MACHINE LEARNING-INSPIRED HIGH-PERFORMANCE AND ENERGY-EFFICIENT HETEROGENEOUS MANYCORE CHIP DESIGN

By

WONJE CHOI

A dissertation submitted in partial fulfillment of the requirements for the degree of

DOCTOR OF PHILOSOPHY

WASHINGTON STATE UNIVERSITY
School of Electrical Engineering and Computer Science

MAY 2018

© Copyright by WONJE CHOI, 2018
All Rights Reserved
To the Faculty of Washington State University:

The members of the Committee appointed to examine the dissertation of WONJE CHOI find it satisfactory and recommend that it be accepted.

Partha Pratim Pande, Ph.D., Chair

Deuk Heo, Ph.D.

Janardhan Rao Doppa, Ph.D.
ACKNOWLEDGMENT

I thank my Ph.D advisor and mentor, Prof. Partha Pande who inspired me to pursue computer engineering. His guidance over the last five years helped me in all the time of research. Without his valuable support it would not be possible to conduct this research.

I thank Prof. Deuk Heo for being a member of my Ph.D. committee and providing precious feedback.

I sincerely thank all the co-authors of my publications. In particular, I am grateful to Prof. Jana Doppa for his insightful comments and encouragement.

I thank my friends, Dr. Karthi Duraisamy and Dr. Ryan Kim for being wonderful colleagues. I will never forget the sleepless nights we were working together before deadlines.

Lastly, I thank my fellow lab-mates for all the fun we have had in the last five years.
Abstract

by Wonje Choi, Ph.D
Washington State University
May 2018

Chair: Partha Pratim Pande

CPU-GPU based heterogeneous manycore architecture is expected to be the dominant computing platform for many emerging application domains including artificial intelligence, bio-computing and big data analytics. This heterogeneous system typically consists of multiple CPU cores, many GPU cores and a few shared Last-Level-Cache (LLC) units and exhibit challenging on-chip traffic patterns with various QoS requirements. The on-chip communication architecture for heterogeneous manycore systems should be designed such that it efficiently handles the various traffic requirement simultaneously. With the incorporation of different types (CPUs, GPUs, accelerators, etc.) of cores, it becomes more difficult to efficiently explore the combinatorial design space of heterogeneous systems. In order to uncover high-quality designs, new methods need to be developed that can quickly search the design space with ever increasing system size.

Aside from on-chip communication architecture, the design of heterogeneous manycore platforms is dominated by power and thermal constraints. In this respect, voltage-
frequency island (VFI) is a promising design paradigm to create scalable energy-efficient platforms. By dynamically tailoring the voltage and frequency of each island, we can further improve the energy savings within given performance constraints. Traditional DVFS techniques operate on a core-by-core basis and use core-level information (e.g., core utilization and communication) to tailor the V/F values of each individual core. These traditional DVFS techniques were applied using the combined information from all cores within the VFI, i.e., the VFI’s average core utilization and communication. However, simple averages may not capture the information required to accommodate every core, router, and link within a VFI, particularly for VFIs with large intra-VFI workload variance.

In this dissertation, we undertake above-mentioned problems of designing efficient heterogenous manycore architectures. First, we propose a hybrid Network-on-Chip architecture consisting of both wireline and wireless links that can seamlessly handle the varied traffic requirements that arise in heterogeneous manycore platforms. Second, we develop a machine learning-based multi-objective optimization (MOO) algorithm that learns an evaluation function and guides the search toward optimal designs in heterogeneous manycore systems. Finally, we propose architecture-independent imitation learning-based methodology for dynamic VFI control in heterogeneous manycore systems to address power and thermal issues.
# TABLE OF CONTENTS

<table>
<thead>
<tr>
<th>Chapter</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>ACKNOWLEDGMENTS</td>
<td></td>
<td>iii</td>
</tr>
<tr>
<td>ABSTRACT</td>
<td></td>
<td>iv</td>
</tr>
<tr>
<td>LIST OF TABLES</td>
<td></td>
<td>vi</td>
</tr>
<tr>
<td>LIST OF STATISTICS</td>
<td></td>
<td>ix</td>
</tr>
<tr>
<td>1</td>
<td>Introduction</td>
<td></td>
</tr>
<tr>
<td>1.1</td>
<td>Interconnect Architecture for Heterogeneous Platforms</td>
<td></td>
</tr>
<tr>
<td>1.2</td>
<td>Design Optimization for Heterogeneous Platforms</td>
<td></td>
</tr>
<tr>
<td>1.3</td>
<td>Power Management Strategies for Manycore Platforms</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>Related Works</td>
<td></td>
</tr>
<tr>
<td>2.1</td>
<td>Interconnect Architecture for Deep Convolutional Network Applications in Heterogeneous Platforms</td>
<td></td>
</tr>
<tr>
<td>2.2</td>
<td>Design Optimization Approaches for Heterogeneous Systems</td>
<td></td>
</tr>
<tr>
<td>2.3</td>
<td>Dynamic Voltage Frequency Scaling for Voltage Frequency Island</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>CPU-GPU Heterogeneous Manycore Architectures for Deep Learning Applications</td>
<td></td>
</tr>
<tr>
<td>3.1</td>
<td>Deep Learning Applications: Convolutional Neural Networks</td>
<td></td>
</tr>
<tr>
<td>3.2</td>
<td>NoC Architecture for Heterogeneous Platforms</td>
<td></td>
</tr>
<tr>
<td>3.3</td>
<td>Experimental Results and Analysis for Hybrid NoC Architecture</td>
<td></td>
</tr>
</tbody>
</table>
3.4 Machine Learning-based Design Optimization Framework for Heterogeneous Manycore Systems ................................................................. 47

3.5 Summary ...................................................................................................... 65


4.1 Problem Formulation .................................................................................. 68

4.2 Baseline DVFI Control Policies .................................................................... 71

4.3 Imitation Learning-based Approach .............................................................. 77

4.4 Experimental Results .................................................................................. 85

4.5 Summary ...................................................................................................... 102

5 Future Research Directions ............................................................................ 104

5.1 Efficient Deep Convolutional Neural Network Inferencing for Embedded MPSoC Platforms ........................................................................ 104

5.2 Application Specific Dynamic Power Management for Mobile Platforms .......... 107

References ......................................................................................................... 109
LIST OF TABLES

Table 3.1 Layer configurations for LeNet and CDBNet .......................................................... 16

Table 4.1 VFI sizes for heterogeneous clustering................................................................. 87

Table 4.2 Hardware Overhead of DVFI Policies ................................................................. 96
# LIST OF FIGURES

<table>
<thead>
<tr>
<th>Figure</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>Figure 3.1</td>
<td>High-level overview of the CNN architecture for digit classification task.</td>
<td>14</td>
</tr>
<tr>
<td>Figure 3.2</td>
<td>Illustration of a heterogeneous architecture incorporating CPUs, GPUs, and MCs.</td>
<td>19</td>
</tr>
<tr>
<td>Figure 3.3</td>
<td>WiHetNoC design flow.</td>
<td>22</td>
</tr>
<tr>
<td>Figure 3.4</td>
<td>Illustration of WiHetNoC Connectivity.</td>
<td>27</td>
</tr>
<tr>
<td>Figure 3.5</td>
<td>Message injection rate for training the (a) LeNet and (b) CDBNet.</td>
<td>31</td>
</tr>
<tr>
<td>Figure 3.6</td>
<td>Traffic breakdown for each layer of the (a) LeNet and (b) CDBNet CNNs.</td>
<td>32</td>
</tr>
<tr>
<td>Figure 3.7</td>
<td>Snapshot of the memory access temporal locality during the forward pass of (a) convolution and (b) pooling layers running LeNet on the proposed heterogeneous architecture.</td>
<td>33</td>
</tr>
<tr>
<td>Figure 3.8</td>
<td>The central $6 \times 5$ portion of the optimized Mesh NoC indicating the link utilizations and the locations of CPUs and MCs for LeNet computations.</td>
<td>34</td>
</tr>
<tr>
<td>Figure 3.9</td>
<td>Traffic-weighted hop count and standard deviation among the link utilizations for the optimized mesh (denoted as OPT) and WiHetNoCs (for each considered $K_{max}$). For mesh NoCs we show the results for both XY and XY+YX routing schemes.</td>
<td>36</td>
</tr>
<tr>
<td>Figure 3.10</td>
<td>Normalized traffic-weighted hop count and standard deviation among the link utilizations of various candidate wireline configurations for WiHetNoC.</td>
<td>38</td>
</tr>
<tr>
<td>Figure 3.11</td>
<td>Variation in network EDP for different router port upper bounds ($K_{max}$).</td>
<td>39</td>
</tr>
<tr>
<td>Figure 3.12</td>
<td>EDP and Wireless utilization for various WI counts.</td>
<td>39</td>
</tr>
<tr>
<td>Figure 3.13</td>
<td>EDP and WI utilization with various number of channels.</td>
<td>40</td>
</tr>
</tbody>
</table>
Figure 3.14 CPU-MC communication latency and overall NoC throughput for optimized mesh NoC and the optimized WiHetNoC. ................................................................. 41

Figure 3.15 Cumulative Distribution Function (CDF) of link utilizations for the WiHetNoC, and optimized mesh architectures. .............................................................................. 42

Figure 3.16 Asymmetric nature of WI utilization for training the (a) LeNet and (b) CDBNet. ........................................................................................................................................... 43

Figure 3.17 Normalized network EDP for training the (a) LeNet and (b) CDBNet. .......... 44

Figure 3.18 Normalized network latency for training the (a) LeNet and (b) CDBNet. ........ 44

Figure 3.19 Normalized execution time and full system EDP for training LeNet and CDBNet. .............................................................................................................................................. 46

Figure 3.20 Illustration of proposed heterogeneous architecture. ................................. 48

Figure 3.21 Algorithm for Local Search: local (Obj, dstart) ........................................... 55

Figure 3.22 Algorithm for MOO-STAGE. ........................................................................... 57

Figure 3.23 (a) PHV for AMOSA and MOO-STAGE, (b) evaluation function prediction error of MOO-STAGE, and (c) the best temperature delay product (TDP) ......................... 60

Figure 3.24 Speed-up achieved by MOO-STAGE over AMOSA. ........................................ 61

Figure 3.25 Pareto fronts for three-objective MOO at T_MOO-STAGE ............................. 62

Figure 3.26 Core and link distribution of 3DHet using (a) MOO-STAGE and (b) AMOSA. .. 63

Figure 3.27 Normalized full-system EDP with respect to 3DMesh. .................................. 64

Figure 4.1 An overview of the DVFI control policy creation and evaluation ............... 69

Figure 4.2 Overview of the feedback control approach to DVFI ..................................... 71

Figure 4.3 Overview of the reinforcement learning approach to DVFI ............................ 73
Figure 4. 4 Overview of the imitation learning approach to DVFI.............................................. 77
Figure 4. 5 Illustration of DVFI decision-making as a structured prediction task.................. 81
Figure 4. 6 Pseudo-code of Dagger......................................................................................... 84
Figure 4. 7 Illustration of the DVFI controller learning mechanism with DAgger.................. 85
Figure 4. 8 Energy and Execution Time penalty (w.r.t. nominal operation) trade-offs as a
function of λ for (a) FFT, and (b) CANNEAL............................................................................. 90
Figure 4. 9 Testing error and area overhead trade-off as a function of QG (Normalized to QG
3)........................................................................................................................................... 91
Figure 4. 10 EDP improvement of the Oracle with increasing values of N............................ 93
Figure 4. 11 Mean average percentage error (MAPE) between the predicted and Oracle/vue for Exact Imitation and DAgger versions of the RT and LR policies............................ 95
Figure 4. 12 High-level DVFI controller structure................................................................. 96
Figure 4. 13 Normalized computation time w.r.t. IL to create the DVFI policies.................. 98
Figure 4. 14 (a) Execution time and (b) energy dissipation of the DVFI policies w.r.t. the FB
policy........................................................................................................................................ 99
Figure 4. 15 execution time and energy dissipation tradeoff for each of the DVFI policies for
(a) FFT and (b) FLUID.............................................................................................................. 100
Figure 4. 16 DVFI comparison in (a) Computation overhead for learning DVFI policy vs. full-
system energy for a fixed 5% allowable execution time penalty, and (b) DVFI controller area
overhead...................................................................................................................................... 101

Figure 5. 1 Normalized number of images classified in each layer........................................ 104
Figure 5. 2 Illustration of early classification approach.......................................................... 105
Dedication

I dedicate this dissertation to my mother, SunYoen Kim who helped me in all things great and small.
1 Introduction

The demand for high-performance, low power computing has grown in recent years due to emerging applications, such as neural networks, graph analytics, and other big data applications. Heterogeneous manycore systems allow large system integration with many diverse tiles (e.g., CPUs, GPUs, accelerators) that look very promising for such data- and compute-intensive applications. However, there are several limitations associated with designing these heterogeneous manycore systems. First, typical on-chip communication infrastructures employed in conventional homogeneous manycore systems are unable to handle various communication requirements arising from the heterogeneous architectures. In order to harvest the full potential of the heterogeneous architectures, it is necessary to use innovative on-chip communication infrastructures. Second, the size and complexity of the overall design space has grown with increasing system size, greater tile diversity, and multiple design objectives due to the emerging applications. Consequently, the time and effort required for designing these heterogeneous architectures increase exponentially. Lastly, high-power consumption of these manycore chips is a critical concern for continued scaling. Indeed, higher system power not only raises the chip temperature and cooling costs, but also decreases chip reliability and performance.

Machine learning techniques have seen great success in diverse application domains, particularly solving complex optimization problems. Subsequently, in this dissertation, we
address the above-mentioned problems in designing heterogeneous manycore systems by leveraging machine learning techniques.

1.1 Interconnect Architecture for Heterogeneous Platforms

As traditional homogeneous manycore systems hit the power wall, system designers had to find innovative ways to improve performance. In recent years, systems with a heterogeneous set of cores, where some cores are significantly more efficient at certain computational tasks, have been employed to enhance performance. Thus, CPU-GPU heterogeneous architectures are expected to be the dominant computing platforms for many emerging application domains, e.g., artificial intelligence, bio-computing, and big data analytics. By incorporating GPU cores, these systems have the potential to accelerate emerging applications by leveraging the GPU’s highly data-parallel and throughput-oriented nature. Furthermore, by integrating CPU and GPU cores onto a single die, seamless data exchange between CPUs and GPUs can be achieved, eliminating the need for explicit memory copies [1].

Typical on-chip communication infrastructures employed on conventional manycore platforms are not suitable to handle the varied traffic requirements of CPUs and GPUs efficiently. It has already been shown that in conventional heterogeneous systems, the shared memory resources are often monopolized by the GPUs, leading to significant degradation in CPU memory access latency and high execution time penalties [2]. To address this issue, we first analyze the on-chip traffic patterns that arise from the computational processes associated with executing applications. By leveraging this knowledge, we design a
hybrid Network-on-Chip (NoC) architecture, which consists of both wireline and wireless links, to improve the performance of CPU-GPU based heterogeneous manycore platforms running diverse application domains.

1.2 Design Optimization for Heterogeneous Platforms

With increasing system size and diversity among cores that can be integrated together (CPU, GPU, accelerators, Memory blocks, etc.) the overall design space grows rapidly. Each core has its unique requirement to deliver high-performance e.g. CPUs require lower latency whereas GPUs are throughput-sensitive. Naturally, the NoC design for these heterogeneous systems can be formulated as a multi-objective optimization (MOO) problem. To solve this optimization problem, one could employ search algorithms, e.g., hill-climbing and simulated annealing (SA), which are very popular in the design community. Unfortunately, these search algorithms do not leverage the knowledge about past explored designs to guide future explorations. As a result, they scale poorly as the number and types of cores increases.

Commonly employed algorithms for MOO problems include Simulated Annealing (SA), AMOSA [4], NSGA-II[5], etc. These algorithms execute multiple unguided, independent searches from different starting points to increase the chances to reach global optima. As the number of objectives involved in the MOO increases, the search space becomes exponentially larger. A large number of independent searches are required to cover the increased search space. Hence, conventional MOO algorithms require significant amount of time to find an acceptable solution as the number of objectives increase. Motivated by the need for
scalability and efficiency in design optimization, we propose to leverage machine learning techniques to learn the knowledge needed to perform guided design space exploration to achieve computational efficiency [6].

1.3 Power Management Strategies for Manycore Platforms

Large-scale manycore chips are employed to solve many compute- and data-intensive applications. However, high-power consumption of these manycore chips is a critical concern for continued scaling. Indeed, higher system power not only raises the chip temperature and cooling costs, but also decreases chip reliability and performance. In order to deal with the continually expanding core count in manycore systems, voltage–frequency islands (VFIs) have emerged as an efficient and scalable power management strategy [7][8]. In such designs, effective VFI clustering techniques allow cores and network elements (routers and links) that behave similar to share the same voltage/frequency (V/F) values without significant performance penalties. Indeed, VFIs present a scalable power management strategy that can reduce the energy consumption and hardware overhead while maintaining a specified level of performance. Naturally, with time-varying workloads, we can dynamically fine-tune the V/F levels of VFIs to further reduce the energy dissipation with minimal performance degradation; to this end, we consider dynamic voltage and frequency scaling (DVFS) techniques for VFIs. However, traditional DVFS techniques operate on a core-by-core basis and use core-level information (e.g., core utilization and communication) to tailor the V/F values of each individual core. In [9], these traditional DVFS techniques were applied using the combined information from all cores within the VFI, i.e., the VFI’s average core utilization and communication. However, simple averages may not
capture the information required to accommodate every core, router, and link within a VFI, particularly for VFIs with large intra-VFI workload variance. In this dissertation, we argue that for applications with highly varying workloads, machine learning (ML) methods are suitable to fine-tune the V/F levels within VFIs.

Rest of this dissertation is organized in the following way: Chapter 2 discusses the prior work related to topics of this dissertation. In Chapter 3, we discuss a design optimization framework for CPU-GPU-based heterogeneous manycore systems. Chapter 4 analyzes multiple machine learning-inspired dynamic power management strategies. Lastly, Chapter 5 summarizes the salient features of this work and discusses the future directions originating from this dissertation.
2 Related Works

2.1 Interconnect Architecture for Deep Convolutional Network Applications in Heterogeneous Platforms

Convolution Neural Network (CNN) technology has seen success in a wide-range of applications including computer vision (e.g., image classification [10] and video classification [11]); speech processing [12]; natural language processing (e.g., text classification [13] and learning semantic representations for web search [14]); drug design [15]; and to represent policies and value functions for driving the Monte Carlo tree search approach employed in AlphaGO [16].

NoC-enabled homogeneous CMP architectures targeting neuroscience applications have already been explored. For instance, a massively parallelized CMP platform incorporating a customized NoC architecture was used to implement spiking neural networks [17]. Multicast-aware mesh NoC architectures have been proposed for reconfigurable neural networks [18][19]. However, due to the highly data-parallelizable nature of the neural networks, these applications have already been demonstrated to be more efficient on discrete GPU systems rather than traditional multi-CPU CMPs [20][21]. The design of a commodity off-the-shelf system for HPC targeting deep learning was proposed recently [22]: a cluster of GPU servers with Infiniband interconnects and a message-passing interface show promise over large CPU-only based systems. Recently, the architecture of a machine learning “supercomputer” [14] that achieves higher performance and lower energy
dissipation than a modern GPU-based system was also proposed. The system relies on a multichip design, where each node is significantly cheaper than a typical GPU while achieving comparable or higher number of operations per unit time in a smaller package. A recent study characterized the performance of two popular CNN architectures (i.e., LeNet, CDBNet) on various embedded platforms including FPGAs, DSPs, and GPUs [23]. The main limitation of [23] is that it mainly focuses on optimizing the software parallelization and mapping, without a detailed on-chip network analysis required for an efficient manycore-based implementation.

Prior works on discrete GPU platforms have focused on improving the system performance by enhancing their NoC architectures [24][25][26]. In a GPU system, processes executed in each GPU core are usually independent of the other GPUs’ processes, resulting in low inter-GPU communication [24][28]. Typically, GPUs only communicate with a few shared memory controllers (MC), causing a many-to-few traffic pattern (i.e., many GPU cores communicating with a few MCs) [24][25][26]. In this case, MCs can potentially become traffic hotspots and lead to performance bottlenecks. Prior research demonstrated that suitable placement of the MCs can help alleviate the associated traffic congestion [25][26]. To prevent traffic imbalance among the links, a checkerboard mesh NoC with a suitable routing strategy was recently proposed [24]. Applications running on GPUs tend to exhibit an asymmetric traffic pattern, where the amount of communication from L1 to MC is usually lower than that from MC to L1 due to memory coalescing in GPU architectures [26]. This asymmetric nature further worsens the performance bottlenecks that are exhibited under the many-to-few traffic pattern. To overcome this issue, an asymmetric virtual channel partitioning and
monopolization technique for discrete GPU NoCs was proposed [25]. The advantage of using a clustered mesh NoC (with four L1s per cluster) over non-clustered mesh and crossbar architectures (i.e., all L1s in a single cluster) for discrete GPU systems was demonstrated [26].

As explained above, training CNNs involves heavy CPU-GPU communication that is best suited for a NoC-enabled heterogeneous CPU-GPU CMP platform rather than a traditional discrete-GPU system with expensive off-chip CPU-GPU data transfers [27]. Due to the differences in the thread-level parallelism of CPUs and GPUs, the NoC employed for heterogeneous systems is expected to handle traffic patterns with varying Quality of Service (QoS) constraints [28]. CPUs are highly sensitive to the memory access times and hence, communications involving CPUs require low-latency data exchanges. On the other hand, GPU communication demands high bandwidth [28]. Modern NoC designs for discrete GPU systems typically attempt to only maximize the overall bandwidth of the system. Consequently, these NoC designs are unsuitable for heterogeneous CMP architectures incorporating multiple CPUs and GPUs on the same die. It has been shown that the shared memory resources in a heterogeneous system are often monopolized by the GPUs, leading to significant degradation in CPU memory access latency and high execution time penalties [29]. Hence, an efficient on-chip network designed for heterogeneous CMPs should balance the inter-core traffic and comply with different QoS standards required for both CPU and GPU communications.

The NoC design for CPU-GPU heterogeneous systems have yet to be studied thoroughly. A system-level discussion regarding the NoC design for CPU-GPU heterogeneous architectures was presented recently [28]. However, this work only considered a ring
interconnect that is known to be inefficient for large-scale systems. A wireless NoC was proposed for heterogeneous systems [30]. However, this work does not provide any detailed analysis of the on-chip traffic patterns and has not considered the nature of traffic patterns that is inherent in CPU-GPU heterogeneous platforms (e.g., many-to-few data exchanges) while designing the wireless NoC. A virtual channel partitioning scheme was proposed to achieve low-latency CPU related memory accesses in the presence of largely latency insensitive GPU communication [31]. However, this strategy can affect the GPU throughput due to the partitioning of physical resources. As we will explain later, our work employs dedicated wireless links for CPU-MC communication to avoid this network contention and hence, ensures a high GPU-MC throughput.

Thus far, most of the NoCs targeting discrete GPU systems are based on conventional wired NoC architectures. The achievable performance benefit from the proposed strategies is restricted due to the inherent limitations associated with these NoCs. Traditional wireline NoCs (such as mesh) use multi-hop, packet-switched communication that lead to high network latencies [32]. To overcome these limitations, small-world network-inspired wireless NoC architectures have been proposed [33][34]. Indeed, by employing a few long-range wireless shortcuts, these architectures enable low latency communication even among the computing cores that are physically far apart. Previous works [35][36][37] have already investigated the feasibility of the on-chip wireless communication. The viability of on chip wireless communication has been demonstrated through prototypes [38]. In addition, a recent study on emerging on-chip interconnects concluded that the Radio Frequency (RF) links, e.g., mm-wave wireless and surface wave interconnects, are more power and cost
efficient than on-chip optical links [39]. Between the two RF interconnects, on-chip wireless technology is more mature and fully CMOS compatible [34][39].

While designing NoCs enabled by wireless links, it should be noted that the number of wireless interfaces (WIs) that can be added in a NoC is usually limited by number of factors. A couple of such factors are: (1) the nature of the medium access control (MAC) protocol employed, and (2) the area and power overhead allowed for the addition of WIs. Hence, it is necessary to use innovative system architecture designs to harvest the full potential of wireless-enabled NoCs [40]. Towards this goal, in this work, we improve the state-of-the-art by presenting a hybrid (wireline + wireless) on-chip interconnection architecture that can meet the communication demands uniquely present in a heterogeneous CMP platform consisting of both CPU and GPU cores.
2.2 Design Optimization Approaches for Heterogeneous Systems

Naturally, the NoC design for these heterogeneous systems can be formulated as a multi-objective optimization (MOO) problem. To solve this optimization problem, one could employ search algorithms, e.g., hill-climbing and simulated annealing (SA), which are very popular in the design community. However, these approaches are not enough to quickly find high-quality solutions due to their poor scalability and excessive amount of execution time.

Heuristics and machine learning-based approaches also have been proposed to search the design space of manycore systems [41][42]. However, these works are based on conventional homogeneous platforms, and design space exploration for CPU-GPU based heterogeneous platforms is significantly more complex and is yet to be studied thoroughly. In [4], the authors demonstrate that AMOSA can outperform NSGA-II [5] in the number of distinct solutions, time needed, and overall performance. However, these existing approaches do not leverage the knowledge gained from past design space exploration. Along this line, work in [6] proposes a machine learning technique called STAGE that aims to learn a function that can evaluate the potential of a starting state to improve the accuracy of local searches. However, STAGE can only handle single-objective optimization problems, e.g., the number of bins in bin-packing or the area in channel routing. In this work, we advance the state-of-the-art by proposing a machine learning inspired MOO algorithm that can efficiently search the design space for heterogeneous manycore systems.
2.3 Dynamic Voltage Frequency Scaling for Voltage Frequency Island

The VFI paradigm has become the design of choice when attempting to minimize the energy consumption with small performance degradation and area overhead [43][44]. The design of a VFI-based system has been presented in [45]. However, most prior work considers only static V/F allocation, which is suboptimal in the presence of highly varying workloads [45][46]. Workload variation can be exploited by dynamically adjusting the V/F of a VFI-based manycore system. DVFI control in a VFI has been demonstrated in prior work [47][48], but the focus has been on the network-on-chip (NoC) and control of inter-VFI queue occupancy as opposed to a FS performance evaluation. Hardware-based control has been demonstrated in the context of application-specific systems [49], but the focus was mainly on the inter-VFI queue without considering NoC-enabled systems. Howard et al.[50] have designed a chip with DVFS and fine-grained V/F domains; however, our target in this work is much coarser-grained VFIs. In [51], by using heuristic-based DVFI approaches, the authors have demonstrated that moving to coarser-grained control (both in the number of V/F levels available and VFI size) does not significantly reduce the efficacy over single-core DVFS. A feedback-loop VFI-controller design methodology that considers the time-varying nature of the workload has been proposed [52]. Similarly, mixed integer linear programming was employed for optimal V/F allocation of a VFI-enabled system [53]. However, this approach uses a randomized rounding-based heuristic and tries to solve multiple instances of a modified integer linear programming problem, resulting in slower speed. The first work on full-system DVFI has been presented in [54], but relies purely on simplistic heuristics and high-level VFI information that may lead to suboptimal decisions for all cores in the VFI. This
particular work [54] serves as the basis for our current platform and represents the baseline DVFI policy. ML techniques have been recently used for system level dynamic power management [55][56][57]. However, these works consider coarse-grained system-level DVFS that is not intended for multiple V/F domains and fine-grain (sub millisecond) DVFS control. There have also been some investigations regarding ML-based core-level DVFS [58][59]. In these prior works, the authors mainly study per-core DVFS, but these ideas can be adapted to DVFI as well. For example, per-core RL in conjunction with global power budget reallocation has been proposed before [60]. However, these prior solutions have three main drawbacks. 1) Consider learning independent controllers that do not take into account the inherent dependencies between VFIs. In contrast, we learn controllers by leveraging the structural relationships between VFIs [61]. 2) Employ online RL algorithms, which require workload stationarity, suffer from (exponential) state explosion, and have weak theoretical guarantees for convergence to the optimal policy. In contrast, we develop a new approach based on IL that overcomes these deficiencies and comes with strong theoretical guarantees (i.e., highly accurate base predictor implies a very good DVFI control performance) [62]. 3) Use look-up tables to store the learned control policy, which would have large hardware overhead for large manycore chips, as targeted by our work. In comparison, we employ efficient representations of the learned control policies that have negligible hardware overhead. Taken together, we improve the state of the art by proposing a new dynamic V/F tuning methodology for VFI-partitioned systems based on IL to improve the energy dissipation of a manycore chip within a target execution time constraint.
3 CPU-GPU Heterogeneous Manycore Architectures for Deep Learning Applications

3.1 Deep Learning Applications: Convolutional Neural Networks

CNN is a class of deep neural network that has widely applied to many applications, including image and video recognition, recommender systems and natural language processing. In this dissertation, we consider CNN architectures employed for image recognition task as an example.

3.1.1 Overview

CNNs usually perform prediction/classification tasks (e.g., finding the output probability scores of different class labels for a given input image) where the inputs and the outputs are fixed size vectors. A typical CNN architecture consists of a stack of alternating convolutional and pooling layers, followed by a fully-connected layer at the end. Figure 3.1 is a high-level overview of the CNN architecture for digit classification task.

![Figure 3.1 High-level overview of the CNN architecture for digit classification task.](image-url)
conceptual diagram illustrating a CNN architecture performing hand-written digit classification.

Convolutional layer: Convolutional layers are the key building blocks of a CNN architecture and are comprised of sets of learnable filters (or kernels) that aim to transform a low-level representation (e.g., raw pixels of an image) into a high-level representation (e.g., edges/sections in the image). During the forward pass, each filter is convolved by sliding over the input to generate a feature map. Convolutional layer computations are commonly followed by an element-wise non-linear activation function (e.g., Rectified Linear Unit (ReLU) or sigmoid).

Pooling layer: Typically, a pooling layer is employed between two convolutional layers. It aggregates the input representation succinctly using a max or average operator. The purpose of this aggregation is to reduce the volume of computations and parameters associated with the network and to avoid over-fitting. Pooling operations can also be seen as a spatial down-sampling of the input 3D tensor.

Fully connected layer: At the end of the convolutional neural networks, all the outputs of the second to last layer (usually a convolutional or a pooling layer) are connected to the entire output vector, constituting a fully connected layer. Thus, the size of this layer, i.e., the number of neurons, is equal to the number of output classes, e.g., ten in digit classification.

Network weights: The weights or parameters of a given CNN architecture, \( W \), correspond to the different filters at the convolutional layers and the weights of the fully connected layer. It should be noted that the pooling layer does not have any parameters.
Prediction with given weights: Given an input image \( x = [x_{11}, x_{12}, \ldots] \) and network weights \( W = [w_{11}, w_{12}, \ldots] \), we propagate the input forward through the network sequentially through all the layers to compute the output of the network: scores for each class; the highest scoring class is selected as the prediction.

3.1.2 CNN Training Datasets

In this work, we consider two of the most widely used image recognition datasets, namely MNIST and CIFAR-10 [63][64], and train suitable CNN architectures to perform the image classification tasks associated with these two datasets. The MNIST dataset consists of handwritten digits (in greyscale) with 60,000 training and 10,000 testing samples. CIFAR-10 is an object recognition dataset consisting of color images containing one of 10 different object classes with 50,000 training and 10,000 testing examples. We have adopted the LeNet architecture [65] to solve the handwritten digit classification task using the MNIST database. This LeNet architecture consists of an input layer followed by two sets of alternating convolutional and max-pooling layers. These layers are then followed by a convolutional layer, a fully connected layer, and finally, the output layer. We have employed the Convolutional Deep Belief Neural Network (CDBNet) architecture from ConvNet [66] to perform an image classification task using the CIFAR-10 dataset. CDBNet consists of three

<table>
<thead>
<tr>
<th>Dataset (CNN)</th>
<th>Input</th>
<th>1st conv</th>
<th>2nd conv</th>
<th>Last conv</th>
</tr>
</thead>
<tbody>
<tr>
<td>MNIST (LeNet)</td>
<td>33x33x1</td>
<td>29x29x16 5x5 kernel</td>
<td>11x11x16 5x5 kernel</td>
<td>1x1x128 5x5 kernel</td>
</tr>
<tr>
<td>CIFAR-10 (CDBNet)</td>
<td>31x31x3</td>
<td>31x31x32 5x5 kernel</td>
<td>15x15x32 5x5 kernel</td>
<td>7x7x64 5x5 kernel</td>
</tr>
</tbody>
</table>
convolution layers followed by a local normalization layer, a max-pooling layer, two average-pooling layers, and one fully connected layer. We have shown the detailed layer configuration of both LeNet and CDBNet in Table 3.1. The object classification task of CIFAR-10 is more complex than the MNIST image recognition task as it involves colored images and ten different object classes. Hence, compared to LeNet that uses only 16 filters, CDBNet is more complex and employs 64 filters. Moreover, unlike LeNet that uses max-pooling method for all pooling layers, CDBNet employs the average-pooling method for its second and third pooling layers.

3.1.2 Training CNNs

Conventionally, the weights of a CNN are learned from a set of training data using the backpropagation algorithm. Given an input image $x$ and weights $W$, we propagate the input forward through the network to compute the predicted output $\hat{y}$. Given a set of $T$ training input-output pairs $(x_k, y_k)$, the weights of the network are learnt such that the error of the predicted outputs of the neural network (as described above) is minimized.

Typically, an iterative optimization training algorithm (backpropagation) is used to learn the network weights. At each iteration, for each input example $x_k$ from the training data, we make a prediction $\hat{y}_k$ by making a forward pass through the network using the current network weights (as described above); we then compare the predicted output $\hat{y}_k$ with the correct output $y_k$. If there is an error, we attempt to correct it by making a backward pass through the network, compute the gradient vector $\delta_k$, and perform a weight update using the cumulative gradient as follows:
\[\delta = \sum_{k=1}^{T} \delta_k \quad (3.1)\]
\[W = W + \eta \delta \quad (3.2)\]

where \(\eta\) is the learning rate and \(W\) is the weight vector corresponding to all network weights.

It should be noted that during both the forward and backward passes, a lot of data parallelism exist in the computations. In fact, all the neurons in a CNN layer perform the same computation with different input and filter parameters. Hence, this data parallelism can be easily exploited using hardware accelerators (such as GPUs) to greatly accelerate CNN training.

For both forward and backward passes, the convolutional layers contain the most computationally expensive operations. Each convolution operation requires multiple dot products of the filter values and the sectional input vectors. Standard implementations of the convolutional layer mainly involve matrix operations and therefore, leverage efficient libraries such as Basic Linear Algebra Subprograms [67] for GPGPU-based systems. The activations at fully connected layer can also be computed using matrix multiplication.
3.2 NoC Architecture for Heterogeneous Platforms

In CPU-GPU heterogeneous manycore architectures, the NoC primarily handles many-to-few communication patterns. Each processing core (either a GPU or a CPU) in the system maintains its own L1 cache. The L1 caches mainly exchange data with a limited number of MC blocks and the volume of inter-GPU data-exchanges is negligible [24][26]. Each MC incorporates a Last Level Cache (LLC) and a mechanism to access the main memory. Figure 3.2 illustrates a CPU-GPU heterogeneous architecture1 with four CPUs, 28 GPUs, and four MCs. As shown in Figure 3.2, in order to perform efficient many-to-few communication, several GPUs can be attached to a single router (concentration) [26].

In a heterogeneous manycore platform, the QoS requirements for the GPU-MC and CPU-MC communications vary from each other. The CPU-MC communications are primarily latency-sensitive, while the GPU-MC communications are more throughput-sensitive [28]. In CPUs,

![Diagram](image)

**Figure 3.2 Illustration of a heterogeneous architecture incorporating CPUs, GPUs, and MCs. Each tile contains a network router enabling NoC interconnections. Because of L1 concentration, GPU tiles incorporate 4 GPU cores and 4 L1s in each tile.**

---

1It should be noted that the tile locations in this heterogeneous architecture figure are just for illustration purposes. They are not optimized for any specific performance metric.
waiting for memory accesses lead to stalled processor cycles resulting in execution time penalties. On the other hand, GPU’s low-cost context switching makes them less susceptible to GPU-MC communication latency. However, each GPU core consists of multiple thread execution units, requiring large streams of data exchanges between the GPU and MC, leading to high throughput requirements [28]. Considering the above-mentioned facts, a NoC designed for heterogeneous CPU-GPU systems must be optimized to ensure that the CPU-MC communication latency is minimized while the overall NoC throughput is maximized. Hence, in this work, we consider these two objectives and jointly optimize them while designing the overall interconnection architecture.

3.2.1 Mesh NoC

It was shown that on a mesh NoC for handling many-to-few communication patterns, placing the MCs closer to the middle rather than along the chip edges reduce traffic congestion in the links and ensures better overall NoC throughput [25]. However, this mesh NoC was designed for a homogeneous GPU-based system. In the case of heterogeneous architectures, which is the focus of this work, we need to consider the placement of the CPU cores in addition to the placement of the MCs to achieve both low latency CPU-MC communication and high NoC throughput. However, as we will show in Figure 3.8, even in a mesh optimized for both CPU and MC placements, there exist a few links that are heavily utilized when compared to the rest of the links present in the NoC. During high traffic, such links will become bandwidth bottlenecks, negatively affecting the overall system performance. The presence of these bandwidth bottlenecks can be attributed to the many-to-few communication pattern and the
multi-hop nature of the mesh NoC architecture, which leads to high traffic aggregation in the intermediate routers and links.

To address the inherent multi-hop nature of mesh NoCs, the design of wireless NoCs (WiNoCs) have been proposed [33][34]. The salient feature of the WiNoC is that the wireless links establish single-hop shortcuts between physically distant cores, thereby improving the hop-count and subsequently, the latency, throughput, and energy dissipation of the whole system. Hence, we aim to design a customized low hop-count WiNoC targeted for the heterogeneous architecture under consideration.

### 3.2.2 Proposed Hybrid NoC

In this work, we propose a NoC architecture comprising both wireline and wireless links customized for the CPU-GPU heterogeneous computing platform. We call this proposed NoC a Wireless-enabled Heterogeneous NoC (WiHetNoC). We intend to use dedicated single-hop wireless links between the CPUs and MCs. It has been already demonstrated in [34][35][40] that for multi-hop on-chip communication, mm-wave wireless links can achieve a lower EDP compared to a wireline link. Hence, we intend to use dedicated single-hop wireless links between CPUs and MCs. The GPU-MC communication is handled through a combination of wireline and wireless links that are tailored to the many-to-few traffic pattern. Aside from enabling low latency CPU-MC data exchanges, the use of dedicated wireless links for CPU-MC communication makes the WiHetNoC design agnostic of the CPU and MC placements, instantly fulfilling the CPU QoS requirements. This is because the wireless links guarantee direct single-hop communication regardless of the physical distance between the transceivers as long as they are within the communication range. We can then focus on
fulfilling the GPU QoS requirements by maximizing the throughput of the WiHetNoC. We begin by creating the underlying wireline connectivity to optimize the GPU QoS through multi-objective optimization (MOO). Then, through careful placement of wireless interfaces we create the final WiHetNoC architecture. Figure 3.3 shows the overall design flow to create the WiHetNoC.

### 3.2.2.1 Problem Formulation

In this section, the main problem we examine is the optimization of the NoC link placement in CPU-GPU heterogeneous platforms that run deep learning applications. To identify potential bandwidth bottlenecks, we compute the expected utilization of each link in the NoC.

For a NoC with $R$ routers and $L$ links, we can find $U_k$, the expected utilization of any link $k$ by using the following equation:

$$U_k = \sum_{i=1}^{R} \sum_{j=1}^{R} f_{ij} p_{ijk}, \quad p_{ijk} = \begin{cases} 1, & \text{if } i, j \text{ communicate along link } k \\ 0, & \text{otherwise} \end{cases} \quad (3.3)$$

where $p_{ijk}$ indicates which links are used for the communication from router $i$ to router $j$.

The $p_{ijk}$ value can be found by using the network connectivity and NoC routing protocol. The

---

**Figure 3.3** WiHetNoC design flow. Required input parameters and the objectives optimized in the creation of WiHetNoC are also shown.
value $f_{ij}$ denotes the frequency of interaction between routers $i$ and $j$. It should be noted that these $f_{ij}$ values are not specific to any neural network (NN) layer of the NN architecture. Rather, the $f_{ij}$ values are a representation of the many-to-few traffic pattern that is associated with the considered heterogeneous system.

In our WiHetNoC, we establish the network connectivity such that both the mean link utilization ($\bar{U}$) and the standard deviation among the link utilizations ($\sigma$) are minimized. $\bar{U}$ and $\sigma$ can be determined as below,

$$\bar{U} = \frac{1}{L} \sum_{k=1}^{L} U_k = \frac{1}{L} \sum_{k=1}^{L} (\sum_{j=1}^{R} f_{ij} p_{ijk})$$

$$= \frac{1}{L} \sum_{i=1}^{R} \sum_{j=1}^{R} (f_{ij} \sum_{k=1}^{L} p_{ijk}) = \frac{1}{L} \sum_{i=1}^{R} \sum_{j=1}^{R} f_{ij} h_{ij} (3.4)$$

$$\sigma = \sqrt{\frac{1}{L} \sum_{k=1}^{L} (U_k - \bar{U})^2} \quad (3.5)$$

Here, $h_{ij}$ denotes the minimum number of inter-router hops between routers $i$ and $j$ with the given network configuration. Intuitively, $h_{ij}$ is equal to the total number of links used in the communication between router $i$ and router $j$.

As it can be observed from the above equations, $\bar{U}$ is directly proportional to the traffic weighted hop count (given by $\sum_{i}^{R} \sum_{j}^{R} f_{ij} h_{ij}$). Thus, minimizing $\bar{U}$ also minimizes the inter-router hop count, leading to high network throughput. On the other hand, minimizing $\sigma$ ensures that the link utilizations in the WiHetNoC are well-balanced and free of bandwidth-bottlenecks.
There are several constraints associated with establishing the network connectivity, and hence, the overall problem can be formulated as a constrained optimization problem with the following parameters:

- **Inputs**: \( f_{ij}, R, L \)
- **Outputs**: Network connectivity, \( d^* \)
- **Objectives**: 
  \[
  d^* = \arg\min_{d \in D} f(U(d), \sigma(d)) \quad (3.6)
  \]
- **Subject to**:
  \[
  \frac{1}{R} \sum_{r=1}^{R} k_r \leq k_{avg} \quad (3.7)
  \]
  \[
  k_r \leq k_{max}, \forall r \quad (3.8)
  \]
  \[
  \text{fullyConnected}(d^*) = \text{true} \quad (3.9)
  \]

where \( D \) is the set of all possible network configurations, \( k_{avg} \) and \( k_{max} \) are average and maximum router port count, \( k_r \) is the router port count for router \( r \), and \( d^* \) is the connectivity obtained from optimizing \( U \) and \( \sigma \). \( \text{fullyConnected}(d^*) \) returns true if there exists a path between all pairs of nodes in the given network \( d^* \). This ensures each node can communicate with any other node.

### 3.2.2.2 Optimizing Wireline Connectivity

Since minimizing both \( U \) and \( \sigma \) together is a MOO problem, and the full search space of (6), before constraints, is \( \binom{R^2}{L} \), where \( R \) is the number of routers (one per tile) and \( L \) is the number of links. Searching this entire space for a realistic NoC involving many cores (64 cores for this work) to determine the global optima is clearly intractable. Additionally, ILP, MILP or any other LP solvers are not suitable for this problem since neither optimization objective is a linear function (average link utilization and variance, Eqns. 4 and 5). In this
work, we employ Archived Multi-Objective Simulated Annealing (AMOSA) [4] to solve (6). AMOSA is a simulated annealing (SA) based algorithm in which the optimization process is guided with the help of an archive of solutions. AMOSA is a widely-used MOO algorithm that can find a near-optimal Pareto front for many applications [4]. Hence, we considered it as a suitable candidate MOO algorithm to solve the problem undertaken here. However, any other MOO methodology (i.e., genetic and evolutionary) can be employed to perform this optimization. Exploring such different MOO optimizations may help us in identifying the technique that has the fastest convergence time with comparable quality to AMOSA (identifying the fastest MOO algorithm may be essential to design large-scale heterogeneous systems). However, in this work, we mainly focus on creating a framework to design the wireless-enabled NoC for heterogeneous systems (using CNN training as a case study) by formulating and analyzing the necessary objectives and constraints. Hence, the exploration of the fastest MOO for heterogeneous architecture design is left for future work.

In AMOSA, during each optimization step, a perturbation is created in one of the archived solutions to generate a new configuration. Depending on the comparative quality of this new configuration over all solutions in the archive, AMOSA then updates the archive. Thus, on completion of AMOSA, we obtain a set of archived candidate configurations.

Since there is no structure enforced in the NoC connectivity while using AMOSA (e.g., forced neighborhood connectivity) and the traffic characteristics show highly imbalanced communication, the optimized NoC would generally follow an irregular connectivity. Therefore, we need to follow certain physical restrictions while establishing the link connectivity in this irregular WiHetNoC. First, we limit the average number of inter-tile
communication ports per router \((k_{avg})\) in (7) to four so that the WiHetNoC does not introduce any additional router port overhead when compared to a conventional mesh. Next, we need to restrict the maximum number of ports in a router \((k_{max})\) in (8) so that no router becomes unrealistically large. The MCs in a heterogeneous NoC are traffic hotspots with heavy volumes of incoming and outgoing messages. Increasing \(k_{max}\) allows the number of router ports attached to an MC to increase, and hence, improves the MC router bandwidth. However, high \(k_{max}\) values can lead to large routers, which result in high network energy consumption. Moreover, large routers make the whole system highly vulnerable to failures. Consequently, we consider a Kmax range of four to seven. For each Kmax value, we create a candidate network set through AMOSA that minimizes the mean link utilization and the standard deviation among the link utilizations. Then, among these candidate networks, we choose the network with the lowest EDP as the optimum wireline connectivity for the WiHetNoC (experimental results are shown later in Section 3.3).

3.2.2.3 Wireless Link Placement

As we illustrate in Figure 3.4, the WiHetNoC designed following the methodology described in Section 3.2.2 generally contains several long wireline interconnections. As these links are extremely costly in terms of power and delay, we employ wireless links operating in millimeter (mm)-wave and sub-THz range of 10-220 GHz to connect the routers that are separated by long distances. Since the CPUs are latency sensitive, CPU-MC communication is handled using dedicated wireless links.

In practice, depending upon the available wireless resources, we can only make a limited number of the longest links wireless, while the other links need to remain wireline. We use
five non-overlapping channels centered around 30, 60, 90, 140 and 200 GHz. Each channel sustains a data rate of 16 Gbps for a communication range of at least 20 mm. It should be noted that by using the current CMOS technology both the frequency range and number of channels could be increased further. However, the proposed design methodology presented in this work is oblivious to these physical design parameters, i.e., the number of wireless channels can be increased or decreased without modifying the proposed algorithm. Using these five channels we overlay the wireline connectivity with the wireless links such that a few routers get an additional wireless port. The wireless ports have a wireless interface (WI) tuned to one of the five different frequency channels.

Given the total number of WIs allowed (NWI), we use a WI placement strategy that focuses on minimizing traffic-weighted hop-count [40]. Following this methodology and by varying NWI, we find both the optimum number of wireless interfaces and the best locations in the
WiHetNoC. The optimum value of NWI is discussed later in experimental results and analysis section (Section 3.3).

### 3.2.2.4 Components of the Wireless Interface

The two principal components of a wireless interface are the antenna and the transceiver. WiHetNoC uses a metal zigzag antenna that has been demonstrated to provide the best power gain with the smallest area overhead [34]. A detailed description of the transceiver circuit is out of the scope of this paper. However, the transceivers were designed following the principles described in [33]. The wireless interface is completely CMOS compatible and no new technology is needed for its implementation. In the 28nm technology node, for data rates of 16 Gbps, wireless links dissipate 1.3 pJ/bit (18 mW power) over a 20mm communication range and each WI occupies 0.25\(mm^2\) area.

### 3.2.2.5 Communication Protocols

The proposed NoC principally has an irregular application-specific topology and requires a topology-agnostic routing method. We follow the Adaptive Layered Shortest Path (ALASH) routing methodology [33]. ALASH is built upon the layered shortest path (LASH) algorithm [68]. The ALASH protocol improves the LASH layering function by considering the expected traffic patterns. We follow the priority layering function detailed in prior work [33]. Priority layering allocates as many virtual layers as possible to source-destination pairs with high traffic intensities. Additionally, for any source-destination pair, a path containing a wireless link is enabled only when using the wireless path gives rise to lower latency than taking the wireline-only path. More precisely, for each source-destination pair, the shortest path(s) are found first. If a path involving a wireless link is found to have a shorter length than the
wireline-only path, this wireless path becomes an enabled path. These features improve message transfers under high traffic intensities by providing greater routing flexibility.

As we discuss later in Section 3.3.1, the CNN computations induce an asymmetric proportion of reply and request packets (i.e., the MC-to-GPU traffic volume is usually much higher than the GPU-to-MC traffic volume). In order to efficiently handle this asymmetric traffic, we require an efficient wireless MAC protocol that can dynamically allocate wireless bandwidth between MC-to-GPU and GPU-to-MC data transfers, depending on the current instantaneous communication requirements. For this purpose, we employ the distributed MAC protocol described in [69]. With this MAC, whenever a message wants to utilize the wireless channel, the state of the wireless medium is first checked. If the wireless medium is free, a wireless request period takes place. Each request period consists of N slots, a dedicated slot for each of the N WIs sharing the same wireless channel. During each request period, every WI sharing the same wireless channel has the opportunity to request the wireless medium by broadcasting to the wireless channel an "on" signal (1) in its allocated slot. An off-signal/no-transmission (0) in a particular slot would indicate that there is no request from the corresponding WI. For example, for a set of 4 WIs sharing a wireless channel, the request period has 4 slots (b3 b2 b1 b0). b0 represents WI 0, b1 represents WI 1, and so on. If WI 2 wants to request the wireless channel, WI 2 broadcasts a 1 during b2 and listens for other requests during b0, b1, and b3. After the request period, all WIs follow a simple fairness-based common node selection algorithm [69]. Following the execution of this node selection algorithm, one of the requesting WIs acquires the wireless channel and starts transmitting data packets. When the wireless channel is busy, the packets are re-routed via the wireline
links. Hence, it should be noted that, irrespective of their actual utilization values, the wireless links are inherently congestion-free and cannot become bandwidth bottlenecks; hence, the wireless channel utilization was not considered for the optimization formulation (Section 3.3.3).

### 3.3 Experimental Results and Analysis for Hybrid NoC Architecture

We utilize Gem5-gpu, a heterogeneous full system simulator to obtain processor- and network-level information [70]. Gem5-gpu is a combination of two well-known simulators: Gem5, a manycore CPU simulator and GPGPU-sim, a detailed GPGPU simulator. Gem5-gpu provides a customizable interconnection model through the Garnet network and supports inter-CPU-GPU cache coherence protocols [71].

We have modified this Garnet network topology to implement the WiHetNoC architecture. We employ Gem5-gpu in the full-system simulation mode with default task mapping/scheduling provided in Linux. The exact processing element composition can be optimized, but it is highly dependent on the specific deep learning architecture (i.e., input data size, network depth and number of classes) and the software implementation of the deep learning framework. However, since the proposed optimization methodology uses traffic characteristics and knowledge of the CPU and GPU traffic requirements, our proposed methodology can be used for any composition of CPUs/GPUs/MC/s and system size. Hence, without loss of generality, in this work, we consider a heterogeneous architecture with 56 GPUs, 4 CPUs, and 4 MCs for our experimental analysis. This gives rise to a system size of 64 tiles, which are arranged in an 8 × 8 grid configuration. In the full-system simulations we
consider NVIDIA Maxwell-based GPU cores and standard x86 CPU cores. We consider the MESI two-level cache coherence protocol. Each CPU and GPU streaming multiprocessor (SM) is allocated private L1 data and instruction caches. The considered memory system also incorporates four LLCs that are shared among all the CPUs and SMs. Table 2 provides detailed configurations of the architecture considered in this work. We limit the number of NoC links to be the same as a conventional mesh architecture so that our NoC does not introduce additional area overhead. We use GPUWattch to obtain detailed processor power profiles from the Gem5-gpu statistics [72].

We use the three-stage router microarchitecture for all NoCs under consideration. The delay of each stage is constrained within one NoC clock cycle (2.5 GHz). For routers with more than four inter-tile router ports, the output arbitration has an additional pipeline stage that takes one clock cycle. This is accounted for when determining the latency and energy of the proposed NoC architectures.

3.3.1 Analysis on On-Chip Traffic Patterns

In this section, we analyze the nature of the on-chip traffic patterns generated by training LeNet and CDBNet on the considered heterogeneous manycore platform. Since each layer of any CNN involves unique computation patterns and memory requirements, the volume of

![Figure 3.5 Message injection rate for training the (a) LeNet and (b) CDBNet (normalized with respect to the highest injection rate layer).](image-url)
the on-chip traffic varies from one layer to another. Therefore, to capture the effects of these distinct traffic patterns, we will individually present the NoC-level experimental results for each convolutional (denoted as C), pooling (denoted as P) and fully-connected layer (denoted as F). We compare the network traffic intensity (measured in number of flits injected per second) associated with each neural network layer in Figure 3.5(a) and (b). From these figures, it is clear that for both LeNet and CDBNet, and for both forward and backward passes, the convolutional layers exhibit high injection rates followed by pooling layer while the fully connected layer exhibits the lowest injection rate.

Another important characteristic that we have observed is the asymmetric nature of the on-chip traffic patterns. The volume of GPU-to-MC traffic in heterogeneous systems is usually lower than the MC-to-GPU traffic volume [26]. Moreover, the CNN applications involve the execution of numerous consecutive compute-kernels 2. Each compute-kernel involves a different ratio of input and output memory requirements as implied in Table 1. As it can be seen from Figure 3.6(a) and (b), the variation in the input and output data volumes further

---

2 Each kernel corresponds to a particular CNN layer.
adds to the asymmetry exhibited between the incoming and outgoing MC traffic for the CNN layers.

Apart from the similarity in injection rates and traffic asymmetry, we have also observed in Figure 3.6 that the NoC traffic consistently follows the many-to-few, across all layers of the CNN computation for both LeNet and CDBNet architectures. The main source of the many-to-few traffic come from traffic moving to/from the MCs (the GPUs and CPUs greatly outnumber the MCs). This type of traffic makes up 93% for LeNet and 89% for CDBNet of the total traffic across all the CNN layers.

![Figure 3.7 Snapshot of the memory access temporal locality during the forward pass of (a) convolution and (b) pooling layers running LeNet on the proposed heterogeneous architecture. Each dot represents receiving/transmitting data from/to the memory controllers. For instance, the dot at (10000, 3) indicates that CPU3 communicated with an MC at time 10000. Similar patterns are observed during subsequent iterations of these layers.](image)

We also consider the temporal locality of the memory access for two representative CNN layers, namely, convolution and pooling, for LeNet (CDBNet exhibits similar behavior). It is clear from the Figure 3.7(a) and (b) that in both layers many GPU cores transmit/receive data at the same time. In such situations, the network resources will be monopolized by GPU traffic, leading to significant degradation in CPU-MC communication latency. This demonstrates the need for single-hop shortcuts between the CPU and MC tiles. As mentioned
in Section 3.2, we resolve this problem by employing dedicated wireless links between CPUs and MCs.

From these discussions, it is clear that both the considered CNN architectures give rise to very similar on-chip traffic patterns. Hence, by optimizing a NoC to efficiently handle the many-to-few and asymmetric nature of the on-chip data transfers, it is possible to design a single heterogeneous manycore architecture for implementing CNN computations.

### 3.3.2 Mesh NoC Link Utilization Characterization

In this section, we analyze the link utilization characteristics of the conventional wireline mesh NoC architecture running the considered CNN applications.

Following the discussions in Section 3.2, for a mesh NoC with a given number of tiles, we first identify the positions of the CPUs and MCs such that the CPU-MC communication latency is

![Figure 3.8](image-url)  
Figure 3.8 The central 6×5 portion of the optimized Mesh NoC indicating the link utilizations and the locations of CPUs and MCs for LeNet computations. All link utilizations are normalized with respect to the mean link utilization. The red arrows indicate the bandwidth bottlenecks whose utilization is at least 100% more than the mean.
minimized while the overall NoC throughput is maximized to enable efficient GPU-MC data exchange (under the many-to-few communication pattern). Following [4], we employ the AMOSA algorithm (mentioned in Section 3.2.2.2) to determine the optimal positions of CPUs, GPUs, and MCs in a mesh NoC to jointly optimize CPU-MC communication latency and the overall NoC throughput. Figure 3.8 shows the core types and interesting link utilization behaviors in the central section of the final optimized mesh NoC. All link utilizations are normalized with respect to the mean link utilization. We can observe from Figure 3.8 that in the optimized mesh NoC, the MCs and CPUs are clustered in the middle so that the CPU-MC communication latency is minimized. Moreover, placing the MCs closer to the middle ensures that handling of GPU-MC traffic is distributed among multiple MC ports, leading to higher NoC throughput.

Also, in Figure 3.8, it is evident that even in this optimized NoC, a few links are more heavily utilized when compared to the rest of the links. In a previous work [3], it was demonstrated that links associated with MCs have nearly 500% higher traffic density than the overall average link utilization for the Rodinia backpropagation benchmark [73]. However, this backpropagation benchmark is much simpler than the workloads addressed in this work and only employs a single NN layer. For the LeNet and CDBNet computations considered in this work, the incoming vertical and outgoing horizontal links associated with the MCs have up to 600% and 700% higher traffic densities than the overall average respectively. These results can be attributed to the fact that the on-chip communications involved with the LeNet and CDBNet applications exhibit a high degree of asymmetric traffic and is many-to-few in
nature (Figure 3.6 and 3.8). When such traffic is handled using the traditional XY routing mechanism in mesh NoCs, the link utilizations become highly skewed.

In order to alleviate the traffic congestion caused by aggregation of traffic along the mesh NoC links, one can also adopt a combination of minimal XY and minimal YX routing as proposed in [25]. However, such an approach cannot eliminate the bandwidth bottlenecks from the optimized mesh NoC under the heterogeneous computing induced traffic patterns. To elaborate more, Figure 3.9 shows the average traffic-weighted hop-count and the standard deviation among the link utilizations for the following NoC configurations: optimized mesh and four different WiHetNoCs candidate architectures (these WiHetNoC candidates are explained later in detail). As it can be observed from Figure 3.9, when compared with WiHetNoC candidates, both the standard deviation and the traffic-weighted hop-count of the optimized mesh NoC are at least 2× higher. This indicates the presence of traffic hotspots that can lead to bandwidth bottlenecks.

It can be observed from Figure 3.8 that in the optimized mesh NoC, since the MCs are placed around the CPU cluster, the CPU tile routers also act as intermediate routers forwarding the
GPU-MC traffic. With continuous streams of data flowing from MCs towards GPUs, the above-mentioned traffic forwarding causes traffic congestion in the CPU tile routers, leading to an undesired increase in the CPU-MC communication latency. To resolve this issue, the MCs can be placed away from the CPUs such that the CPU tiles are not involved in the GPU-MC traffic forwarding. However, due to the lack of long-range shortcuts in mesh NoC, placing the MCs away from the CPUs increases the number of hops required for CPU-MC communication leading to network latency penalties. Hence, our target is to design an application-specific heterogeneous NoC architecture with suitable long-range shortcuts to improve both CPU-MC latency and the overall NoC throughput. Since the wireless links make the CPU-MC latency agnostic of their placements, we keep the CPUs at the center of the system and distribute the four MCs to the center tiles in each of the four quadrants of the system. All the other tiles are occupied by the GPU cores surrounding the CPUs and MCs. The whole system is then integrated using the WiHetNoC framework. In the following section, we determine the various network parameters of this WiHetNoC.

### 3.3.3 Determining WiHetNoC Parameters

In this section, we determine the overall architecture and the network parameters of WiHetNoC.

#### 3.3.3.1 Router and Port Upper Bound

In the WiHetNoC design process, we first find the optimum value of \( k_{\text{max}} \) (maximum number of inter-tile communication ports in a router). As we discussed in Section 3.2.2, we start with a \( k_{\text{max}} \) range of 4 to 7. For each \( k_{\text{max}} \) value we use the above-mentioned
AMOSA optimization to find a final candidate solution set. Figure 3.10 shows the mean link utilization ($\bar{U}$) and the standard deviation among the link utilizations ($\sigma$) for all the solutions in the final candidate set corresponding to each $k_{max}$ value. All the values in this figure are normalized with respect to the final WiHetNoC configuration. As it can be observed from the graph, as the $k_{max}$ value is increased, the $\bar{U}$ and $\sigma$ values decrease. This happens because for higher $k_{max}$ values, more inter-router connections are allowed in each MC, leading to lower average hop counts between GPUs and the MCs. However, Figure 3.10 shows diminishing gains in hop count and link utilization standard deviation reductions with increasing $k_{max}$ and there is no gain in exploring beyond $k_{max} = 7$. As explained in Section 3.2.2, the final WiHetNoC wireline connectivity is identified from the candidate networks by comparing their network Energy-Delay-Products (EDP). Average message latency and energy are used in this EDP computation. Figure 3.11 shows the EDPs of the optimal networks corresponding to each $k_{max}$, (thus a total of four
optimal networks are shown corresponding to $k_{\text{max}}$ values ranging from 4 to 7). From Figure 3.11 it is evident that the optimal value for $k_{\text{max}}$ is 6. Beyond this value of $k_{\text{max}}$, the EDP worsens due to higher router energy consumptions without significant gains in network latency and hop count. In addition to our comments in Section 3.3.2, Figure 3.9 demonstrates the network characteristics for the four optimal WiHetNoC architectures (for the four considered $k_{\text{max}}$ values), showing an inflection point specifically for the standard deviation at $k_{\text{max}} = 6$. Hence, we select the network corresponding to $k_{\text{max}} = 6$ as the optimal wireline connectivity for WiHetNoC.

Figure 3.11 Variation in network EDP for different router port upper bounds ($K_{\text{max}}$).

Figure 3.12 EDP and Wireless utilization for various WI counts.
3.3.3.2 Number of WIs

Next, we identify the number of WIs needed for the GPU-MC communication in the WiHetNoC architecture. As mentioned earlier, we are able to create five non-overlapping channels. We dedicate one channel to achieve single-hop CPU-MC communications. Hence, four wireless channels are available for GPU-MC communication.

Figure 3.12 shows the variation in EDP and wireless utilization observed with varying WI counts. The wireless utilization parameter represents the percentage of total messages that are using the wireless channels. As observed from this figure, the EDP initially reduces as the WI count increases: higher number of wireless shortcuts improves the wireless utilization and hence, lowers the overall network latency. However, beyond a WI count of 24, with more than six WIs allocated on a single wireless channel, the MAC overhead (and hence, the channel access latency) and, therefore, the network EDP starts to increase [69]. Hence, in our WiHetNoC, we employ 24 WIs for GPU-MC communication (four channels are used with six WIs operating on each channel). Figure 3.13 shows the effects of adding wireless channels for GPU-MC communication on the performance of the WiHetNoC. With increasing number of wireless channels, the amount of data using wireless medium increases and the overall

![Figure 3.13 EDP and WI utilization with various number of channels.](image-url)
EDP improves. However, the enhancement in wireless utilization and subsequent improvement in EDP slows down beyond a certain number of wireless channels. For the 64-tile system size, increasing the number of wireless channels beyond 4 does not enhance the system performance noticeably as the opportunity for more wireless utilization diminishes.

Since each WI transceiver occupies an area of 0.25mm², the addition of 24 WIs introduce a total of 1.82% silicon area overhead for a die with dimensions of 20 mm×20mm.

### 3.3.3.3 Characteristics of WiHetNoC

Figure 3.14 compares the CPU-MC communication latency and overall NoC throughput achieved with the optimized WiHetNoC against the optimized mesh (explained earlier in Section 3.3.1). From this graph, we can observe that WiHetNoC achieves both higher NoC throughput and lower CPU-MC communication latency than the optimized mesh NoC (while requiring a similar implementation overhead). The WiHetNoC improves the throughput by a factor of two compared to the optimized mesh configuration. This demonstrates the effectiveness of the proposed methodology in designing NoC architectures for heterogeneous platforms with many-to-few communication patterns.
In Figure 3.15, we show the Cumulative Distribution Function (CDF) of the link utilizations for the WiHetNoC and the optimized mesh NoC with XY+YX routing scheme (denoted as Mesh_opt). The utilizations in this figure are normalized with respect to the mean link utilization observed in Mesh_opt (U = 1 represents this mean utilization). It is clear from Fig. 15 that, 20% of the Mesh_opt NoC links have at least 2x higher utilization compared to the mean utilization. In addition, more than 90% of WiHetNoC links fall under the mean link utilization of the mesh NoC. Generally, when compared to Mesh_opt NoC, the WiHetNoC CDF curve is shifted left indicating a reduction in the overall link utilizations, which is obtained through lowered inter-router hop counts. Moreover, as shown in Figure 3.15, unlike Mesh_opt, WiHetNoC has no links with very high utilizations (no links with U > 2). Thus, WiHetNoC greatly reduces bandwidth bottlenecks.

Aside from avoiding the wireline bandwidth bottlenecks, the WiHetNoC is also highly suitable in handling the asymmetric traffic patterns that arise from heterogeneous CNN training computation. By using the distributed MAC protocol (explained in Section 3.2.2.5), the WiHetNoC enables efficient dynamic allocation of wireless bandwidth for MC-to-Core
(both CPU and GPU cores) and Core-to-MC data transfers, depending on the instantaneous communication requirements. This is further corroborated by Figure 3.16 (a) and (b), which compare the ratio between the MC-to-Core and Core-to-MC data transfers that utilize the on-chip wireless interfaces. It is evident that the wireless utilization aligns well with the asymmetry ratios shown earlier in Figure 3.6. This proves that the WiHetNoC is inherently capable of performing efficient dynamic bandwidth allocations as per the communication requirements.

3.3.4 NoC Performance Evaluation

In this section, we compare the performance of the WiHetNoC and the optimized mesh NoC for running the CNN training applications considered in this work. In this comparative performance evaluation, we also consider the HetNoC: a wireline-only architecture that replaces all of WiHetNoC’s wireless links with pipelined long-range metal wires.

We first present the network-level analysis showing both the latency and message EDP (energy delay product per message). Figure 3.17 (a-b) and Figure 3.18 (a-b) show EDP and the network latency, respectively, of WiHetNoC and HetNoC, with respect to the optimized mesh configuration for each layer of LeNet and CDBNet. For LeNet, on an average, the
HetNoC reduces the network latency and EDP by 23% (22% for CDBNet) and 44% (42% for CDBNet) respectively, when compared to the mesh NoC. With the use of long-range shortcuts between physically remote nodes, the HetNoC enables a lower average hop count than the optimized mesh NoC, and hence, achieves significant reductions in intermediate flit counts. However, as we stated in Section 3.2.2.3, the long wireline links of the HetNoC suffer from high link latency and energy consumption. In the WiHetNoC architecture, many of these long-range wireline links are replaced with energy-efficient wireless links, and hence, WiHetNoC enables 24% more latency improvement for both LeNet and CDBNet and 30% and 28% more EDP improvement in average for LeNet and CDBNet respectively, when compared to the HetNoC. From Figure 3.18 (a-b), it is evident that the WiHetNoC achieves 41.8% and 42%
lower network latency on an average for LeNet and CDBNet respectively compared to the mesh architecture. In addition, as shown in Figure 3.17 (a-b), WiHetNoC also saves the EDP by 60% and 58% on an average for LeNet and CDBNet respectively, compared to the same mesh NoC.

From Figure 3.17(a-b) and 3.18(a-b) we can also observe that the achieved WiHetNoC gain varies from one CNN layer to another. Layers that exhibit high traffic intensities (layers P1, C2 and P2 in Figure 3.5(a)) benefit more from the advantages provided by the WiHetNoC and hence achieve more latency and EDP improvement (over mesh in Figure 3.17 and Figure 3.18) than the layers that involve low injection rates (layers F1 and F2 in Figure 3.5(a)). The fully connected layers show the lowest injection rate, and most of the NoC traffic in this layer is exchanged between CPU and MCs. For this case, the optimized mesh can handle the CPU-MC traffic without creating noticeable bottleneck as MCs are placed closer to the CPUs when compared to CNN layers with higher injection rates. Therefore, as shown in Figure 3.17 (a-b) and Figure 3.18 (a-b), the achievable performance gain of WiHetNoC is lower for the fully connected layers when compared to that of the convolutional layers.
3.3.5 Full-System Performance Evaluation

In this subsection, we compare the overall application execution time and the full-system EDP for the optimized mesh, HetNoC, and WiHetNoC interconnected manycore platforms. Figure 3.19 shows the execution time and full-system EDP for WiHetNoC and HetNoC, with respect to the optimized mesh architecture for both CNN training applications. The HetNoC achieves 8.1% and 8% execution time improvement for CDBNet and LeNet respectively over the mesh while the WiHetNoC shows 13.1% and 13.2% improvement for CDBNet and LeNet respectively over the mesh NoC. The dedicated wireless channel for CPUs on the WiHetNoC enables a highly efficient data transfer between CPUs and MCs. Also, the use of wireless shortcuts helps in achieving high-bandwidth and low-latency GPU-MC communications. These benefits translate to a 25% full-system EDP reduction with WiHetNoC for both CDBNet and LeNet, when compared to the optimized mesh NoC.

Figure 3.19 Normalized execution time and full system EDP for training LeNet and CDBNet.
3.4 Machine Learning-based Design Optimization Framework for Heterogeneous Manycore Systems

In Section 3.3, we have proposed design optimization framework for heterogeneous manycore systems targeted for deep learning applications. As explained earlier, the design of heterogeneous manycore platforms to accelerate deep learning is formulated as a MOO problem due to multiple necessary performance metrics. To solve this problem, we initially employ AMOSA algorithm. AMOSA is a simulated annealing (SA) based algorithm in which the optimization process is guided with the help of an archive of solutions. Although AMOSA provides a near-optimal solution set, it still requires considerable computation time.

To speed up this process, in this Section, we propose a machine learning algorithm, MOO-STAGE that efficiently solve the MOO problems to design a high-performance and energy-efficient heterogeneous manycore systems. MOO-STAGE algorithm employs a machine learning based optimization algorithm called STAGE. The key insight of STAGE algorithm is that it leverages past search experience to learn a function approximator that predicts outcome of given input state. Using the learnt function approximator, computationally expensive hill-climbing process can be avoided and hence significant amount of network design time can be saved. As a case study, we apply MOO-STAGE for designing 3D heterogeneous manycore systems.

3.4.1 3D Heterogenous Manycore Systems

To reduce the performance bottlenecks arising out of the planar interconnects, three-dimensional (3D) integrated circuits (IC) have been introduced [74][75][76][77]. By
stacking planar dies on top of one another and connecting them with Through-Silicon Vias (TSVs), the communication latency can be greatly reduced. However, since 3D ICs have considerably higher power density than their 2D counterparts, the manycore system must be optimized by examining both the performance and the thermal effects of the manycore system. In addition to 3D ICs, the Network-on-Chip (NoC) paradigm has emerged as a revolutionary methodology for integrating many embedded cores in a single die. Together, NoC architectures for 3D ICs offer an unprecedented performance gain beyond the Moore's law regime. It has already been shown that 3D NoCs enable the design of a low latency and high throughput communication backbone for manycore chips [78].

### 3.4.2 Optimization Challenges in 3D heterogeneous platforms

CPU-GPU manycore systems typically consist of a few CPUs, a few shared MCs, and many GPUs. Each CPU and GPU has dedicated L1 cache, while the MCs contain the system’s Last-

![Illustration of proposed heterogeneous architecture.](image-url)
Level-Cache (LLC) and the access mechanism to off-chip memory. Figure 3.20 illustrates the considered 3D heterogeneous architecture.

### 3.4.2.1 Traffic Characteristics

Due to the heterogeneity, memory configuration, and uneven distribution of tile types, CPU-GPU heterogeneous systems display unique traffic characteristics. First, since GPU cores usually operate independently of one another, the inter-GPU communication is nearly nonexistent. Second, since both LLC and main memory accesses reside in the MCs, CPUs and GPUs primarily communicate with the MCs. These features give rise to a significant amount of many-to-few traffic patterns [24][25][26]. Therefore, the NoC in heterogeneous manycore systems needs to accommodate these unique (many-to-few) traffic patterns to sustain high performance and energy efficiency.

### 3.4.2.2 MOO Formulation

In addition to handling many-to-few traffic patterns, NoCs for heterogeneous systems are expected to simultaneously handle two different communication requirements, one for CPUs, and another for GPUs (due to significant architectural differences). In addition, 3D integration results in greater power density that cause thermal hotspots, and hence reducing temperature is another key challenge of the 3D heterogeneous manycore system under consideration.

#### 3.4.2.2.1 Optimization Objectives

In this section, we discuss the necessary objectives to ensure the design of an efficient heterogeneous system.
CPU Communication Objective: In heterogeneous systems, the shared resources are often monopolized by GPUs, thus resulting in significant degradation in CPU-MC communication latency [29]. Since CPU-MC communication is sensitive to latency, we consider CPU-MC latency (Lat) as an objective in the optimization problem to prevent execution delays due to CPUs. For C CPUs and M MCs, the expected CPU-MC latency is defined as follows:

\[
Lat = \frac{1}{C \cdot M} \sum_{i=1}^{C} \sum_{j=1}^{M} (r \cdot h_{ij} + d_{ij}) \cdot f_{ij}
\]  

(3.10)

where \( r \) is the number of router stages, \( f_{ij} \) is the frequency of traffic interaction between tiles \( i \) and \( j \), \( h_{ij} \) is the number of hops from CPU \( i \) to MC \( j \), and \( d_{ij} \) indicates the total link delay. Since the constructed NoC may be irregular, \( h_{ij} \) is determined by a shortest path and deadlock-free routing strategy [68].

GPU Communication Objective: Unlike CPUs, GPUs can hide much of the memory access latency due to their massively parallel and throughput-oriented nature. Therefore, we maximize the throughput of GPU-related traffic by reducing the congestion within the system to allow more messages to utilize the network at a time. To do this, we seek to minimize the amount of expected contention for each link in the system. Hence, given a frequency of traffic interaction \( f \) and the routing paths, we want to simultaneously reduce the likelihood of traffic flows competing for the same link and the overall path length. Therefore, we consider both the mean (\( \bar{U} \)) and variance (\( \sigma \)) of expected link utilization as suitable objectives. The expected utilization of link \( k \) (\( u_k \)) can be obtained by following equation:

\[
u_k = \sum_{i=1}^{R} \sum_{j=1}^{R} f_{ijk}p_{ijk} = \begin{cases} 1, & \text{if } i, j \text{ communicate along link } k \\ 0, & \text{otherwise} \end{cases}
\]  

(3.11)
where R is the total number of tiles and $p_{ijk}$ is obtained by using the network connectivity and routing protocols. Then, $\bar{U}$ and $\sigma$ can be determined from the following equations:

$$\bar{U} = \frac{1}{L} \sum_{k=1}^{L} u_k$$

(3.12)

$$\sigma = \sqrt{\frac{1}{L} \sum_{k=1}^{L} (u_k - \bar{U})^2}$$

(3.13)

By minimizing (3) and (4) simultaneously, we can achieve low traffic aggregation, well-balanced traffic, and high network throughput. Since (3) incorporates the traffic profiles ($f_{ij}$), the NoC will be implicitly optimized for many-to-few traffic.

Thermal requirements: To minimize potential thermal hotspots in the 3D heterogeneous platform, we use the thermal model proposed in [79]. The many-core system is divided into N single-tile stacks with K layers. The temperature of a core within a single-tile stack $n$ located at layer $k$ from the sink ($T_{n,k}$) due to the vertical heat flow is given by:

$$T_{n,k} = \sum_{i=1}^{k} (P_{n,i} \sum_{j=1}^{i} R_j) + R_b \sum_{i=1}^{k} P_{n,i}$$

(3.14)

where $P_{n,i}$ is the power consumption of the core $i$ layer away from the sink in single-tile stack $n$, $R_j$ is the vertical thermal resistance, and $R_b$ is the thermal resistance of the base layer on which the dies are placed. The values of $R_j$ and $R_b$ are obtained using 3D-ICE [80]. The horizontal heat flow is represented through the maximum temperature difference in the same layer $k$ ($\Delta T(k)$):

$$\Delta T(k) = \max_{n} T_{n,k} - \min_{n} T_{n,k}$$

(3.15)
The overall thermal prediction model includes both vertical and horizontal heat flow equations. Following [79], we use $T$ as our comparative temperature metric for any given 3D architecture:

$$T = \left( \max_{n,k} T_{n,k} \right) \left( \max_k \Delta T(k) \right)$$

(3.16)

### 3.4.2.2.2 Physical Constraints

To come up with practical, feasible designs, we introduce several physical constraints to the design optimization: 1) Due to area constraints, we must restrict the number of links, $L$. In this work, we set $L$ to the number of links in a standard 3D mesh to ensure that this NoC has similar overhead to a standard mesh. It should be noted that while we impose the physical constraints based on a conventional 3D mesh architecture, the constraints can be configured based on any other baseline. Also, the average number of inter-router ports per router ($K_{avg}$) is set to be equal to that of a 3D mesh. 2) We restrict a router’s maximum number of inter-router ports, $K_{max}$, so that no router becomes unrealistically large. In general, having large $K_{max}$ for the router attached to an MC will improves the bandwidth to the MC under the many-to-few traffic pattern. However, high $K_{max}$ values can lead to large routers, resulting in high network energy consumption. Hence, we vary $K_{max}$ from five to eight to determine a suitable $K_{max}$ value. Our experimental results show that the solution set obtained using $K_{max} = 7$ results in the lowest network EDP among all the configurations. Therefore, in this work, we employ a $K_{max}$ value of seven. 3) All NoCs must have a viable communication path between any two tiles.
3.4.2.3 MOO Problem and Challenges

The overall optimization problem is to find a design \( d = (V, E) \in D \), where \( V \) is the tile configuration and \( E \) is the set of links, that:

\[
\forall n: \text{minimize } Obj_n \text{ in } Obj = (Lat(d), \bar{U}(d), \sigma(d), T(d)) \tag{3.17}
\]

subject to

\[
\begin{aligned}
\Sigma_{i=1}^R \deg(r_i) / R &= 2L = K_{avg} \\
\forall i: \deg(r_i) &\leq K_{max} \\
\forall i, j: \text{Path}(r_i, r_j) &= 1
\end{aligned} \tag{3.18}
\]

where \( Obj_n \) is the \( n^{th} \) optimization objective, \( r_i \) is router \( i \), \( \deg(r_i) \) is the number of links connected to \( r_i \), and \( \text{Path}(r_i, r_j) = 1 \) if there exists a communication path between \( r_i \) and \( r_j \).

Eqs. (3.17-18) represents a MOO problem, and its full search space (before adding any constraints) would be \( R! \left( \frac{R}{K} \right)^R \left( \frac{R-1}{L} \right)^L \). Searching this entire space for the global optima is clearly intractable even for moderate sizes of \( R, K \) (number of planar layers), and \( L \). The relative importance of any of the objectives depends on the characteristics of the target application. Hence, combining these objectives using a weighted sum approach (i.e., linear scalarization) is not viable because: 1) Searching over different candidate weights via a single-objective solver is computationally very expensive; and 2) It is impossible to find the solutions on the non-convex portion of the Pareto-optimal set with this approach [81].

3.4.3 Proposed MOO Algorithm: MOO-STAGE

To efficiently solve the optimization problem presented in Eqs. (3.17-18), we propose MOO-STAGE, a new MOO algorithm that extends a machine learning based optimization algorithm called STAGE [6]. Our proposed algorithm is general as it can handle any number of
independent or correlated objectives. The key idea behind MOO-STAGE is to explore the search space intelligently such that the MOO problem is efficiently solved. More precisely, MOO-STAGE is a learning algorithm that leverages past search experience (Local search) to learn an evaluation function that can then estimate the outcome of performing a local search from any given state in the design space (Meta search). Using this evaluation function, MOO-STAGE guides the search towards promising areas of the search space. In practice, the MOO-STAGE algorithm iteratively executes Local and Meta searches in a sequence. Next, we describe the steps of the newly proposed algorithm in detail.

3.4.3.1 MOO-STAGE: Local Search

Given an objective (or set of objectives), the goal of a local search algorithm (e.g., greedy search) is to traverse through a sequence of neighboring states to find a solution that minimizes the objective(s). To accommodate multiple objectives, we employ the Pareto Hyper-Volume (PHV) [82] heuristic to evaluate the quality of a set of solutions (higher is better). Essentially, the PHV is the size of the objective space that is dominated by a set of solutions. A design $P$ is dominated by design $Q$ ($P < Q$) when:

$$\forall i: \text{Obj}_i(P) \leq \text{Obj}_i(Q) \land \exists i: \text{Obj}_i(P) < \text{Obj}_i(Q)$$  \hspace{1cm} (3.19)

Despite its simplicity, guided search using the PHV heuristic has two strong advantages over other metrics for comparing solutions [82]: 1) Improvement in any objective will be captured by the PHV. If a new state has a better PHV than the current state, it guarantees that the quality of the new state is better than the current state. 2) Allows handling any number of objectives as part of the MOO problem (i.e., generality) since PHV maps to a single output (cost). This is particularly useful for learning the evaluation function via a regression learner.
Therefore, the proposed algorithm can be applied to MOO problems that involve any number of objectives and has broad applicability to diverse problems in EDA. To compute the PHV, we employ a fast and scalable PHV algorithm called Hypervolume by Slicing Objectives [83]. It employs the divide-and-conquer principle to achieve efficiency: it repeatedly divides the PHV computation into simpler problems with fewer objectives and aggregates the solutions of simpler problems to compute the total Hypervolume.

In this work, we use a simple greedy search with the objective of maximizing PHV with respect to the input objective set ($PHV_{obj}$) as the local search procedure (Figure 3.21). However, it should be noted that any other local search method can be used to similar effect.

\begin{figure}[h]
\centering
\begin{tabular}{|l|}
\hline
\textbf{Input:} $Obj$ (Set of optimization objectives), $d_{start}$ (Starting design) \\
\hline
\textbf{Output:} $S_{local}$ (Non-dominated set of designs), $S_{traj}$ (Trajectory set), $d_{last}$ (Last design) \\
\hline
1: \textbf{Initialize:} $S_{local} \leftarrow \{d_{start}\}, S_{traj} \leftarrow \{d_{start}\},$ \\
$d_{curr} \leftarrow d_{start}$ \\
2: \textbf{While} 1: \\
3: \hspace{1em} $d_{next} = \arg \max_{d \in \text{neigh}(d_{curr})} PHV_{Obj} (S_{local} \cup \{d\})$ \\
4: \hspace{1em} \textbf{If} $PHV_{Obj} (S_{local} \cup \{d_{next}\}) > PHV_{Obj} (S_{local})$: \\
5: \hspace{2em} $S_{local} \leftarrow S_{local} \cup \{d_{next}\}$ \\
6: \hspace{2em} $S_{local} \leftarrow \{d \in S_{local} \mid (\exists k \in S_{local}) [k < d]\}$ \\
7: \hspace{1em} \textbf{Else:} \\
8: \hspace{2em} \textbf{Return} ($S_{local}, S_{traj}, d_{last} = d_{curr}$) \\
9: \hspace{1em} $d_{curr} \leftarrow d_{next}$ \\
10: \hspace{1em} $S_{traj} \leftarrow S_{traj} \cup \{d_{curr}\}$ \\
\hline
\end{tabular}
\caption{Algorithm for Local Search: local ($Obj, d_{start}$)}
\end{figure}
Starting from the initial state $d_{\text{start}}$, we find the best neighboring state $(\text{neigh}(d_{\text{curr}}))$ that improves the PHV heuristic at each greedy search step (Figure 3.21, line 3). In the context of designing 3D heterogeneous systems, a neighboring state is where exactly one planar link is repositioned, or two tiles are swapped (both irrespective of layers). If this best neighboring state improves the PHV value, we add this state to the set of local optima ($S_{\text{local}}$) while ensuring that all designs in $S_{\text{local}}$ are non-dominated (Figure 3.21, lines 4-5). This is repeated until the best neighboring state does not improve the PHV value, at which point, we return the local optima set, search trajectory ($S_{\text{traj}} = d_{\text{start}} ... d_{\text{last}}$), and the final search state ($d_{\text{last}}$). Essentially, the algorithm explores the neighborhood of the current solutions to expand the Pareto front to dominate as much of the objective space as possible.

3.4.3.2 MOO-STAGE: Meta Search

Unfortunately, the quality of local search procedures critically depends on the starting point of the search process ($d_{\text{start}}$). If the search constantly begins near some poor local minima, it is possible that the search will never find a high-quality solution. MOO-STAGE attempts to solve this problem by learning a function approximator (evaluation function) that can predict the outcome of a local search procedure from a particular starting point and use this information to intelligently select high potential starting states to significantly reduce the computation time. We discuss the details of this procedure in the following paragraphs.

After completing the local search, we add the local optima set to the global optima set ($S_{\text{global}}$) ensuring that all states in the global optima set is non-dominated (Figure 3.22, lines 3-4). If the local optima set didn’t add any new entries to the global optima set, the algorithm completes and returns the global optima set (Figure 3.22, lines 5-6). Otherwise, we add the
local search trajectory ($S_{traj}$) and PHV of this trajectory ($PHV_{Obj}(S_{traj})$) as a training example to the aggregated training set ($S_{train}$) and learn the evaluation function $Eval$ (Figure 3.22, lines 7-8). In this work, we employ Regression Forest as the base learner for inducing $Eval$, but other regression learners that are fast and expressive to fit the training data can be used to similar effect. Then, starting from the last state of the local search ($d_{last}$), we employ a standard greedy search using $Eval$ as the heuristic to find the starting state for the next local search iteration ($d_{restart}$). If $d_{last} = d_{restart}$, we choose a random design from the design space instead ($rand(D)$) (Figure 3.22, lines 9-13). Essentially, the algorithm attempts to learn a regressor that can predict the PHV of the local optima from any starting design and guide the search towards predicted high-quality starting designs.

---

**Figure 3.22 Algorithm for MOO-STAGE.**

```
Input: Obj (Set of optimization objectives),
iter_{max} (Maximum iterations), D (Design space)
Output: S_{global} (Non-dominated set of designs)

1: Initialize: $S_{global} \leftarrow \emptyset$, $S_{train} \leftarrow \emptyset$, $d_{start} \leftarrow rand(D)$
2: For $i = 0$ to $iter_{max}$:
3:   $(S_{local}, S_{traj}, d_{last}) = local(Obj, d_{start})$
4:   Maintain non-dominated global set:
5:       $S_{global} \leftarrow S_{global} \cup S_{local}$
6:       $S_{global} \leftarrow \{d \in S_{global} | \forall k \in S_{global} \leftarrow d \neq k \}$
7:   If $S_{global} \cap S_{local} = \emptyset$: [If algorithm converged]
8:       Return $S_{global}$
9:   Add training example for each design $d \in S_{traj}$:
10:      $S_{train} \leftarrow S_{train} \cup \{(d, PHV_{Obj}(S_{traj}))\}$
11:   Train evaluation function: $Eval \leftarrow train(S_{train})$
12:   Greedy Search: $d_{restart} = greedy(Eval, d_{last})$
13:   If $d_{last} = d_{restart}$:
14:       $d_{start} \leftarrow rand(D)$
15:   Else
16:       $d_{start} \leftarrow d_{restart}$
17: Return $S_{global}$
```
Using these two computational search processes (local and meta search), we can find high-quality solutions by selecting good starting states and learning how the starting state is related to the quality of the solutions. By iterating over these two processes, MOO-STAGE progressively learns the structure of the solution space, and then directs the search towards increasingly promising solutions which significantly improves the convergence time to find high-quality MOO solutions.

3.4.4 Experimental Results

3.4.4.1 Experimental Setup

In this work, we use Gem5-GPU [70], a heterogeneous full-system simulator to obtain network and processor level information. We modify the Garnet network model within Gem5-GPU to implement the proposed heterogeneous NoC architecture. For evaluation purposes, different applications from the Rodinia benchmark suite [73], namely Breadth-First search (BFS), LU Decomposition (LUD), Hotspot, Nearest Neighbor (NN), and Needleman-Wunsch (Needle) have been used.

For all experiments, we use full-system simulation mode with the MESI Two Level cache protocol. We use a 64-tile system comprised of 8 CPUs, 16 MCs, and 40 GPUs. The tiles have been distributed in a 4×4×4 3D system (see Figure 3.20) with the heat sink at the bottom. It should be noted that the proposed optimization methodology can be applied to any composition of tile types and layer/stack size. The CPUs and GPUs are based on standard x86 and NVIDIA Maxwell architectures respectively. Each CPU and GPU have a private 32Kb L1 cache. A 4MB L2 Cache is distributed among the MCs equally. The operating clock frequencies for CPUs and GPUs are 2.5GHz and 0.7GHz respectively. The NoC architectures
employ the deadlock-free ALASH routing [68]. We use GPUWattch [72], McPat [84] and 3D-ICE [80] tools to obtain the power and thermal profiles. All analyses have been performed on an Intel® Xeon® CPU E5-2620 @ 2GHz machine with 16 GB RAM running CentOS 6.

In this work, we consider AMOSA as the baseline approach to solve the MOO problem undertaken here. AMOSA is a widely employed MOO algorithm due to its ability to achieve near-optimal solutions [4]. AMOSA is a simulated annealing (SA) based algorithm in which the optimization process is guided with the help of an archive of solutions. However, like other SA techniques, AMOSA relies on a hill-climbing process and needs to be annealed sufficiently slowly to obtain the optimum solution set [4].

3.4.4.2 AMOSA vs. MOO-STAGE

In this section, we present the comparative performance evaluation between AMOSA and MOO-STAGE for optimizing 3D heterogeneous architectures. To capture the effectiveness of each optimization algorithm, we consider convergence time and solution quality as the relevant performance metrics. For brevity, we present the results considering the BFS benchmark as a representative example. We observed the same trends with all other benchmarks under consideration.
Figure 3.23(a) shows the evolution of the PHV metric (solution set quality) over time for AMOSA and MOO-STAGE. It is evident from Figure 3.23 (a) that MOO-STAGE achieves higher PHV values significantly faster than AMOSA; MOO-STAGE converges after 11 hours while AMOSA takes 95 hours to converge. MOO-STAGE’s key advantage is its ability to use machine learning to choose good starting points for design exploration.

To demonstrate the quality of MOO-STAGE’s starting points, in Figure 3.23 (b) we show the prediction error of the evaluation function $E$ (Figure 3.22, lines 7-8) as a function of time. The prediction error (in %) represents the difference between the estimated PHV value obtained by Eval and the actual PHV value obtained by the subsequent local search. After only a few hours, the prediction error is less than 5%, and less than 1% after 10 hours. This low error rate indicates that the evaluation function Eval can accurately provide good starting points for the local search. Hence, MOO-STAGE continuously improves its search by choosing promising starting points. As seen in Figure 3.23 (a), this allows MOO-STAGE to find solutions much quicker than AMOSA’s naïve explorations.

To see how improvements in PHV translate into overall system performance, in Figure 3.23 (c) we show the temperature delay product (TDP) of the best solution found over time. TDP
is an appropriate measure since it incorporates the joint effects of both core temperature and network latency into a single term. TDP is analogous to the well-known energy-delay product (EDP) and is more suited to our optimization objectives. We can see that Figure 3.23 (c) follows Figure 3.23 (a) very closely, an improvement (higher) in PHV results in a similar improvement (lower) in TDP, demonstrating through system metrics that MOO-STAGE can find high-quality designs much quicker than AMOSA.

To demonstrate the scalability of the proposed algorithm, we vary the number of objectives for the proposed 3D heterogeneous architecture. As an example, we use the following objective sets. For two-objectives, we consider mean (Eq. 3.12) and variance of link utilization (Eq. 3.13): \( \{ \bar{U}, \sigma \} \). For three-objectives, we add CPU-MC latency (Eq. 3.10): \( \{ \bar{U}, \sigma, Lat \} \). Finally, for four-objectives, we add temperature (Eq. 3.16): \( \{ \bar{U}, \sigma, Lat, T \} \). Figure 3.24 shows the normalized convergence time of AMOSA with respect to MOO-STAGE for the different objective sets. As the number of objectives increases, AMOSA simply isn’t able to keep up with MOO-STAGE due to increased complexity of the optimization problem. For further analysis, in Figure 3.25 we examine the Pareto fronts of AMOSA and MOO-STAGE for the three-objective case (three-objectives are used to easily plot the objective space) at the instant MOO-STAGE converges \( T_{MOO-STAGE} \). At this point, MOO-STAGE finds much better
solutions and the MOO-STAGE Pareto front completely dominates the archive solutions of AMOSA.

Next, we analyze the four-objective 3D heterogeneous architectures generated by MOO-STAGE and AMOSA at $T_{MOO-STAGE}$, specifically, we look at the placement of the tiles and planar links. Figure 3.26 shows the tile and link distributions of both the architectures. To attempt to optimize temperature, both MOO-STAGE and AMOSA mostly fill the bottom layer (closest to the heat sink) with high-power consuming GPUs. However, due to the large number of GPUs, all GPUs cannot fit on the bottom layer. Unfortunately, AMOSA ends up placing two GPUs in the top layer which negatively affects the thermal profile without any performance advantage (MOO-STAGE keeps GPUs in the bottom three layers). Of note, it is
interesting to see that AMOSA and MOO-STAGE make similar tile placement configurations but very different link placements. We conjecture that this is due to the link placement search space being much larger than the tile placement search space (i.e., only three types of tiles and swapping placement of identical GPU tiles is unnecessary). In Figure 3.26, we see that MOO-STAGE distributes more links to the layers with MCs, as MCs are the main traffic bottlenecks under many-to-few traffic. However, as the search space involving the links is much larger, AMOSA fails to explore it adequately and ends up distributing links more evenly among all the layers. These analyses demonstrate that the solutions obtained from AMOSA at $T_{MOO-STAGE}$ suffer in terms of performance and temperature when compared to MOO-STAGE.

![Figure 3.26 Core and link distribution of 3DHet using (a) MOO-STAGE and (b) AMOSA at $T_{MOO-STAGE}$](image-url)
3.4.4.3 Full-System Performance Evaluation

In this section, we examine the performance of the 3D heterogeneous architecture (3DHet) optimized using the proposed MOO-STAGE algorithm with respect to a 3D mesh architecture with four-objective MOO-STAGE optimized tile placement (3DMesh). From MOO-STAGE, we obtain a final candidate solution set for each 3DMesh and 3DHet architecture. To highlight the performance improvement of the proposed 3D heterogeneous architecture, we select the configuration that gives the lowest full-system EDP within the same maximum temperature (75°C was chosen as an example) for both 3DMesh and 3DHet (normalized full-system EDP shown in Figure 3.27). It should be noted that the improvement in both GPU throughput and CPU latency are captured in full-system EDP. For the same target peak temperature, 3DHet achieves up to 23% full system EDP reduction for NN and 16.6% reduction on average across all considered applications when compared to a 3DMesh.

![Normalized full-system EDP with respect to 3DMesh.](image)

Figure 3.27 Normalized full-system EDP with respect to 3DMesh.
3.5 Summary

The computation patterns arising from training CNNs can be efficiently handled by a single-chip heterogeneous manycore platform. Conventional NoC architectures, such as mesh, are unable to handle the contrasting communication requirements imposed by CPUs and GPUs. In this dissertation, we have proposed the design of a hybrid NoC-enabled single-chip heterogeneous computing platform for energy-efficient acceleration of CNN training. By virtue of using single-hop wireless links, the proposed WiHetNoC architecture achieves a much better GPU-L2 throughput and a lower CPU-L2 communication latency, when compared to a highly-optimized mesh NoC. Thus, the proposed NoC architecture is able to efficiently fulfill the communication requirements of both CPU and GPU cores. For the considered CNN applications, WiHetNoC achieves 25% lower full system EDP with respect to the mesh and 15% lower full system EDP when compared to a fully wireline application-specific architecture.

Designing heterogeneous manycore architectures involves searching through large combinatorial design spaces. Conventional heuristic-based approaches scale poorly for large design spaces as they perform unguided search. In this dissertation, we have proposed a machine learning-inspired scalable MOO framework, MOO-STAGE, that finds high-quality solution sets with significantly less computation time when compared to AMOSA, a state-of-the-art MOO approach. With MOO-STAGE, the optimization process can learn how to identify and explore better areas of the solution space (guided exploration), which significantly improves the scalability of optimization process. Moreover, using this unique quality, we have demonstrated that MOO-STAGE is able to achieve a high-quality solution with 9X faster
runtime than AMOSA (for four-objectives). Finally, MOO-STAGE quickly finds (near-) optimal designs that can efficiently handle the communication requirements arising from both CPUs and GPUs.


4 Dynamic VFI Control Algorithms: Machine Learning-based Approaches

High-power consumption of manycore systems is a critical concern for continued scaling. Higher system power not only raises the chip temperature and cooling costs, but also decreases chip reliability and performance. In order to deal with the continually expanding core count in manycore systems, voltage–frequency island (VFI) has emerged as an efficient and scalable power management strategy [85][86]. This is because, for VFI-based multicore systems, it is possible to implement efficient power and thermal management via dynamically fine-tuning the voltage and frequency (V/F) of each island under given performance constraints. Moreover, dynamically tuned VFIs (DVFI) reduce the area overhead associated with a fully distributed per core dynamic voltage frequency scaling (DVFS). Hence, a hierarchy of globally distributed (inter-VFI) and locally centralized (intra-VFI) control mechanisms can provide the best trade-off between power and resource management.

Conventional DVFS techniques operate on a core-by-core basis and use core-level information to tailor the V/F values of each individual core. Simple averages core-level information may not capture the information required to accommodate every core, router, and link within a VFI, particularly for VFIs with large intra-VFI workload variance. In this dissertation, we propose machine learning-based algorithms that are suitable to fine-tune the V/F levels within VFIs.
4.1 Problem Formulation

We take advantage of the temporal variations in the application and the structural relationship between VFIs to create DVFI policies that dynamically tune the V/F of each VFI. Figure 4.1 illustrates the design and evaluation process for the DVFI policies and the underlying VFI system. First, we profile each application to gather detailed core and communication statistics. Then, using this information, we create application-specific VFI clusters and DVFI policies offline. Finally, we evaluate the application-specific DVFI policy on the VFI system by running the application under consideration. In this paper, we study the ideal scenario, learning and evaluating DVFI policies based on the same application. This dissertation is important to benchmark the performance and scalability of RL and IL approaches, and to drive future work on online learning in the context of multiple applications.

4.1.1 Optimization Objective

Consider a manycore system with \( n \) cores. Without loss of generality, let us assume that there exist \( k \) VFIs. The DVFI control policy \( \pi \), at each time interval (window) \( t \) (where \( N \) is the total number of intervals), takes the current system state \( s_t \) and generates the V/F allocation for all \( k \) VFIs, i.e.,

\[
\pi: s_t \rightarrow \{V_1/F_1, V_2/F_2, ..., V_k/F_k\} \quad (4.1)
\]

We then define the energy consumption using this policy as:

\[
E(\pi) = \sum_{t=1}^{N} E_w(t, \pi(s_t)) \quad (4.2)
\]
where $E_w(t, \pi(s_t))$ is the energy consumption of the system for the time window $t$ given the joint V/F allocation $\pi(s_t)$. Similarly, we can define the execution time when using the DVFI policy $\pi$ as:

$$T(\pi) = \sum_{t=1}^{N} T_w(t, \pi(s_t))$$ \hspace{1cm} (4.3)

where $T_w(t, \pi(s_t))$ is the execution time of the system for the time window $t$ given the joint V/F allocation $\pi(s_t)$.

Given a VFI-enabled manycore architecture, an application, and a maximum allowable performance penalty, our objective is to create a DVFI control policy $\pi^*$ that minimizes the energy dissipation within the specified performance penalty, i.e.,

$$\pi^* = \arg \min_{\pi} E(\pi)$$ \hspace{1cm} (4.4)
Subject to: $\frac{T(\pi)}{T(\pi_{nom})} \leq p \quad (4.5)$

where $p$ is the user-specified maximum allowable performance penalty and $\pi_{nom}$ is the control policy that simply chooses the highest V/F for all VFIs during each time window.

### 4.1.2 Overhead Considerations

To achieve optimal energy-efficiency in the presence of dynamic workloads, the DVFI controllers must be able to quickly make decisions and adapt the V/F values accordingly. Therefore, in this work, we consider developing a hardware DVFI controller for each VFI. By keeping the controllers in hardware and judiciously choosing the DVFI policy, we can keep the energy and area overhead small when compared to the overall energy savings due to dynamic VFI control.

### 4.1.3 Static VFI Creation

For any DVFI-based system, the initial step is to cluster the cores with similar behavior so that all cores and their associated routers and links in a VFI can benefit from coordinated V/F tuning. In this respect, instructions per cycle (IPC) per core and traffic statistics are employed to capture the utilization and communication behavior of the cores, respectively.

In this work, we follow the basic methodology from [9] and create static VFI clusters using the k-means clustering algorithm to minimize each VFI’s intra-cluster variation in both time-varying IPC and traffic statistics. This makes these clusters ideal for applying the DVFI methodologies that we investigate. We note that although k-means clustering is used in this work, any other clustering methodology that clusters based on the time-varying dynamics of
core utilization and traffic statistics can be implemented instead. It should be noted that the physical configurations of the VFI clusters are fixed, and the term DVFI refers to dynamically tuning the V/F level of each cluster at run time.

4.2 Baseline DVFI Control Policies

To evaluate the effectiveness of the proposed IL-based methodology, we consider two well-known baseline DVFI methods: 1) The feedback controller for DVFI from a recent work [9] as a representative of non-ML approaches; and 2) RL-based techniques from DVFS-based power management strategies [58][87] adapted for DVFI control. In the following sections, we review the pertinent details of each of these baseline DVFI control methodologies.

4.2.1 Feedback Control Approach

In traditional single core/router DVFS heuristics, two application characteristics emerge as the prevailing features when determining V/F decisions: CPU utilization [88] and inter-tile communication [89]. In [9], these techniques are combined to create a feedback control policy for DVFI systems, Figure 4.2 provides an overview of this control policy.

In order to maintain a simplistic hardware, the feedback DVFI controller is designed to work with simple, intuitive inputs. Two key characteristics of the VFI’s workload, VFI average core IPC and average inter-VFI communication are used as input features. To cope with diverse
application characteristics and intra-VFI variability, we profile each application and introduce a bias that takes application-specific behavior into account. In this work, we use this DVFI methodology from [9] as the baseline DVFI control policy.

4.2.1.1 Drawbacks and Challenges

The feedback control policy described above is based on average VFI characteristics (i.e., VFI average core utilization and average inter-VFI link utilization). This technique may not be able to accommodate every core and network element within the VFI, especially for VFIs with high intra-VFI workload variance. Although the introduced bias helps to alleviate this problem, this may not be enough for applications that change dramatically during the runtime.

4.2.1.2 ML as a Solution

By using ML, we are able to leverage its ability to create data-driven models by exploiting additional information as needed. Therefore, for both RL (Section 4.4.2) and IL (Section 4.3) policies, we incorporate more information and use the following input feature set: the VFI average ($f_1$) and maximum core IPC ($f_2$); the VFI average ($f_3$) and maximum inter-VFI communication ($f_4$); and the VFI's previous V/F state ($f_5$). These particular features are chosen in order to capture the average computation and communication patterns of the VFI ($f_1$ and $f_3$), to account for variance of the computation and communication patterns within the VFI ($f_2$ and $f_4$), and to use the contextual knowledge of the previous prediction ($f_5$) to infer the best V/F for the VFI. Therefore, we propose to use RL and IL to learn more sophisticated DVFI control policies.
We emphasize that the ML-based methodologies used (i.e., RL and IL) to learn the DVFI control policies are performed off-line. Subsequently, the learned DVFI control policy is implemented on-chip to actuate the dynamic V/F control.

4.2.2 Reinforcement Learning Approach

Recently, RL techniques have been frequently used to learn efficient DVFS power management policies [58][60][87]. These RL methods learn how to control a system via a trial-and-error approach by interacting with the system and observing the resulting costs or reinforcements.

We formulate the RL problem in the framework of a Markov Decision Process (MDP). A MDP is defined by a set of states $S$, the possible states of the system; a set of control actions $A$, representing the V/F allocations for the VFI; a Markovian transition function $T$ that specifies the dynamics of the system; and the cost function $C$ that assigns real numbers to state and action pairs, i.e., $C(s, a)$ represents the immediate cost of taking an action $a$ in state $s$.

Given an MDP, the goal of an RL algorithm is to return a policy $\pi$ that minimizes the cumulative cost over a sequence of decision steps. Figure 4.3 provides an overview of the RL

![Figure 4.3 Overview of the reinforcement learning approach to DVFI.](image)
approach used in this work. The processes in Figure 4.3 are described in the following subsections.

### 4.2.2.1 Q-Learning

In this work, we adopt the popular RL algorithm Q-Learning [90] to learn which action is optimal for each state. By exploring the state space and the resulting costs from particular actions, Q-Learning attempts to learn a representation or policy $\pi$ that is exploited to minimize the cost. At the core of the Q-Learning algorithm is the Q-values that represent the learned value of taking an action from a particular state. After taking the action $a_t$ for state $s_t$ at time $t$, the Q-value is updated as follows [90]:

$$Q(s_t, a_t) \leftarrow (1 - \alpha_t)Q(s_t, a_t) + \alpha_t \left( C(s_t, a_t) + \gamma \min_a Q(s_{t+1}, a) \right) \quad (4.6)$$

where $\alpha_t$ is the learning rate, $C(s_t, a_t)$ is the cost observed after performing $a_t$ in $s_t$, and $\gamma$ is the discount factor for expected future costs. Using these Q-values, we select actions with an $\epsilon$-Greedy decision-making (explore/exploit) policy to allow the Q-Learner to explore the state-action space $\epsilon$ of the time, while exploiting the currently learned Q-values $(1 - \epsilon)$ of the time.

In order to allow designer-specified trade-offs between energy consumption and execution-time, we adopt the generalized cost function from [87]:

$$C(s, a) = P(s, a) + \lambda \cdot T(s, a) \quad (4.7)$$

where $P(s, a)$ is the power of the system with state $s$ and action $a$, $T(s, a)$ is the execution time with state $s$ and action $a$, and $\lambda$ is a user-defined tradeoff parameter. Since we are mainly concerned with the trade-offs between the different possible actions, we want $P(s, a)$
to represent the relative power of taking action $a$ with respect to all other actions when the
system state is $s$. Therefore, we define $P(s, a)$ as:

$$P(s, a) = \frac{m(a_V, a_f)}{m(V_{\text{nom}}, f_{\text{nom}})} \quad (4.8)$$

where $a_V$ and $a_f$ are the voltage and frequency after taking action $a$, $V_{\text{nom}}$ and $f_{\text{nom}}$ are the
nominal voltage and frequency, and $m(V, f)$ is a model representing the proportional
relationship between the $V/F$ values and power. This way, Eq. (8) captures the cost of
performing an action with respect to all other actions.

Since it is difficult to determine the overall system execution time from any single VFI, we
attempt to estimate the possible repercussions of each VFI’s action on the system execution
time. To this end, the relevant information is the maximum utilization (an indicator for
possible computational bottlenecks, $B_{\text{comp}}$) and the level of inter-VFI traffic (an indicator for
possible communication bottlenecks, $B_{\text{comm}}$). With this in mind, we define $T(s, a)$ as:

$$B_{\text{comp}} = \max(s_{U_{\text{max}}} - U_{\text{ref}}, 0)$$
$$B_{\text{comm}} = \frac{f_{\text{nom}} - a_f}{f_{\text{nom}}^{-}} s_{T_{\text{avg}}}$$
$$T(s, a) = \phi \cdot B_{\text{comp}} + (1 - \phi) \cdot B_{\text{comm}} \quad (4.9)$$

where $\phi$ is a parameter that trades off the computation and communication impact on $T$,
$s_{U_{\text{max}}}$ is the VFI maximum core utilization for state $s$, $U_{\text{ref}}$ is a reference that indicates at
what point the maximum utilization may indicate a computational bottleneck, $f_{\text{nom}}$ is the
nominal frequency, $s_{T_{\text{avg}}}$ is the average inter-VFI traffic for state $s$. Intuitively, the first part
of Eq. (9) represents the chance for creating a computational bottleneck; the more VFI maximum utilization exceeds the reference utilization, the more likely an execution time penalty will occur. On the other hand, the second part of this equation represents the chance for creating a communication bottleneck; the lower the frequency, the longer the inter-VFI flits take to reach their destination. Subsequently, a frequency-ratio-weighted flit count takes into account how many messages and how greatly these messages were affected by the action taken.

We can then achieve various power-performance trade-offs by sweeping the value of $\lambda$ across a wide interval (Eq. (7)). Intuitively, high values of $\lambda$ will result in policies that favor (i.e. maximize) performance, while lower values of $\lambda$ will favor (i.e., minimize) power. Existing work on applying RL to trade-off speed and accuracy of structured prediction [91] follows a similar approach.

4.2.2.2 Drawback and Challenges

The main drawbacks of RL methodology are as follows: 1) RL methods do not scale for large-scale manycore systems. Since learning is done via trial-and-error, the learner needs to sufficiently explore the state space to learn high-quality control policies. Therefore, the computational complexity can be very high for large state spaces. Additionally, the hardware overhead to store the DVFI control policies also scales with the state space and can become unreasonably large; 2) It is hard to select the value for parameter $\lambda$ in the cost function to achieve the desired power-performance trade-off. Hence, we need to run the RL method multiple times to tune the value of $\lambda$, which can be computationally expensive; and 3) The
RL method learns independent controllers that do not take into account the inherent dependencies between VFIs.

Our proposed solution based on IL aims at mitigating these limitations of RL approaches.

4.3 Imitation Learning-based Approach

We formulate and solve the problem of learning a DVFI controller in the framework of imitation learning (IL) [92][93]. To the best of our knowledge, this is the first attempt to utilize this promising learning technique for solving an important optimization problem in multicore systems design. Figure 4.4 provides an overview of the IL approach used in this work. The processes in Figure 4.4 are described in the following subsections.

4.3.1 Overview of Imitation Learning

Imitation learning is considered to be an exponentially better framework than RL for learning sequential decision-making policies [94][95], but assumes the availability of a good Oracle (or expert) policy to drive the learning process. At a very high-level, the difference between IL and RL is the same as the difference between supervised learning and exploratory learning [94][95]. In the supervised setting, the learner is provided with the best action for a given state. In the exploratory setting, the learner only receives weak supervision.
in the form of immediate costs and needs to explore different actions at each state, observe the corresponding costs, and learn from past experiences to figure out the best action at a given state. From a complexity perspective, when it is possible to learn a good approximation of the expert, the amount of data and time required to learn an expert policy is polynomial (quadratic or less) in the time horizon (i.e., number of decision steps) [62]. However, near-optimal RL is intractable for large state spaces [94][95]. Our work is targeted for large system sizes where the state space grows exponentially with the number of cores. Therefore, RL methods may not scale very well.

In traditional IL, expert demonstrations are provided as training data (e.g., demonstrations of a human expert driving a car), and the goal of the learner is to learn to imitate the behavior of an expert performing a task in a way that generalizes to similar tasks or situations. Typically, this is done by collecting a set of trajectories of the expert's behavior on a set of training tasks. Then, supervised learning is used to find a policy that can replicate the decisions made on those trajectories. Often the supervised learning problem corresponds to learning a classifier or regressor to map states to actions [62][92][93].

4.3.2 Challenges of Imitation Learning for DVFI Control

For our DVFI control problem, the expert corresponds to an Oracle controller that provides the supervision on how to make good control decisions for V/F tuning. We list the two main challenges in applying the IL framework to learn DVFI control policies, and provide the corresponding solutions below.
4.3.2.1 Oracle Construction

In traditional IL, expert demonstrations are used as the Oracle policy to provide supervision for the IL process. Unfortunately, we don’t have any training data for the DVFI control problem. For VFI-enabled systems, we define the Oracle controller as the policy that allocates the best V/F levels for each VFI to minimize the overall power consumption subject to some performance constraints. The total number of candidate Oracle policies is \((L^k)^N\), where \(N\) is the length of the system trajectory (number of time intervals or windows), \(k\) is the number of VFIs, and \(L\) is the number of V/F levels for each VFI. A naïve approach would be to evaluate each candidate Oracle policy by running the application with the corresponding sequence of joint V/F allocations in terms of both Energy-Delay Product (EDP) and execution time; and select the feasible (i.e., satisfies \(p\%\) execution time penalty constraint) policy that has the lowest EDP. Clearly, the problem of computing an optimal Oracle policy that has the minimum EDP subject to a particular performance constraint (e.g., \(p\%\) execution time penalty) is intractable due to the combinatorial space of V/F assignments for all VFIs along the system trajectory.

Since the learning process is offline, we access the future system states and perform a look-ahead search to find the best joint V/F allocation for all VFIs. This is accomplished by running the application with different V/F assignments to optimize the global performance (i.e., EDP of the system). To overcome the computational challenge and closely approximate optimality, our key insight is to perform local optimization followed by aggregation for global optimization.
First, we compute the optimal V/F (which minimizes EDP) for each VFI, at each control epoch, for \( m \) different execution time penalties (e.g., 0%, 5%, and 10% for \( m = 3 \)). This gives us \( m \) different V/F assignments for each time step. Second, for every \( n \) control epochs, we compute the best V/F assignments by performing an exhaustive search over all possible combinations of local optima from the first step (\( m^n \)). Note that it is easy to find a small \( m \) that works well in practice, but both the quality and computation time of the Oracle depends on \( n \).

To estimate the power and performance values under different V/F levels, we employ the power and performance model from [96], which was shown to be very accurate with a Root-Mean-Squared-Percentage-Error (RMSPE) of only 4.37%. In our experiments, we found that beyond a small value of \( n \) (\( n = 15 \)), the quality of the Oracle (evaluated using EDP) saturates. These results indicate that we can construct high-quality Oracle policies in a computationally-efficient manner, which is important for the effectiveness of our imitation learning approach.

**4.3.2.2 Fast and Accurate Decision-Making**

In traditional IL, at any given state, the number of control actions is very small. This means learning and decision-making can be performed efficiently by leveraging standard supervised learning techniques (e.g., classifiers and regressors). Unfortunately, in the DVFI control problem, there are a large number of joint V/F actions for all VFIs, which poses significant scalability challenges for learning and decision-making.

We formulate the problem of DVFI control decision-making as a structured output prediction task [61]. This is the task of mapping from an input structured object (a graph with features
on the nodes and edges) to an output structured object (a graph with labels on the nodes and edges). In DVFI control, the input graph contains a node for each VFI with edges defined based on the inter-VFI traffic density. The label for each node in the output graph represents the V/F allocation of the corresponding VFI. The edges in the graph correspond to the structural dependencies between different input and output variables. Figure 4.5 illustrates the structured output prediction task corresponding to the DVFI control decision-making. It is important to note that the DVFI control policy learning problem is a significant generalization of the traditional classification problem (i.e., classification with exponential number of output classes).

A standard approach to structured prediction is to learn a function $\pi$ for scoring a candidate output graph for each input graph [61]. Given such a scoring function $\pi$ and a new input graph, the output computation involves solving the so-called Argmax inference problem, which is to find the highest scoring output graph for the corresponding input graph. Unfortunately, solving the Argmax inference problem is often intractable except for a few special cases, such as for input graphs in the form of a chain or tree with small width [61]. The main challenge for learning and decision-making for DVFI control is choosing the best V/F from the large space of all possible V/F assignments ($L^k$, where $k$ is the number of VFIs).
and L is the number of V/F levels for each VFI). This is particularly challenging for our DVFI control problem: we are trying to predict the joint V/F allocation for all VFIs to save energy, but if the computation for making the prediction consumes more energy than the energy saved, we don't gain anything. Therefore, we want a fast and accurate predictor whose energy overhead is very small when compared to the overall energy savings due to DVFI control.

To address the above-mentioned challenge, we learn pseudo-independent structured controllers to achieve efficiency without losing accuracy. Specifically, we assign one controller for each VFI and learn k controllers. These controllers are pseudo-independent in the sense that each controller predicts the V/F allocation for only a single VFI but has the context of previous predictions from all the controllers and the structural dependency information with other VFIs, while making predictions. Intuitively, the different controllers are trying to help each other via the additional contextual information. We encode the structural dependencies for each VFI into features that will be used by each controller to make its V/F prediction.

4.3.3 Imitation Learning Methodology

4.3.3.1 Exact Imitation Algorithm

Our goal is to learn a controller that closely follows the Oracle in terms of V/F allocation. Given an Oracle controller \( \pi^* \), we want to learn a controller \( \pi \) that minimizes the regret; in this work, we define the regret of a controller \( \pi \) as:

\[
REGRET(\pi) = EDP(\pi) - EDP(\pi^*)
\] (4.10)
We reduce the problem of learning the controller to a series of simple regression problems. This reduction allows us to leverage powerful off-the-shelf regression learners.

In the exact imitation training approach, we run the given application on the VFI-enabled system using the Oracle controller $\pi^*$. Without loss of generality, let $(s_1, s_2, \ldots, s_N)$ correspond to the sequence of system states on the execution trajectory, where given any integer $t, t \in [1, N)$, $s_{t+1}$ is obtained when the Oracle was used to make V/F assignments for each VFI at system state $s_t$. For each state $s_t$ on the execution trajectory, we collect one regression training example (system input/output pairs) for each VFI controller: state $s_t$ features and the Oracle V/F allocation correspond to the input and output, respectively. For each controller, the set of aggregate imitation examples collected over all states are divided into ten folds. We separate out three randomly selected folds for validation and use the remaining seven folds for training. These sets are then given to a regression learner (e.g., linear regression or regression trees) to learn the controller.

Unlike standard supervised learning problems that assume IID (Independent and Identically Distributed) input examples, our controller learning problem is non-IID because the next state $s_{t+1}$ depends on the decision of the controller at the previous state $s_t$. Therefore, controllers learned via exact imitation can be prone to error propagation: errors in the previous state may result in a next state that is very different from the distribution of states the learner has seen during the training, and contributes to more errors. To address the error-propagation problem associated with exact imitation training, we employ an advanced IL approach called DAgger [62] as explained in the next subsection.
4.3.3.2 DAgger Algorithm

Figure 4.6 provides the pseudocode for learning the DVFI control policy with DAgger. The key idea behind DAgger is to generate additional training data so that the learner is able to learn from mistakes. DAgger is an iterative algorithm, where each iteration adds imitation data to an aggregated data set. The first iteration follows the exact imitation approach. After each iteration, we learn controllers $\pi = (\pi_1, \pi_2, \ldots, \pi_k)$ using the current aggregated data. Subsequent iterations run the given application on the VFI-based manycore system using the learned controller $\pi$ to generate a trajectory of system states. At each decision along this trajectory, we add a new imitation example if the control decision of the learned controller $\pi$ is different from the Oracle controller $\pi^\ast$. Therefore, later iterations allow DAgger to learn from states visited by errors in the learned controller and correct its mistakes using the input.

```
1 Input: ARCH = Manycore Architecture
2 APP = Application
3 $\pi^\ast$ = Oracle control policy
4 $d_{\text{max}}$ = DAgger iterations
5 Output: $\pi$, the learned control policy
6 Initialization:
7 $D$ = regression training data from exact imitation
8 $\pi$ = control policy from exact imitation data
9 For each DAgger iteration $j = 1$ to $d_{\text{max}}$
10     Run APP with control policy $\pi$
11     For every mistake made by $\pi$ //Compare with Oracle
12         Add regression example to $D$ //Aggregate data
13     End For
14     Learn control policy $\pi$ with $D$
15 End For
16 Return the best control policy over all iterations
```

Figure 4.6 Pseudo-code of Dagger.
from the Oracle controller $\pi^*$. Figure 4.7 illustrates the DAgger algorithm for learning DVFI control policies. This iterative learning approach comes with strong theoretical guarantees that can learn policies with very small regret (i.e., closely follow the Oracle) [62]. It is important to note that the sequence of policies learned during the DAgger iterations may not have a monotonic convergent behavior in general. The theoretical result only guarantees that there exists a policy with small regret among the sequence of policies [62]. Similar to the exact imitation algorithm, we divide DAgger’s aggregate dataset into ten folds. We separate out three randomly selected folds for validation and use the remaining seven folds for training. We select the best policy over all the iterations based on the performance on the validation dataset.

### 4.4 Experimental Results

In this section, we first present the VFI-based platform and baseline configurations used in analyzing the DVFI control methodologies considered in this work. Then, we discuss the design and relevant tradeoffs of each of the DVFI control policies outlined in Section 4.2 and 4.3. Lastly, we provide a full-system analysis in terms of execution time and energy dissipation using popular CMP workloads.
4.4.1 Experimental Setup

We employ GEM5 [97], a full system simulator, to obtain detailed processor and network-level information. In all the experiments, we consider a system running Linux within the GEM5 platform in full-system (FS) mode. Since running Linux in FS mode with Alpha cores is limited to a maximum of 64 cores, we use a 64-core system.

The MOESI_CMP_directory memory setup is used with private 64KB L1 instruction and data caches, and a shared 8MB (128 KB distributed per core) L2 cache. Four SPLASH-2 benchmarks: FFT, RADIX (RAD), LU, and WATER (WAT) [98]; and four PARSEC benchmarks: CANNEAL (CAN), FLUIDANIMATE (FLU), DEDUP (DED), and VIPS [99] are considered. The performance statistics generated by the GEM5 simulations are incorporated into McPAT to determine the power values [84].

In this work, we consider nominal range operation in the 28nm technology node. Hence, the adopted DVFI strategy uses discrete V/F pairs that maintain a linear relationship. Eight V/F pairs are considered (in Volts/GHz): 1.0/3.0, 0.95/2.75, 0.9/2.5, 0.85/2.23, 0.8/1.94, 0.75/1.64, 0.7/1.33, and 0.65/1.02. By using on-chip voltage regulators with fast transitions, latency penalties and energy overheads due to voltage transitions can be kept low. We estimate the energy overhead introduced by the regulators due to voltage transition as:

$$E_{\text{regulator}} = (1 - \eta) \cdot C_\text{filter} \cdot |V_2^2 - V_1^2| \quad (4.11)$$

where, $\eta$ is the power efficiency of the regulator, $C_\text{filter}$ is the regulator filter capacitance, and $V_1$ and $V_2$ are the two voltage levels before and after the voltage transition [100]. Both
the regulator switching energy and DVFI controller energy are taken into account when analyzing the overall system energy.

<table>
<thead>
<tr>
<th>Benchmark</th>
<th>VFI1</th>
<th>VFI2</th>
<th>VFI3</th>
<th>VFI4</th>
</tr>
</thead>
<tbody>
<tr>
<td>CANNEAL</td>
<td>22</td>
<td>22</td>
<td>16</td>
<td>4</td>
</tr>
<tr>
<td>DEDUP</td>
<td>40</td>
<td>16</td>
<td>4</td>
<td>4</td>
</tr>
<tr>
<td>FFT</td>
<td>29</td>
<td>23</td>
<td>7</td>
<td>5</td>
</tr>
<tr>
<td>FLUID</td>
<td>40</td>
<td>16</td>
<td>4</td>
<td>4</td>
</tr>
<tr>
<td>LU</td>
<td>32</td>
<td>24</td>
<td>4</td>
<td>4</td>
</tr>
<tr>
<td>RADIX</td>
<td>37</td>
<td>19</td>
<td>4</td>
<td>4</td>
</tr>
<tr>
<td>VIPS</td>
<td>30</td>
<td>26</td>
<td>4</td>
<td>4</td>
</tr>
<tr>
<td>WATER</td>
<td>41</td>
<td>15</td>
<td>4</td>
<td>4</td>
</tr>
</tbody>
</table>

We have also incorporated the synchronization delay associated with the mixed-clock/mixed-voltage (MCMV) FIFOs at the boundaries of each VFI into the simulations [101]. For each VFI, we have incorporated the energy overhead of generating additional clock signals [102] assuming a traditional H-tree clock distribution using our target ST 28nm technology, the voltage regulator’s switching and the MCMV FIFOs.

Any manycore platform requires a suitable NoC architecture as the communication backbone. We have already demonstrated the efficacy of an emerging wireless NoC (WiNoC) architecture in designing a VFI-based manycore system [9][45]. Hence, we consider the WiNoC as the communication backbone in this work. It should be noted that instead of WiNoC, any other NoC architecture can be used for comparative performance evaluation of the DVFI policies.
4.4.2 VFI System and Baseline DVFI Configuration

In this work, we consider four VFI clusters while imposing a minimum VFI cluster size of four cores [9]. By using the k-means algorithm, we cluster the cores to minimize each VFI’s intra-cluster variation in the time-varying IPC and traffic statistics [9]. We reiterate that the analysis of VFI clustering methodologies lies beyond the scope of this paper and any clustering methodology could be used to similar effect. Table I shows the cluster size of each VFI. For instance, for the FLUID benchmark, VFI 1 has 40 cores, VFI 2 has 16 cores, and VFI 3 and 4 have 4 cores.

An important parameter in the DVFI controller is the V/F switching period $T_{SVF}$. In this work, we ensure that the DVFI controller delay is less than 1% of $T_{SVF}$ so that the controller impose significant delay. Table II presents the delay for all controllers considered in this work, we then set the lower bound $1\mu s \leq T_{SVF}$. Then, we sweep $T_{SVF}$ throughout the range $1\mu s \leq T_{SVF} \leq 1ms$ and choose the value of $T_{SVF}$ that minimizes the EDP of the Oracle for each benchmark. The optimal values of $T_{SVF}$ were found to be between $21\mu s$ and $227\mu s$. For each benchmark, we use the same $T_{SVF}$ for all the different DVFI techniques.

To ensure that the controller has up-to-date data when making its decisions, each core sends its traffic and IPC data at $T_{SVF}$ intervals. Due to the simplicity of the application statistics used, each core only sends traffic and IPC data to its respective DVFI controller. This core-to-controller traffic is insignificant when compared to the total amount of traffic traversing the system. In fact, we have measured that the total traffic generated to the DVFI controllers contribute less than 0.05% of the total traffic for all benchmarks considered. The features for the DVFI policies are calculated within the VFI controllers.
4.4.3 Reinforcement Learning Implementation

In order for the Q-Learning algorithm to converge to a policy that satisfies the objectives, the parameters need to be carefully chosen. Similar to many other previous works on RL [87], we choose to implement a decaying learning rate $\alpha$ as a function of the learning iteration number. Logically, we initially start with low confidence in our policy and want to learn or change the policy aggressively. Over time, our confidence in the current policy increases, allowing us to reduce $\alpha$ and the changes made through the Q-Learning process. For the discount factor, we use a static value of $\gamma = 0.9$, placing a fairly high importance on future costs. We also utilize an $\epsilon = 0.1$, for the $\epsilon$-Greedy decision-making (explore/exploit) policy to allow the Q-Learner to explore the state space 10% of the time while exploiting the current learned policy 90% of the time.

For the cost function, we need to choose $m(V, f)$, $\phi$, $U_{ref}$, and $\lambda$ in order to achieve a policy that reflects our constraints and optimization objectives. In this work, since Eq. (8) shows the relationship between V/F values and power, we use $m(V, f) \propto V^2 f$ to simply capture the relative cost of taking various V/F decisions. Since both dynamic and total power are monotonically decreasing and convex with respect to voltage, this simple equation captures the correct relative cost associated with policy actions for changing V/F values. In this work, we choose the $U_{ref}$ for each benchmark to be the average of the system’s maximum core utilization (IPC) running at nominal V/F.

$$U_{ref} = \frac{\sum_n \max_c U_c(n)}{N}$$
where $U_c(n)$ is the utilization of core $c$ during control interval $n$, $N$ is the total number of control intervals.0

Intuitively, $U_{ref}$ is representative of the level at which we want to maintain the VFI maximum utilization in order to manage similar performance to the nominal case. To balance $T$'s (Eq. (9)) dependence on the potential computational and communication bottlenecks, we set $\phi$ to be the ratio of computation to communication in the application. Lastly, $\lambda$ serves as our power/performance trade-off parameter. We sweep $\lambda$ to obtain a set of policies at different power/performance trade-off points.

In Figure 4.8, as an example, we demonstrate how energy and execution time are traded off with different values of $\lambda$ for two benchmarks, i.e., FFT and CANNEAL. In this work, we minimize the energy dissipation for a given limit on the performance penalty, which in this case is 5%. For our situation, $\lambda = 100$ and $\lambda = 500$ are suitable selections for FFT and CANNEAL respectively. We similarly choose $\lambda$ for all other benchmarks. However, it should be noted that in CANNEAL, if a higher performance penalty can be tolerated, $\lambda = 10$ is a
much more desirable energy/performance tradeoff. This demonstrates the ease with which the cost function can be tuned with $\lambda$ to allow easy trade-off analysis.

In order to keep the representation simple, we implement the Q-learning policies in the form of state-action tables as done in prior work. Simply put, given an input VFI state, the table will return a row of Q-values corresponding to the policy’s evaluation of each action at that input VFI state. However, in order to use the table representation, the input state must be quantized from continuous values to discrete values. When determining the number of discrete levels or Quantization Granularity (QG) for the input features, two important factors must be considered: the quality of the resulting quantized policy and the overhead of implementing this policy. Due to the inherent loss of information related to quantizing the input features, low QG will make it harder to learn high-quality DVFI policies. On the other hand, increasing QG would result in quickly increasing table overhead resulting from state space explosion.

To evaluate the QG’s effect on the quality of potential policies, we use an Oracle policy (we utilize the policy created for IL in Section 4.4.4.1) to learn a decision table-based policy for
each level of QG and compare their accuracy with respect to the Oracle. To evaluate the area overhead, we implement a Q-Learning Table for each level of QG, a lookup table with enough entries (8-bit) to accommodate all states possible (a function of the number of actions and quantization levels), and evaluate it using Synopsys™ Design Vision in ST 28nm CMOS process. Figure 4.9, we demonstrate the trade-off between area overhead and quality of the policy (testing error) averaged over all eight benchmarks considered in this work. After a QG value of six, the testing error starts to saturate while the overhead still increases rapidly. Consequently, we choose to implement the RL policy with QG of six.

4.4.4 Imitation Learning Implementation

In this section, we discuss the implementation details and evaluate the accuracy and efficiency of multiple IL-based DVFI control policies in terms of the prediction accuracy and the implementation overhead required for each decision-making module. First, we describe the creation of the Oracle that is to drive the learning of IL-based policies. Second, we discuss the various candidate regression models that we evaluate for DVFI controller representation. Third, we determine the prediction accuracy by comparing the outputs for each of the DVFI policies against the Oracle. Lastly, we demonstrate the utility of DAgger in improving the accuracy of the learned policies.

4.4.4.1 Oracle Policy Implementation

In order to allow the DVFI policy to save significant energy without large penalties, we use 5% for the allowable execution time penalty. As discussed in Section 4.4.4.1, the values of $m$ (the number of local execution time penalties evaluated) and especially $n$ (the number of time steps over which an exhaustive search is performed), dictate the quality of the Oracle
policy. In our experiments, we employed $m = 3$ (0%, 5%, 10% penalties) and found that beyond a small value of $n$, the quality of the Oracle saturates (shown in Figure 4.10). We consider EDP as the relevant measure to evaluate the quality of the Oracle. These results indicate that we can efficiently construct high-quality Oracle controllers, which is important for the effectiveness of our IL approach. For our experiments, we create the Oracle policy using values $n = 15$ and $m = 3$.

### 4.4.4.2 Regression Learner

Our goal is to learn a function to predict the V/F of a VFI by examining its characteristics or features. We can employ any regression-learning algorithm, such as $k$ nearest neighbor ($k$-NN), linear regression (LR), support vector regression (SVR), and regression tree (RT). However, due to performance, energy, and area overhead constraints of an on-chip environment, some regression learners are not suitable for DVFI control. For example, $k$-NN and non-linear SVR based predictors need large area for on-chip implementation (store
subset of training examples), and cause large delay and consume more power due to high prediction time (distance/kernel computation for each of the stored training examples).

On the other hand, LR needs to store only one weight for each of the features, and each control decision computation involves performing multiplications and additions; the number of such operations is linear in the number of features. RT can efficiently represent non-linear functions in the form of trees with small depth and each control decision computation involves a linear number of comparisons with the depth of the tree.

In this work, we explore RT and LR as the base regression learners to evaluate our IL-based DVFI control methodology. We employ Weka [103], an open source ML toolkit to learn our LR and RT policies.

4.4.4.3 DVFI Accuracy with Exact-Imitation and DAgger

To evaluate the prediction accuracy of the controller, we compute the Mean Absolute Percentage Error (MAPE) across all DVFI decisions. In this work, we consider the error as the difference between the controller’s predicted voltage and the Oracle’s voltage. In Figure 4.11, we show the MAPE and one standard deviation away from the mean (error bars) for each benchmark, as well as the average across all benchmarks (AVG) for both LR and RT policies. As we can see, both RT and LR policies closely follow the Oracle, with less than 10% error for all benchmarks considered with only one exception, LR with exact-imitation learning on FFT. However, by applying the Dagger algorithm we are able to greatly reduce LR’s MAPE for FFT. Also, as illustrated in Figure 4.11, RT outperforms LR in every benchmark (0.96% on average).
To overcome the deficiencies of exact imitation training, i.e., the error propagation problem, we implemented the DAgger algorithm. Training examples generated during exact imitation correspond to ideal conditions (considering all the past DVFI decisions to be correct). However, when we execute the learned policy, we cannot guarantee these conditions to hold. By using DAgger, we train the policies to recover from potential mistakes caused by the imperfect policy learned via exact imitation. We ran DAgger for five iterations and picked the best performing policy using a validation dataset (we create this validation dataset by dividing DAgger’s aggregated dataset into ten folds. We separate out three randomly selected folds for validation and use the remaining seven folds for training). We noticed that increasing the number of iterations did not have a noticeable improvement in the results.

In Figure 4.11, we demonstrate the MAPE for the DAgger-enabled ML policies. By using DAgger, both ML policies are able to more closely follow the Oracle compared to their exact-imitation learning counterparts. Also, as demonstrated in Figure 4.11, RT with DAgger
outperforms LR with DAgger in every benchmark in MAPE (0.95% on average). Therefore, for the rest of this work, we use DAgger-enabled RT as our IL policy.

### 4.4.5 Comparison of DVFI Policies

In this section, we provide a detailed comparison between the different DVFI policies described in this work. We consider the cost of implementing each of these DVFI policies, their respective full-system performance under the given constraints, and the maximum achievable quality of each of the policies.

#### 4.4.5.1 Cost of Hardware Implementation

The DVFI controllers are synthesized from an RTL-level design using an ST 28-nm CMOS process and Synopsys™ Design Vision. Also, the circuits have been designed targeting the worst-case scenario, a VFI that contains the full system of 64-cores. Figure 4.12 shows a high-level view of the DVFI controller structure. We use the same general structure for each DVFI

<table>
<thead>
<tr>
<th>Policy</th>
<th>Area ($mm^2$)</th>
<th>Delay (ns)</th>
<th>Energy/Decision (nJ)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Feedback</td>
<td>0.0129</td>
<td>1.63</td>
<td>0.0382</td>
</tr>
<tr>
<td>RL-QG4</td>
<td>0.0507</td>
<td>1.44</td>
<td>0.1591</td>
</tr>
<tr>
<td>RL-QG6</td>
<td>0.2016</td>
<td>1.92</td>
<td>0.6380</td>
</tr>
<tr>
<td>RL-QG8</td>
<td>0.5858</td>
<td>2.08</td>
<td>1.9371</td>
</tr>
<tr>
<td>IL-RT</td>
<td>0.0229</td>
<td>1.37</td>
<td>0.0442</td>
</tr>
<tr>
<td>IL-LR</td>
<td>0.0203</td>
<td>1.84</td>
<td>0.0403</td>
</tr>
</tbody>
</table>

Figure 4.12 High-level DVFI controller structure.
policy: (i) the network interface receives policy parameter updates and core utilization/traffic information, (ii) the register table holding the core utilization/traffic information until it is ready to be used, (iii) the VFI statistics calculation module generate the necessary information for the DVFI policy (e.g., average and maximum utilization), (iv) the counter module notifies the VFI statistics calculation module every $T_{SVF}$ cycles, and (v) the DVFI policy module uses the VFI statistics to generate the V/F decision to send to the regulator. The only differences between DVFI techniques is (i) the DVFI policy module and (ii) the VFI statistics calculation module which generates only the statistics needed by the particular DVFI technique. We present the area, delay, and energy per decision in Table II for the feedback control policy (feedback), three RL policies at varying levels of QG (RL-QG4, RL-QG6, RL-QG8), and the LR- and RT-based IL policies (IL-LR and IL-RT). By using Weka, all learned RT policies have a maximum depth of six. Therefore, the hardware RT is designed for a tree depth of six. It can be seen that the overhead for feedback and both IL policies are equally small; the feedback controller is able to outperform in area overhead due to only using two features (compared to five for RL and IL). RL on the other hand, even for the low granularity case (RL-QG4), requires much more energy and area than both feedback and IL policies. Also, for the RL and IL policies chosen in Sections 4.4.3 and 4.4.4 (RL-QG6 and IL-RT), IL is able to reduce the area overhead by 8.8x. The delay and energy/decision for all policies are small and do not present any significant overhead to implementing the DVFI control mechanism. However, it should be noted that IL-RT has the lowest delay and the IL-based policies take 3-40x less energy/decision than RL while remaining comparable to the Feedback controller.
4.4.5.2 Computational Overhead

For both RL and IL, the different learning processes result in varying computational overheads to learn the corresponding DVFI policy. In Figure 4.13, we illustrate the computational overhead in terms of the time it takes to learn each respective policy, normalized to IL. We are able to learn IL policies 2.5-3.6x faster than RL policies. This is due to the amount of iterations that RL has to complete in order to converge to a good policy. On the other hand, due to its simplistic nature, the feedback control policy has negligible computational overhead when compared to both ML-based policies. However, this simplistic nature comes at the cost of achievable performance as demonstrated in the next sub-section.

![Figure 4.13 Normalized computation time w.r.t. IL to create the DVFI policies.](image)

4.4.5.3 Full-System Performance Evaluation

In this section, we do a comparative full-system analysis among the DVFI policies considered in this work. We analyze the total execution time (performance metric) and the full-system energy dissipation considering the SPLASH-2 and PARSEC benchmarks.

In Figure 4.14, we show the execution time and energy dissipation figures of the Feedback Controller (FB), RL-QG6 (RL), DAgger-enabled RT (IL) with respect to FB. We also include the average across all benchmarks (AVG).
In Figure 4.14 (a), we can see that both ML-based policies are able to perform equally or slightly better than FB for the benchmarks considered. It is in the energy dissipation results where the ML-based policies demonstrate their strength in DVFI decision-making. It can be seen that the IL and RL DVFI policies are able to reduce the average energy dissipation of the FB DVFI policy by 9.5% and 4.8% respectively. Due to the difficulties in learning good policies using RL, RL does not consistently beat the simplistic FB DVFI policy. For example, RL loses to FB for LU and performs exactly the same for WAT in energy dissipation. Significantly, IL is able to outperform all other policies in every benchmark considered.

4.4.5.4 Achievable Performance and the Pareto Frontier

One important aspect of analyzing the DVFI control policies is their potential achievable performance. To help estimate this, we demonstrate the execution time and energy dissipation trade-off in Figure 4.15 for benchmarks FFT and FLUID as examples. Due to the nature of the RL tuning process (sweeping $\lambda$), a set of execution times and energy tradeoffs are already generated for each $\lambda$ investigated. In order to give a clear picture, we also generate additional FB policies by varying the bias and additional IL policies at various performance penalties along the performance penalty range seen by the RL and FB sets.

![Figure 4.14](image-url)
These policy sets outline the Pareto Frontier of each of the DVFI methodologies. It can be seen in Figure 4.15 that, consistent with the results in previous section, IL is able to consistently outperform both RL and FB. IL is able to favorably shift the Pareto Frontier to the lower left, achieving lower energy at each performance penalty point. For the two examples in Figure 4.15, at the 5% performance penalty, IL reduces the energy dissipation over RL by 5.5% (FFT) and 3.1% (FLUID). Significantly, this gap widens if higher performance penalties are allowed. IL is able to reduce the energy dissipation over RL by 22.4% near the 15% performance penalty point for FFT and 28.7% near the 20% performance penalty point for FLUID. Also, consistent with the results in previous section, RL does not always outperform FB, with various points performing worse than FB. For example, in FFT (Figure 4.15), FB is able to outperform RL near the 15% performance penalty point.

To illustrate the relevant comparative performance between the two ML-based policies considered in this work, we show the DVFI controller area overhead, computation time overhead to learn the DVFI control policy, and full-system energy for a 5% allowable execution time penalty in Figure 4.16. Each RL point represents a different benchmark.
normalized to the benchmark’s respective IL performance. It can be seen that IL is able to outperform RL in full-system energy at significantly less computational and area overhead. IL’s computational overhead and full-system energy advantages over RL was determined to be statistically significant through paired t-test.

We do acknowledge that FB may be desirable to some designers due to its computational and hardware overhead qualities. However, this computational overhead is incurred only once per application during the offline learning process and the hardware overhead of IL only accounts for 0.0229% of the chip area for a four-VFI, 400 mm² system. Due to the combined generally favorable qualities in incorporating more VFI information, hardware overhead, computational overhead, execution time, energy dissipation, and flexibility for incorporating desired power/performance trade-off of IL over its DVFI brethren, we demonstrate that IL is a promising alternative to create efficient DVFI controllers.

![Figure 4.16 DVFI comparison in (a) Computation overhead for learning DVFI policy vs. full-system energy for a fixed 5% allowable execution time penalty, and (b) DVFI controller area overhead. Each FB RL point in (a) is a different benchmark normalized w.r.t. to IL (1,1).](image)

101
4.5 Summary

In VFI-based systems, it is difficult to incorporate all the information needed to make good control decisions. In the context of VFI-based systems, V/F tuning decisions must intelligently select actions that conform to the requirements of all elements of the VFI. When using simple heuristics, a tradeoff is required to either incorporate the information needed to make good decisions and incur large implementation overheads or choose a more simplistic representation that leads to a sub-optimal policy. By using ML, we can find appropriate middle ground in terms of complexity and quality. Although RL has been utilized to solve similar problems in the DVFS space, IL has been shown as a promising learning alternative.

In this dissertation, for the first time, we have made the following contributions: 1) Introduced the paradigm of IL for dynamic V/F control in VFI-based manycore systems, 2) Extended the RL methodology to VFI platforms, and 3) Using the IL and RL policies, in addition with a simple feedback control-based mechanism, we have conducted an extensive comparative performance analysis of various DVFI control strategies.

We have demonstrated that IL is able to consistently outperform all other DVFI policies by a significant margin. For instance, at 5% performance penalty, IL reduces the energy dissipation over RL by up to 11.5% (LU benchmark) and 5.0% on average. Significantly, this gap widens if higher performance penalties are allowed. At 15% performance penalty, IL reduces the energy dissipation over RL by 22.5% for FFT. Due to IL’s consistent better performance, lower implementation and computational overheads, we conjecture that IL is
an efficient, scalable, and robust methodology to learn DVFI policies for large-scale manycore systems.
5 Future Research Directions

5.1 Efficient Deep Convolutional Neural Network Inferencing for Embedded MPSoC Platforms

Deep Convolutional Neural Network is widely used on embedded platforms for many diverse applications, including self-driving vehicle, military drone, and face recognition. CNN processing involves highly complex computations and hence consumes high energy. This is a major limitation for running these applications on the battery-powered devices. In this section, we briefly introduce a technique that can be used for reducing the high-energy consumption of CNN applications.

5.1.1 Early Classification Approach

As we explained in Section 3.1, CNNs usually consist of multiple convolutional layers, and in each layer, it learns the weight of neurons. Adding layers increases number of learnable neurons and hence in general, improves the accuracy. However, this increases computation

![Figure 5.1 Normalized number of images classified in each layer.](image.png)
complexity, resulting in higher energy consumption. Figure 5.1 shows number of images classified in each layer in a sample CNN architecture. This sample network consists of three convolutional layers and trained for classifying the CIFAR-10 dataset. From Figure 5.1, it is evident that not all the images require full pass through all the convolutional layers to make correct prediction. Some of the images can be correctly classified using only the intermediate layers. Figure 5.1 essentially tells us 50% of images can be classified with very simple CNN. In other word, it is possible to classify an input image at very early stage of CNN processing, and this allows us to terminate CNN computation early. In this section, our aim is to exploit this early classification opportunity and reduce energy consumption by avoiding unnecessary computation.

![Illustration of early classification approach.](image-url)
5.1.2 Methodology

Figure 5.2 illustrates the overall idea of the early classification framework. In the context of CNN, the confidence level indicates the probability that the input image classifies as a specific object. We observe the confidence level at every CNN layer to exploit the early classification opportunity. This is accomplished by inserting an additional fully-connected layer (IFC) at the end of each convolutional layer. As we mentioned in Section 3.1, fully-connected layer outputs classified object with confidence level. If the confidence level is high enough then, we stop executing downstream layers and output the predicted object.
5.2 Application Specific Dynamic Power Management for Mobile Platforms

Dynamic power management mechanism employed in the mobile devices rely on the built-in OS module, namely DVFS governor. The built-in DVFS governor is designed for various applications [104], and it changes V/F of CPU cores simply depending on CPU load. As we demonstrated in Section 4, in order to maximize energy saving under a target performance constraint, the dynamic power management policy should be tuned per application. Modern mobile OS provides six different DVFS governor policies including, interactive, on-demand, performance, conservative, power-save and *userspace*, and the major difference of these policies are a target performance (*i.e.*, allowable execution time penalty). Among these policies, the *userspace* policy provides an interface called *cpufreq* for controlling V/F of CPU cores from user-space. Hence, it is possible to create an application specific power management algorithm by controlling the *cpufreq* interface. We can follow the proposed imitation learning framework (Section 4.3) for designing the power management algorithm for each application.

Most modern mobile MPSoC features Big.Little architecture to enhance energy efficiency. The Big.Little architecture consists of two groups of cores, big and little. Big cores are designed to handle computationally intensive tasks (*i.e.*, game and image processing) while little cores are efficient when they used for simple tasks (*i.e.*, web-browsing). The OS scheduler determines which group of cores to utilize for a given task by monitoring CPU load. If CPU load is higher than a predefined threshold, the OS switch workload to Big cores and vice versa. This mechanism designed for various applications, does not consider the
performance and energy trade-off of specific application and target performance. We can add multiple predefined thresholds for each application and various target performance.
References


