be developed to handle communication traffic as well as possible deadlock scenarios of networks at the architectural level. Furthermore, a traditional memory hierarchy system is not adequate
to handle a chip with thousands of cores; thus, many-core architectures require a new innovative solution to the memory bandwidth problem [2].

Many techniques have been developed in the literature to reduce the overall power consumption in VLSI circuits [7][12][13][14][15]. An effective technique to reduce active power is by lowering the supply voltage of the non-critical parts of the design. In multi-core architectures, this is a suitable power saving technique since a multi-core system usually contains cores with different performance requirements. Thus, the voltage supply of the cores can be adjusted appropriately to meet the computational load requirements. Cores executing non-critical tasks are slowed down by reducing their supply voltage to save power. On the other hand, cores with heavy workloads are supplied the nominal voltage level so that the required overall performance is delivered [14].

Multi-supply voltage design, also referred to as multi-$V_{dd}$ design, incurs additional design complexities such as voltage level shifters, retention and isolation cells, and a complicated routing of the power grid network [7]. Furthermore, designers usually group cores with the same voltage level together to reduce the number of level-shifters required, and to reduce the power grid routing complexity [16][17][18]. These groups of cores operating at the same voltage level are referred to as Voltage Islands. Thus, a voltage island is a collection of cores in a contiguous physical area of the design operating at the same voltage level. The more flexibility in the implemented power saving technique, the more overhead it has in power grid routing and supporting blocks. Such flexibility can be provided by allowing on-the-fly adjustment for supply voltages of the islands based on workload, or by providing each core its power control mechanism.
Accordingly, these techniques have to be evaluated on many-core architectures, highlighting the advantages and disadvantages of each method, and then proper amendments and recommendations proposed to optimize the power saving on architectures with such a massive number of cores.

As technology scales down, process, voltage, and temperature variations are causing other major obstacles in multiprocessor design [3][19][20]. In the executive summary of the 2007 report, the International Technology Roadmap for Semiconductors (ITRS) continued to classify these variations as grand challenges to high chip yield. On-chip variations include process variations due to manufacturing deficiencies at nano-scale technologies, and voltage and temperature variations due to activity and workload distribution.

Process variations cause a deviation of transistor parameters from their expected values. These variations are due to the limited precision capabilities of photolithography at the fine-grain nano-scale level which results in transistor dimension inaccuracies and doping level variations [21][22][23][24]. The two key parameters most affected by these problems are the transistor threshold voltage and its gate length. The variability of these two parameters significantly impacts the transistor’s speed and power characteristics. By its physical extent, this impact is divided into two different categories: die-to-die, and within die variations. The outcome of the first category is a wide range of speed and power differences between different chips. On the other hand, the second category results in variability within each chip. Thus, within die variations lead to timing and power consumption variations within a chip and we address this problem for many-core designs in this thesis.
The main issue is that within-die process variation creates speed discrepancies among the different cores. Thus, some cores are fast and others are slow [25][26][27]. As a result, a many-core chip design has a wide range of core speeds differences that results in variations in timing and power consumption. In a voltage island based design, an island consists of a number of cores so similar effect occurs at this level. Thus, speed discrepancies due to within-die process variation can cause some of these cores to run slower than others within one island. We can mitigate this effect by increasing the supply voltage of the slow cores; however, this requires increasing the voltage of all cores within the island. Accordingly, the voltage level for each island is dictated by the slowest core in that island, and any increase of the island’s supply voltage results in increase of the total power consumed by that island.

Typically, voltage islands in small multi-core platforms are shaped as rectangular structures [28][16][29][30]. In this research, we show that this is no longer an acceptable choice in the presence of PVT variations for many-core architectures. Due to the gradient distribution of core speeds, inherent to within-die process variation, circular or cloud-based islands result in better power characteristics than long rectangular islands. The reason for this is as follows: the spatial correlation of within-die process variations ensures that neighboring cores have relatively close speeds. As the distance between cores increases, so do their potential speed differences. This variation in core-speed causes the fast cores within an island to be idle at times and consume static power while waiting for data transfer from the slow cores. Since the supply voltage value of an island is dictated by its slowest core, the wider the speed difference between its slow and fast cores, the larger the dynamic and static power consumption. Thus, to decrease these core-
speed discrepancies within an island, its cores have to be chosen from close neighbouring cores in a circular or a cloud-based formation such that the average distance between them is minimized.

Despite the unpredictability of Within-Die (WID) process variation, its nature is such that it will usually create a distribution gradient of core speeds. For this reason, creating the voltage islands from neighbouring cores has a better chance to reduce core-speed differences within an island after fabrication, and without predicting the exact core-speed before fabrication.

Such a policy, on the other hand, may have a negative impact on inter-island communication. For instance, the data transfer between two circular-shaped islands is worse than that of two rectangular islands sitting side-by-side, because the rectangular islands share a larger common border to simplify inter-island communication routing. Therefore, the shape of the island should also adapt to the level of communication between islands. Based on this insight, we pursue algorithms that create voltage islands that are optimized for the requirements imposed by communication patterns and also process, voltage, and temperature variations.

1.3. Research Goals

The primary goal of this dissertation is to investigate algorithms that attempt to reduce the power consumption of many-core architectures under the influence of core-speed variability due to process, voltage, and temperature variations. We approach this goal by proposing a design flow that utilizes multi-$V_{dd}$ techniques that involve voltage islands and by using algorithms that are suitable for systems with a large number of cores. For this purpose, we also develop an evaluation platform considering PVT impact
where multiple applications with hundreds to thousands of tasks are mapped onto many-core platforms with hundreds to thousands of cores to evaluate the proposed techniques.

The research methodology is as follows:

1. To develop a design flow for energy optimization for application mapping on many-core architectures and that accounts for the influence of process, voltage and temperature variations.

2. To develop a new voltage selection algorithm that can deliver near-optimal results and that is fast enough to handle large number of voltage levels and many-core architectures with hundreds to thousands of cores. This problem called the Voltage Selection Problem (VSP) [31].

3. To implement voltage islands with shapes that are well-suited to handle intra- and inter-island communication, and to limit their spatial extent in order to minimize core-speed differences due to PVT.

The results of this research work have been published in [32], [33], [34], and [35].

1.4. Thesis Organization

In Chapter 2, we present background information and prior work in this field. In Chapter 3, we present the improved design flow developed for energy optimization of application mapping on many-core architectures with initial experimental results. In Chapter 4, we focus on the voltage selection problem and describe a new method called the Removal Cost Method that is compared to prior solutions to the problem. In Chapter 5, we present a novel technique of voltage island formation for many-core architectures along with a new set of results. Chapter 6 provides conclusions and directions for future work.
Chapter 2: Background

2.1. Chapter Overview

This chapter presents the background and a literature survey on previous work in the areas of energy-efficient and PVT-tolerant multi- and many-core systems. Section 2.2 first describes trends in multi-core architectures and the recent shift to many-core designs. As it is the main platform for the work presented in the thesis, this section describes various types of many-core architectures, their components, and several prototypes that have been developed. Section 2.3 discusses how the impact of process, voltage, and temperature (PVT) variations on multi-core architectures can be modeled. The section first outlines sources of such variations, and then presents techniques used to mitigate their impact on cores in a multi-core system. Addressing the PVT problem for many-core designs is a key research direction in this thesis.

Section 2.4 presents common design techniques used in low-power multi-core designs. Section 2.5 presents a multi-$V_{dd}$ design flow previously described in the literature that creates an energy-efficient multi-core architecture. An improved flow will be presented in Chapter 3 and compared to this flow. Section 2.6 then describes and compares several techniques for implementing and managing designs with multiple voltage islands. A new approach to voltage island formation that incorporates effects of communication and PVT will be proposed in Chapter 5. Finally, Section 2.7 describes the voltage selection problem and previous techniques used to solve this problem that sets the stage for a new method to be described in Chapter 4.
2.2. Many-Core Architectures

The emergence of multi-core processor technology is a turning point in the computer architecture field. As technology scales down, it is increasingly difficult to obtain more computational power from a single processor core. Therefore, designers now implement several large processor cores on a single chip; this overall implementation is termed a *multi-core* architecture [2][4][36][37][38].

As chips get larger, it is anticipated that more and more cores will be integrated on a single chip. It is expected that chips will be available containing hundreds to thousands of cores. Typically, these cores will be relatively small and identical to one another, called *homogenous* cores. Such architectures are referred to as *many-core* architectures. The design of such architectures leads to important, yet unsolved, research questions related to the system modeling, optimal task mapping, power optimization, system evaluation and system verification of these architectures [2][39][40][41].

Recent many-core architecture prototypes include a heterogeneous mix of different core types such as traditional processing units, configurable datapaths, reconfigurable logic, custom co-processors, and memory blocks. In some cases, a small number of large processors supplement the many small cores to deliver good single thread performance. Special-purpose processors such as graphics engines can also be integrated to boost performance. In most cases, these cores are chosen and customized to provide the best performance for a specific range of applications [9][36][42][43]. However, the trend is to design and replicate a simple processor and data routing switch, both horizontally and vertically, to generate an architecture that is symmetric in nature. Each of the cores and routers are carefully designed and laid out so that they can be
easily tiled to create a large processing array of the desired size for the intended target applications. An example of a many-core architecture with a mesh Network-on-Chip is shown in Figure 2.1.

A Network-on-Chip is the backbone of such a system. The network carries data and instruction traffic to and from the cores [2]. Cache coherence traffic is also delivered using the network. Such networks often consist of a mesh of switches and routers. Each router is associated with a core, and it is connected to five two-way links, as shown in the inset. Four of the links are used to connect to other routers, while the fifth is used to connect to the corresponding core.

A critical issue in the design of such systems is power dissipation and thermal gradient management. Common techniques to address these issues include voltage and frequency scaling, dual or multiple threshold voltages, and power gating [2][44]. These methods reduce both dynamic and static power dissipated by the chip. The supply voltage of the core, its frequency, and the threshold voltages of the devices are set to
suitable values such that it satisfies the workload and performance requirements while minimizing energy consumption. The selection of these values can be fixed during design time, or adjustable during runtime.

![Figure 2.2: 167-processor many-core][1]

To date, several many-core architectures have been described in the literature. In [44], the authors present a 167-processor many-core prototype, optimized for DSP, communication, and multimedia applications. This mixed homogeneous/heterogeneous architecture is shown in Figure 2.2. The device contains 164 identical programmable processors, three application-specific processors, and three 16KB shared memories. The three application-specific processors are a Fast Fourier Transform (FFT) processor, a Viterbi decoder, and a motion estimation processor. The three shared memory blocks support multiple addressing modes and provide efficient on-chip storage. Each of the 164 homogeneous cores consists of a six-stage RISC processor that executes over 60 basic instructions, and contains an instruction register, a data register, and two FIFO memory blocks. The cores are connected using statically-configurable, circuit-switched, 16-bit wide links. Each core is equipped with a power management unit, capable of

---

[1]: https://example.com/figure2.png
changing the voltage and frequency dynamically to meet workload and power requirements.

In [45], the authors present a Massive Parallel Processing Array (MPPA) for reconfigurable computing. The array contains hundreds of compute units (CU); the CUs consist of 32-bit RISC processors, DSP processors, a configurable interconnect block, and memory blocks. The processors are configured by loading instructions into their dedicated memory blocks. Data traffic is carried using a network consisting of self-synchronizing channels. This network is hierarchal, containing intra-CU, inter-CU, and global network levels. The architecture was built assuming a Structural Object Programming Model. Objects consist of one or more software programs running concurrently on an asynchronous array of computing units and memories. Objects run independently in an encapsulated fashion. The data transfer between different objects is done through self-synchronized channels where each channel synchronizes its objects at each end.

The TILE-Gx processor family is a multi-core computing power that integrates 100 identical 64-bit processor cores; each core is referred to as a tile [46]. Each tile consists of a processor, level 1 (L1) and level 2 (L2) cache, and a router that connects the tile to the mesh network. Each tile can run an independent operating system, or a group of tiles can run a multiprocessor operating system. The tiles are connected using the TILERA’s iMesh™ on-chip network. The platform provides frequency scaling and a low power sleep-mode to reduce dynamic and static power.

There have been many attempts to utilize Graphic Processor Units (GPUs) for general-purpose computing [47]. Originally, GPUs were specifically designed as
graphics accelerators and they did not have the flexibility necessary to handle general purpose computing. However, recent GPU infrastructure includes support for general-purpose programming languages, such as C.

In [48], a many-core prototype named LARRABEE is presented. This architecture, designed by Intel© as a General-Purpose Graphics Processing Unit (GPGPU), utilizes multiple x86 CPU cores. Each core is supported by a Vector Processing Unit (VPU). Each CPU is supported by a two-level cache system. The L1 cache contains 64KB, and is divided equally between a data cache and instruction cache. There is also a L2 cache divided into separate subsets, one for each core. Each core can access its subset of the L2 cache independently. Finally, a ring network is used to carry the traffic between cores, memory, and the I/O interface.

2.3. The Impact of Variations on Many-Core Architectures

One of the key issues that must be addressed in these many-core architectures is the impact of on-chip or intra-die process, voltage and temperature (PVT) variations. These variations in CMOS integrated circuits pose a major obstacle for high-performance, low-power systems. Section 2.3.1 discusses how on-chip PVT variations can be modeled. The impact of PVT on many-core architectures is also explained. Finally, Section 2.3.2 presents methods from the literature to handle the impact of PVT.

2.3.1. Process, Voltage, and Temperature (PVT) Variations

The nature of the PVT variations changes the design issues from a deterministic problem to a probabilistic problem [20]. These variations are the result of many fundamental effects. Process variation is difficult to precisely predict at the pre-
fabrication stage. Such variations are caused by the impurities and imperfections of the manufacturing process in small-feature technologies. These imperfections have an impact on all parameters but are often characterized by their effect on two key CMOS transistor parameters: threshold voltage and gate length. Furthermore, these two parameters have major implication on the power and speed of the CMOS device [3][21][49].

Temperature and supply voltage fluctuations, on the other hand, occur dynamically during runtime. A heavy workload can result in high thermal impact, i.e., hotspots, and large voltage drops in the core’s supply voltage. Such variations cause a short-term impact on the speed/functionality and a long-term impact on the reliability of CMOS circuits. When the workload of the core is reduced or distributed more evenly across many cores, the impact of these two types of variations can be alleviated [32].

Process variations fall into two categories: die-to-die, and within-die. The first category is variation between different chips. The second category, the focus of this work, is the variation between cores within a single chip. It is also called on-chip or intra-die variations. When a chip experiences this kind of variation, some cores may be fast due to a lower threshold voltage but they are leaky and consume more static power. Other cores may be slow due to higher threshold voltage, and but consume less static power [3][50][23]. Faced with such variations, a designer may elect to run the entire chip at the speed of the slowest core. Better solutions are discussed later in this chapter.

Within-die (WID) process variations manifest themselves in two components: random and systematic. The systematic component is often due to lithographic lens aberrations that cause deviations in transistor dimensions. Random variations, on the other hand, are due to dopant density fluctuations, line-edge roughness, and other
localized effects [50]. In the last decade, researchers have developed modeling
techniques for this type of process variation. Commonly, statistical methods are used,
based on data extracted from actual manufactured prototypes. These models mimic the
actual behavior of process variation and its impact on both the threshold voltage ($V_t$) and
gate length ($L_{eff}$) distribution, and other important parameters [22][24][51] [52][53][50]
[54]. In particular, in [50], systematic variation is modeled as a multivariate normal
distribution, and random variation is modeled as uncorrelated normal distribution. This
type of modeling is simple enough to be used to analyze many-core architectures and it is
known to correlate well with empirical data from silicon measurements [32].

The maximum core speed in a multi-core design is inversely proportional to its
critical path delay, which is a function of the CMOS device’s threshold voltage and the
gate length. The critical path can be simply modeled as an inverter chain with the
appropriate sizing and loading effects taken into account [55]. As shown in Equation
(2.1), an inverter delay is a function of the supply voltage, $V_{dd}$, threshold voltage, $V_t$,
critical path’s capacitive load, $C_o$, and the technology dependent factors $\alpha$ and $K$. The
value of $K$ is dependent on the device characteristics and dimensions, as shown in
Equation (2.2), where $\mu$ is the mobility, $C_{ox}$ is the oxide capacitance, and $W$ and $L$ are the
transistor width and gate length, respectively.

$$D = \frac{C_0 V_{dd} / 2}{K \times (V_{dd} - V_t)^\alpha} \quad (2.1)$$

$$K = \mu \times C_{ox} \times \frac{W}{L} \quad (2.2)$$

The systematic process variation for two key transistor parameters, namely the $V_t$
and $L_{eff}$, can be captured using a multivariate normal distribution with a spatial
correlation structure. Thus, the impact of PVT on the core’s critical path delay can be measured through the delay equation considering different $V_t$ and $L$ values generated by the model.

To apply this model approach to an entire chip, we must first divide the chip into small equally-sized and square-shaped regions. Each region is given a normal distribution for $V_t$ and $L_{eff}$ with mean and a standard deviation. The correlation between two different regions on the chip is dependent on the distance between them [50]. Thus, given two points $\vec{x}$ and $\vec{y}$ on the chip, the systematic correlation, $\rho(r)$, between variation parameters $P(\vec{x})$ and $P(\vec{y})$ depends on the normalized distance $r$ shown in Equation (2.3).

$$\text{corr}(P(\vec{x}), P(\vec{y})) = \rho(r); \quad r = |\vec{x} - \vec{y}|$$

(2.3)

The systematic variation is then modeled as multivariate normal distribution with a spatial correlation function.

$$\rho(r) = \begin{cases} 
1 - \frac{3r}{2\varphi} + \frac{r^3}{2\varphi^3}, & r \leq \varphi \\
0, & \text{otherwise}
\end{cases}$$

(2.4)

The correlation function is shown in Equation (2.4), where $\varphi$ is the normalized distance at which the two regions are totally uncorrelated. For example, if $\varphi=0.5$, then two cores are uncorrelated if they are apart by a distance of half of the chip width or more; otherwise, they are close enough to be correlated in their speeds, depending on the value of $r$. Similarly, if $\varphi=0.1$, then they are uncorrelated if the two cores are apart by more than 10% of the distance across the array. Thus, by definition when $r = 0$, $\rho(0) = 1$,
i.e., totally correlated, and when \( r \geq \varphi \), \( \rho(r) = 0 \), i.e., totally uncorrelated. Given the correlation function, \( \rho(r) \), and \( M \) regions, the correlation matrix \((CM)\) is shown in Equation (2.5). Using MATLAB\textsuperscript{TM}, the systematic variation population can be generated.

\[
CM = \begin{bmatrix}
1 & \rho_{(2,1)} & \rho_{(3,1)} & \cdots & \rho_{(M,1)} \\
\rho_{(1,2)} & 1 & \rho_{(3,2)} & \cdots & \\
\rho_{(1,3)} & \rho_{(2,3)} & 1 & \cdots & \\
\vdots & \vdots & \vdots & \ddots & 1 \\
\rho_{(1,M)} & \cdots & \rho_{(M-1,M)} & 1
\end{bmatrix} \quad (2.5)
\]

\[
\sigma_{\text{total}} = \sqrt{\sigma_{\text{rand}}^2 + \sigma_{\text{sys}}^2} \quad (2.6)
\]

After adding the random component, the total \( V_t \) standard deviation is shown in Equation (2.6). Since these two components, systematic and random, are formed from two different physical phenomena, they are first computed separately and then added together [50].

An example of the systematic variation of \( V_t \) generated using the VARIUS\textsuperscript{©} model of [50] is shown in Figure 2.3. In this example, the range at which any two regions are totally uncorrelated is considered to be half of the chip size. A similar plot can be generated for \( L_{\text{eff}} \).
The random variation component of each region is represented by uncorrelated normal distribution and is due to random doping effects. The random and systematic components are modeled as having equal impact on the total variation, and they are assumed to be additive [50]. The total variation is measured using the ratio of the total standard deviation to the mean, $\sigma_{\text{total}}/\mu$. The standard deviation of both components can be calculated as $\sigma_{\text{sys}} = \sigma_{\text{rand}} = 0.12/\sqrt{2}$, assuming $\sigma_{\text{total}}/\mu=0.12$. On the other hand, the variation due to gate length variability is assumed to be half of that the threshold voltage [50]; thus, the standard deviation due to $L_{\text{eff}}$ are $\sigma_{\text{sys}} = \sigma_{\text{rand}} = 0.06/\sqrt{2}$.

![Probability density function](image.png)

Figure 2.4: Probability distribution of normalized frequencies in a chip with respect to $V_t$’s $\sigma_{\text{total}}/\mu$ ratio, nominal $V_t = 400\text{mV}$, $T = 25^\circ\text{C}$.

Figure 2.4 shows an example of the normalized frequency distribution for different values of $\sigma_{\text{total}}/\mu$, from 0.04 to 0.18 [32]. The mean frequency of the chip is decreasing and the range of the frequency distribution is increasing. Since variations are increasing at each generation, this figure shows that methods to handle WID process variation must be developed for many-core designs.
In a multi-core architectures, WID process variation manifest itself at the granularity of a core [38][56][27]. Thus, the model can be extended to a multi-core platform by dividing the chip into separate core regions. The process variation within each core region has a direct impact on the speed of the core and its switch. Based on the variation across a core and its location on the chip, the maximum frequency can be computed for each core. Using these results, a frequency table that indicates the maximum operating frequency for each core can be constructed [32].

![Figure 2.5: A 30x30 many-core with the normalized core frequency due to within-die process variation.](image)

Figure 2.5 illustrates a many-core platform with WID process variation. Process variation causes a frequency variation among cores in the same platform. As shown in the figure, WID process variation results in neighbouring cores to have similar speeds [27], but processors that are not in close proximity to one another may vary drastically in their speeds. This asymmetric distribution of frequencies creates major scheduling and synchronization problems that can result in system malfunction [19][20][27][26][32].

The second type of variation addressed in this work is temperature variation. The temperature of chip affects parameters such as threshold voltage [32] and leakage
The threshold voltage tends to decrease as temperature increases, leading to a higher speed and an increase in power consumption. The leakage current also increases as the temperature increases. In the literature, many techniques have been developed to capture the temperature variation of a multi-core architecture. The core temperature is normally modeled as being proportional to the power consumed by this core [57]. A heavy workload will result in an increase in the power consumed, thus increasing the core’s temperature. Temperature variation across a multi-core design is primarily due to non-uniform workload distribution and can impact speed, power and long-term system reliability [25][32][58]. Additional factors, such as neighboring cores’ temperatures and the capabilities of the cooling system, also affect the core’s temperature. Furthermore, the thermal behavior of the core also depends on its temperature history. Usually, the thermal behavior of the design is modeled as an equivalent circuit of thermal resistances and capacitances representing the cores at the architectural level. This circuit can also account for the heat flow between the cores [58]. If the temperature increase overwhelms the cooling system, a chip may enter a thermal runaway condition that eventually leads to the destruction of individual cores or the whole chip.

In order to compute the temperature variation in the system, Computer-Aided Design (CAD) tools, such as HotSpot©, can be used to estimate the temperature of each core during execution of a segment of code. To model the impact of the temperature on the threshold voltage $V_t$, a temperature scaling factor is used. The scaling factor for $V_t$ is shown in Equation (2.7), where $V_{t,nom}$ is the nominal $V_t$.

$$V_t = Scale \times T + V_{t,nom} \quad (2.7)$$
The final type of variation considered in this work is supply voltage variation. Usually, the power grid can be modeled accurately as a resistive network with each core represented as a current source in parallel with a resistance. The current source represents the dynamic power, while the resistor represents the leakage [32]. Each voltage supply has its own power grid which spans the entire chip. An example of a power grid is shown in Figure 2.6.

![Example of resistor network used to emulate the power grid.](image)

Figure 2.6: Example of resistor network used to emulate the power grid.

The current source representing the core is connected to only one of the grid nodes, depending on which voltage rail the core requires. This can be formalized by Equation (2.8).

\[
    i_m = \begin{cases} 
    i_n, & \text{if } C_n \text{ connected to } V_m \\ 
    0, & \text{otherwise}
    \end{cases} \quad (2.8)
\]

where core \( C_n \) is connected to a power grid \( m \) if the voltage of this core is connected to the supply voltage \( V_m \) as shown in Figure 2.7. Given an equivalent circuit, Modified Nodal Analysis (MNA) can be used to calculate the actual core’s supply voltage [59]. The matrices are calculated using technology-dependent resistor values. This method is very efficient, with acceptable accuracy at the system level, and thus is suitable for power supply variations for a chip containing thousands of cores.
2.3.2. Prior Work on Mitigating PVT Impact on Multi-Core Designs

In the literature, voltage and frequency scaling is the most common technique to mitigate WID process variations in multi-core systems. This technique is often used adaptively to overcome any changes in speed or power due to WID process variation [60][61][62][63][64]. The supply voltage of slow cores can be raised to reach acceptable performance. The supply voltage of fast cores can be reduced to lower the power dissipation and avoid undesirable temperature increases or hotspots. Such techniques are a preemptive way to avoid timing and scheduling errors. The voltage and frequency adjustment is conducted adaptively using specialized circuitry in a post-fabrication context.

In more sophisticated methodologies, tunable replica circuits are used in addition to voltage and frequency scaling. These replica circuits predict the error before it happens and adjust the supply voltages appropriately. Additional circuitry can also be implemented for error correction in some systems [61][64][65][66][67][68].

In some cases, PVT handling techniques, such as voltage and frequency scaling, are combined with energy-aware approaches to save on power as well. For instance, an intelligent assignment of tasks to individual cores can also be used to optimize for energy. In this approach, tasks are mapped to cores, and then the core’s supply and
frequency values are selected for each task to minimize energy while meeting performance and thermal requirements. Unlike the previous method, the voltage and frequency values change dynamically as the task running on each core changes [64][69]. This issue is discussed in detail in a later section.

![Diagram showing voltage interpolation to compensate process variation effects](image)

**Figure 2.8:** Voltage interpolation to compensate process variation effects.

Voltage interpolation with dual supply voltages is a simpler method to handle frequency discrepancies among cores due to process variation [70][71]. In its simplest form, there are two supply voltages available to each core in the design; slow cores connect to the higher supply voltage and fast cores connect to the lower one as shown in Figure 2.8. This method is usually realized through $V_{dd}$-gating transistors acting as a switch to route the correct supply voltage to the corresponding cores. Compared to dynamic voltage and frequency scaling, this technique requires simpler circuitry and less overhead, but does not provide high flexibility for optimization after fabrication.

Variation-aware task scheduling and application mapping using statistical methods is another technique to handle speed variability among cores [72][73][74][75][76][77]. The simplest scheduling policy assumes a worst-case scenario to avoid timing errors; thus, each task is given execution time assuming it is running on the
slowest core. This worst-case analysis can result in an overly pessimistic situation. Task scheduling based on statistical timing analysis provides more optimistic and realistic results. Although statistical timing analysis was first introduced at the gate-level, recently, these techniques have been extended to the system-level in many-core architectures. Each task is given a distribution of the execution times at which it can start or finish. The total execution time of the application is then represented by a normal distribution rather than by a single delay number.

Circuit redundancy is a traditional technique in fault-tolerant systems to handle fabrication defects and system failures. It is also been introduced to help eliminate core speed and power variability problems in many-core platforms. Faulty (or slow) cores due to process variation can be avoided and replaced by spare cores. The topology of the design can be modified post-fabrication. This technique requires reconfigurability to allow such modifications in a post-fabrication context.

There have been many published techniques to reduce hot spots and thermal variation [78][79][80][81][82]. Although a low threshold voltage due to process variation contributes to the creation of hotspots, temperature variation is primarily a transient phenomenon that results from workload allocation. Traditional as well as new load-balancing techniques are thus used to avoid thermal problems in the design. These techniques use task migration and rescheduling to optimize thermal properties while minimizing scheduling overhead. Scheduling overhead is especially significant when tasks are migrated, since task migration involves the complete reassignment of a task to a cooler core while obeying all of its data dependencies.
2.4. Prior Energy-Aware Techniques for Multi-core Designs

2.4.1. Power Consumption Basics

The two types of power consumption in VLSI CMOS circuits are dynamic and static power, also known as active and standby power, respectively. The dynamic component occurs while the circuit is switching [7]. Dynamic power is proportional to the average amount of switched capacitance per cycle, $C_{eff}$, the frequency of operation, $f$, and the square of the supply voltage, as shown in Equation (2.9).

$$P_{dynamic} \propto C_{eff} V_{dd}^2 f \quad (2.9)$$

The other type of power consumption is static power, which is consumed in CMOS circuits while the circuit is idle. The static power is proportional to the leakage current and the supply voltage as shown in Equation (2.10), where $V_t$ is the threshold voltage, $T$ is the temperature, $q$ is the electron charge, $k$ is the Boltzmann constant, and $n$ and $I_o$ are technology-dependent constants [7].

$$P_{static} = I_o e^{\frac{qV_t}{nkT}} V_{dd} \quad (2.10)$$

2.4.2. Low Power Design Techniques

Many techniques have been developed for controlling dynamic and static power consumption at the circuit level. Techniques such as transistor size adjustments, clock gating, power gating, multi-$V_t$, and multi-$V_{dd}$ methods are commonly used to reduce power in CMOS circuits. Some of these techniques can be implemented for managing the power consumption at the core-level.
Multi-V$_{dd}$ methods can be realized by assigning lower voltage levels to cores running less critical workload to save on dynamic power. Voltage assignment can be performed during design time with a fixed assignment, or during runtime with a dynamic assignment to handle variable workloads, as we will elaborate in later sections.

Multi-$V_t$ methods are primarily used to save on static power. As shown in Equation (2.10), reducing the supply voltage has essentially a linear impact on static power. However, increasing the threshold voltage $V_t$ has exponential effect on power reduction. On the other hand, increasing $V_t$ makes CMOS circuits slower, hence it results in speed degradation for the core. Therefore, a multi-$V_t$ system can be implemented in each core, where slower or idle cores use a higher $V_t$ value and more critical cores would use a lower $V_t$ to keep the desired performance [7][83].

Power gating is another technique used to reduce dynamic as well as static power. It prevents undesired switching activity, also known as glitches, from propagating into idle logic which would consume power unnecessarily. Typically, unused cores in a multi-core platform are switched off completely [1].

2.5. Multi-Vdd Design Flow

This section discusses approaches for the implementation of multi-core architectures with multiple supply voltages (often termed multi-Vdd design or voltage island design). Multi-V$_{dd}$ design is a technique that aims to reduce power by reducing the supply voltage of the non-critical parts of the design but maintaining the higher voltages for critical parts of the design [7][10][14][15][84][85][86][87][88][89][90][91]. Reducing the supply voltage also lowers the required clock frequency of a core and this helps to reduce the power even further, according Equation (2.9). Often, the supply
voltage is set by considering the amount of core idle time, or its available *slack time* (time between the end of a task and when the results are needed by another task).

Multi-Vdd voltage assignment is normally carried out as part of a larger flow, as outlined in Figure 2.9 [14][16][38]. This will be the reference flow for the work described in Chapter 3. At the top level of the flow, the applications are usually represented by directed acyclic graphs (DAG), referred to as *task-graphs*. Each task-graph consists of nodes representing tasks, where each task has a specified computational time. The directed edges of the task-graph represent the flow of data between tasks. A task can be executed only when all its incoming communications are finished.

The proper underlying multi-core architecture that delivers the needed functionality is selected based on the type of the application. In doing so, the types of computational blocks that will be used in the implementation (sometimes referred to as tiles) are selected. Examples of such tiles include DSP blocks, programmable blocks, memory blocks, and input/output blocks. Tiles can also be the basic processor/router unit that comprises the homogeneous many-core arrays, as mentioned earlier.

The first step in the flow shown in Figure 2.9 is the tile mapping which is choosing the proper tile type for each task. In prior work, a genetic algorithm (GA) is utilized in the tile mapping step. A genetic algorithm is an evolutionary algorithm that utilizes the same analogy as evolutionary biology. It is used frequently in this flow, so a brief description of GA is presented.

The algorithm starts by creating a population of possible solutions. Then, it performs crossover and mutation of selected solutions from the population based on a fitness function where the offspring inherit the good qualities of the parent solutions. The
population solution space evolves into better quality solutions following successive 
crossover and mutations until it reaches stopping criteria.

Figure 2.9: Multi-V\textsubscript{dd} Design Flow.

In next step of the flow, the routing paths that carry the data traffic among 
different tiles are specified using another genetic algorithm. Tasks are scheduled such 
that dependencies are obeyed and timing constraints are satisfied [83]. Note that as the 
number of cores in the array increases, the above algorithms become unsuitable as they 
were originally intended for architectures with a small number of cores, e.g., 9 cores.

After the tasks are scheduled, the design can be optimized for energy. The voltage 
scaling step slows down selected tasks based on their slack times [87][92][93][94][95]. 
Again, the available slack is defined as the difference between the time the task finishes 
and its deadline. Non-critical blocks having high slack times can be assigned to lower 
supply voltages. In the example of Figure 2.10, Task 1 has 3ns of slack time, meaning 
that the supply voltage of core A can be reduced.
In the worst case, each core would have its own unique voltage level. In doing so, the number of unique $V_{dd}$ values selected for the design can be considerably large. In such a situation, it would be impractical to build a power grid on a chip that can deliver so many different voltage levels. Therefore, the number of unique voltage levels needs to be reduced in an efficient manner, as described in Chapter 4. This problem is referred to as the Voltage Selection Problem or VSP. Furthermore, it is more desirable to group cores that share the same voltage level to reduce the complexity of power grid routing. These groups of cores are referred to as voltage islands or power domains, and it is performed during the Voltage Island Formation step as shown in Figure 2.9. All cores in a given voltage island operate at the same frequency, but different islands will operate at different frequencies.

The next step in the flow is to evaluate the potential implementation using a cost function. If the power or performance does not meet the requirements, the flow iterates through the steps again, producing a new solution. This can be guided by either a
deterministic algorithm or an optimization algorithm employing some amount of randomness, such as simulated annealing or a genetic algorithm (GA).

Although the main optimization stages are the same in most energy optimization flows, the details may vary [14][83][86]. For instance, routing the data traffic can be performed using GA, XY-routing, or Dijkstra algorithm [16]. Furthermore, the tile-mapping step can be eliminated in the case of a homogeneous multi-core design. These modifications can speed up the optimization process with some implications on the optimization results. The subject of Chapter 3 is the development of an improved flow that directly addresses the issues of many-core designs, starting with the flow of Figure 2.9 as a baseline. Prior implementations of this flow did not address the large number of processors of many-core designs in the algorithms at each stage, including task scheduling, routing and voltage selection. Nor did they address the formation of voltage islands considering PVT and communication patterns.

2.6. Voltage Island Design Styles

This section presents three methodologies to realize designs with multiple supply voltages. These three approaches are: fixed voltage islands or Static Voltage Scaling (SVS), Dynamic Voltage and Frequency Scaling (DVFS), Multiple Voltage Scaling (MVS). In later chapters, only the SVS and MVS approaches will be investigated further for reasons cited below.

2.6.1. Fixed Voltage Islands with Static Voltage Scaling (SVS)

The simplest technique for creating multiple voltage islands is to design voltage islands before fabrication based on static voltage scaling. Fixed voltage islands can be
realized at fine or coarse granularity levels [96]. Fine granularity requires significant overhead in terms of area and power. In multi-core architectures, voltage islands are built at the core level and hence employ a more coarse granularity [97][98].

The use of voltage islands brings additional complexity to the design process. For example, timing analysis, power grid routing, and floorplanning all become more complex [99]. Furthermore, voltage islands need to be placed close to the power pins to avoid power routing complexity and high IR drop. All cores with the same supply voltage should be grouped together to reduce the number of islands, which also reduces the complexity [28]. Each island requires its own power grid. Finally, power and area overhead of the supporting blocks, such as level shifters, must be accounted for during power optimization and estimation [98][100].

The drawback of using a fixed voltage island design is that the voltage levels cannot be changed after fabrication. Thus, it must be optimized for a specific application running on the many-core architecture; if the application changes, the voltage island design may no longer be suitable.

2.6.2. Dynamic Voltage and Frequency Scaling (DVFS)

Dynamic voltage and frequency scaling (DVFS) is more flexible than SVS. In DVFS, the voltage and/or frequency are adjusted after fabrication based on given performance and power budgets [44][101][102]. This technique is useful when the workload to be implemented on the many-core architecture is unknown at design time. DVFS can be used to handle process variation and thermal management as well. In most cases, DVFS is implemented at a high granularity, (at the core level); otherwise it requires considerable area and power overhead [103][104][105]. Dynamic voltage
scaling is sometimes referred to as Adaptive Voltage Scaling or AVS when the change in $V_{dd}$ happens at very small intervals [106][107].

In order to implement the DVFS technique in a multi-core, significant area and power overhead is required, as reported in [108][109][110][111][112][113][114]. A power management unit consisting of a feedback control loop to adjust the voltage and frequency is required. A workload monitor assesses the required speed needed for the core based on currently allocated workload. It attempts to predict the correct frequency and voltage at which the core should be operating.

Figure 2.11 shows a block diagram of the main components in a power management unit for DVFS. The process is composed of three steps. The first step is performance monitoring to estimate the current processor performance. After assessing the current performance, a performance comparator measures the difference between the current and desired performance and evaluates the voltage difference, $\Delta V$, and reports it to the power supply to deliver the required voltage level [103][106][115].

As stated before, DVFS may provide a better adaptation to the workload dynamically during run-time but it incurs considerable overhead. In a multi-core design,
each voltage island has to have its own power management unit. Typically, a DVFS implementation in a multi-core design considers each core to be a separate island. The DVFS technique may also suffer from robustness problems. To avoid these problems, constraints on the range through which the supply voltage can vary are required. While it is a promising approach, these constraints limit the benefits of DVFS [116]. Therefore, it is not considered further in the research work described in later chapters.

2.6.3. **Voltage Islands with Multiple Voltage Scaling (MVS)**

Multiple Voltage Scaling (MVS) does not suffer the same drawbacks as DVFS. In this technique, the number of available supply voltages is usually small, typically three or four voltage levels. Each voltage level has its own power grid. Each core can be programmably connected to one of the power grids based on its workload requirements [104][116][117][118][119]. These connections can be programmed at configuration time. Thus, this approach is more flexible than fixed voltage islands and has less design complexity than dynamic voltage scaling [120][121][17][122][123][124].

The simplest form of MVS has two supply voltages. Each core has two header devices, one header device for gating to a high voltage \(V_{ddH}\) and a second one for the low voltage \(V_{ddL}\). The value of these two voltages is static and do not change during run time. Compared to DVFS, the transition time and energy overhead is significantly reduced in MVS. On the other hand, the design of the MVS power grid is challenging, and is an on-going research topic [2][41]. The complexity of the power grid increases as the number of the supply voltages increases [118][122][33][125][126].

Figure 2.12 shows an example of MVS with three supply voltages and total of four power modes. The power grid selector selects the proper supply voltage. Unused
cores can be switched off by selecting the last available power mode shown in the figure.

In an MVS design, the selection process is performed during system configuration and not on-the-fly as in DVFS. A more sophisticated system allows changes in the values of the power modes, i.e., the values of $V_{ddH}$, $V_{ddM}$, and $V_{ddL}$ [44][127].

![Figure 2.12: Different power modes in MVS design.](image)

### 2.7. Voltage Selection Problem

As described above, if the voltage level for each core is selected independently during optimization, the number of unique voltage levels selected for a multi-$V_{dd}$ design can become very large. Determining a smaller set of voltage levels for a design is referred to as the Voltage Selection Problem. Many heuristics have been proposed to solve the VSP problem [31][34].

Voltage selection techniques are classified into on-line and off-line approaches. On-line voltage selection is performed during run time to adaptively adjust to a changing workload. It requires a special circuitry and introduces additional complexity. In this thesis, we focus on off-line techniques where voltages are chosen at design time [14][31].
Integer linear programming (ILP) is one of the many techniques used to solve the voltage selection problem. The problem is modeled to optimize performance and power given time constraints. Modified versions of ILP are used to account for static power optimization by including body biasing and power shutdown in the ILP problem formulation. The proposed methods claim near-optimal solutions for the benchmarks they examined [31].

One-Step-Look-Ahead (1-STEPLA) is an algorithm used for voltage selection for multi-core system [16]. It first groups cores together and assigns a voltage to each group that satisfies timing constraints. The algorithm employs a trial-and-error approach grouping two neighbouring cores at a time. The algorithm continues until it reaches the allowed number of voltages in the design. This algorithm can provide acceptable results for a multi-core with small number of cores, e.g., 16 cores as in [16]. However, as the number of cores and corresponding number of voltages increase, this approach begins to reach its computational limits. Therefore, a better approach will be proposed in Chapter 4 and compared to the 1-STEPLA method.

Dynamic programming is another approach used to optimize the selection of voltage levels. This algorithm starts by first assigning the highest available voltage for all cores. Then, combinations of lower voltages are considered. A higher voltage is replaced by a lower voltage if the total energy is decreased and the time constraints are still satisfied. The algorithm stops when it reaches the number of allowed voltages set by the designer. This approach gives close-to-optimal results [29], and will also be compared to the new approach in Chapter 4. Methods based on randomized heuristics such as simulated annealing have also been used [128].
Other researchers have presented heuristics that combine voltage selection with other goals. In [29], voltage selection is combined with threshold voltage selection to optimize dynamic as well as static power. Furthermore, the energy consumed in the network links and switches are considered in the selection of the optimal voltage levels to minimize the total energy. Level shifters are also considered in some techniques [128].
Chapter 3: Energy Optimization Flow

3.1. Chapter Overview

In this chapter, we describe the proposed improvements to the energy optimization flow for many-core architectures in the presence of on-chip process, voltage and temperature (PVT) variations. This flow is contrasted with the one presented in Chapter 2, Figure 2.9, and provides the context within which task scheduling, voltage scaling, voltage selection and island formation techniques are utilized to produce an energy-aware mapping onto many-core architectures.

Our proposed algorithms employ a number of novel strategies relative to the previous approach, particularly during task scheduling, voltage selection, and core and island placement. Specifically, the task scheduling and voltage selection improvements address the prior limitations when handling the large numbers of cores in many-core designs, while the core and island placement incorporate the communication and PVT factors previously absent.

This chapter is organized as follows. Sections 3.2 and 3.3 describe the target platform characteristics and the application representation, respectively. Section 3.4 then presents an overview and limitations of the previous CAD flow and introduces the improved flow to address the limitations. The new flow details are then described in Sections 3.5 to 3.11. Finally, Section 3.12 presents the preliminary results and Section 3.13 concludes the chapter. The details of the voltage selection process are quite involved

---

1 This chapter is based on the contents of the papers [32], [33] and [35].
and will be provided in Chapter 4. Similarly, the voltage island formation process is complex and covered separately in Chapter 5.

3.2. Platform Characteristics

The baseline multiprocessor architecture is the same for the implementation of the previous and new flows. We consider a homogeneous many-core architecture containing $N$ processing cores that can run multiple applications simultaneously. Each core contains a processor, along with a local cache. The cores are interconnected using a mesh topology consisting of $N$ routers/switches. As shown in Figure 3.1, each core is connected to a switch, and every switch is connected to four adjacent switches via two-way links. The network-on-chip is a two-dimensional mesh topology network that carries the data traffic between cores. The operating frequency of the network is twice as fast as that of the cores. The core and its router are operated by the same supply voltage. We also assume voltage scalable communication links; thus, the bandwidth of the link is scaled with the corresponding supply voltage. The data traffic is routed by each router on First-Come-First-Serve basis, where the buffer at each router has enough capability to store all incoming messages. The router can start transferring data on a link only when the link is idle. One message at each router clock cycle can be sent.

![Diagram of a core, its communication switch/router and buses.](image)

Figure 3.1: A core, its communication switch/router and buses.
3.3. Application Representation

The many-core architecture is intended to run multiple applications separately. Each application to be run on the architecture is represented by a directed acyclic task-graph (DAG) as shown in Figure 3.2. A task-graph contains $M$ vertices, or tasks (shown with a $T_i$ label for Task $i$), and $L$ directed edges, each representing one or more data dependencies between two tasks.

A task can not be executed until the data is available on each incoming edge. Each task has an associated nominal computation delay, $D_N$ (not shown in the figure), which is an indication of its run-time without considering affects such as PVT impact or voltage scaling. Each edge has a communication weight $CW$, which is proportional to the amount of data transferred between tasks connected using the edge. The application is assumed to start execution at a set of root tasks; in general, there may be more than one root task for a given application.

![Figure 3.2: A Directed Acyclic Graph (DAG).](image)

Each application is associated with a user defined application priority level. In addition, each task within an application is associated with a scheduler determined task priority level. During mapping, these priority levels will be respected to maximize the efficiency of the resulting implementation.
3.4. Overview of the Optimization Flow

The purpose of the flows to be described is to map the set of task-graphs representing the applications to the many-core architecture. Each task (node) is assigned to a core, and each edge is assigned a communication path through the routing network. This section begins with an overview of the reference flow; the details of the new flow are presented in subsequent sections.

The energy optimization flow used in previous work is shown in Figure 2.9 and was discussed in Chapter 2. Since we address only homogeneous many-core architectures, the outer-most genetic algorithm is eliminated from the previous flow. It uses the As-Soon-As-Possible scheduling algorithm to schedule tasks onto cores. Next, voltage scaling is performed followed by core grouping to form voltage islands.

Unfortunately, the previous flow was intended for heterogeneous architectures with only a small number of processors and is therefore not suitable for homogeneous many-core architectures with a large number of cores considered here. In fact, the two nested genetic algorithms can become very slow as the numbers of tasks and cores increase, which is the case in the many-core architecture. Furthermore, the previous scheduler does not consider the root task’s communications with its successors when scheduling. This may not be an issue for small multi-cores, e.g., 3x3 cores where the maximum Manhattan distance is 6 blocks, but it is of vital importance in the case of many-core designs with hundreds of cores; e.g., 30x30 cores have a maximum Manhattan distance of 60 blocks. Additionally, the effects of PVT variations, along with the degree of intra- versus inter-island communications, are not considered when forming islands in the previous flow.
The new flow addresses all of these issues. A key contribution of our flow is the use of a two-pass approach to eventually form cloud-based voltage islands. An initial voltage selection and island formation scheme using communication information (used in the first pass) is subsequently refined in a second pass when the final voltage levels are defined based on actual task assignments to processors.

The new flow is as shown in Figure 3.3. The flow has two main parts, voltage island formation (Steps 1 through 5), and a genetic algorithm loop for data routing optimization (Step 6). In the first part, voltage islands are built in five successive steps.

![Figure 3.3: Improved Energy Optimization Flow.](Image)

Initially, voltage islands are formed based on computed task voltages using the delay information and rough estimates of communication routing costs through the network before any placement is performed (Step 1). Using these initial voltage islands,
regions of the chip are assigned different voltages (Step 1) and then a scheduling of tasks onto the architecture is performed (Step 2). It is now possible to compute a better estimate of the routing delay of each data transfer. Using the more accurate task and routing delays, voltage assignments for each task are adjusted (Step 3). Voltage selection is then performed to determine which islands will be merged together later to reduce the number of supply voltages in the design (Step 4). Final voltage island formation (Step 5) is realized according to two factors: communications and the spatial extent of the island to limit the effects of PVT. They are implemented through weighting models that accounts for the communication patterns between islands and PVT impact.

As a last step of the process, the actual routing paths for each data transfer are determined (Step 6). We use the routing algorithm described in [16] and [100]. The goal during routing is to find paths for each data transfer while minimizing the amount of routing congestion. A genetic algorithm is used with a fitness function that reflects the total energy consumed by the architecture including cores, switches, communication links, and level shifters. Further details of all the steps are given below.

### 3.5. Initial Voltage Scaling

In this section, we describe the algorithm for initial voltage scaling and the initial phase of constructing the voltage islands (Step 1). The goal is to establish a first-order plan for constructing the islands, which can be improved on the next pass with more accurate information. We also list some of the assumptions considered in this phase. The first step is to perform a timing analysis of the tasks to determine the slacks. We can take advantage of the available slack time by reducing the supply voltage to slow down a given task of its prospective core to minimize the power dissipation.
The challenge in using slack to scale voltage this early in the design flow is that, at this point, we have not yet determined a physical allocation of tasks onto cores. Thus, we do not have information about routing delays or expected PVT variations. However, we can still estimate the amount of slowdown that can be tolerated using each task’s nominal computational time and communication weights, by assuming worst-case PVT and worst-case routing delay. The worst-case routing delay is calculated assuming that data to be transferred simultaneously from several cores to the same destination core, at the same time, sharing the same routing path. So there will be significant contention for the same routing resources.

The algorithm for performing initial voltage scaling, shown in Figure 3.4, is the same as in the previous work [16]. For each task, we calculate the slack of that task, which is defined as the difference between its As-Soon-As-Possible (ASAP) and As-Late-As-Possible (ALAP) times. Intuitively, this is the amount that the task can be slowed down without increasing the length of the critical execution path [16][129]. Based on the estimated slack time for each task, the voltages of their prospective cores are determined. The algorithm first starts by identifying the tasks on the critical execution path. The tasks of the critical path are assigned the highest available voltage. Then the algorithm iterates through non-critical tasks, starting with the ones with the minimum slack.

Given the slack time of a task, a preferred voltage is then assigned to the task. The algorithm cycles through a set of allowable voltages between $MinV_{dd}$ and $MaxV_{dd}$. For each prospective voltage, Equation (3.1) is used to refine the delay estimate of that task by assuming that the potential voltage is supplied to the core implementing that task.
In this equation, $D_N$ is the nominal delay, $V_{ddS}$ is the scaled voltage, $V_{ddN}$ is the nominal voltage, $V_t$ is the threshold voltage and $\alpha$ is a technology-dependent constant.

$$D_{scaled} = D_N \times \frac{V_{ddS}(V_{ddN} - V_t)^\alpha}{V_{ddN}(V_{ddS} - V_t)^\alpha} \quad (3.1)$$

<table>
<thead>
<tr>
<th>L.#</th>
<th>Algorithm 3.1. Voltage Scaling</th>
</tr>
</thead>
<tbody>
<tr>
<td>Inputs</td>
<td>Task graph, Available Voltage levels;</td>
</tr>
<tr>
<td>Outputs</td>
<td>Tasks' required Voltages and new delays, Number of cores in each island; Inter-Island Communication;</td>
</tr>
</tbody>
</table>

```plaintext
begin
  /* Initialize the number of cores in each Island to zero*/
  IslandNoCor{M invV_{dd}, MaxV_{dd}} = 0;
  for i = 1 to M; all Tasks do
    calculate ASAP for task $T_i$
    calculate ALAP for task $T_i$
    Slack($T_i$) = ALAP($T_i$) - ASAP($T_i$)
  end
  for i = 1 to M; all Tasks do
    select $T_i$ such that Slack($T_i$) is minimum
    for $V_{dd} = M in V_{dd}$ to Max$V_{dd}$; all available supply voltages do
      $D_{scaled} = D_N (T_i) \times \frac{V_{dd}(V_{ddN} - V_t)^\alpha}{V_{ddN}(V_{ddS} - V_t)^\alpha}$
      if $D_{scaled} \leq D_N (T_i) + Slack(T_i)$ then
        $V_{sat}(T_i) = V_{dd}$;
        New $D(T_i) = D_{scaled}$;
        break loop
      end if
    end for
    if all available Cores of IslandNoCor{$V_{dd}$} are occupied during $T_i$ time interval then
      /* increment the size of voltage island$V_{dd}$ by one core */
      IslandNoCor{$V_{dd}$} = IslandNoCor{$V_{dd}$} + 1;
    end if
    for $T_i$ all Predecessor tasks of $T_i$ do
      Inter-Island[$V_{dd}$ of $T_i$; $V_{dd}$ of $T_j$] = Inter-Island[$V_{dd}$ of $T_i$; $V_{dd}$ of $T_j$] + CW($T_i$, $T_j$);
    end for
  end for
end
```

Figure 3.4: Voltage scaling algorithm.

Since the algorithm cycles through potential voltages in increasing order, the delay from Equation (3.1) will decrease as we iterate through the loop. Iteration continues until the scaled delay is less than the nominal delay plus the slack of that task.
is found, i.e., $D_{scaled} \leq D_N + slack$. That voltage is selected as the *preferred* voltage for that task.

Note that a discrete number of voltages is used for the inner loop of the algorithm. This reflects the fact that only a small number of voltages can be physically supplied to a chip. Clearly, the step size in the inner loop will influence the total number of different voltage levels selected for all tasks. We choose a relatively fine granularity at this step, and then the total number of selected voltages is reduced to a smaller number, in the voltage selection step described in Section 3.8 [16][29][128].

In addition to calculating the preferred voltage for each task, the algorithm also keeps track of the number of tasks assigned to each voltage level, and constructs initial voltage islands based on these assignments. All tasks assigned to the same voltage level are assumed to be grouped in the same voltage island (so there is one voltage island per prospective voltage level). The algorithm also keeps track of the amount of communication between each pair of voltage islands.

### 3.6. Initial Voltage Island Formation

The algorithm in the previous section provides a preferred voltage assignment for each task, as well as an initial assignment of tasks to voltage islands. The next step, shown in Step 1 in Figure 3.3, of our flow is to position and shape the islands on the chip in such a way as to maximize the effectiveness of voltage islands with regard to inter- and intra-island communications. This section describes the manner in which we perform this positioning.

Our new algorithm for this purpose attempts to find a solution that balances two objectives. First, we wish to shape the islands such that pairs of cores from two different
islands that share large amounts of communication be close together. Second, we ensure that the furthest distance between any two cores within the same island is as small as possible. This will tend to reduce the amount of within-die variation between cores in the same island, and it will improve intra-island communications. These two goals are often conflicting; the first goal suggests that long rectangular islands are best since highly-communicative inter-island neighbours can be placed next to each other in adjacent rectangles, while the second goal suggests that circular islands are best since this would tend to shorten the maximum distance between intra-island cores. The results of our approach produce irregular islands, with different sizes and shapes determined by the objectives listed above, and we refer to them as Voltage Island Clouds or VICs.

The initial voltage island placement algorithm is shown in Figure 3.5. We first choose a physical location on the chip as the placement seed. In our implementation, the placement seed is set to be one corner of the chip (other locations were tried, but this one produced the best results) and the islands are positioned in a contour-like pattern around the placement seed. We then greedily determine the physical extent of each voltage island as follows. We cycle through all voltage islands determined by the voltage scaling algorithm in a number of iterations:

1. In the first iteration, we consider the island with the highest voltage, since it executes tasks with the highest priority.

2. In subsequent iterations, we consider the island that has the highest inter-island communication with those islands already placed.

For each voltage island, we then first choose a location on the chip as the local seed for that island. The Euclidian distance is considered in all distance measurements in
this phase. The local seed is the closest unused slot to the placement seed. The algorithm then iterates through all cores within that island starting with the one with lowest inter-island communication. For each core, we identify the closest unused slot to the local seed, and place the core in that slot as illustrated in Figure 3.6. Eventually, cloud-based voltage islands are formed using this approach and this constitutes a key result of our research on many-core partitioning in the context of PVT variations.

<table>
<thead>
<tr>
<th>L#</th>
<th>Algorithm 3.2. Initial Voltage Island Placement</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>begin</td>
</tr>
<tr>
<td>2</td>
<td>select placement seed $PS$</td>
</tr>
<tr>
<td>3</td>
<td>for $j = 1$ to $d$, all voltage islands do</td>
</tr>
<tr>
<td>4</td>
<td>select $VI_j$ with highest inter-island traffic with placed islands</td>
</tr>
<tr>
<td>5</td>
<td>select Local Seed $LS_j$, such that distance $</td>
</tr>
<tr>
<td>6</td>
<td>for $k = 1$ to $n_j$, all cores in $VI_j$ do</td>
</tr>
<tr>
<td>7</td>
<td>place Core $C_k$, such that distance $</td>
</tr>
<tr>
<td>8</td>
<td>end for</td>
</tr>
</tbody>
</table>

Figure 3.5: Initial voltage island placement algorithm

![Figure 3.5: Initial voltage island placement algorithm](image)

Figure 3.6: Illustration of the initial voltage island placement with initial seeding.
3.7. Task Scheduling

After the islands are formed as described in the previous sections, the scheduler assigns tasks to cores and routes the connections between the cores (Step2). A task is scheduled to a core within an island with respect to the supply voltage that delivers the task’s required performance deadline. Moreover, the shortest path, or Manhattan distance, from source to destination is used for each route. Unlike a small multi-core design, there might be hundreds of available cores to choose from in a many-core architecture so a new approach is needed here.

We view the scheduling problem as an unconstrained minimum-latency problem where the task-graph has to finish as soon as possible. Therefore, scheduling uses the As-Soon-As-Possible (ASAP) approach. Tasks are scheduled based on the order of execution in the task-graph. A task can be scheduled only if all its predecessors have finished and its data dependencies are routed to its core. When all predecessors for a certain task have finished, this task is marked as ready to be scheduled. A core is assigned a task until it finishes. If a core finishes executing a task, only then it can be re-assigned to another task.

A task priority function, \( Pr_i(T) \), is defined to schedule ready tasks. It is measured by the difference between the task’s Absolute Earliest Start Time (AEST) and the Absolute Latest Start Time (ALST), [16][129], as shown in Equation (3.2).

\[
Pr_i(T_i) = ALST(T_i) - AEST(T_i) \quad (3.2)
\]

This priority function defines the degree of flexibility of a task during scheduling. A lower \( Pr_i(T) \) value means \( T_i \) has very tight timing constraints (and hence has a higher
priority during scheduling). Tasks with same priority are scheduled in a random order. Clearly, tasks on the critical path are scheduled first, as they have the highest priority.

The location of the core has to be chosen in a way to facilitate the task’s communications with its predecessors or successors. There are two types of tasks: a root task and a regular task. A root task in a task-graph is defined as a task with no predecessors and one or more successors. A regular task, on the other hand, is a task that has one or more predecessors and it may or may not have successors. When scheduling a task on a core, this core is selected from the available cores based on its distances to its successors (if it is a root task) or its predecessors (if it is a regular task).

The root task is scheduled on a core depending on its potential data transfer to its successors executed on other cores within the same or other voltage islands. If the root task is communicating with only one island, then it is positioned at the closest distance to that island. However, if the root task is communicating with more than one island, then a communication weighting model is used to select a core positioned at a close proximity to the islands with which it is communicating. The more communication with an island, the closer the task has to be to that island. The weighting model is referred to as the Root Assignment Weight (RAW) and it is shown in Equation (3.3).

\[
RAW_i = \sum_{l=1}^{I} CW_{ii} \times D_{li}; \quad i \in \{1...I\} \quad (3.3)
\]

In Equation (3.3), \( CW_{ii} \) is the amount of communication between root task \( T \), to be executed on core \( C_i \), of \( I \) available cores, and each of \( L \) voltage islands, \( VI_l \), to which it is communicating. The distance \( D_{li} \) is the Euclidean distance between core \( C_i \) and the seed of a given island \( VI_l \). The island seed is the origin of the cloud-based island as discussed previously in Section 3.6.
For a given root task $T$ to be assigned to core $C_i$ at a distance $D_{li}$ from island $VI_l$, $RAW$ is the summation of the multiplication of the communication weight $CW_l$ by the distance $D_{li}$ for all $L$ islands. The core providing the minimum $RAW$ value is assigned the root task. If more than one core have the same minimum value, a core is selected at random.

Figure 3.7 shows an example of how $RAW$ is calculated to find a core $C_i$, shown in dashed lines, to schedule a root task $T$ that belongs to $VI_l$ where there are two available cores: $C_1$ and $C_2$. In this example, the communication of task $T$ with island $VI_2$ has $CW_2=10$ which is larger than the communication with island $VI_3$ which has $CW_3=5$, so $C_i$ has to be closer to $VI_2$. $C_i$ is far from $VI_2$ since $D_{2i} = 5$, and closer to $VI_3$ since $D_{3i}=1$. On the other hand, $C_2$ is closer to $VI_2$ with $D_{22}=2$, and far from $VI_3$ with $D_{32}=4$. Following the calculations in the figure, task $T$ should be executed on core $C_2$.

Figure 3.7: Example of RAW weighting model for root tasks scheduling

A regular task is scheduled on a core positioned at the minimum routing cost to its predecessors’ cores. If it has one predecessor, then it has to be scheduled at the closest available core to its predecessor. However, if it has more than one predecessor, then, in a
similar way as in root tasks, a Task Assignment Weight (TAW) below is used to select a suitable core location.

\[ TAW_i = \sum_{l=1}^{L} RC_{li}; \quad i \in \{1..I\} \quad (3.4) \]

In Equation (3.4), \( RC_{li} \) is the routing cost between core \( C_i \), executing task \( T \), and \( L \) cores executed its predecessors. The routing cost \( RC_{li} \) is the time taken to send all data from a predecessor of task \( T \) to \( C_i \).

At this point, the predecessor tasks are already scheduled, thus their exact core’s location is known. Therefore, the TAW model uses the exact routing cost from the predecessors’ cores. Accordingly, a task \( T \) is scheduled on the core \( C_i \), with minimum \( TAW_i \), which is equal to the summation of the \( RC_{li} \) of all \( L \) predecessor tasks.

The task scheduling algorithm is shown in Figure 3.8. The algorithm starts by defining four lists of tasks: the tasks that are waiting for their predecessors to finish (TaskList), the tasks that are ready to be scheduled (ReadyTasks), the tasks currently running (RunningTasks), and the finished tasks (FinishedTasks). Then, the scheduler starts by calculating the priority of the ready to be scheduled tasks and starts scheduling those of the highest priority. The scheduler selects a task at random from those of equal priority. If the task to be scheduled is a root task, then the RAW model is used and the core that produces the minimum RAW is assigned the task. If the task is a regular task, the TAW model is used instead.

When a task is scheduled, it is added to the RunningTasks list. A counter keeps track of each core executing a task and checks if this task is finished. The finished tasks are added to the FinishedTasks list and removed from the RunningTasks list. When all predecessors for a task are finished, it is then added to the ReadyTasks list and removed
from the TaskList. The algorithm schedules all tasks of the task-graph on the many-core architecture and produces exact timing for all tasks.

Figure 3.8: Task scheduling algorithm
3.8. Voltage Scaling and Selection

The initial voltage scaling, described in Section 3.5, is performed using worst-case estimates for routing delays and PVT. The next step in our flow is to refine the voltage scaling (Step 3), this time using actual routing delays obtained after the scheduling described in Section 3.7. The same voltage scaling algorithm described earlier in Section 3.5 is used again. In effect, this is a second pass of the voltage assignment. As a result, the sizes of the voltage islands may change since new voltages may be assigned to certain cores.

The voltage scaling algorithm can result in a voltage assignment with many different unique voltages. In our implementation, there can be as many as 18 unique voltages. However, having such a large number of unique voltages is not practical. The next step in our flow, (Step 4), is to perform a local refinement of the voltage assignment to each core, in order to reduce the number of unique voltages, but with as small an impact on energy consumption as possible. Our proposed algorithm for doing this is called the Removal Cost Method, and is described in Chapter 4. Our algorithm demonstrates superior runtime and provides better energy results than 1-STEPLA algorithm used in the previous flow.

3.9. Voltage Island Cloud Formation

In the next step (Step 5), the voltage islands are placed using our proposed Voltage Island Cloud (VIC) placement algorithm which is described in detail in Chapter 5. The communication patterns between islands and the islands spatial extent are two factors considered in the placement model. Islands that communicate heavily are placed
next to each other to reduce the routing cost. Furthermore, islands assigned the same voltage, as dictated by the voltage selection step, are place next to each other as well.

An island is defined by its voltage level, number of cores, intra- and inter-island communication patterns, and a chosen origin or seed. The cores of an island are placed based on the distances to origin, or seed, of its island and the level of inter-island communication. Weighting models are utilized to determine the best core placement location. Thus, within an island, cores that communicate heavily with the outside, i.e., extensive inter-island traffic, are placed at the border of that island, closest to the islands with which they communicate. Accordingly, a core is pulled away from its island and towards other islands at distances determined by the proposed weighting model. Cores that show high intra-island communication and low inter-island communication, on the other hand, are placed in the middle of the island, away from the border.

The islands are also formed based on the capabilities of the power grid delivery resources. Traditional fixed power grid systems set more restrictions on voltage islands such that all cores of the same voltage level of different applications still prefer to be grouped together to reduce power grid design complexity [16]. This voltage island style is referred to as Static Voltage Scaling and results from this approach will be shown in this chapter to demonstrate the level of improvements that can be achieved. On the other hand, another approach called Multiple Voltage Scaling allows voltage islands of different applications to be separated and shows improved results. In this approach, each application has its own islands in a separate region of the platform. In Chapter 5, we further elaborate on the difference between the two methods and show the advantage and the disadvantage of each approach [116].
3.10. Routing Optimization using Genetic Algorithms

Finally, a routing optimization technique is needed to keep the routing delays as low as possible while reducing congestion whereby multiple data transfers contend for the same routing resources at the same time (Step 6). We use the routing optimization algorithm described in [16][100].

In this section, we describe the genetic algorithm (GA) used to determine the routing path that will be used for each data transfer operation. Genetic algorithms are an optimization technique that attempts to find solutions to complex optimization problems, as described in Chapter 2.

In our implementation, the GA algorithm consists of a number of iterations through the loop shown in Figure 3.3. Each iteration represents a successive refinement of the solution in search of the best one with the lowest energy. First, a crossover between two genes and a mutation for another one from the population is performed. A gene, in this case, represents one possible route for each of the data transfers in the mapping. Then, another task scheduling operation, previously described in Section 3.7, takes place using the new routing solution resulting from the crossover and mutation step, which may provide additional slack and flexibility to produce a better solution. Of course, the islands are now fixed in shape, size and location. Therefore, task rescheduling can only take place within existing islands. The next step is to reduce the larger number of voltage islands to fewer islands by merging pre-selected islands together, as determined earlier in the voltage selection step. Finally, the total energy consumed in the overall new mapping is determined by the fitness function of the current solution. The GA repeats the same process until a fitness criterion is satisfied.
The routing fabric in our architecture consists of a horizontal and vertical grid of routing tracks. As stated before, there is a router/switch placed at each cross-section in the network. In every clock cycle, the switch routes the data to the proper link of the four links connected to it. The router sends data on a First-Come-First-Serve basis. On each link, there is a buffer in which the data has to wait until the link is cleared. The buffer is assumed to be big enough to store all incoming data. The details of this operation are described below.

### 3.10.1. Routing

We use wormhole-based routing as it is considered to be the most appropriate routing for NoC to reduce buffering resources and communication latencies [130]. The routing optimization is aimed at reducing the waiting time at each buffer in the network.

![Routing path optimization using GA](image)

Figure 3.9: Routing path optimization using GA, example of how GA would solve conflicting routes.

Figure 3.9 shows an example of how GA would resolve routing congestion. In this example, the congestion is heavy at router $C_{10}$ due to three competing paths. Each
routing path $P_s(C_i,C_j)$ from $C_i$ to $C_j$ is represented by a one-dimensional array of binary numbers: a horizontal move is represented using a 0, and a vertical move is represented using a 1.

Since each path is restricted to take a shortest path between source and destination, the length of the array is the sum of the horizontal and vertical distances between source and sink. In the example of Figure 3.9(a), assume the first data coming into router $C_{10}$ is from route $P1$ then $P2$ and $P3$ in the figure. In that case, $P1$ goes through the router in the first clock cycle. Then, $P2$ has to wait 1 cycle and $P3$ two cycles. The use of GA resolves the congestion after several iterations of crossover and mutation, as shown in Figure 3.9(b).

Multiple-sink connections are handled as separate source-sink paths. For a given connection, there are $(n+r)!/(n!\times r!)$ possible routes, where $r$ is the number of horizontal
moves (or 0's) and \( n \) is the number of vertical moves (or 1's). Subsequently, these arrays are used in the genetic algorithm as genes.

At the start of the algorithm, each member of the population (corresponding to the route of one data transfer) is selected randomly from all legal routes. Then, the GA attempts to resolve congestion by choosing non-conflicting paths for the routes.

<table>
<thead>
<tr>
<th>1.</th>
<th>Algorithm 3.4. CrossOver AND Mutation</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Inputs.</strong> Parent genes: ( P_1, P_2, ) and ( P_3 )</td>
<td><strong>Outputs.</strong> Children genes: ( C_1, C_2 ) and ( C_3 )</td>
</tr>
<tr>
<td>1</td>
<td><strong>begin</strong></td>
</tr>
<tr>
<td>2</td>
<td>Cross _Point1 = ceil(length((P_1)/2);</td>
</tr>
<tr>
<td>3</td>
<td>Cross _Point2 = floor(length((P_2)/2);</td>
</tr>
<tr>
<td>4</td>
<td>( HP_1 = P_1(1:Cross _Point1) ) ( P_2(Cross _Point1+1:end); )</td>
</tr>
<tr>
<td>5</td>
<td>( HP_2 = P_2(1:Cross _Point1) ) ( P_1(Cross _Point1+1:end); )</td>
</tr>
<tr>
<td>6</td>
<td><strong>if</strong> ( HP_1 ) or ( HP_2 ) illegal <strong>then</strong></td>
</tr>
<tr>
<td>7</td>
<td>( HP_1 = P_1(1:Cross _Point2) ) ( P_2(Cross _Point2+1:end); )</td>
</tr>
<tr>
<td>8</td>
<td>( HP_2 = P_2(1:Cross _Point2) ) ( P_1(Cross _Point2+1:end); )</td>
</tr>
<tr>
<td>9</td>
<td><strong>if</strong> ( HP_1 ) &amp; ( HP_2 ) illegal <strong>then</strong></td>
</tr>
<tr>
<td>10</td>
<td>( P_1 = HP_1; )</td>
</tr>
<tr>
<td>11</td>
<td>( P_2 = HP_2; )</td>
</tr>
<tr>
<td>12</td>
<td><strong>end</strong></td>
</tr>
<tr>
<td>13</td>
<td><strong>else</strong></td>
</tr>
<tr>
<td>14</td>
<td>( P_1 = HP_1; )</td>
</tr>
<tr>
<td>15</td>
<td>( P_2 = HP_2; )</td>
</tr>
<tr>
<td>16</td>
<td><strong>end</strong></td>
</tr>
<tr>
<td>17</td>
<td>( P_3 = swap ‘1’ and a ‘0’ within ( P_3 ) selected at random; )</td>
</tr>
<tr>
<td>18</td>
<td><strong>end</strong></td>
</tr>
</tbody>
</table>

Figure 3.11: Pseudo-code for crossover and mutation.

The crossover operation, represented in binary numbers, must be selected carefully [16][100]. An arbitrary crossover operation may lead to an illegal routing, i.e., the selected path might not reach the expected destination. For example, consider the two parents: \( P_1: 011100 \) and \( P_2: 100101 \), as shown in Figure 3.10. The crossover at the middle point would result in the following offspring: \( P_1: 011101 \) and \( P_2: 100100 \) which is
an illegal routing path, because they do not lead to the intended destination, as shown in the figure on the right. However, if the crossover happened after the second bit, i.e., $P_1$: 010101 and $P_2$: 101100, the routing is legal. Thus, the number of 1’s has to be preserved in the offspring during crossover to avoid illegal routes. On the other hand, the mutation operation, following the same rule, involves swapping the places of a ‘1’ and a ‘0’ of a parent path [16][100]. The pseudo-code of the cross over and mutation algorithm is shown in Figure 3.11.

The GA loop explores 100 solutions with a population size of 50. The genes are first chosen at random for the first 20 iterations. Then, the GA performs random selections every 5 iterations. After 50 iterations, if the fitness difference between all genes is less than 5% the GA chooses the best and quits.

Before evaluating the total energy consumed in the resulting mapping, island merging is performed to reduce the number of voltages as obtained from RCM, as mentioned previously in Section 3.8.

### 3.10.2. Power and Delay Calculation for the Fitness Function

Previous techniques for system-level modeling of power and delay of a many-core design vary in complexity. In our work, we utilized models that have been shown to provide acceptable estimation and are efficient enough for high-level simulation [16][29][100]. The power consumed by each core is abstracted in three different states: active, idle, and sleep. During the active state, the core’s power consumption is due to dynamic power and during the idle state, the core’s power consumption is due to leakage power, as described in Chapter 2. In sleep mode, the core is in complete shutdown and it
is not consuming any power. The core’s critical path delay is measured by assuming it to be an equivalent inverter chain delay [29].

The other two components considered as source of power dissipation and additional delays are the core’s switch and communication links. Each time the data is sent or received, switches and links along the routing path consume power. They are assumed to consume a constant amount of energy every time they are accessed, which is measured per data unit, e.g., bit or byte. Furthermore, transferring a data unit from one switch to another requires one clock cycle [16][100].

The supply voltage of every switch is assumed to be the same as the core associated with it, and the PVT impact on a core and its switch is also the same. The frequency at which the network-on-chip is operating can be different than the core’s frequency [16][100].

Finally, the power consumption of the voltage level shifters is accounted for whenever a shifter is placed. The total number of shifters in the design is determined by the number of links between islands and it is calculated after the voltage islands are merged.

### 3.10.3. Fitness Function Calculations

The same fitness function used in [16] is used here. The fitness function is an estimation of the total energy consumed, which includes the energy consumed in every core, level-shifter, bus, and switch in the platform. The total core energy is shown in Equation (3.5).

\[
E_{\text{core}} = (P_{\text{dynamic}} + P_{\text{static}}) \times D + E_{\text{shifters}} \quad (3.5)
\]
where $P_{\text{dynamic}}$, $P_{\text{static}}$, and $D$ are the dynamic power, static power and core critical path delay as explained in Chapter 2. $E_{\text{shifters}}$ is the total energy consumed in the level-shifters of that core. The communication energy per bit is given by Equation (3.6).

$$E_{\text{bit}} = n_{\text{hops}} \times e_{S_{\text{bit}}} + (n_{\text{hops}} - 1) \times e_{B_{\text{bit}}} \quad (3.6)$$

where $n_{\text{hops}}$ is the number of utilized switches during transmission (or “hops”), and $e_S$ and $e_B$ are the energy levels consumed at the nominal voltage by the switch and the link, respectively. Thus, the total energy consumed in a routing path is as shown in Equation (3.7).

$$E_{\text{route}} = \sum_{i} \sum_{j} E_{\text{bit}} \quad (3.7)$$

where CW is the communication weight transferred on the route, LW is the link width which is considered 2-byte data wide. Then the fitness function is shown in Equation (3.8)

$$E_{\text{fitness}} = \sum_{r=1}^{R} E_{\text{route}} + \sum_{c=1}^{N} E_{\text{core}} \quad (3.8)$$

The fitness function is the total energy consumed in all routes $R$ and all cores $N$ in the mapped application.

### 3.11. PVT Variations

The models used to estimate process, voltage and temperature variations are considered to have acceptable level of accuracy and efficient enough to handle many-core architectures with hundreds to thousands of cores [50][58][59]. Within-die (WID) process variation impact is generated using VARIUS model [50], as described in Chapter
2. The total variation, which is measured by the total standard deviation divided by the mean $\sigma_{\text{total}}/\mu$, is assumed to be 0.12, 0.16 and 0.18 in our analysis.

The process variation model generates a matrix of core frequencies. This matrix is generated offline, and then used with the optimization platform. As shown in Figure 3.12, the core frequency is used later to calculate the time a task should take when executed on that core.

The temperature variation is estimated using the HotSpot™ CAD tool [58]. The power consumed by each core is input to the tool to compute the current temperature of the core. The algorithm, first, assumes that the temperature = 300K, and $V_t = 400$ mV. Then the correct temperature from the Hotspot™ tool, and the calculated $V_t$ for each core are used in the equations for delay and the static power. The temperature and $V_t$ are re-evaluated after every task-scheduling step.

![Diagram](image)

**Figure 3.12:** Delay and power numbers are updated depending on PVT values.

Finally, voltage variation is computed using a technology-dependent resistive network as discussed in the Chapter 2. The voltage variation affects the supply voltage $V_{\text{dd}}$ due to the workload. Thus, it is calculated at the start of each clock cycle during the scheduling. The delays of all running tasks are re-evaluated to reflect any change in supply voltage.
3.12. Experiments and Results

In this section, we present the results of the application mapping onto the many-core architecture using the proposed energy optimization flow and compare it to the previous flow. The optimization algorithm started with a process variation profile for calibration purposes; this part is assumed to emulate the pre-fabrication scenario. Afterwards, in a post-fabrication scenario, the resulting mapping was enforced on other arbitrary profiles (eight in total) as shown in Figure 3.13. Moreover, the supply voltages of the islands in the many-core platform were raised up until the slowest core in each island meets the timing requirements.

![Diagram showing Pre-Fabrication and Post-Fabrication mapping optimization](image)

Figure 3.13: Pre- and Post-Fabrication experimental setup

<table>
<thead>
<tr>
<th>Number of Test Cases</th>
<th>Number of Taskgraphs per Test Case</th>
<th>Number of Tasks</th>
<th>Parallelism Range</th>
<th>Critical Path Length Range (Time Units)</th>
<th>Number of Communication Edges</th>
</tr>
</thead>
<tbody>
<tr>
<td>30 randomly generated</td>
<td>4 TGs grouped together</td>
<td>950 Tasks</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>500 tasks</td>
<td>16 up to 55</td>
<td>67 up to 315</td>
<td>504 up to 4504 edges</td>
</tr>
<tr>
<td></td>
<td></td>
<td>300 tasks</td>
<td>11 up to 47</td>
<td>45 up to 264</td>
<td>277 up to 2901 edges</td>
</tr>
<tr>
<td></td>
<td></td>
<td>100 tasks</td>
<td>8 up to 20</td>
<td>29 up to 122</td>
<td>93 up to 915 edges</td>
</tr>
<tr>
<td></td>
<td></td>
<td>50 tasks</td>
<td>7 up to 12</td>
<td>24 up to 70</td>
<td>46 up to 941 edges</td>
</tr>
<tr>
<td>Sparse Matrix Solver (SMS)</td>
<td>8 TGs grouped together</td>
<td>768 tasks</td>
<td>96 tasks</td>
<td>15.8</td>
<td>122</td>
</tr>
<tr>
<td>Robot Control Program (RCP)</td>
<td>8 TGs grouped together</td>
<td>704 tasks</td>
<td>88 tasks</td>
<td>4.36</td>
<td>569</td>
</tr>
<tr>
<td>SPEC95,fpmmm</td>
<td>2 TGs grouped together</td>
<td>668 tasks</td>
<td>334 tasks</td>
<td>6.7</td>
<td>1062</td>
</tr>
</tbody>
</table>

Table 3.1: Test Cases used in the Experiments [131].
We applied the two energy optimization flows to 33 benchmarks of [131], 30 of which are generated at random and 3 real applications: a Sparse Matrix Solver (SMS), a Robot Control program (RCP), and a SPEC95 (fppp-kernel). The number of tasks in each test case, parallelism, and communication edges are shown in Table 3.1. The parallelism is the total computational time of all tasks divided by the computational time of the tasks on the critical path.

Figure 3.14: Energy savings improvement compared to previous technique, (a) geometric average of 30 randomly generated TGs, (b) SMS (c) RCP (d) SPEC95, fppp-kernel.

The mapping was performed first on PV0 and then tested on 8 other variation profiles (PV1-PV8). The number of voltages used initially is 18 voltages, 0.6V to 1.4V in 0.05V increments. A list of the number of cores considered in the experiments is as follows: 400, 625, 900, 1225, and 1600. The shown results are the achieved energy saving as compared to previous methods of [16], where As-Soon-As-Possible algorithm,
1-STEPLA for voltage selection, and rectangular voltage islands are utilized to produce the energy-aware mapping.

Figure 3.14 shows the energy savings improvements using our proposed energy flow as compared to the prior approach. Figure 3.14(a) shows the geometric mean of the 30 randomly generated benchmarks. The achieved energy saving improvement is between 7-23% for the randomly generated task-graphs. As shown in Figure 3.14(b) through (d), all cases showed improvement. The energy saving improvement in the case of Sparse Matrix Solver (SMS), Robot Control Program (RCP), and SPEC95 (fpppp-kernel) are 5-25%, 7-9%, 10-15%, respectively.

![Graph](image)

Figure 3.15: Energy-Delay-Product savings improvements, (a) geometric average of 30 randomly generated TGs (b) SMS (c) RCP, and (d) SPEC95, fpppp-kernel.

The energy saving portions due to the new task scheduling and the voltage island formation, and due to the new voltage selection algorithm, RCM, are discussed later in Chapter 5.
Figure 3.15(a) through (d) shows the Energy-Delay-Product savings improvement for the benchmarks. The saving improvements are approximately 12-37%, 12-35%, 8-12%, and 16-20% for the randomly generated benchmarks, Sparse Matrix Solver, Robot Control and SPEC95, respectively.

3.13. Chapter Summary

An improved energy optimization flow has been proposed to generate energy-aware-mapped applications onto many-core architectures considering process, voltage, and temperature variations. A voltage islands formation technique is utilized to save on power while delivering the required performance. The flow involves voltage scaling, island creation, and voltage selection. A novel task scheduling technique is proposed to assign tasks onto cores, taking into account communication patterns of tasks.

The energy saving for the proposed energy flow shows geometric average for 33 benchmarks of 8-25% improvement as compared to previous methods. Moreover, the energy-delay-product shows 10-37% improvement of the proposed flow as compared to previous methods. In the upcoming chapters, we elaborate in further details on the novel techniques used to solve the voltage selection problem and the voltage island formation.
Chapter 4: Voltage Selection Problem

4.1. Chapter Overview

The previous chapter described the improvements to an energy-aware flow for mapping an application to a many-core architecture using a multi-Vdd island-based approach. It further discussed several of the energy optimization steps, Steps 1 through 5 of Figure 3.3, that showed voltage island formation using supply voltage assignment to cores based on workload requirements. Initial results were presented to demonstrate that the new flow produces better results relative to the previous flow.

In this chapter, we focus on the details of Step 4 of the flow. Voltage selection can be divided into optimization methods for the continuous versus discrete cases. Continuous voltage selection allows the voltage of the tasks to be adaptively selected from a continuous interval. Discrete voltage selection requires the selection from a finite set of predetermined voltage levels [31][128][132][133].

A novel approach for the discrete problem is presented to reduce the number of unique voltage levels in the design. Theoretically, each core could be assigned its own voltage level. However, a large number of voltage levels leads to significant overhead and complexity in the power distribution network. Thus, it is important to minimize the number of discrete voltage levels in the design. This selection process is referred to as the Voltage Selection Problem (VSP) [29][31][34][128]. Our new approach is very efficient.

2 This chapter is based on the work published in [34].
in handling architectures with hundreds to thousands of cores and provides near-optimal results for the examined benchmarks.

Section 4.2 illustrates the problem using a motivating example. Comparisons between the voltage selection problem and similar problems, such as well-known knapsack and partitioning problems are presented. Section 4.3 then provides a problem formulation for voltage selection. Section 4.4 describes the proposed algorithm, which we refer to as the Removal Cost Method (RCM). The algorithm is first presented along with an illustrative example. Then the computational complexity of the proposed algorithm is evaluated and compared to previous methods. Section 4.5 describes the experimental platform within which the proposed algorithm is evaluated. Section 4.6 quantifies the energy savings obtained using this algorithm as well as the algorithm’s run-time relative to previous algorithms.

4.2. Motivating Example

In this section, a voltage selection example is presented and then solved using different methods. The example is shown in Figure 4.1. The task-graph, shown in Figure 4.1(a), consists of eleven tasks and is to be scheduled on the multi-core architecture with nine cores (shown in Figure 4.1(b)). Each task can be assigned a physical core and scheduled based on the task’s criticality. In this example, the critical path of the task-graph is through tasks $T_4$, $T_9$, and $T_{10}$; the length of this path is 22 time units. We assume that all tasks on the critical path are placed on the same core to avoid extra routing costs.

---

3 In practice, the communication cost between cores should also be considered during scheduling but, for simplicity, we ignore it in this example.
In addition, we assume for the purposes of this example only that the other tasks are assigned to physical cores such that the routing delays between tasks are as shown in Figure 4.1(d). Given the delay of each task, as well as the routing delay between each pair of tasks, the slack time of each task that is not on the critical path can then be determined by comparing its finish time and its deadline.

![Taskgraph with Computational Times](image)

![Multi-core with 9 cores](image)

![Scheduling with Slack](image)

![Routing cost](image)

Figure 4.1: A task-graph scheduled on 9 cores showing the task’s slack times.

Given the schedule shown in Figure 4.1(c), it is apparent that the tasks, \(T_0, T_1, T_3, T_5, T_6, T_7, \) and \(T_8,\) are less critical than the others. We can take advantage of this by lowering the voltage assigned to those cores, thereby reducing the overall power. In doing so, however, we must ensure that the tasks finish and data is available before the
start of $T_g$ in the original schedule. In this example, we can slow down tasks $T_0$, $T_1$, $T_3$, and $T_7$, by 3 units, tasks $T_5$, $T_8$ by 2 units and $T_6$ by 1 unit. This translates into the voltage assignments and schedule shown in Figure 4.2.

Once the tasks have been scheduled in this way, a voltage selection algorithm can begin to reduce the number of voltages to an acceptable value. Suppose, in this example, physical limitations dictate that our architecture can be constructed with at most two distinct voltage levels. This would mean that we need to select two voltage levels from the set \{1.0, 1.1, 1.2, 1.4\} such that the total energy is minimized and the critical path is not increased.

One way to represent the information needed by a voltage selection algorithm is to construct an energy cost matrix. The energy cost matrix tabulates the energy consumed by each core for all possible supply voltages. The energy cost matrix for the example shown in Table 4.1. Each row in the table corresponds to a different core in the architecture, and each column corresponds to a potential voltage assignment. Each entry indicates the energy that would be consumed by that core if it is assigned that voltage. If
a particular voltage would slow the core down so much that the critical path would increase, then that entry is marked with an infinity sign to indicate that it is illegal.

Table 4.1: Energy Cost Matrix.

<table>
<thead>
<tr>
<th>Cores</th>
<th>$VI_1$=1.0V</th>
<th>$VI_2$=1.1V</th>
<th>$VI_3$=1.2V</th>
<th>$VI_4$=1.4V</th>
</tr>
</thead>
<tbody>
<tr>
<td>$C_1$</td>
<td>8.1</td>
<td>8.3</td>
<td>8.5</td>
<td>10.5</td>
</tr>
<tr>
<td>$C_2$</td>
<td>4.3</td>
<td>4.7</td>
<td>5.6</td>
<td>8</td>
</tr>
<tr>
<td>$C_3$</td>
<td>6.3</td>
<td>7</td>
<td>7.5</td>
<td>10.3</td>
</tr>
<tr>
<td>$C_4$</td>
<td>$\infty$</td>
<td>$\infty$</td>
<td>2.4</td>
<td>3</td>
</tr>
<tr>
<td>$C_5$</td>
<td>3.5</td>
<td>3.8</td>
<td>4.1</td>
<td>5.5</td>
</tr>
<tr>
<td>$C_6$</td>
<td>$\infty$</td>
<td>2.8</td>
<td>3.2</td>
<td>4.6</td>
</tr>
<tr>
<td>$C_7$</td>
<td>$\infty$</td>
<td>$\infty$</td>
<td>$\infty$</td>
<td>8</td>
</tr>
<tr>
<td>$C_8$</td>
<td>$\infty$</td>
<td>$\infty$</td>
<td>$\infty$</td>
<td>15</td>
</tr>
<tr>
<td>$C_9$</td>
<td>$\infty$</td>
<td>3.2</td>
<td>3.9</td>
<td>5.3</td>
</tr>
</tbody>
</table>

Figure 4.3: Four Voltage Islands to be merged to two islands.

In the table, the shaded cells represent the energy consumed by the cores operating at the minimum voltage that delivers the required performance; hence, the minimum total energy is obtained by adding up the shaded cells to produce 53.6mJ. However, that solution would require four distinct voltage levels. The goal of the voltage selection algorithm in this example is to select 2 voltages out of 4 voltages such that the overall energy is as small as possible, effectively as close to 53.6mJ as possible.
4.2.1. One Step Look Ahead Algorithm (1-STEPLA)

We first describe the 1-STEPLA algorithm proposed by [16] for solving the voltage selection problem. The algorithm uses a Trail-and-Error approach by merging neighbouring islands and measuring the increase in total energy. Consequently, the merging operation that results in the minimum energy increase is applied. Then, the algorithm proceeds to merge the next pair of islands until the target number of voltages is reached.

The algorithm starts by identifying neighbouring islands that can be merged. In our example, the possibilities are: \{VI_1, VI_2\}, \{VI_1, VI_3\}, \{VI_1, VI_4\}, \{VI_2, VI_4\}, or \{VI_3, VI_4\} as shown in Figure 4.3. Each of these is attempted, and for each, another merging is attempted. Once all combinations of two successive merges are attempted, the result that gives the minimum total energy is selected. In this example, the islands \{VI_3, VI_4\} are merged together first and then \{VI_1, VI_3\}. Thus, the final supply voltages are 1.1V and 1.4V and the total increase in energy then is 2.2mJ, i.e., a 4% energy increase.

4.2.2. Dynamic programming (DP)

Dynamic programming is another approach to solve the voltage selection problem [29]. Again, combinations of different merging possibilities are explored in an attempt to find the combination that results in the minimum energy. Table 4.2 graphically shows the voltage selection performed using dynamic programming on our example. The algorithm starts first by allowing only one supply voltage; this is shown in the first row of the table. When there is only one unique voltage level permitted, it has to be the highest, i.e., 1.4V; otherwise the timing constraints are violated as indicated in the cost matrix, Table 4.1.
Table 4.2: Voltage selection using dynamic programming.

<table>
<thead>
<tr>
<th>Number of Allowed Voltages</th>
<th>Available Voltages</th>
<th>1.4V</th>
<th>1.2V</th>
<th>1.1V</th>
<th>1.0V</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 voltage level</td>
<td></td>
<td>(1.4V)</td>
<td>(1.4V)</td>
<td>(1.4V)</td>
<td>(1.4V)</td>
</tr>
<tr>
<td></td>
<td></td>
<td>E = 70.2 mJ</td>
<td>E = 70.2 mJ</td>
<td>E = 70.2 mJ</td>
<td>E = 70.2 mJ</td>
</tr>
<tr>
<td>2 voltage levels</td>
<td></td>
<td>(1.4V)</td>
<td>(1.4V, 1.2V)</td>
<td>(1.4V, 1.1V)</td>
<td>(1.4V, 1.1V)</td>
</tr>
<tr>
<td></td>
<td></td>
<td>E = 70.2 mJ</td>
<td>E = 58.2 mJ</td>
<td>E = 55.8 mJ</td>
<td>E = 55.8 mJ</td>
</tr>
<tr>
<td>3 voltage levels</td>
<td></td>
<td>(1.4V)</td>
<td>(1.4V, 1.2V)</td>
<td>(1.4V, 1.2V, 1.1V)</td>
<td>(1.4V, 1.1V, 1.0V)</td>
</tr>
<tr>
<td></td>
<td></td>
<td>E = 70.2 mJ</td>
<td>E = 58.2 mJ</td>
<td>E = 55.2 mJ</td>
<td>E = 54.2 mJ</td>
</tr>
</tbody>
</table>

The algorithm then permits the selection of two voltage levels; this is illustrated in the second row of Table 4.2. The selection process chooses 2 voltages of the available 4 voltages which are at first 1.4V only, then 1.4V and 1.2V, then 1.4V, 1.2V, and 1.1V, and so on. In each case, the total energy is calculated and the set of voltages that results in the minimum total energy is selected. For example, in the third column, second row, of Table 4.2, the available voltages are 1.4V, 1.2V, and 1.1V; hence, the possible selections of two voltages are: (1.4V, 1.2V), and (1.4V, 1.1V) which produces total energy of 58.2mJ, and 55.8mJ, respectively. Thus, the set (1.4V, 1.1V) is selected with a total energy equal to 55.8mJ. In the next iteration, in the fourth column, the available voltages are 1.4V, 1.2V, 1.1V, and 1.0V, so the algorithm calculates the energy considering all combinations of voltages again. The total energy calculated for the set (1.4V, 1.0V) is 58.1mJ, and since it is higher than 55.8mJ obtained by (1.4V, 1.1V), it is not considered as a possible solution.

The algorithm continues in the same way for sets of three voltages as shown in the third row. The selected voltages are 1.4V and 1.1V in case of 2 allowed voltages with
4% total increase in energy, as before, and a total of 55.8\text{mJ} consumed energy. The voltages 1.4V, 1.1V, and 1.0V with 54.2\text{mJ} total energy are selected for 3 allowed voltages.

As demonstrated in this example, both methods use approaches that require a considerable amount of computational time. Thus, in the remainder of this chapter, we propose a faster method that can deliver better or similar energy results.

### 4.2.3. Knapsack Problem

The voltage selection problem is a combinatorial optimization problem. The knapsack problem is often used by designers to represent the voltage selection problem due to their similarities so that the same methods can be used to solve it [29][134]. The knapsack problem is defined as follows:

**Knapsack Problem:** Given a set of items where each one is assigned a cost and a weight values. Determine the items that can be put in a knapsack such that the total weight is less than a weight constraint and the total cost is minimized.

The knapsack problem is an NP-hard problem. Many algorithms are used to solve it in a pseudo-polynomial time; however, none can be both optimal and fast at the same time for all cases [133].

While the knapsack problem may seem to be similar to the voltage selection problem, it is not exactly the same. First, all items have the same weight in the voltage selection problem. The cost of each voltage may differ but all voltages have the same weight value. For example, while selecting two voltages out of four, the total weight, \( W \), is equal to two, where the voltages can be any two of the four voltages such that the total
energy cost is minimized. Eliminating the weighting of the items makes the optimization problem target the cost minimization only.

Another important aspect in voltage selection is that the energy cost of each of the voltages might change when any of the voltages is excluded from the selection. For instance, consider the example shown in Table 4.1. If $V_1$ is removed, all cores assigned $V_1$ are now assigned $V_2$. Thus, the energy cost of removing $V_2$ will increase. Accordingly, the costs of the supply voltages have to be updated whenever a voltage is excluded during optimization. Based on this insight, the voltage selection problem can be transformed into a simpler optimization problem with one parameter.

### 4.2.4. VSP as a Partitioning Problem

The voltage selection problem has also been considered as a partitioning problem [132]. In a multi-core design, the cores have to be partitioned into groups, where each group is given a specific supply voltage. Considering the example shown in Table 4.1 and Figure 4.3, the cores have to be assigned one of the two voltages out of a four possible voltages. Given that the highest voltage, i.e., 1.4V, has to be included to meet the performance requirement, there is only one other voltage that needs to be selected. Thus, the possible combinations are: $\{V_1, V_4\}$, $\{V_2, V_4\}$, $\{V_3, V_4\}$. For each possible voltage combination, each core is assigned a voltage that produces the minimum energy and delivers the required performance. For example, in the case of $\{V_1, V_4\}$, cores: $C_1$, $C_2$, $C_3$, and $C_5$ are assigned $V_1$ and the rest of the cores are assigned $V_4$. The total energy due to this combination is computed and compared to other combinations and then the one with the minimum energy is selected. Here, the problem complexity increases as the number of cores in the design increases.
4.3. Problem Formulation

In this section, we more precisely state our problem definition. When an algorithm is mapped to a many-core, critical path analysis can be performed to determine the \textit{slack} time of each task. Once these tasks are mapped to physical cores, the voltage levels of each core can be adjusted to be as small as possible such that all tasks assigned to that core meet their timing constraints. If the voltage of each core is adjusted independently, there will be $m$ distinct voltage levels such that $m \leq N$ where $N$ is the number of cores. If $m$ is large, it is not practical to implement so many distinct voltages. In that case, a subset of the $m$ voltages must be selected, such that each core can be powered by one of the selected voltages. Since there are many possible subsets, the Voltage Selection Problem is to select the subset that results in the minimum overall energy with a desired small number of voltage levels across the entire many-core design.

More precisely, the VSP can be formalized as follows:

\textbf{Input:}

a) A set of possible voltage levels: $X=\{V_1, V_2, \ldots, V_m\}$

b) An initial assignment of voltages to cores: $v_0(i) \in X$ with $1 \leq i \leq N$ where $N$ is the number of physical cores.

c) A target number of voltage levels $d$.

\textbf{Find:} A final assignment of voltages to cores, $v_f(i) \in X$, $1 \leq i \leq N$

\textbf{Such that:}

a) The delay constraint is no worse than with the initial voltage assignment

b) The overall power is minimized.

c) The number of unique voltage levels is $d$
The delay constraint can be ensured by enforcing $v_f(i) \geq v_0(i), \ 1 \leq i \leq N$.

4.4. Removal Cost Method (RCM)

The new RCM algorithm requires an energy cost matrix $E$ with $N$ rows and $m$ columns, as shown earlier in Table 4.1. Element $E_{i,j}$ is the amount of energy required by core $C_i$ if implemented using voltage level $V_j$. In our implementation, we compute this by summing the energy dissipated by the cores, using the power-delay product ($PxD$), and the energy dissipated in the core communication links between cores $C_i$ and $C_c$, as shown in Equation (4.1). For an $N_{bit}$ transmission, the energy consumed per bit across the Network-on-Chip during data routing, $e_{bit}$, is computed as shown in Equation (4.2), such that $n_{hops}$ is the number of utilized switches during transmission (or “hops”), while $e_{S_{bit}}$ and $e_{L_{bit}}$ are the energies consumed by the switch and the link, respectively [16].

$$E_{i,j} = \sum_{1}^{N_{bit}} e_{bit} + D_{core} \times P_{core} \quad (4.1)$$

$$e_{bit} = n_{hops} \times e_{S_{bit}} + (n_{hops} - 1) \times e_{L_{bit}} \quad (4.2)$$

The power dissipated in the core is computed using the standard dynamic and static power equations as shown in Equation (4.3).

$$P_{core} = C_{eff} V_{dd}^2 f + I_o e^{-\frac{qV_t}{nkT}} V_{dd} \quad (4.3)$$

where $C_{eff}$ is the effective switching capacitance, $V_{dd}$ is the selected operating supply voltage, $f$ is the frequency at which the core is running, $T$ is the temperature, $k$ is the Boltzmann constant, and $I_o$ and $n$ are technology-dependent constants. The delay of the core is assumed to be an inverter chain delay and is shown in Equation (4.4).
\[ D_{\text{core}} = \frac{C_0 V_{dd} / 2}{K \times (V_{dd} - V_i)^\alpha} \]  

(4.4)

where \( C_0 \) is the switching capacitance of the critical path, \( K \) and \( \alpha \) are technology-dependent constants.

### 4.4.1. RCM Algorithm

Figure 4.4 shows the high-level pseudo-code for the RCM algorithm. The algorithm maintains an ordered list of selected voltage levels, \( X \), and a voltage assignment for each core, \( Y \). Initially, \( Y = \{v_0(i), 1 \leq i \leq N\} \), and \( X \) is the set of unique voltages in \( Y \). In each iteration, one element is greedily removed from \( X \), and \( Y \) is updated appropriately. The cost of removing this element is computed, hence the name removal cost method. The algorithm ends when there are \( d \) elements in \( X \).

During each iteration, the voltage to be removed from \( X \) is selected using a vector \( R \) which contains the removal cost for each voltage. Each element of \( R \) corresponds to a voltage level in \( X \) and is equal to the increase in energy that would occur if that voltage level was removed, and all cores currently assigned that voltage level were instead assigned the next highest voltage level. More precisely, the removal cost \( R \) is calculated using the expression shown in Equation (4.5).

\[ R_j = \sum_{\text{all } n \text{ cores such that } v_n = V_j; V_j \in X} (E_{n,j+1} - E_{n,j}) \]  

(4.5)
**Figure 4.4: RCM Algorithm.**

As an example of our method, consider the cost matrix shown in Table 4.1 again.

Assume that we require only 2 voltage levels, i.e., \( d = 2 \). Initially, \( X = \{V_1, V_2, V_3, V_4\} \) and \( Y = \{v_0(1) = V_1, \ (v_0(2) = V_1, \ (v_0(3) = V_1, \ (v_0(4) = V_2, \ (v_0(5) = V_1, \ (v_0(6) = V_2, \ (v_0(7) = V_3, \ (v_0(8) = V_4)\}. \) The initial cost, by summing the shaded regions is 53.6mJ. The removal cost of \( V_1 \) is:

\[
R_1 = (8.3 - 8.1) + (4.7 - 4.3) + (7 - 6.3) + (3.8 - 3.5) = 1.6.
\]

Similarly, the removal cost of \( V_2 \) is \( R_2 = 1.1 \) and the removal cost of \( V_3 \) is \( R_3 = 0.6 \). Thus, the vector \( R \) is \( R = \{1.6, 1.1, 0.6, \infty\} \). Based on the removal vector, we can see that \( V_3 \) should be removed because it has the minimum removal cost \( R_3 = 0.6 \).
Table 4.3: Voltage selection using RCM.

<table>
<thead>
<tr>
<th>Available Voltages</th>
<th>1st iteration</th>
<th>2nd iteration</th>
<th>3rd iteration</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Removal Cost</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>$R_1$</td>
<td>$R_2$</td>
<td>$R_3$</td>
</tr>
<tr>
<td>$V_1=1.0V$</td>
<td>1.6</td>
<td>1.6</td>
<td>-</td>
</tr>
<tr>
<td>$V_2=1.1V$</td>
<td>1.1</td>
<td>3.9</td>
<td>3.9</td>
</tr>
<tr>
<td>$V_3=1.2V$</td>
<td>0.6</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>$V_4=1.4V$</td>
<td>$\infty$</td>
<td>$\infty$</td>
<td>$\infty$</td>
</tr>
</tbody>
</table>

Thus $X=\{V_1, V_2, V_4\}$, $Y=\{(v(1)=V_1), (v(2)=V_1), (v(3)=V_1), (v(4)=V_2), (v(5)=V_2), (v(6)=V_2), (v(7)=V_4), (v(8)=V_4)\}$ and $R=\{1.6, 1.1, \infty\}$. After removing $V_3$, only $R_2$ needs to be updated since $R_4=\infty$. The same process will repeat again for iterations 2 and 3 as shown in Table 4.3. The final result for this example is that $V_2$ and $V_4$ will be selected. The final $Y$ vector would be $Y=\{(v(1)=V_2), (v(2)=V_2), (v(3)=V_2), (v(4)=V_2), (v(5)=V_2), (v(6)=V_2), (v(7)=V_4), (v(8)=V_4)\}$ and $X=\{V_2, V_4\}$. The total energy increase after selecting two voltages is $R_1+R_2 = 2.2mJ$, i.e., 4%, and the total energy is 55.8mJ.

4.4.2. Computational Complexity of RCM

As shown in Figure 4.4, the initial values for $R_j$ require $O(m+N)$ runtime to compute (lines 1-6). Then, the while loop requires $O(m – d)$, since the loop starts with $m$ voltages until $d$ voltages are reached. Finding the supply level, $V_k$, with minimum $R_k$ requires $O(m)$, which is then removed. The cores previously assigned to $V_k$ is now assigned to a higher supply voltage, $V_{k+1}$, which requires $O(N)$ operations. Finally,
updating $R_{j+1}$ and $R_{j-1}$, line 16, each iteration requires $O(N)$ operations to finish, since if voltage $V_j$ is removed, only $R_{j-1}$ and $R_{j+1}$ are updated.

The overall complexity of the algorithm is $O(m+N+(m-d)m+2(m-d)N)$, which can be simplified to $O(mN)$. The best previous heuristic to solve the VSP has at least $O(m^2d^2N)$ complexity with near optimal solutions [16][29][128]. We will also show, later in Section 4.6, that, in practice, our algorithm is much faster, and computes solutions which are also near optimal for the considered cases.

4.5. Experimental Platform

The energy optimization flow discussed in Chapter 3 is used to evaluate the different voltage selection techniques. The energy flow is repeated for convenience in Figure 4.5. First, different applications are allocated to a portion of the available resources. Tasks are then scheduled and assigned to cores using an As-Soon-As-Possible (ASAP) scheduling algorithm. The data routing at this point is selected at random.

The slack time of each task then is computed and used to scale the voltage of each core as low as possible such that all tasks can meet their deadlines. Voltage islands are then formed by grouping cores with the same voltage. The routing traffic is then optimized using a genetic algorithm. Voltage selection using the RCM algorithm is then performed, and islands that are deemed to have the same voltage are merged.

To validate our proposed RCM, we compared it to dynamic programming from [29], as it is the best previous method, and one-step look ahead, 1-STEPLA, from [16]. We evaluated 30 test cases for each many-core design. In every test case, a randomly generated task-graph (with a total of 950 tasks, and its associated parallelism ratio, critical path length and number of communication edges) was mapped onto a many-core
architecture with N cores [131]. The parallelism ratio is defined by the total computation time of all tasks divided by the computational time of the critical path. The characteristics of the task-graphs used in the experiments are shown in Table 4.4. The 30 test cases are also presented in the chart of Figure 4.5 in terms of their communication edges and parallelism.

![Energy Optimization Flow Diagram](image)

Figure 4.5: Energy Optimization Flow.

Experiments on architectures with 400, 625, 900, 1225 and 1600 cores were performed. It was assumed that two, three and four voltage levels were to be selected from the range 0.6V to 1.4V in 0.05V increments, i.e., \( m=18 \) and \( d=2, 3, \text{ and } 4 \). In our power computations, we assumed that any unused cores were switched off completely. The maximum speed at which the processor can run was assumed to be 300MHz with the
network running at 600MHz. The nominal dynamic power of a core was estimated to be 0.45mW/MHz [135].

Table 4.4: Characteristics of the Benchmarks used [131]

<table>
<thead>
<tr>
<th>Number of Tasks</th>
<th>Parallelism Range</th>
<th>Critical Path Length Range (Time Units)</th>
<th>Number of Communication Edges</th>
</tr>
</thead>
<tbody>
<tr>
<td>500 tasks</td>
<td>16 up to 55</td>
<td>67 up to 315</td>
<td>504 up to 4504 edges</td>
</tr>
<tr>
<td>300 tasks</td>
<td>11 up to 47</td>
<td>45 up to 264</td>
<td>277 up to 2901 edges</td>
</tr>
<tr>
<td>100 tasks</td>
<td>8 up to 20</td>
<td>29 up to 122</td>
<td>93 up to 915 edges</td>
</tr>
<tr>
<td>50 tasks</td>
<td>7 up to 12</td>
<td>24 up to 70</td>
<td>46 up to 941 edges</td>
</tr>
</tbody>
</table>

Figure 4.6: Application parallelism versus Communication.

4.6. Results

In this section, we discuss the voltage selection energy results considering 2, 3, and 4 voltage levels. We also show the impact of applications with heavy
communication, hence lower slack times, on the voltage selection problem. We finally compare the energy savings and runtime results to previous work.

Figure 4.7 shows the voltage distribution of core voltages before performing voltage selection for selected benchmarks mapped onto the many-core platform with 900 cores. Figure 4.7(a) shows the case for a subset of the benchmarks with a parallelism ratio between 50 and 100 and the number of communication edges between 1000 and 3100. On the other hand, Figure 4.7(b) shows another subset with parallelism ratio between 30 and 45, and the number of communication edges between 4000 and 8000.

![Diagram](image.png)

Figure 4.7: Cores’ voltage distribution before VSP for selected benchmarks with (a) high parallelism and low number of communication edges, and (b) low parallelism and high number of communication edges.

From the figure, a high degree of parallelism and low communication between tasks implies more available slack time; thus, the distribution appears to be more evenly spaced among all available voltages, although there is a double-humped characteristic due to the distribution of slack times in these applications. In any case, voltage selection could be omitted here and evenly spaced voltages can be selected from all possible voltages. Low parallelism and high communication, on the other hand, are indications of
low available slack times. Thus, distributions shift towards higher voltages since most tasks have very little slack time and need to be completed as soon as possible.

![Figure 4.8](image)

Figure 4.8: Energy increase for 25 benchmarks (RCM versus pre-selected voltages), due to selecting (a) two (b) three and (c) four out of 18 voltages.

As a result of the previous observations, we try to compare the energy saving in two cases: 1) performing voltage selection using RCM and 2) selecting evenly spaced voltages without going through the voltage selection optimization. We do that in the case of the selection of 2, 3, and 4 voltages out of 18 voltages.

Figure 4.8(a), (b), and (c) show the percentage energy increase of 25 benchmarks in the case of 2, 3, and 4 selected out of 18 original voltages, respectively. As shown in
Figure 4.8(a), the energy increase difference, between RCM and pre-selected supply voltages (2 out of 18 voltages) is considerably large, approximately 3-13%. Figure 4.8(b) shows a moderate difference between the two methods (3 out of 18 voltages), approximately 0-4%, of the percentage energy increase. Thus, it is recommended to perform voltage selection optimization for these two cases.

Figure 4.8(c), on the other hand (4 out of 18 voltages), shows minimal energy difference, between RCM and pre-selected Vdd values, approximately 0-2%. With such a small improvement, the voltage selection optimization could be omitted from the optimization flow if runtime is an issue. It can be simply replaced by 4 uniformly distributed voltages out of the 18 voltage levels without a significant penalty.

Table 4.5: Percentage energy increase (geometric mean of 30 benchmarks) when 2, 3, and 4 are selected out of 18 Vdds using RCM

<table>
<thead>
<tr>
<th>Number of Cores</th>
<th>Number of Vdds</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>2 Vdds</td>
</tr>
<tr>
<td>400</td>
<td>10.4%</td>
</tr>
<tr>
<td>625</td>
<td>12.38%</td>
</tr>
<tr>
<td>900</td>
<td>12.79%</td>
</tr>
<tr>
<td>1225</td>
<td>13.32%</td>
</tr>
<tr>
<td>1600</td>
<td>13.74%</td>
</tr>
</tbody>
</table>

Table 4.5 shows the energy increase of the geometric mean of all benchmarks averaged using our proposed RCM algorithm when 2, 3 and 4 voltages are selected out of
18 voltages. The reference value of energy is when all 18 voltages are used. The energy increase from this value due to selecting 3 and 4 out of 18 are 6-7% and 3-4%, respectively. The energy increase when 2 are selected out of 18 goes up to 12-13%.

Table 4.6 shows the energy increase of the geometric mean of all benchmarks averaged when selecting 4 out of 18 for each of the three algorithms. As shown, the 1-STEPLA algorithm is showing highest energy increase, approximately 9%, while the dynamic programming and the proposed RCM show lower energy increase, approximately 4%.

Table 4.6: Percentage Energy Increase when selecting 4 out of 18 voltages (geometric mean of 30 benchmarks) for different methods

<table>
<thead>
<tr>
<th>Number of Cores</th>
<th>Voltage Selection Method</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>1-STEPLA</td>
</tr>
<tr>
<td>400</td>
<td>6.41%</td>
</tr>
<tr>
<td>625</td>
<td>8.01%</td>
</tr>
<tr>
<td>900</td>
<td>8.78%</td>
</tr>
<tr>
<td>1225</td>
<td>9.42%</td>
</tr>
<tr>
<td>1600</td>
<td>9.94%</td>
</tr>
</tbody>
</table>

Figure 4.9 shows the normalized runtime for RCM and DP. The 1-STEPLA method is more than 20X slower, and not shown in the figure, because it produces suboptimal results and it follows a different strategy than RCM and DP. RCM and DP are performed before island placement. On the other hand, the 1-STEPLA algorithm
attempts to merge neighbouring islands after placing them. Unlike small multi-core systems, islands may contain hundreds of cores, hence have larger borders, in a many-core architecture. Therefore, checking islands that share a common border can be time consuming, so a direct comparison with the other two methods cannot be carried out.

The RCM and DP algorithms were implemented using MATLAB© and executed on Linux Debian with a 2.8GHz ZION Intel© processor and 3G of RAM. As shown, our proposed RCM is 10 times faster than the DP algorithm. Given the significant advantage of RCM in terms of speed and its superiority with respect to energy savings over 1-STEPLA, we conclude that RCM is preferable to the other two algorithms.

4.7. Chapter Summary

In this chapter, we presented a novel linear time algorithm, in terms of number of islands, to solve the voltage selection problem. The voltage selection problem is an important step of the overall energy-aware flow that maps applications onto many-core architectures with hundreds to thousands of cores. We compared both the energy results
and the run-time of our algorithm to previous algorithms. We found that we get similar energy results as compared to dynamic programming, but achieve them with an order of magnitude improvement in run-time.
Chapter 5: Voltage Island Formation

5.1. Chapter Overview

A many-core architecture usually implements only a few voltage islands, each typically containing hundreds of cores. In general, the research questions to be addressed relate to how many islands should exist with what voltages, how they should be constructed, how many cores should be placed in each island, where should the islands be placed relative to one another and what criteria should be used during their formation and placement. Once the islands are established, the application tasks that are timing critical are mapped to voltage islands powered by relatively higher voltages, while tasks that are not timing critical are mapped to voltage islands powered by lower voltages. The regions powered by higher voltages will consume more power but execute faster, while the regions powered by low voltages will consume less dynamic and static power, but run slower [7][14][129]. In overall, the energy consumption is reduced, as compared to operation at the nominal voltage, while the required performance is delivered.

In the previous chapters, we described the different steps of the energy optimization flow for mapping applications onto many-core systems and addressed many of the above research questions. Details were presented on the task-graph timing analysis and slack calculations, task scheduling, voltage scaling, voltage selection, and data routing. In this chapter, we provide the details of the new algorithm for voltage island formation and placement (Step 5 of the flow in Figure 3.3).

---

4 This chapter is based on the contents of the papers published in [33] and [35]
The proposed algorithm, *Voltage Island Cloud* placement, or *VIC*, is a novel approach to construct voltage islands in many-core architectures by adjusting the island’s shape in order to: 1) fit island communication patterns, and 2) reduce the impact of within-die (WID) process variations. The islands are created by balancing the shape constraints imposed by island communication against the desire to limit their spatial extent to minimize WID process variations impact.

### 5.2. Voltage Island Design Methodology

The implementation of a voltage island design methodology increases the complexity of the IC design flow. For instance, power grid routing complexities are increased in order to deliver the required supply voltages to all the cores in the chip. In addition, extra blocks such as level-shifters, retention cells, isolation cells, and power gating switches are needed to realize multi-voltage domains. Thus, employing multi-supply voltage techniques in many-core architectures increase silicon area and power consumption due to the supporting blocks, and cause extra execution delays due to transition from low to high voltage modes. However, this power and delay overhead can be compensated by the final energy saving gains [7][14].

As mentioned in Chapter 2, there are three different approaches to implement multiple supply voltage design, also referred to as multi-Vdd design. These approaches differ in terms of design complexity and power management capability. The first method is called Static Voltage Scaling (SVS) [28][16][100][96]. In this case, the power grid is designed and fixed statically before fabrication. It is not possible to alter the location, size, or layout of the islands after fabrication. Furthermore, it is preferred in this technique to have a single island for each voltage level [28][16][100][96][29][99].
Therefore, when implementing more than one application, the applications have to be merged and analyzed together such that when islands are generated, there is a single island for each voltage level [18][29]. Results for the flow using SVS were presented in Chapter 3.

Another technique consists of having a power grid with multiple voltages. This technique is referred to as Multiple Voltage Scaling (MVS) [121][17][122][125][127]. A power grid with multiple voltages is designed and routed to cover all cores in the chip. Each core is allowed the selection of a voltage level, out of 2 or 3 options, that delivers the performance deadline. Although, this approach is limited in terms of supply voltage options, it provides a certain level of flexibility that allows the partitioning of a chip to separate application regions where each application can have its own islands. This flexibility reduces traffic congestions, voltage island size, and allows a better adaptation to the inter- and intra-island communication patterns. Results will be presented in this chapter for MVS and compared to SVS.

A third approach consists in building a design with dynamic voltage scaling (DVS) for each core. This technique is the most flexible as it allows cores to change the voltage supply depending on the workload dynamically during run time [104][44]. However, this approach requires a considerable timing and energy transition overhead, a complex power delivery system, and a complicated timing and power analysis during pre-fabrication [116]. As a result, we did not consider this option in this thesis.

5.3. Motivation for a New Voltage Island Formation Approach

In voltage island design, all cores within an island should ideally have the same performance. However, due to within-die variations individual cores in one island have
slightly different speeds. This speed discrepancy introduces additional idle time whenever the fastest cores wait for the slower ones to complete a task. This effect can be mitigated by increasing the supply voltage of the entire island to speed up the slower cores, but at the expense of higher power dissipation. In effect, the supply voltage level for each island is ultimately dictated by its slowest core [27][32][75][136][56].

In thousand-core platforms, the physical spread of cores makes speed differences among cores significant [20][61][64][69][136][137][138]. Since WID variation pattern cannot be predicted before fabrication, we aim to design islands with shapes that can better tolerate WID spatial variations in the fabricated chip, regardless of the direction of the gradients and the degree of variation. This is a very challenging task but some interesting progress can be made with the approaches to be described here.

Figure 5.1(a) shows an illustrative example of a possible pattern of within-die (WID) spatial variations. The dotted lines represent an example of the gradient change in core speed from one corner of the chip to another. In this work, we assume that the maximum distance at which the correlation between cores’ speeds become insignificant is half of the chip [23][50]. Figure 5.1(a) shows a rectangular island, VI1, and a circular island, VI2, with the same number of cores, and approximately the same area. The rectangular voltage island is affected the most by the systematic variations due to its span across the chip, whereas the circular voltage island VI2 is much more variation tolerant.

While some insight can be gained in this example, it is impossible to predict the actual gradients and the degree of variation. A possible post-fabrication solution to the problem is to include an on-chip monitoring system that increases the supply of the affected islands containing the slow cores. However, such a solution reverses any benefit
gained from the voltage island partitioning before fabrication [60][70][62]. Our goal is to form islands prior to fabrication that works well after fabrication, as in Figure 5.1(b), with as little adjustment to the target supply voltage as possible after fabrication.

![Diagram](image)

**Figure 5.1:** Many-core (a) circular vs. rectangular shapes, (b) cloud-based Voltage Islands.

However, to achieve this end, we have to take a closer look at the problem. For example, assume that VI1 in Figure 5.1(a) runs at 0.8V. As a consequence of process variation, a slow core at the slow corner requires 1.0V. The adaptive voltage scaling (AVS) monitor must raise the supply voltage for the whole island, which may offset the energy savings of the initial voltage island optimization. Thus, it is important to minimize the supply voltage adjustment due to core-speed discrepancy within islands.

Circular-shaped voltage islands would minimize the maximum distance between cores within each island, and consequently core-speed differences, as compared to rectangular-shaped voltage islands. This is because the maximum distance between two cores is less in a circle-shaped island than in a square- or a rectangular-shaped island of the same area. Adjustments of the supply voltage to 0.85V may be sufficient here.
A circular island, then, might better mitigate the process variation impact by reducing the maximum distance between cores. Also, it improves intra-island communications between cores within the same island. However, the circular shape has two disadvantages: it is hard to force all voltage islands into circular shapes, and it is difficult to closely-pack circular voltage islands together without penalizing inter-island communications. Moreover, any pre-designated rigid shape, whether rectangular, square or circular, might have a negative impact on communication. Therefore, the shape also has to be adjusted and adapted to the communication traffic among cores and islands, which could result in the islands shown in Figure 5.1(b). Based on this, we propose to build irregular cloud-based formations for our voltage islands, which reduce the core-speed differences in an island. In our approach, these cloud-based islands are defined by initial seeds from which they will grow. The exact shape of these islands depends on the communication patterns and the degree of PVT variation expected on the chip.

5.4. Problem Formulation

In this section, we formalize the problem we are solving and define the inputs and outputs of our island placement algorithm. We have to achieve two goals: (1) determine the location of each island, and (2) place the cores of within each island. Then we present our proposed cloud-based voltage island placement algorithm.

Problem Statement: Given an undirected task-graph with \( m \) subgraphs (islands), each consisting of \( Y \) nodes (cores) with communication weights (CW) among cores, find a placement of the islands on a many-core platform such that:

(a) Islands with heavy communication are placed close to each other.

(b) Cores in an island are placed based on the level of inter- and intra-island
communication. Thus, cores with heavy intra-island traffic are placed close to the center of the island, and cores with high inter-island traffic are placed at the border of this island and closer to the island within which they communicate.

(c) Islands that will be supplied the same voltage level, i.e., merged later as dictated by the Voltage Selection step (Removal Cost Method discussed in Chapter 4), should be placed next to each other.

(d) Distances among cores in the same island are minimized to reduce WID process variations impact.

The requirements listed above are illustrated in a simple example in Figure 5.2. It shows, on the left of the figure, an undirected graph of cores with its subgraphs (islands) indicated by shaded areas. The derived voltage island placement is shown on the right of the figure. As shown, all cores in an island are placed close to each other to reduce PVT impact. $C_2$, $C_3$ and $C_7$ communicate heavily and therefore islands $VI_1$ and $VI_3$ are placed close to one another. Typically, islands that have voltages that are close in value are placed next to one another to make power routing more convenient when the final voltages are established. Assuming that, in this example, $VI_1$ and $VI_2$ will be merged later as dictated by the Voltage Selection step, they are placed next to one another. In the following sections we show the details of the algorithm to highlight this point further.

Figure 5.2: Example of the undirected graph of voltage islands (VI) to be placed.
5.5. Voltage Island Seeding

Voltage island seeding is a step in which we select an origin for each island from which the island is gradually grown. In this section, we discuss the selection of seeds from a conceptual point of view, while the implementation itself is explained later in Section 5.9.

Figure 5.3: Seeding hierarchy used in the Voltage Island Clouds (VIC) placement.

Assume that the number of cores in each Voltage Island Cloud (VIC) has already been determined and that the VICs have been grouped into $d$ sets. All VICs in a set initially have different voltages but some will eventually be assigned the same voltage level and, hence, merged into a single larger island. The number of VICs in set $j$ is denoted $n_j$. This is illustrated in Figure 5.3 where there are $d=3$ sets, each indicated by a
different color. The three VIC sets are, \{VIC_1, VIC_7, VIC_8, VIC_9\}, \{VIC_2, VIC_5, VIC_6\}, and \{VIC_3, VIC_4\}, where \(n_1=4\), \(n_2=3\), and \(n_3=2\), respectively. The cores within each set initially have different voltages but are placed close to each other so that they can be merged later on, when they will be assigned the same voltage level, which is determined at the Voltage Selection step.

The placement of the VICs is carried out using hierarchical seeding. Before placing the cores, the seeds of the voltage islands are determined. Each seed represents a physical location on the platform. There are three types of seeds. The highest level seed, called the overall placement seed, is the origin of the entire placement as it is used to anchor all sets on the chip. The location of the overall placement seed is discussed later in Section 5.9. For each set, there is one intermediate-level seed, called a set seed. This seed anchors the voltage islands within the set. Finally, there is a local seed, or island seed, for each voltage island that is used to anchor individual cores within an island. The shortest Euclidean distances between seeds are used to place the VICs and determine the core location in the design.

As shown in Figure 5.3, the seeding algorithm starts by choosing the overall placement seed, which always coincides with the set seed of the first VIC to be placed, e.g., VIC_1 in the figure. The seed of VIC_1 pulls together the seeds of VIC_7, VIC_8 and VIC_9. The other two set seeds are those of VIC_2, pulling VIC_5 and VIC_6, and VIC_3 pulling VIC_4. Then, the cores of VIC_1 are placed such that they are as close as possible to the VIC_1’s seed. VIC_7’s seed is selected at the closest available location to VIC_1’s seed. The cores of VIC_7 are placed around the VIC_7’s seed. VIC_8 and VIC_9 are placed in the same way. The islands within a set, i.e., VIC_7, VIC_8 and VIC_9, are placed according to the level
of communication between each one and $VIC_i$, as follows. The island that has the highest level of communication is placed first, and the one with the least communication is placed last. Moving to the next set of islands, assuming that $VIC_2$ communicates more with the first set as compared to $VIC_3$, $VIC_2$ has to be placed second. $VIC_2$’s set seed is placed at the closest available location to the placement seed. The placement of the rest of the VICs continues in the same fashion.

5.6. Communication and PVT Weighting Model

In this section, we discuss the weighting models for inter- and intra-communication and PVT variations that control the placement of each core. PVT variations require cores within an island to be pulled together to minimize distances between cores, hence reducing the possibility of large speed differences among them. Moreover, to facilitate communication between cores, a core has to be placed close to another core if there is heavy communication traffic between them. Intra-island communications require cores within an island to be placed close to each other, hence circular-shaped islands are preferable in this case. On the other hand, inter-island communications require islands to share longer borders, if many cores involved, hence long rectangular-shaped islands are better-suited for in this case. Thus, we develop a weighting model that attempts to find a balance between these conflicting factors.

Cores associated with a specific VIC should also be placed as close as possible to this VIC's local seed to keep the island intact. Their location should also satisfy the requirements imposed by the communication with other VICs. Thus, cores with heavy intra-island communication converge towards the middle of the island. Cores with heavy inter-island communications are instead placed at the border and closer to the islands
they are communicating with. As an example, in Figure 5.3, core \( C_i \) is pushed away from the \( VIC_i \)’s seed and pulled towards \( VIC_2 \) and \( VIC_3 \) seeds with whom it communicates.

Following this strategy, the weighting model, called \textit{Core Assignment Weight} (CAW), is used to compute the best core placement location. The model is shown in Equation (5.1).

\[
CAW_i = SW_{ki} \times D_{ki} + \sum_{l} CW_{li} \times D_{li}; \quad i \in \{1,...,I\} \quad (5.1)
\]

where \( SW_{ki} \) is the seed weight of the \( VIC_k \) to which core \( C_i \) belongs. The calculation of the seed weight, \( SW_{ki} \), is discussed in the next section. \( CW_{li} \) is the communication weight; that is, the amount of the communication between core \( C_i \) and all \( L \) VICs. The quantity \( \{1,...,I\} \) is the set of available cores, and finally the \( D_{li} \)’s are the Euclidian distances between core \( C_i \) and \( VIC_i \)’s seed. This function is used to evaluate all available core locations. The location that results in the minimum \( CAW \) value is used as the best location to place the core.

\section*{5.7. Seed Weight Calculation}

The quantity \( SW \) is a tuning parameter used to determine whether to force all cores to be tightly grouped together for PVT or to relax the island shape in order to improve communication. The value of \( SW \) can be any number between 0 and \( \infty \). This value depends on the limitations and the capabilities of the two voltage island implementation approaches: SVS and MVS. Each case is described below.

\subsection*{5.7.1. SW Calculation for Static Voltage Scaling (SVS)}

As stated before, implementing voltage islands with SVS forces all cores supplied with same voltage to be grouped in one single island to reduce power grid routing
complexity [18]. Thus, in this case, $SW$ is set to $\infty$. As a result, the communication weights, $CW$, are ignored and each voltage island is forced to be tightly grouped. This approach also minimizes the distances between cores, hence reducing the cores’ frequency differences due to within-die process variation impact and improving intra-island communication. On the other hand, since the communication weight is ignored in shaping the islands, this approach has a negative impact on inter-island communications.

5.7.2. SW Calculation for Multiple Voltage Scaling (MVS)

In this second approach, we assume a chip-wide power grid with multiple supply voltages where each island can select its appropriate voltage level. Thus, the shape of the islands can be adjusted based on the communication weights, $CWs$, to facilitate inter-island communications. Although cores with same supply voltage still have to be grouped together [121][17][123], cores can be placed far from the local seed in order to correspond to actual data transfer patterns.

We calculate the $SW$ to be the arithmetic mean of all $CWs$ as shown in Equation (5.2).

$$SW_k(C_i) = \text{mean} \left( \sum_{l=1}^{b} CW_l \right); \quad (5.2)$$

Setting the $SW$ to be the mean of the communication weights keeps the core attached to its island. Increasing the $SW$, beyond this value, pulls the cores tighter around the seed which tends to degrade the inter-island communication.

On the other hand, reducing the value of $SW$ to a minimum, e.g., $SW=0$, might scatter the cores too far from their island’s seed, which also impacts the inter- and intra-island communication traffic. This situation could also increase the number of separate islands that have the same voltage level. Although the chip-wide power grid design may
allow for this, there will be an increase in the number of level shifters, hence more power dissipation and design complexity [121][17]. Furthermore, WID process variation impact will increase as the speed differences among the cores of that island increases as well. As we show later in Section 5.9, setting the seed weight to the mean of the communication weights creates islands that provide larger energy savings for designs with heavy inter-island communications.

Table 5.1: Example of CAW Calculation

<table>
<thead>
<tr>
<th>Core’s to be placed</th>
<th>Seed Weight: $SW_1$</th>
<th>Comm Weight: $CW_2$</th>
<th>Comm Weight: $CW_3$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$C_i$</td>
<td>6</td>
<td>10</td>
<td>2</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Available Cores</th>
<th>$D_{H_1}$</th>
<th>$D_{H_2}$</th>
<th>$D_{H_3}$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$C_1$</td>
<td>1</td>
<td>10</td>
<td>3</td>
</tr>
<tr>
<td>$C_2$</td>
<td>10</td>
<td>6</td>
<td>1</td>
</tr>
<tr>
<td>$C_3$</td>
<td>2</td>
<td>3</td>
<td>10</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>CAW Calculations</th>
</tr>
</thead>
<tbody>
<tr>
<td>$CAW$ of $G$</td>
</tr>
<tr>
<td>$D_{H_1} \times SW_1 + D_{H_2} \times CW_2 + D_{H_3} \times CW_3 = CAW_1$</td>
</tr>
<tr>
<td>$1 \times 6 + 10 \times 10 + 3 \times 2 = 112$</td>
</tr>
<tr>
<td>$CAW$ of $G$</td>
</tr>
<tr>
<td>$10 \times 6 + 6 \times 10 + 1 \times 2 = 122$</td>
</tr>
<tr>
<td>$CAW$ of $G$</td>
</tr>
<tr>
<td>$2 \times 6 + 3 \times 10 + 10 \times 2 = 62$</td>
</tr>
</tbody>
</table>

Table 5.1 shows a simple example of calculating CAW. As shown in Figure 5.3, core $C_i$ in island $VIC_1$ is pulled across the island towards $VIC_2$ and $VIC_3$. Assume that $C_i$ is communicating with $VIC_2$ and $VIC_3$ and has communication weights $CW_2=10$ and $CW_3=2$, respectively, as shown in Table 5.1. Then, $SW$ is calculated using Equation (5.2), i.e., $SW = (10 + 2) / 2 = 6$.

Assume that there are three available core sites $C_1$, $C_2$, and $C_3$ in island $VIC_1$ (although not shown in Figure 5.3), such that the distances to the seeds of $VIC_1$, $VIC_2$, and...
and VIC3 from each location are shown in the table. As indicated in the table, C1 is closer to VIC1 and VIC3, but far from VIC2. The location of C2 is closer to VIC3 but far from VIC1 and VIC2, and location of C3 closer to VIC1 and VIC2 but far from VIC3. Given that C1 belongs to VIC1, the seed weight, SW=6, is multiplied by the distances of the three locations to the VIC1 seed, i.e., 1x6, 10x6, and 2x6, for the three locations, respectively. Then, the CAW function multiplies the communication weights by the distances to each VIC, as shown in the table, for the three locations, respectively. The total CAW values for the three locations are 112, 122, and 62, respectively, and C3 has the minimum CAW; thus, it is the best choice, according to the model, to place C1 within island VIC1.

5.8. Application Partitioning

Since many-core architectures consist of hundreds to thousands of cores, such architectures are suitable for executing multiple applications simultaneously. This permits a higher utilization of the array than one application would provide, but requires partitioning the available cores between the applications. An application is allocated to a part of the platform based on its priority and on the number of cores it requires.

In a voltage island based design, the application allocation also varies depending on the voltage island design approach. For the SVS design approach, there is a single voltage island per voltage level. Therefore, voltage islands must be shared by different applications [7][16][29][100]. Figure 5.4(a) shows an illustration of two islands, A and B, each one divided into four regions each implementing a different application. Sharing voltage islands among applications increases inter-island congestion and results in extra routing costs.
Figure 5.4: Illustration of two islands and four applications in the case of (a) SVS and (b) MVS voltage island design approaches.

For the MVS design approach, there can be more than one island with the same voltage level [121][17][122][127]. Thus, each application can have its dedicated group of islands as shown in Figure 5.4(b). Since the platform is divided into separate application regions, this reduces the traffic congestion as compared to the previous approach.

The partitioning algorithm used for the MVS approach is shown in Figure 5.5. It is a variation of Two-Bullies algorithm [139], which is simple and efficient enough for our application partitioning problem. The platform is partitioned into rectangles, one for each application. The algorithm uses recursion where in each call the sizes of the applications and the dimensions of the available part of the platform are passed on. Figure 5.6 shows an illustration of a many-core divided into four regions to implement four applications. In this example, consider a 30x30 platform and four applications with the sizes of 600, 300, 100, and 50 cores. Note that the total application sizes is 1050, i.e., more than the total number of available cores.
The algorithm starts by splitting the applications into two equally sized groups, e.g., application 1 (600 cores), and applications 2, 3, and 4 (450 cores). Accordingly, the platform is partitioned into two parts of sizes: 18 (the result of ceil\(600\times30/1050\) =18)
by 30 for application 1 and 12x30 for the rest of the applications. The algorithm then progresses to split the area of 12x30 into 12x20 for application 2 and 20x10 for the last two applications. Finally, the last iteration splits the last portion into 14x10 for application 3 and 6x10 for application 4.

Figure 5.6: An illustration of platform partitioning for four applications.

5.9. Voltage Island Cloud Placement Algorithm

Figure 5.7 shows the pseudo-code of the proposed algorithm. The algorithm includes both voltage island design approaches: SVS and MVS. The Seed Weight, $SW$, determines the implementation approach. In the SVS approach, the $SW$ is set to infinity, such that the communication weights are ignored. In the MVS approach, the $SW$ is calculated as described in Section 5.7.2. Moreover, the algorithm is performed for each application in its designated region.
To summarize the algorithm, we first set the overall placement seed to be at the corner of the chip (other strategies are possible, but this one produces the best results).

Figure 5.7: Pseudo-code of Voltage Island Cloud placement algorithm.
We then cycle through each VIC set, each time considering the set that has the highest inter-island communication with the islands already placed. For this set, we determine the closest unused location to the placement seed, and assign this to the set seed. We then cycle through all voltage islands within the set. For each voltage island, we determine the closest unused location to the set seed, and assign this to the island's local seed. Finally, the cores are placed using the weighting model, $CAW$, Equation (5.1).

![Voltage Island Color Code](image)

(a) ![Cloud-based voltage islands before merging](image)  
(b) ![Cloud-based voltage islands after merging](image)

Figure 5.8: Cloud-based voltage islands (a) before and, (b) after island merging, on 30x30 many-core, implemented using SVS, with $SW=\infty$.

Figure 5.8 shows an example of an optimized cloud-based voltage islands generated using the Algorithm 5.5 above and energy optimization flow discussed in Chapter 3. In this example, a task-graph with a total of 1300 tasks is mapped onto an architecture contains 900 cores. The possible island voltage levels are selected between 0.6V and 1.4V with 0.05V increments. The voltage levels of the islands are shown in the shading outside the core square, and the frequency values, due to WID process variation,
are shown by the shading inside the core square.

Figure 5.8(a) shows islands, placed using SVS, before merging, with the seed weight set to $SW=\infty$. Figure 5.8(b) shows the islands after island merging. In this example, the placement seed is selected at the top left corner of the platform. The islands shown in white dotted lines, i.e., 0.7V, 0.75V, 0.8V and 1.0V, have all been merged into one 1.0V island. The merging order is determined by the Voltage Selection algorithm, discussed previously in Chapter 4. Furthermore, the 1.4V island is positioned closer to the 1.1V and 1.0V islands, but further away from the 0.6V island. This order implies that the algorithm has taken into account the fact that the 1.4V island is communicating with 1.1V and 1.0V more than the 0.6V island.

![Voltage Island Color Code](image)

(a) ![Islands Before Merging](image) (b) ![Islands After Merging](image)

Figure 5.9: Cloud-based voltage islands (a) before and, (b) after island merging, on 30x30 many-core, implemented using MVS, $SW = \text{mean} \left( \sum CW \right)$.

Figure 5.9(a) shows the cloud-based voltage islands placed using MVS, before island merging using a seed weight, $SW$, as in Equation (5.2). Unlike the SVS case, the
applications are assigned separate regions of the platform as described earlier in Section 5.8. Figure 5.9(b) shows the islands after island merging. In addition, the placement seed is placed in the top left corner of the platform.

Cloud-based islands are assumed to have a smaller average distances between cores within an island. However, whenever there are large islands, this assumption sometimes cannot be satisfied as shown in Figure 5.8. On the other hand, Figure 5.9 shows smaller distances due to smaller island sizes. Therefore, we expect better results from MVS, although the power routing is more complicated.

5.10. Experimental Platform

In this section, we describe the experimental setup used to evaluate the proposed techniques. The energy optimization flow, shown again in Figure 5.10 for convenience, is used to generate the voltage islands. As before, we used 33 test cases. Table 5.2 shows the characteristics of the mapped applications. The first 30 task-graphs were generated at random by [131] and the last 3 were actual applications from prior work: Sparse Matrix Solver (SMS), a Robot Control Program (RCP), and a SPEC95 fppp-kernel [131].

It is important to note that it is currently difficult to obtain standard benchmarks with thousands of tasks targeted for many-core architectures. Therefore, it is useful to investigate the nature of the benchmarks to be used. Table 5.2 characterizes the task-graphs by three parameters: critical path length, parallelism ratio, and number of communication edges. The critical path length is the processing time of the tasks that lie on the critical path. The parallelism ratio is defined by the sum of computational times of all tasks divided by the critical path time. The last column shows the amount of communications in the task-graphs.
The numbers of cores in each many-core platform considered in the experiments are 400, 625, 900, 1225, and 1600 cores. Furthermore, the value of the voltage levels can anywhere from 0.6V to 1.4V in 0.05V increments [16][35]. Finally, the number of permitted voltage levels is 4 and any unused cores are switched off completely. The maximum speed at which the processor can run is 300MHz with the network running at 600MHz. The core nominal dynamic power is estimated to be 0.45mW/MHz (assumed to be ARM© Cortex-A8) [135].

In every test case, a group of task-graphs, as listed in Table 5.2, are mapped to a many-core platform. At first, the optimization and placement operations were carried out assuming a sample process variation profile; this part is assumed to simulate the pre-fabrication scenario.
Table 5.2: Characteristics of the applications used in the Experiments [131]

<table>
<thead>
<tr>
<th>Number of Test Cases</th>
<th>Number of Taskgraphs per Test Case</th>
<th>Number of Tasks Total</th>
<th>Number of Tasks Per TG</th>
<th>Parallelism Range</th>
<th>Critical Path Length Range (Time Units)</th>
<th>Number of Communication Edges</th>
</tr>
</thead>
<tbody>
<tr>
<td>30 randomly generated</td>
<td>4 TGs grouped together</td>
<td>950 tasks</td>
<td>500 tasks</td>
<td>16 up to 55</td>
<td>67 up to 315</td>
<td>504 up to 4504 edges</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>300 tasks</td>
<td>11 up to 47</td>
<td>45 up to 264</td>
<td>277 up to 2901 edges</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>100 tasks</td>
<td>8 up to 20</td>
<td>29 up to 122</td>
<td>93 up to 915 edges</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>50 tasks</td>
<td>7 up to 12</td>
<td>24 up to 70</td>
<td>46 up to 941 edges</td>
</tr>
<tr>
<td>Sparse Matrix Solver (SMS)</td>
<td>8 TGs grouped together</td>
<td>768 tasks</td>
<td>96 tasks</td>
<td>15.8</td>
<td>122</td>
<td>67 edges</td>
</tr>
<tr>
<td>Robot Control Program (RCP)</td>
<td>8 TGs grouped together</td>
<td>704 tasks</td>
<td>88 tasks</td>
<td>4.36</td>
<td>569</td>
<td>131 edges</td>
</tr>
<tr>
<td>SPEC95.fpppp</td>
<td>2 TGs grouped together</td>
<td>668 tasks</td>
<td>334 tasks</td>
<td>6.7</td>
<td>1062</td>
<td>1145 edges</td>
</tr>
</tbody>
</table>

Figure 5.11: Selected task-graphs characteristics, number of communication edges versus
(a) Parallelism, and (b) Critical path length.

Afterwards, in the post-fabrication scenario, the resulting mapping was enforced on a number of other arbitrary profiles; that is, the pre-fab islands and their placements were all used in a post-fab context to determine if they were able to handle the unknown PVT variations. Eight different within-die process variation profiles generated at random,
according to VARIUS© model explained in Chapter 2, were used in the post-fabrication scenarios. Then, the supply voltages of the islands in the many-core platform were adjusted until the slowest core in each island met the frequency requirements. Typically, the supply voltage adjustments are small but some required between 5% to 10%.

Figure 5.11 shows the characteristics of the randomly generated and real benchmarks (circled) [131]. As shown in Figure 5.11(a), the benchmarks span the spectrum from highly-serial to highly-parallel task-graphs. Since the number of tasks per benchmark is the same, the parallelism is expected to go down as communication among tasks increases. Also shown, the real applications, i.e., SMS, RCP, and SPEC95 (fpppp-kernel), are either highly-serial, e.g., RCP and SPEC95, or highly-parallel, e.g., SMS. Similarly, Figure 5.11(b) shows an increase in critical path length as the number of communication edges per test case increases. This information is used later to analyze the proposed methods.

5.11. Results and Analysis

In this section, we discuss the energy saving results of the proposed techniques from several different perspectives. The key questions to be addressed are: to what degree does the voltage island shape influence the overall energy improvement obtained in the flow; what the level of improvement of MVS over SVS; what the effect of speed correlation and what is the effect of large PVT variations on the results.

5.11.1. Voltage Island Shape

The question of how important the voltage island shape is in determining the improvement is addressed here. The results presented are all relative to Static Voltage
Scaling with Rectangular Voltage Islands (SVS-RVI), against Static Voltage Scaling with Voltage Island Cloud (SVS-VIC), and Multiple Voltage Scaling with Voltage Island Cloud (MVS-VIC). Figure 5.12(a), and (b) show the normalized energy, with respect to SVS-RVI, for selected 33 benchmarks implemented on 30x30 many-core design, versus communication, and parallelism, respectively.

![Figure 5.12](image)

**Figure 5.12:** Normalized energy consumption relative to SVS-RVI, versus (a) Communication edges, and (b) Parallelism.

The results from the 33 task-graph mappings on the eight different process variation profiles were grouped and averaged (geometrical mean). As shown in Figure 5.12(a), the energy consumption, normalized to 1.0 for SVS-RVI, is reduced by using either SVS-VIC and MVS-VIC, which means that both methods are better than SVS-RVI as communication increases. In most cases shown, MVS-VIC is showing less energy consumption as compared to SVS-VIC. Hence, it is best of the three methods when considering task-graphs with high communication levels. Note that in one case, SVS-VIC and MVS-VIC are showing worse energy numbers than the SVS-RIC. This is an indication that for this specific application, the communication is mostly inter-island.
traffic where rectangular shape is superior in that case. Another example shows SVS-VIC to be better than MVS-VIC, and this is an indication that the communication is mostly intra-island communication, where SVS-VIC would be the best.

![Graphs (a) to (d)]

Figure 5.13: Energy saving over SVS-RVI, $\alpha/\mu=0.12$; $\varphi=0.5$, (a) geometric mean of 30 randomly generated TGs, 950 tasks (b) Sparse Matrix Solver, 8x96 tasks, (c) Robot Control Program, 8x88 tasks, (d) SPEC95 fpppp-kernel.

Figure 5.12(b) shows that the normalized energy consumption increases, for both SVS-VIC and MVS-VIC, as parallelism increases. This makes sense relative to Figure 5.12(a) because communication decreases as parallelism increases, as seen earlier in Figure 5.11(a). However, both methods are still better than the SVS-RVI, with MVS-VIC outperforming SVS-VIC.

Figure 5.13(a) through (d) show the total energy savings improvement for both methods, SVS-VIC and MVS-VIC, over SVS-RVI before island merging, assuming
variation ratios of $\sigma/\mu=0.12$ and maximum correlation distance of $q=0.5$, normalized to the chip width. These results represent the energy savings achievable with 18 different voltage levels.

Figure 5.13(a) shows the geometric mean of the energy savings improvement compared to SVS-RVI of the 30 test cases of randomly generated task-graphs as discussed in Table 5.2. The savings in MVS-VIC is approximately 8% more than that of SVS-VIC. This is consistent with the expectations, as discussed earlier in Section 5.7.2. Setting the seed weight to $SW=\text{mean} \sum(CW)$ produces better results, where the islands incorporate the anticipated inter-island communication load, as compared to $SW=\infty$. Moreover, assigning each application its own part of the platform avoids any increase in traffic due to the sharing of islands among all applications as in SVS-VIC.

SVS-VIC and MVS-VIC achieve savings improvement of 6-16% and 14-25% over SVS-RVI. This is also consistent with our expectations, as the VIC placement is optimized for both communication load and PVT.

Figure 5.13(b), (c), and (d) show the energy savings improvements for Sparse Matrix Solver (SMS), Robot Control Program (RCP), and SPEC95 kpppp-kernel, respectively. As shown in Figure 5.11(a) and (b), SMS has a high parallelism ratio, a low communication load, and a short critical path length. These characteristics suggest that SMS is a highly parallel application. As shown in Figure 3.14(b), the savings improvement achieved by SVS-VIC and MVS-VIC over SVS-RVI are approximately 6-12%.

Figure 5.13(c) shows that the RCP energy savings improvement for SVS-VIC and MVS-VIC are very close to each other, approximately 4-5%, and that they are not
achieving high energy saving due to the low communication load, low parallelism and long critical path length, as shown in Figure 5.11(a) and (b). The very long critical path length, and low parallelism suggest that this application is mostly serial and cannot benefit as much from a multi-core architecture. Also, SPEC95 (fpppp-kernel) benchmark has similar characteristics as RCP, and it is achieving 6-8% as compared to SVS-RVI as shown in Figure 5.13(d). In conclusion, the expected energy savings for the proposed techniques are dependent on the characteristics of the application, so these results are not very surprising given the nature of the benchmarks used in the experiments.

5.11.2. Core-Speed Correlation Impact

Recall that the normalized correlation distance, $\phi$, is defined as the maximum distance at which the speed correlation between two cores becomes insignificant. In Chapter 2, we described that $\phi=0.5$ means that 50% of the chip is correlated in core speed, whereas $\phi=0.1$ implies a reduced correlation of 10% of the chip. The previous section showed the results with $\phi=0.5$. In this section, we study the case where $\phi=0.1$. As usual, the process variation was generated using the VARIUS model of [50] described in Chapter 2. The results relative to SVS-RVI are shown in Figure 5.14.

Figure 5.14(a) through (d) assume the maximum correlation distance to be $\phi=0.1$. The drop in savings due to the correlation decrease is about 2-4% for both SVS-VIC and MVS-VIC relative to their original values in Figure 5.13, where $\phi=0.5$. Therefore, if the correlation in processor speed is high across the chip, then the energy improvement is high. A low correlation implies the opposite.
5.11.3. Increased Process Variation Level

In this section, we investigate the drop in savings as the variation ratio increases. Figure 2.4 showed the probability density function of the frequencies on the chip for different variation ratios. As was shown, when variation increases, the frequency distribution becomes more spread out. In this work, we investigated variation ratios of $\sigma/\mu=0.16$, and $\sigma/\mu=0.18$. Here, we show the 0.18 case only as the 0.16 case shows a similar trend.
Figure 5.15: Energy savings over SVS-RVI, $\sigma/\mu=0.18$, $\phi=0.5$, (a) geometric mean of 30 randomly generated task-graphs (b) SMS (c) RCP, and (d) SPEC95, fpppp-kernel.

Figure 5.15(a) through (d) show the energy savings improvement over SVS-RVI, assuming a variation ratio of $\sigma/\mu=0.18$ and $\phi=0.5$. The results are lower as compared to Figure 5.14. The drop in the total energy savings is 5-8% in both SVS-VIC and MVS-VIC. In most cases, the proposed methods are very sensitive to higher level of variation; nonetheless, the results are still showing better than SVS-RVI. This is an important point to emphasize at this stage since the results are affected by the application characteristics, core frequency correlations and degree of process variations. However, the MVS-VIC method consistently outperforms the rectangular approach in all cases.
5.11.4. Final Energy Savings after Voltage Selection

The energy savings are all calculated after the island merging. The merging is by the results of voltage selection. We compare RCM approach and 1-STEPLA algorithm of [16]. This will identify the improvement that is derived from the voltage selection method versus that derived from voltage island formation of our work. Dynamic programming is not shown here as it gives the same results as RCM.

![Figure 5.16: Energy saving after Voltage Selection, $\sigma/\mu = 0.12$, and $\varphi=0.5$; (a) geometric mean of 30 randomly generated, (b) SMS, (c) RCP, and (d) SPEC95, fppp-kernel.](image)

Figure 5.16 shows the final energy savings improvements relative to SVS-RVI with 1-STEPLA after Voltage Selection, for a variation ratio of $\alpha/\mu=0.12$ and $\varphi=0.5$. We do not show the cases where $\varphi =0.1$, $\sigma/\mu=0.16$ or $\sigma/\mu=0.18$ variations as they follow the
same trend explained earlier. A total energy savings improvement of 8-30% is obtained using our proposed techniques over rectangular-shaped islands with 1-STEPLA.

Table 5.3: Runtime Geometric Mean of the 33 Test Cases [131]

<table>
<thead>
<tr>
<th>N# of Cores</th>
<th>Method</th>
<th>Runtime (hours)</th>
<th>Machine Specifications</th>
</tr>
</thead>
<tbody>
<tr>
<td>400</td>
<td>SVS-RVI, 1-STEPLA</td>
<td>17.97</td>
<td>Westgrid, Glacier, 3.06GHz, Intel Xeon 32-bit processor, 512KB cache, 2GB RAM</td>
</tr>
<tr>
<td></td>
<td>SVS-VIC, RCM</td>
<td>13.05</td>
<td>Westgrid, Glacier, 3.06GHz, Intel Xeon 32-bit processor, 512KB cache, 2GB RAM</td>
</tr>
<tr>
<td></td>
<td>MVS-VIC, RCM</td>
<td>9.16</td>
<td>Westgrid, Glacier, 3.06GHz, Intel Xeon 32-bit processor, 512KB cache, 2GB RAM</td>
</tr>
<tr>
<td>625</td>
<td>SVS-RVI, 1-STEPLA</td>
<td>26.69</td>
<td>Westgrid, Glacier, 3.06GHz, Intel Xeon 32-bit processor, 512KB cache, 2GB RAM</td>
</tr>
<tr>
<td></td>
<td>SVS-VIC, RCM</td>
<td>13.05</td>
<td>Westgrid, Glacier, 3.06GHz, Intel Xeon 32-bit processor, 512KB cache, 2GB RAM</td>
</tr>
<tr>
<td></td>
<td>MVS-VIC, RCM</td>
<td>9.16</td>
<td>Westgrid, Glacier, 3.06GHz, Intel Xeon 32-bit processor, 512KB cache, 2GB RAM</td>
</tr>
<tr>
<td>900</td>
<td>SVS-RVI, 1-STEPLA</td>
<td>36.93</td>
<td>Westgrid, Glacier, 3.06GHz, Intel Xeon 32-bit processor, 512KB cache, 2GB RAM</td>
</tr>
<tr>
<td></td>
<td>SVS-VIC, RCM</td>
<td>23.40</td>
<td>Westgrid, Glacier, 3.06GHz, Intel Xeon 32-bit processor, 512KB cache, 2GB RAM</td>
</tr>
<tr>
<td></td>
<td>MVS-VIC, RCM</td>
<td>15.50</td>
<td>Westgrid, Glacier, 3.06GHz, Intel Xeon 32-bit processor, 512KB cache, 2GB RAM</td>
</tr>
<tr>
<td>1225</td>
<td>SVS-RVI, 1-STEPLA</td>
<td>44.09</td>
<td>Westgrid, Glacier, 3.06GHz, Intel Xeon 32-bit processor, 512KB cache, 2GB RAM</td>
</tr>
<tr>
<td></td>
<td>SVS-VIC, RCM</td>
<td>27.56</td>
<td>Westgrid, Glacier, 3.06GHz, Intel Xeon 32-bit processor, 512KB cache, 2GB RAM</td>
</tr>
<tr>
<td></td>
<td>MVS-VIC, RCM</td>
<td>18.79</td>
<td>Westgrid, Glacier, 3.06GHz, Intel Xeon 32-bit processor, 512KB cache, 2GB RAM</td>
</tr>
<tr>
<td>1600</td>
<td>SVS-RVI, 1-STEPLA</td>
<td>19.28</td>
<td>anan.ece.ubc.ca, 2.66GHz, Intel Core (TM) 2 Quad CPU, 32-bit processor, 6MB cache, 8GB RAM</td>
</tr>
<tr>
<td></td>
<td>SVS-VIC, RCM</td>
<td>12.65</td>
<td>anan.ece.ubc.ca, 2.66GHz, Intel Core (TM) 2 Quad CPU, 32-bit processor, 6MB cache, 8GB RAM</td>
</tr>
<tr>
<td></td>
<td>MVS-VIC, RCM</td>
<td>7.38</td>
<td>anan.ece.ubc.ca, 2.66GHz, Intel Core (TM) 2 Quad CPU, 32-bit processor, 6MB cache, 8GB RAM</td>
</tr>
</tbody>
</table>

The proposed voltage island formation along with the task scheduling contribute to approximately 8-25%, while the RCM for voltage selection contributes by approximately 5-7% to the total energy savings improvement.
Table 5.3 shows the runtimes taken to execute the optimization for each of the three methods. SVS-VIC-RCM is faster by approximately 1.5X, while MVS-VIC-RCM is faster by approximately 2X, both compared to SVS-RVI-1-STEPLA. The shown numbers are geometric means of all 33 benchmarks.

*Based on these results, the MVS-VIC approach combined with RCM is the most effective approach for voltage island formation, placement and voltage selection, on many-core architectures and this is the key finding from the all experimental work carried out in this thesis.*

### 5.12. Limitations

Voltage island implementation on many-core is subjected to many factors that influence the final overall energy consumption. These factors can be categorized into two different categories: application and technology related factors. Voltage island size, final voltage levels, number of islands, and adjacency of an island to another island are factors imposed by the application characteristics such as parallelism, available slack, and communications. If an application, with thousands of tasks, is highly parallel, then the islands will most probably be large. If the island is too large then PVT impact on islands placed using cloud- or rectangular-based techniques is similar. The larger the island, the higher the chance to have a slow core within its border. If the cloud-based island spans the whole chip then the distances between cores are too large and speed discrepancy will exist anyway.

Technology dependent factors such as core speed correlation, the level of variation will also have a negative impact on energy consumption as discussed in the previous section. Core speed correlation is an assumption used in cloud-based islands. As
technology scales down, if the random variation becomes more considerable as compared to the systematic one, then our proposed technique might be less effective.

5.13. Chapter Summary

In this chapter, we proposed a novel approach for voltage island formation in many-core designs considering the influence of communication load and within-die process, temperature, and voltage variations. Cloud-based voltage islands are formed by balancing the communication load constraints and core-grouping to reduce PVT impact. Because of the locality of PVT variations, cloud-based voltage islands are more PVT-tolerant than rectangular-shaped voltage islands.

The voltage islands formation is realized in two different island design styles: the SVS and the MVS. The proposed algorithm, Voltage Island Cloud or (VIC), utilizes the available power grid capabilities in each design approach to build the islands. The islands are formed and shaped based on the inter- and intra-island communication. The geometric average energy savings improvement obtained for 33 benchmarks using the new techniques is 8% to 30% compared to the previous methods.
Chapter 6: Conclusions and Future Work

6.1. Research Context

The driving force for technology scaling is to increase transistor performance, to double transistor density in order to integrate more systems on a single chip in each successive technology generation, and to reduce energy consumption by reducing the supply voltage [140]. However, simply harvesting transistor performance to boost the overall system speed is no longer possible [3]. Moreover, transistor density is expected to reach 100 billion transistors per 300mm$^2$ area by 2015 [2]. This high transistor density results in higher power consumption and heat dissipation beyond the capabilities of the current battery and cooling technologies. Thus, there is a shift in processor design to the multi-core and many-core architectures to overcome these problems.

The many-core architecture is built from hundreds to thousands of small cores. Even though each core delivers less performance than a large monolithic processor, a higher throughput is achieved by the whole system and this is much more important than raw processor performance. The backbone of the many-core architecture is a Network-on-Chip that carries the data/communication traffic between cores, I/O, and memory.

A many-core architecture is also a much more suitable design paradigm to reduce energy. The workload is usually a group of different tasks with different computational requirements distributed in a non-uniform fashion during their execution. Some of these tasks are not critical and they can be slowed down. Thus, energy consumption can be managed by reducing the supply voltage of cores running non-critical workload or switching off idle cores completely to reduce the overall dissipated energy.
Process, Voltage and Temperature (PVT) variations are another obstacle that can significantly impact integrated circuits in small featured technologies. These variations are expected to increase as technology scales down [20]. In a many-core architecture, core speeds vary from core to core due to process variation during fabrication. Within die process variations cannot be predicted before fabrication and may be different from die-to-die on a given wafer. On-chip voltage and temperature variations, on the other hand, are dependent on the activity on the chip and can be managed to a large extent by workload allocation.

The high demand for hand-held and portable devices and the wide gap between battery technology and power consumption make energy reduction a vital step towards the wide-spread use of many-core architectures in mobile computing. One of the effective methodologies to reduce power consumption is the use of multiple supply voltage design. Cores in the system that are executing non-critical workload are supplied voltages below the nominal voltage to save on power.

Power saving strategies must also consider PVT variations. The problems caused by these variations can be performance degradation well below its target at design time, reliability and thermal issues, and increased energy consumption beyond its designated energy budgets. Thus, power saving techniques must be developed with the goal of mitigating the impact of PVT variations.

6.2. Research Contributions

The energy optimization flow described in this thesis is a promising step towards resolving many of the issues described above. The proposed design flow is an effective approach that reduces the overall energy consumption compared to previous methods.
Furthermore, our proposed approach is PVT-aware which increases its significance since PVT is projected to increase as technology scales down.

The main contributions of this thesis can be summarized as follows:

- To the best of our knowledge, this work is the first to include process voltage and temperature variations impact in energy-aware optimization flow for application mapping on many-core architectures.
- This work is the first to investigate the relation between the impact of within-die process variation and communication to determine the shape of voltage islands.
- A fast Voltage Selection approach, *Removal Cost Method*, which provides near optimal solution and it is 10 times faster as compared to dynamic programming. This algorithm can be used in any voltage island design flow, for instance SoCs, MPSoCs, or multi-core platforms.
- A voltage island formation and placement algorithm, *Voltage Island Cloud*, by which cloud-based clusters of cores are grown from selected seeds, or point of origins. The cores are placed around the seed using a weighting model to balance the distance between the island local seed and the seeds of other islands they communicate with. The islands are placed to reduce the impact of PVT as well. The proposed *Voltage Island Cloud* placement algorithm can be utilized in the two voltage island design styles: Static Voltage Scaling, and Multiple Voltage Scaling.
- A complete flow was developed to create voltage islands on many-core platforms starting from application task graphs using the new techniques to reduce the energy by 8-30% compared to previous methods. The proposed techniques were
analyzed based on the application characteristics, such as parallelism, communication, critical path length, to identify the effectiveness of the proposed technique based on application characteristics such that communication load, parallelism, and critical path length.

6.3. Research Observations and Limitations

Application-mapping onto many-core architectures has many parameters that influence the final overall energy consumption. One of these parameters is the voltage island size. The number of cores of an island is determined by the application size. If the island is too large then the impact of PVT on cloud-based islands or rectangular islands is similar. This is because if the cloud-based island spans the whole chip then the distances between cores are too large and speed discrepancy will exist anyway.

The characteristics of the application also impact the energy savings obtained from using cloud-based islands. For instance, for an application with high parallelism and very low traffic, i.e., “embarrassingly parallel” applications [141], both cloud-based and rectangular islands produce similar energy savings. Also, an application with very low parallelism and high traffic, i.e., a serial application, also produces similar energy results regardless of whether rectangles or clouds are applied.

Traditionally, power rails used to deliver the supply voltages to the cores are built to cover rectangular-shaped islands [7][28][29][30][99]. The impact on power grid design when building islands with irregular shapes needs to be analyzed and then quantified to estimate the complexity of the power grid routing and an analysis of the maximum voltage drop for the proposed island formation.
6.4. Future Work

This thesis opens the door to many follow-up studies. In this section, we discuss the near- and long-term future work.

6.4.1. Near-Term Future Work

The complexity of the power grid design has to be evaluated more thoroughly and included in the voltage island methodology. It has to be investigated for both SVS and MVS power grid styles. Currently, only the average voltage drop in the power grid is included during simulation due to the limitation in the current voltage variation model. IR drop and dynamic voltage variation should be measured along with dynamic noise in the power grid network. Then, techniques such as decoupling capacitance allocation for the new proposed island shapes to counter supply voltage drop should be studied and implemented.

The impact of process variation on the static power can be measured in each of the proposed techniques. Process variation models that provide information about threshold variation can be used to evaluate the delay and the static power values. Then, methods to reduce its effect, such as multi-threshold voltage, can be used to reduce the static power consumed during idle periods.

The communication routing optimization is performed using a genetic algorithm in the current energy flow. Other routing optimization methods should be tried such as XY-routing to reach optimized mapping in less runtime.
6.4.2. Long-Term Future Work

Heterogeneous many-core architectures should be investigated as a follow-up to the work described here. In particular, the placement of specialized cores has to be studied. In a small multi-core, the location might not be an issue due to small routing distances. In a many-core architecture, however, the routing distances can be ten times higher. Thus, new strategies for the placement of specialized cores in the platform have to be developed. A heterogeneous many-core might include cores such as configurable cores, memory blocks, DSP blocks, and graphics engines in the many-core architecture. These special cores have to be positioned so that they are easily accessed by other cores and their communication traffic is optimized. The underling architecture can be an influencing factor on the power and performance of such systems. Then, power and performance optimization techniques have to be improved to optimize the application mapping on architectures with different types of cores.

Different network architectures and topologies can also be explored. A combination of dedicated networks for neighbouring cores, combined with global network to connect different groups of cores can be helpful to separate local and global traffic. For example, each group of cores can share a cache via a dedicated bus. This scenario can help reduce the memory traffic in the global network. Furthermore, special routing scenarios have to be examined to identify possible deadlocks during routing that might occur in many-core and study methods to resolve such problems for each of the proposed techniques.

Memory bandwidth in such a massive architecture is another domain that has to be investigated. The location of cache between cores, dedicated versus shared cache
among cores, and memory-dedicated network versus shared network are all topics that need to be pursued for the many-core architectures. Previous techniques used for small multi-cores can be evaluated and improved to handle large number of cores.

Many-core architectures also demonstrate important advantages as a fault-tolerant architecture. Redundant cores can be used to replace faulty ones. In effect, not all the cores are used at the same time. Thus, core consuming high power, e.g., leaky cores due to low threshold voltage as a result of process variations, can be replaced by ones with a higher threshold voltage. Other cores in the platform can be used for on-chip verification. The integration of large number of cores may require techniques to solve potential problems after fabrication. Thus, some cores can be used as on-chip monitors that can report speed and power variations of cores. The feedback from these monitor can go to the operating system to use only working cores.

The adoption of the proposed approach by industry requires building an actual many-core prototype containing voltage islands created by the methods described here. This is the only way to fully validate the approach. Also a wider range of real applications must be mapped onto many-core architectures fabricated at 45nm process technology or beyond. Providing power saving numbers of real architectures by the research community will likely have a significant impact on the industry. In addition, the software development tools and systems to integrate hardware and software are also needed before widespread adoption takes place. However, it is clear that this many-core architecture will be a major thrust in academia and industry in the years ahead.
References


142


[135] ARM, "Cortex-A8 Processor - ARM."


