MODELING THE IMPACT OF MEMORY ARCHITECTURE FOR DYNAMIC ADAPTATION IN HPC RUNTIMES

by

MOHAMMAD ALAUL HAQUE MONIL

A DISSERTATION

Presented to the Department of Computer and Information Science
and the Division of Graduate Studies of the University of Oregon
in partial fulfillment of the requirements
for the degree of
Doctor of Philosophy

December 2021

DISSERTATION APPROVAL PAGE

Student: Mohammad Alaul Haque Monil

Title: Modeling the impact of memory architecture for dynamic adaptation in HPC runtimes

This dissertation has been accepted and approved in partial fulfillment of the requirements for the Doctor of Philosophy degree in the Department of Computer and Information Science by:

Allen Malony	Chair
Boyana R. Norris	Core Member
Hank R. Childs	Core Member
Seyong Lee	Core Member
Douglas Toomey	Institutional Representative

and

Krista Chronister	Vice Provost of Graduate Studies

Original approval signatures are on file with the University of Oregon Division of Graduate Studies.

Degree awarded December 2021

© 2021 Mohammad Alaul Haque Monil
This work, including text and images of this document but not including supplemental files (for example, not including software code and data), is licensed under a Creative Commons
Attribution-ShareAlike 4.0 International License.

DISSERTATION ABSTRACT

Mohammad Alaul Haque Monil

Doctor of Philosophy

Department of Computer and Information Science

December 2021

Title: Modeling the impact of memory architecture for dynamic adaptation in HPC runtimes

From the advent of the message-passing architecture in the early 1980s to the recent dominance of accelerator-based heterogeneous architectures, high performance computing (HPC) hardware has gone through a series of changes. At the same time, HPC runtime systems have also been adapted to harness this growth in computational capabilities. Specifically, modern HPC runtime systems have transformed into active entities capable of making dynamic decisions during the execution of an application. These dynamic decisions improve performance, reduce energy consumption, and increase the overall utilization of the underlying HPC hardware. However, a runtime system needs insight into the application and the underlying hardware to make efficient decisions. This dissertation identifies that information gained from modeling the memory architecture is critical for efficient decision-making within the runtime system. After outlining the research challenges associated with dynamic adaptation in HPC runtimes, different modeling approaches are explored to gather insight into the memory architecture of modern HPC hardware.

By studying the evolution of HPC runtime systems for the last 35 years, this dissertation first identifies the opportunities for dynamic adaptation. Then, the research undertaken capitalizes upon these opportunities in the form of four major projects: (1) application and machine agnostic approaches to dynamically adapt the HPX runtime system, (2) modeling memory contention in a heterogeneous system where processors share the same memory, (3) understanding and modeling the handshake between memory access pattern and modern cache hierarchy to statically predict the memory transactions between the last level cache (LLC) and system memory of modern Intel processors, and (4) an exploration of similarities and dissimilarities between Intel CPUs and NVIDIA and AMD GPUs to pave the way to model LLC-device memory transactions in GPUs. This dissertation includes previously published and co-authored material, as well as unpublished co-authored material.

CURRICULUM VITAE

NAME OF AUTHOR: Mohammad Alaul Haque Monil

GRADUATE AND UNDERGRADUATE SCHOOLS ATTENDED:

University of Oregon, Eugene, OR, USA
North South University, Dhaka, Bangladesh
Khulna University of Engineering Technology, Khulna, Bangladesh

DEGREES AWARDED:

Doctor of Philosophy, Computer and Information Science, 2021, University of Oregon
Master of Science, Computer Science and Engineering, 2013, North South University
Bachelor of Science, Computer Science and Engineering, 2006, Khulna University of Engineering Technology

AREAS OF SPECIAL INTEREST:

High Performance Computing
Memory-centric Performance Modeling
Runtime Systems

PROFESSIONAL EXPERIENCE:

Research Collaborator/Affiliate, Oak Ridge National Laboratory Future Technologies Group, 2019-present, Advisor: Seyong Lee, Jeffrey Vetter

Graduate Teaching Assistant, University of Oregon, Advisor: Allen Malony, 2016, 2018, 2021

Graduate Research Assistant, University of Oregon, Advisor: Allen Malony, 2017, 2018, 2019

Lead Manager, Grameenphone Ltd, A Business Unit of Telenor, 2006-20015

GRANTS, AWARDS AND HONORS:

Research Collaboration Grant, Oak Ridge National Laboratory, 2020

PUBLICATIONS:

Monil, M. A. H., Lee, S., Vetter, J. S., and Malony, A. MAPredict: Static Analysis Driven Memory Access Prediction Framework for Modern CPUs. **In preparation.**

Monil, M. A. H., Lee, S., Vetter, J. S., and Malony, A. A Survey of HPC Runtimes: Evolution, Current Trend and Opportunities. **In preparation.**

Lambert, J., Monil, M. A. H., Lee, S., Vetter, J. S., and Malony, A. Exploring Heterogeneous Programming for Future Diverse Exascale Platforms. **In preparation.**

Monil, M. A. H., Lee, S., Vetter, J. S., and Malony, A. (2021). Comparing LLC-memory Traffic between CPU and GPU Architectures. **To appear** in *Redefining Scalability for Diversely Heterogeneous Architectures (RSDHA) Workshop at SC21.*

Monil, M. A. H., Belviranli, M., Lee, S., Vetter, J. S., and Malony, A. (2020). MEPHESTO: Modeling Energy-performance in Heterogeneous SOCs and Their Trade-offs. *International Conference on Parallel Architectures and Compilation Techniques (PACT).*

Monil M. A. H., Lee, S., J. Vetter, and Malony, A. (2020). Understanding the Impact of Memory Access Patterns in Intel Processors. *Memory Centric High Performance Computing (MCHPC) Workshop at SC20.*

Equal Contribution(Wagle B. and Monil, M. A. H.), Huck, K., Malony, A., A. Serio, and H. Kaiser. (2019). Runtime Adaptive Task Inlining on Asynchronous Multitasking Runtime Systems. *International Conference on Parallel Processing (ICPP).*

Monil, M. A. H., Wagle B., Huck, K., and H. Kaiser. (2018). Adaptive Auto-tuning in HPX Using APEX. *A poster at International Conference on Parallel Processing (ICPP).*

Monil, M. A. H., Malony, A., Toomey, D., and Huck, K. (2018). Stingray-HPC: A Scalable Parallel Seismic Raytracing System. *International Conference on Parallel, Distributed and Network-Based Processing (PDP).*

Monil, M. A. H., and Malony, A. (2017). QoS-aware virtual machine consolidation in cloud datacenter. *International Conference on Cloud Engineering (IC2E).*

Malony, A., Monil, M. A. H., Huck, K., Rasmusen, C., Byrnes, J., and Toomey, D. (2016). Towards Scaling Parallel Seismic Raytracing. *International Conference on Computational Science and Engineering (CSE).*

ACKNOWLEDGEMENTS

I want to thank Dr. Allen Malony for his guidance and unwavering support throughout this journey. Starting my PhD in one of the most renowned HPC tool teams without having much background in HPC was difficult. I want to express my gratitude to Dr. Malony for training me with patience. I have learned many things from Dr. Malony over the years. The fact that I felt inspired and could see things clearer after every meeting with him is extraordinary. I also want to thank Dr. Hank Childs and Dr. Boyana Norris for guiding me to reach the end of this journey and being on my committee; especially, the support they provided during my area exam is noteworthy. The pace and interest toward my dissertation topic increased significantly while working with Oak Ridge National Laboratory (ORNL). I want to thank Dr. Seyong Lee and Dr. Jeffrey Vetter for providing me the opportunity to work with ORNL. The internship that I did with Dr. Mehmet Belviranly at ORNL benefited me significantly, and I want to thank Dr. Belviranly for that. I want to acknowledge Dr. Seyong Lee's phenomenal support in defining the plans for my research for the last two years of my PhD. I also want to thank Dr. Lee for continuously guiding me and patiently meeting me every week. I want to thank Dr. Kevin Huck for his mentorship during the Phylanx project. I also want to thank Wyatt Spear for his help in proofreading my submissions. The PhD program is a difficult journey but having friends makes it tolerable. I want to thank Srinivasan Ramesh for being a wonderful friend, helping me, and inspiring me for the last five years. I also want to thank Dr. Jacob Lambert for his friendship and for lending a helping hand whenever I needed it. Last but not least, I want to thank Erik Keever and Steve Moulton for their continuous support to set up the environment necessary for my research.

To my father and my mother, who struggled their whole life so that I can become successful; my wife, for all the sacrifices she made during my PhD journey.

TABLE OF CONTENTS

Chapter Page

I. INTRODUCTION 1

1.1. Working Toward Solutions - This Dissertation 4

1.2. Chapter II — Dynamic Adaptation Techniques and Opportunities in HPC Runtimes 4

1.3. Chapter III — Dynamic Adaptation in HPX Runtime 6

1.4. Chapter IV — Studying Memory Contention in a Heterogeneous System 7

1.5. Chapter V — Static Analysis Framework for Memory Access Prediction in Modern CPUs 9

1.6. Chapter VI — Understanding and Modeling the Impact of Memory Access Patterns in GPUs 10

1.7. A Unified Diagram 11

II. DYNAMIC ADAPTATION TECHNIQUES AND OPPORTUNITIES TO IMPROVE HPC RUNTIMES 13

2.1. Introduction and Motivation 13

2.1.1. Organization 14

2.2. Definition of Runtime Systems 14

2.2.1. Programming Models and Execution models 14

2.2.2. Parallel Programming APIs, Libraries, Language Extensions, and Languages 15

2.2.3. HPC Runtimes 15

2.3. Evolution of HPC Runtimes 17

2.3.1. Before 1985 17

2.3.2. HPC Runtimes in 1985-1995: Message Passing Architecture . 18

Chapter Page

2.3.3. HPC Runtimes in 1996-2005: Shared Memory and Distributed Shared Memory 20

2.3.4. HPC Runtimes 2005-2015: Multicore, Manycore and Heterogeneity 21

2.3.5. HPC Runtimes during 2015-Present: Asynchronous Many Task and Abstraction 22

2.3.6. Reduction and Identification of State-of-the-art 22

2.4. Shared Memory Runtime Systems 23

2.4.1. Cilk Plus 23

2.4.2. TBB 24

2.4.3. OpenMP 24

2.4.4. OmpSs and Nanos++ 24

2.4.5. Qthreads 25

2.5. Task Based Runtime Systems 25

2.5.1. Charm++ 25

2.5.2. HPX 25

2.5.3. Legion 26

2.5.4. OCR 26

2.5.5. Argobots 26

2.5.6. Unitah 27

2.5.7. PaRSEC 27

2.6. GAS Based Runtime Systems 27

2.6.1. UPC 28

2.6.2. Chapel 28

2.6.3. X10 28

2.7. Heterogeneous Runtime Systems 29

- 2.7.1. OpenCL . . . 29
- 2.7.2. OpenACC . . . 29
- 2.7.3. StarPU . . . 29
- 2.7.4. CUDA . . . 30
- 2.7.5. HIP . . . 30

2.8. Runtime Feature Comparison . . . 30

- 2.8.1. Programming API and Model . . . 30
  - 2.8.1.1. Directive-Based . . . 31
  - 2.8.1.2. Expressing Asynchronous Execution . . . 32
  - 2.8.1.3. GPU Programming . . . 33
  - 2.8.1.4. New Language with Special Compiler . . . 33
- 2.8.2. Execution Model . . . 34
  - 2.8.2.1. Task Parallel vs Data Parallel . . . 35
  - 2.8.2.2. Asynchronous Many Task Parallel . . . 35
  - 2.8.2.3. Message Driven vs. Message Passing . . . 36
  - 2.8.2.4. GPU Execution . . . 36
- 2.8.3. Memory Model and Synchronization . . . 37
  - 2.8.3.1. Memory Model . . . 37
  - 2.8.3.2. Synchronisation . . . 39
- 2.8.4. Communication, Distributed Support, and GPU Support in HPC runtimes . . . 40
- 2.8.5. Opportunities . . . 42
  - 2.8.5.1. **Opportunity - 1 : Standardization of Task based Runtime Systems** . . . 42
  - 2.8.5.2. **Opportunity - 2 : Addressing Heterogeneity** . 42

2.9. Dynamic Adaptation in Runtimes . . . 43

2.9.1. Scheduling and Load Balancing 43

2.9.1.1. Scheduling Using OS Threads 44

2.9.1.2. Worker Thread Pool 45

2.9.1.3. GPU Scheduling 46

2.9.1.4. Heterogeneous Scheduling 47

2.9.1.5. **Opportunity - 3 : Task Placement in Heterogeneous Systems** 47

2.9.2. Energy Aware Features and Studies 47

2.9.2.1. Dynamic Voltage and Frequency Scaling (DVFS) . . 48

2.9.2.2. Power Capping 49

2.9.2.3. Energy-aware Features in Runtimes 49

2.9.2.4. **Opportunity - 4 : Energy-aware Decision Making Capability** 50

2.9.3. Dynamic Adaptation Tools and Interfaces 50

2.9.3.1. Interfaces for Runtimes 50

2.9.3.2. Dynamic Adaptation Tools for Runtimes 52

2.9.3.3. **Opportunity - 5 : Dynamic Adaptation Tools and Interfaces** 53

2.10. Summary 53

III.DYNAMIC ADAPTATION IN HPX RUNTIME 55

3.1. Introduction: Two Problems 55

3.2. Background 58

3.2.1. HPX 58

3.2.2. PHYLANX 59

3.2.2.1. Alternating Least Squares (ALS) Benchmark 60

3.2.3. APEX 61

3.2.4. Static Baseline Policies 62

3.2.4.1. Baseline Policy for Parcel Coalescing 62

3.2.4.2. Baseline Policy for Task inlining 63

3.2.4.3. Limitation of Static Policy 64

3.3. Dynamic Policy for Parcel Coalescing: A Proof of Concept 64

3.3.1. Motivation Behind Adaptive Policy 64

3.3.2. What to Tune and Which Metrics Indicate Better Performance 64

3.3.3. Granularity of the Adaptive Policy 65

3.3.4. How the Adaptive Policy Works 65

3.3.5. Results 65

3.3.6. Discussion 67

3.4. Dynamic Policy for Task Inlining 67

3.4.1. What to Tune and Which Metrics Indicate Better Performance 67

3.4.2. Granularity of the Adaptive Policy 67

3.4.3. How the Adaptive Policy works 68

3.5. Experimental Results for the Task Inlining Policy 70

3.5.1. Experimental Testbed 70

3.5.2. AMD processors 71

3.5.3. Intel Xeon processors 73

3.5.4. Overhead of the APEX policy 74

3.6. Related Work for Task Inlining 74

3.7. Summary 76

IV. MEPHESTO: MODELING ENERGY-PERFORMANCE IN HETEROGENEOUS SOCS AND THEIR TRADE-OFFS 77

4.1. Introduction 77

4.2. Understanding the Effects of Collocated Execution on iSMHS 80

- 4.2.1. Contention vs. Energy and Performance 81
- 4.2.2. Need for Characterizing Kernels and PUs 83
  - 4.2.2.1. Justification for Using Operational Intensity 84
- 4.2.3. Ordering and Placement 84
- 4.2.4. Kernel Collocation for Varying Energy and Performance . . . 85

4.3. An Empirical Model for Memory Contention 86

- 4.3.1. Definitions 87
- 4.3.2. Characterization of Memory Contention 88
- 4.3.3. Collocation Estimator Algorithm 91

4.4. Defining Optimal Ordering and Placement 93

4.5. Kernel Collocation Strategy 94

- 4.5.1. Dynamic Programming Formulation 94
- 4.5.2. DP-Based Kernel Collocation Algorithm 95
- 4.5.3. Complexity 97

4.6. Experimental Setup 97

4.7. Experimental Results 99

- 4.7.1. Kernel Collocation Using Scientific Kernels 100
- 4.7.2. Accuracy of the Empirical Model 101
- 4.7.3. Experiments with Different EPTO Goals 103
- 4.7.4. Comparison with Greedy Algorithms 105
- 4.7.5. Three-PU Scenario: CPU, GPU, and PVA 106
- 4.7.6. Overhead Analysis of DP-Based Search 108

4.8. Related Work 109

- 4.8.1. Memory Contention Studies 109
- 4.8.2. Kernel Collocation in CPU-GPU Systems 110

4.8.3. Energy-Aware Algorithm Studies 111

4.9. Summary 111

V. MAPREDICT: STATIC ANALYSIS DRIVEN MEMORY ACCESS PREDICTION FRAMEWORK FOR MODERN CPUS 113

5.1. Introduction and Motivation 113

5.2. Understanding Memory Reads and Writes in Intel Processors 117

5.2.1. Description of the Hardware 118

5.2.2. A Tool for Measuring the LLC-DRAM Traffic 120

5.2.3. Different Read and Write strategies 121

5.2.3.1. Impact of Cache Line Size 121

5.2.3.2. Impact of Page Size 122

5.2.3.3. Impact of Initialization 122

5.2.3.4. Impact of Hardware Prefetchers 122

5.2.3.5. Impact of Compiler 124

5.2.3.6. Impact of Multi-threaded Parallel Execution 124

5.3. Modeling Different Types of Access 125

5.3.1. Sequential Streaming Access Pattern 125

5.3.1.1. Read Traffic 126

5.3.1.2. Write Traffic 126

5.3.2. Strided Access Pattern 127

5.3.2.1. Streaming Region 127

5.3.2.2. No Prefetching Region 127

5.3.2.3. Prefetching Zone 128

5.3.3. Stencil Access Pattern 129

5.3.3.1. One-dimensional Stencil 129

5.3.3.2. Two-dimensional Stencil 129
5.3.3.3. Three-dimensional Stencil 129
5.3.4. Random Access Pattern and Empirical Factor 130
5.3.4.1. Data Structure Randomness 130
5.3.4.2. Algorithmic Randomness 130
5.3.4.3. Machine Randomness and Empirical Factor 131
5.4. MAPredict Framework 131
5.4.1. Aspen, OpenARC, and COMPASS 132
5.4.2. Description of MAPredict Framework 132
5.4.2.1. Source Code Preparation Phase 132
5.4.2.2. Compile-time Static Analysis phase 133
5.4.2.3. Machine Model Generation Phase 134
5.4.2.4. Prediction Generation Phase 135
5.4.3. Identifying Randomness and $Empirical_{factor}$ 135
5.5. Experimental Setup 136
5.5.1. Accuracy Calculation 138
5.5.2. Comparison with Literature 138
5.6. Experimental Results 139
5.6.1. Regular Access Patterns 139
5.6.1.1. Sequential Streaming Access Pattern 139
5.6.1.2. Stencil Memory Access Pattern 140
5.6.1.3. Strided Memory Access Pattern 140
5.6.1.4. Single vs. Multi-threaded Execution 141
5.6.1.5. Comparison of Different Input Sizes 142
5.6.1.6. Comparison of GCC vs. Intel Compiler 142

5.6.2. Irregular Access and Large Application with Mixed Patterns . 143

5.6.2.1. Algorithmic Randomness 143

5.6.2.2. A Large Application with Mixed Patterns: Lulesh 145

5.6.2.3. Function Categorization of Lulesh 145

5.6.2.4. $Empirical_{factor}$ 146

5.6.2.5. Traffic: Number of LLC-DRAM Transactions 146

5.6.2.6. Scaling and Accuracy for Lulesh 146

5.6.3. Discussion 147

5.6.3.1. Overhead of MAPredict 147

5.6.3.2. Usability of $Empirical_{factor}$ 147

5.7. Related Works 147

5.7.1. Memory Access Prediction 148

5.7.2. Understanding Intel processors 148

5.7.3. Novelty in MAPredict 150

5.8. Summary 150

VI. UNDERSTANDING AND MODELING LLC-MEMORY TRAFFIC IN GPUS 152

6.1. Introduction and Motivation 152

6.2. Methodologies for GPU Access Investigation 154

6.2.1. NVIDIA and AMD GPUs 154

6.2.2. Strided Vector Multiplication for CUDA and ROCm 155

6.2.3. Measuring LLC-memory Traffic in GPUs 159

6.2.3.1. Using *nvprof* and *ncu* for NVIDIA GPUs 159

6.2.3.2. Using *rocprof* for AMD GPUs 159

6.2.3.3. Scripts for Data Collection 160

6.3. Understanding NVIDIA and AMD GPUs 160
    6.3.1. LLC-memory Traffic of NVIDIA GPUs 160
        6.3.1.1. Similarities between P100 GPU and Skylake CPU 160
        6.3.1.2. Similarities between P100, V100, and A100 161
        6.3.1.3. A Tailored Graph to Realize the Similarities 162
    6.3.2. LLC-memory Traffic of AMD GPUs 164
        6.3.2.1. Similarities between MI50 GPU and Skylake CPU 164
        6.3.2.2. Similarities between MI50, MI60, and MI100 166
    6.3.3. Comparison of the Profiling Tools 167
    6.3.4. Discussion about the Hypotheses 167
6.4. Experiment and Prediction: A Proof of Concept 167
    6.4.1. Prediction Model for NVIDIA GPUs 169
    6.4.2. Prediction Accuracy for NVIDIA GPUs 169
    6.4.3. Prediction Model for AMD GPUs 170
    6.4.4. Prediction accuracy for AMD GPUs 171
    6.4.5. Discussion 171
6.5. Related Work 172
6.6. Summary 173
VII CONCLUSION AND FUTURE WORK 174
7.1. Future Work 176
    7.1.1. Extending MAPredict 176
    7.1.2. Execution Time and Energy Consumption Prediction 178
    7.1.3. Static Characterization of Workloads 178
    7.1.4. Dynamic Adaptation in HPC Runtime 178

Chapter Page

7.1.5. Exploring New Architectures 179

REFERENCES CITED 181

LIST OF FIGURES

Figure Page

1. Challenges and research questions addressed in different chapters. . . . 5

2. A unified diagram of dynamic adaption in an HPC runtime systems by using performance models. 11

3. Layers of a runtime system. 15

4. Evolution of HPC runtimes for the last 35 years. 17

5. Different state-of-the-art HPC runtime systems. 23

6. The architecture of HPX and Phylanx along with APEX. 57

7. Partial code for the while loop in the Alternating Least Squares algorithm. 60

8. Visualization of the while loop in the Alternating Least Squares algorithm. 61

9. Interaction of APEX with the HPX runtime. 61

10. Dynamic parcel coalescing policy determines the value of the parameters at runtime to reduce network overhead. 66

11. Comparison of the Adaptive APEX and Baseline policies on an AMD Opteron node. 71

12. Comparison of the Adaptive APEX and Baseline policies on an AMD Opteron node for larger data sizes. 72

13. Comparison of the Adaptive APEX and Baseline policies on Intel Xeon node. 74

14. A logical representation of iSMHS with CPU and GPU. 81

15. Energy and performance behavior under different scenarios in an iSMHS. 82

16. Different total energy and performance behaviors for various collocation combinations for the same kernels. 86

Figure Page

17. Collocation factors: collocated/standalone values of time, power, and energy for different operational intensities. 89

18. Result for benchmark kernels for EPTO (0, 0). 99

19. Model accuracy of MEPHESTO. 101

20. Demonstration of the need for EPTO. 102

21. Comparison with greedy scheduling. Blue dots are EPTO points, and the black circle is the greedy solution. 105

22. Experiments with three processors: CPU, GPU, and PVA. 107

23. Overhead analysis of MEPHESTO. 109

24. LLC-DRAM traffic for different read and write scenarios in Intel processors. 119

25. Impact of Intel compiler on sequential streaming and strided access patterns. 124

26. Comparing single vs. multi-threaded runs. 125

27. Workflow of MAPredict framework. 133

28. Accuracy comparison of different regular access patterns. 137

29. Accuracy of MAPredict for various input sizes. 141

30. Accuracy comparison of GCC and Intel compiler. 142

31. Accuracy of algorithmic randomness for XSBench. 145

32. A simplistic representation of execution on a CPU and GPU for strided access. 157

33. Read and write traffic for the P100 GPU. 161

34. Read and write traffic for all NVIDIA GPUs. 162

35. Tailored comparison between NVIDIA GPUs and Intel CPU. 163

36. Read and write traffic for the MI50 GPU compared with the Intel Skylake CPU. 164

37. Read and write traffic for all AMD GPUs. 166

38. Prediction accuracy for NVIDIA GPUs for different input sizes. 168

Figure Page

39. Prediction accuracy for AMD GPUs for different input sizes. 169

40. Future work opportunities. 177

LIST OF TABLES

Table Page

1. Programming model and API. 32

2. Similarity between CUDA, HIP and OpenCL. 33

3. Execution models in different HPC runtimes. 34

4. Memory models and synchronization techniques in different HPC runtimes. 38

5. Communication, distributed support, and GPU support. 41

6. Scheduling and load balancing strategies in HPC runtimes. 43

7. Tools and interface for dynamic adaptation for HPC runtimes. 51

8. Opportunities addressed in this dissertation. 53

9. Nodes used to evaluate dynamic adaptation in HPX runtime system. . . 71

10. Roofline kernels used to understand the outcomes of different O&P variations. 81

11. The notations used by the model of MAPredict. 87

12. Benchmark kernels used for evaluating MEPHESTO's accuracy. 98

13. OpenCV kernels for PVA. 106

14. Machines and microarchitectures used for evaluating MAPredict. 118

15. Benchmarks used for evaluating MAPredict. 139

16. Analysis of Lulesh using MAPredict. 143

17. Analysis of multi-threaded execution of Lulesh using MAPredict. 144

18. Comparison of MAPredict with other works. 149

19. NVIDIA GPUs studied in Chapter VI. 155

20. AMD GPUs studied in Chapter VI. 156

Table Page

21. LLC-memory traffic prediction for NVIDIA GPUs. 170

22. LLC-memory traffic prediction for AMD GPUs. 171

LIST OF SOURCE CODE LISTINGS

Listing Page

5.1. Application model - Matrix Multiply (partial view). 134

5.2. Machine model (partial view). 134

5.3. MAPredict's memory access analysis. 136

6.1. Strided vector multiplication in CUDA. 158

CHAPTER I

INTRODUCTION

Since the beginning of parallel computing, the capabilities of runtime systems have been shaped by innovations in computer architectures. From the message-passing architecture to the latest heterogeneous architectures that include accelerators, runtime systems have been adapted to provide higher levels of abstractions and ensure efficient utilization of High Performance Computing (HPC) systems. Over the years, different runtime systems have emerged to address the needs of the HPC community. Early HPC runtimes started as bulk-synchronous Message Passing Interfaces (MPI [1]), where heavyweight processes communicate with each other through messages. However, the recent asynchronous multitasking runtime systems decompose the computation into fine-grained work units. Along with addressing both shared and distributed memory architectures, modern HPC runtime systems employ different execution models for running work units on different processors in heterogeneous systems. Runtime systems designed for task-based execution (e.g., OpenMP [2], HPX [3], and Charm++ [4]) operate by decomposing the total workload into sub-workloads. Runtime systems designed for heterogeneous platforms (e.g., StarPU [5]) maintain processor-wise queues to increase overall system utilization to meet specific user demands. There are also runtime systems designed specifically for accelerators such as GPUs (e.g., CUDA runtime by NVIDIA [6] and HSA runtime from ROCm software platform [7] by AMD). Runtime systems for accelerators explicitly leverage the throughput-oriented execution available on GPUs.

Runtime systems sit between the application and hardware and provide abstractions to applications for efficient execution on the underlying hardware. While developing an application, a programmer expresses the computation by following a programming model. Then, the compiler inserts the runtime system's functionalities when the executable is built. So, a runtime system implements the execution model. All the high-level programming languages have corresponding runtime systems that provide the necessary abstractions. For example, a C compiler inserts the stack management functionalities in the executable. However, HPC runtimes have significantly more responsibilities. A modern HPC runtime can make dynamic decisions for scheduling, load balancing, inter-node communication, and distributed memory management. These dynamic decisions can improve performance and reduce energy consumption. The main focus of this dissertation is to explore the opportunities for dynamic decision-making in HPC runtimes. However, implementing dynamic capabilities requires addressing several challenges that we now outline.

Challenge A - dynamic adaptation in HPC runtimes: The first and foremost challenge is to identify dynamic adaptation opportunities in modern HPC runtime systems. HPC runtime systems are active entities during the execution and can be interacted with through tools and interfaces. Some runtime systems expose interfaces to collect data; however, only a few runtime systems provide tools or interfaces for dynamically tuning their parameters that influence dynamic decisions. These tools and interfaces can provide critical insight about the applications and the machines to the runtime system. This insight facilitates the runtime system in making intelligent decisions. However, identifying the correct type of information from applications and machines is critical to performance improvement. Further, techniques that allow the gathering of this information need to be explored.

Challenge B - understanding the impact of memory architecture: Memory transactions have a significant impact on performance. Thus, understanding the implications of memory architecture can open the door for intelligent, dynamic decision-making. However, understanding the handshake between the complex memory hierarchy in modern HPC hardware and applications is challenging. A typical computing node of modern HPC systems houses a shared-memory multicore processor along with an accelerator. In some cases, the multicore processors share the system memory with the accelerator. Multiple processors executing memory-intensive kernels can create memory contention, negatively impacting performance and energy consumption.

Moreover, modern CPUs and GPUs employ a complex cache-hierarchy. Depending on the application's memory access pattern, cache-hierarchy could play an essential role in determining the overall performance. Runtime systems can transfer data between nodes and optionally provide mechanisms to transfer data between CPUs and GPUs. These data transfers are done from one processor's dynamic random access memory (DRAM) to another, thereby bringing data close to the target processors. However, a runtime system does not have visibility beyond inter-node and host-accelerator data transfers. Therefore, providing information about the interplay between application and memory hierarchy can enable a runtime system to perform intelligent, dynamic decisions.

Challenge C - modeling and predicting the impact of memory architecture: Modern HPC performance measurement and analysis tools such as TAU [8] can generate a plethora of information about an application's interaction with the cache hierarchy. Insight gathered from that information can be used to to improve performance further. However, these performance tools collect this information as the application is executing. Runtime systems would need such information *before* executing a kernel to make optimal decisions (such as kernel placement within a heterogeneous system). Static analysis techniques provide compile-time information collection to tackle this problem. However, existing static analysis tools provide instruction counts (such as load and store counts) instead of considering the impact of the cache hierarchy. Therefore, analytical or empirical models that capture the essence of the interaction between the application and the memory hierarchy need to be explored to generate a static prediction. This prediction can, in turn, provide crucial detail to a runtime system.

1.1 Working Toward Solutions - This Dissertation

This dissertation strives to investigate these challenges and answer the following research question: **How can information gathered from applications and machines at compile time empower modern HPC runtime systems for intelligent and dynamic decisions?** This broad research question is further decomposed into smaller, narrower questions, each corresponding to one chapter of this dissertation. Figure 1 depicts the specific questions addressed in each chapter. Figure 1 also depicts the "flow" of this dissertation by noting the connection between successive chapters. Brief descriptions of each chapter and connections are presented in the following sections.

1.2 Chapter II — Dynamic Adaptation Techniques and Opportunities in HPC Runtimes

Chapter II addresses **RQ1**: What are the dynamic adaptation opportunities in modern HPC runtimes? This study explores HPC runtimes for the last 35 years to survey the dynamic adaptation techniques and opportunities in contemporary runtime systems. First, the evolution of the runtime systems is studied to identify the state-of-the-art runtime systems. Runtime systems are then categorized, and the main features are discussed. Existing dynamic adaptation capabilities are then summarized, followed by an exploration of the opportunities. The material in this

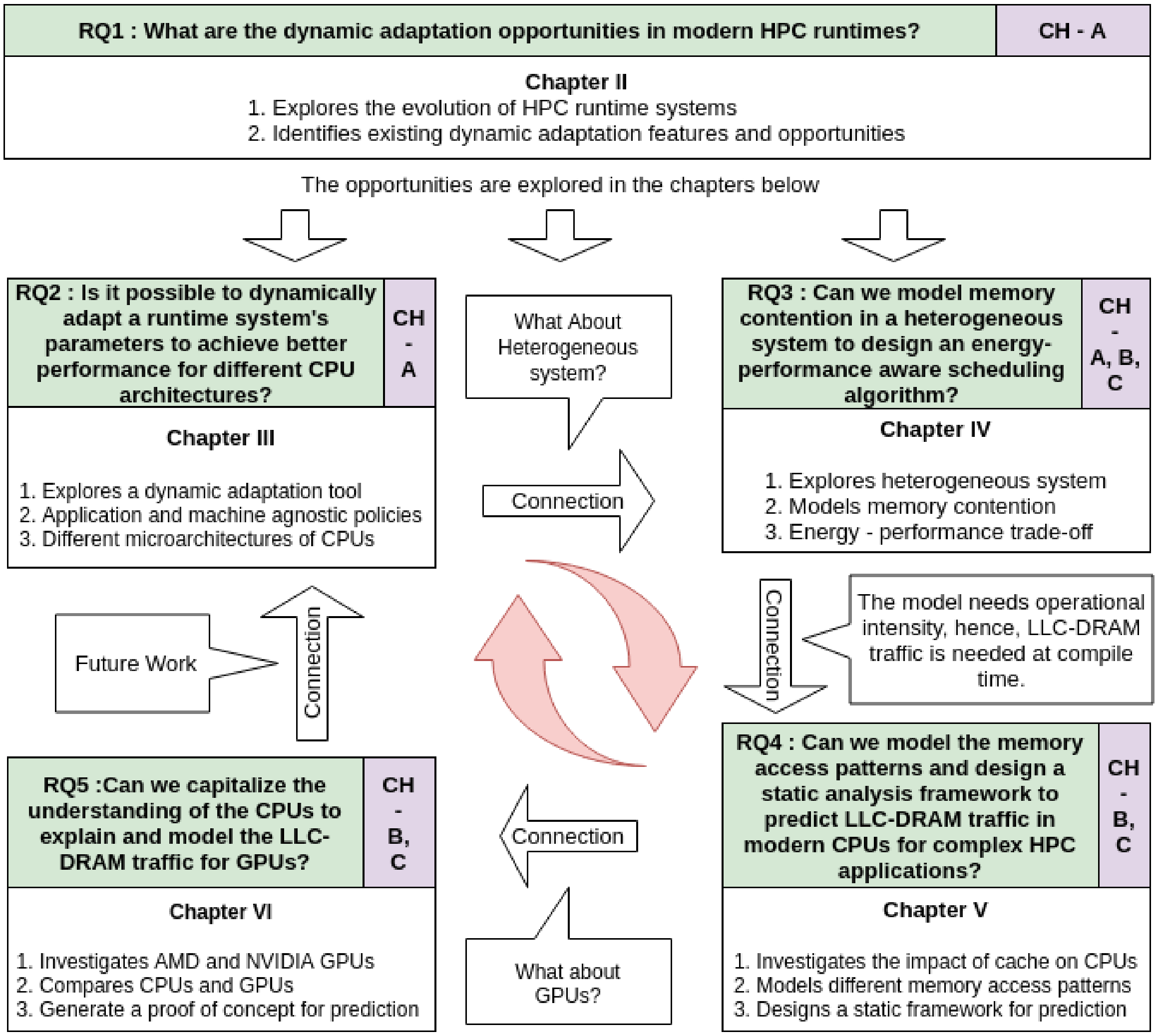

RQ1 : What are the dynamic adaptation opportunities in modern HPC runtimes?
CH - A
Chapter II
1. Explores the evolution of HPC runtime systems
2. Identifies existing dynamic adaptation features and opportunities
The opportunities are explored in the chapters below
RQ2 : Is it possible to dynamically adapt a runtime system's parameters to achieve better performance for different CPU architectures?
CH - A
Chapter III
1. Explores a dynamic adaptation tool
2. Application and machine agnostic policies
3. Different microarchitectures of CPUs
What About Heterogeneous system?
Connection
RQ3 : Can we model memory contention in a heterogeneous system to design an energy-performance aware scheduling algorithm?
CH - A, B, C
Chapter IV
1. Explores heterogeneous system
2. Models memory contention
3. Energy - performance trade-off
Future Work
Connection
Connection
The model needs operational intensity, hence, LLC-DRAM traffic is needed at compile time.
RQ4 : Can we model the memory access patterns and design a static analysis framework to predict LLC-DRAM traffic in modern CPUs for complex HPC applications?
CH - B, C
Chapter V
1. Investigates the impact of cache on CPUs
2. Models different memory access patterns
3. Designs a static framework for prediction
RQ5 :Can we capitalize the understanding of the CPUs to explain and model the LLC-DRAM traffic for GPUs?
CH - B, C
Chapter VI
1. Investigates AMD and NVIDIA GPUs
2. Compares CPUs and GPUs
3. Generate a proof of concept for prediction
Connection
What about GPUs?

Figure 1. Challenges and research questions addressed in different chapters. Research questions are presented in green boxes. The following challenges are presented in violet boxes.
Challenge A - *Dynamic adaptation in HPC runtimes,*
Challenge B - *Understanding the impact of memory architecture,*
Challenge C - *Modeling and predicting the impact of memory architecture.*

chapter is unpublished with no co-authorship. However, revision suggestions were given by Seyong Lee and the dissertation committee (Allen Malony, Hank Childs, and Boyana Norris) during the area exam. This chapter is in preparation for submission to a journal as a survey.

Connection to RQ2 and Chapter III: Chapter II answers **RQ1** and finds out the dynamic adaptation opportunities for HPC runtimes. To realize one of these opportunities, **RQ2** is explored in Chapter III which involves dynamic adaptation in HPX runtime system.

1.3 Chapter III — Dynamic Adaptation in HPX Runtime

Chapter III addresses **RQ2**: Is it possible to dynamically adapt a runtime system's parameters to achieve better performance for different CPU architectures? This study investigates dynamic adaptation opportunities in HPX, an asynchronous task-based runtime system. HPX works by dividing a problem into a large number of fine-grained tasks. However, as the number of tasks created increases, the overheads associated with task creation, communication, and management also increase. Chapter III generates application and machine agnostic dynamic adaptation solutions for two problems: (1) task inlining, and (2) parcel coalescing. Task inlining, a method where the parent thread consumes a child, enables the runtime system to balance the parallelism and its overhead. On the other hand, parcel coalescing is related to reducing communication overhead. Using the APEX framework [9], we design an adaptive policy for deciding, at runtime, whether a particular task should be inlined or not and when to coalesce parcels. We show that a baseline policy that makes these decisions using a fixed threshold is outperformed by adaptive policies that dynamically decide the threshold at runtime. We also evaluate and justify the performance of these policies on different processor architectures from Intel and AMD.

Content of this chapter is published at ICPP 2019 [10] and at ICPP 2018 [11] (poster). These publications are co-authored by Dr. Kevin Huck and Dr. Allen Malony from University of Oregon and Dr. Bibek Wagle, Adrian Serio, and Dr. Hartmut Kaiser from Louisiana State University.

Connection to RQ3 and Chapter IV: This study answers **RQ2** and shows the performance improvement by dynamically adapting the runtime system's parameter. However, only multicore CPUs are considered for executing the tasks. Since heterogeneous systems are now commonly found in HPC facilities, it is desirable to investigate dynamic adaptation in a heterogeneous platform. For this reason, **RQ3** is investigated in Chapter IV, which involves placing kernels in a heterogeneous system and addressing the challenges associated with such placements.

1.4 Chapter IV — Studying Memory Contention in a Heterogeneous System

Chapter IV addresses **RQ3**: Can we model memory contention in a heterogeneous system to design an energy-performance aware scheduling algorithm? This study investigates integrated shared memory heterogeneous architectures with specialized processing units (PUs) that share a unified system memory to improve performance and energy efficiency by reducing data movement. However, they introduce memory contention when multiple PUs access the memory. Chapter IV introduces MEPHESTO, a novel and holistic approach for achieving energy-performance trade-off. We characterize applications and PUs in terms of two memory contention factors - time factors and power factors - to achieve the desired trade-off between energy and performance for collocated kernel execution on heterogeneous systems. This investigation combines these factors and presents a simple knob-based approach that expresses the target trade-off. The approach is evaluated on a diverse integrated shared memory heterogeneous system with a CPU, GPU, and programmable vision accelerator. Using an empirical model for memory contention that provides up to 92% accuracy, the kernel collocation approach can provide a near-optimal ordering and placement based on the user-defined, energy-performance trade-off parameter. Moreover, the dynamic programming-based heuristics provide up to 30% better energy or 20% performance benefits when compared with the greedy approaches commonly employed by previous studies. Content of this chapter is published at PACT 2020 [12]. This publication is co-authored by Dr. Allen Malony from University of Oregon, Mehmet Belviranly from Colorado School of Mines, and Dr. Seyong LEE and Dr. Jeffrey Vetter from Oak Ridge National Laboratory.

Connection to RQ4 and Chapter V: This study answers **RQ3**. MEPHESTO shows that intelligent scheduling algorithms can make energy-performance trade-off-aware decisions if a runtime system is equipped with a memory contention model. However, MEPHESTO depends on the operational intensity of the kernels (Roofline model), which is the ratio of FLOPS and LLC-DRAM traffic. Moreover, a MEPHESTO needs this information during scheduling (i.e., before executing the kernels). For this reason, determining operational intensity at compile time is desired. While the number of FLOPS can be deduced statically using the COMPASS framework [13], statically deducing the number of LLC-DRAM traffic for modern HPC hardware is still an open problem. To develop a performance model to predict LLC-DRAM traffic statically, we need to understand the handshake between applications and modern processors' caches. For this reason, modern processors need to be studied for different memory access patterns, which relates to **RQ4**, investigated in Chapter V.

1.5 Chapter V — Static Analysis Framework for Memory Access Prediction in Modern CPUs

Chapter V addresses **RQ4**: Can we model the memory access patterns and design a static analysis framework to predict LLC-DRAM traffic in modern CPUs for complex HPC applications? This study explores the handshake between HPC applications and the memory hierarchy of modern Intel CPUs to understand and model the behavior for static prediction. Recent advancements in memory hierarchy make it difficult to predict the traffic between LLC and DRAM statically. Prefetching algorithms, compiler choice, and parallel execution on multicore processors make predictions more difficult. Continuous innovation and the release of new manufacturer-specific microarchitectures add to this complexity. Different memory access patterns in HPC applications introduce another layer of challenges for statically predicting the LLC-DRAM. HPC applications can have four types of memory access patterns, 1) sequential streaming, 2) strided, 3) stencil and 4) random memory access patterns. Moreover, multiple access patterns can co-exist in the same kernel, which makes the prediction more difficult.

Chapter V introduces MAPredict, a static analysis-driven framework that addresses these challenges to provide memory access prediction by gathering application and machine properties at compile time. MAPredict depends on the OpenARC compiler [14] for static analysis of the code and the COMPASS [13] framework for expressing an application in the Aspen [15] domain-specific modeling language. MAPredict defines the limit of static analysis for memory access prediction and provides a means of overcoming those limits by capturing dynamic information. By exploring and analyzing the behavior of modern Intel processors, MAPredict formulates compiler and microarchitecture-aware analytical models. MAPredict then invokes the analytical model to predict LLC-DRAM traffic by combining the application model, the machine model, and user-provided hints for capturing dynamic information. Evaluation using various workloads with different memory access patterns, input sizes, and code-base sizes on four recent Intel microarchitectures demonstrates that MAPredict can predict the memory traffic with high accuracy in modern CPUs. Content of this chapter is published at MCHPC 2020 workshop at SC 2020 [16] and is under review in HPCA 2022. These publication efforts are co-authored by Dr. Allen Malony from University of Oregon, and Dr. Seyong LEE and Dr. Jeffrey Vetter from Oak Ridge National Laboratory.

Connection to RQ5 and Chapter VI: This study answers **RQ4** and shows MAPredict can predict LLC-DRAM for complex HPC applications by combining static and dynamic information. MAPredict investigated different microarchitectures of CPUs. However, it is desirable to enable MAPredict to provide a similar prediction for GPU kernels; this relates to **RQ5** which is investigated in Chapter VI.

1.6 Chapter VI — Understanding and Modeling the Impact of Memory Access Patterns in GPUs

Chapter VI addresses **RQ5**: Can we capitalize the understanding of the CPUs to explain and model the LLC-DRAM traffic for GPUs? This study explores GPUs to understand and model the behavior of cache hierarchy by comparing them with CPUs. Both the programming model and the execution model are significantly different in GPUs when compared to CPUs. Even though the directive-based GPU programming approach makes the code similar to CPUs, the compiler transforms the source code and enables the GPU execution model. Moreover, the memory hierarchy is also different and comparatively less studied than CPUs. While CPUs have three levels of cache (usually), GPUs introduce new concepts, such as user manageable shared-memory that resides on the L1 cache and the abstract concept of thread-local memory that resides on DRAM (global memory). These challenges and new concepts in the memory hierarchy make it difficult to develop models for predicting LLC-DRAM traffic in GPUs. Chapter VI strives to investigate the regular access patterns (sequential streaming and strided patterns). Results from different NVIDIA GPUs (A100, V100, and P100) and AMD GPUs (MI50, MI60, MI100) show striking similarities between the traffic generated by CPUs and GPUs. Content of this chapter is accepted in RSDHA 2021 workshop at SC 2021 workshop. This publication effort is co-authored by Dr. Allen Malony from University of Oregon, and Dr. Seyong LEE, and Dr. Jeffrey Vetter from Oak Ridge National Laboratory.

With Chapter VI, this dissertation concludes it's exploration to answer the main question (presented in in Section 1.1) to dynamically adapt HPC runtimes.

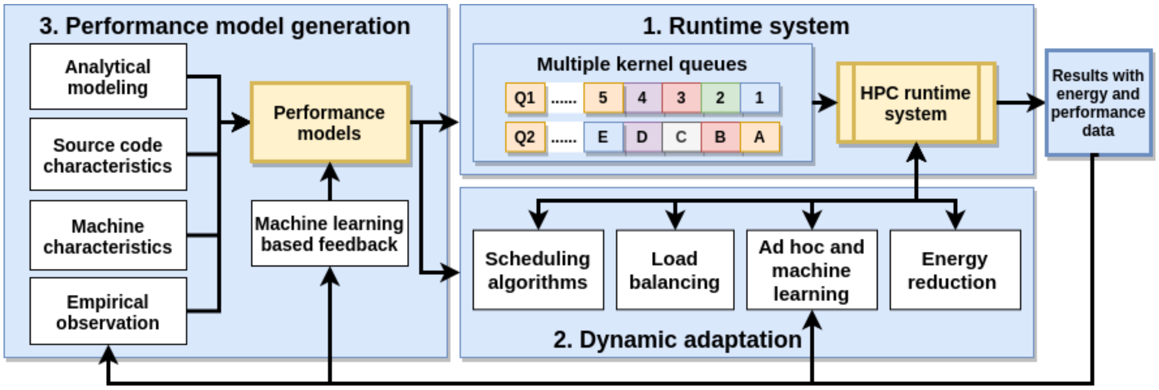


Figure 2. A unified diagram of dynamic adaption in an HPC runtime systems by using performance models.

1.7 A Unified Diagram

Figure 2, which was inspired by the problems studied in this dissertation, presents a unified diagram of dynamic adaptation in HPC runtimes. Box 1 represents the runtime systems with different kernel queues. Box 2 represents the dynamic adaptation engine that interacts with the runtime to facilitate scheduling, load balancing, and energy consumption reduction. Box 3 represents the offline performance model generation to feed the kernels of the runtime system and the dynamic adaption engine. Research questions and studies depicted in Figure 1 contribute to each of the boxes, thereby constituting the answer to the main research question presented in Section 1.1.

This dissertation includes prose, figures, and tables from previously published conferences, workshops, and journal proceedings.

CHAPTER II

DYNAMIC ADAPTATION TECHNIQUES AND OPPORTUNITIES TO IMPROVE HPC RUNTIMES

This chapter contains unpublished material with co-authorship. The survey presented in this chapter was developed as part of my departmental Area Exam, where I received the main guidance from my advisor Dr. Allen Malony. For the survey formation, Dr. Seyong Lee provided regular guidance. While working on my Area Exam, I received feedback and suggestions from my dissertation committee members (Dr. Hank Childs and Dr. Boyana Norris). Dr. Jeffery Vetter also provided high-level feedback to formulate the survey work. All the data collection and writing was done by me while the committee members helped to proofread the Area Exam document.

2.1 Introduction and Motivation

This chapter explores the state-of-the-art HPC runtime systems to find out the opportunities for dynamic adaptation (corresponding to Research Question 1 — **RQ1**: What are the dynamic adaptation opportunities in modern HPC runtimes?). As mentioned in Chapter I, the evolving ecosystem of HPC runtimes continues to provide abstractions at the cost of increasing responsibilities at different layers. With access to the application that is running and the underlying hardware, an HPC runtime positions itself as an active component that can make application- and hardware-aware decisions. By providing a way to interpret the relationship between application and hardware, runtime systems are capable of performing more dynamic decisions to improve performance and reduce energy consumption. For this reason, the features and dynamic decision capabilities of runtime systems need to be studied. Moreover, the evolution of the HPC runtimes needs to be understood to identify what drives the change in the HPC ecosystem. For this reason, this study is organized as follows.

2.1.1 Organization.

There are three components of this study: 1) runtime systems, 2) dynamic adaptation techniques and capabilities of the runtime systems, and 3) opportunities for dynamic adaption. Having these three goals, this study is conducted in multiple steps. First, the definition of different terms and runtime systems are provided. Next, in order to understand the driving force behind the evolution of HPC runtimes, major events of the last 35 years are studied. This evolution study also helps to identify contemporary state-of-the-art HPC runtimes. Then, HPC runtimes are categorized. Each runtime system is then briefly studied to understand the concepts. The main features of the runtime systems are then compared to identify similarities and dissimilarities. Dynamic adaptation capabilities and features of the runtime systems are then studied. Finally, based on the trend of the runtime systems and observation acquired over the past 35 years, opportunities that would increase the dynamic adaption capabilities of the runtime systems are identified.

2.2 Definition of Runtime Systems

In order to properly define a runtime system, programming models, execution models, APIs, libraries, language extensions, and languages are discussed first.

2.2.1 Programming Models and Execution models.

Both the programming model and execution model are logical concepts. The programming model is related to how a programmer designs the source code to perform a particular task to take advantage of the underlying runtime system and the hardware[17]. On the other hand, the execution model decides how a program written by following a particular programming model is executed in real time[18]. In short, the programming model is the style that is followed to program, and the execution model is how the program executes. For example, the programming model of OpenMP defines how to express parallelism using directives and parallel regions, and the execution model refers to the multi-threaded execution of the code.

2.2.2 Parallel Programming APIs, Libraries, Language Extensions, and Languages. Parallel programming APIs are entities designed to express the programming and execution models in code. An API can be a new language, a language extension, or a collection of libraries. For example, the OpenMP [19] programming API is a language extension while the API for HPX [20] is based on libraries. Every programming language provides an execution model to be used at runtime.

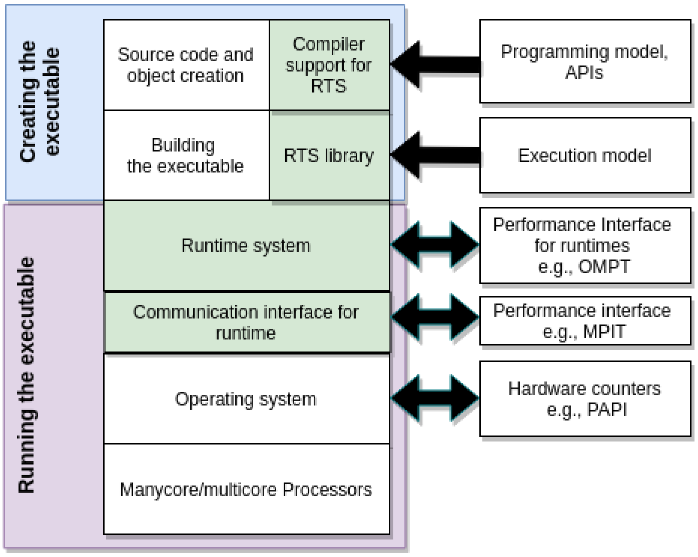

Creating the executable
Source code and object creation
Compiler support for RTS
Programming model, APIs
Building the executable
RTS library
Execution model
Running the executable
Runtime system
Performance Interface for runtimes e.g., OMPT
Communication interface for runtime
Performance interface e.g., MPIT
Operating system
Hardware counters e.g., PAPI
Manycore/multicore Processors

Figure 3. Layers of a runtime system.

2.2.3 HPC Runtimes. A runtime system that defines the execution environment is an interface between the Operating System (OS) and the application. It abstracts the complexity of an OS and ensures portability to different OSs [21]. The execution model of a program is written by following a programming model through an API and is ensured by the runtime system. For example, the runtime system of the C language provides memory management. C compiler inserts instructions into the executable, and when it runs, the runtime system, which is a part of the executable, manages the stack to provide all the memory management functionalities. However, HPC runtimes go far beyond the capabilities provided by the basic languages. An HPC runtime system becomes an active entity that is capable of making dynamic decisions based on the execution model and in some cases, performs communication while the executable is running. Some programming languages in the HPC domain host powerful runtime systems. For example, Chapel [22] is a Partitioned Global Address Space (PGAS or GAS) language; however, its runtime system manages a global address space. For this reason, some HPC languages are also considered in this study.

Figure 3 shows a layer-wise diagram of a runtime system. At the top layer, the programming model is expressed through source code by calling the API. When the object file is created, the compiler inserts necessary codes to carry out necessary runtime activities. In the second layer, the execution model is expressed when the executable is built by resolving all the library calls (partially). When the program is executed, the runtime system at layer three sits on top of the communication layer (or the operating system) to reinforce the execution model by providing all the runtime facilities such as scheduling, load balancing, collecting data, etc. Since most HPC runtime systems are active components, interfaces, tools, or techniques are available for direct interaction with the runtime layers. Using these interfaces, the activities of runtime systems can be monitored, altered, or controlled. At level five, the operating system sits where hardware counter data can be collected. Some runtime systems collect the hardware counters from the operating system.

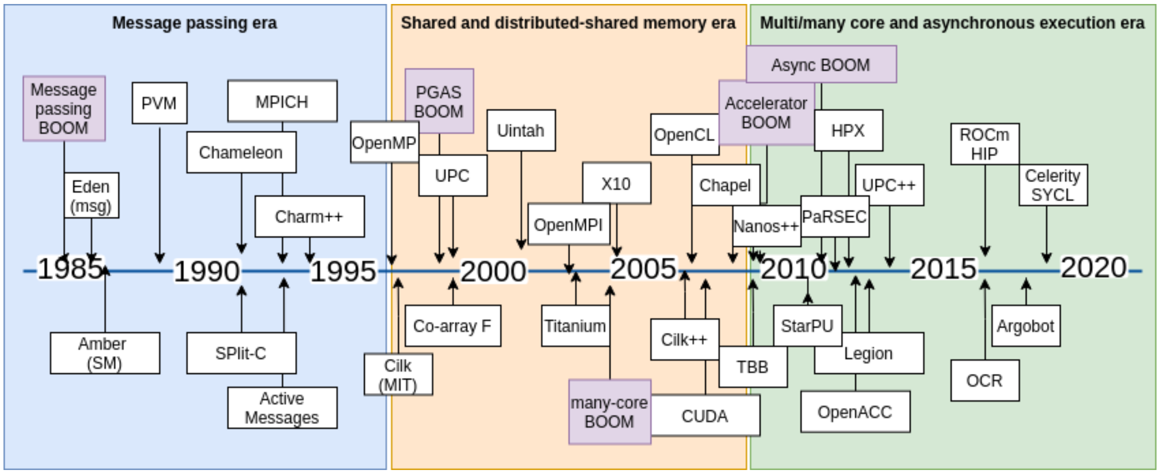


Figure 4. Evolution of HPC runtimes for the last 35 years.

2.3 Evolution of HPC Runtimes

In this section, the evolution of HPC runtimes is discussed. The major events that transpired and runtimes developed over the last 35 years are explored to understand what shaped contemporary HPC runtimes. The reason behind choosing 1985 as the starting year is because parallel computers started becoming more available around that time. It is no secret that innovations in the field of computer architecture majorly shape HPC runtimes. A closer look is taken at the correlation between the ever-changing computer architectures and the evolution in runtime systems. From 1985 to the present day, the whole period is divided into four parts considering the events in 10 years duration. After, the state-of-the-art runtimes are identified to take a closer look.

2.3.1 Before 1985. In the 1970s, for providing the highest performance, vector computing was the go-to solution [23] (such as single-processor machine Cray-1 [24] and multiprocessor machine Illiac [25]). In the 1980s, multiprocessor vector machines started becoming available in the form of Massively Parallel Processors (MPPs) (e.g., the concept of a hypercube [25])). Such an example is Caltech's Cosmic

Cube [26] (fully operational in 1983) which was built using 64 nodes in point-to-point connection without shared memory. With this configuration, message passing between nodes was necessary. Even though the idea of messaging passing architecture is successfully demonstrated by Cosmic Cube, the concept of message passing for distributed memory systems already existed in other efforts, such as the Eden project for object style programming for distributed systems [27]. After the success of Cosmic Cube, other manufacturers started building cube machines, and the trend lasted for a decade. At the same time, cluster computing, where compute nodes were loosely connected through network interface cards, was also gaining popularity in the 1980s for its simplicity in design [28]. Even though there were differences in architectures between hypercubes and clusters, during this decade, it was established that a message passing architecture could provide high scalability at a low cost.

2.3.2 HPC Runtimes in 1985-1995: Message Passing Architecture.

Having realized the potential of message passing architectures, vendors, national labs, and academia started providing programming interfaces that could facilitate message passing for distributed memory systems. Intel developed the NX/2 operating system [29] which provided messaging passing interface through system calls. Parasoft developed the Express [30] library that made a program portable to different machines (supported C and Fortran). Argonne National Laboratory (ANL) introduced p4 [31]. Similarly, IBM introduced the Venus [32] communication library for message passing. Moreover, PVM [33], a collaboration between Oak Ridge national laboratory (ORNL), University of Tennessee, and Emory University supported message passing for heterogeneous parallel computing. Chameleon [34] from ANL provided interoperability between different message passing libraries. These efforts focused on enabling message passing for parallel computing and took place in the late 1980s and early 1990s. However, because of different implementations by different manufacturers, portability became an important aspect. For this reason, in 1992, 60 people from 40 organizations started the Message Passing Interface (MPI) standardization effort at a workshop called "Workshop on Standards for Message Passing in a Distributed Memory Environment". By 1993, the standard was completed and presented at SC93. In hindsight, this probably was one of the biggest initiatives that shaped the modern high performance computing domain. After the standardization, MPICH [1] was released. MPICH (CH comes from Chameleon) is still one of the popular MPI implementations today. Even in 2021, MPI is the de-facto standard for HPC, and it will not be an overstatement to say "MPI is everywhere".

Apart from the rise of MPI, some other events took place which did not immediately become as successful as MPI but planted the seed for the future. One such effort is the C language extension Split-C [35], which enabled the idea of global address space by providing the option for declaring a distributed array. With this configuration, one processor could reference pointers to another processor through communication. The idea of global address space was not new at that time. Amber systems [36] mention a similar idea about global address spaces. However, Split-C is considered to be the precursor of UPC, a PGAS language of modern times (SHMEM is also credited for one-sided communication, which is one of the ideas of PGAS model [37]). Moreover, Charm++ [4], introduced in 1993, implemented a task programming model and is a precursor to the asynchronous many tasks (AMT) based runtimes. Another work, Active Messages [38], provided the idea of efficient message-driven computation (opposed to message passing), which is found in modern HPC runtimes. The idea of loose synchronization is mentioned in Midway [39], which provided an opportunity to synchronize caches in distributed memory systems. The runtime system carried out the synchronization through a barrier that is specified by the programmer. The main idea was to have loosely bound synchronization criteria where the programmer was in charge of deciding whether a synchronization was necessary or not.

2.3.3 HPC Runtimes in 1996-2005: Shared Memory and Distributed Shared Memory. MPI standardization effort provided a great example of how a community-led effort could streamline an HPC Programming model. Riding off the success of MPI in distributed memory, a standard for shared memory programming models was introduced in 1997. First for Fortran and then for C, the OpenMP architecture review board (OARB) [2] released its standard to provide an alternative to MPI for increasing parallelism in shared memory systems. The standardization provided portability for OpenMP and was widely accepted by industry and academia. With OpenMP and MPI standardized, the HPC community was given two means to program both shared memory and distributed memory systems. By the end of the 1990s, the HPC community realized the drawbacks of explicit synchronous message passing at the front end for data transfer. For this reason, the concept of distributed shared memory gained traction, and the concept of Partitioned Global Addressed Spaces (PGAS) was introduced. Three PGAS languages were launched in the late 1990s and early 2000s. UPC for C [40], Co-array Fortran for Fortran [41] and Titanium for Java [42]. These separate languages provided options to declare distributed arrays where the runtime systems of these languages performed one-sided communication through communication interfaces like the GASNet communication library [43]. The GASNet communication interface is capable of using MPI for its one-sided communication. Later two more prominent PGAS languages were introduced, X10 [44] and Chapel [45], which introduced an asynchronous task model for distributed execution while utilizing a global address space.

2.3.4 HPC Runtimes 2005-2015: Multicore, Manycore and Heterogeneity.

In the mid-2000s, processor manufacturers were hit with the temperature barrier, which limited them from increasing processor performance by increasing the frequency in a single core. This obstacle led to the start of the manycore boom. Even though manycore processors existed previously, after 2005, they became mainstream for all levels of computing. NVIDIA, a graphics processor manufacturer, realized this opportunity, who started manufacturing their graphics processor for general-purpose computing. This brought about the era GPGPUs. With the release of CUDA [46] by NVIDIA in 2007, accelerator programming drew the attention of the HPC community. As a result, runtime system providers started working towards supporting GPUs in their programming environments. Heterogeneous systems, where CPUs and GPUs are housed in a single node, led runtime developers to invent new approaches to harness the computing power from such systems. Programming standards and APIs for heterogeneous systems such as OpenCL [47], OpenACC [48] and StarPU [49] were introduced. These two major changes (multicore CPUs and manycore GPUs) increased in-node parallelism and exposed the disadvantages of using synchronization-based message passing in MPI. Since explicit message passing with bulk synchronous barriers make both sender and receiver wait, it restricts the utilization of processor cores to reach their peak utilization (later MPI started providing asynchronous communication). For this reason, fine-grain tasks (instead of heavy MPI ranks or OpenMP threads) became one the most active fields in runtime systems research. Since lightweight tasks can be easily yielded and resumed compared to heavy-weight OS threads, the asynchronous task based execution model received attention for increasing utilization of multicore processors. To provide high computation and communication overlap, many Asynchronous Many Task (AMT) runtime systems appeared, such as HPX [3], Cilk Plus [50], TBB [51], Legion [52], etc. Moreover, with C++11 released, highly templated code with improved asynchronous features began to be adopted by the runtimes. The AMT execution model is considered by the community to be a better fit for exascale computing, which is envisioned to appear by the early 2020s.

2.3.5 HPC Runtimes during 2015-Present: Asynchronous Many Task and Abstraction. After 2015, the HPC community started focusing more on abstraction since multiple computing paradigms (CPU and GPU) in one node became common. For this reason, initiatives for providing programming approaches in a portable way (such as Kokkos [53]) started appearing. Moreover, AMD released its open-source GPU programming capability ROCm platform (HIP) [7]. Since there were already a considerable number of AMT runtimes introduced, the HPC community also started realizing the need for an AMT interface. A group of runtime system researchers from industry, national labs, and academia launched Open Community Runtime (OCR) [54] by releasing its specification. Moreover, Argobots [55] was introduced to work with different asynchronous many task execution runtimes.

2.3.6 Reduction and Identification of State-of-the-art. At this point, it is clear that there are several domains into which HPC runtimes can be divided. Even though the MPI+X model is the commonly adopted model for scientific applications nowadays, many modern runtime systems use MPI at a communication level where MPI message sending and receiving is done by the runtimes rather than the programmer. Considering the recent developments, HPC runtimes can be categorized

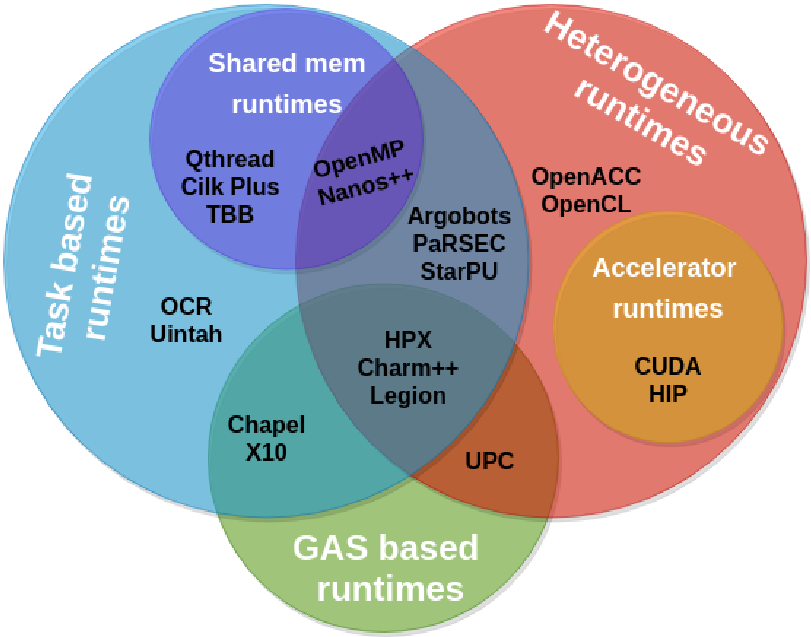


Figure 5. Different state-of-the-art HPC runtime systems. White Text = Category name and Black Text = Runtime name. Runtime systems' color coding, Blue = Many task runtime, Green = GAS based runtime, Red = Heterogeneous capability enabler runtimes (Accelerator runtime included in this category), and Purple = Shared memory runtime. Note: MPI is not here. Because it's the top view. MPI is now part of many of the runtime systems.

into four categories: 1. shared memory runtimes, 2. task based runtimes, 3. GAS based runtimes (languages), and 4. heterogeneous runtimes. Figure 5 shows the distribution of HPC runtimes. It is easy to notice that 20 runtime systems are chosen that represent the whole HPC runtime spectrum. HPC runtime survey papers [56, 57] are consulted to choose these runtimes. Some runtimes can provide multiple features, but the categorization is based on the commonly known feature of the runtime.

2.4 Shared Memory Runtime Systems

In this section, shared memory runtime systems are briefly discussed.

2.4.1 Cilk Plus. Cilk [58] originated from MIT in the mid-1990s, and later Intel acquired it when MIT licensed it to Cilk Arts [59]. Intel released Cilk

Plus [60, 50] as a part of the ICC compiler suite. Cilk plus uses a nonblocking spawn function to generate new tasks in a DAG that later syncs (spawn-sync), implementing the fork-join model. Cilk plus extended C/C++ by adding three keywords cilk_for, cilk_spawn, and cilk_sync. Cilk Plus provides a compiler-driven approach for task-level parallelism in shared memory machines.

2.4.2 TBB.

OS thread-based solutions for programming multicore systems are not portable. For this reason Intel TBB [61, 51] (oneTBB) is a C++ template library for threading abstraction. TBB is mainly designed for shared memory multicore CPUs. It expresses parallelism in terms of logical tasks (C++ objects), which are scheduled to a pool of OS threads. In other words, it provides a wrapper to use the OS threads to make the program portable.

2.4.3 OpenMP.

OpenMP [62] is one of the most popular and widely used names in the HPC community for its shared memory programming model. OpenMP Architecture Review Board manages it (OARB) [63]. This board has members from all leading manufacturers. OARB published the first specification [19] in 1997 for Fortran, and in the following year, a C/C++ standard was released. There is one master thread that forks multiple threads for data and task parallel computation. When a computation finishes, all the threads are joined to the master thread. For this reason, OpenMP is often referred to as a fork-join model.

2.4.4 OmpSs and Nanos++.

OmpSs [64, 65] is an effort from Barcelona Supercomputing Center (BSC), which made an appearance in the HPC world in 2011. The main idea of OmpSs is to extend OpenMP and StarSs [66] for a directive-based asynchronous task execution model that also supports accelerators such as GPUs, FPGAs, etc. along with CPUs. OmpSs is implemented as an extension to OpenMP that enables asynchronous task features that target newer architectures like GPU, FPGA, etc. Over the years, many features from OmpSs were included in OpenMP specification [65]. For this reason, OmpSs is considered a forerunner of accelerator-based OpenMP. OmpSs is built on top of Mercurium source to source compiler [67] and the Nanos++ runtime system [68].

2.4.5 Qthreads.

Qthreads [69, 70], an effort from Sandia lab introduced in 2008, is a user-level library for on-node multithreading. The initial target was to provide massive level multithreading with rich synchronisation [71]. With Qthreads, when an application exposes parallelism (specified by the user) in a massive number of lightweight user-level threads, the runtime system dynamically manages the scheduling of tasks.

2.5 Task Based Runtime Systems

Task based runtime systems are discussed in this section.

2.5.1 Charm++.

Charm++ [4] is one of the pioneers of modern asynchronous task based runtimes. It originated at the University of Illinois at Urbana Champaign (UIUC) in 1993. The Charm++ programming model and runtime implement a message-driven paradigm where computation starts after receiving messages. It works through parallel processes called *chares* which are C++ objects. These objects have entry points that are executed when a message is received. A program is over decomposed in terms of *chares* and the execution is completely non-deterministic since *chares* are invoked asynchronously [72].

2.5.2 HPX.

HPX [3] runtime is from Louisiana State University (LSU) and was introduced in 2014. HPX implements the concepts of the ParalleX execution model [73]. HPX strictly conforms to C++ standards and enables wait-free asynchronous execution. HPX implements active messages where computation is sent to data instead of sending data towards computation. In HPX, active messages are called *parcels* and processing elements are called *localities*. The runtime system implements an Active Global Address Space (AGAS) that is capable of object migration. AGAS generates the Global ID and GIDs that are used to locate an object in the system.

2.5.3 Legion. Legion [74, 75] is an effort from Stanford University and Los Alamos National Laboratory (LANL). Legion is a data-centric programming model targeted for heterogeneous distributed systems. Legion aims to provide locality (data close to the computation) and independence (computation on disjoint data and can be placed on any compute component of the system). The main idea of Legion is based on three abstractions for data partitioning: using logical regions, a tree of tasks for using the regions, and a mapping interface for the underlying hardware. Legion provides communication through another low-level runtime system called Realm [76] which supports asynchronous, an event-based runtime for task based computations.

2.5.4 OCR. A comparatively new runtime system, the Open Community Runtime (OCR) [77] is a joint work from Intel and Rice University. Currently, the University of Vienna [78] and PNNL have implementations of OCR, which are called OCR-Vx [78] and P-OCR [54], respectively. The main target of the runtime is to realize the opportunity of exascale systems. In the exascale era, the authors argue that the HPC community will look for an alternative to the MPI+X model. OCR started with its formal specifications [79]. OCR is an asynchronous many task (AMT) runtime system for exascale where the main idea is to express computations through tasks, events, and data blocks.

2.5.5 Argobots. Argobots is a lightweight low-level threading API developed at Argonne National Laboratory (ANL) as part of the project Argo in 2016 [55, 80]. Argobots provides integrated support for MPI, OpenMP, and

I/O services. Argobots provides richer capabilities when compared to existing runtimes, offering more efficient interoperability than production OpenMP, a lower synchronization cost when MPI is used, and better I/O services. In Argobots, functions are expressed as ULT (ultra-light tasks) and tasklets. ULTs have a stack (similar to OS threads but smaller) that provides faster context-switching.

2.5.6 Unitah. Uintah [81, 82, 83] is a set of libraries for large-scale simulation. It provides a unified heterogeneous task scheduler and runtime originating from the University of Utah's Imaging institute. Originally, Uintah supported an MPI-only approach for out-of-order execution. However, when multicore processors became common, the MPI-only approach did not work very well because MPI ranks need to send and receive messages to transfer data, even if the ranks are housed in the same SMPs. For this reason, a master-slave model is adopted by Uintah runtime, where MPI ranks have multi-threaded execution. The master thread does the data communication with other MPI ranks, and other threads work on the computation. Later, the design of the scheduler was changed in Uintah to support a computation offload model where Uintah can work on heterogeneous systems to offload work for CPUs and GPUs.

2.5.7 PaRSEC. PaRSEC [23, 84], an effort from the University of Tennessee, Knoxville, was introduced in 2012. PaRSEC provides a dataflow programming model. The main idea is to express a program through dataflow between different parts of the code. When dataflow is defined, the dependencies get exposed. This representation of the dataflow acts as a hint to the runtime system for orchestrating the DAG execution on available hardware.

2.6 GAS Based Runtime Systems

In this section, GAS based runtimes are discussed.

2.6.1 UPC. UPC [85] is one of the pioneers of modern PGAS languages. It originated from LBNL in 1999. As previously mentioned, it is considered to be the descendent of SPLIT-C [35]. It provides an option for distributed data structures for reading from and writing to different nodes. In other words, data structures reside in nodes but can be accessed from other nodes. UPC provides a fixed SPMD model where parallelism is fixed from the beginning of a program. UPC can be imagined as a collection thread executing in a globally shared address space.

2.6.2 Chapel. Chapel [22] is a programming language that emerged from CRAY's effort in DARPA high performance computing system program (HPCS). Chapel [86] is a PGAS language (a separate language) similar to high-level programming languages like C, Java, and Fortran that provides a global view of the system it is running on and supports a block-imperative programming style. The creators argue that the main reason for a new language is to set the users in the right state of mind where users know that this is not a sequential program; instead, it is a parallel program. Chapel provides all the basics of high-level programming such as loops, conditions, types, etc.

2.6.3 X10. The X10 language [87, 44] is a member of the PGAS family. The IBM Watson lab introduced it in 2005 as part of the DARPA High-Performance computing program (HPCS). X10 was made available at the start of the many-core era and targeted large shared multiprocessor (SMP) environments where processors would have non-uniform access to memory (NUMA). X10 introduces object-oriented facilities by having JAVA as the foundation for sequential programming languages. X10's goal was to provide a way for programmers to go beyond standard JAVA constructs and provide HPC-specific constructs that do not depend on JAVA, such as asynchronous execution and multidimensional arrays.

2.7 Heterogeneous Runtime Systems

Heterogeneous runtime systems are discussed in this section.

2.7.1 OpenCL.

The Open Computing Language (OpenCL) [47] standard is managed by the Khronos group. The first specification for OpenCL 1.0 was released in 2009. OpenCL is designed for heterogeneous systems with different devices from different manufacturers. OpenCL provides queues for each device, and the CPU is considered as host. The host can enqueue kernels for execution in a blocking and non-blocking way. The API provides means to transfer data between the host and the device and various synchronization functionalities. The abstraction layer provided by OpenCL makes creating scalable code for different vendors easy. The OpenCL execution model has different hierarchies. When a device from a specific vendor is chosen, those hierarchical execution constructs are mapped to the underlying device driver.

2.7.2 OpenACC.

Realizing the popularity of the directive-based programming approach, Cray, NVIDIA and PGI developed the OpenACC [88] programming standard for accelerators in 2012. The main idea was to simplify parallel programming for heterogeneous CPU/GPU systems. High-level abstractions through directives hide all the detail of offloading a kernel to GPUs. Moreover, it ensures portability to different manufacturers.

2.7.3 StarPU.

The StarPU [5] runtime system was introduced in 2011 by the Inria Institute, located in France. The main idea of StarPU is to provide a task based programming model capable of heterogeneous execution (CPU/GPU). The primary data structure of StarPU is called a *codelet*. A computational kernel is expressed as a *codelet* where the kernel can be executed in a CPU, CUDA device, or in an OpenCL device.

2.7.4 CUDA. CUDA [46] is a platform and application programming interface developed by NVIDIA and was introduced in 2007. CUDA is the catalyst for bringing GPGPUs to the HPC community. Because of its throughput-oriented approach, CUDA was capable of providing significant computation power. CUDA became popular quickly, and now, CUDA devices are found in every large computing facility. CUDA devices have a large number of low-performance cores where CUDA threads run. CUDA implements the host and device concept where the host CPU can offload computation to a CUDA device through the CUDA API.

2.7.5 HIP. Similar to CUDA, AMD launched its ROCm [7] platform for GPUs. The ROCm platform consists of different tools, compilers, and libraries. In 2016, AMD introduced the Heterogeneous Compute Interface for Portability (HIP) API for GPUs [89]. The ROCm stack consists of user code at the top, the HIP API that expresses the programming model, the HCC compiler that compiles HIP code, the HSA API and runtime for AMD GPUs, and the amdkfd driver for AMD GPUs. In this chapter, the name HIP is used to describe both the programming API and the driver.

2.8 Runtime Feature Comparison

This section compares and contrasts runtime systems based on their programming models, APIs, execution models, memory models, and synchronization strategies. Later in this section, runtime systems are compared based on their communication and distributed execution features.

2.8.1 Programming API and Model. A programming API provides a means of expressing the programming model for a runtime system. Programming APIs that conform to a high-level language standard come in various forms. This layer is typically the highest level of abstraction provided by the underlying runtime system.

Programming APIs play a critical role in determining the usability of a runtime system. There is a trade-off between abstraction and control. On one hand, if the API provides a very high level of abstraction, a user may unwillingly cede control of fine-grained optimizations to the runtime. On the other hand, if a user wants to control fine-grained optimizations through the API, the source code can lose readability. For example, the code explicitly expresses the memory mapping strategy and synchronization techniques in MPI programs. Compared to modern runtime systems such as HPX, MPI does not provide a high level of abstraction. However, it offers the user full control over the key factors affecting performance. Modern runtime systems carefully balance this trade-off by providing different levels of control to empower users.

The APIs of modern HPC runtimes are written in a high-level language such as C/C++/Fortran/Java. Table 1 shows the language and the compiler for different programming APIs. There are similarities in how these programming models are expressed. We describe these similarities below.

2.8.1.1 Directive-Based. Directive-based programming models are favored by many runtimes for their capability to provide a high level of abstraction and the ease with which they allow the user to express loop-level parallelism. The two most common directive-based programming models are OpenMP and OpenACC. Both of these programming models provide execution schemes for the CPU and the GPU. To express parallelism, a user identifies a parallel region, and through a compiler directive (pragma), notifies the runtime of the execution strategy to implement. This simplicity has made OpenMP one of the most popular and ubiquitous programming

Table 1. Programming model and API.

Runtime	**Language**	**Compiler support**
Cilk Plus	C/C++	Built in Intel compiler and others
TBB	C++	Built in Intel compiler and others
OpenMP	C/C++/Fortran	In all major compilers
Nanos++	C/C++	Mercurium for OmpSs
Qthread	C/C++	Standard compilers
Charm++	C/C++	Charm has it's compiler
HPX	C++	Standard C++11, 14, 17
Legion	C/C++/Regent	Standard/Regent compiler
OCR	C/C++/Fortran	Several implementation
Argobots	C/C++	OpenMP (GNU) and MPI
Uintah	C/C++	MPI+X
PaRSEC	C/C++	Own compiler for two stages
UPC	UPC	UPC compiler
Chapel	Chapel	Chapel compiler
X10	X10	X10-Java compiler
StarPU	C	Standard compilers
OpenCL	C/C++/Python	Standard compilers
OpenACC	C/C++/Fortran	Standard compilers
CUDA	C/C++/Fortran	nvcc compiler from CUDA
HIP	C/C++	hcc compiler from ROCm

models in high-performance parallel computing. The OmpSs programming model from Nanos++ and Bolt [90] from Argobots also provide a directive-based approach for expressing parallelism.

2.8.1.2 Expressing Asynchronous Execution. Many runtimes offer asynchronous execution. However, the programming model and the programming API provide the flexibility to the user to specify which portion of the code to run asynchronously. While some AMT runtimes such as Charm++, HPX, Cilk, and StarPU provide implicit asynchronous execution schemes based on data-dependency graphs and non-blocking execution, the APIs of other runtimes offer special constructs for asynchronous execution. Chapel introduces the “cobegin” construct that instructs the runtime system to execute the task in parallel. However, a descendent or child of the parallel task executes asynchronously depending on the implicit data dependencies in the program. Similarly, OmpSs uses the "concurrent" construct to implement relaxed data-dependency. OCR uses event-driven tasks (EDT) for asynchronous execution. X10 uses "async" to create asynchronous tasks, while Charm++ and HPX use futures.

2.8.1.3 GPU Programming.

Programming models for enabling parallel execution on GPUs are significantly different from those that run on CPUs. OpenMP 4.0 and OpenACC successfully hide GPU-specific support, and the runtime system takes care of implementing those details. However, OpenCL, CUDA, and HIP provide a low-level programming API to express the GPU programming model. Table 2 shows the similarities in how OpenCL, CUDA, and HIP allow the user to express parallelism. Generally, all of these programming models divide a GPU computation into a grid of thread blocks. The runtime then maps these thread-blocks onto the streaming multiprocessors (SM) (AMD calls these "compute units" (CU)). While the concept is similar, the terminologies are different in OpenCL, CUDA, and HIP. OpenCL provides separate computation queues for different heterogeneous compute elements on the system. OpenCL supports both NVIDIA and AMD GPUs. ROCm provides a layer for translating CUDA code into HIP code that allows CUDA code to run on AMD GPUs.

Table 2. Similarity between CUDA, HIP and OpenCL.

Runtime	**Grid**	**Thread Block**	**Thread**	**Warp**
OpenCL	NDRange	work group	work item	sub-group
CUDA	grid	block	thread	warp
HIP	grid	block	work item/thread	wavefront

2.8.1.4 New Language with Special Compiler.

Some runtime systems offer compilers alongside their programming APIs. Chapel provides a compiler for its

Table 3. Execution models in different HPC runtimes.

Runtime	**Execution model of different HPC runtimes**
Cilk Plus	Asynchronous task (DAG), Fork-join, SIMD
TBB	Asynchronous task (DAG), Fork-join, SIMD
OpenMP	Fork-join, SIMD
Nanos++	Asynchronous task (DAG), Fork-join, SIMD
Qthread	Asynchronous task (DAG), Fork-join, SIMD
Charm++	Message driven asynchronous execution. DAG of tasks
HPX	Message driven asynchronous execution. DAG of tasks
Legion	Asynchronous execution builds a Tree of tasks
OCR	Event driven Asynchronous execution. DAG of tasks
Argobots	Fork-Join execution that builds a DAG of tasks
Uintah	MPI + X DAG of Tasks
PaRSEC	Event driven Asynchronous execution.DAG of tasks
UPC	Pthreads with GAS (supports asynchronous execution)
Chapel	Asynchronous execution builds a DAG of tasks
X10	Asynchronous execution builds a DAG of tasks
StarPU	Asynchronous execution builds a DAG of tasks
OpenCL	Heterogeneous execution: different execution scheme
OpenACC	Heterogeneous execution: different execution scheme
CUDA	Data parallel execution
HIP	Data parallel execution

language. X10's compiler translates X10 code to Java or C++ code. Charm++ uses its compiler wrapper for Charm++ codes. PaRSEC also provides a pre-compiler to translate its data-flow representation of the task-graphs into C code. The programming APIs that are implemented as library and language extensions can be compiled through standard compilers. Some APIs use recent features of high-level programming languages. For example, HPX uses constructs from C++11, C++14, and C++17.

2.8.2 Execution Model. The execution model refers to the actual execution scheme a program follows while executing. After the program is compiled, the binary has all the instructions for the runtime to shape its execution. Similar-looking code can behave differently depending on the underlying runtime system.

Some HPC runtime systems support distributed execution, usually through the SPMD execution model. However, the detail of the execution model varies. Table 3 provides an overview of the execution models supported by different HPC runtime systems. The commonly found characteristics are described below.

2.8.2.1 Task Parallel vs Data Parallel. In the task-parallel model, distinct tasks execute in parallel. This is in contrast to the data-parallel model, which extracts parallelism from SIMD instructions. Historically OpenMP followed a data-parallel execution model before OpenMP 3.0, which introduced task parallelism. Task-parallel execution provides more flexibility for the user to extract parallelism from situations where data-parallelism does not apply. However, OpenMP tasks are heavy-weight OS threads which make context switching slow. Thus, OpenMP provides tasking at a coarse-grained level. The same is true when MPI + X applications employ OpenMP. OpenMP follows a fork-join model. Even though OmpSs and Nanos++ are considered forerunners of OpenMP, they do not support the fork-join model. However, Cilk, TBB, and Argobots follow the fork-join model.

2.8.2.2 Asynchronous Many Task Parallel. Because OS threads are bulky and creating and destroying them incur a prohibitively high overhead, it is not feasible to use them within many task runtimes. For this reason, many task runtimes use lightweight tasks. These tasks can be a simple function call or a group of instructions within a function. These tasks are easy to create and destroy, and they also yield quickly. Asynchronous execution runtimes which employ lightweight tasks take advantage of the fact that these tasks leave a small memory footprint to reduce context-switching latency. The asynchronous many task model of execution has become popular in modern runtime systems. Such execution models create a graph of tasks with and without dependency among the nodes. In such a model, the number of tasks can be in the range of millions (some distributed memory runtimes report handling up to 100 million tasks). Cilk, TBB, and Qthreads are examples of such many task runtimes. As they employ a shared memory model, the number of tasks is significantly smaller when compared to the distributed memory model. Runtime systems that can work with distributed memory architectures such as HPX, Charm++, and Legion can spawn a very high number of tasks to provide parallelism. OCR, Argobots, PaRSEC, Chapel, X10, and StarPU also follow the many task execution model.

2.8.2.3 Message Driven vs. Message Passing. The message-passing model expresses code in an SPMD model. Messages allow the transfer of data between two processes. In such a model, computation is in the driving seat. For example, the MPI + X model follows the message-passing paradigm. Runtime systems such as Uintah and Argobots follow the message-passing model. In the message-driven model, the data dependency dominates program execution, and messages coordinate the execution flow. HPX and Charm++ follow the message-driven execution model. Data or a message is sent to different *chares* in Charm++, while HPX sends computation towards the data. Moreover, there is another model called event-driven execution that is similar to the message-driven paradigm. OCR and PaRSEC belong to this category.

2.8.2.4 GPU Execution. The GPU execution model follows a single-instruction, multiple-thread (SIMT) model. As shown in Table 2, the runtime scheduler assigns thread blocks to different streaming processors. The runtime then divides the threads into a set of warps (32 threads form a warp on an NVIDIA GPU and 64 threads in AMD GPUs) [91]. These warps or wavefronts execute in SIMD fashion. OpenACC and OpenCL provide a high-level construct to utilize the heterogeneous platform. Both OpenCL and OpenACC employ device-wise queues to enqueue different kernel executions.

2.8.3 Memory Model and Synchronization. Memory handling is one of the main bottlenecks of achieving high performance since synchronization is needed when multiple compute entities try to access the same memory. Moreover, synchronization is necessary for both the application developer to express the computation in a well-coordinated manner, and also for the runtime system to coordinate with different computing entities. Early runtime systems provided synchronization constructs where the entire program synchronized both in distributed memory and shared memory systems. This is referred to as the bulk synchronous model. Bulk synchronous models are easy to implement and understand. However, they suffer from performance penalties and also reduces the utilization of the system. For this reason, asynchronous models are now popular in the runtime community, as they provide fine-grained synchronization to improve utilization. Table 4 shows the memory model and synchronization in runtime. The memory model is discussed first, followed by a discussion of synchronization in the runtime systems.

2.8.3.1 Memory Model. In the shared memory model, all the processors share the same cache-coherent memory space. Every compute element can access any memory location through the memory channel without communicating through the network interface. Cilk, TBB, OpenMP, OmpSs, and Qthreads are examples of runtime systems that run on shared memory. Threads can access shared data structures where they can read and write. However, individual threads have their own memory space and private data. In a distributed memory model, network interfaces connect different nodes. OCR, Argobots, Uintah, and PaRSEC have distributed memory runtimes. In distributed memory runtimes, over-the-network communication

Table 4. Memory models and synchronization techniques in different HPC runtimes.

Runtime	**Memory model**	**Synchronization**
Cilk Plus	Shared memory	Cilk join
TBB	Shared memory	Mutex
OpenMP	Shared memory	Directives
Nanos++	Shared memory	Directives
Qthreads	Shared memory	Mutex and FEB
Charm++	Distributed shared memory	Message, Futures
HPX	Distributed shared memory	LCOs
Legion	Distributed shared memory	Custom locks
OCR	Distributed memory	Events, Data-Block
Argobots	Distributed memory	Mutex, futures
Uintah	Distributed memory	Mutex, MPI
PaRSEC	Distributed memory	Dependency based
UPC	Distributed shared memory	Locks, barriers
Chapel	Distributed shared memory	Sync, single
X10	Distributed shared memory	Clocks
StarPU	Heterogeneous memory	Locks, barriers
OpenCL	Heterogeneous memory	Barriers
OpenACC	Heterogeneous memory	Directives
CUDA	GPU memory	Library call
HIP	GPU memory	Library call

is necessary to access remote memory. However, this communication is not explicitly visible to the user, and the runtime system performs the communication underneath the hood. Charm++, HPX, Legion, UPC, Chapel, and X10 provide a distributed shared memory space programming model. Within a compute node consisting of GPUs, the PCIe bus connects the GPU and CPU memories. Data transfers between the CPU and GPU memories are required to share memory between the two devices. StarPU, OpenACC, and OpenCL operate through this type of GPU-CPU shared memory model. CUDA and HIP operate within a GPU memory structure that hosts a global memory, a local memory, and a shared memory. Both global memory and local memory are slow. Global memory is accessible by all the threads, whereas local memory is local to every thread. Shared memory resides on a streaming multiprocessor (SM), and the threads of the thread block share this memory.

2.8.3.2 Synchronisation. As described in the previous section on execution models, many runtime systems support asynchronous execution where the sequence of execution is non-deterministic. In such execution scenarios, synchronization happens at the task level based on the DAG of dependencies. On the one hand, Cilk, TBB, OmpSs, and Qthreads provide fine-grained synchronization since asynchronous execution is allowed in these runtimes. Cilk provides join [50], TBB provides atomic locks and mutexes [92], OmpSs provides data dependencies and directives [65] and Qthreads provides mutex and FEB for synchronizing the tasks [70]. On the other hand, OpenMP synchronizations are not fine-grained. OpenMP provides directive-based barriers [19]. HPX also uses local control objects such as futures, dataflow objects, etc., that implement the synchronization in a way that ensures that the tasks can keep working without being completely blocked [20]. Charm++ uses messages for data synchronisation [72] and uses futures for task synchronization.

For data access, Legion provides an option for relaxed synchronization. There are two types of coherence: exclusive coherence and relaxed coherence. In exclusive coherence, the synchronization is strict, where Legion follows an order. However, in relaxed coherence, the synchronization order is not maintained. Rather, Legion ensures access. Atomic coherence serializes the access without ordering, and as the name suggests, simultaneous coherence lets two threads partially execute simultaneously. Legion provides reservation (small scope) and phase barriers (user-defined larger scope) for synchronization [74]. In OCR, events are the main synchronization point since OCR uses an event-driven paradigm. For data synchronization between tasks, OCR uses data blocks where access priority determines the serialization of data accesses [54]. PaRSEC provides dependency-based synchronisation [93]. UPC uses locks and barriers for synchronisations [94]. Chapel uses full or empty syntax for synchronization. It supports two types of synchronization variables: *sync* and *single* [22]. The *sync* variable switches state from full to empty for access control, and the *single* variable indicates that it can only be read once. X10 uses clocks for synchronisation [44]. Clocks ensure deadlock-free operation by waiting for some time and then releasing. StarPU provides locks, nested locks, critical sections, and barriers for synchronization [49]. OpenCL provides three types of barriers. The first kind is to ensure that OpenCL executes all the items in the queue. The second kind synchronizes all the work-items in a work-group on a device. The third kind ensures the synchronization among sub-groups [47]. OpenACC provides directive-based synchronisation through barriers [88]. CUDA and HIP provide synchronization for devices, streams, and threads [46, 89].

2.8.4 Communication, Distributed Support, and GPU Support in HPC runtimes.

Communication is necessary to provide distributed support. Table 5 shows communication mechanisms and distributed execution options for runtime systems. It also shows the time when a runtime system first reported support for GPUs.

Efficient communication is a prerequisite for ensuring high performance in runtimes. Runtime systems either use their own communication framework or use already existing, optimized libraries. Charm++ uses messages for communication. It supports different communication libraries such as MPI and UDP. Charm++ also provides the option for one-sided communication in an RDMA-enabled network. Through the communication interface, Charm++ provides a global address space for supporting distributed execution [95]. HPX consists of a Parcel subsystem that

Table 5. Communication, distributed support, and GPU support.

Runtime	**Communication**	**Distributed support**	**GPU support**
Cilk Plus	None	No (through MPI+X)	NA
TBB	None	No (through MPI+X)	NA
OpenMP	None	No (through MPI+X)	2013
Nanos++	None	No (through MPI+X)	2011
Qthread	None	No (through MPI+X)	NA
Charm++	RDMA	Yes (through GAS)	2016
HPX	Parcel	Yes (through GAS)	2014
Legion	GASNet	Yes (through GAS)	2012
OCR	MPI messages	Yes (through MPI)	NA
Argobots	MPI Messages	yes (MPI + X)	2016
Uintah	MPI Messages	yes (MPI + X)	2013
PaRSEC	MPI messages	yes (Through MPI)	2012
UPC	GASNet	Yes (through GAS)	2014
Chapel	GASNet	Yes (through GAS)	2019
X10	MPI Messages	Yes (through MPI)	NA
StarPU	None	No (through MPI+X)	2011
OpenCL	None	No (through MPI+X)	2011
OpenACC	None	No (through MPI+X)	2012
CUDA	None	No (through MPI+X)	2009
HIP	None	No (through MPI+X)	2016

carries out communication across nodes. The Parcel subsystem can communicate using TCP ports or MPI. By using active messages through the Parcel subsystem, HPX enables a global address space for distributed execution [20]. Legion, UPC, and Chapel use the GASNet [43] one-sided communication library for distributed execution. PaRSEC and X10 use MPI messages for communication. The shared memory and accelerator runtimes do not use communication across nodes. However, they can be a part of the MPI + X execution model to support distributed execution.

Most of the modern runtimes support GPUs. Runtime systems use a CUDA, HIP, or OpenCL runtime to provide GPU support in their ecosystem. Cilk Plus, TBB, Qthread, OCR, and X10 do not report support for GPUs.

2.8.5 Opportunities. Two trends are visible in the runtime system domain. The first trend is the adoption of an asynchronous task based approach for improving overall resource utilization. The second trend is including heterogeneous capabilities in runtime systems. The first trend shows the direction where the runtime system community is heading, whereas the second trend results from the evolution of computer architectures. These two trends unveil two opportunities, which are discussed below.

2.8.5.1 Opportunity - 1 : Standardization of Task based Runtime Systems. The HPC community is yet to agree upon a standard for task based runtimes. While some features are similar, every task based runtime system has its methodologies for implementing the asynchronous task based approach. Initiatives such as OCR showcase attempts from the HPC community to design a specification for task based systems. Argobots is another research effort to bring a variety of runtime systems under one umbrella. It is high time for the runtime system community to come together and standardize task based runtime systems.

2.8.5.2 Opportunity - 2 : Addressing Heterogeneity. The trend of heterogeneous systems is likely to be continued. As of now, the majority of the runtime systems include support for GPUs. Runtime systems such as OpenCL also include support for FPGAs. However, managing how the processors will be used during execution is mostly the programmer's responsibility. Moreover, newer architectures are also being released (such as deep learning accelerators). For these reasons, efficiently addressing heterogeneity under the umbrella of a runtime system is still an opportunity.

2.9 Dynamic Adaptation in Runtimes

This section explores the dynamic features of modern HPC runtime systems. Specifically, we consider the features that are primarily responsible for providing better performance and energy consumption.

Table 6. Scheduling and load balancing strategies in HPC runtimes.

Runtime	**Scheduling**	**Load balancing**
Cilk Plus	Tasks on Worker thread pool	Yes (work-stealing)
TBB	Tasks on Worker thread pool	Yes (work-stealing)
OpenMP	Heavy OS threads	Yes (work-stealing)
Nanos++	Tasks on Worker thread pool	Yes (work-stealing)
Qthread	Tasks on Worker thread pool	Yes (work-stealing)
Charm++	Tasks on Worker thread pool	Yes (through migration)
HPX	Tasks on Worker thread pool	Yes (through migration)
Legion	Tasks on Worker thread pool	Yes (work-stealing)
OCR	Tasks on Worker thread pool	Yes (work-stealing)
Argobots	Stacked custom scheduling	Yes (work-stealing)
Uintah	Tasks on Worker thread pool	Yes (Dynamic adaptive)
PaRSEC	Tasks on Worker thread pool	Yes (work-stealing)
UPC	None	None
Chapel	Future plan	Future plan
X10	Tasks on Worker thread pool	Yes (work-stealing)
StarPU	Multiple	Yes (work-stealing)
OpenCL	Heterogeneous queues	Dependent
OpenACC	Heterogeneous queues	Dependent
CUDA	GPU scheduling	Yes
HIP	GPU scheduling	Yes

2.9.1 Scheduling and Load Balancing.

Scheduling is one of the most critical tasks that an HPC runtime performs. In early parallel programming models, scheduling used to be mostly static. The mapping between work and resource did not change during execution after expressing the parallelism through programming APIs in early MPI or OpenMP applications. However, in modern HPC runtimes, the scheduling scenario is highly dynamic. It is almost impossible to determine the optimal mapping between work and resources since the optimal fine-grained mapping keeps changing. The main idea behind this non-deterministic mapping is to increase the utilization of underlying hardware. As a result, scheduling is one of the areas where dynamic adaptation plays an important role. Dynamic scheduling enables better load-balancing that, in turn, results in better system utilization. We discuss the scheduling and load balancing strategies of different runtime systems below.

Table 6 shows the scheduling approaches adopted by different runtime systems. Each category presented in the table is further elaborated in the following sections.

2.9.1.1 Scheduling Using OS Threads. OpenMP uses direct task mapping on OS threads. Every time OpenMP creates a task, OpenMP assigns the task to an OS thread. This OS thread is created at the beginning of the parallel region and joins with the master thread when the parallel region ends. However, OpenMP is unaware of the task-to-thread mapping strategy to implement in advance (it can be specified). OpenMP standard provides five types of scheduling: 1) static, 2) dynamic, 3) guided, 4) auto, and 5) runtime [19]. It also provides the option to change the chunk size. The number of loop iterations each OS thread gets assigned depends on the chunk size. Static scheduling distributes the number of iterations equally if the chunk size is not specified. If the chunk size is specified, OpenMP allocates chunks to different threads in a round-robin fashion. In dynamic scheduling, each thread works on an initial chunk and requests more chunks as required. Guided scheduling works like dynamic scheduling, except that the chunk size keeps decreasing. When the user specifies auto as the scheduling option, the compiler decides the data distribution. When the runtime scheduling is selected, OpenMP determines the chunk sizes at runtime. Using dynamic scheduling, OpenMP can achieve work-stealing load-balancing. OmpSs [65] also implements the same strategies.

2.9.1.2 Worker Thread Pool. The most common strategy for scheduling in many task runtime systems is to have a pool of worker threads. The runtime system has task queues that contain ready-to-be-executed tasks. Like the producer-consumer approach, when a task's dependencies are resolved, they are placed on the ready queue. The pool of worker threads keeps pulling tasks from the ready queue. A majority of the many task runtimes implement this strategy. Cilk Plus, TBB, Nanos++, Qthread, Charm++, HPX, Legion, OCR, Uintah, PaRSEC, and X10 all implement some variant of this strategy. The main benefit is the increased utilization of the resources. However, the queue structure and the number of queues differ in different runtime systems. Uintah implements a unified schedular where MPI, Pthread, and CUDA can work together in an out-of-order fashion where the pthreads are the worker pool that consumes work from the CPU queues. It has a scheduling option for MPI processes as well. The load balancer in Uintah can provide dynamic adaptation in runtime by changing how much computation each processor performs [83]. Nanos++ holds a ready task queue where all the tasks have their dependencies resolved (supports yielding) [65]. Qthread employs a similar strategy where the worker pool is called a "collection of a shepherd" (uses chunk size) [69].

In Charm++, each PE (worker thread) has its pool of messages and a collection of chares. As Charm++ employs a message-driven paradigm, each PE selects a message from the pool and executes the method of a chare for which the message is meant for. Charm++ provides an advanced load balancing strategy through migration. It can provide load balancing in a centralized or distributed way. It also employs a measurement-based load balancing strategy. Charm++ creates a database of information that facilitates periodical load balancing using the prediction of the imbalance. It also provides different algorithms for load balancing (Greedy, Refine, Rotate, etc.) [95]. Like Charm++, HPX also keeps queue(s) of tasks for each OS thread. HPX also provides multiple priority queues where HPX executes high priority queues first. HPX provides different scheduling options (Priority local scheduling, Priority ABP scheduling, etc.). Through the Priority ABP scheduling, HPX can provide NUMA sensitive scheduling where HPX assigns the highest priority to the same NUMA domain [20]. HPX provides a load balancing option for in-node through work-stealing among the worker threads and also distributed load balancing through task migration [96]. In Legion, the underlying Realm runtime manages the worker thread pool. This pool creates a queue for each thread and asynchronously executes them [76]. The mapping interface of legion runtime provides a mechanism for distributed task-stealing for load balancing [97]. OCR also uses a worker thread concept where the load balancing is supported through work-stealing using a work-first or help-first mechanism. The Habanero runtime (an upgrade of X10) [98, 99] is the inspiration for OCR's load balancing strategy. Argobots uses a stacked scheduler concept where multiple schedulers can be applied for different software modules during execution [55]. While using a set of worker threads, it also uses work-stealing load balancing. Like HPX, PaRSEC and Nanos++ also provide NUMA-aware scheduling for better performance [84, 65]. PaRSEC also supports inter-node and intra-node load balancing using work-stealing [84]. X10 provides load balancing custom work-stealing method through GLB library [100].

2.9.1.3 GPU Scheduling. Both CUDA and HIP provide streams for devices. Streams execute the kernels sequentially in a first-come, first-served manner [101]. However, at a lower level, each streaming processor (SM) schedules warps (a set of threads) from the assigned thread blocks [91]. Each streaming processor has multiple warp schedulers that pull ready warps to execute from the queue to increase utilization. When a kernel starts executing on the GPUs, the scheduler assigns thread blocks to SMs. Much detail of the scheduling at the SM level is not revealed [101].

2.9.1.4 Heterogeneous Scheduling. Scheduling in StarPU also follows a group of workers where the workers can be accelerators as well [5]. The default schedular in StarPU is a work-stealing scheduler. However, StarPU has different options for scheduling. StarPU can implement performance models to find out the appropriate target for specific tasks. Moreover, when declaring "codelets" in StarPU, the user can specify a priority for tasks that acts as a hint to the runtime. Based on these hints, StarPU schedulers can provide greedy scheduling [49]. The gang, worker, and vector constructs define the scheduling in OpenACC [102]. Based on the specified size of these variables underlying heterogeneous processors is used. Further, the underlying driver for the device implements the scheduling decision. OpenCL provides device-wise queues for heterogeneous systems [103]. When scheduled to the device queues, the device uses its internal scheduling at the execution time.

2.9.1.5 Opportunity - 3 : Task Placement in Heterogeneous Systems. Only the StarPU runtime system provides scheduling mechanisms for task placements on CPUs and GPUs. Other runtime systems do not do so. However, the capability to operate on heterogeneous systems has become common in almost all modern runtime systems. For this reason, a heterogeneous task placement scheduler would be a fruitful addition to them. There are ad-hoc studies that implement task placement scheduling algorithms on the OpenCL runtime system [104, 105], which provide the validation of this opportunity.

2.9.2 Energy Aware Features and Studies. Energy consumption is one of the biggest concerns surrounding the operation of exascale systems [106]. For this reason, new processors (both CPU and GPU) come with predefined TDP levels and frequency sets. These new technologies enable the processors to adjust their clock frequencies dynamically to ensure that they adhere to the power budget. Moreover, CPU and GPU vendors provide interfaces to monitor and allow changing these states through those interfaces. For example, Intel provides running average power limit (RAPL) [107] and NVIDIA provides NVIDIA management library (NVML) [108] to monitor and control power-related attributes. These interfaces enable runtime systems (or OS) to select certain settings to limit energy consumption by sacrificing processing power. This "soft" control enables the runtime system to dynamically select the energy consumption mode depending on the priority, need, or hardware status. Such control has proved beneficial as it provides an extra layer of control to make energy-aware decisions. This section discusses energy-aware capabilities in runtimes and methods. At first, we discuss the most common energy-aware techniques suitable for runtime systems. Later, we present a discussion of the energy-aware decision capability that exists in current runtimes.

2.9.2.1 Dynamic Voltage and Frequency Scaling (DVFS). Dynamic voltage and frequency scaling (DVFS) is one of the oldest methods to achieve dynamic power behavior. Many of the current processors have DVFS capability. In a DVFS capable system, processors and memory have a set of frequencies in which they can operate. In most of the devices, the frequency is selected by the operating system when DVFS is enabled. Usually, the frequency selection depends on the utilization of the unit. There has been a considerable amount of research done in the area of DVFS. Ma et al. [109] designed a GreenGPU that dynamically throttles the frequency of the GPU and the memory. Komoda et al. [110] also studied power capping using DVFS to find near-optimal frequency settings for CPU-GPU. Liu et al. [111] designed an energy-aware kernel mapping strategy that assigns different frequencies to PUs in a heterogeneous system using DVFS.

2.9.2.2 Power Capping. Power capping is a technique that restricts the instant power consumption of a device. The main components of a system are the processors and the memory. Each processor has a certain number of frequencies that it can operate in, and the same is true for the system memory. Selecting a higher frequency guarantees a higher speed for the processor or memory but also consumes more power. For this reason, by opting for a lower frequency, the runtime can limit the maximum power consumption of a device instantly. A modern integrated device such as the NVIDIA Xavier has a predefined power cap. For example, Xavier has five predefined power caps that the runtime software can dynamically invoke. For example, Zhu et al. [112] dynamically finds the appropriate frequency to keep the application execution under a power cap for a heterogeneous system consisting of a CPU and a GPU. Using a machine learning technique, the strategy proposed by the authors can select a frequency that is capable of keeping a device under a power cap.

2.9.2.3 Energy-aware Features in Runtimes. Some runtimes include energy-aware features in their design. Charm++ provides the capability to change the CPU core frequency using DVFS [72]. Charm++ provides a load balancer that monitors the average temperature of the chip and changes the core frequency when the temperature crosses a threshold. These thresholds are application-specific, and the user can set them. When the frequency is lowered for a set of cores, the runtime calculates the load of the processor cores and identifies under-utilized and overloaded cores. After identification, the runtime load balancer migrates tasks from the overloaded to underloaded cores. The runtime repeats this process many times during the application's execution. This approach provided energy savings without much performance penalty [113]. HPX does not provide energy-aware features, but they claim to improve overall energy efficiency by increasing the utilization of the resources through over decomposition of tasks [3]. The same argument is made by OCR [78]. PaRSEC provides integration with PAPI [114] for power measurement at the task level. However, the runtime does not dynamically adapt itself using the power measurements [115]. StarPU provides energy-aware scheduling where the runtime system turns off the CPU cores to save energy. A "codelet" can be specified with an energy model, and based on that model, the runtime system adjusts the task distribution [49]. Other runtime systems do not provide energy-aware capabilities.

2.9.2.4 Opportunity - 4 : Energy-aware Decision Making Capability. Studies suggest that runtime systems can provide efficient energy-aware decisions and offer a good trade-off between energy and performance. At the same time, it is also evident that not many runtime systems provide energy-aware features. For this reason, an energy-aware runtime that works well with hardware from different processor architectures would be a critical feature for the runtimes targeted for future exascale systems.

2.9.3 Dynamic Adaptation Tools and Interfaces. The runtime system is an active component that can interact with external entities. Such interaction can impose control on the decisions taken by the runtime. However, proper APIs need to be exposed by the runtimes for external systems to interact and influence their behavior. Table 7 provides a summary of the different tools and interfaces for dynamic adaptation of the runtime systems. We discuss these in detail below.

2.9.3.1 Interfaces for Runtimes. Some runtime systems reveal interfaces for the sake of collecting performance data during execution. Runtimes allow an external entity to register callback functions to provide the status or value of different

Table 7. Tools and interface for dynamic adaptation for HPC runtimes.

Runtime	**Interface and online tools**	**Adaptation capability**
Cilk Plus	No	No
TBB	No	No
OpenMP	OMPT interface	Exists
Nanos++	Event collection	No
Qthread	RCRdaemon tool	Exists
Charm++	PICs tool	Exists
HPX	APEX tool	Exists
Legion	Profiling Interface	No
OCR	No	No
Argobots	No	No
Uintah	No	No
PaRSEC	No	No
UPC	GASP interface	No
Chapel	No	No
X10	No	No
StarPU	Profiling Interface	No
OpenCL	Profiling Interface	No
OpenACC	Profiling Interface	No
CUDA	CUPTI (profiling)	No
HIP	roc-profiler library	No

runtime variables through these interfaces. Having such a generic interface defined enables tuning runtime variables during execution. MPI 3.0 specification included MPI_T interface that allowed the community to design runtime introspection tools to change different parameters for efficient communication [116, 117]. Similarly, OpenMP 5.0 included OMPT, which is a tool interface for OpenMP. Similar to MPI_T, OMPT provides the ability to register callbacks to get the status of various runtime system parameters and timers [118, 106]. UPC provides the GASP [119] interface for registering callbacks. However, GASP does not provide the flexibility to change any runtime variables. Rather, GASP provides the facility for other tools to collect data (similar to the PMPI profiling interface in MPI). CUDA also provides the CUPTI profiling interface, but it does not provide an option for changing runtime variables [120]. The roc-profiler [121] of ROCm collects GPU performance data from AMD's HSA runtime [122]. StarPU also provides an online profiling interface and does not provide the option to change runtime variables [49]. Both OpenACC and OpenCL also provide a profiling interface designed only to enable querying and collecting runtime events [47, 88]. Similarly, Legion provides a profiling interface where the status of memory and tasks, execution time, and current load of the system can be obtained [97]. The Nanos++ and OmpSs ecosystems also provide an instrumentation option that can provide runtime events [65].

2.9.3.2 Dynamic Adaptation Tools for Runtimes. APEX [9] is an autonomic performance measurement and analysis tool designed for task based runtimes. It has support for HPX and OpenMP runtimes. APEX hosts a policy engine that can monitor runtime events and activate a policy based on that. Moreover, APEX can also implement a periodic policy. The APEX policy engine uses the Active harmony library [123] to change runtime parameters and observe their impact on performance. If APEX finds that performance improves, the policy engine continues modifying the runtime knob until it finds a near-optimal solution. Charm++ provides PICS [124] which can optimize application performance based on a control-point centric mechanism. Similar to APEX, PICS also collects information from the runtime system about the overall status. Unlike APEX, PICS employs control points both in the application and the runtime. Using a decision tree, PICS can tune different applications and runtime knobs based on the observed data. The RCRdaemon [125] can work with the Qthreads runtime. RCRdaemon continuously monitors the memory and utilization status from the OS. When the Qthreads scheduler starts execution with its worker thread pools (Pthreads), the adaptive scheduler in Qthreads communicates with RCRdaemon to find the optimal number of threads.

2.9.3.3 Opportunity - 5 : Dynamic Adaptation Tools and Interfaces. While most runtime systems provide profiling interfaces, only a few provide tools or expose an API to enable dynamic adaptation. Therefore, developing tools capable of dynamically adapting runtime systems is desired. Even though many runtime systems are open-sourced, building a custom tool for dynamic adaptation ties that tool to that particular runtime. For this reason, modern many-task runtimes need a general tool or interface specification solution such as MPI_T or OMPT.

Table 8. Opportunities addressed in this dissertation.

Opportunities	**Chapter**	**Contribution**
Standardization of Task based Runtime Systems	N/A	This dissertation does not address it
Addressing Heterogeneity	Chapter IV	Considers a heterogeneous system with a CPU, GPU and vision processor
Task Placement in Heterogeneous System	Chapter IV	Develops a scheduling algorithm for task placement
Energy-aware Decision Making Capability	Chapter IV	Considers energy-performance trade-off
Dynamic Adaptation Tools and Interfaces	Chapter III	Develops policies in a dynamic adaptation tool

2.10 Summary

This chapter investigates the evolution of HPC runtimes to identify the dynamic adaptation techniques and opportunities. Since the beginning of parallel computing, HPC runtimes have gone through major changes. These changes are caused by new architectures, increasing compute capabilities, upgrades in interconnect technologies, and the introduction of heterogeneity. Moreover, continuous innovation by the community and the decision to come together to standardize popular programming models had a major impact in shaping today's runtimes. Additional layers of abstraction are observed which helps the modular design of the runtime systems. These abstractions are compounding the role of the runtime system during execution.

Runtime systems now perform complex scheduling and load balancing, orchestrate communication, drive accelerators, and asynchronously execute graphs with billions of tasks. In order for the runtime systems to perform as an active entity during execution, dynamic decision making and adaptation have become crucial. Even though the current features of runtime systems provide some dynamic adaptation capabilities, this study identifies that there are dynamic adaptation opportunities in this area.

Table 8 shows the identified opportunities and contribution from this dissertation. Since Opportunity — 1 has a broader scope, this dissertation does not address it. However, Opportunities 2-5 are addressed in Chapters III and IV. Later chapters (Chapters V and VI) address the unsolved sub-problems of Chapters III and IV, therefore contributing to the opportunities listed in Table 8.

CHAPTER III

DYNAMIC ADAPTATION IN HPX RUNTIME

This chapter contains previously published material with co-authorship. All of the presented research in this chapter was conducted as a collaboration between the University of Oregon and Louisiana State University. The parcel coalescing policy (Section 3.3) was presented at ICPP 2018 [11] (poster). The task inlining policy was presented at ICPP 2019 [10]. I was the first author of the ICPP 2018 poster, whereas, for the ICPP 2019 paper, I was a joint first author with equal contribution. For both publications, Dr. Bibek Wagle implemented the baseline static policy in the HPX runtime system, whereas I implemented the dynamic policies. Dr. Kevin Huck was instrumental in conceptualizing dynamic adaptive policies in HPX Runtime by developing APEX tool. Dr. Huck implemented the initial version of the parcel coalescing policy, which I carried forward; however, the task inlining policy was solely developed by me. In both of these publications, Dr. Wagle wrote sections for the baseline policies; I wrote the sections for the dynamic policies. All the co-authors helped in proofreading. This chapter is formulated by gathering all my contributions from these two publications. I listed the contributions from the co-authors in the background section to formulate a coherent story.

3.1 Introduction: Two Problems

This chapter explores the dynamic adaptation opportunity in the HPX [126] asynchronous task based runtime system for CPU architectures (corresponding to the Research Question 2 — **RQ2**: Is it possible to dynamically adapt a runtime system's parameters to achieve better performance for different CPU architectures?). The HPX runtime system decomposes an algorithm into fine-grained units of work and executes them asynchronously. While fine-grained units provide more parallelism, over-decomposition results in higher task overhead and can negatively impact performance. So, there must be a balance between available parallelism and tasking overhead. To ensure this balance, a strategy called task inlining [127] is applied. The task inlining strategy decides whether a parent task would consume the work assigned to a child task in addition to its own, thereby reducing the amount of parallelism. For this reason, the task inlining threshold should be defined in such a way that a proper balance between parallelism and task overhead is achieved.

Tasks in the HPX runtime system communicate using a type of message called a *parcel*. When a task graph is scheduled in a distributed environment, tasks need to send parcels across nodes. Since the number of tasks in the HPX runtime system can be high, a large number of parcels need to be sent between nodes. Large numbers of parcel messages create communication overhead and can negatively impact the overall performance. To reduce the communication overhead, the HPX runtime system implements a strategy called parcel coalescing [128]. The parcel coalescing strategy combines a certain number of messages and periodically sends them to the destination. In doing so, it reduces the communication overhead — however, parcels are also delayed. Thus, there is a trade-off between the communication delay and the communication overhead. For this reason, a proper balance is needed so that the selected parameters of parcel coalescing improve performance.

While task inlining is related to computation, parcel coalescing is related to communication. This chapter delves into these two problems and devises dynamic adaptive solutions to improve performance. At first, a proof of concept for adaptive policy is shown for parcel coalescing. Then, a detailed study of the task inlining policy is presented.

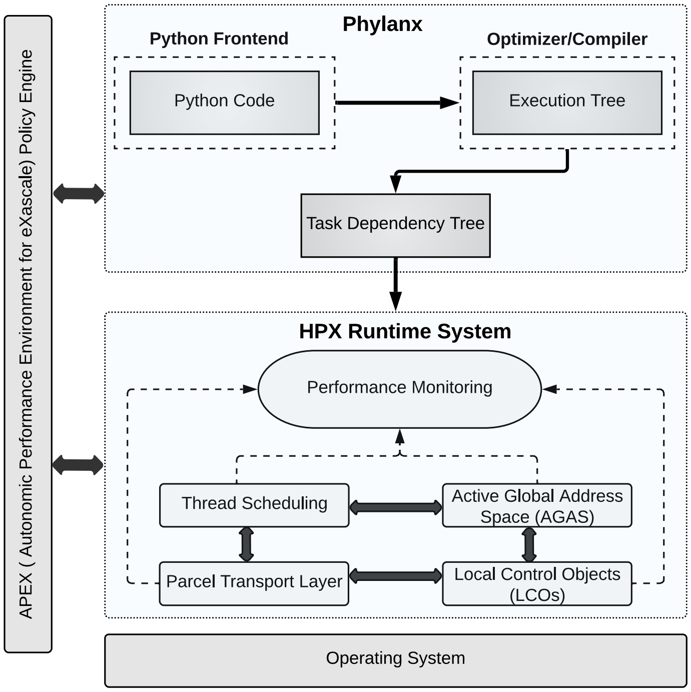


Figure 6. The architecture of HPX and Phylanx along with APEX. HPX consists of a *Threading Subsystem* responsible for scheduling *HPX threads* (lightweight tasks), a *Parcel Transport Layer* for handling message passing and remote method invocations, *Local Control Objects (LCOs)* for synchronization among tasks and an *Active Global Address Space (AGAS)* for addressing object across nodes. Phylanx, developed on top of HPX, transforms Python code into a task dependency tree which HPX executes. APEX provides performance monitoring facilities as well as a policy engine that enables runtime adaptive capabilities.

3.2 Background

3.2.1 HPX. HPX is an asynchronous task based runtime system with a C++ standards-compliant API. The architecture of HPX, along with its various subsystems, is shown in Figure 6. Detailed information about HPX in [129]. In this section, we highlight the relevant information about HPX vital to the comprehension of this chapter.

HPX exploits parallelism by executing lightweight tasks which are scheduled on top of the kernel threads. By default, HPX creates one kernel thread per core. The HPX scheduler schedules the lightweight tasks on top of these kernel threads. HPX can execute a newly created task either as a new thread asynchronously or synchronously in the parent thread, which we will refer to as inlined execution. Asynchrony in HPX is managed via *futures* [130, 129]. A *future* is a placeholder for the result of some computation that is not yet ready. A task requesting the result of a *future* is suspended if the result is unavailable. When the *future* becomes ready, wherein the results of the computation are available, the suspended tasks are resumed.

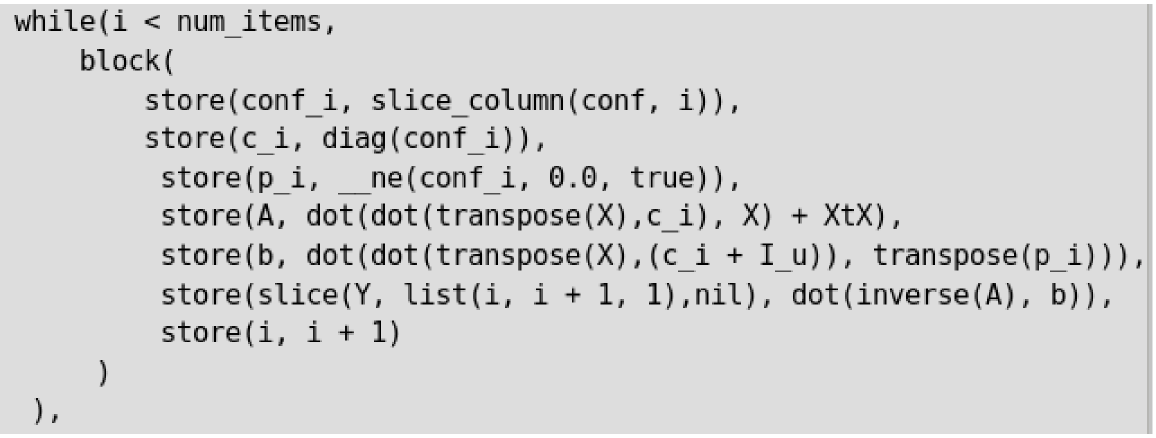
Another important feature of HPX is the *dataflow* [131, 132] utility. HPX makes use of *dataflow* objects for managing data dependencies. A *dataflow* waits until a provided set of *futures* have become ready before executing a predefined callable, which relies on the results referenced by the *futures*. In this work, we use the *dataflow* objects as an injection point for our threading policies.

Finally, HPX provides system-wide support for gathering performance information, known as the performance counter framework. Users can employ this feature to extract information about the state of the application and runtime. If the predefined performance counters do not provide the user with needed functionality, one can easily create a new counter which will report the requested information. This tool is useful for instrumentation and debugging purposes. In addition, HPX and the performance counter framework integrate with APEX, described in Section 3.2.3, which provides additional measurements and runtime adaptive capabilities.

3.2.2 PHYLANX.

Phylanx is a task based, asynchronous array computing toolkit designed to support machine learning applications. User code, written in Python, is transformed into a tree of Phylanx *primitives* known as an execution tree. A *primitive* is an object which can take input, such as the result of a previously executed *primitive*, and exposes a method named *eval* which operates on the object's inputs. Instead of returning the value computed by the *primitive*, however, the *eval* function will return a *future* to the computed value. An execution tree is a collection of these objects which describe the dependencies between all the operations in an application. In this formulation, the nodes of the tree are the *primitives* while the edges of the tree represent dependencies between them. The architecture of Phylanx is shown in Figure 6.

During execution, Phylanx starts to evaluate the execution tree by calling the *eval* function on the root node. Each dependency of this *primitive* calls the *eval* function on each of its dependencies. This operation traverses the tree until a *leaf node*, or a node with no dependencies, is reached. It is important to note that as the execution tree is being traversed, the actual execution of the tasks has not yet begun. Instead, a task graph of futures is being created where each *future* represents a dependency on a previous operation. Once the leaf nodes have been reached, the task graph then begins to execute, as the execution of a leaf *primitive* does not depend on the results of another calculation. The task graph is then summarily executed as the results of dependencies are met, eventually returning the result of the entire tree. As the evaluation of a child node is completed, the result of its execution is passed to the



```
while(i < num_items,
    block(
        store(conf_i, slice_column(conf, i)),
        store(c_i, diag(conf_i)),
         store(p_i, __ne(conf_i, 0.0, true)),
         store(A, dot(dot(transpose(X),c_i), X) + XtX),
         store(b, dot(dot(transpose(X),(c_i + I_u)), transpose(p_i))),
         store(slice(Y, list(i, i + 1, 1),nil), dot(inverse(A), b)),
         store(i, i + 1)
     )
 ),
```

Figure 7. Partial code for the while loop in the Alternating Least Squares algorithm.

parent node. The result of the entire tree is ready after the root node has finished its execution.

Because *eval* uses HPX *dataflow* to launch a *primitive's* operations, we have a runtime injection point where we can decide whether to execute a *primitive's* children asynchronously in a new task or synchronously by inlining the execution. We have added Phylanx specific performance counters that report the amount of time spent executing each subtree of the execution tree and a counter that reports the number of times a node was executed. Using these tools, we can take measurements of executing *primitives* and apply this information to future scheduling decisions.

3.2.2.1 Alternating Least Squares (ALS) Benchmark. To analyze the effects of overheads of scheduling and executing tasks on Phylanx, We used a reference implementation of Alternating Least Squares [133]. Figure 7 is a while loop taken of the reference implementation of the Alternating Least Squares in Phylanx. In Figure 8, this loop is visualized using the Phylanx visualization tool [134] where every node is a particular instance of a *primitive*. The full code for the benchmark can be found in the Phylanx Github repository [135].

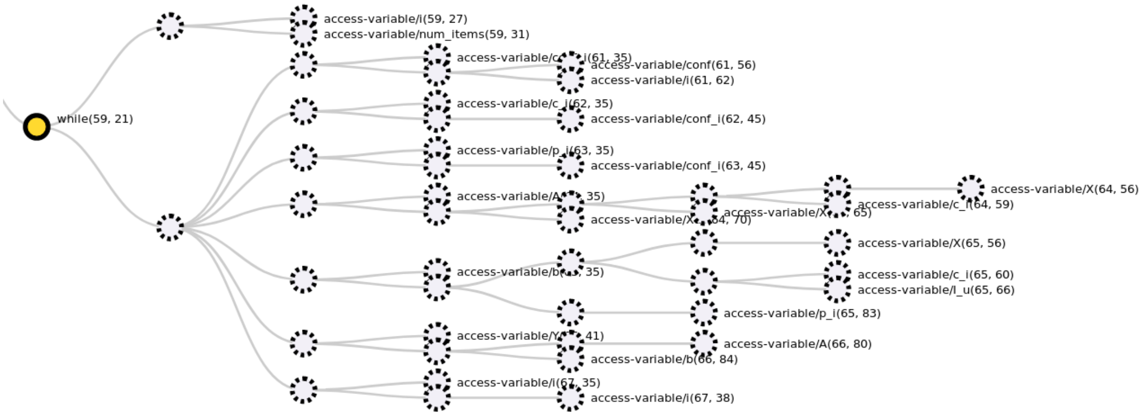

while(59, 21)
access-variable/i(59, 27)
access-variable/num_items(59, 31)
access-variable/c i(61, 35)
access-variable/conf(61, 56)
access-variable/i(61, 62)
access-variable/c_i(62, 35)
access-variable/conf_i(62, 45)
access-variable/p_i(63, 35)
access-variable/conf_i(63, 45)
access-variable/A 35)
access-variable/X 64, 70)
access-variable/X 65)
access-variable/c_i(64, 59)
access-variable/X(64, 56)
access-variable/X(65, 56)
access-variable/b 35)
access-variable/c_i(65, 60)
access-variable/l_u(65, 66)
access-variable/p_i(65, 83)
access-variable/Y 41)
access-variable/A(66, 80)
access-variable/b(66, 84)
access-variable/i(67, 35)
access-variable/i(67, 38)

Figure 8. Visualization of the while loop in the Alternating Least Squares algorithm.

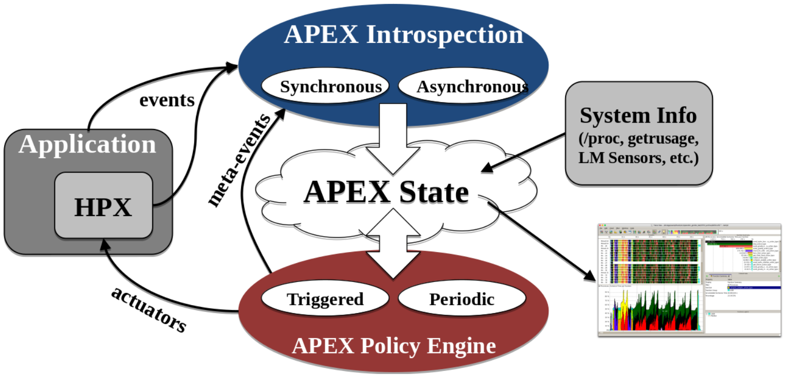


Figure 9. Interaction of APEX with the HPX runtime.

3.2.3 APEX. APEX [136] (Autonomic Performance Environment for Exascale) is a performance measurement library for distributed, asynchronous task based runtime systems such as HPX. It provides lightweight measurement (task ¡ 1ms) and high concurrency. To support performance measurement in systems that employ user-level threading, APEX uses a dependency chain rather than the call stack to produce traces. APEX supports both synchronous and asynchronous introspection. As depicted in Figure 9, APEX collects data through inspectors. The synchronous module of APEX uses an event API and event listeners. APEX decides to start, stop, yield, or resume timers for correct measurements whenever an event occurs.

However, the asynchronous module does not rely on events; instead, it executes desired functionality periodically.

The policy engine of APEX provides a lightweight API to engineer policies that can improve the application's performance, execute a desired functionality on the runtime or select important runtime and application parameters. There are two ways to register a policy: 1. Triggered, and 2. Periodic. A triggered policy can be initiated by a specific event within the HPX runtime. Several of these events are available by default to the user. Additionally, it is also possible to provide a user-defined event, known as a custom trigger. The second class of policies, the periodic policy, operates without any event. Rather, this policy uses a defined timer that is specified during the policy's registration. All policies are stored in a policy queue and executed as instructed. The policy engine is integrated with Active Harmony [137], an online tuning library. Defined policies can use this library to converge on a set of optimum parameters by observing the wall time of the application or by looking at the introspection data gathered by APEX.

3.2.4 Static Baseline Policies. HPX and PHYLANX are equipped with static baseline policies for task inlining and parcel coalescing. Both of these baseline policies are implemented by Bibek Wagle of Louisiana State University. In these static policies, parameters are statically defined and do not change during the execution of an application. The two baseline policies are outlined below.

3.2.4.1 Baseline Policy for Parcel Coalescing. The baseline parcel coalescing policy is implemented in the HPX runtime system. The parcel coalescing policy has two parameters: 1) the number of messages to coalesce, and, 2) coalescing interval. These two parameters are provided as static values and they act as thresholds. The HPX runtime system sets a timer (coalescing interval) and counts the number of messages to send. When the timer reaches the coalescing interval, the gathered messages are coalesced and sent. However, if the number of messages reaches the maximum threshold, HPX performs parcel coalescing without waiting for the timer to reach its maximum.

3.2.4.2 Baseline Policy for Task inlining. The baseline policy for task inlining is implemented in Phylanx. As described earlier in section 3.2.2, each *primitive* in Phylanx has a method called *eval* which evaluates the work defined by that *primitive*. HPX can decide whether to execute the *eval* method asynchronously as a new task or synchronously by inlining the work in the parent task. The baseline policy depends on three parameters: 1) *count_threshold* (set to 5), 2) *lower_threshold* (set to 350 μs), and 3) *upper_threshold* (set to 500 μs).

Given an iterative application, the execution time for each *primitive* instance is evaluated *count_threshold* times to obtain the average execution time of the *primitive* instance. During execution, if *count_threshold* measurements are not obtained for a *primitive*, no decision will be made regarding the inlining of the task. If the previous *primitive* was executed asynchronously, the next execution would be executed asynchronously. Conversely, the execution will be synchronous if the previous execution was synchronous. On the other hand, if measurements are obtained and the average execution time is below the *lower_threshold*, future tasks created for that *primitive* instance will be executed synchronously. If the average execution time is above the *upper_threshold*, future tasks created for that *primitive* instance will be executed asynchronously. When the average execution time of the *primitive* instance lies between the thresholds, the task will be executed with its previous mode of execution until more measurements for the execution time are gathered.

3.2.4.3 Limitation of Static Policy. The drawback of the baseline policy lies in the fact that the optimal threshold values vary with different applications and architectures. Moreover, the number of threads used for an application also impacts these thresholds, thereby changing the optimal threshold. For these reasons, the static definition of such thresholds often leads to non-optimal performance.

3.3 Dynamic Policy for Parcel Coalescing: A Proof of Concept

This section shows a proof of concept for dynamic adaptation in the HPX runtime system for parcel coalescing.

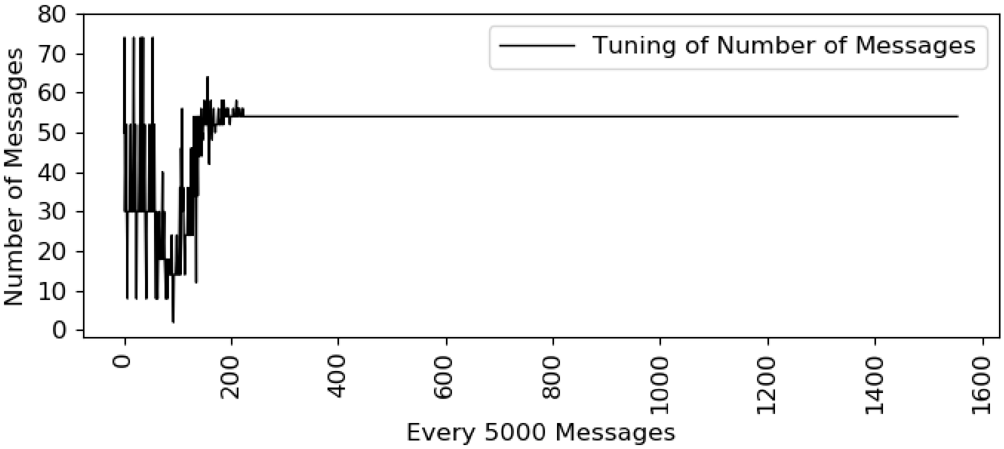
3.3.1 Motivation Behind Adaptive Policy. Adaptive policies are proven techniques used to tune parameters that depend on the architecture, the degree of parallelism, and the communication pattern exhibited by the application. Runtime adaptivity can be applied to runtime systems (such as HPX) or the framework (Phylanx). The motivation for using adaptive policies emerges from the limitations exhibited by the baseline policy. A fixed threshold might work for a given architecture and known application characteristics, but setting a generic threshold for all nodes is not practical for a truly heterogeneous system where different nodes are participating in a distributed environment. One way to solve this problem would be to determine the threshold at compile time. However, applications are often compiled on a login node and then executed on different nodes. Therefore, it is sensible to determine the thresholds at runtime instead.

3.3.2 What to Tune and Which Metrics Indicate Better Performance. As described in Section 3.2.4.1, there are two parameters in parcel coalescing: 1) the number of messages to coalesce, and 2) coalescing interval. The dynamic policy should find the values for these parameters, thereby reducing communication overhead. However, communication overhead should be related to the execution time. A previous study by Wagle et al. [128] showed a correlation between the network overhead counter in HPX and execution time (here, network overhead corresponds to communication overhead). Therefore, the selected parcel coalescing parameters should reduce the network overhead.

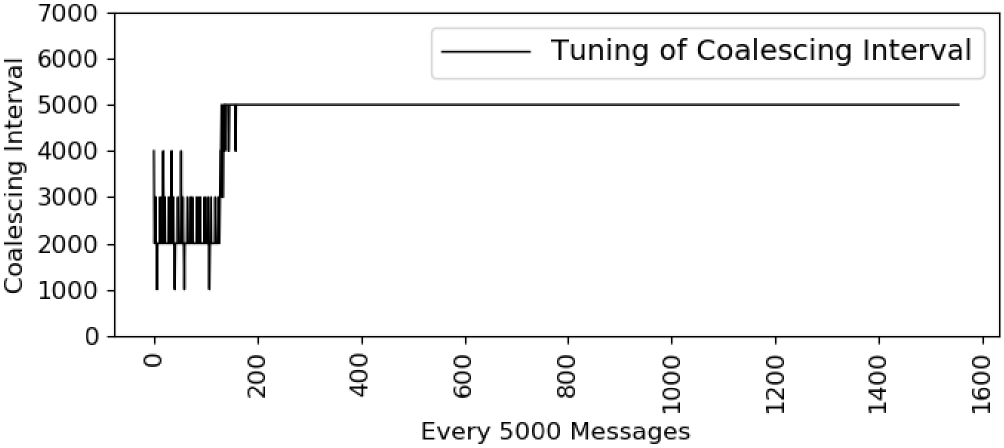
3.3.3 Granularity of the Adaptive Policy. For parcel coalescing, one policy instance is designed for a single node. When the application starts executing, the policy is triggered to determine the parcel coalescing parameters.

3.3.4 How the Adaptive Policy Works. The adaptive policy is implemented in the APEX policy engine. In the policy, a set of possible values for the parameters are assigned to each parameter. APEX policy invokes active harmony to use a different combination of the parcel coalescing parameters to reduce network overhead counter. When no more reduction is possible, the policy converges (more detail on how policy works is given in Section 3.4.3).

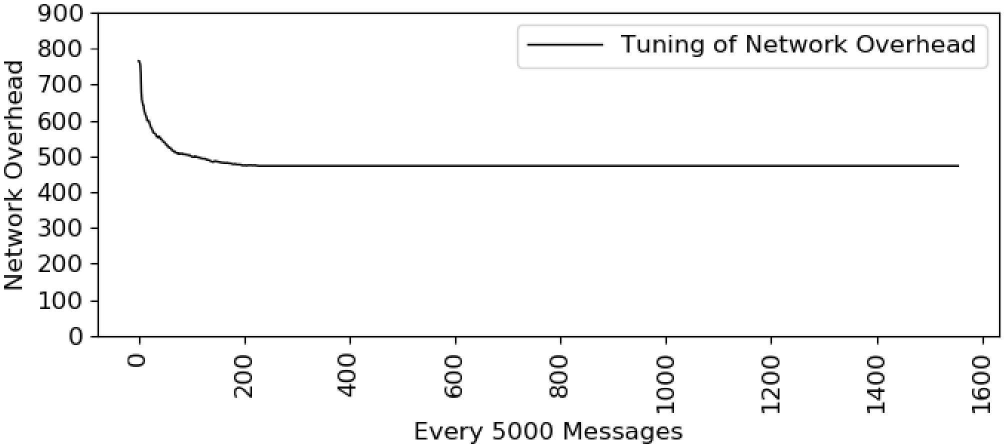
3.3.5 Results. Figure 10 demonstrates the results based on a ping-pong application [2] that sends millions of messages between two nodes. We have conducted these experiments on Talapas, an HPC cluster at the University of Oregon (each Talapas compute node has dual E5-2690v4 processors from Intel Broadwell microarchitecture). For this application, the parcel coalescing policy is triggered once every 5000 messages. The first two graphs show how optimal parameters are searched. With every change in the number of messages and intervals, the network overhead is sent to Active harmony to evaluate and change the parameters' values in a particular direction. By observing the overhead counter, the policy reaches a point where no more improvement is possible. The third graph shows how network overhead is reduced and reaches convergence.



(a) Tuning of number of messages.



(b) Tuning the coalescing interval.



(c) Reduction of network overhead.

Figure 10. Dynamic parcel coalescing policy determines the value of the parameters at runtime to reduce network overhead.

3.3.6 Discussion. The parcel coalescing policy demonstrates that APEX policies can find the runtime parameters that reduce network overhead, thereby providing a proof of concept of an adaptive approach. However, only one instance of a policy is used per node. In the next section, a more complex problem — task inlining is studied.

3.4 Dynamic Policy for Task Inlining

To decide which tasks we want to inline during execution, a dynamic policy is implemented in APEX. Unlike the baseline policy, this policy is fully automated, wherein the runtime system handles all the decision making.

3.4.1 What to Tune and Which Metrics Indicate Better Performance. In the baseline policy, the decision regarding task inlining is made based on a fixed threshold. However, the two thresholds (the upper and the lower) provide a gap that acts as a hysteresis so that the decision does not fluctuate when there is a small change in the execution time. Thus, tuning one threshold with a fixed hysteresis is logical and helps reduce the policy's search space and overhead. The threshold can be tuned based on the observed average execution time for the *primitive* instances for a pre-defined window.

3.4.2 Granularity of the Adaptive Policy. An important question arises about the granularity of the adaptive policy. Should there be one threshold for each type of *primitive* for the entire application or is a more granular control of the threshold desired? To answer this question, we look at the structure of the Phylanx framework. As described in the previous sections, Phylanx translates Python-like user code into an execution tree made up of Phylanx *primitives.* Each instance of each type of *primitive* exhibits different behavior as the inputs to each *primitive* may be different. For this reason, tuning a class of *primitives* to a common threshold does not make much sense. Rather, adapting the threshold for each instance of a *primitive* is expedient.

As mentioned in the description of the Alternating Least Squares benchmark in section 3.2.2.1, the Phylanx framework creates a different number of HPX tasks depending on the inputs to the algorithm. We define the threshold for each instance and tune the threshold based on its average execution time for an observed period to decide on a threshold that improves performance (reduction in execution time) for that instance only. There are two challenges that come with this approach. The first challenge stems from creating an APEX policy instance for each primitive that we need to tune. The second challenge is to manage the overhead associated with the creation of these policies. APEX itself addresses the first challenge as it can handle more than a thousand policies to tune parameters. For the second challenge, an overhead study is given in the experimental result section.

3.4.3 How the Adaptive Policy works.

The pseudocode of the adaptive APEX policy is shown in Algorithm 1. In the beginning, all *primitive* instances are set to use a default threshold and hysteresis value. It is important to note that a policy is not required for every *primitive* instance. A policy should only be created when the instance is executed many times (such as a *primitive* within a for loop). Only then does the policy have the opportunity to converge. A new policy must be registered to the APEX policy engine. We define a custom policy which is triggered when *eval* or *exec_count* is called more than *count_threshold_1* times (at least 5 times). During registration, several properties of the policy are defined, including the search space, the search strategy, and the tuning parameter. The search space defines the number of values needed to be explored to find an optimum value. The search strategy refers to the algorithms used to determine the optimum values. APEX uses search strategies provided by Active Harmony to find these values in a given search space. Active Harmony provides strategies such as EXHAUSTIVE, which tests every possible combination and determines the most efficient one, and PARALLEL_RANK_ORDER, which attempts to find suboptimal local minima in the search space. Each of these strategies works best when combined with an "appropriately sized" search space.

Finally, the tuning parameter is the metric that determines the effectiveness of different parameter settings. In our case, we use the average execution time of the *primitive* to tune our policy. Due to the overheads associated with measuring tuning parameters and executing policies, it is important to cease triggering the policy. We do this by monitoring the convergence of our policy parameters. Once we have determined that a policy has converged, it is de-registered and will no longer be triggered.

Algorithm 1 Adaptive APEX Policy for Task Inlining

```
for <For Every Primitive Instances in Parallel>
    $Threshold \Leftarrow 425000$
    $Hysteresis \Leftarrow 75000$
    if $exec\_count \geq count\_threshold\_1$
        $RegisterAPEXPolicy$
    if $exec\_count \geq count\_threshold\_2$  &&  $PolicyNotConverged$
        $LaunchAPEXPolicy$
        $SendCounterValuesToActiveHarmony$
        $RecieveNewThresholdFromAPEX$
        $ConfigureThresholdToHPX$
        $ResetCounter$
    if $exec\_count \geq count\_threshold\_1$
        $avg\_exec\_time \Leftarrow \frac{exec\_time}{exec\_count}$
        if $avg\_exec\_time \geq Threshold + Hysteresis$
            $inline\_task = false$
        else if $avg\_exec\_time \leq Threshold - Hysteresis$
            $inline\_task = true$
        else
            $inline\_task = undecided$
```

When a *primitive* instance reaches the defined number of executions, the custom policy is called, and APEX launches the policy. APEX reads the metric and a threshold from the search space and sends them to Active Harmony. Active Harmony stores the metric and the threshold and uses the defined search strategy to propose the next threshold to APEX. APEX sets this threshold in HPX and observes the impact of the decision at the next policy invocation. After the policy is successfully invoked, the counters are reset to start a new observation window. Every time the policy executes the search for optimal threshold progress. Based on the search strategy, when the impact of all (or a subset of) possible thresholds are tried, Active Harmony sends a signal to APEX about convergence, and the policy is de-registered after setting the optimal threshold in HPX. This threshold is used to make the task inlining decision for the rest of the application's execution.

After the policy is invoked or checked for convergence, the average execution time is calculated. Similar to the baseline policy, a new task is created if the average execution time is bigger than the threshold plus hysteresis. If the average execution is smaller, the work will be attached to the current task. The next section provides results from our investigation of measuring task scheduling and execution overheads.

3.5 Experimental Results for the Task Inlining Policy

In this section, we compare the APEX policy with the baseline policy on different CPU microarchitectures.

3.5.1 Experimental Testbed. For the experiments, we used the Marvin and Trillian nodes of the ROSTAM [138] cluster located at LSU running the 64 bit Centos GNU/Linux kernel version 3.10.0. The specifications for the nodes are listed in Table 9.

Table 9. Specifications of Nodes used.

Node	**Marvin**	**Trillian**
Microarchitecture	Sandy Bridge	Bulldozer
Processor Number	E5-2450	6272
Number of CPUs	2	4
Cores per CPU	8	16
Total Cores	16	64
Frequency	2.1GHz	2.1GHZ
Memory	48 GB	128GB

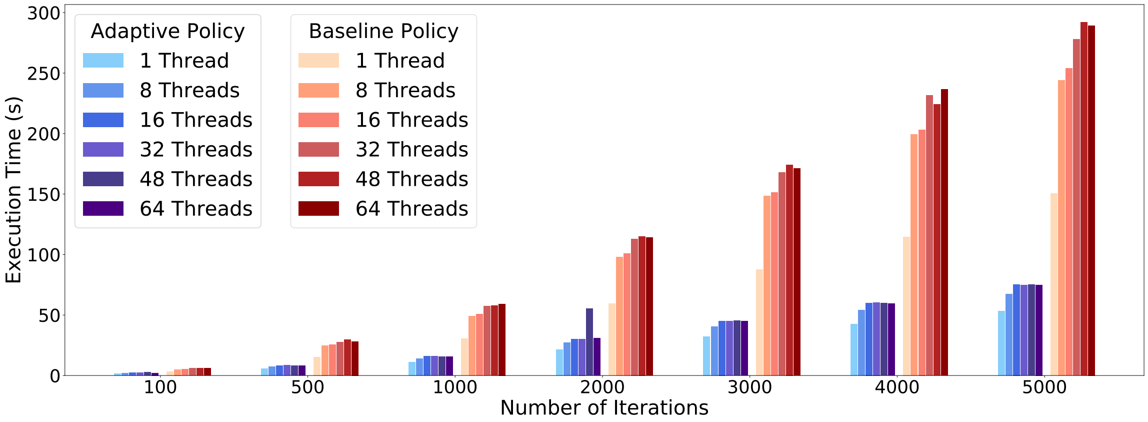


Figure 11. Comparison of the Adaptive APEX and Baseline policies on AMD Opteron (Node — Trillian) node for the Alternating Least Squares benchmark for an input size of 10 x 5 elements running for various iterations and number of threads.

3.5.2 AMD processors.

To observe the impact of the APEX policy, we ran the Alternating Least Squares benchmark on an AMD processor (Node — Trillian) with a different number of threads and a varying number of iterations. We ran each experiment five times, and then the results were averaged. At first, we ran the application using the baseline policy, and then we ran the same experiments with the APEX policy turned on. The result is depicted in Figure 11.

We observed a considerable improvement while using the APEX policy for each experiment. The APEX policy improved upon the baseline policy by up to 74%. For various numbers of threads, the APEX policy provides consistent behavior. The

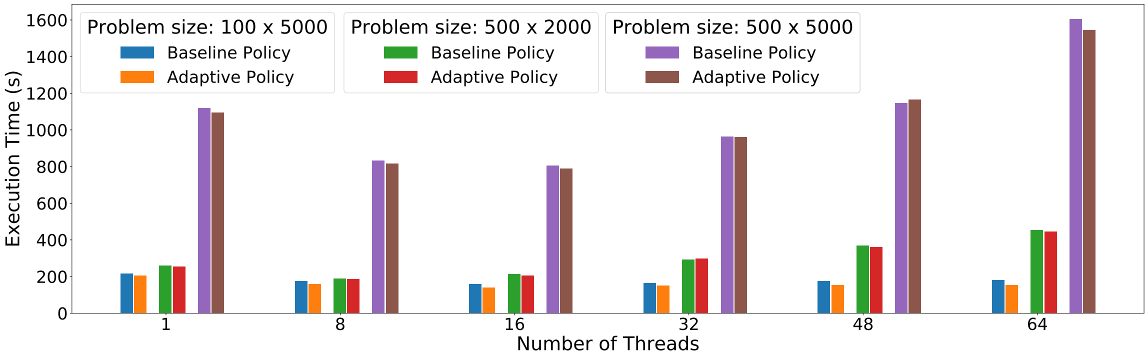


Figure 12. Comparison of the Adaptive APEX and Baseline policies on AMD node (Node — Trillian) for the Alternating Least Squares benchmark for an input size of 100 x 5000, 500 x 2000 and 500 x 5000 elements and various number of threads.

experiment demonstrates that the APEX policy can set the threshold to a value that reduces the tasking overhead by eliminating unnecessary task creation and scheduling. Additionally, our experiments do not exhibit any scaling. This observation tells us that there is not enough parallelism in the application itself and suggests running the policy with larger problem sizes. Nevertheless, the available parallelism of the application does not impact the APEX policy — it enables Phylanx users to run applications without worrying about the number of threads.

We use the Phylanx ALS algorithm with a bigger data set to create enough parallelism in the application. This application takes the data set from a CSV file instead of using a default dataset. We varied the data size (in rows and columns) for these experiments and the number of threads. The result is portrayed in Figure 12. We observe that the baseline policy does scale as we use more threads. This finding supports our previous conjecture that our previous experiments would benefit from more parallelism. However, even these larger data sets provide the algorithm with a limited amount of parallelism. After several threads have been added to the execution, the execution time increases. For example, if we select the data set with 500 rows and 5000 columns, the benchmark takes 1119 seconds to execute using the baseline policy. As we increase the number of threads to 8 and 16, we see the execution time decrease to 831 seconds and 805 seconds, respectively. However, when the number of threads goes higher than 16, we can see the execution time begins to increase.

We observe the benefit of the APEX policy from Figure 12. For 18 cases depicted in this figure, we can find improvement in 16 cases, while two cases show that the baseline policy performs better with a margin of 1%. However, the 16 instances where APEX policy shows improvement vary in a range of 0.4% - 16%. The average improvement for all the improving cases is 5% (standard deviation is 4%). So, we can expect 1% - 9% improvement for most cases. Even though large improvements in the application's execution time were not seen as opposed to the previous case with small data sizes, APEX policy still provides a considerable improvement margin for an application with a large data set and enough parallelism.

3.5.3 Intel Xeon processors. We repeated these experiments on an Intel Xeon processor. We ran our experiments on the Marvin nodes of the ROSTAM [138] cluster. The results are depicted in Figure 13. A similar trend as Figure 11 is found in this experiment. However, there is not much improvement visible from the APEX policy. We observe almost identical execution times with the baseline and APEX policy. Out of 20 experiments, we found improvement in 7 cases and performance degradation for the remaining cases. The average improvement for the 7 cases is 1.8%, while for the 13 cases where the baseline policy performed better, the average stands at 2%. On average, there is no improvement found from using the APEX policy.

The APEX policy determines the optimal threshold for each primitive instance, and the baseline policy defines the static threshold. If the baseline policy defined threshold is already optimal or close to the optimal, we will not find a visible difference between the baseline and APEX policy. The baseline policy sets the upper and the

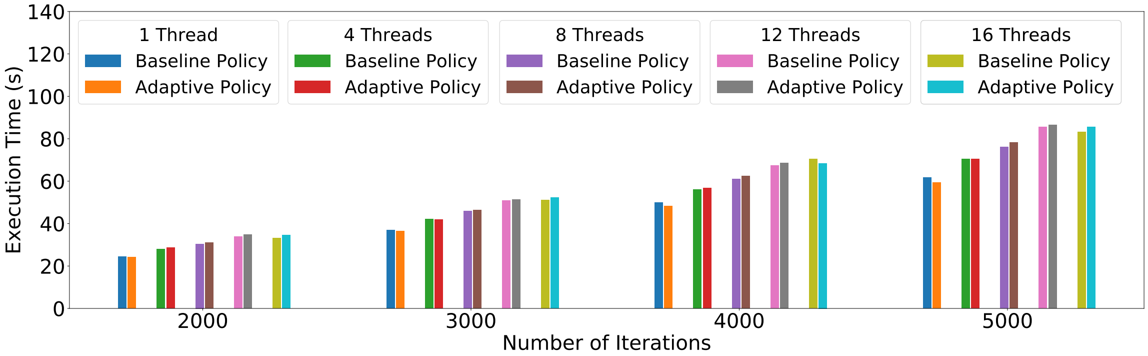


Figure 13. Comparison of the Adaptive APEX and Baseline policies on Intel Xeon node (Node — Marvin) for the Alternating Least Squares benchmark for an input size of 10 x 5 elements running for 2000, 3000, 4000 and 5000 iterations.

lower threshold to 350 and 500 μs, respectively. We find that as long as the lower threshold is more than 300 μs almost all the threshold configurations provide similar results. Furthermore, the lower threshold of the baseline policy is 350 μs the baseline policy is providing a close to the optimal result. For this reason, we do not observe any visible improvement from the APEX policy. Moreover, APEX policy contributes a small overhead which negates the small improvement observed.

3.5.4 Overhead of the APEX policy. Theoretically, the overhead of the APEX policy is constant for an algorithm and does not change with the data size. Every algorithm has a fixed number of primitive instances that call an APEX policy a fixed number of times. We have compared 100 runs of the Alternating Least Squares benchmark, each using different data sizes to measure the overhead. We found that the APEX policy introduces a total of 5 seconds, on average, to the execution time for this algorithm. For a larger data set, where the Alternating Least Squares benchmark takes 30 minutes to execute, 5 seconds is a negligible amount of overhead.

3.6 Related Work for Task Inlining

A lazy task creation strategy was proposed in [127] where task creation was avoided until processing resources were free. With regards to OpenMP tasks, Duran [139] proposed a cutoff technique to improve performance. The cutoff was based on the max number of tasks in the system or max recursion level. ATC (Adaptive Task Cutoff) was proposed in [140], where the cutoff decision was based on the profiling data obtained from the application at runtime and assumed that all tasks at a given level have similar behavior. Adaptive Task granularity(ATG) was proposed in [141] for irregular task-parallel programs. ATG switches between help first and serialization policy depending upon the number of tasks created in the system. However, the study did not consider the effects of processor architectures. With regards to compiler-based approaches, a multi-versioning approach was proposed in [142], where a combination of compiler and runtime approaches were used. Here, multiple versions of tasks with varying granularity were generated at compile-time, and one was then selected at runtime by tracking task demand. A compiler-based static cutoff along with two optimizations, namely code-bloat-free inlining, and loopification was proposed in [143]. An auto-tuning framework for divide and conquer task-parallel programs was proposed in [144] which was implemented as an optimization pass in LLVM. In the context of asynchronous multitasking runtime systems, Sun [145] developed the ParSSSE (Parallel State Space Search Engine) Framework for Charm++ and looked at adaptive grain size control in the context of parallel state search methods. Grubel [146] used performance counters in HPX for dynamically tuning grain size of 1d-stencil application. Our proposed method is application-agnostic, and no change in application source code is required. Furthermore, our proposed method relies on the actual execution time of the tasks to make decisions regarding task inlining for future executions of the tasks.

3.7 Summary

This chapter investigates adaptive techniques for task inlining and parcel coalescing in the HPX runtime system by using APEX. The experimental results suggest that dynamic adaptive policies are capable of showing significantly improved performance. The method outlined in this chapter only considered CPU architectures. Moreover, adaptive policies are designed based on the response of the runtime system without considering the interaction of the application and the platform. Since dynamic adaptive policies are influenced by different CPU architectures, it is desired to enable runtime decisions that consider application and platform factors. More recently this diversity in computing architectures is being found in heterogeneous systems, forcing us to consider the problem of dynamic adaptation in such an environment. For this reason, the next chapter explores the energy-performance trade-off in a heterogeneous environment where dynamic decisions are being made at the level of individual kernels assigned to different processing units.

CHAPTER IV

MEPHESTO: MODELING ENERGY-PERFORMANCE IN HETEROGENEOUS SOCS AND THEIR TRADE-OFFS

This chapter contains previously published material with co-authorship. All of the presented research in this chapter was conducted as a collaboration between the University of Oregon and Oak Ridge National Laboratory (ORNL). The research included in this chapter was presented at PACT 2020 [12]. While working on MEPHESTO, I received regular guidance from Dr. Mehment Belviranli during my internship at ORNL. Dr. Seyong Lee also provided advice. I received high-level guidance from Dr. Allen Malony and Dr. Jeffrey Vetter. I did all the experiments, writing, and data collection. Dr. Belviranly helped by thoroughly proofreading and added his words when necessary. Other co-authors also helped with proofreading.

4.1 Introduction

This chapter investigates the dynamic kernel placement problem in a heterogeneous system (corresponding to Research Question 3 — **RQ3**: Can we model memory contention in a heterogeneous system to design an energy-performance aware scheduling algorithm?). As pointed out in Chapter II, heterogeneous systems are now the go-to solution for overcoming the temperature barrier when designing processing units (PUs) with high computation capabilities [147, 148]. Following this trend of heterogeneity, special-purpose hardware for emerging domains—such as tensor PUs, bionic processors, and vision accelerators—have become a commodity in data centers, mobile devices, and autonomous platforms. Moreover, chip manufacturers are embedding a variety of PUs that serve different types of computing needs on a single die in the form of integrated shared memory heterogeneous systems (iSMHS) [112]. Although Intel's Ivy Bridge [149] and AMD's Fusion [150, 151] architectures were among the early systems that combined two compute capable PUs (i.e., CPUs and GPUs) under the same memory subsystem, later generations of integrated heterogeneous systems—such as NVIDIA's Tegra Xavier, Apple's A12 Bionic chip, and Qualcomm's Snapdragon 855 system on chip (SoC)—have brought the degree of heterogeneity within the same chip to a new extreme. In such systems, dozens of PUs with diverse instruction set architectures work together to accelerate kernels that belong to emerging application domains.

One of the most prominent features of iSMHS is that all PUs can directly access the system memory, alleviating additional data transfer costs between the CPU and device memory [152]. The optimal use of these systems heavily relies on collocating tasks simultaneously in different PUs while using the system memory as an intermediate medium for inter-PU data communication. For example, an iSMHS for an autonomous car must simultaneously run image and video processing, inference for object detection, and other decision-making continuously in a pipeline-like execution scheme.

Collocated kernel execution on an iSMHS will likely cause contention on the shared memory bus, and the resulting interference could negatively affect perceived bandwidth (BW) on collocated kernels. Several studies [152, 112, 153, 154] focused on identifying the memory access patterns of collocated kernels in CPU+GPU iSMHS and suggested smart scheduling mechanisms to minimize contention effects, mostly via ad hoc approaches. However, these approaches do not provide a systematic solution for systems with different heterogeneity characteristics and an arbitrary number of PUs.

Apart from performance, memory contention also has a considerable impact on chip-level power consumption. Several studies [155, 109, 156, 111, 147, 104, 157] focused on iSMHS energy characteristics and relied on greedy algorithms or machine learning-based approaches to control the power consumption via dynamic voltage and frequency scaling (DVFS) and thermal design power (TDP) based power caps. However, the impact of contention on energy usage of PU and the chip is not addressed, and these studies do not build an analytical approach that establishes a direct relationship between energy and memory contention.

The ubiquitous deployment of iSMHS in environments in which the use-case priorities can dramatically vary makes the kernel collocation problem more challenging. Since the objectives are dynamic and constrained by the throughput and power budget needs, meeting them is crucial for optimizing overall utilization of iSMHS. For example, in an autonomous driving scenario, the system software should collocate kernels in the most performance-maximized manner when approaching an intersection with multiple objects to track. On the other hand, while cruising on a highway at a stable speed for which processing throughput needs are lower, kernels can be scheduled to minimize total energy usage. Ideally, such an objective-aware kernel collocation could be achieved with a simple parameter that controls the energy-performance trade-off (EPTO).

This chapter presents MEPHESTO, which proposes a holistic approach for controlling the EPTO in the collocated kernel execution on iSMHS. A per-PU kernel operational intensity [158, 159] was used to approximate the effects of contention on performance and energy along with the kernel collocation algorithm to intelligently find near-optimal collocation that satisfies the provided EPTO objective. We believe that this is the first effort to propose a generic formulation for memory contention in iSMHS that considers factors that directly correlate to energy and performance during kernel collocation.

The chapter makes the following contributions:

- Integrated performance and energy behavior representation are introduced based on time factors and power factors, which are nonlinear functions of the ratios between standalone and collocated execution measurements of a PU in a given iSMHS.
- A novel empirical model was built to estimate the energy and performance of a set of collocated kernels on an arbitrary number of PUs while considering the variation caused by memory contention.
- A collocation algorithm was designed that takes the target EPTO as a user-defined input parameter and employs a novel heuristic to reach a near-optimal ordering and placement (O&P) of a given set of kernels on a target set of PUs.
- The feasibility of MEPHESTO was empirically evaluated by collocating a collection of scientific kernels across three heterogeneous PUs of NVIDIA's Tegra Xavier platform: CPU, GPU, and programmable vision accelerator (PVA). The proposed scheduling algorithm was demonstrated to be able to find near-optimal O&P with a reasonable (on an average 10%) modeling error rate. It also provides up to 30% improvement over a greedy approach.

4.2 Understanding the Effects of Collocated Execution on iSMHS

This section explores the energy and performance implications of collocated kernel execution on an iSMHS similar to that portrayed in Figure 14. In this example, the CPU and GPU are connected to a shared memory, and the GPU does not have a private memory. There could also be other heterogeneous PUs, such as PVAs and deep learning accelerators (DLAs), connected to the same memory subsystem; however, they are excluded in this specific case for simplicity. In this system, the collocation of five ready-to-execute kernels with the O&P configuration of *123|45*, in

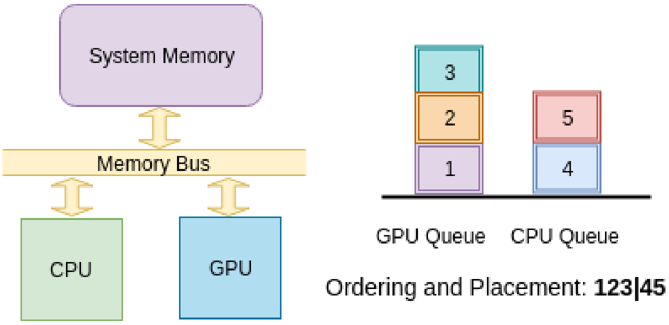


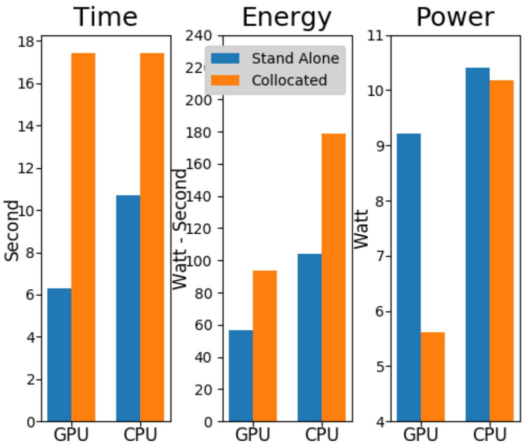
Figure 14. A logical representation of iSMHS with CPU and GPU.
A logical representation of iSMHS with CPU and GPU and kernel queues for O&P: *123|45*.

which kernels 1, 2, and 3 will be orderly executed on the GPU and kernels 4 and 5 will be placed for CPU execution. Kernels placed on different queues could execute in a collocated manner and result in contention on the memory bus.

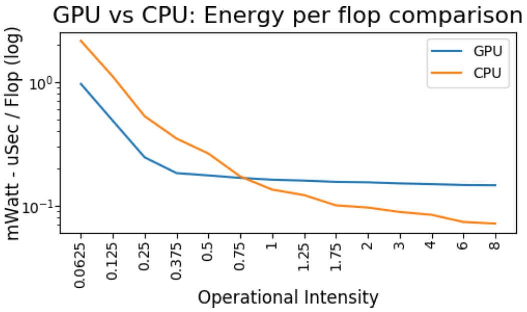
4.2.1 Contention vs. Energy and Performance. To observe the effects of the collocated execution of two maximum memory BW-demanding kernels on performance and energy, we ran the STREAM benchmark [160] on the ARM Carmel CPU and Volta GPU of NVIDIA's Xavier platform concurrently. The amount of data that the CPU and GPU were able to process during execution was recorded,

Table 10. Roofline kernels used to understand the outcomes of different O&P variations.

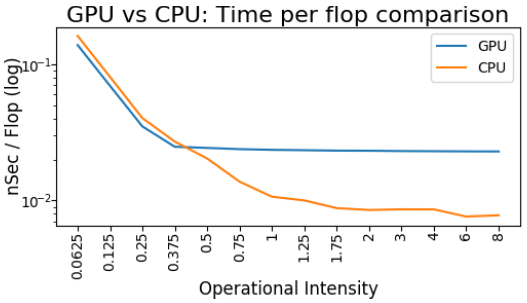
Kernel name	**1**	**2**	**3**	**4**	**5**
Flop per array index	1	6	12	20	48
DRAM R/W byte	16	16	16	16	16
Operational intensity	0.0625	0.375	0.75	1.25	3.0
Total flop	273.8 G	273.8 G	273.8 G	273.8 G	273.8 G
GPU Flop/s	7.2 G	40.3 G	41. 9G	42.7 G	43.3 G
GPU Avg. power (Watt)	6.9	7.3	7.0	6.8	6.5
CPU Flop/s	6.2 G	36.9 G	72.6 G	99.8 G	115.9 G
CPU Avg. power (Watt)	13.3	12.8	12.4	12.1	10.3



(a) Effect of memory contention on time, energy, and power for collocated/standalone execution of STREAM benchmark on CPU and GPU.



(b) Per-flop energy consumption when kernels with different operational intensities are run in a standalone mode.



(c) Per-flop execution time when kernels with different operational intensities are run in a standalone mode.

Figure 15. Energy and performance behavior under different scenarios in an iSMHS.

and the total execution time and power consumed by the CPU and GPU were measured at 50 ms intervals. The runs are repeated for the same amount of data in standalone mode, and the execution time, total energy consumption, and average power consumption are reported in Figure 15a. Significant increases in execution time and energy consumption were observed for both kernels while running simultaneously on the CPU and GPU. On the other hand, average power consumption decreased for collocation, particularly from the GPU perspective. Although the we ensured that CPU and GPU utilization was 100% all the time and that the system did not throttle CPU, GPU, or memory frequencies, the BW utilization dropped to almost half of the standalone value during collocated execution. This observation resulted in the realization that the impact of contention on time and energy is different for CPUs and GPUs and further motivated the establishment of a PU-centric contention model for energy and performance (i.e., execution time).

4.2.2 Need for Characterizing Kernels and PUs. Since contention depends on the amount of memory access requests generated by the kernel, the kernels running on each PU must be characterized in terms of their memory access requests to identify the level of contention they might incur. Also, contention only occurs when the processor cannot find the requested data in the cache and data must be brought from the system memory. For this reason, this work identified kernels based on their frequency of system memory accesses by using an operational intensity metric, which is the ratio of the total number of flops and total bytes read/write (R/W) between last-level cache (LLC) and system memory, as defined by the Roofline model [158]. The lesser the operational intensity, the more likely it is to have contention. Moreover, contention also depends on the physical BW of the system and the amount of cache attached to the processors. For this reason, PU behavior must also be characterized by cross-running different kernels with different operational intensities in both PUs simultaneously to observe the level of contention.

4.2.2.1 Justification for Using Operational Intensity.

Operational intensity depends on traffic between LLC and memory, which represents the common contention point for different PUs in iSMHS. Other factors affect the performance of collocation, such as the memory access patterns of the kernels and caching optimizations, but operational intensity can indirectly include those effects since traffic between LLC and memory reflects those effects. In this way, operational intensity provides a simplified and unified metric that presents a fair trade-off between modeling complexity and prediction accuracy.

4.2.3 Ordering and Placement.

Figure 15a shows that collocating kernels can significantly impact energy and performance. As a result, kernel ordering is important for energy and performance. The example given in Figure 14 implies that kernel 1 will be collocated with kernel 4 and that kernel 2 will be collocated with kernel 5. This work strives to determine the best O&P for a given set of kernels. One systematic way to approach this problem is to start by running synthetic kernels standalone on the CPU and GPU with different operational intensities to observe how much time and energy is spent for each flop. The Empirical Roofline Toolkit [161] was modified to produce various intensities, and the energy and performance behaviors are shown in Figs. 15b and 15c, respectively. The results show that with greater operational intensity, the CPU becomes more favorable than the GPU since it spends less energy and time per flop.

Another observation at the intersection point of the CPU and GPU curve is that if the operational intensity that identifies a specific kernel falls to the left of the intersection, it is more suited for GPU, and vice versa. Thus, if the operational intensity of a kernel being run is known, an accurate placement decision can be made as to where the kernel should run. However, the intersection points might be different for energy and performance behavior, making the placement decision more complicated. More importantly, Figs. 15b and 15c do not consider collocation or contention. Therefore, a combined solution is needed that will incorporate the outcomes of Figs. 15a, 15b, and 15c together. The confluence of O&P for a given set of kernels must be considered.

4.2.4 Kernel Collocation for Varying Energy and Performance. To demonstrate how various O&P configurations affect energy consumption and performance differently, the Empirical Roofline Toolkit was used to generate five kernels, whose details are given in Table 10, with varying operational intensity ratios. These kernels operate in a read-compute-write fashion. The first 8 bytes of data (i.e., double-precision floating point) are read from an array, a series of additions and multiplications is performed, and the data are written back to the same memory location. In this way, data are ensured to be fully read from and written back to the DRAM. The second and third columns of Table 10 show the number of floating-point operations and the total amount of bytes R/W from DRAM for every array index the kernels process, respectively. Based on these values, the operational intensities per kernel were calculated. Through profiling, flops per second and the average power data for each kernel in the standalone mode were generated for CPU and GPU.

Four different O&P configurations were used, and the execution time and energy consumption are shown in Figure 16. The results show that different O&P configurations lead to different time/energy profiles. Although the common strategy in the related literature is to collocate compute-intensive kernels with memory-intensive kernels to improve overall performance, the optimal O&P strategy might

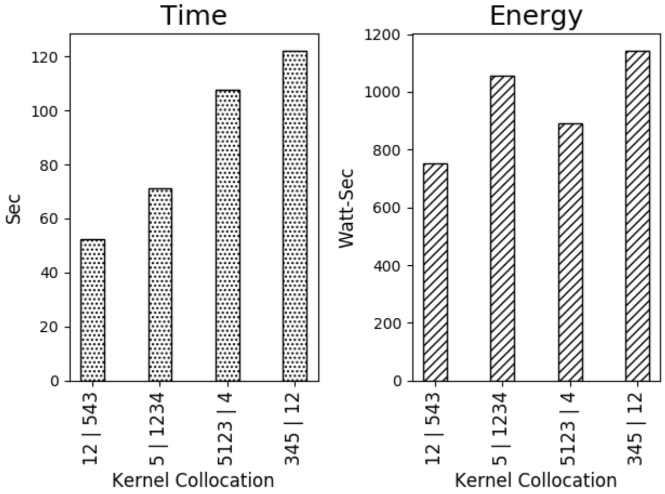


Figure 16. Different total energy and performance behaviors for various collocation combinations for the same kernels.

vary when a trade-off is being sought between energy and performance. This situation raises a few questions: How can the cut-off point for the compute-memory intensity be defined? What happens if the number of memory-intensive kernels is more than the compute-intensive kernels, or vice versa? How can trade-off control be established between energy and performance? As shown in the following sections, greedy algorithms commonly employed by the related studies are not sufficient to address all these considerations.

This research revolves around addressing the four motivations presented in this section. The remainder of this chapter presents an empirical model for defining memory contention by considering the kernel and processors, defines optimal O&P, and presents the kernel collocation algorithm for obtaining a desired EPTO.

4.3 An Empirical Model for Memory Contention

This section presents a core component of MEPHESTO, an empirical model that characterizes kernels and PUs with respect to memory contention. The impact of memory contention is defined in terms of energy consumption and performance (i.e., execution time). For a given O&P of kernels, this model predicts the execution time and energy consumption.

Table 11. The notations used by the model.

Notation	**Explanation**
K_i	ith kernel from n kernels $K = \{K_1, K_2, ..., K_n\}$
P_j	jth PU from m PUs $P = \{P_1, P_2, ..., P_m\}$
F_{K_i}	Number of flops in kernel K_i
$T_{P_j}^{K_i}$	Standalone execution time of K_i on PU P_j
$Pw_{P_j}^{K_i}$	Standalone power of PU P_j for K_i
OI_{P_j}	Operational intensity of the kernel on P_j
TF_{P_j}	Time factor of P_j
PF_{P_j}	Power factor of P_j
$TC_{P_j}^{K_i}$	Collocated execution time of K_i on PU P_j
$E_{P_j}^{K_i}$	Energy consumption of K_i on PU P_j
C	Total number of possible O&P
ρ	PU wise queues (i.e., O&P)
τ_c	Execution time of current O&P
ε_c	Energy consumption of current O&P
$S(V)$	Resultant kernel collocation of V kernels
ω	Weight for a given O&P

4.3.1 Definitions.

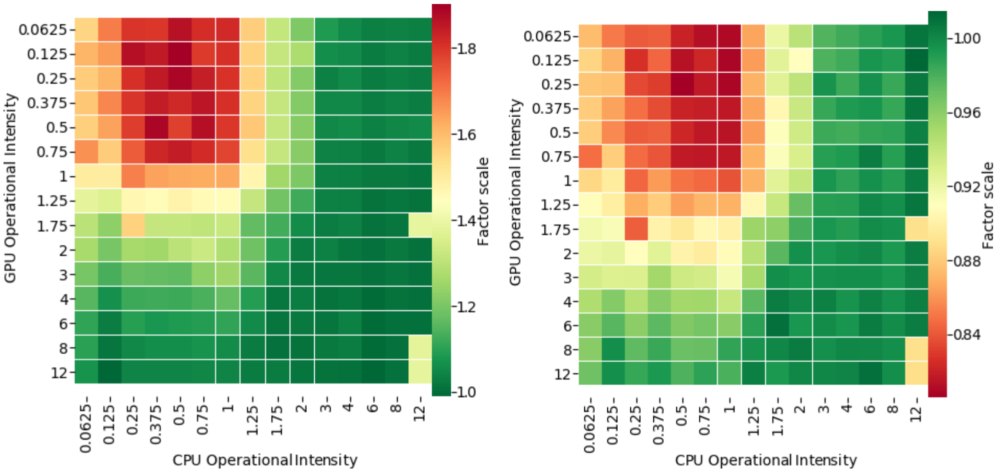
All symbols and terms used in this model are introduced in this section and are also presented in Table 11. An *O&P* is a set of n kernels $K = \{K_1, K_2, ..., K_n\}$ in which a kernel is an uninterrupted computation that is ready to be executed on any of the available PUs in the system. The *kernel collocator* finds the placement of every K_i on a set of m heterogeneous PUs, which is represented by $P = \{P_1, P_2, ..., P_m\}$.

$$K_i = [F_{K_i}, OI_{K_i}, \forall_{P_j \epsilon P}\{T_{P_j}^{K_i}, Pw_{P_j}^{K_i}\}]. \tag{4.1}$$

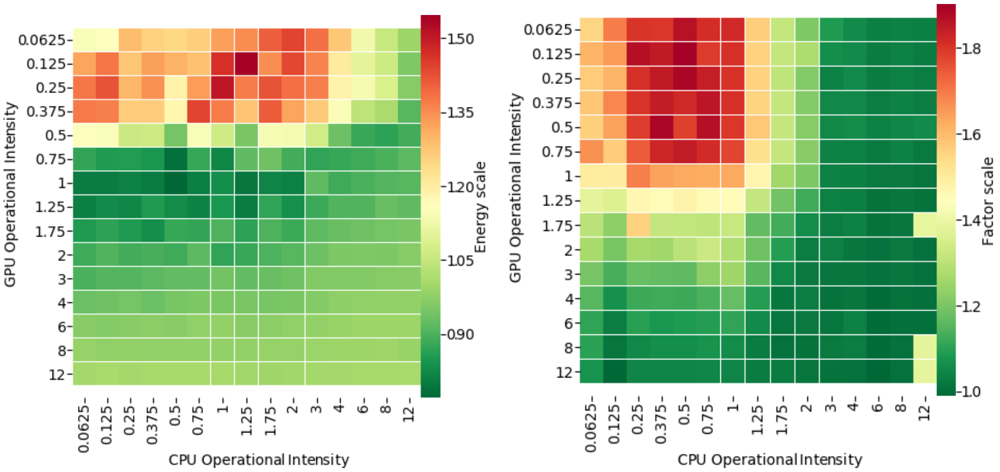
A kernel K_i is represented by using three terms, as shown in Eq. (4.1). The first term, F_{K_i}, is the number of floating point operations of that kernel. The second term, OI_{K_i}, is the operational intensity. When a kernel, K_i, is placed on a processor, P_j, the standalone execution time and average power consumption are represented by $T_{P_j}^{K_i}$ and $Pw_{P_j}^{K_i}$, respectively. So, the third term represents the pair of standalone execution time and average power consumption for each PU. The objective is to determine these values at compile time and also by partially profiling the kernels at run time. However, when collocated, each K_i exhibits different slowdown in execution time and consumes different average power based on their compute and memory intensity. As suggested previously, in Figure 15, there is a factor for execution time, TF_{P_j}, and a factor for average power, PF_{P_j}, for a given PU P_j. These factors are the ratio of their collocated to standalone values. Thus, PU is represented as $P_j = [TF_{P_j}, PF_{P_j}]$.

4.3.2 Characterization of Memory Contention.

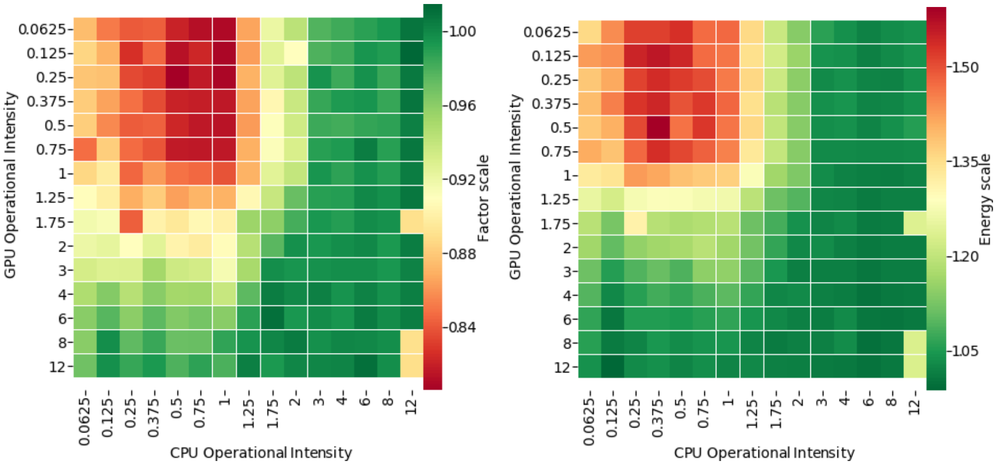
To define and characterize memory contention, three factors must be determined: (1) how kernels, K, are characterized; (2) how PUs, P, are characterized, and (3) how time factor, TF_{P_j}, and power factor, PF_{P_j}, are formulated. To determine the first factor, operational intensity must be considered as a measure to characterize a kernel's compute or memory intensity. The Empirical Roofline Toolkit [158, 161] was modified to generate kernels with different operational intensities, and their execution was observed. Roofline kernels are designed in a read-compute-write fashion. For this reason, a fixed number of bytes are exchanged between the cache and system memory; as a result, the operational intensities of those kernels do not vary across different PUs with different cache hierarchy. To determine the second factor, a range of kernels was



(a) GPU time factor.
(b) GPU power factor.



(c) GPU energy factor.
(d) CPU time factor.



(e) CPU power factor.
(f) CPU energy factor.

Figure 17. Collocation factors: collocated/standalone values of time, power, and energy for different operational intensities.

collocated with different operational intensities in PUs of iSMHS. A kernel was kept running in one processor, and the impact on another was observed. While doing this, execution time and power consumption were recorded. From these values, energy consumption was calculated. To capture the impact in a structured way, a ratio of collocated to standalone time and power was made, which are termed *time factor* and *power factor*, respectively. This addresses the third concern.

This PU characterization is a one-time effort for any new system. In this case, we were interested in NVIDIA Xavier's CPU and GPU. Figure 17 plots the results. In these heat maps, the x-axis is the operational intensity of kernels that are running on the CPU, and the y-axis is for the GPU. Time and power are captured for both the CPU and GPU, from which energy consumption was calculated. When operational intensity is low (i.e., high memory intensity), the impact of contention is visible in both the CPU and GPU. The time factor goes up to $2.2x$ for the GPU (Figure 17a) and $1.8x$ for the CPU when contention is present (Figure 17d). Similar behavior is observed for energy consumption (Figure 17c). The opposite is observed for the power factor; $0.6x$ is observed for the GPU (Figure 17b), and $0.8x$ is observed for the CPU (Figure 17b). Interestingly, after a certain operational intensity occurs, the impact of contention vanishes because the available system BW is enough for the request, and there is no bus contention.

After capturing the impact of different kernels, the time factor, TF_{P_j}, and power factor, PF_{P_j}, can be represented as a function of the operational intensity of the kernel of the current PU, OI_{P_j}, and collocated PU, $OI_{P_{col}}$, as in Eq. (4.2). A fifth order multivariate polynomial regression curve was generated by using the CxxPolyFit [162] tool that supports up to the ninth order for TF_{P_j}, and PF_{P_j}, in which the independent variables are OI_{P_j} and $OI_{P_{col}}$. Using a lower order provides faster evaluation with low accuracy, whereas higher order (e.g., fifth order or higher) provides more accuracy but takes more time to evaluate. Fifth order was used due to a reasonable balance between the evaluation time and accuracy, as recommended by CxxPolyFit [162].

$$TF_{P_j} or PF_{P_j} = f(OI_{P_j}, OI_{P_{col}}) \tag{4.2}$$

4.3.3 Collocation Estimator Algorithm. The objective of the collocation estimator algorithm is to take an O&P of variable length kernels and predict the execution time and energy consumption while considering memory contention. To calculate the total execution time, τ_c, for the current O&P, c, the collocated execution time must be estimated from time factor and standalone execution time. For the total energy consumption, ε_c, collocated average power must be estimated from collocated power factor and standalone average power. Moreover, the length (i.e., the time span of execution) of a kernel must be considered since one kernel can be collocated with multiple shorter kernels during its lifetime.

Algorithm 2 Collocation Estimator

```
Input: PU wise kernel queue $\rho = \{\rho_1, \rho_2 ... \rho_m\}$
Output: Execution time, $\tau_c$, and energy consumption, $\varepsilon_c$
Initialize: $\tau_c \leftarrow 0 \& \varepsilon_c \leftarrow 0$
while $\exists_{\rho_j \epsilon \rho} Size(\rho_j) > 0$
    for each $\rho_j \epsilon \rho$ where $Size(\rho_j) > 0$
        $K_{P_j} = \rho_j.POP()$
    for each $K_{P_j} \neq NULL$
        $TF_{P_j} = time\_factor(OI_{P_j}, OI_{P_{col}})$
        $PF_{P_j} = power\_factor(OI_{P_j}, OI_{P_{col}})$
        $TC_{P_j} = T^{K_i}_{P_j} * TF_{P_j}$
    $P_{min} = \min_{\forall P_j} [TC_{P_j}]$
    $\tau_c$ += $TC_{P_{min}}$
    $\varepsilon_c$ += $\Sigma_{\forall P_j} [Pw^{K_i}_{P_j} * PF_{P_j} * TC_{P_{min}}]$
    for each $TC_{P_j} > TC_{min}$
        $TCremaining_{P_j} = [TC_{P_j} - TC_{min}] / TF_{P_j}$
        $\rho_j.PUSH(TCremaining_{P_j})$
```

The collocation estimator algorithm given in Algorithm 2 was designed by considering the aforementioned objectives. This algorithm takes an O&P of kernels, c, scheduled on PUs, P. Every PU has its own queue $\rho = \{\rho_1, \rho_2, ..., \rho_m\}$ in which kernels are stored in such an order so that the kernel in the head of the queue will be scheduled first to the corresponding PU. The algorithm determines the overall execution time, τ_c, and total energy, ε_c, incrementally. In the beginning, at [Line 3], τ_c, and ε_c are initialized. Then, at [Line 4], a loop is started, which continues until all queues are empty or all kernels are scheduled. In [Lines 5–7], the queue item at the head is popped from each ρ_j and stored as K_{P_j}. At [Lines 9–10], the time factor, TF_{P_j}, and power factor, PF_{P_j}, are calculated by using Eq. (4.2) for all nonempty K_{P_j}. Collocated time, TC_{P_j}, is then calculated by multiplying the standalone execution time and collocation time factor at [Line 11]. At [Line 13], the processor with the smallest kernel, P_{min}, is determined, and at [Line 14], the minimum time is added to the total time, τ_c. The minimum execution time was taken because the other kernels in other processors will now have a different kernel as collocated since the minimum one has finished its execution. At [Line 15], energy is calculated by considering the minimum time and collocated average power. Collocated average power is determined by multiplying standalone average power, $Pw_{P_j}^{K_i}$, with the power factor, PF_{P_j}. Since one kernel has finished its execution, the remaining part of the longer kernels must be calculated. For this reason, at [Lines 16–18], the remaining part of collocated time, TC_{P_j}, is factored back to standalone time, $TCremaining_{P_j}$, and pushed to the corresponding queue, ρ_j. These leftover kernels are considered to be just like a new kernel in the next iterations, and this occurs until every queue is empty. The total execution time, τ_c, and energy, ε_c, are determined when every queue is empty. Algorithm 2 finds τ_c and ε_c in *O(nm)* time, where n is the number kernels, and m is the number of processors.

4.4 Defining Optimal Ordering and Placement

This section discusses the cost function design to define an optimal O&P based on a given trade-off target. For this reason, all possible combinations of O&P must be considered for all kernels, $K = \{K_1, K_2, ..., K_n\}$ in all PUs, $P = \{P_1, P_2, ..., P_m\}$. Let C represent all the possible ways that n kernels can be ordered and placed on m processors. The execution time of all possible O&P is denoted as $\tau = \{\tau_1, \tau_2, ..., \tau_c\}$, and energy is denoted as $\varepsilon = \{\varepsilon_1, \varepsilon_2, ..., \varepsilon_c\}$, where minimum and maximum execution times are represented by τ_{min} and τ_{max}, respectively. The minimum and the maximum energy consumption for all O&P are represented by ε_{min} and ε_{max}. For example, for five jobs in two processors, there are 482 possible O&P and thus 482 pairs of energy and time. Since there are two parameters—energy and performance—a reference point is needed to define the optimal O&P. This reference point is called the *EPTO parameter*.

The EPTO parameter is represented as a pair of energy performance in the following format—*(performance, energy)*—where the value of performance or energy can be 0–100. If EPTO is set to (0,100), then minimizing execution time is given the highest priority. If EPTO is set to (100,0), then minimizing energy consumption is given the highest priority. If EPTO is set to (30,70), then 70% priority is given to minimize execution time and 30% priority is given to minimize energy consumption.

To achieve this functionality, energy and time pairs of every O&P were converted to a range from 0 to 100. Then, every O&P becomes a point at which energy and time can vary from 0 to 100, where 0 represents the minimum time or energy and 100 represents the maximum. In this way, the energy-time pair of all O&P can be plotted in a 100 × 100 plot in which EPTO also becomes another point. Then, the distance from EPTO to every O&P is measured. The lower the distance, the higher the weight assigned, and at the end, the O&P with the highest weight is selected. This is achieved by a cost function expressed in Eq. (4.3). Based on the value of EPTO, τ_c, and ε_c, the weight of every O&P is calculated. A set of C weight is expressed as $\omega = \{\omega_1, \omega_2, ..., \omega_c\}$. The O&P of kernels, $S(J)$, were selected where the weight is the highest, and this is the optimal O&P for the kernels and defined EPTO.

$$\omega_c = 1/dist\left[\left\{\frac{\tau_c - \tau_{min}}{\tau_{max} - \tau_{min}} * 100, \frac{\varepsilon_c - \varepsilon_{min}}{\varepsilon_{max} - \varepsilon_{min}} * 100\right\}, \{EPTO\}\right]. \tag{4.3}$$

4.5 Kernel Collocation Strategy

This section formulates a heuristics based on dynamic programming (DP) for MEPHESTO. By using the DP-based heuristics, a kernel collocation algorithm is designed, followed by a discussion of the complexity analysis of the approach.

4.5.1 Dynamic Programming Formulation. An exhaustive search throughout all the kernel O&Ps guarantees an optimal solution but is computationally expensive. For this reason, a DP approach was formulated to reach a near-optimal solution and reduce the complexity [163]. The solution is built from a smaller set and recursively builds the bigger ones by selecting maximum weighted subsolutions. At each step, a new placement for one kernel, K_j, is found, which maximizes the weight for the current O&P. V is considered a varying set of kernels, where $V \subseteq K$. Kernel O&P is represented as *S(V)*, which provides the processor wise queue information $\rho = \{\rho_1, \rho_2, ..., \rho_m\}$. In Eq. (4.4), *S(V)* is built recursively.

$$\begin{cases} S(\{K_1, K_2\}) = Max[Collocate(\{\emptyset\}, \{K_1, K_2\}, \rho_j)], & \text{if } |V| = 2 \\ S(V) \qquad = Max[Collocate(S(V - \{K_i\}), \{K_i\}, \rho_j)], & \text{if } |V| > 2 \end{cases} \tag{4.4}$$

Here, the function $Collocate(S(V), K_i, \rho_j)$ represents the collocation estimator algorithm given at Algorithm 2. As the collocation estimator algorithm takes a specific O&P as an input, the parameters constitute the processor wise queue ρ (i.e., the O&P). The parameters are given as follows:

$S(V)$ is the current O&P.

K_i is a new kernel that will be added to $S(V)$.

ρ_j is the processor queue into which K_i will be added.

The *Collocate* function provides the execution time and energy consumption for that O&P. The MAX operation then considers the placement of K_i in all processors, P, and selects the placement where the weight is the maximum based on the cost function and EPTO. The base case of Eq. (4.4) determines the placement with maximum weight for two kernels since it takes (at least) two kernels to collocate. For example, there are four O&Ps in a scenario with two kernels and two PUs. The base case determines the best O&P from these four O&Ps by using the cost function. For a scenario with three kernels and two PUs, the second case of Eq. (4.4) is used. In this case, one kernel is separated and placed in all the PUs along with the best O&P of the remaining two kernels, which are derived from the base case. Again, the cost function and EPTO are applied to determine the best O&P of the three kernels. In this way, the total set gets bigger by applying the cost function while DP eliminates unnecessary combinations.

4.5.2 DP-Based Kernel Collocation Algorithm.

The objective of the DP-based kernel collocation algorithm is to determine a near-optimal O&P based on a defined EPTO. Algorithm 3 provides a simplistic pseudo-code that implements Eq. (4.4). This algorithm takes three inputs: (1) list of kernels, (2) list of PUs, and

Algorithm 3 DP-Based Kernel Collocation

```
1: [t] **Input:** $n$ Kernels, $K = \{K_1, K_2, ..., K_n\}$,
              $m$ Processors, $P = \{P_1, P_2, ..., P_m\}$, and $EPTO$
2: **Output:** O&P with MAX weight, $S(V)$.
3: for $i = 2$ to $n$
4:     if $i == 2$
5:         Calculate all possible base cases.
6:         continue
7:     for each $V \in K$ where $|V| = i$
8:         $S(V) \leftarrow \{\emptyset\}$
9:         for each $K_i \in V$
10:            $Max\_weight \leftarrow 0$
11:            $V\_partial = V - \{K_i\}$
12:            for each $P_j \in P$
13:                $S(V_{P_j}) = Collocate(S(V\_partial), \{K_i\}, \rho_j)]$
14:                Get time $\tau_c$ and energy $\varepsilon_c$
15:                Update $\tau_{max}$, $\tau_{min}$, $\varepsilon_{max}$, $\varepsilon_{min}$
16:            for each $P_j \in P$
17:                $\omega(V_{P_j}) = Calculate\_weight(\tau_c, \varepsilon_c, EPTO)$
18:                if $\omega(V_{P_j}) > Max\_weight$
19:                    Set $Max\_weight = \omega(V_{P_j})$
20:                    Update $S(V) = S(V_{P_j})$
```

(3) EPTO. The output of the algorithm is the near-optimal O&P, which is denoted as $S(V)$. This starts with finding the O&P for minimal subset V (i.e., the base case where $|V| = 2$) by finding the maximum weight based on EPTO and the cost function. The algorithm then increases the size of V by one new kernel while reusing the saved maximum weighted (i.e., best) O&Ps from past iterations. The algorithm finds the best O&P, which is processor wise queues, $\rho_{maxweight} = \{\rho_1, \rho_2, ..., \rho_m\}$. At [Lines 1–2], input and output are defined. At [Line 3], the algorithm iterates through the smallest ($|V| = 2$) to the largest subset size $|V| = n$. In [Lines 4–7], the base case of Eq. (4.4) is calculated by considering two kernels and m PUs. In [Line 8], the algorithm iterates over every possible subset V of K, where $|V| = i$. [Line 9] initializes $S(V)$. In [Line 10], every K_i is considered from the current set V. In [Line 12], K_i is separated, and a partial set $V\_partial$ is formed. The best O&P for this partial set is already

calculated in the previous iteration. At [Line 13], every processor, P_j, is considered for a potential placement for kernel K_i. At [Line 14], K_i is added to the O&P of $S(V\_partial)$, *Collocation Estimator* algorithm (i.e., Algorithm 2) is called, and the result is stored at $S(V_{P_j})$. In [Line 15], execution time, τ_c, and energy consumption, ε_c, are updated, which are the output of Algorithm 2. In [Line 16], τ_{min}, τ_{max}, ε_{min}, and ε_{max} are updated. In this way, time and energy are calculated for all the possible subsets that are built on top of the best O&P of previous iterations. Now, there is a set of execution time and energy consumption. At [Line 19], the cost function in Eq. (4.3) is invoked by using the $Calculate\_weight()$ function that uses energy, time, EPTO, and all the minimum-maximum values. In [Lines 21–22], the algorithm checks whether the current O&P provides maximum weight. If it does, Max_weight and $S(V)$ are updated. When the algorithm finishes its iterations, $S(V)$ contains the desired near-optimal O&P for the given inputs.

4.5.3 Complexity. As mentioned previously, Algorithm 2 has a complexity of $O(nm)$. The outer loop of Algorithm 3 at [Line 3] is iterated $(n-1)$ times, and the selection of subsets V with size i results in the loop at [Line 8] and the innermost loop to be iterated $\sum_{i=2}^{n} \binom{n}{i}$ and $\sum_{i=2}^{n} i \binom{n}{i}$ times, respectively, resulting in a complexity of $O(n^2 2^{n-1} m^2)$. However, a brute force search over all the combinations will reach a complexity of $O(nm^2 n!)$. The DP solution is faster but might not always yield the optimal O&P. The performance of the DP-based strategy is evaluated in the next section.

4.6 Experimental Setup

These experiments were conducted on NVIDIA's Tegra Xavier SoC development platform. For parallel kernel execution on the CPU, the OpenMP programming model was used. All the CPU executions refer to the multithreaded execution that

Table 12. Benchmark kernels. All the benchmarks are taken from Rodinia benchmark suite except *triad*, which is taken from Roofline tool. Two benchmark names are abbreviated (*pf* = *particlefinder* and *hw* = *heartwall*).

Serial Number	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**
Benchmarks	*cfd*	*srad1*	*srad2*	*pf*	*triad*	*hw*	*nw*	*lud*
DRAM byte	210 G	449 G	663 G	6 G	918 G	221 G	91 G	18 G
Total flop	518 G	441 G	1.3 T	33 G	115 G	1.1 T	12 G	74 G
OI	2.46	0.88	2.0	5.2	0.125	5.27	0.14	3.93
GPU Flop/s	31.0 G	29.8 G	77.1 G	2.8 G	22.6 G	113.8 G	1.2 G	1.3 G
GPU Avg. Power	8.3	8.3	7.7	7.9	9.1	15.6	5.5	12.1
CPU Flop/s	22.5 G	87.1 G	21.8 G	2.3 G	19.4 G	9.3 G	2.4 G	18.9 G
CPU Avg. Power	9.5	9.0	10.6	17.3	15.3	9.4	12.1	10.6
Favorable PU	Both	CPU	GPU	Both	Both	GPU	Both	CPU

uses all the available cores. For GPU and PVA execution, the CUDA and OpenCV programming models are used, respectively. The Xavier platform gives users the ability to measure power consumption for the CPU, GPU, and PVA separately. The *tegra_parser* tool [164] was used to measure PU-wise power consumption. To measure the number of flops and memory R/W bytes from the LLC to the system memory (i.e., operational intensities) for the kernels, NVIDIA's proprietary profiling tool *nvprof* was used. Since the ARM Carmel CPU of Xavier does not yet have the required counters to calculate operational intensity, the values reported by *nvprof* were used. This approach led to a reasonable approximation for the CPU execution, which is further explained in Section 4.7.2.

For GPU and CPU characterization and execution, scientific kernels from the Rodinia benchmark suite [165] and synthetic kernels from the Roofline toolkit [158] were used to demonstrate the effectiveness of MEPHESTO. For PVA, applications from the OpenCV [166] benchmark suite, which is bundled with NVIDIA's Vision Programming Interface (VPI) software development kit [167], were used. The data corresponding to the characteristics of these kernels is presented in Tables 12 and 13.

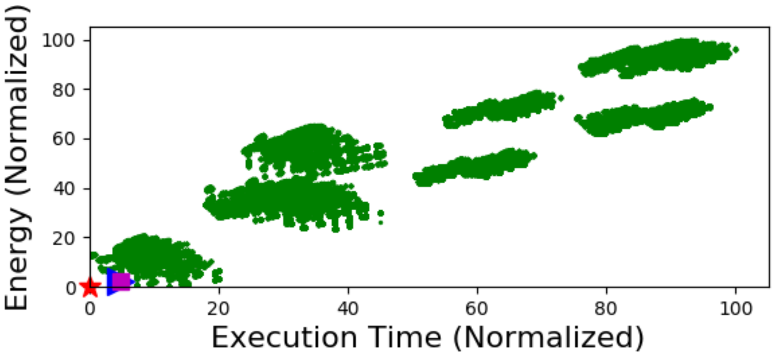


Figure 18. Result for benchmark kernels for EPTO (0, 0): near-optimal O&P selection for all combinations. The red star is EPTO, the blue triangle is the optimal solution, and the magenta square is the near-optimal solution by DP. The absolute minimum-maximum value pairs for energy and time are 0.72–3.3 kWatt-seconds and 45.5–255.5 seconds.

While experimenting, no power cap was set in the device. Power caps in Xavier limit the maximum frequency in PUs and thus change the kernel behavior; DVFS picks frequency from a defined set of frequency for a power cap. Changing power caps requires the kernels in the affected PUs to be reprofiled to generate data for Tables 12 and 13. By using these kernels, the proposed DP-based scheduling was compared with three other scheduling approaches: (1) optimal scheduling for a specific EPTO, (2) random scheduling, and (3) a greedy scheduling approach commonly used by the related work [112].

4.7 Experimental Results

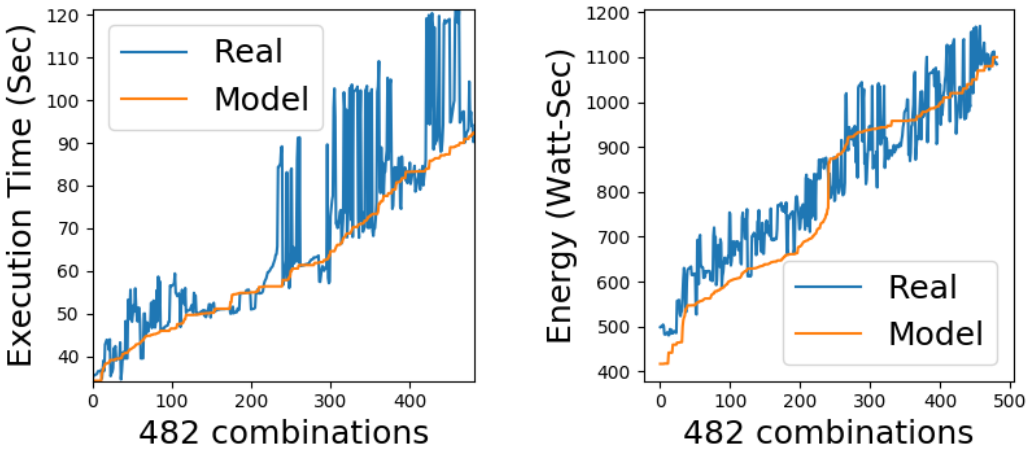
This section evaluates the efficacy of MEPHESTO in six steps: (1) the model and algorithms are shown to result in a near-optimal solution in a CPU+GPU collocated execution scenario, (2) the prediction accuracy of the model is evaluated, (3) different EPTO goals are evaluated, (4) a comparison with a greedy algorithm is presented, (5) the experiments are extended to include concurrent execution on CPU+GPU+PVA, and (6) the overhead associated with MEPHESTO is discussed.

4.7.1 Kernel Collocation Using Scientific Kernels.

The first experiment was a high-level feasibility study to demonstrate how the proposed collocation technique can find a near-optimal solution among hundreds of thousands of possible O&P combinations. In this experiment, the EPTO goal is set as (0,0), which indicates that energy and performance should both be optimized. Although this is an unrealistic goal for the targeted platforms, this experiment is used to argue why the EPTO should be treated as a trade-off knob rather than an "optimize-all" target between energy and performance with more realistic goals such as (100,0), (0,100), and (50,50).

The proposed algorithm was evaluated by using the eight benchmark kernels listed in Table 12. The LLC-to-DRAM R/W bytes and the number of flops reported in the table are generated by using the counters provided by NVIDIA. The execution time and average power data for each kernel are collected in standalone mode for the CPU and GPU. There are $282,241$ possible ways in which eight kernels can be collocated (i.e., $282,241$ O&Ps on two PUs). To apply the EPTO of (0,0), the energy consumption and execution time of each O&P were normalized to a value between 0 and 100. Then, all the possible O&Ps were plotted in Figure 18. The green circles represent all the possible O&P (random scheduling). The EPTO (0, 0) was marked with a red star, the optimal solution with a blue triangle, and the DP-based solution with a magenta square. The optimal O&P and the near-optimal solution found by the algorithm are closely located. Hence, the DP-based strategy is capable of selecting a reasonable near-optimal solution.

A trade-off between the execution time and the energy consumption is necessary when the inverse relationship between them is observed. However, in Figure 18, the inverse relation is absent; hence, the need for a trade-off seems unnecessary (i.e., there is no need for EPTO) because some kernels are better suited for one processor. Running a kernel on the ill-suited processor leads to a nonoptimal result (i.e., higher execution time and a higher level of energy consumption). This explains why there are multiple clusters in Figure 18. Section 4.7.3 further delves into the most close-to-optimal cluster, which is the leftmost-bottom cluster and shows the effects of different EPTO values on the success of the proposed scheduling technique.

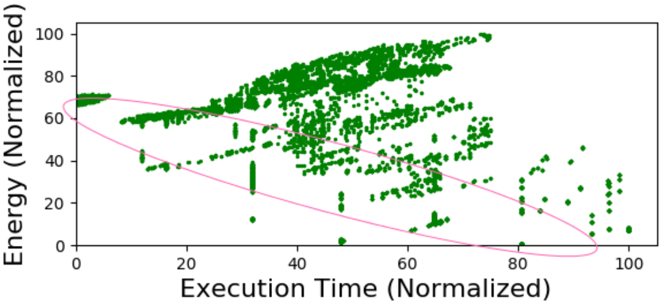


(a) Execution time comparison.

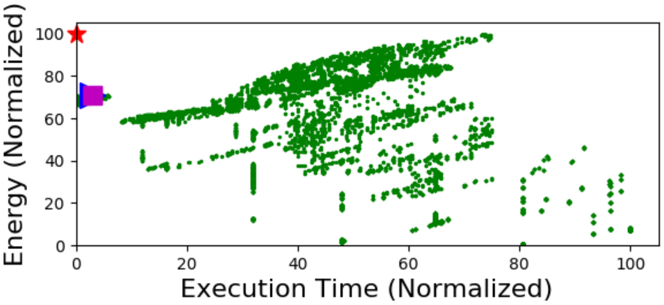
(b) Energy consumption comparison.

Figure 19. Model accuracy of MEPHESTO.

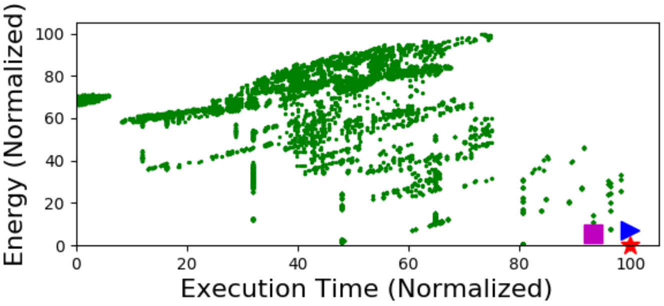
4.7.2 Accuracy of the Empirical Model. The proximity of the O&P generated by the algorithm to the optimal O&P relies on the accurate estimation of collocated execution times and energy consumption for each kernel produced by the model. To further evaluate the prediction accuracy of the models, we focused on a subset of five kernels in Table 12. We observed how the modeled energy and execution time match with the actual execution for all the combinations possible with five kernels, which is 482. Figures 19a and 19b depict the execution time and energy consumption for all 482 combinations by comparing the estimations from the model and the actual execution. In these figures, the x-axis shows the



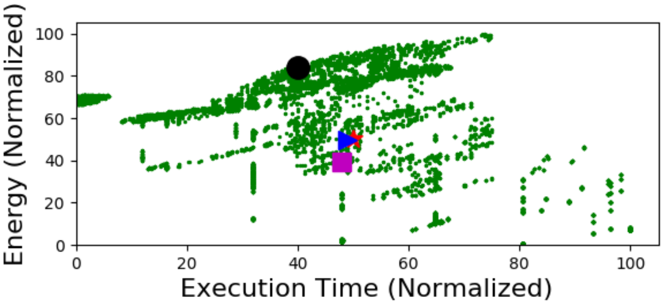
(a) Inverse relation of time and energy.



(b) EPTO (0,100).



(c) EPTO (100,0).



(d) EPTO (50,50).

Figure 20. Demonstration of the need for EPTO. The red star is EPTO, the blue triangle is the optimal solution,the magenta square is the near-optimal solution by DP, and the black circle is the greedy solution. The x- and y-axes are normalized. The absolute minimum-maximum value pairs for energy and execution time are 0.72–1.2 kWatt-seconds and 45.5–77.4 seconds.

specific O&P combination. The model energy consumption and execution time values, which are denoted by the orange lines, are sorted, resulting in a smooth curve. The real execution time and energy consumption corresponding to the specific O&P for every data point are denoted by blue lines. The analysis shows that the model estimate is on par with the actual values for the execution time and energy consumption. Relative accuracy for an O&P is computed by the formula, $accuracy = [1 - Absolute(Real - Model)/Real]$*100, and then averaged for all. The average model accuracy for execution time and energy consumption is 88.4 and 92%, respectively.

4.7.3 Experiments with Different EPTO Goals.

The experiment presented in Figure 18 includes many possible O&P combinations that result in unpractical high-energy consumption and execution times. To better demonstrate the scale at which various EPTO goals can be used to achieve the desired trade-off, the amount of O&P combination space was reduced by identifying the kernels that are more suitable to run on CPUs or GPUs and fixing them to the corresponding PU.

A kernel is considered to be more suited for a specific processor if the ratio of the execution time is at least two times faster while taking average power into consideration. For example, a kernel has a $Flops/s$ value of $t1$ and $t2$ in PU1 and PU2 and the $Avg.Power$ of $pw1$ and $pw2$ in PU1 and PU2. If $t1/t2 > 2$ (at least two times faster) and $pw1/pw2 < 2$, then the kernel is suitable for PU1. In the same way, we determined whether the kernel is suitable for PU2. If the kernel does not satisfy the condition for any processor, then the kernel is considered favorable by both processors, and its placement in all these processors is considered.

Based on the $Flops/s$ and the $Avg.Power$ values reported in Table 12, the CPU-friendly kernels were identified as $srad1$ and lud, and the GPU-friendly kernels were identified as $srad2$ and $heartwall$ (last row of the table). On the other hand, the cfd, $particlefinder$, $stream-triad$, and nw kernels can be run on any processor since no PU always favors their execution.

After the fixed affinities are set, badly performing O&Ps are eliminated, and the inverse relationship between the execution time and energy consumption is revealed in Figure 20a. The circled diagonal region demonstrates that there is no best solution that optimizes energy and performance, and there is a clear need for different EPTO goals. Figures 20b, 20d, and 20c show the optimal O&P combination (triangle), the near-optimal combination found by the algorithm (square), and the solution found by a greedy algorithm (circle) for different EPTO goals (shown by stars) of (0, 100), (50, 50), and (100, 0), respectively. These figures demonstrate that the DP-based scheduling can select the near-optimal O&P to achieve the desired trade-off between energy consumption and execution time.

EPTO provides significant control over kernel collocation decisions. If the user wants to pick an O&P without having a desired trade-off in mind, EPTO (0, 0) can be chosen. EPTO (0, 0) will always provide an O&P that is close to the diagonal line shown in Figure 20a, but no specific trade-off is guaranteed. If the user wants the fastest execution time or the least energy consumption, EPTO (0, 100) and EPTO (100, 0), respectively, will guarantee that trade-off. Other EPTOs, such as EPTO (50, 50), also guarantee the desired trade-off. This strategy provides the run time system with more control over the device and, if necessary, empowers the run time system to dynamically choose different EPTO values based on the device's energy consumption priority. Moreover, this algorithm works irrespective of the system power cap, which in turn provides an extra level of control.

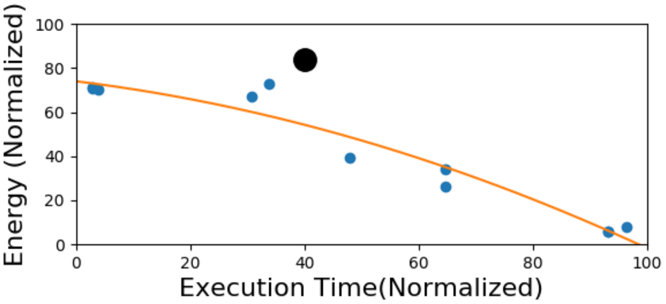


Figure 21. Comparison with greedy scheduling. Blue dots are EPTO points, and the black circle is the greedy solution.

4.7.4 Comparison with Greedy Algorithms. We believe that a study that consults both execution time and energy consumption to mitigate memory contention has never been performed. For this reason, the closest work in which the greedy algorithm is only dependent on the execution time [112] was chosen for comparison. The greedy algorithm [112] starts by scheduling the longest GPU-friendly kernel in GPU and selects a CPU-friendly kernel to collocate. This strategy of collocation excludes lower operational intensity for reducing memory contention. This is a classic scenario that stems from the fact that a compute-intensive kernel should be collocated with a memory-intensive kernel. When one kernel in one processor finishes its execution, the next PU-friendly kernel to that processor is chosen. When all PU-friendly kernels are scheduled, the neutral kernels are chosen.

Following this strategy, an O&P of 3675|2841 is chosen by the greedy solution for which the numbers represent the kernels from Table 12. This O&P is compared in Figure 20d by using a black circle. The positioning of the black circle reveals that there are many better solutions available in terms of the energy consumption and the execution time. Figure 21 presents a better comparison between the greedy approach and the proposed DP-based algorithm. The blue points are DP-based solutions for different EPTO points of 0–100, 10–90, 20–80, ..., 100–0, and a trade-

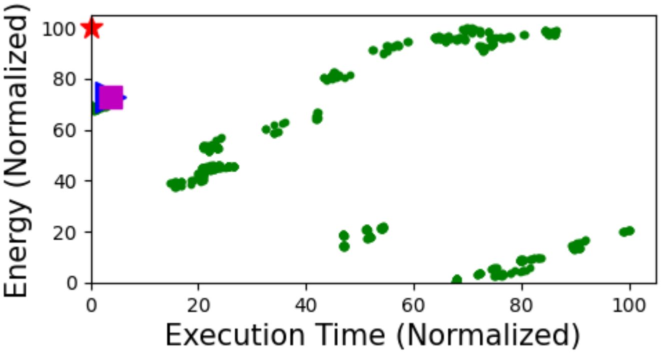
Table 13. OpenCV kernels for PVA.

Benchmark name	**klt_tracker**	**convolve_2D**	**timing**
Benchmark source	VPI.0.1	VPI.0.1	VPI.0.1
Total DRAM R/W byte	26 G	7 G	15 G
Total flop	72 G	70 G	122 G
Operational intensity	2.41	5.35	7.87
PVA Execution time (sec)	2.41	5.35	7.87
PVA Flop/s	30.0 G	13.2 G	15.5 G
PVA Avg. Power (Watt)	1.24	1.39	1.39

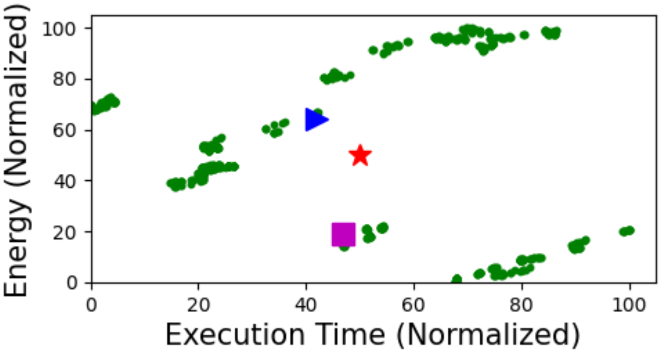
off line is also drawn based on the positioning of different levels of EPTO. The greedy solution is observed to be significantly far from the trade-off line. The best case for execution time—EPTO (0, 100)—provides a scheduling in which execution time is 46.5 seconds for eight kernels, whereas the scheduling picked by the greedy solution provides an execution time of 58.3 seconds, which is 11.8 seconds more (i.e., 20% savings by the DP-based approach). On the other hand, the best case for energy consumption—EPTO(100, 0)—provides the total energy consumption of 0.75 kWatt-seconds, whereas the greedy solution shows the energy consumption of 1.1 kWatt-seconds, which translates to 32% energy savings. Although the DP-based solution is more computationally expensive than the greedy algorithm, it can save more execution time. For example, the DP-based solution takes 1.1 seconds to find the scheduling for eight kernels but can save 11.8 seconds. Moreover, the DP-based approach provides the means to achieve the desired EPTO.

4.7.5 Three-PU Scenario: CPU, GPU, and PVA.

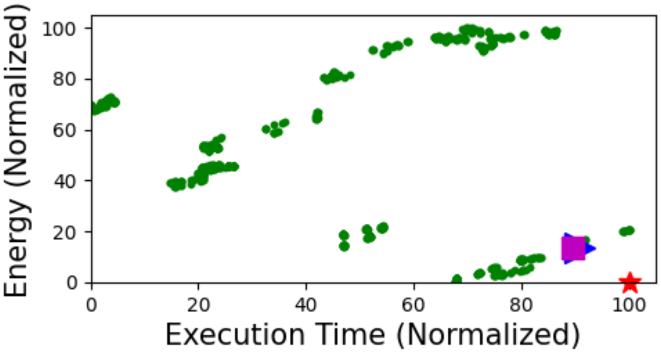
To demonstrate that MEPHESTO can work for diversely heterogeneous systems, the experiments were extended to cover three different PUs: CPU, GPU, and PVA. In this experiment, a different subset of eight kernels was used: five top kernels from Table 12 for execution on CPU and GPU and three OpenCV kernels from the NVIDIA VPI samples from



(a) EPTO (0,100).



(b) EPTO (50,50).



(c) EPTO (100,0).

Figure 22. Experiments with three processors: CPU, GPU, and PVA. The red star is the EPTO, the blue triangle is the optimal solution, and the magenta square is the near-optimal solution picked by DP. The x- and y-axes are normalized. The absolute minimum-maximum value pairs for energy and execution time are 0.47–0.78 kWatt-seconds and 34.01–58.47 seconds.

Table 13 for execution on PVA. Among the five kernels from Table 12, *srad*1 is CPU-friendly, *srad*2 is GPU-friendly, and remaining three are considered for both CPU and GPU. For all eight kernels in which one kernel favors CPU, one favors GPU, and three favor PVA, there are 720 O&Ps. The optimal and near-optimal O&P selections are plotted in Figure 22. For two different EPTO targets—(0, 100) and (100, 0)—the DP-based algorithm was able to select near-optimal solutions in Figs. 22a and 22c, respectively. However, EPTO(50, 50) in Figure 22b shows an interesting case where optimal and the DP-based solution are far away from each other. While the distance between the optimal point (denoted by a triangle) and the EPTO point (denoted by a star) is shorter, the DP-based solution was able to locate a more energy efficient solution. This is mainly due to the fact that the Euclidean-distance-based method of finding the optimal reference point (denoted by a triangle) relies on the absolute distances and ignores whether the optimal point uses more or less energy/execution-time. However, the DP-based approach was able to pick a solution from one of the closest clusters from the EPTO point. This experiment demonstrates that the proposed algorithm can also achieve the desired trade-off for three PU diversely heterogeneous systems.

4.7.6 Overhead Analysis of DP-Based Search. The overhead of the proposed DP-based solution in MEPHESTO for varying numbers of kernels to be ordered and placed is shown in Figure 23. For eight kernels, the algorithm finds the near-optimal solution in 1,061 milliseconds, and this time corresponds to only 1.9% (mentioned in the x-axis) of the total execution time compared with the minimum execution time. On the other hand, since the overhead increases exponentially, for larger kernel counts and shorter kernel execution times, the algorithm should be complemented with a windowing technique similar to the one proposed in Belviranli et al. [163]. Although this technique limits the benefits that can be obtained from considering all potential O&P possibilities, it is a simple and effective approach for controlling the increasing overhead. Moreover, we foresee that a multithreaded implementation of DP-based algorithm will help reduce the overhead.

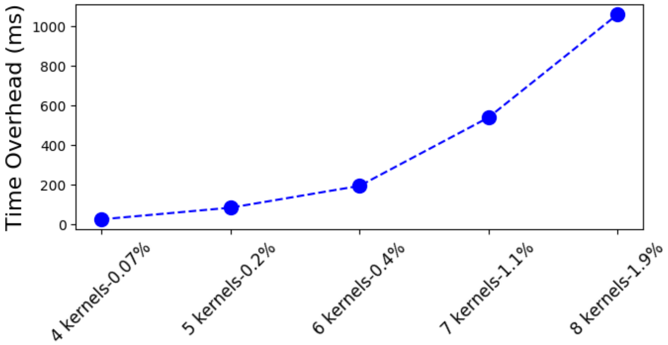


Figure 23. Overhead analysis of MEPHESTO.

4.8 Related Work

4.8.1 Memory Contention Studies. This section reports two studies that are similar to this work. The first study was done by Zhu et al. [112, 168] in which the authors studied co-scheduling on an integrated CPU-GPU system and considered a power cap. They devised a greedy algorithm that addressed memory contention from degradation in the execution time perspective while selecting frequency for power capping. However, they did not consider the impact of memory contention on power or energy. Moreover, the greedy algorithm does not provide any trade-off opportunity. The second study was done by Lee et al. [154] in which the authors designed a strategy to dynamically predict the slowdown due to memory contention. However, this study only considered execution time. Compared with these works, the strategy in this chapter defines memory contention from both the energy and time perspectives while achieving the desired trade-off. Other works [169, 170, 152] studied memory contention and stalling in heterogeneous systems with shared LLC. Pan et al. [171] designed an LLC management strategy for better performance. Cavicchioli et al. [153] studied different SoCs and fused CPU-GPU devices to characterize memory contention. Hill et al. [172] extended the Roofline model for mobile SoCs to address memory contention from the perspective of PU BW usage. These studies mainly focused on performance and did not consider the impact of memory contention on power or energy consumption.

4.8.2 Kernel Collocation in CPU-GPU Systems. Kernel collocation in a CPU-GPU environment is also another well-studied area. Kaleem et al. [157] studied scheduling in integrated heterogeneous systems in which an online profile was used for load balancing between CPU and GPU. Panneerselvam et al. [147] devised a task placement strategy in a CPU-GPU system that achieves application-specific performance goals. Zhu et al. [112] designed a greedy algorithm with post-local refinement for memory contention-aware kernel collocation. Cho et al. [104] devised an on-the-fly strategy to partition irregular workloads in integrated CPU-GPU systems without considering energy consumption. Zhang et al. [173, 174] designed a decision tree-based model to determine the impact of kernel collocation on different applications in integrated CPU-GPU systems. Pandit et al. [105] designed a dynamic work distribution that considered the data transfer need of kernels in OpenCL run time. Liu et al. [175] designed a scheduling policy for tree traversal algorithms in which CPU and GPU transfer information to make a decision. Although there are more studies in the literature that investigated kernel collocation under memory contention, to the best of our knowledge, there are no schemes that consider kernel collocation with the intention of addressing energy and performance simultaneously while considering the effects of memory contention on both factors. Moreover, almost all existing work focuses only on CPU/GPU-based systems, whereas this method works for more diverse heterogeneous systems, such as SoCs consisting of CPU, GPU, and PVA.

4.8.3 Energy-Aware Algorithm Studies.

Barik et al. [156] introduced a black-box approach for finding energy-aware scheduling by characterizing applications. Ma et al. [109] designed GreenGPU, which dynamically throttles the frequency of GPU and memory. Zhu et al. [112] dynamically finds the appropriate frequency for applications to keep the execution under a power cap. Komoda et al. [110] also studied power capping by using DVFS to find near-optimal frequency settings for CPU-GPU. Intel introduced a power capping mechanism RAPL (running average power limit) in CPUs [176]. Liu et al. [111] designed an energy-aware kernel mapping strategy in a heterogeneous system in which PUs are assigned different frequencies by using DVFS. Unlike these studies, as mentioned previously, the method in this chapter considers finding a collocation mapping that can lead to user-defined energy-performance balance while considering contention.

4.9 Summary

This chapter presents MEPHESTO, which defines memory contention in an integrated shared memory heterogeneous system in terms of energy and performance. Using operational intensity, MEPHESTO presents an empirical model to estimate a kernel collocation scenario for multiple kernels and devises a strategy to reach a desired energy-performance balance. MEPHESTO's task placement can be implemented in a runtime system following a window technique. Based on experiments, this strategy can predict execution time and energy consumption with an acceptable error rate and find a near-optimal solution that outperforms a greedy approach. Moreover, experiments demonstrated the efficacy of MEPHESTO by yielding near-optimal solutions for more than two processors. However, MEPHESTO needs an awareness of the operational intensity of kernels to model the impact of memory contention. In order to implement MEPHESTO's dynamic task placement algorithm, a runtime system must know the operational intensities of the kernels waiting in the queue before placing tasks in PUs. For this reason, operational intensity needs to be deduced at compile time.

Operational intensity is the ratio between the FLOPs and bytes transferred between LLC and DRAM. Compile time analysis tools such as COMPASS framework [13] can deduce the number FLOPS through static analysis. However, statically predicting LLC-DRAM traffic is still an open area. The next chapter addresses this problem for modern Intel CPUs.

CHAPTER V

MAPREDICT: STATIC ANALYSIS DRIVEN MEMORY ACCESS PREDICTION FRAMEWORK FOR MODERN CPUS

This chapter contains previously published and unpublished material with co-authorship. All of the presented research in this chapter was conducted as a collaboration between the University of Oregon and Oak Ridge National Laboratory (ORNL). Section 5.2 of this chapter provides a summary of the research presented at MCHPC 2020 [16]. The complete work of MAPredict is in preparation for a submission. Both of these works have the same co-authors. While working on MAPredict, I received regular guidance from Dr. Seyong Lee. MAPredict extends the COMPASS framework, which Dr. Lee developed. MAPredict also uses the Aspen domain-specific language, which was developed at ORNL. I received high-level guidance from Dr. Allen Malony and Dr. Jeffrey Vetter. I did all the experiments, writing, and data collection. Dr. Malony provided a thorough proofreading of the MCHPC 2020 paper and added his words where necessary. Dr. Lee helped with proofreading for both of the submissions.

5.1 Introduction and Motivation

This chapter outlines the design of a framework that is capable of predicting LLC-DRAM traffic using analytical models in modern CPUs (corresponding to Research Question 4 — **RQ4**: Can we model the memory access patterns and design a static analysis framework to predict LLC-DRAM traffic in modern CPUs for complex HPC applications?). With the increasing hardware complexity intended to address the memory wall problem [177], designing analytical models becomes a non-trivial task. With the rise of heterogeneous systems, the importance of such a modeling approach for prediction has increased significantly. As shown in the previous chapter, the execution of an application on an ill-suited processor may lead to non-optimal performance [12], therefore, the runtime system needs to make a quick decision on where to execute kernels on the fly. Predicting a kernel's performance and energy consumption can enable runtime systems to make intelligent decisions. For such prediction, floating-point operations (FLOPs) and memory traffic need to be counted. While calculating FLOPs is straightforward, memory traffic prediction is complex and dynamic because the memory access request can be served by the cache or the DRAM.

Statically predicting LLC-DRAM traffic is vital for three reasons. Firstly, a heterogeneous runtime system can make intelligent scheduling decisions if it can statically identify compute and memory-bound kernels based on the Roofline model [178]. The previous chapter, which introduced MEPHESTO [12], demonstrated that energy-performance-aware scheduling decisions can be made based on operational intensity (FLOPs/LLC-DRAM Bytes) of kernels. There are tools such as Intel Advisor [179] and NVIDIA Nsight Compute [180] that report the operational intensity of a kernel. However, a runtime system needs this information *before* executing the kernel to make better placement decisions. While static analysis tools can provide the FLOP count at compile time [13], statically predicting LLC-DRAM traffic needs to be explored. Simulation frameworks can provide LLC-DRAM traffic, but they are not fast enough to be integrated into a runtime system [181]. Secondly, developing a framework to predict the energy and performance of modern CPUs requires predicting LLC-DRAM memory transactions because LLC-DRAM transactions incur a significant amount of energy and time [182]. Finally, a static approach for predicting the LLC-DRAM traffic enables simulation-based design space exploration to determine the best memory configuration [183]. For these reasons, this in this chapter we aims to build a framework capable of predicting the LLC-DRAM traffic statically. However, a static analysis approach for predicting the LLC-DRAM traffic encounters three major challenges: 1) it must keep up with the continuous innovation in the processors' memory hierarchy,2) it must deal with the complex memory access patterns and different execution models (sequential and parallel), and 3) it does not have access to the dynamic information necessary to obtain high prediction accuracy. These challenges are detailed further below.

Microprocessor manufacturers have implemented many innovations to improve performance. In particular, the memory hierarchy has gone through significant changes, bringing us to the first challenge. Multi-level cache hierarchy with different sizes and cache policies, prefetching algorithms, and non-inclusive victim L3 cache are note-worthy examples [16, 184]. While these changes improve performance, analyzing and understanding a processor's memory hierarchy behavior through performance monitoring interfaces at runtime (e.g., performance counters) has become complex. Moreover, performing such analysis at compile time is significantly harder. Indeed, every microarchitecture brings some new features. Studies have been done to evaluate and understand the cache hierarchy's performance at runtime by using different benchmarks [185, 186, 187, 188, 189, 190, 184]. However, a systematic formulation and end-to-end flow to predict LLC-DRAM traffic through static analysis is still an open research area.

The second challenge stems from the fact that the application's memory access pattern impacts the LLC-DRAM traffic. For example, sequential streaming access patterns yield the best performance from the cache hierarchy, whereas a strided or random access pattern generally exhibits worse cache performance. Moreover, different compilers (this study investigates Intel and GNU compilers) can also produce different results due to variation in the generation of cache-friendly instructions.

Additionally, multi-threaded parallel execution may also impact the LLC-DRAM traffic. Previous studies [191, 192] consider memory access patterns and build analytical models to predict the LLC-DRAM traffic based on simulation. However, they fall short when modern CPU features (such as prefetching) are taken into account. Moreover, in a runtime system where prediction is needed to make a decision during execution, simulations are not fast enough.

The third challenge involves the lack of availability of dynamic information (e.g., input size) to the static analyzer. One study [15] demonstrated that instruction counts could be evaluated with respect to machine property to provide a prediction. A static analysis framework, COMPASS [13], demonstrated the capability of capturing dynamic information from user input in source code or from the runtime system to make a timely prediction [13]. However, it did not employ CPU-specific models for memory access prediction; instead, it used instruction count-based prediction. Ideally, a framework would provide a lightweight prediction based on static analysis, yield high accuracy, operate independently, and be available to the runtime system through a library call. For this reason, a framework is needed that addresses these challenges by understanding modern CPUs, employing a modern cache-aware analytical model for different access patterns, and capturing dynamic information to provide fast and accurate prediction.

This research presents MAPredict, a framework that predicts the LLC-DRAM traffic for applications in modern CPUs. To the best of our knowledge, this is the first framework that *simultaneously* addresses all of the challenges above. We present systematic experimentation on different Intel microarchitectures to elicit their memory subsystem behavior and build the analytical model for a range of memory access patterns. Through static analysis at compile-time, MAPredict creates Aspen [15] application models from annotated source code, captures the dynamic information, and identifies the memory access patterns. It then couples the application and machine model to accurately predict the LLC-DRAM traffic.

This study reports the following contributions:

- A systematic unveiling of the behavior of modern Intel CPUs for different read and write strategies, accounting for prefetchers, compilers, and multi-threaded executions;
- A formulation of a cache- and prefetching-aware analytical model using application, machine, and compiler features;
- A static analysis driven framework named MAPredict to predict LLC-DRAM traffic at compile time by source code analysis, dynamic information, and analytical modeling; and
- An evaluation of the MAPredict framework using 130 workloads (summation of number_of_functions * input_sizes) from different benchmarks in four microarchitectures of Intel, where we achieve higher prediction accuracy for regular access patterns when compared to the models from literature. MAPredict also provides means to combine static and empirical observation for irregular access.

5.2 Understanding Memory Reads and Writes in Intel Processors

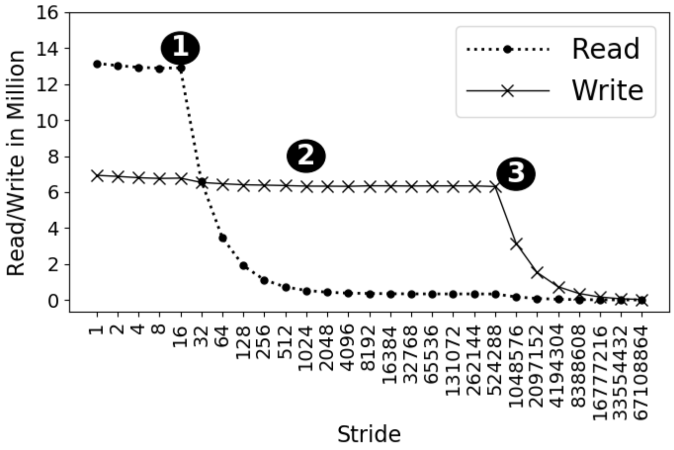
To design a static analysis framework for LLC-DRAM traffic prediction, it is necessary to explore the factors that trigger an LLC-DRAM transaction. This section systematically unveils these factors triggering LLC-DRAM memory transfers by studying the handshake between the application and the processor's memory hierarchy. From the application's viewpoint, the memory access pattern plays a vital role. The two most common memory access patterns — sequential streaming

Table 14. Machines and microarchitectures.

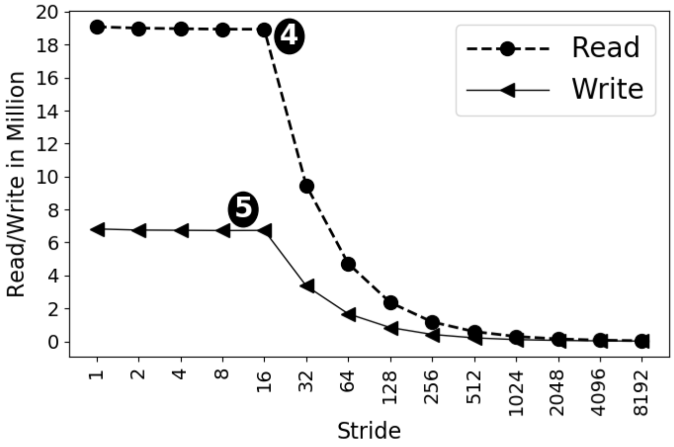
Name	**Year**	**Processor detail. here L3 = LLC**
Broadwell	2016	Xeon E5-2683 v4, 2 sockets, 32 cores, 2 NUMA domains, L2 - 256 KiB, L3 - 40 MiB
Skylake	2017	Xeon Silver 4114, 2 sockets, 20 cores, 2 NUMA domains, L2 - 1 MiB, L3 - 14 MiB
Cascade Lake	2019	Xeon Gold 6248, 2 sockets, 40 cores, 2 NUMA domains, L2 - 1 MiB, L3 - 28 MiB
Cooper Lake	2020	Xeon Gold 6348H, 4 sockets, 96 cores, 8 NUMA domains, L2 - 1 MiB, L3 - 132 MiB

access and strided memory access are considered. Cache line size, page size, initialization, prefetching mechanism, and parallel execution are identified as the key hardware factors. This section also explores the effects of the evolution of CPU microarchitectures.

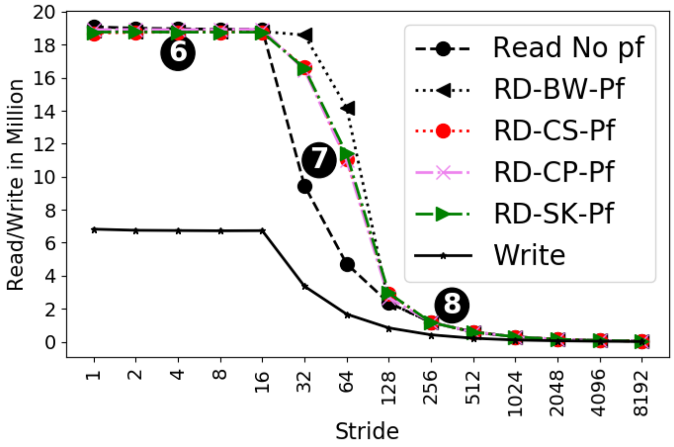
5.2.1 Description of the Hardware. In this study, Intel CPUs are considered because they are the most widely available processors in HPC facilities [193]. By applying our techniques across Intel CPU microarchitecture generations, we can test MAPredict's robustness with respect to newer memory system features. Further, the findings from this work can be extended for use on processors from other manufacturers such as AMD and ARM. Table 14 depicts the four recent microarchitectures that are a part of this study — Broadwell (BW), Skylake (SK), Cascade Lake (CS), and Cooper Lake (CP). The introduction of the non-inclusive victim L3 cache and the larger L2 cache (starting from the SK processors) is the most important change concerning the memory subsystem [184]. Even though the design of the cache has remained largely unchanged since the SK microarchitecture, we find it important to consider the effect of different cache sizes in newer processors for completeness.



(a) Impact of cache line size and page size.



(b) Impact of array initialization.



(c) Impact of prefetching on read traffic.

Figure 24. LLC-DRAM traffic for different read and write scenarios in Intel processors. LLC-DRAM traffic is shown at Y-axis.

5.2.2 A Tool for Measuring the LLC-DRAM Traffic. To understand the behavior of the cache subsystem of Intel processors, LLC-DRAM traffic needs to be measured. (MAPredict aims to generate such data using performance models). To accurately measure the LLC-DRAM traffic, uncore counters of memory channels associated with the integrated memory controllers (IMC) are measured since L3 cache miss in Intel processors do not reflect the write traffic [16]. The BW processor has two IMCs, whereas SK, CS, and CP have three IMCs. Each IMC controls two memory channels, and there are two counters (read and write) for each channel. For a single socket, there are a total of eight counters for BW and twelve for others. These counters provide LLC-DRAM traffic measurement in the unit of cache line (64 bytes), and this unit is followed throughout this study. Multiplying the measured counter value with the cache line size provides the total bytes transferred between the LLC and the DRAM.

TAU [8] is used to measure the corresponding PAPI [194] uncore counters (imcX::UNC_M_CAS_COUNT). TAU and PAPI are used for their portability and their ability to measure function-wise LLC-DRAM traffic for an application. PAPI counters are measured socket-wise, and for this reason, counters are read from the socket where the code is executing. When multi-threaded execution is used, threads are pinned to the cores of the socket from where the counters are read. This is to ensure the correctness of the measurement and is *not* related to thread migration. If cores belonging to multiple sockets are used, counters are read from multiple sockets.

We develop a script-based dynamic analysis tool to execute the application and collect the TAU profiles for each counter. The dynamic analysis tool parses the profiles to generate function-wise LLC-DRAM traffic for read and write operations. Internally, a custom tool [195] is invoked to study the impact of prefetching (by toggling it). The LLC-DRAM traffic measurements reported in this study are gathered through this dynamic analysis tool.

5.2.3 Different Read and Write strategies. To investigate the application-cache interplay, a variant of vector multiplication code that exhibits sequential streaming (stride = 1) and strided access pattern (stride > 1) is considered. The code has three arrays (100 million 32-bit floating-points) allocated and aligned for vector multiplication. Since the cache line length of these Intel processors is 64 bytes, in an ideal case, an array size of 100M (M represents million) for sequential streaming vector multiplication should generate 6.25M writes of cache lines (100M * size_of_32bit_float / cache_line_size = 6.25M) and 12.5M reads. However, Figure 24 tells a different story, where read and write traffic for varying stride (by doubling) is shown. Cases portrayed in Figure 24 are discussed below.

5.2.3.1 Impact of Cache Line Size. In Figure 24a, the read-write traffic is shown where the read traffic is close to 12.5M for stride 1. This trend continues until stride 16 (64 bytes/size_of_32bit_float=16), referenced by ❶. Because a cache line is 64 bytes long, while fetching one 32-bit floating-point data, the memory subsystem fetches 15 (60 bytes worth) additional neighboring data. Thus, the read traffic does not reduce until stride 16. However, after stride 16 at ❶, the read traffic halves every time the stride is doubled. This clearly shows the impact of cache line size on the LLC-DRAM traffic. Write traffic for stride 1 is also close to 6.25M. However, for the write traffic, it appears that the cache line size does not have any impact, given that the region ❷ stretches up to a stride of 524,288. For a stride of one (100M access) and a stride 524,288 (only 190 access), the same number of cache lines (6.25M) are transferred. Thus, the cache line size has an impact on the read traffic but not on the write traffic. This observation is explained in Section 5.2.3.2.

5.2.3.2 Impact of Page Size. In Figure 24a, the cache line length has no impact on the write traffic because the write array was only allocated but not initialized. When the write array is accessed, page zeroing occurs, and irrespective of the actual accesses, all the cache lines in a page become dirty. Page zeroing helps avoid information leakage from previous content [196]. For this reason, the write traffic in Figure 24a is not affected by the cache line size. Instead, it depends on the page size. The default page size on Intel processors is 4 KiB, i.e., a stride of 1024 for 32-bit floating-point. Because Linux supports "transparent huge pages", it allows larger page sizes. Intel processors support large pages of 2 MiB and 1GiB size. Because the data structure size in our study was 100M, a page size of 2MiB was selected. This explains why we see a transition at ❸ on a stride of 524,288. After that point, the write traffic is halved every time the stride is doubled.

5.2.3.3 Impact of Initialization. In Figure 24b traffic is shown when the write array is initialized. The write traffic is close to 6.25M at stride 1, and at this point, the impact of the cache line size is visible at ❺. Specifically, until a stride of 16, no page zeroing takes place. After a stride of 16, the traffic is reduced by half when the stride is doubled. However, the read traffic is close to 18.75M for stride 1, indicating that three vectors are read instead of two (the region pointed by ❹). The extra read traffic occurs due to the "write-allocate" cache policy. When a store miss happens, a cache line is retrieved from the DRAM to the cache, and this causes the extra read (also called an "allocating store"). After a stride of 16, traffic is reduced by half when the stride is doubled. All four microarchitectures considered in this study show this same trend.

5.2.3.4 Impact of Hardware Prefetchers. Intel implements aggressive prefetching, but not all the details are openly available to the community. In the experimental results shown in Figure 24a and Figure 24b, prefetching is disabled. The impact of toggling prefetching support on the read traffic is shown in Figure 24c. (BW-Pf means Broadwell with prefetching). Note that prefetching has no impact on the write traffic. Intel has four types of hardware prefetchers per core determined by four MSR bits [197]. Three regions in read traffic are shown in Figure 24c. The regions ❻ (stride 1 to 16) and ❽ (stride 128 and onward) show no visible difference with prefetching enabled. Further investigation by experimenting with a smaller stride confirms that the impact of prefetching vanishes after a stride of 80 in all microarchitectures. Hence, a stride of 80 is the starting point of region pointed by ❽. However, ❻ (i.e., a stride of 1 to 16) is the region that benefits most from prefetching. For example, the execution times for stride of 1 with and without prefetching are 384 and 897 milliseconds.

The region ❼ (from a stride of 32 to a stride of 128) shows interesting behavior and a visible difference in performance when prefetching is enabled. The difference here arises from the extra cache lines that are fetched. The first bit of MSR causes an additional cache line to be fetched, making the effective cache line length 128 bytes (as opposed to 64). Hence, for a stride of 32, the read traffic behaves similarly to a sequential stream of data. The second bit of MSR causes an additional cache line to be fetched in the L2 cache, resulting in three cache lines being fetched for one access. For this reason, for a stride of 64, each access could result in three cache lines being fetched (the last two bits are for L1 and history data-based prefetcher). Moreover, the prefetching behavior in region ❼ is not the same for all microarchitectures. Read traffic is 10% higher in BW when compared to others (SK, CS, and CP show the same level of read traffic). This observation can potentially be attributed to the change in the cache subsystem design following the BW microarchitecture.

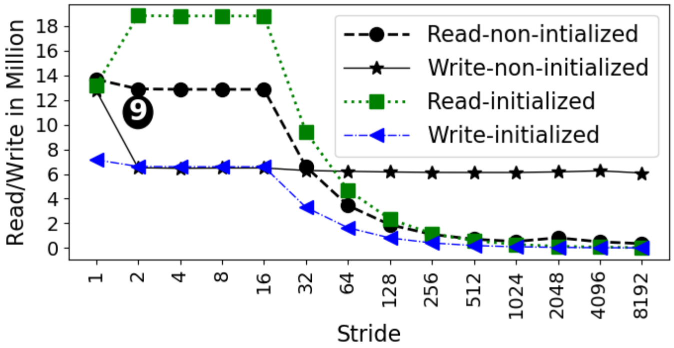


Figure 25. Impact of using Intel compiler.

5.2.3.5 Impact of Compiler. The GNU compiler is used to generate Figure 24. However, using the Intel compiler can provide a different result depicted by Figure 25. From a stride of 2 and onward, all the data in Figure25 shows a similar trend as the GNU compiler. However, the region pointed by ❾ shows the difference between the two compilers when the stride is 1. The Intel compiler has a default option known as the "streaming store" or "non-temporal store", where sequential streaming access can be automatically detected. The GNU compiler does not implement this feature by default. However, non-temporal store can be turned on or off using compiler flags and intrinsics in the Intel and GNU compilers. When the streaming store option is used, data is not read from the DRAM for a store miss. Instead, the data is directly written to DRAM (bypassing the cache) through a write-combining buffer. This explains why the extra read traffic for an initialized write array is not present for a stride of 1. The use of a streaming store improves performance significantly (upto 20%). Another observation for a stride of 1 is the write traffic for the non-initialized case. The write traffic doubles for this case, suggesting that the page zeroing is separated from the actual store operation.

5.2.3.6 Impact of Multi-threaded Parallel Execution. Because streaming and strided access patterns exhibit regular memory access, no visible

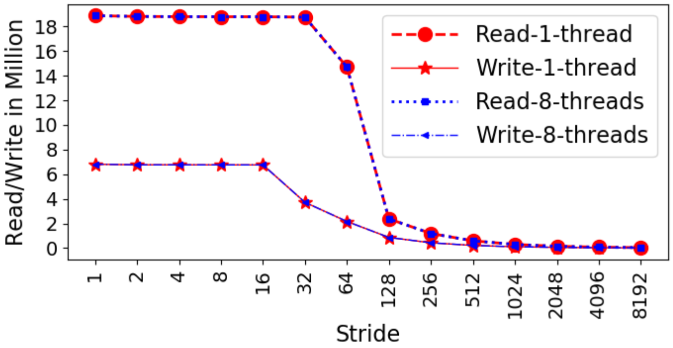


Figure 26. Comparing single vs. multi-threaded runs.

difference is observed when comparing single-threaded execution with multi-threaded execution (using 8 OpenMP threads). Figure 26 depicts this observation. However, for a large dataset with a complex irregular pattern, multiple threads share the cache, and hence, the data is overwritten by other threads, thereby increasing the DRAM traffic.

5.3 Modeling Different Types of Access

A static analysis framework needs analytical models for different types of memory access patterns to predict the LLC-DRAM traffic. For this reason, this section builds on the findings from Section 5.2 to formulate analytical models for different access patterns. Three kinds of regular access patterns are discussed in this section. First, the model is formulated for the sequential streaming access pattern to predict LLC-DRAM cacheline transfer. Then, models are prepared for other access patterns by using the model for streaming access patterns. In the end, random access patterns (irregular) are discussed.

5.3.1 Sequential Streaming Access Pattern. The sequential streaming access pattern (where memory access is consecutive, i.e., stride = 1) is one of the most common access patterns found in applications. Prefetching does not impact the amount of traffic transferred between LLC and DRAM for this pattern. However, the impact of the cache line and page size needs to be considered.

5.3.1.1 Read Traffic.

Because the LLC-DRAM read transaction is done in a unit of cache lines, the amount of read traffic can be expressed using Eq 5.1. In Eq 5.1, a data structure size is $\mathrm{Element}_{\mathrm{count}}$ and the size of each element is $\mathrm{Element}_{\mathrm{size}}$ bytes. $\mathrm{Read}_{\mathrm{count}}$ is the number of LLC-DRAM transactions for reading a data structure. Data structure initialization has no impact on $\mathrm{Read}_{\mathrm{count}}$. Because alignment is not certain, the ceiling is considered.

$$\mathrm{Read}_{\mathrm{count}} = \left\lceil \frac{\mathrm{Element}_{\mathrm{count}} * \mathrm{Element}_{\mathrm{size}}}{\mathrm{Cacheline}_{\mathrm{size}}} \right\rceil \tag{5.1}$$

5.3.1.2 Write Traffic.

The initialization of the data structures plays an important role for write traffic. At first, the case where the data structure is not initialized but only memory is allocated is discussed. For such a case, the page size becomes the deciding factor because of page zeroing (as shown in Section 5.2.3.2). In Eq 5.2, $\mathrm{Write}_{\mathrm{not\_init}}$ is the number of cache line transfers when the data structure is not initialized). Because the machines in Table 14 support transparent huge pages by default, the page size picked by the operating system (OS) depends on the data structure size (We made no changes in the OS). The ceiling is considered to capture the extra traffic from the fragmented access on the last page.

When a data structure is initialized, page zeroing does not take place, and the cache line becomes the deciding factor. Because of write-allocate policy in Intel, existing data is read from DRAM before writing on to it. So one write operation also causes one read operation. The write traffic ($\mathrm{Write}_{\mathrm{init}}$) is shown in Eq 5.3. The extra read traffic ($\mathrm{Read}_{\mathrm{for\_write}}$) generated for the write operation is shown in Eq 5.4.

$$\text{Write}_{\text{not\_init}} = \left\lceil \frac{\text{Element}_{\text{count}} * \text{Element}_{\text{size}}}{\text{Page}_{\text{size}}} \right\rceil * \frac{\text{Page}_{\text{size}}}{\text{Cacheline}_{\text{size}}} \tag{5.2}$$

$$\text{Write}_{\text{init}} = \left\lceil \frac{\text{Element}_{\text{count}} * \text{Element}_{\text{size}}}{\text{Cacheline}_{\text{size}}} \right\rceil \tag{5.3}$$

$$\text{Read}_{\text{for\_write}} = \begin{cases} 0 & \text{if data structure is not initialized} \\ \text{Write}_{\text{init}} & \text{if data structure is initialized} \end{cases} \tag{5.4}$$

Thus, total read traffic for streaming access, $\text{Read}_{\text{stream}} = \text{Read}_{\text{count}} + \text{Read}_{\text{for\_write}}$ and total write traffic for streaming access, $\text{Write}_{\text{stream}} = \text{Write}_{\text{not\_init}}$ or $\text{Write}_{\text{init}}$ based on data structure initialization.

Since streaming store operations do not cause extra read traffic for initialized write data structure (shown in Figure 25), $\text{Read}_{\text{for\_write}}$ is set to zero when Intel compilers are used. When the write array is not initialized, $\text{Write}_{\text{stream}}$ is multiplied by two to accommodate the extra page zeroing traffic (shown in Figure 25).

5.3.2 Strided Access Pattern. The strided access pattern is another common pattern. Based on the observation in Figure 24c, there are three regions. Read and write traffic formulation for each region is presented below.

5.3.2.1 Streaming Region . When the ($\text{Stride} * \text{Element}_{\text{size}}$) is smaller than the $\text{Cacheline}_{\text{size}}$, both reads and writes are the same as streaming access (region ❻ in Figure 24c). In this region (stride 1 to 16), read and write traffic are same as streaming access because the whole cache line is transferred. For this reason, total read and write traffic for this region is presented by $\text{Read}_{\text{stream}}$ and $\text{Write}_{\text{stream}}$.

5.3.2.2 No Prefetching Region . As discussed in Section 5.2.3.4, the impact of prefetching vanishes after stride 80, and hence, this is the starting point of a "no prefetching" region which is pointed by ❽ in Figure 24c. For this reason, when $(\text{Stride} * \text{Element}_{\text{size}})$ is larger than $(5 * \text{Cacheline}_{\text{size}})$, no prefetching region is considered since $(5 * \text{Cacheline}_{\text{size}})$ = stride 80 for 32-bit floating-point.

At first, write traffic is considered. If the data structure is initialized, the write traffic is decided by the cache line size and stride size. It also causes extra read traffic. This case is expressed in Eq 5.5.

$$\text{Write}_{\text{init}} \text{ or } \text{Read}_{\text{for\_write}} = \text{Write}_{\text{stream}} / \left(\frac{\text{Stride} * \text{Element}_{\text{size}}}{\text{Cacheline}_{\text{size}}} \right) \tag{5.5}$$

If the data structure is not initialized, the write traffic is decided by the $\text{Page}_{\text{size}}$. If $(\text{Stride} * \text{Element}_{\text{size}}) > \text{Page}_{\text{size}}$ then Eq 5.6 expresses write traffic, otherwise write traffic is equal to $\text{Write}_{\text{stream}}$. Read traffic is expressed as Eq 5.7.

$$\text{Write}_{\text{non\_init}} = \text{Write}_{\text{stream}} / \left(\frac{\text{Stride} * \text{Element}_{\text{size}}}{\text{Page}_{\text{size}}} \right) \tag{5.6}$$

$$\text{Read}_{\text{count}} = \text{Read}_{\text{stream}} / \left(\frac{\text{Stride} * \text{Element}_{\text{size}}}{\text{Cacheline}_{\text{size}}} \right) \tag{5.7}$$

5.3.2.3 Prefetching Zone . Only when $(\text{Stride} * \text{Element}_{\text{size}})$ is larger than the cache line and smaller than five times of the cache line, the impact of prefetching becomes visible (denoted by region ❼ which starts from stride 16 and ends at stride 80 in Figure 24c). In this region, if prefetching is disabled, write and read traffic can be expressed as Eq 5.5, Eq 5.6, and Eq 5.7. However, the main difference is observed when prefetching is enabled, and in that case, only read traffic is impacted. Intel prefetching suggests fetching an adjacent cache line and an additional cache line if all MSR bits are set. For this reason, the number of data access is multiplied by three in the prefetching zone. This is expressed in Eq 5.8. Since prefetching has no impact on write traffic, the write traffic is expressed as the non-prefetching formula given at Eq 5.5 and Eq 5.6.

$$\text{Read}_{\text{count}} = 3 * \left(\frac{\text{Element}_{\text{count}}}{\text{Stride}} \right) \tag{5.8}$$

Moreover, SK, CS, and CP show a 10% read traffic drop when compared to BW (from Figure 24c), which is considered in the model.

5.3.3 Stencil Access Pattern. Stencil access patterns are also common in scientific applications. The write operation in a stencil access pattern usually follows a sequential streaming pattern, and hence, the equations for streaming access are followed. However, read operations are complicated and need to be considered for different dimensions.

5.3.3.1 One-dimensional Stencil. In a one-dimensional stencil pattern, usually consecutive elements are accessed in each operation. Since adjacent elements can be served by cache, the read operations follows a sequential streaming pattern.

5.3.3.2 Two-dimensional Stencil. In a two-dimensional stencil, if the neighboring elements are sequentially accessed, the repetitive accesses are served by the cache. Hence, read traffic is equal to the streaming access pattern. However, if the data structure size is larger than the cache and the distance between stencil points is large, additional reads may need to be performed for bringing old elements for multiple iterations.

5.3.3.3 Three-dimensional Stencil. Like a two-dimensional stencil, if the elements are adjacent, a streaming access pattern is followed. However, if the distance between stencil points is high for a large data set, the cache size becomes a limiting factor by causing capacity misses. Usually, the operating space of a three-dimensional stencil is larger than the other stencils. For a large data set, old data may need to be brought to the cache more frequently.

5.3.4 Random Access Pattern and Empirical Factor.

The random access pattern is found in applications with irregular access [198]. The number of total access in irregular cases is expressed by $\text{Access}_{\text{random}}$. Moreover, modern CPUs introduce randomness in data reuse because of their replacement policies and the cache size since all data can not be retained in cache for further use. Therefore, LLC-DRAM traffic prediction for random access must consider the randomness derived from applications and machines. We first discuss different kinds of randomness in applications, followed by a discussion of randomness derived from machines.

5.3.4.1 Data Structure Randomness.

In data structure randomness, the reuse behavior becomes uncertain because of how the data structures are accessed, e.g., A[B[i]] (A's memory access can be random). In this case, the randomness is one-dimensional since only the location of access is random, and the total number of access, $\text{Access}_{\text{random}}$ is known. In such cases, cache reuse is non-deterministic at compile time because the access depends on another data structure at runtime. Furthermore, prefetchers may fetch some extra cache lines, which adds more uncertainty. So, machine randomness needs to be considered for this case.

5.3.4.2 Algorithmic Randomness.

The worst case of randomness is algorithmic randomness which has two dimensions, 1) randomness in the number of total access, $\text{Access}_{\text{random}}$ and 2) randomness in which locations are accessed. While the first randomness depends on the data structure size, the second kind may introduce data reuse in the cache, reducing LLC-DRAM access. Complex branching usually exists in this kind of randomness, which plays an important role in deciding LLC-DRAM traffic. An example of algorithmic randomness is searching algorithms, such as binary search. For such cases, algorithmic complexity analysis provides an upper-bound of memory access on a data structure and is considered to define $\text{Access}_{\text{random}}$. The second dimension is captured through machine randomness.

5.3.4.3 Machine Randomness and Empirical Factor. Machine randomness depends on cache size, replacement policies, and memory access location. In recent Intel processors (Since SK), replacement policies (dictating which data from cache will be replaced) are dynamically selected from a set of policies at runtime, and the policy is chosen for a given scenario is not disclosed [184]. Moreover, in the cases of algorithmic and data structure randomness (as described above), the location of access is random. So, multiple dimensions of randomness from the machine and the application make statically determining the LLC-DRAM traffic a complex problem. Moreover, the undisclosed mapping of dynamic replacement policies from Intel makes it further complicated. To the best of our knowledge, statically determining LLC-DRAM traffic in modern CPUs for irregular cases is an unsolved problem. This study does not claim to solve this problem statically; rather, it combines static analysis and empirical observation. At this point, an empirically obtained $\text{Empirical}_{\text{factor}}$ is introduced to represent machine randomness. The $\text{Empirical}_{\text{factor}}$ is calculated from memory access obtained from the dynamic analysis tool (described in Section 5.2.2) and statically obtained total access ($\text{Access}_{\text{random}}$) where $\text{Empirical}_{\text{factor}}$ = measured_access / statically_obtained_access. This ratio captures the randomness of the application and the underlying machine.

5.4 MAPredict Framework

This section describes the MAPredict framework. MAPredict statically gathers information from an application and a machine to invoke the appropriate model presented in Section 5.3 and generates a prediction for LLC-DRAM traffic. MAPredict depends on OpenARC compiler [14] for static analysis of the code and the

COMPASS [13] framework for expressing an application in the Aspen [15] domain-specific modeling language. First, an overview of OpenARC, Aspen, and COMPASS is presented. After, a description of the workflow of the MAPredict framework is provided.

5.4.1 Aspen, OpenARC, and COMPASS. Aspen (Abstract Scalable Performance Engineering Notation) [15] is a domain-specific language that provides the opportunity for analytical performance modeling in a structured fashion. Aspen's formal language and methodology provide a way to express applications and machines' characteristics abstractly (e.g., Aspen application model and machine model). Built-in or custom Aspen tools can provide various predictions, such as predicting resource counts (e.g., number of loads, stores, FLOPs, etc.), execution times, power consumption, etc. Open Accelerator Research Compiler (OpenARC) [14] is an open-source compiler framework for various directive-based programming research. It provides source-to-source translation, a desired feature for this research to create Aspen application models. COMPASS [13] is an Aspen-based performance modeling and prediction framework, which is built on OpenARC. COMPASS provides a set of Aspen directives (pragma-based) that can be used in source code. MAPredict extends COMPASS by adding new Aspen directives for enabling cache-aware memory access prediction.

5.4.2 Description of MAPredict Framework. The workflow of the MAPredict framework is shown in Figure 27. Four phases of MAPredict are described below.

5.4.2.1 Source Code Preparation Phase. The main idea of MAPredict is to prepare a source in such a way that when the preparation is done, MAPredict can statically provide memory access prediction. This one-time effort of source code

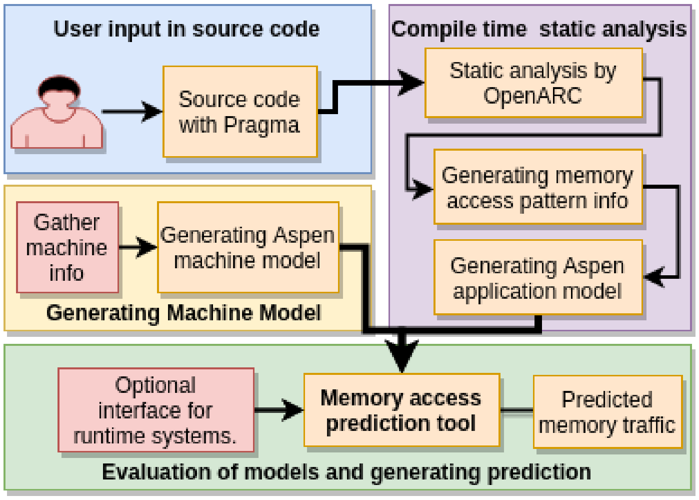

User input in source code
Source code with Pragma
Compile time static analysis
Static analysis by OpenARC
Generating memory access pattern info
Generating Aspen application model
Gather machine info
Generating Aspen machine model
Generating Machine Model
Optional interface for runtime systems.
Memory access prediction tool
Predicted memory traffic
Evaluation of models and generating prediction

Figure 27. Workflow of MAPredict framework.

preparation (i.e., phase 1) is necessary to capture the dynamic information unavailable at compile time. First, COMPASS-provided Aspen compiler directives (i.e., pragmas) are used to identify the target model region in the code for capturing information at compile time. MAPredict introduces new traits that need to be included in the directives to specify memory access patterns where necessary. Access pattern traits such as sequential streaming and strided access patterns are automatically generated; however, user input (through pragmas) is needed for stencil and random access patterns. The user inputs in the source code are input sizes of data structures and $\mathrm{Empirical}_{\mathrm{factor}}$ for random access patterns. These inputs are required because of their unavailability at compile time.

5.4.2.2 Compile-time Static Analysis phase. In phase 2, MAPredict gathers application information that is required to execute the model presented in Section 5.3. MAPredict invokes OpenARC's compile-time static analysis capability, which generates an intermediate representation of the code and captures variables, variable sizes (i.e.,$\mathrm{Element}_{\mathrm{size}}$), instruction types (load or store), FLOPs, loop information, access pattern information, machine-specific $\mathrm{Empirical}_{\mathrm{factor}}$, etc., from source code. After gathering the needed information, the source-to-source translation feature of OpenARC is invoked to generate the Aspen language's abstract application model by following Aspen's grammar [15]. An application model combines different types of statements in a graph of kernels with one or more execution blocks. An example of an application model is given in Listing 5.1 which shows load and store information of matrix multiplication. Every load and store statement is coupled with the access pattern of that data structure. In this example, the only manual input is the *param N* (data structure size). The other information in Listing 5.1 is automatically generated.

5.4.2.3 Machine Model Generation Phase. In phase 3, the machine model is generated by gathering information about the machine, following the Aspen grammar (a manual process). The machine model contains information unavailable in the application model, and is required to execute the model presented in Section 5.3. MAPredict gathers information about the microarchitecture, $\text{Cacheline}_{\text{size}}$, $\text{Page}_{\text{size}}$, prefetching status, compiler, etc., from the machine model. The machine model in Listing 5.2 shows cache-related information necessary for memory prediction. The machine models are smaller than the application models.

Listing 5.1 Application model - Matrix Multiply (partial view).

```
model matmul {
param N = 512
data a [((4*N)*N)]
kernel Matmul_openmp {
  execute [N] "block_Matmul" {
    loads [((1*sizeof_float)*N)] from b as stride(1)
    loads [((1*sizeof_float)*N)] from c as stride(N)
    stores [(1*sizeof_float)*N] to a as stride(1)
}}}
```

Listing 5.2 Machine model (partial view).

```
param bwNumCores = 32
param bwCacheCap = 40 * mega
cache bwCache {
   property capacity  [bwCacheCap]
   property cacheline [64]
   property pagesize_1 [4046]
   property pagesize_2 [2 * mega]
}
```

5.4.2.4 Prediction Generation Phase. MAPredict's prediction engine is invoked at phase 4. This invocation can be standalone, which requires passing the application and machine model to MAPredict. MAPredict invocation can also be made from a runtime system using the optional runtime invocation feature of COMPASS. When MAPredict is invoked, it traverses the call graph of the Aspen application model in a depth-first manner. In this graph, each node represents an execution block (a part of a function). MAPredict walks through every load and store statement of the application model, collects the access pattern, and evaluates the expression to obtain $\text{Element}_{\text{count}}$, $\text{Element}_{\text{size}}$, Stride, etc. Then, MAPredict uses the machine model information to invoke the appropriate prediction model to generate memory access prediction for that statement. MAPredict does this evaluation for each statement and generates a prediction for the execution block, which is recursively passed to make a kernel/function-wise prediction. When the graph traversal finishes, MAPredict provides a total memory access prediction for the application. MAPredict can provide kernel-wise memory access (shown in Listing 5.3) and execution block-wise memory access. In debug mode, it offers statement-wise detail analysis.

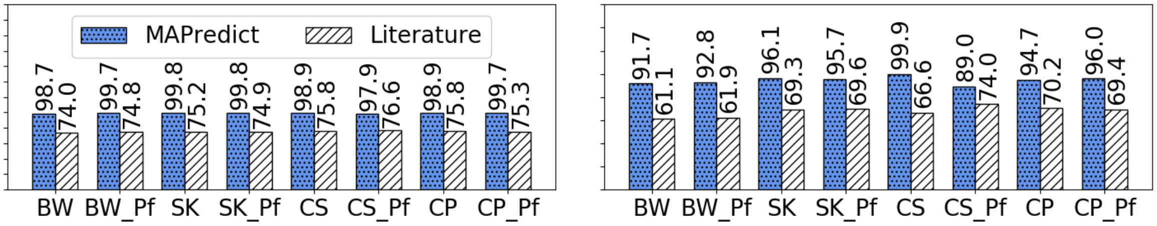
5.4.3 Identifying Randomness and $Empirical_{factor}$. MAPredict combines static and empirical approaches to address randomness. In a large codebase, identifying algorithmic and data structure randomness is challenge because randomness usually exists only in certain functions (not in the entire application). MAPredict provides a method of identifying randomness in source code. At first, the source is annotated with basic MAPredict traits (without any $\text{Empirical}_{\text{factor}}$). When MAPredict is executed, it provides function-wise memory access prediction for the application. Then the dynamic analysis tool is run on real hardware to get the same function-wise data. Comparing the results from both tools makes it apparent which functions provide low accuracy, indicating a potential source of randomness. However, a function can be large. MAPredict provides execution block-level and statement-wise detailed analysis to pin-point the randomness. After identifying, as described in Section 5.3.4.3, the $\text{Empirical}_{\text{factor}}$ is calculated by comparing the output from the dynamic analysis tool (measured value) and MAPredict (statically obtained value). Then the $\text{Empirical}_{\text{factor}}$ is annotated in the source code for that statement or execution block. When MAPredict is rerun, it uses the $\text{Empirical}_{\text{factor}}$ to generate the prediction.

Listing 5.3 MAPredict's memory access analysis.

```
< MAPredict - Kernel Level Analysis >
  Kernel Name                      Memory access
  InitStressTermsForElems :             27000000
  IntegrateStressForElems :            81000000
  ....................
  LagrangeLeapFrog :                 1567400990
Total Memory access : 1567400990
Total time (millisecond):        28.383000
```

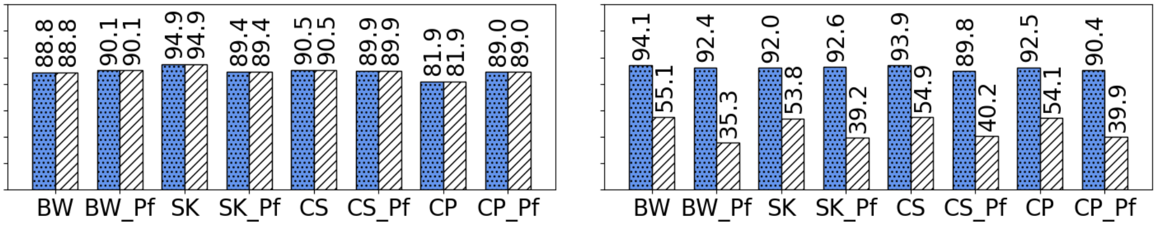
5.5 Experimental Setup

The experiment environment is discussed in this section. Processors in Table 14 were used in the experiments. The operating system of these processors is Centos-



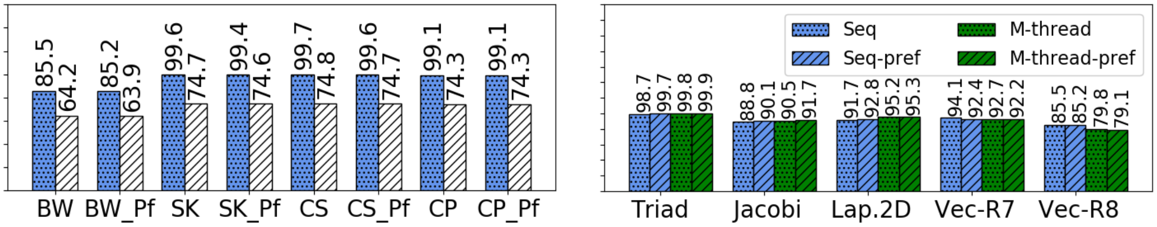
(a) Stream access : Triad.

(b) Stencil access : Laplace2D.



(c) Stencil access : Jacobi.

(d) Strided access in Region 7 : Vecmul_50.



(e) Strided access in Region 8 : Vecmul_200.

(f) Comparing single vs. multi-threaded execution.

Figure 28. Accuracy comparison of different regular access patterns. Y-axis is accuracy, and X-axis is microarchitectures with prefetching disabled and enabled. BLUE=MAPredict, WHITE=literature, and GREEN=multi-threaded execution.

7, and it supports transparent huge pages by default. The applications, along with their input sizes and access patterns, are listed in Table 15. Forty-four functions from these applications are evaluated for different input sizes, making it a total of 130 workloads. All the input sizes are bigger than the cache sizes of the machines in Table 14. GCC-9.1 and Intel-19.1 compilers are used for experimentation. For parallel execution, the OpenMP programming model is used. In the graphs, BW stands for Broadwell without prefetching, and BW_pf represents Broadwell with prefetch enabled. A similar convention is used for others.

5.5.1 Accuracy Calculation. Relative accuracy is considered, where accuracy = [100 - Absolute {(measured-predicted)/measured*100}]. The measured value is generated by the dynamic analysis tool described in Section 5.2.2. The predicted values are generated using MAPredict. Both MAPredict and the dynamic analysis tool provide output resembling Listing 5.3, making the accuracy calculation possible for each application function.

5.5.2 Comparison with Literature. The prediction accuracy of MAPredict is compared with the model from the literature [192] (referred to as "model from literature"). Even though this study [192] investigates data vulnerability, the main contribution is the analytical model for LLC-DRAM traffic prediction. Two other studies investigate memory access prediction for static analysis [191, 13]. The main reason they are not considered for comparison is the lack of a detailed analytical model with equations (detail in Table 18). Moreover, one of these research projects depends on cache simulation [191], while another depends on instruction counts without considering machine properties [13]. The study presented in [192] is selected for comparison because it provides analytical models, considers all access patterns, and does not solely rely on instruction count obtained from static analysis.

Table 15. Benchmarks.

Name	Pattern	Input sizes
STREAM Triad [160]	Sequential streaming access pattern	50M, 100M, 150M
Jacobi [14]	Stencil access pattern without initialization	67M, 268M, 1B
Laplace2D [14]	Stencil access pattern with initialization	16M, 64M, 100M
Vector Multiplication for region ❼ [14]	Strided pattern in prefetching zone	50M, 100M, 200M
Vector Multiplication for region ❽ [14]	Strided pattern in no prefetching zone	100M, 200M 400M
XSBench [199]	Algorithmic randomness	large
Lulesh [200]	Mixed patterns	15M, 27M, 64M

It considers machine properties such as cache line size, cache size, etc., for predicting memory access for different access patterns. However, it does not consider prefetchers, compilers, and changes in different microarchitectures.

5.6 Experimental Results

In this section, the accuracy of the MAPredict framework is evaluated. The evaluation is done in two steps. In the first step, the prediction accuracy of different applications with regular memory access patterns is evaluated. In the second step, irregular access patterns and a large application with mixed access patterns are investigated.

5.6.1 Regular Access Patterns. Regular access patterns are investigated for various microarchitectures, input sizes, compilers, and execution models.

5.6.1.1 Sequential Streaming Access Pattern. To evaluate the model for sequential streaming memory access pattern, the triad kernel of STREAM [160] is used. The data structure is initialized, and the size is 50M 64 bit floating-points. The total traffic, which is the summation of read and write traffic, are measured for all the microarchitectures with prefetching disabled and enabled. The prediction accuracy from MAPredict and the model from the literature [192] are compared in Figure 28a. MAPredict invoked Eq 5.3 and provided 99.1% and 99.1% average accuracy in all processors when prefetching is disabled and enabled. For the same cases, the model from literature provided 75.0%, and 75.4% average accuracy.

5.6.1.2 Stencil Memory Access Pattern. To evaluate the prediction accuracy of MAPredict for stencil pattern, two benchmark kernels are selected, Laplace2d and Jacobi [14]. Both of these kernels have a 2D stencil access pattern with adjacent stencil points. However, Laplace2D has the write array initialized, and Jacobi has the write array non-initialized. Laplace2D operates on a 4000×4000 matrix of 64-bit floating-points, whereas Jacobi operates on an 8912×8912 matrix of 32-bit floating-points. In Figure 28b the comparison for Laplace2D is shown. Since the data structure is initialized, allocating-store causes extra read, which the model from literature does not consider. MAPredict provided 95.9% and 92.5% average accuracy when prefetching is disabled and enabled, respectively. On the other hand, the model from literature provided 65.7% and 68.5% average accuracy. In Figure 28c portrays the prediction accuracy of Jacobi. Since the write data structure is not initialized, page zeroing took place. Even though the model in the literature did not consider page zeroing, the equation remained the same. For this reason, both MAPredict and the model from the literature provided same accuracy.

5.6.1.3 Strided Memory Access Pattern. To provide an evaluation of strided access pattern of prefetching region pointed by ❼ and no-prefetching region pointed ❽ in Figure 24c, vector multiplication of 100M size is used with stride 50 and 200. The stride size 50 is used with a non-initialized write array to evaluate the page zeroing effect for strided pattern. Initialized write array is considered for stride

200. For the prefetching zone, traffic becomes significantly different across different microarchitectures. Moreover, for stride 50, the whole array is written to the memory instead of one in fifty. The accuracy comparison is shown in Figure 28d. MAPredict captured the prefetching differences between different microarchitectures successfully and provided 93.3% and 91.6% average accuracy when prefetching is disabled and enabled, respectively. On the other hand, the model from literature provided 54.6% and 38.2% average accuracy since it is not equipped to predict prefetching and non-initialized cases. For stride 200, the initialized data structure causes allocating-store. The comparison is shown in Figure 28e where MAPredict provided 88.5% average accuracy in all processors. On the other hand, the model from literature provided 66.2% average accuracy.

5.6.1.4 Single vs. Multi-threaded Execution. Multi-threaded execution is compared to single-threaded execution in Figure 28f. Eight threads of BW are used for experimentation, and OpenMP from GCC is used. No significant difference is observed for sequential streaming, stencil, and strided access patterns. Other microarchitectures show a similar trend.

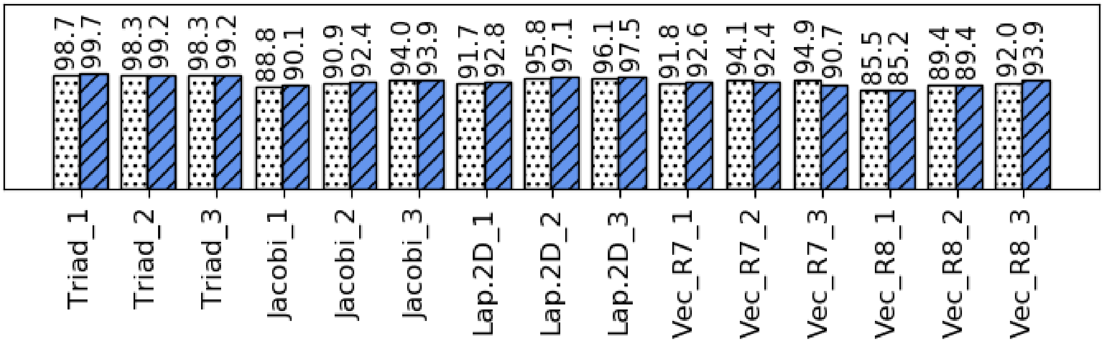


Figure 29. Accuracy of various input sizes. WHITE = prefetching disabled and BLUE = prefetching enabled.

5.6.1.5 Comparison of Different Input Sizes. MAPredict's accuracy is evaluated for different input sizes for each application with regular access pattern given in Table 15. Triad is tested with array sizes of $50M$, $100M$, and $150M$. Matrix sizes for Jacobi are 8192×8192, and 16384×16384, and 32768×32768. Laplace2D is tested with 4000×4000, 8000×800, and 1000×1000 matrix sizes. Strided vector multiplication is tested with vector sizes of $50M$, $100M$, and $200M$ for prefetching region and $100M$, $200M$, and $400M$ for no prefetching region. Prediction accuracy of each data set for prefetching enabled and disabled cases are presented in Figure 29. The accuracy of different input sizes demonstrates that MAPredict's provides consistent accuracy for varied input sizes. The BW processor is used for this evaluation, and a similar trend is observed for others.

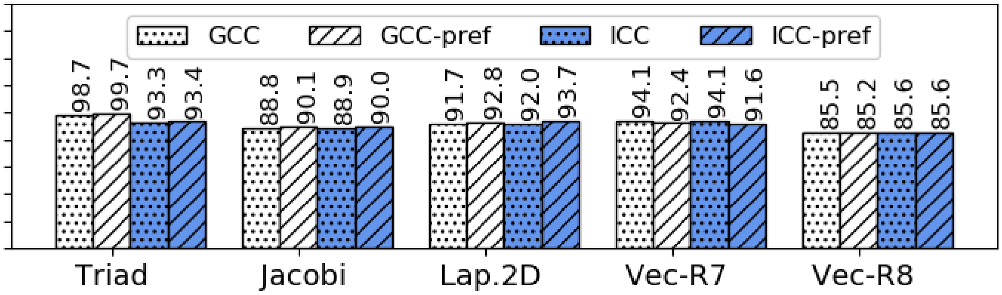


Figure 30. Accuracy comparison of GCC and Intel compiler.

5.6.1.6 Comparison of GCC vs. Intel Compiler. The impact of different compilers is portrayed in Figure 30. When ICC is used, the most significant difference in traffic is observed for Triad because of the streaming store operation by Intel compiler (observation at ❾ in Figure 25). Compiler information is made available to MAPredict through the machine model, and MAPredict invokes the analytical model for ICC. For this reason, high accuracy is observed for Triad in Figure 30. However, this streaming only resolves at compile time. For example, if a

Table 16. Analysis of Lulesh (selected functions) for single threaded execution. d1=data size 1 without prefetching, p-d1=with prefetching.

Function name (Shortened)	**Access Pattern**	**MAP redict**	**TAU PAPI**	**Single thread - 3 data sizes**					
				d1	**p-d1**	**d2**	**p-d2**	**d3**	**p-d3**
IntegrateStressF.Elm	St,S	81M	83M	99.0	97.4	88.5	88.9	91.8	91.7
CFBHour.ForceF.Elm	St,S	239M	241M	96.9	99.1	97.0	96.5	82.2	82.6
CHourg.Cont.F.Elm	St,I,DR	604M	647M	92.8	93.1	93.0	92.7	76.2	77.9
LagrangeNodal	All	824M	874M	94.2	94.2	95.3	95.1	85.4	86.5
CKinematicsF.Elm	S,St	99M	100M	98.7	99.7	96.0	96.6	98.3	98.7
CLagrangeElements	St,S,I	126M	130M	98.3	97.5	99.7	99.9	98.5	98.3
CMon.QGrad.F.Elm	S,St	99M	105M	95.2	94.4	95.6	95.6	94.6	94.5
CMon.QReg.F.Elm	DR,N,S	141M	150M	94.9	94.6	95.4	93.2	94.3	93.2
CEnergyF.Elm	S	249M	261M	97.7	95.6	94.0	98.6	99.9	93.4
EvalEOSF.Elm	DR,S	429M	451M	99.1	95.1	95.7	97.9	98.4	93.0
UpdateVol.F.Elm	S	10M	10M	99.9	99.9	99.7	99.9	99.9	99.9
LagrangeElements	All	824M	869M	98.4	95.0	99.3	96.7	96.9	93.7
LagrangeLeapFrog	All	1.6B	1.7B	95.0	93.0	96.6	94.3	95.8	99.2

loop is in the following format $for(i = 0; i < n; i = i + x)\ where\ the\ value\ of\ x\ is$ 1, streaming store is not activated since x is a variable. Other applications do not perform streaming stores and hence provided no difference. Figure 30 shows the data for BW; others demonstrate similar behavior.

5.6.2 Irregular Access and Large Application with Mixed Patterns.

To evaluate MAPredict's capability of combining static and dynamic information for irregular access and mixed patterns, two applications, XSBench and Lulesh, are considered.

5.6.2.1 Algorithmic Randomness. XSBench [199] is a proxy application based on the Monte Carlo transport algorithm. It calculates the macroscopic neutron cross-section by randomly searching for energy and material. The energy search (grid_search) for each lookup is done by employing a binary search on a unionized energy grid, an example of algorithmic randomness (total access = $\text{Access}_{\text{random}}$ *

Table 17. Analysis of Lulesh (selected functions) for multi-threaded execution. d1=data size 1 without prefetching, p-d1=with prefetching.

Function name (Shortened)	**Multi-threaded - 3 data sizes**					
	d1	**p-d1**	**d2**	**p-d2**	**d3**	**p-d3**
IntegrateStressF.Elm	86.3	83.5	86.9	86.0	86.0	84.9
CFBHour.ForceF.Elm	96.9	94.7	96.6	96.7	72.3	72.6
CHourg.Cont.F.Elm	77.0	75.7	77.1	76.6	80.0	82.5
LagrangeNodal	79.7	78.5	79.9	79.4	90.6	92.4
CKinematicsF.Elm	89.8	88.6	90.0	89.7	89.9	89.5
CLagrangeElements	89.5	88.6	89.8	89.6	89.5	89.2
CMon.QGrad.F.Elm	81.8	81.0	82.0	81.6	82.0	81.5
CMon.QReg.F.Elm	95.0	99.6	95.5	99.1	94.3	100
CEnergyF.Elm	89.7	84.2	91.3	85.5	88.6	83.2
EvalEOSF.Elm	86.8	82.8	87.8	83.6	85.6	81.7
UpdateVol.F.Elm	99.3	99.4	98.9	98.6	99.9	99.9
LagrangeElements	86.8	85.7	87.6	86.3	86.1	85.2
LagrangeLeapFrog	82.1	80.7	82.6	81.5	95	93.4

$\text{Empirical}_{\text{factor}}$). As discussed in Section 5.3.4.2, both the number of access and the location of access are random. Since grid_search follows a binary search, algorithm complexity (log$n$) is used to measure $\text{Access}_{\text{random}}$. The $\text{Empirical}_{\text{factor}}$ is calculated for BW with prefetching disabled and used for all other processors ($\text{Empirical}_{\text{factor}}$ = the ratio of measured value and $\text{Access}_{\text{random}}$). The predicted value is then compared to the average of five measurements (up to 5% standard deviation) for accuracy calculation. The blue bars in Figure 31 show that only BW provided high accuracy when prefetching is disabled and hence demonstrating the need for machine-specific $\text{Empirical}_{\text{factor}}$. When individual $\text{Empirical}_{\text{factor}}$ is used, the accuracy of each processor improved (yellow bar). MAPredict provides the option to include multiple machine specific $\text{Empirical}_{\text{factor}}$ for a single statement; thus, a single source code can be updated for multiple machines. The method presented in [192] does not calculate the total number of random access rather focuses on the access location, which makes the

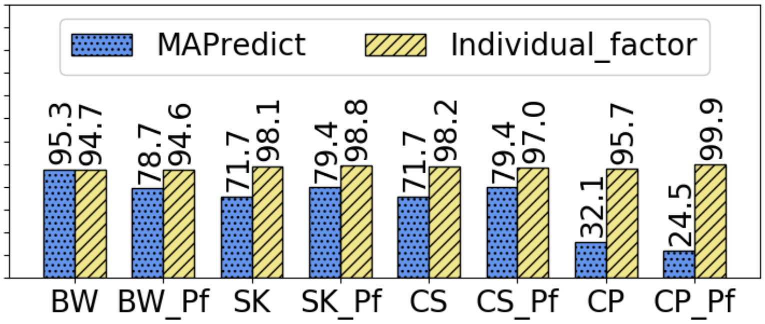


Figure 31. Accuracy of algorithmic randomness for XSBench.

comparison irrelevant. Algorithmic randomness is an extreme case, and it is only present in a certain function. For this reason, a large application with mixed patterns is investigated next for different input sizes and execution models.

5.6.2.2 A Large Application with Mixed Patterns: Lulesh. To demonstrate that MAPredict can work with a large application with different memory access patterns (including random), Lulesh [200] is considered. Lulesh is a well-known app with different memory access patterns for a 3D mesh data structure. It has 38 functions with a complex call graph and 4474 lines of code, making it a large and complex example. Moreover, functions have multiple access patterns. Three large data structure sizes ($250 \times 250 \times 250$, $300 \times 300 \times 300$, and $400 \times 400 \times 400$) are used. The SK machine is selected for experimentation because it has the smallest cache and hence, stresses the capability of MAPredict by increasing the probability of machine randomness.

5.6.2.3 Function Categorization of Lulesh. Out of 38 functions in Lulesh, 24 functions provide significant memory transactions (> 1 Million LLC-DRAM transactions for the data size of $300 \times 300 \times 300$). The 24 memory intensive functions have different memory access patterns. Most memory intensive functions are shown at Tables 16 and 17 , where column-2 in Table 16 shows access patterns. Here,

St=stencil (eight-point non-adjacent 3D stencil), S=stream, DR=data structure randomness, and I=non-initialized arrays, N=nested randomness (data structure randomness with branches), and All=all the above patterns.

5.6.2.4 $Empirical_{factor}$. Lulesh has data structure randomness in three functions. The $\mathrm{Empirical}_{\mathrm{factor}}$ is calculated by comparing the static and dynamic data to address this randomness (a one-time effort). So, a total of three $\mathrm{Empirical}_{\mathrm{factor}}$ are used in three functions (out of 38).

5.6.2.5 Traffic: Number of LLC-DRAM Transactions. Columns 3 and 4 in Table 16 show the LLC-DRAM transaction (M=Million and B=Billion) obtained for MAPredict and TAU+PAPI (dynamic analysis tool). The last function, which is the parent of all functions, shows a total of 1.7 Billion LLC-DRAM transactions. However, for the largest data size, Lulesh exhibits 3.5 Billion LLC-DRAM transactions.

5.6.2.6 Scaling and Accuracy for Lulesh. Scaling in terms of input sizes and the number of threads provides a measure of success for one-time calculation of the $\mathrm{Empirical}_{\mathrm{factor}}$ in this complex case. Column 5-10 of Table 16 show the accuracy of different functions for different data sizes for prefetching enabled and disabled cases. Since some functions are parents to other functions and the last function is the parent to all (total traffic), inaccuracy in one function impacts the overall accuracy. MAPredict showed more than 93% accuracy for all data sizes, which demonstrates the model and $Empirical_{factor}$ scaled well in terms of input size. For the multi-threaded case, maximum threads (10 threads) in a socket are used. Multi socket runs are avoided because memory-intensive kernels provide worse performance when different NUMA nodes are used. For example, Lulesh showed a 48% performance drop when ten threads in one NUMA node and 20 threads in two NUMA nodes runs are compared. Columns 2-7 in Table 17 show the accuracy of different data sizes in multi-threaded experiments. A drop in accuracy is observed in the multi-threaded results since multiple threads occupy the cache in a complex access scenario. However, two data sizes showed more than 80% overall accuracy, and one data size provided more than overall 93% accuracy.

5.6.3 Discussion. For regular access patterns, MAPredict's static analysis provides higher accuracy than the literature model and can handle different input sizes, microarchitectures, cache sizes, compilers, and execution models. However, MAPredict requires empirical observation for irregular patterns.

5.6.3.1 Overhead of MAPredict. One of the objectives of MAPredict is to make it usable from runtime systems for fast decisions. The evaluation of Lulesh takes 28.3 milliseconds (38 functions), averaging to less than a millisecond per function. For source code preparation, 249 lines of Aspen directives are used for 4474 lines of code, which is 5.5% source code overhead. 79 directives are for enabling MAPredict.

5.6.3.2 Usability of $Empirical_{factor}$***.*** The calculation of $\text{Empirical}_{\text{factor}}$ for irregular accesses is needed in complex applications (i.e., Lulesh). However, the $\text{Empirical}_{\text{factor}}$ calculation is a one-time effort. Once calculated, it becomes a part of the source code and can provide prediction statically. Moreover, randomness usually occurs only in a small portion of an application (regular access patterns are more commonly found). So, the $\text{Empirical}_{\text{factor}}$ calculation is needed only where randomness exists.

5.7 Related Works

Related works presented in this section are divided into two categories. The first category shows the literature related to memory access prediction and static analysis. The second category shows the importance of understanding Intel processors.

5.7.1 Memory Access Prediction.

Several studies investigated memory access patterns to make a reasonable prediction. Yu et al. [192] used analytical models of different memory access patterns. In Tuyere [191], Peng et al. used data-centric abstractions in an analytical model to predict memory traffic for different memory technologies. Application models in these aforementioned studies are manually prepared. Moreover, MAPredict goes beyond these wrok by including the impact of page size, prefetchers and compilers in machine model. Moreover, Tuyere framework showed the benefit of analytical models over trace-based or cycle accurate simulator (such as Ramulator [201], DRAMSim [202]) both in terms of time and space. MAPredict further improves upon Tuyere by providing prediction in 1-3 milliseconds per function. Allen et al. [203] investigated the impact of two memory access patterns on GPUs. Some previous works used load and store instruction counts to measure memory access and used that count to predict performance(e.g., COMPASS by Lee at al. [13]). Compile-time static analysis tools, such as Cetus [204], OpenARC [14], and Caascade [205] are also used to measure instruction counts at compile time and can provide a prediction. MAPredict does not solely depend on instruction counts; it captures the impact of cache hierarchy through analytical models. In contrast to MAPredict's near-accurate prediction, analytical models such as Roofline Model [178] and Gable [172] provide an upper bound for a system.

5.7.2 Understanding Intel processors.

Some studies delved into Intel processors to understand their performance by using benchmarks. Using the Intel advisor tool, Marques et al. [179] analyzed the performance of benchmark applications to understand and improve cache performance. Alappat et al. [184] investigated Intel BW and CS processors to understand the cache behavior using the likwid tool suite [206]. Hammond et al. investigated the Intel SK processor [190] by running

Table 18. Comparison with other works. A=All, P=Partial.

Studies by	**Static analysis**	**Analytical model**	**Access patterns**	**Different microarchitecture**	**Different compilers**	**Multi-threaded Execution**	**Prefetchers**
Peng et al.[191]	✓	✗	A	✗	✗	✓	✗
Yu et al. [192]	✓	✓	A	✗	✗	✗	✗
Monil et al. [16]	✗	✗	P	✓	✗	✗	✓
Lee at al. [13]	✓	✗	P	✓	✗	✓	✗
Marques et al. [179]	✗	✗	P	✗	✗	✓	✗(disabled)
Alappat et al. [184]	✗	✗	P	✓	✓	✓	✓
Hammond et al. [190]	✗	✗	P	✓	✗	✓	✗
Molka et al. [187]	✗	✗	P	✓	✗	✓	✗(disabled)
MAPredict	✓	✓	A	✓	✓	✓	✓

different HPC benchmarks. Hofmann et al. also investigated different Intel processors to analyze core and chip-level features [189, 188]. Park et al. also investigated the performance of different Intel microarchitectures and optimized HPC benchmarks to perform better. Molka et al. [187] used a micro-benchmark framework to analyze the main memory and cache performance of Intel Sandy bridge microarchitecture (also AMD Bulldozer processors). Performance evaluation using benchmarks is also done by Saini et al. for Ivy Bridge, Haswell, and Broadwell microarchitectures [185, 186]. These works investigated Intel microarchitectures using benchmarks but did not develop strategies for predicting memory traffic.

5.7.3 Novelty in MAPredict.

Table 18 shows the comparison of MAPredict with other literature. The first four rows represent the study of memory access patterns and static analysis. The next four rows represent studies that are focused on an in-depth understanding of Intel microarchitectures. MAPredict bridges these two areas and provides a unique framework that can provide memory access prediction in modern CPUs.

5.8 Summary

This chapter presents the MAPredict framework, which predicts memory traffic for Intel processors. MAPredict is a prerequisite for MEPHESTO presented in Chapter IV. This study investigates the interplay between an application's memory access pattern and Intel micro-architectures' cache hierarchy. Based on the observation from Intel processors, an analytical model is derived that takes memory access patterns of an application, properties of a processor, and choice of the compiler into consideration. MAPredict generates an application model for a given application through compile-time analysis. The application is combined with a target machine model to synthesize the appropriate analytical model to predict LLC-DRAM traffic. Through experimentation with benchmarks on processors from Intel Broadwell, Skylake, Cascade Lake, and Cooper Lake micro-architectures, the analytical model's validity is verified by achieving average accuracy of 99% for streaming, 91% for strided, and 92% for stencil patterns. MAPredict also facilitates providing hints in the source code to capture dynamic information and randomness either from the application or machine to obtain better accuracy. By combining static and empirical approaches, MAPredict achieved up to 97% average accuracy on different micro-architectures for random access patterns.

By providing the means to predict LLC-DRAM traffic in modern CPUs, MAPredict solves a important piece of the puzzle for MEPHESTO. However, a similar prediction is needed for GPUs as well to implement MEPHESTO's dynamic task placement in a runtime system for heterogeneous system. For this reason, the next chapter strives to understand and model LLC-DRAM traffic in GPUs.

CHAPTER VI

UNDERSTANDING AND MODELING LLC-MEMORY TRAFFIC IN GPUS

This chapter contains unpublished material with co-authorship. All of the presented research in this chapter was conducted as a collaboration between the University of Oregon and Oak Ridge National Laboratory (ORNL). Research presented in this chapter is accepted at RSDHA 2021 [207] workshop at SC 2021. While working on this research, I received regular guidance from Dr. Seyong Lee. I also received high-level guidance from Dr. Allen Malony and Dr. Jeffrey Vetter. I did all the experiments, writing, and data collection. Dr. Lee helped with proofreading the submission.

6.1 Introduction and Motivation

This chapter investigates LLC-DRAM traffic in GPUs so that MAPredict can provide prediction for both CPUs and GPUs (corresponding to Research Question 5 — **RQ5**: Can we capitalize the understanding of the CPUs to explain and model the LLC-DRAM traffic for GPUs?). From a hardware design perspective, latency-focused CPUs are vastly different than throughput-focused GPUs. Moreover, instruction set architectures, programming models, and execution models also differ. For this reason, finding similarities and identifying dissimilarities among CPUs and GPUs from different manufacturers can provide a better way of generating analytical models.

Where the previous chapter focused on Intel CPUs, this chapter focuses on GPUs from NVIDIA and AMD. The intense race between GPU manufacturers to provide more computational power is propelling frequent releases of new, more capable, and more complex GPUs. For example, NVIDIA's recent Ampere GPUs (A100) [208] were countered by AMD's Instinct GPUs (MI100) [209]. Both host more than 30 GB of device memory, and NVIDIA's Ampere GPUs have more than double the device memory of the previous Volta GPUs. To increase the potential of machine-learning applications and take advantage of this growing discipline, both manufacturers have added dedicated cores—NVIDIA Tensor Cores and AMD Matrix Cores—to accelerate machine-learning calculations. This competition is reflected in large-scale supercomputers as well. For example, Frontier, the upcoming exascale machine from Oak Ridge National Laboratory (ORNL), will use AMD's Instinct GPUs, whereas the new Perlmutter machine, being installed at the National Energy Research Scientific Computing Center, will use NVIDIA A100 GPUs. Due to the current trend, this chapter considers different generations of NVIDIA and AMD GPUs.

Because CPUs have existed much longer than GPUs, the research community has a significantly wider breadth of knowledge and understanding of CPU architectures than the comparatively new GPU architectures. Although previous studies have investigated the impact of memory-access patterns on CPUs and GPUs, not many studies have compared the LLC-memory traffic patterns between CPUs and GPUs [210, 16, 203]. Investigating similarities would provide opportunities to apply similar optimization techniques, and finding the dissimilarities would provide a better understanding of the differences between CPU and GPU memory traffic patterns.

This study adopts an experimental evaluation approach to explore and understand the impact of memory-access patterns on different GPUs from NVIDIA and AMD and attempts to find the similarities and dissimilarities with Intel CPUs. Using the two most common memory-access patterns (i.e., sequential streaming and strided access patterns), this study reveals the factors that decide LLC-memory transactions. Finally, this study formulates analytical models that can be included in MAPredict. The following contributions are described:

- Common factors in the cache hierarchy that trigger an LLC-memory transaction;
- Strategies to measure LLC-memory traffic by using NVIDIA's Nsight Compute and AMD's ROCm profiler;
- Investigation and comparison of three NVIDIA GPUs (i.e., P100, V100, A100) with Intel CPUs for two memory access patterns;
- Investigation and comparison of three AMD GPUs (i.e., MI50, MI60, MI100) to explore similarities and dissimilarities between Intel CPUs and NVIDIA GPUs; and
- A proof concept for predicting LLC-memory traffic of NVIDIA and AMD GPUs.

6.2 Methodologies for GPU Access Investigation

This section provides the methodologies used to investigate LLC-memory traffic for sequential streaming and strided access patterns in NVIDIA and AMD GPUs. Here, *memory* refers to the device memory of the GPUs. First, detailed information about the NVIDIA and AMD GPUs used in this study is presented. Then, the application that exhibits sequential streaming and strided access patterns is presented (i.e., the strided *vecMul* function for GPUs). Finally, the strategies for measuring the LLC-memory traffic for different NVIDIA and AMD GPUs are discussed.

6.2.1 NVIDIA and AMD GPUs. NVIDIA and AMD have been releasing different GPUs for decades. With the rejuvenation of machine learning, GPUs are garnering even more attention, and an intense race between manufacturers to provide better performance for diverse workloads is under way. Current NVIDIA and AMD GPUs include tensor/matrix cores capable of performing fused matrix multiply and accumulate operations to facilitate machine-learning workloads. This study considers three recent GPUs from both NVIDIA and AMD. Table 19 and 20 show an overview of hardware information for three NVIDIA Tesla and AMD Radeon GPUs, of which the most recent are NVIDIA's A100 and AMD's MI100 [211, 212, 208, 209, 213]. The machines used in this study are part of ORNL's Experimental Computing Laboratory (ExCL) [214] and the Oregon Advanced Computing Institute for Science and Society (OACISS) [215].

6.2.2 Strided Vector Multiplication for CUDA and ROCm. To investigate sequential streaming and strided access patterns, the vector multiplication application used for the CPU study in Section 5.2 in Chapter V is modified for

Table 19. NVIDIA GPUs.

Item description	**NVIDIA Tesla GPUs**		
	Pascal: P100	***Volta: V100***	***Ampere: A100***
Release year	2016	2017	2020
Architecture	Pascal	Volta	Ampere
Number of SMs/CUs	56	80	108
Number of cores	3,584	5,120	6,912
Peak performance FP32	13.41 TFLOPS	14.13 TFLOPS	19.49 TFLOPS
Peak performance FP64	6.705 TFLOPS	7.066 TFLOPS	9.746 TFLOPS
Tensor/Matrix cores	No	Yes (640)	Yes (432)
Device memory size	16 GB HBM2	16 GB HBM2	40 GB HBM2e
Memory bus	4,096 bit	4,096 bit	5,120 bit
Bandwidth	732.2 GB/s	900 GB/s	1,555 GB/s
L1 cache per SM/CU	24 KB	128 KB	192 KB
L2 cache size	4 MB	6 MB	40 MB
Cache line size	32 bytes	32 bytes	32 bytes
Warp/Wavefront size	32 threads	32 threads	32 threads
Compiler	nvcc	nvcc	nvcc
Profiler	nvprof	nvprof	Nsight compute
Software stack	CUDA-11.0	CUDA-11.2	CUDA-11.2
Machine name	Oswald01	Leconte	Illyad
Facility	ExCL	ExCL	OACISS

the CUDA and ROCm platforms [16]. The basic difference between multithreaded CPU code and GPU code is presented in Figure 32. Programming models, such as OpenMP, divide a data structure among the available threads in a CPU, where each thread continuously (based on stride size) executes the array indices. Multithreaded execution for two threads is shown in Figure 32. However, a GPU decomposes the total computation in blocks consumed by warps/wavefronts in the SM/CU. For this reason, blocks are usually chosen as multiples of the warp/wavefront size. Each warp/wavefront employs 32 (NVIDIA) and 64 (AMD) threads for computation.

To make the comparison straightforward, the GPU execution shown in Figure 32 displays a hypothetical situation in which each warp has only two threads. So, the

Table 20. AMD GPUs.

Item description	**AMD Radeon Instinct GPUs**		
	MI50	***MI60***	***MI100***
Release year	2018	2018	2020
Architecture	GCN 5.1	GCN 5.1	CDNA 1.0
Number of SMs/CUs	60	64	120
Number of cores	3,840	4,096	7,680
Peak performance FP32	13.41 TFLOPS	14.75 TFLOPS	23.07 TFLOPS
Peak performance FP64	6.705 TFLOPS	7.373 TFLOPS	11.54 TFLOPS
Tensor/Matrix cores	No	No	Yes (each CU)
Device memory size	16 GB HBM2	32 GB HBM2	32 GB HBM
Memory bus	4,096 bit	4,096 bit	4,096 bit
Bandwidth	1,024 GB/s	1,024 GB/s	1,229 GB/s
L1 cache per SM/CU	16 KB	16 KB	16 KB
L2 cache size	4 MB	4 MB	8 MB
Cache line size	64 bytes	64 bytes	64 bytes
Warp/Wavefront size	64 threads	64 threads	64 threads
Compiler	hipcc	hipcc	hipcc
Profiler	rocprof	rocprof	rocprof
Software stack	ROCm-3.9.0	ROCm-4.3.0	ROCm-4.3.0
Machine name	Gilgamesh	Explorer	Cousteau
Facility	OACISS	ExCL	ExCL

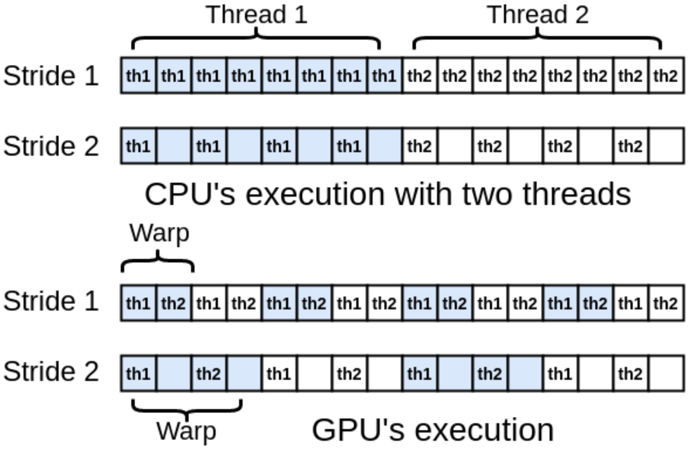


Figure 32. A simplistic representation of execution on a CPU and GPU for strided access. Here, two threads are considered for the CPU. For the GPU, only two threads per warp/wavefront (hypothetical) are considered to show the difference. However, the warp/wavefront sizes are 32 threads for NVIDIA GPUs and 64 threads for AMD GPUs.

computation is done on hardware threads in a multithreaded execution on CPUs (considering OpenMP), where the same thread processes the neighboring elements, which increases cache locality. On the other hand, warps/wavefronts consume thread blocks in the GPU, where different threads access adjacent data (depending on stride size). Because warps/wavefronts are scheduled in the SM/CU, having the same warp threads working on neighboring data provides the best cache locality and can take advantage of memory coalescing in the L1 cache. For this reason, the vector multiplication application is modified in such a way that neighboring threads in a warp/wavefront execute the adjacent elements (a standard practice [203]). Because this study focuses on the LLC-memory (L2-global memory for GPUs), the impact on shared memory at L1 is not considered.

The modified CUDA code is shown in Listing 6.1, in which neighboring threads execute neighboring elements for various strides. When the stride is 1, *vecMul* exhibits a standard sequential stream access pattern like the STREAM benchmark [160]. Note that only the vectors read by *vecMul* are allocated and initialized before transferring to the device (h_a and h_b). The write array is only allocated. While measuring the traffic, only the *vecMul* function is considered. The *hipify* tool from the ROCm software stack is used to convert the CUDA code to HIP code. Because the code is short, the conversion using *hipify* compiled without any error. For this study, the *nvcc* (cuda-11.0) and *hipcc* (rocm-3.9.0 and rocm-4.3.0) compilers are used.

Listing 6.1 Strided vector multiplication in CUDA.

```
__global__ void vecMul(float *a, float *b, float *c, int n, int stride){
    // Get our global thread ID
    int id = blockIdx.x * blockDim.x + threadIdx.x;
    // Ensuring strided access and boundary checking
    if (id*stride < n)
        c[id*stride] = a[id*stride] * b[id*stride];
}
int main( int argc, char* argv[] ){
    int n = 100000000;
    // host and device data structures
    float *h_a, *h_b, *h_c, *d_a, *d_b, *d_c;
    size_t total_size = n*sizeof(float);
    // Allocate the vectors in the host
    h_a, h_b, h_c = allocate_host(total_size);
    // Initialize a and b vectors int the host
    for( int i = 0; i < n; i++ ) {
        h_a[i] = sin(i); h_b[i] = cos(i);
    }
    // Allocate the vectors in the device
    d_a, d_b, d_c = allocate_device(total_size);
    // Initiate host to device synchronous transfer
    copy_to_device(h_a, d_a, h_b, d_b)
    Start_memory_counters(); // done automatically by profiler
    vecMul<<<gridSize, blockSize>>>(d_a, d_b, d_c, n, stride);
    Stop_memory_counters(); // done by profiler
    // Initiate device to host synchronous transfer
    copy_from_device(h_c, d_c);
    // free host and device memory
    free(h_a, h_b, h_c, d_a, d_b, d_c);
}
```

6.2.3 Measuring LLC-memory Traffic in GPUs.

LLC-memory traffic is measured for the GPUs listed in Table 19 and 20to investigate the impact of memory-access patterns. NVIDIA's CUDA software stack provides *nvprof* and Nsight Compute (*ncu*) for profiling GPU kernels. AMD's ROCm software stack provides the ROCm profiler (*rocprof*) for profiling the AMD GPUs. These tools are used in this study to gather LLC-memory traffic for the strided *vecMul* function.

6.2.3.1 Using nvprof and ncu for NVIDIA GPUs.

The *nvprof* profiling tool provides the functionality to measure hardware metrics for GPU kernels (*vecMul* in this case). Unlike Intel CPUs, where uncore counters are read from the integrated memory controller, *nvprof* provides a direct metric for LLC-memory byte transfer. The name of the metrics are *dram_read_bytes* and *dram_write_bytes*. Here, *dram* indicates the device memory. These metrics are specified in the command line, whereas the executable is attached to *nvprof*. In some cases, the kernels are rerun multiple times to provide an accurate count of the LLC-memory bytes transferred. LLC-memory traffic for P100 and V100 is measured using *nvprof*. However, support for *nvprof* was discontinued in CUDA Compute Capability 8.0 and onward. For this reason, Nsight Compute (*ncu*) has been used for A100, in which the metric names are *dram__bytes_read* and *dram__bytes_write* (one extra underline in the counter name).

6.2.3.2 Using rocprof for AMD GPUs.

AMD's ROCm profiler (*rocprof*) is used in this study to measure LLC-memory traffic for MI50, MI60, and MI100 GPUs. Like *nvprof*, the *rocprof* command-line tool can measure basic hardware counters and derived metrics. Unlike *nvprof*, counters/metrics are specified in a file provided in the command line along with the executable. The value of the hardware counters/metrics is then generated and stored in a CSV file. Two derived metrics represent the LLC-memory traffic for AMD Instinct GPUs: *FETCH_SIZE* and *WRITE_SIZE*. These metrics provide the traffic as KiB, which is converted to MB for an even comparison.

6.2.3.3 Scripts for Data Collection. To gather the data seamlessly, scripts are prepared both for NVIDIA and AMD GPUs to execute and collect the LLC-memory traffic. These scripts vary the stride size and collect the traffic, which is then plotted for analysis.

6.3 Understanding NVIDIA and AMD GPUs

This section explores the similarities and dissimilarities of LLC-memory traffic between Intel CPUs and NVIDIA and AMD GPUs. LLC-memory traffic is measured on the GPUs using the methodologies presented in Section 6.2. The memory traffic is then compared with the memory traffic from the Intel CPUs, as described in Section 5.2 of Chapter V. Three NVIDIA GPUs are investigated, followed by an exploration of AMD GPUs. All graphs presented in this section show the stride along the x-axis, and the read/write traffic in MB along the y-axis. Through the course of this investigation, some key observations and hypotheses are formulated.

6.3.1 LLC-memory Traffic of NVIDIA GPUs. Three NVIDIA GPUs are investigated: Pascal (P100), Volta (V100), and Ampere (A100).

6.3.1.1 Similarities between P100 GPU and Skylake CPU. LLC-memory traffic for strided *vecMul* on the P100 is presented in Figure 33. The blue lines represent the data for the P100. The P100's traffic trend is very similar to the CPU when the write data structure is initialized and compiled using the Intel compiler. Notably, *vecMul* for the GPU does not have the write array initialized. Even though the write array is uninitialized, the P100 shows a trend similar to the

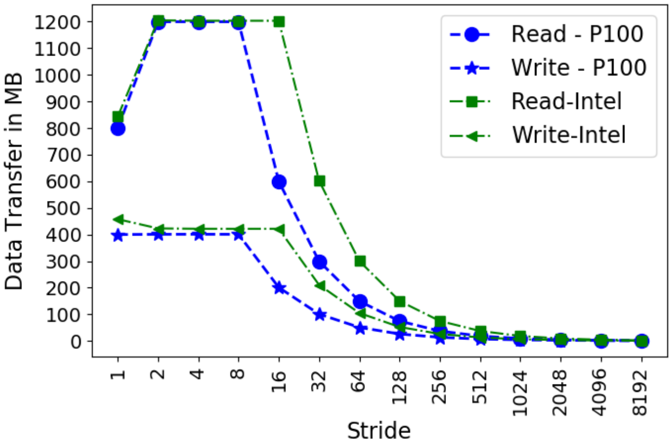


Figure 33. Read and write traffic for the P100 GPU. The P100 traffic follows a trend similar to the Skylake CPUs when the write data structure is initialized, and the Intel compiler is used.

CPU with allocating store. The CPU traffic is presented using green lines. In this case, the primary difference is observed for the strides of 8 and 16. This difference is caused by the cache line sizes in the NVIDIA GPU (32 bytes) vs. the Intel CPU (64 bytes). The following observations can be made from the results in Figure 33.

Observation-1 Like the Intel compiler, the *nvcc* compiler implements the streaming-store operation when the stride is 1.

Observation-2 The NVIDIA GPU performs the allocating store even when the data structure is not initialized.

Observation-3 The NVIDIA GPU's write traffic is the same as the write traffic for the initialized case of the Skylake CPU.

6.3.1.2 Similarities between P100, V100, and A100. Figure 34 depicts memory traffic from all NVIDIA GPUs studied here. As one can see, the write traffic for all GPUs is the same. Moreover, all GPUs implement streaming store and allocating store. Therefore, Observations 1, 2, and 3 are also applicable for V100 and A100. For both read and write traffic, V100 and A100 show little to

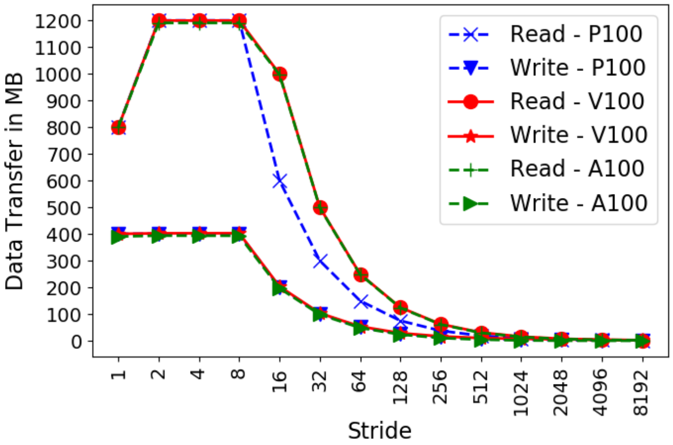


Figure 34. Read and write traffic for all NVIDIA GPUs.

no difference. The main difference between P100 and later GPUs is observed for the read traffic when the stride size is larger than the cache line size (stride of 8). On average, the read traffic for V100 and A100 is 1.6× higher than for P100. When the stride size is larger than the cache line size, the difference between P100 and A100 (also V100) shows a striking similarity to the prefetching-enabled and disabled cases for Intel CPUs presented in Figure 24c in Section 5.2 of Chapter V. This difference suggests that a major change was applied to the prefetchers of the Volta architecture and onward. Therefore, the following observation and hypothesis can be made.

Observation-4 When the stride size is larger than the cache line on V100 and A100 GPUs, the read traffic shows a similar pattern to the prefetching-enabled Skylake CPU. Traffic is about 1.6× higher than on the P100 and on the prefetching-disabled Skylake CPU.

Hypothesis-1 From the Volta GPU architecture onward, NVIDIA GPUs implement an Intel CPU–like prefetching mechanism.

6.3.1.3 A Tailored Graph to Realize the Similarities. A hypothetical scenario is considered in which the cache line size of the Skylake CPU is 32 bytes

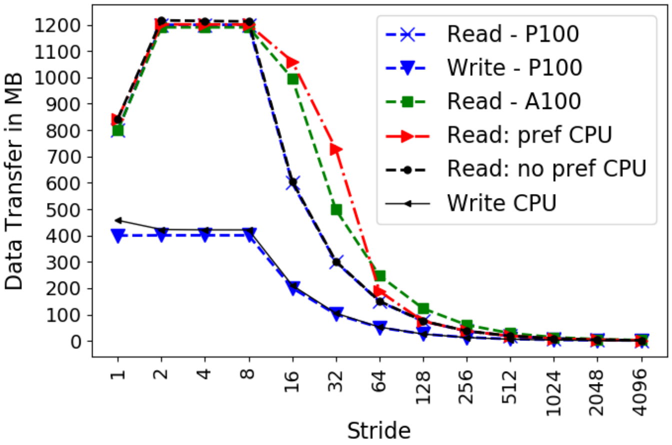


Figure 35. Tailored comparison between NVIDIA GPUs and Intel CPU.

instead of 64 bytes to demonstrate the similarities between the Intel CPU and the NVIDIA GPUs. Even though the consideration is hypothetical, the data presented in Figure 35 are actual data. To prepare Figure 35, the read and write traffic for stride 16 is removed only for the CPU and shifted to the left for all strides greater than 16. This conversion shows the CPU data for a 32-byte cache line because one read or write would fetch/store 32 bytes instead of 64 bytes. The highest stride shown in Figure 35 is 4,096 instead of 8,192. After this conversion, the write traffic for all GPUs and the CPU shows similar data and trends. The similarity is observed between the read traffic for the P100 and the CPU with prefetching disabled (see the overlapped blue and black lines in Figure 35 for the read traffic).

All GPUs and the CPU in Figure 35 show similar data for the read traffic until a stride of 8. The A100, V100, and the prefetching-enabled CPU show the same trend after a stride size of 8 (denoted by green and red lines in Figure 35). However, there is no overlap between the green and red lines, and the CPU reports higher read traffic than the A100 GPU until stride 64, and then the opposite is observed. The CPU keeps prefetching until stride 80, which is not shown in the figure and is determined experimentally. In contrast, the GPU keeps prefetching even for higher strides where there should not be any benefit for such action because the memory accesses are more than four cache lines apart. In summary, it can be said that there are more similarities than dissimilarities. Therefore, the following hypothesis can be formulated.

Hypothesis-2 A model prepared to predict LLC-memory traffic for sequential streaming and strided access patterns for an Intel Skylake CPU with a cache line size of 32 bytes can be customized to predict LLC-memory traffic for NVIDIA GPUs.

6.3.2 LLC-memory Traffic of AMD GPUs.

Compared with the NVIDIA GPUs, the AMD GPUs considered in this study are relatively new. The MI50 and MI60 Instinct GPUs have the same GCN 5.1 architecture, whereas MI100 adopts a CDNA 1.0 architecture. Because these GPUs were released within two years, similarity is expected.

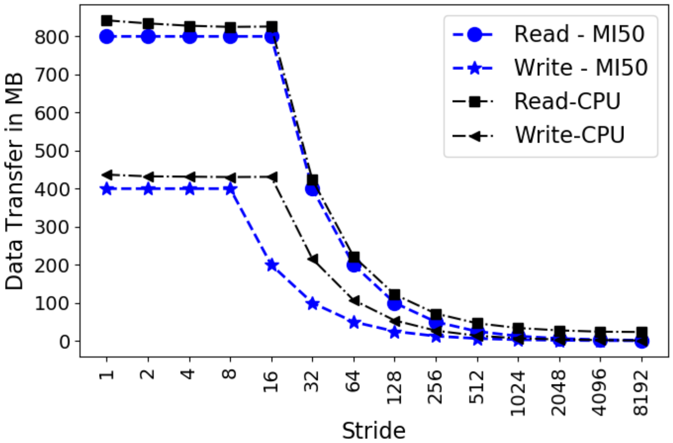


Figure 36. Read and write traffic for the MI50 GPU compared with the Intel Skylake CPU.

6.3.2.1 Similarities between MI50 GPU and Skylake CPU.

The LLC-memory traffic for MI50 is shown in Figure 36. The read and write traffic for stride 1 is very close to the theoretical lower bound. The read traffic in Figure 36 for MI50 follows the same trend as the Skylake CPU when the data structure is not initialized (shown initially in Figure 24b in Section 5.2 of Chapter V). The *vecMul* function for the GPU does not have the write array initialized. So, unlike NVIDIA GPUs, the MI50 does not implement allocating store, which is more appropriate for this function because there is no value in the write array, and bringing the cache line from memory is unnecessary.

Observation-5 AMD GPUs do not implement allocating store for an uninitialized write array.

Even though the read traffic shows similarity with the non-initialized case, the write traffic in Figure 36 shows the same trend as the write traffic for the initialized case. Hence, it proves that page zeroing, like on the CPU, is not occurring even though the write array is not initialized.

The most interesting observation in Figure 36 is that a drop in read traffic occurs when the stride is 16, but a drop in write traffic occurs when the stride is 8. This difference suggests that the cache line length is 64 bytes for the read transactions and 32 bytes for the write transactions. Such a scenario is not observed in Intel CPUs or NVIDIA GPUs. To confirm the cache line length, the *rocminfo* command is used in all AMD GPUs, in which the cache line length is 64 bytes. The write traffic dropping at a stride of 8 instead of 16, even though the cache line length is 64 bytes, must be investigated.

To investigate this anomaly, the metric used to measure the write traffic is explored. The metric's formula is *WRITE_SIZE = (TCC_MC_WRREQ_sum*32) / 1024* (this formula is found in the metrics.xml file inside the ROCm software stack's rocprofiler directory). The metric is described as, The total kilobytes fetched from the video memory. This is measured with all extra fetches and any cache or memory effects taken into account. For further investigation, the *TCC_MC_WRREQ_sum* metric is explored, where the formula is *sum(TCC_MC_WRREQ,16)*. The metric is described as, Number of 32-byte transactions going over the *TC_MC_wrreq* interface. Sum over TCC instances. This description confirms that AMD GPUs perform 32-byte transactions for write traffic, and the write traffic in AMD GPUs follows the same trend as the NVIDIA GPUs.

Observation-6 AMD GPUs show 64-byte LLC-memory transactions for read traffic and 32-byte transactions for write traffic.

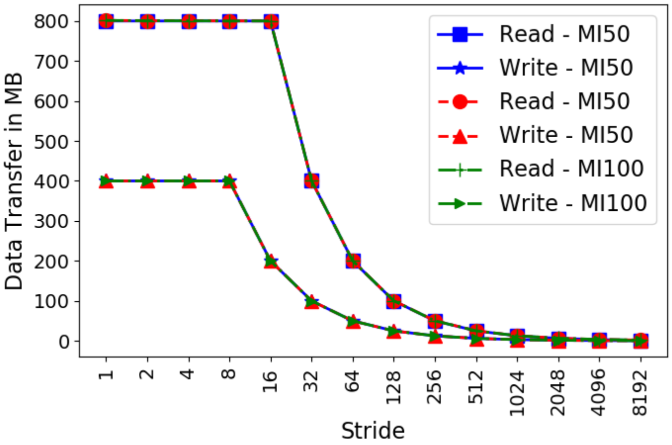


Figure 37. Read and write traffic for all AMD GPUs.

6.3.2.2 Similarities between MI50, MI60, and MI100. The LLC-memory traffic for MI50, MI60, and MI100 is shown in Figure 37. All AMD GPUs show similar trends and data. Therefore, Observations 4 and 5 apply to all AMD GPUs (transitive property), and the following hypothesis can be made.

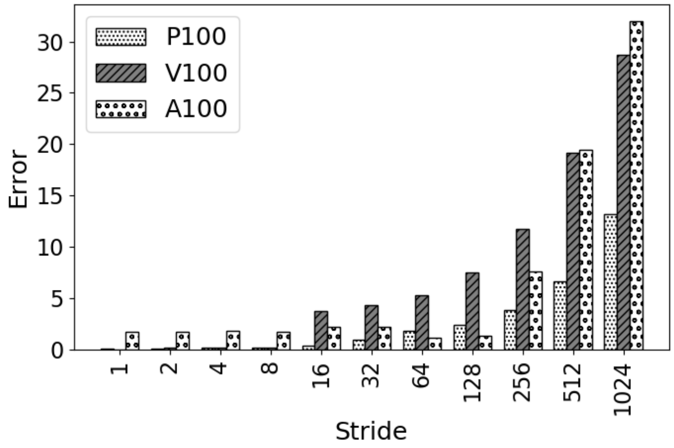
Hypothesis-3 Because of the many similarities, a model prepared to predict LLC-memory traffic for sequential streaming and strided access patterns for an Intel Skylake CPU can be customized to predict LLC-memory traffic for AMD Instinct GPUs.

6.3.3 Comparison of the Profiling Tools. This study found that NVIDIA's *ncu* tool provides more hardware counters and metrics for GPU execution than AMD's *rocprof*. However, the detailed formula behind a metric can be found in *rocprof*, which is helpful for further investigation. Moreover, *rocprof* provides the facility to write custom-derived metrics. From an accuracy standpoint, both *ncu* and *rocprof* provide reasonably accurate traffic counts that are close to the theoretical traffic count for lower strides. Unfortunately, this is not the case for CPUs. Our previous study observed some extra traffic while measuring CPUs. One common problem found for all cases is that when the number of accesses is low (i.e., higher strides), the traffic count is not accurate when compared with the theoretical traffic. Based on this study, it is possible that the inaccuracies in higher strides come from having low memory access and an application with a shorter lifespan. A shorter application lifespan makes pinpointing the memory traffic of a function difficult for the profiler. While modeling and comparing accuracy, this factor must be taken into consideration.

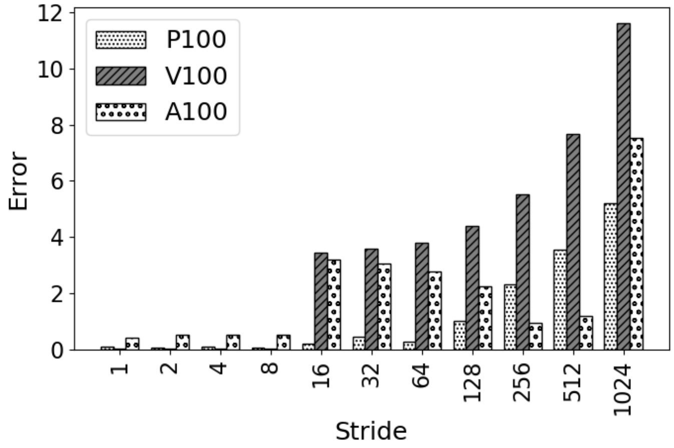
6.3.4 Discussion about the Hypotheses. Three hypotheses are formulated in this study. To prove Hypothesis-1, one would need to study the details of the changes in NVIDIA's prefetching algorithm from Pascal to Volta. However, Hypotheses 2 and 3 can be verified by preparing a prediction model.

6.4 Experiment and Prediction: A Proof of Concept

In this section, a prediction model is formulated to test Hypotheses 2 and 3. The model is evaluated for different input sizes for the *vecMul* function on the NVIDIA and AMD GPUs. In Section 6.3, the input size used is 100 million of 32-bit floating-point data. However, input sizes of 50 million and 200 million of 32-bit floating-point data are evaluated in this section. The predicted and measured total traffic are compared



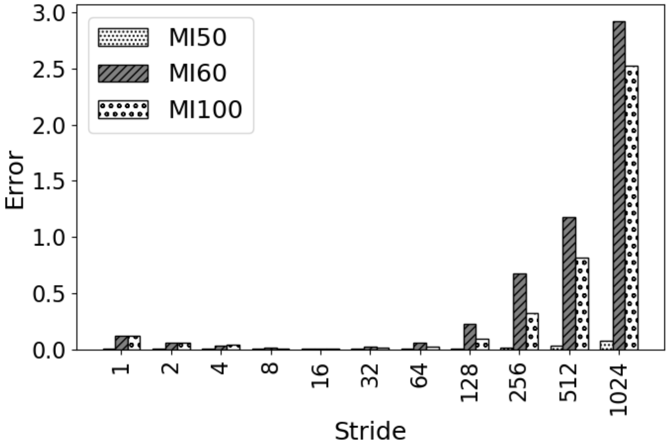
(a) Prediction accuracy for input size of 50 M.



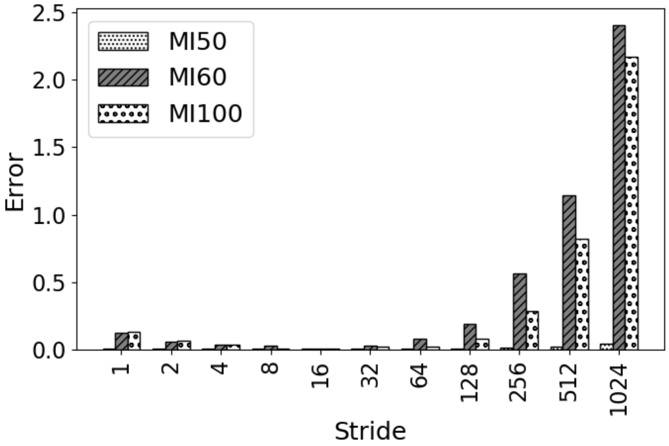
(b) Prediction accuracy for input size of 200 M.

Figure 38. Prediction accuracy for NVIDIA GPUs for different input sizes. (Here, M = million)

to ascertain the prediction error. Relative accuracy is considered to determine the error, where *error = Absolute[(measured-predicted)/measured*100]*, and the formula for accuracy is *accuracy = [100 - error]*. While experimenting, it is observed that NVIDIA Nsight Compute does not generate data for smaller data sizes with less computation (also explained in Section 6.3.3). For this reason, strides considered in this evaluation range from 1 to 1,024.



(a) Prediction accuracy for input size of 50 M.



(b) Prediction accuracy for input size of 200 M.

Figure 39. Prediction accuracy for AMD GPUs for different input sizes. (Here, M = million)

6.4.1 Prediction Model for NVIDIA GPUs.

The prediction strategy for NVIDIA GPUs is presented in Table 21, which incorporates Observations 1–4 (reported in Section 6.3). Here, *stream* stands for the calculated theoretical traffic for one data structure. For a 100 million 32-bit floating-point data structure size, the stream equals 400 MB. To reference the cell above, the term *prev* is used.

6.4.2 Prediction Accuracy for NVIDIA GPUs.

The prediction accuracy of the model presented in Table 21 is depicted in Figure 38. For the input

Table 21. Prediction for NVIDIA GPUs.

Stride	**Read P100**	**Read V100/A100**	**Write for all**
1	stream * 2	stream * 2	stream
2	stream * 3	stream * 3	stream
4	stream * 3	stream * 3	stream
8	stream * 3	stream * 3	stream
16	prev/2	prev/2 * 1.6	prev/2
32	prev/2	prev/2	prev/2
64	prev/2	prev/2	prev/2
128	prev/2	prev/2	prev/2
256	prev/2	prev/2	prev/2
512	prev/2	prev/2	prev/2
1,024	prev/2	prev/2	prev/2

size of 50 million, high accuracy is observed for NVIDIA GPUs (P100 showed 97.3% accuracy, V100 showed 92.6% accuracy, and A100 showed 93.4% accuracy). However, the error increases with higher strides. When the 200 million input size is used, the average accuracy increased (P100 showed 98.8% accuracy, V100 showed 96.4% accuracy, and A100 showed 97.9% accuracy). This increased accuracy for larger data sizes confirms that the profiler cannot report exact memory traffic for applications with short lifespans that operate on smaller data sizes. Another observation can be made from Figure 38b: from stride 16 onward, V100 and A100 report a higher amount of errors. This inaccuracy stems from using the factor 1.6× (Observation-4) to capture the impact of the prefetchers. Therefore, a better understanding of NVIDIA's prefetchers is needed to improve the model's accuracy. However, Figure 38b shows that this method still provides above 88% accuracy for all cases.

6.4.3 Prediction Model for AMD GPUs.

The prediction strategy for AMD GPUs is presented in Table 22, which incorporates Observations 5 and 6 (reported in Section 6.3). Because all AMD GPUs in this study follow a similar pattern, the same model is used for all.

Table 22. Prediction for AMD GPUs.

Stride	**Read for all**	**Write for all**
1	stream * 2	stream
2	stream * 2	stream
4	stream * 2	stream
8	stream * 2	stream
16	stream * 2	prev/2
32	prev/2	prev/2
64	prev/2	prev/2
128	prev/2	prev/2
256	prev/2	prev/2
512	prev/2	prev/2
1,024	prev/2	prev/2

6.4.4 Prediction accuracy for AMD GPUs.

The prediction accuracy for the model presented in Table 22 is portrayed in Figure 39. AMD GPUs provided higher accuracy than the NVIDIA GPUs. For the input size of 50 million, higher average accuracy is observed (MI50 showed 99.98% accuracy, MI60 showed 99.5% accuracy, and MI100 showed 99.6% accuracy). Even higher inaccuracies are observed with a 200 million input size (MI50 showed 99.9% accuracy, MI60 showed 99.6% accuracy, and MI100 showed 99.7% accuracy). For both input sizes, error rates increased for higher strides; however, all strides provided above 97% accuracy.

6.4.5 Discussion.

The seemingly simplistic model generated higher accuracy than the CPU-prediction model presented in our previous study [16]. Sequential streaming and strided memory access patterns are common in large applications. Models for other access patterns, such as stencils, can be derived by following similar methodologies used in Chapter V. Therefore, a proof of concept for these two patterns opens the door for predicting LLC-memory traffic for larger applications.

6.5 Related Work

Memory access patterns play an important role in deciding the performance of an application running on GPUs [210]. Two studies delved into sequential streaming and strided access patterns. Allen et al. investigated the impact of memory access patterns on power and performance of GPUs [203]. Their focus was to understand the impact on attained bandwidth and average power. Ding et al. proposed an instruction Roofline model for GPUs [216]. Even though the study focused on different instructions, the authors looked into the impact of sequential streaming and strided memory access patterns to define the theoretical upper bound. Unlike these studies, we strive to understand the memory transactions that take place between the LLC and memory. Other studies also focused on understanding the impact of memory accesses on GPU performance and power [217, 218]. Ben-Nun et al. investigated different memory-access patterns for a multi-GPU scenario [219]. The main objective of their work was to provide task partitioning and device-level optimization for a multi-GPU environment. Our study considered regular access patterns with no possibility of bank conflict at the shared memory for a warp/wavefront. This is the best-case scenario for performance. However, multiple threads in a warp can access the same bank for an irregular application, thereby causing bank conflicts that impact performance. Burtscher et al. investigated such irregular applications on GPUs [220].

Our study differs from these efforts because the main objective of our investigation is to separate the LLC-memory traffic from other observations, such as execution time, attained bandwidth, and energy consumption. This separation allows us to find the apparent similarities between CPUs and GPUs.

6.6 Summary

This chapter investigates the impact of sequential streaming and strided memory access patterns on different NVIDIA and AMD GPUs. By presenting a similar study to the Intel Skylake CPU, this effort attempts to identify the similarities and dissimilarities in LLC-memory traffic on different generations of NVIDIA and AMD GPUs. Through investigations, some key observations and hypotheses are made. Models are prepared by incorporating those key observations. Experimental evaluation of models shows that the LLC-memory traffic can be predicted for different memory-access patterns in GPUs. These models can be extended and implemented in MAPredict for CPU like prediction shown in Chapter V to enable MEPHESTO (presented in Chapter IV) in a heterogeneous system.

CHAPTER VII

CONCLUSION AND FUTURE WORK

Since the end of Dennard scaling [221], heterogeneous architectures have become the go-to solution for modern high-performance computing (HPC) systems, and this trend is expected to continue in the future [222]. Heterogeneity meets the diverse needs of HPC users by hosting powerful CPUs and GPUs in the same node. However, it also increases the complexity of programmability, hardware design, and the role of a runtime system. For this reason, runtime systems have been adapting themselves to provide support for newer programming and execution models. Since extremely heterogeneous systems are becoming more available, runtime systems living in the middle of the software stack and underlying hardware can make intelligent decisions dynamically during execution. However, a runtime system needs information about applications and machines to make such decisions. Therefore, this dissertation explores dynamic adaptation techniques and strives to answer the following main question: **How can information gathered from applications and machines at compile time empower modern HPC runtime systems for intelligent and dynamic decisions?** Chapter I provides an introduction to the challenges that need to be addressed to answer this main question. Chapter I also introduces five fine-grained questions that are addressed in the subsequent chapters to realize the solution to the main question (the flow is presented in Figure 40). The summary of these chapters is given below.

In Chapter II, we explore the evolution of HPC runtime systems for the last 35 years to identify what drives change. Based on the survey, it is safe to say **changes in architecture majorly influence the evolution of HPC runtime systems**. This finding is instrumental in providing proof that dynamic adaption in runtime systems needs to account for the underlying hardware architecture. Chapter II then identifies the dynamic adaptation opportunities and shows a correlation between these identified opportunities and the chapters of this dissertation.

In Chapter III, we explore dynamic adaption in HPX runtime on different CPU microarchitectures. By delving into parcel coalescing and task inlining strategies in HPX runtime, Chapter III demonstrates the benefit of dynamically adaptive policies over static policies. However, this chapter does not consider heterogeneous systems.

In Chapter IV, we investigated task placement opportunities in a heterogeneous system with shared memory. We implemented MEPHESTO, an energy-performance trade-off aware scheduling approach capable of mitigating memory contention at the hardware level. Chapter IV demonstrated that knowing the operational intensity of kernels empowers a runtime system to make intelligent decisions that can achieve an energy-performance trade-off. Since operational intensity is a metric that derives from the handshake between the application and hardware architecture, Chapter IV successfully demonstrates the need for investigating static deduction of LLC-DRAM traffic at compile time.

In Chapter V, we investigated LLC-DRAM traffic at different Intel CPU microarchitectures. We implemented MAPredict, a static analysis-driven LLC-DRAM traffic-prediction framework for Intel CPUs. By unveiling factors that initiate an LLC-DRAM transaction, Chapter V formulates analytical models, which can be invoked from the MAPredict framework at compile time to deduce LLC-DRAM traffic prediction.

In Chapter VI, we delved into efforts to understand LLC-memory traffic in GPUs from AMD and NVIDIA (here, memory represents the device memory of GPUs and the system memory for CPUs). Chapter VI shows striking similarities between CPUs and GPUs for LLC-memory traffic patterns for sequential streaming and strided memory access. From the understanding of LLC-memory traffic, analytical models are generated that can predict LLC-DRAM traffic for small-scale applications, thereby opening the door to enhance MAPredict for GPUs.

Dynamic adaptation opportunities identified in Chapter II are addressed in Chapters III and IV by exploring APEX and introducing MEPHESTO, respectively. Chapters V and VI explores the sub-problems of Chapters III and IV and investigated CPUs and GPUs to enable static prediction of LLC-memory traffic by introducing MAPredict. Integrating MAPredict and MEPHESTO (or MAPredict and APEX) would enable a runtime system to make application and hardware aware intelligent and dynamic decisions for meeting energy and performance goals in a heterogeneous system and thereby constitutes an answer to the main question. Such integration completes the loop of the flow of this dissertation, which is shown in Figure 40 using an orange box. Moreover, MAPredict and MEPHESTO can open the door to future research opportunities discussed in the following section.

7.1 Future Work

We foresee this dissertation contributing to the following five areas that would further strengthen application and hardware aware decision making in HPC runtimes.

7.1.1 Extending MAPredict.

Understanding and statically measuring LLC-DRAM traffic provides important performance insight of an application. Chapter VI showed the similarities for the LLC-DRAM traffic between Intel CPUs and NVIDIA and AMD GPUs. The modular design of MAPredict allows the inclusion of manufacturer-specific LLC-DRAM traffic prediction models. Therefore, exploring CPUs and GPUs from other manufacturers (such as CPUs from Arm, AMD, and IBM and GPUs from Intel) would add to the usability of MAPredict.

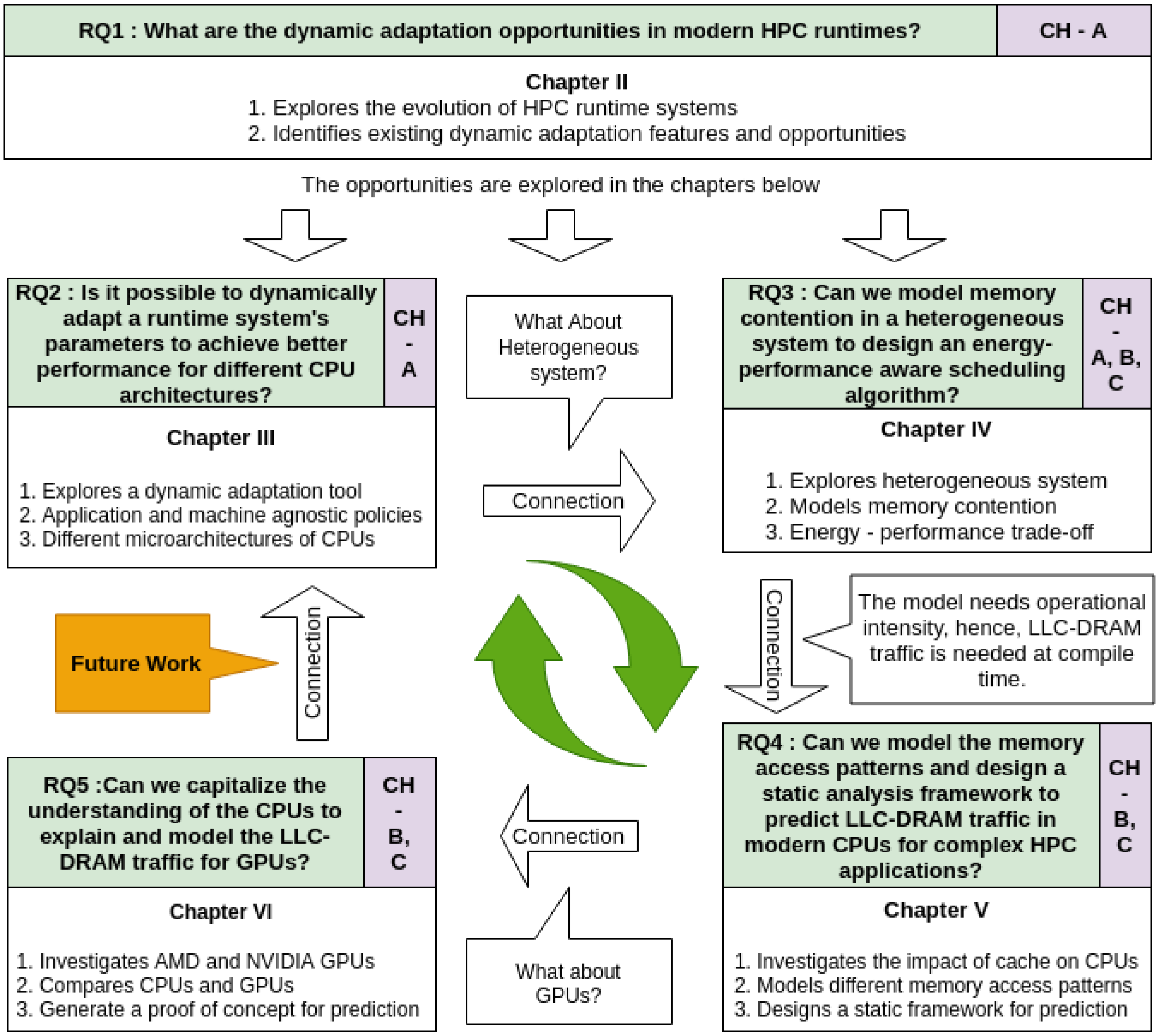

RQ1 : What are the dynamic adaptation opportunities in modern HPC runtimes?
CH - A
Chapter II
1. Explores the evolution of HPC runtime systems
2. Identifies existing dynamic adaptation features and opportunities
The opportunities are explored in the chapters below
RQ2 : Is it possible to dynamically adapt a runtime system's parameters to achieve better performance for different CPU architectures?
CH - A
Chapter III
1. Explores a dynamic adaptation tool
2. Application and machine agnostic policies
3. Different microarchitectures of CPUs
What About Heterogeneous system?
Connection
RQ3 : Can we model memory contention in a heterogeneous system to design an energy-performance aware scheduling algorithm?
CH - A, B, C
Chapter IV
1. Explores heterogeneous system
2. Models memory contention
3. Energy - performance trade-off
Future Work
Connection
Connection
The model needs operational intensity, hence, LLC-DRAM traffic is needed at compile time.
RQ4 : Can we model the memory access patterns and design a static analysis framework to predict LLC-DRAM traffic in modern CPUs for complex HPC applications?
CH - B, C
Chapter V
1. Investigates the impact of cache on CPUs
2. Models different memory access patterns
3. Designs a static framework for prediction
RQ5 :Can we capitalize the understanding of the CPUs to explain and model the LLC-DRAM traffic for GPUs?
CH - B, C
Chapter VI
1. Investigates AMD and NVIDIA GPUs
2. Compares CPUs and GPUs
3. Generate a proof of concept for prediction
Connection
What about GPUs?

Figure 40. The flow of this dissertation and future work opportunities.

7.1.2 Execution Time and Energy Consumption Prediction.

The Aspen [15] and COMPASS [13] framework can provide execution-time and energy-consumption prediction based on instruction counts. However, relying on instruction counts leads to inaccuracies because both cache and DRAM may serve memory transactions. The LLC-DRAM transactions are one of the slowest factors during the execution of a kernel. Therefore, MAPredict can be integrated with the prediction tools in the Aspen or COMPASS frameworks for higher accuracy.

7.1.3 Static Characterization of Workloads.

The roofline model [178] is widely used for identifying compute and memory-bound workloads based on operational intensity (FLOPs per LLC-DRAM byte). The primary motivation of MAPredict is to generate operational intensity for MEPHESTO. However, statically deducing operational intensity provides the opportunity to generate Roofline positioning at compile time. Since the COMPASS framework can statically deduce the number of FLOPs, MAPredict can be enhanced to use the FLOP counting feature of COMPASS to enable static characterization of workloads. Dynamic performance measurement and analysis tools, such as Intel Advisor and NVIDIA Nsight Compute, generate Roofline graphs by executing the application. MAPredict will be able to do such analysis at compile time.

7.1.4 Dynamic Adaptation in HPC Runtime.

As this dissertation suggests, MAPredict and MEPHESTO can be integrated and implemented in a runtime system to provide energy-performance trade-off-aware decisions. Moreover, MAPredict's capability of identifying memory-intensive kernels can be used for more dynamic decisions. For example, MAPredict can be integrated with the APEX tool to enable NUMA-aware thread number selection in OpenMP runtime. Section 5.6.2.6 in Chapter V shows that scheduling memory-intensive kernels with large data structures in a single NUMA domain provide better performance. APEX can be used to communicate to the OpenMP runtime system (through OMPT) for dynamic selection of NUMA domain based on the memory intensity reported by MAPredict.

7.1.5 Exploring New Architectures. Figure 4 in Chapter II suggests that heterogeneity will be continued in the future. Heterogeneity is now present both in processing units and memory architectures. Specialized processors tuned for solving machine learning problems are coming into existence. Processors such as deep learning accelerators (NVIDIA DLA) and vision processors (NVIDIA PVA) deviate from traditional CPUs and GPUs. Moreover, mainstream NVIDIA and AMD GPUs include specialized tensor/matrix cores to facilitate machine learning applications. Intel also launched its OpenVINO toolkit to facilitate AI workloads in FPGAs. These solution-specific processors are introducing new architectures that need to be understood to generate models to empower a runtime system to make better decisions. Moreover, exciting new ideas like NVIDIA DPU or Intel IPU aim to provide computation power to network interface card level and need to be explored. Last but not least, the impact of heterogeneity in memory hierarchy where high bandwidth memory acts as a cache for slower memory in a vertical organization requires investigation. Therefore, the loop presented in Figure 40 needs to be iterated to include these new architectures in the proposed flow of this dissertation, which is shown using the green cycle.

Computer architecture experts predict that the trend of heterogeneous systems to meet diverse computing needs will continue for at least a decade. Specialized hardware will keep coming into existence until we find a way to overcome Dennard scaling and the memory wall. Therefore, understanding and modeling the impact of memory hierarchy will remain crucial and must be an ongoing process for general performance improvement efforts and enabling runtime systems for intelligent dynamic decisions.

REFERENCES CITED

[1] "Mpich," https://www.mpich.org/, accessed: 2020-11-01.

[2] OpenMP, "OpenMP reference," 1999.

[3] H. Kaiser, T. Heller, B. Adelstein-Lelbach, A. Serio, and D. Fey, "Hpx: A task based programming model in a global address space," in *Proceedings of the 8th International Conference on Partitioned Global Address Space Programming Models*. ACM, 2014, p. 6.

[4] L. V. Kale and S. Krishnan, "Charm++: a portable concurrent object oriented system based on c++," in *ACM Sigplan Notices*, vol. 28, no. 10. ACM, 1993, pp. 91–108.

[5] C. Augonnet, S. Thibault, R. Namyst, and P.-A. Wacrenier, "Starpu: a unified platform for task scheduling on heterogeneous multicore architectures," *Concurrency and Computation: Practice and Experience*, vol. 23, no. 2, pp. 187–198, 2011.

[6] NVidia, "CUDAZone," 2008.

[7] "Amd corporation. rocm, a new era in open gpu computing." https://rocmdocs.amd.com/en/latest/, accessed: 2020-11-01.

[8] S. Shende and A. Malony, "The tau parallel performance system," *The International Journal of High Performance Computing Applications*, vol. 20, no. 2, pp. 287–311, 2006.

[9] K. A. Huck, A. Porterfield, N. Chaimov, H. Kaiser, A. D. Malony, T. Sterling, and R. Fowler, "An Autonomic Performance Environment For Exascale," *Supercomputing frontiers and innovations*, vol. 2, no. 3, pp. 49–66, 2015.

[10] B. Wagle, M. A. H. Monil, K. Huck, A. D. Malony, A. Serio, and H. Kaiser, "Runtime adaptive task inlining on asynchronous multitasking runtime systems," in *Proceedings of the 48th International Conference on Parallel Processing*, 2019, pp. 1–10.

[11] M. A. H. Monil, B. Wagle, K. Huck, and H. Kaiser, "Adaptive auto-tuning in hpx using apex," in *47th International Conference on Parallel Processing (ICPP 2018)*, 2018.

[12] M. Monil, M. Belviranli, S. Lee, J. Vetter, and A. Malony, "Mephesto: Modeling energy-performancein heterogeneous socs and their trade-offs," in *Proceedings of the 29th International Conference on Parallel Architectures and Compilation Techniques (PACT)*, 2020.

[13] S. Lee, J. S. Meredith, and J. S. Vetter, "COMPASS: A framework for automated performance modeling and prediction," in *Proceedings of the 29th ACM on International Conference on Supercomputing.* Newport Beach, California, USA: ACM, 2015, pp. 405–414.

[14] S. Lee and J. S. Vetter, "OpenARC: Open accelerator research compiler for directive-based, efficient heterogeneous computing," in *ACM Symposium on High-Performance Parallel and Distributed Computing (HPDC).* Vancouver: ACM, 2014.

[15] K. L. Spafford and J. S. Vetter, "Aspen: A domain specific language for performance modeling," in *SC12: International Conference for High Performance Computing, Networking, Storage and Analysis*, Salt Lake City, 2012, pp. 1–11.

[16] M. Monil, S. Lee, J. Vetter, and A. Malony, "Understanding the impact of memory access patterns in intel processors," 2020.

[17] "Programming model," https://en.wikipedia.org/wiki/Programming_model, accessed: 2020-11-01.

[18] "Execution model," https://en.wikipedia.org/wiki/Execution_model, accessed: 2020-11-01.

[19] "Openmp main site," https://www.openmp.org, accessed: 2020-11-01.

[20] "Hpx: Programming language manual," http://stellar.cct.lsu.edu/files/hpx-0.9.9/html/index.html, accessed: 2020-11-01.

[21] "Runtime ystem," https://en.wikipedia.org/wiki/Runtime_system, accessed: 2020-11-01.

[22] "The website of chapel parallel programming language," https://chapel-lang.org/, accessed: 2020-11-01.

[23] G. Bosilca, A. Bouteiller, A. Danalis, M. Faverge, T. Hérault, and J. J. Dongarra, "Parsec: Exploiting heterogeneity to enhance scalability," *Computing in Science & Engineering*, vol. 15, no. 6, pp. 36–45, 2013.

[24] "History of supercomputing," http://wotug.org/parallel/documents/misc/timeline/timeline.txt, accessed: 2020-11-01.

[25] “Parallel computing works,” http://www.netlib.org/utk/lsi/pcwLSI/text/BOOK.html, accessed: 2020-11-01.

[26] C. L. Seitz, “The cosmic cube,” *Communications of the ACM*, vol. 28, no. 1, pp. 22–33, 1985.

[27] G. T. Almes, A. P. Black, E. D. Lazowska, and J. D. Noe, “The eden system: A technical review,” *IEEE Transactions on Software Engineering*, no. 1, pp. 43–59, 1985.

[28] N. P. Kronenberg, H. M. Levy, and W. D. Strecker, “Vaxcluster: A closely-coupled distributed system,” *ACM Transactions on Computer Systems (TOCS)*, vol. 4, no. 2, pp. 130–146, 1986.

[29] P. Pierce, “The nx/2 operating system,” in *Proceedings of the third conference on Hypercube concurrent computers and applications: Architecture, software, computer systems, and general issues-Volume 1*, 1988, pp. 384–390.

[30] J. Flower and A. Kolawa, “Express is not just a message passing system current and future directions in express,” *Parallel Computing*, vol. 20, no. 4, pp. 597–614, 1994.

[31] R. Butler and E. Lusk, “User’s guide to the p4 programming system,” Technical Report TM-ANL {92/17, Argonne National Laboratory, Tech. Rep., 1992.

[32] V. Bala and S. Kipnis, “Process groups: a mechanism for the coordination of and communication among processes in the venus collective communication library,” in *[1993] Proceedings Seventh International Parallel Processing Symposium*. IEEE, 1993, pp. 614–620.

[33] J. Dongarra, G. Geist, R. Manchek, and V. S. Sunderam, “Integrated pvm framework supports heterogeneous network computing,” *Computers in physics*, vol. 7, no. 2, pp. 166–175, 1993.

[34] W. Gropp and B. Smith, “Users manual for the chameleon parallel programming tools,” Argonne National Lab., IL (United States); Office of Naval Research . . . , Tech. Rep., 1993.

[35] D. E. Culler, A. Dusseau, S. C. Goldstein, A. Krishnamurthy, S. Lumetta, T. Von Eicken, and K. Yelick, “Parallel programming in split-c,” in *Supercomputing’93: Proceedings of the 1993 ACM/IEEE conference on Supercomputing*. IEEE, 1993, pp. 262–273.

[36] J. Chase, F. Amador, E. Lazowska, H. Levy, and R. Littlefield, “The amber system: Parallel programming on a network of multiprocessors,” in *Proceedings of the twelfth ACM symposium on Operating systems principles*, 1989, pp. 147–158.

[37] K. Feind, “Shared memory access (shmem) routines,” *Cray Research*, vol. 53, 1995.

[38] T. Von Eicken, D. E. Culler, S. C. Goldstein, and K. E. Schauser, “Active messages: a mechanism for integrated communication and computation,” *ACM SIGARCH Computer Architecture News*, vol. 20, no. 2, pp. 256–266, 1992.

[39] B. N. Bershad and M. J. Zekauskas, “Midway: Shared memory parallel programming with entry consistency for distributed memory multiprocessors,” 1991.

[40] E. Upchurch, P. L. Springer, M. Brodowicz, S. Brunett, and T. D. Gottschalk, “Performance analysis of blue gene/L using parallel discrete event simulation,” in *Lecture Notes in Computer Science : High Performance Computing - HiPC 2003*, 2003, pp. 2–11.

[41] “Co-array fortran,” https://bluewaters.ncsa.illinois.edu/caf, accessed: 2021-01-01.

[42] “Titanium,” http://titanium.cs.berkeley.edu/, accessed: 2021-01-01.

[43] I. Grasso and S. Pellegrini, “Libwater: heterogeneous distributed computing made easy,” *Proceedings of the 27th . . .* , pp. 161–171, 2013.

[44] “The website of x10,” http://x10-lang.org/, accessed: 2020-11-01.

[45] K. B. Wheeler, R. C. Murphy, D. Stark, and B. L. Chamberlain, “The chapel tasking layer over qthreads.” Sandia National Lab.(SNL-NM), Albuquerque, NM (United States), Tech. Rep., 2011.

[46] “Cuda runtime api,” https://docs.nvidia.com/cuda/cuda-runtime-api/index.html, accessed: 2020-11-01.

[47] “The opencl specification,” https://www.khronos.org/registry/OpenCL/specs/2.2/html/OpenCL_API.html, accessed: 2020-11-01.

[48] OpenACC, “OpenACC: Directives for accelerators,” 2015.

[49] “Documentaion of starpu,” https://files.inria.fr/starpu/doc/starpu.pdf, accessed: 2020-11-01.

[50] “Intel cilk plus,” https://www.cilkplus.org/, accessed: 2020-11-01.

[51] “Intel tbb,” https://software.intel.com/content/www/us/en/develop/tools/threading-building-blocks.html, accessed: 2020-11-01.

[52] M. Bauer, S. Treichler, E. Slaughter, and A. Aiken, "Legion: Expressing locality and independence with logical regions," *International Conference for High Performance Computing, Networking, Storage and Analysis, SC*, 2012.

[53] H. C. Edwards, C. R. Trott, and D. Sunderland, "Kokkos: Enabling manycore performance portability through polymorphic memory access patterns," *Journal of Parallel and Distributed Computing*, vol. 74, no. 12, pp. 3202–3216, 2014.

[54] "The high performance open community runtime (p-ocr)," https://hpc.pnl.gov/projects/POCR/, accessed: 2020-11-01.

[55] S. Seo, A. Amer, P. Balaji, C. Bordage, G. Bosilca, A. Brooks, P. Carns, A. Castelló, D. Genet, T. Herault *et al.*, "Argobots: A lightweight low-level threading and tasking framework," *IEEE Transactions on Parallel and Distributed Systems*, vol. 29, no. 3, pp. 512–526, 2017.

[56] T. Sterling, M. Anderson, and M. Brodowicz, "A survey: runtime software systems for high performance computing," *Supercomputing Frontiers and Innovations*, vol. 4, no. 1, pp. 48–68, 2017.

[57] P. Thoman, K. Hasanov, K. Dichev, R. Iakymchuk, X. Aguilar, P. Gschwandtner, E. Laure, H. Jordan, P. Lemarinier, K. Katrinis *et al.*, "A taxonomy of task-based technologies for high-performance computing," in *Proceedings of International Conference Parallel Processing and Applied Mathematics (To appear) Google Scholar*, 2018.

[58] R. D. Blumofe, C. F. Joerg, B. C. Kuszmaul, C. E. Leiserson, K. H. Randall, and Y. Zhou, "Cilk: An efficient multithreaded runtime system," *ACM SigPlan Notices*, vol. 30, no. 8, pp. 207–216, 1995.

[59] "The cilk project from mit," http://groups.csail.mit.edu/sct/wiki/index.php?title=The_Cilk_Project, accessed: 2020-11-01.

[60] A. D. Robison, "Composable parallel patterns with intel cilk plus," *Computing in Science & Engineering*, vol. 15, no. 2, pp. 66–71, 2013.

[61] T. Willhalm and N. Popovici, "Putting intel® threading building blocks to work," in *Proceedings of the 1st international workshop on Multicore software engineering.* ACM, 2008, pp. 3–4.

[62] L. Dagum and R. Menon, "Openmp: an industry standard api for shared-memory programming," *IEEE computational science and engineering*, vol. 5, no. 1, pp. 46–55, 1998.

[63] ——, "OpenMP: : An industry-standard API for shared-memory programming," *IEEE Computational Science AND Engineering*, vol. 5, no. 1, pp. 46–55, 1998.

[64] A. Duran, E. Ayguadé, R. M. Badia, J. Labarta, L. Martinell, X. Martorell, and J. Planas, “Ompss: a proposal for programming heterogeneous multi-core architectures,” *Parallel Processing Letters*, vol. 21, no. 02, pp. 173–193, 2011.

[65] “Ompss programming model,” https://pm.bsc.es/ompss, accessed: 2020-11-01.

[66] T. Dallou and B. Juurlink, “Hardware-based task dependency resolution for the starss programming model,” in *2012 41st International Conference on Parallel Processing Workshops.* IEEE, 2012, pp. 367–374.

[67] “Mercurium is a source-to-source compilation infrastructure,” https://pm.bsc.es/mcxx, accessed: 2020-11-01.

[68] “The nanos++ runtime system,” https://pm.bsc.es/nanox, accessed: 2020-11-01.

[69] K. B. Wheeler, R. C. Murphy, and D. Thain, “Qthreads: An api for programming with millions of lightweight threads,” in *2008 IEEE International Symposium on Parallel and Distributed Processing.* IEEE, 2008, pp. 1–8.

[70] “The qthread library,” https://cs.sandia.gov/qthreads/, accessed: 2020-11-01.

[71] R. B. Brightwell and S. L. Olivier, “Qthreads: Run time library support for task parallel programming.” Sandia National Lab.(SNL-NM), Albuquerque, NM (United States), Tech. Rep., 2016.

[72] B. Acun, A. Gupta, N. Jain, A. Langer, H. Menon, E. Mikida, X. Ni, M. Robson, Y. Sun, E. Totoni *et al.*, “Parallel programming with migratable objects: Charm++ in practice,” in *High Performance Computing, Networking, Storage and Analysis, SC14: International Conference for.* IEEE, 2014, pp. 647–658.

[73] H. Kaiser, M. Brodowicz, and T. Sterling, “Parallex an advanced parallel execution model for scaling-impaired applications,” in *2009 International Conference on Parallel Processing Workshops.* IEEE, 2009, pp. 394–401.

[74] M. Bauer, S. Treichler, E. Slaughter, and A. Aiken, “Legion: Expressing locality and independence with logical regions,” in *Proceedings of the international conference on high performance computing, networking, storage and analysis.* IEEE Computer Society Press, 2012, p. 66.

[75] “The website of legion,” https://legion.stanford.edu/, accessed: 2020-11-01.

[76] S. Treichler, M. Bauer, and A. Aiken, “Realm: An event-based low-level runtime for distributed memory architectures,” in *Proceedings of the 23rd international conference on Parallel architectures and compilation*, 2014, pp. 263–276.

[77] T. G. Mattson, R. Cledat, V. Cavé, V. Sarkar, Z. Budimlić, S. Chatterjee, J. Fryman, I. Ganev, R. Knauerhase, M. Lee *et al.*, "The open community runtime: A runtime system for extreme scale computing," in *2016 IEEE High Performance Extreme Computing Conference (HPEC)*. IEEE, 2016, pp. 1–7.

[78] "Ocr-vx - an alternative implementation of the open community runtime," https://www.univie.ac.at/ocr-vx/, accessed: 2020-11-01.

[79] T. Mattson, R. Cledat, Z. Budimlic, V. Cave, S. Chatterjee, B. Seshasayee, R. van der Wijngaart, and V. Sarkar, "Ocr: the open community runtime interface," *Technical report*, 2015.

[80] "Argobots: A lightweight low-level threading framework," https://www.argobots.org/, accessed: 2020-11-01.

[81] Q. Meng, A. Humphrey, and M. Berzins, "The uintah framework: A unified heterogeneous task scheduling and runtime system," in *2012 SC Companion: High Performance Computing, Networking Storage and Analysis*. IEEE, 2012, pp. 2441–2448.

[82] J. D. d. S. Germain, J. McCorquodale, S. G. Parker, and C. R. Johnson, "Uintah: A massively parallel problem solving environment," in *Proceedings the Ninth International Symposium on High-Performance Distributed Computing.* IEEE, 2000, pp. 33–41.

[83] "The uintah website," http://www.uintah.utah.edu/, accessed: 2020-11-01.

[84] "Parsec website," https://icl.utk.edu/parsec/index.html, accessed: 2020-11-01.

[85] W. W. Carlson, J. M. Draper, D. E. Culler, K. Yelick, E. Brooks, and K. Warren, "Introduction to upc and language specification," Technical Report CCS-TR-99-157, IDA Center for Computing Sciences, Tech. Rep., 1999.

[86] B. L. Chamberlain, D. Callahan, and H. P. Zima, "Parallel programmability and the chapel language," *The International Journal of High Performance Computing Applications*, vol. 21, no. 3, pp. 291–312, 2007.

[87] P. Charles, C. Grothoff, V. Saraswat, C. Donawa, A. Kielstra, K. Ebcioglu, C. Von Praun, and V. Sarkar, "X10: an object-oriented approach to non-uniform cluster computing," in *Acm Sigplan Notices*, vol. 40, no. 10. ACM, 2005, pp. 519–538.

[88] "The openacc application programming interface," https://www.openacc.org/sites/default/files/inline-files/OpenACC.2.6.final.pdf, accessed: 2020-11-01.

[89] “Hip documentation,” https://rocmdocs.amd.com/en/latest/Programming_Guides/Programming-Guides.html, accessed: 2020-11-01.

[90] S. Iwasaki, A. Amer, K. Taura, S. Seo, and P. Balaji, “Bolt: Optimizing openmp parallel regions with user-level threads,” in *2019 28th International Conference on Parallel Architectures and Compilation Techniques (PACT).* IEEE, 2019, pp. 29–42.

[91] N. Otterness and J. H. Anderson, “Amd gpus as an alternative to nvidia for supporting real-time workloads,” in *32nd Euromicro Conference on Real-Time Systems (ECRTS 2020).* Schloss Dagstuhl-Leibniz-Zentrum für Informatik, 2020.

[92] W. Kim and M. Voss, “Multicore desktop programming with intel threading building blocks,” *IEEE software*, vol. 28, no. 1, pp. 23–31, 2010.

[93] R. Hoque, T. Herault, G. Bosilca, and J. Dongarra, “Dynamic task discovery in parsec: a data-flow task-based runtime,” in *Proceedings of the 8th Workshop on Latest Advances in Scalable Algorithms for Large-Scale Systems.* ACM, 2017, p. 6.

[94] “Berkeley upc - unified parallel c,” https://upc.lbl.gov/, accessed: 2020-11-01.

[95] “Charm++: Documentations,” https://charm.readthedocs.io/en/latest/, accessed: 2020-11-01.

[96] “Hpx: Website,” https://stellar-group.org/libraries/hpx/docs/, accessed: 2020-11-01.

[97] M. E. Bauer, “Legion: Programming distributed heterogeneous architectures with logical regions,” Ph.D. dissertation, Stanford University, 2014.

[98] V. Cavé, J. Zhao, J. Shirako, and V. Sarkar, “Habanero-java: the new adventures of old x10,” in *Proceedings of the 9th International Conference on Principles and Practice of Programming in Java*, 2011, pp. 51–61.

[99] Y. Guo, J. Zhao, V. Cave, and V. Sarkar, “Slaw: A scalable locality-aware adaptive work-stealing scheduler,” in *2010 IEEE International Symposium on Parallel & Distributed Processing (IPDPS).* IEEE, 2010, pp. 1–12.

[100] W. Zhang, O. Tardieu, D. Grove, B. Herta, T. Kamada, V. Saraswat, and M. Takeuchi, “Glb: Lifeline-based global load balancing library in x10,” in *Proceedings of the first workshop on Parallel programming for analytics applications*, 2014, pp. 31–40.

[101] T. Amert, N. Otterness, M. Yang, J. H. Anderson, and F. D. Smith, "Gpu scheduling on the nvidia tx2: Hidden details revealed," in *2017 IEEE Real-Time Systems Symposium (RTSS)*. IEEE, 2017, pp. 104–115.

[102] "Profiling and tuning openacccode," http://developer.download.nvidia.com/GTC/PDF/GTC2012/PresentationPDF/S0517B-Monday-Programming-GPUs-OpenACC.pdf, accessed: 2020-11-01.

[103] J. Tompson and K. Schlachter, "An introduction to the opencl programming model," *Person Education*, vol. 49, p. 31, 2012.

[104] Y. Cho, F. Negele, S. Park, B. Egger, and T. R. Gross, "On-the-fly workload partitioning for integrated cpu/gpu architectures." in *PACT*, 2018, pp. 21–1.

[105] P. Pandit and R. Govindarajan, "Fluidic kernels: Cooperative execution of opencl programs on multiple heterogeneous devices," in *Proceedings of Annual IEEE/ACM International Symposium on Code Generation and Optimization*. ACM, 2014, p. 273.

[106] M. A. S. Bari, N. Chaimov, A. M. Malik, K. A. Huck, B. Chapman, A. D. Malony, and O. Sarood, "Arcs: Adaptive runtime configuration selection for power-constrained openmp applications," in *2016 IEEE international conference on cluster computing (CLUSTER)*. IEEE, 2016, pp. 461–470.

[107] T. Patki, D. K. Lowenthal, B. Rountree, M. Schulz, and B. R. De Supinski, "Exploring hardware overprovisioning in power-constrained, high performance computing," in *Proceedings of the 27th international ACM conference on International conference on supercomputing*, 2013, pp. 173–182.

[108] "Nvidia management library (nvml)," https://developer.nvidia.com/nvidia-system-management-interface, accessed: 2020-11-01.

[109] K. Ma, X. Li, W. Chen, C. Zhang, and X. Wang, "Greengpu: A holistic approach to energy efficiency in gpu-cpu heterogeneous architectures," in *2012 41st International Conference on Parallel Processing*. IEEE, 2012, pp. 48–57.

[110] T. Komoda, S. Hayashi, T. Nakada, S. Miwa, and H. Nakamura, "Power capping of cpu-gpu heterogeneous systems through coordinating dvfs and task mapping," in *2013 IEEE 31st International Conference on Computer Design (ICCD)*. IEEE, 2013, pp. 349–356.

[111] C. Liu, J. Li, W. Huang, J. Rubio, E. Speight, and X. Lin, "Power-efficient time-sensitive mapping in heterogeneous systems," in *Proceedings of the 21st international conference on Parallel architectures and compilation techniques*. ACM, 2012, pp. 23–32.

[112] Q. Zhu, B. Wu, X. Shen, L. Shen, and Z. Wang, “Co-run scheduling with power cap on integrated cpu-gpu systems,” in *2017 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*. IEEE, 2017, pp. 967–977.

[113] O. Sarood and L. V. Kale, “A’cool’load balancer for parallel applications,” in *Proceedings of 2011 International Conference for High Performance Computing, Networking, Storage and Analysis*, 2011, pp. 1–11.

[114] “Papi measurement for llc-dram traffic,” https://groups.google.com/a/icl.utk.edu/g/ptools-perfapi/c/NxpY2loJg4M/m/ESyCCBwYAAAJ?pli=1, accessed: 2020-09-01.

[115] H. McCraw, J. Ralph, A. Danalis, and J. Dongarra, “Power monitoring with papi for extreme scale architectures and dataflow-based programming models,” in *2014 IEEE International Conference on Cluster Computing (CLUSTER)*. IEEE, 2014, pp. 385–391.

[116] M.-A. Hermanns, N. T. Hjlem, M. Knobloch, K. Mohror, and M. Schulz, “Enabling callback-driven runtime introspection via mpi_t,” in *Proceedings of the 25th European MPI Users’ Group Meeting*, 2018, pp. 1–10.

[117] S. Ramesh, A. Mahéo, S. Shende, A. D. Malony, H. Subramoni, A. Ruhela, and D. K. D. Panda, “Mpi performance engineering with the mpi tool interface: the integration of mvapich and tau,” *Parallel Computing*, vol. 77, pp. 19–37, 2018.

[118] A. E. Eichenberger, J. Mellor-Crummey, M. Schulz, M. Wong, N. Copty, R. Dietrich, X. Liu, E. Loh, and D. Lorenz, “Ompt: An openmp tools application programming interface for performance analysis,” in *International Workshop on OpenMP*. Springer, 2013, pp. 171–185.

[119] H.-H. Su, D. Bonachea, A. Leko, H. Sherburne, M. Billingsley, and A. D. George, “Gasp! a standardized performance analysis tool interface for global address space programming models,” in *International Workshop on Applied Parallel Computing*. Springer, 2006, pp. 450–459.

[120] “Cupti documenation,” https://docs.nvidia.com/cupti/Cupti/index.html, accessed: 2020-11-01.

[121] “Library for collecting amd gpus: roc-profiler,” https://github.com/ROCm-Developer-Tools/rocprofiler, accessed: 2020-11-01.

[122] N. Chaimov, S. Shende, and A. D. Malony, “Multi-platform sycl profiling with tau,” in *Proceedings of the International Workshop on OpenCL*, 2020, pp. 1–2.

[123] C. Tapus, I.-H. Chung, and J. K. Hollingsworth, “Active harmony: Towards automated performance tuning,” in *SC’02: Proceedings of the 2002 ACM/IEEE Conference on Supercomputing*. IEEE, 2002, pp. 44–44.

[124] Y. Sun, J. Lifflander, and L. V. Kalé, “Pics: a performance-analysis-based introspective control system to steer parallel applications,” in *Proceedings of the 4th International Workshop on Runtime and Operating Systems for Supercomputers*. ACM, 2014, p. 5.

[125] A. Porterfield, R. Fowler, A. Mandal, D. O’Brien, S. Olivier, and M. Spiegel, “Adaptive scheduling using performance introspection,” TR-12-02. RENCI, 2012. R: http://www. renci. org/technical-reports/tr- 12 . . . , Tech. Rep., 2012.

[126] H. Kaiser, T. Heller, B. Adelstein-Lelbach, A. Serio, and D. Fey, “HPX: A task based programming model in a global address space,” in *Proceedings of the 8th International Conference on Partitioned Global Address Space Programming Models*. ACM, 2014, p. 6.

[127] E. Mohr, D. A. Kranz, and R. H. Halstead, Jr., “Lazy task creation: A technique for increasing the granularity of parallel programs,” in *Proceedings of the 1990 ACM Conference on LISP and Functional Programming*, ser. LFP ’90. New York, NY, USA: ACM, 1990, pp. 185–197. [Online]. Available: http://doi.acm.org/10.1145/91556.91631

[128] B. Wagle, S. Kellar, A. Serio, and H. Kaiser, “Methodology for adaptive active message coalescing in task based runtime systems,” in *2018 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)*. IEEE, 2018, pp. 1133–1140.

[129] H. Kaiser, T. Heller, B. Adelstein-Lelbach, A. Serio, and D. Fey, “Hpx: A task based programming model in a global address space,” in *Proceedings of the 8th International Conference on Partitioned Global Address Space Programming Models*, ser. PGAS ’14. New York, NY, USA: ACM, 2014, pp. 6:1–6:11. [Online]. Available: http://doi.acm.org/10.1145/2676870.2676883

[130] H. C. Baker, Jr. and C. Hewitt, “The incremental garbage collection of processes,” in *Proceedings of the 1977 Symposium on Artificial Intelligence and Programming Languages*. New York, NY, USA: ACM, 1977, pp. 55–59. [Online]. Available: http://doi.acm.org/10.1145/800228.806932

[131] J. B. Dennis, “First version of a data flow procedure language,” in *Programming Symposium, Proceedings Colloque Sur La Programmation*. Berlin, Heidelberg: Springer-Verlag, 1974, pp. 362–376. [Online]. Available: http://dl.acm.org/citation.cfm?id=647323.721501

[132] J. B. Dennis and D. P. Misunas, “A preliminary architecture for a basic data-flow processor,” in *Proceedings of the 2Nd Annual Symposium on Computer Architecture*, ser. ISCA ’75. New York, NY, USA: ACM, 1975, pp. 126–132. [Online]. Available: http://doi.acm.org/10.1145/642089.642111

[133] Y. Hu, Y. Koren, and C. Volinsky, “Collaborative filtering for implicit feedback datasets,” in *Data Mining, 2008. ICDM’08. Eighth IEEE International Conference on.* Ieee, 2008, pp. 263–272.

[134] K. Williams, A. Bigelow, and K. Isaacs, “Visualizing a Moving Target: A Design Study on Task Parallel Programs in the Presence of Evolving Data and Concerns,” *arXiv e-prints*, p. arXiv:1905.13135, May 2019.

[135] S. Group, “ALS algorithm code in PHYSL,” https://github.com/STEllAR-GROUP/phylanx/blob/master/examples/algorithms/als/als.physl, 2018.

[136] K. A. Huck, A. Porterfield, N. Chaimov, H. Kaiser, A. D. Malony, T. Sterling, and R. Fowler, “An autonomic performance environment for exascale,” *Supercomputing frontiers and innovations*, vol. 2, no. 3, pp. 49–66, 2015.

[137] C. Ţăpuş, I.-H. Chung, J. K. Hollingsworth *et al.*, “Active harmony: Towards automated performance tuning,” in *Proceedings of the 2002 ACM/IEEE conference on Supercomputing.* IEEE Computer Society Press, 2002, pp. 1–11.

[138] S. Group, “Running HPX on ROSTAM,” https://github.com/STEllAR-GROUP/hpx/wiki/Running-HPX-on-Rostam, 2017.

[139] A. Duran, J. Corbalán, and E. Ayguadé, “Evaluation of openmp task scheduling strategies,” in *OpenMP in a New Era of Parallelism*, R. Eigenmann and B. R. de Supinski, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2008, pp. 100–110.

[140] A. Duran, J. Corbalán, and E. Ayguadé, “An adaptive cut-off for task parallelism,” in *Proceedings of the 2008 ACM/IEEE Conference on Supercomputing*, ser. SC ’08. Piscataway, NJ, USA: IEEE Press, 2008, pp. 36:1–36:11. [Online]. Available: http://dl.acm.org/citation.cfm?id=1413370.1413407

[141] J. Bi, X. Liao, Y. Zhang, C. Ye, H. Jin, and L. T. Yang, “An adaptive task granularity based scheduling for task-centric parallelism,” in *Proceedings of the 2014 IEEE Intl Conf on High Performance Computing and Communications, 2014 IEEE 6th Intl Symp on Cyberspace Safety and Security, 2014 IEEE 11th Intl Conf on Embedded Software and Syst (HPCC,CSS,ICESS)*, ser. HPCC ’14. Washington, DC, USA: IEEE Computer Society, 2014, pp. 165–172. [Online]. Available: https://doi.org/10.1109/HPCC.2014.32

[142] P. Thoman, H. Jordan, and T. Fahringer, "Adaptive granularity control in task parallel programs using multiversioning," in *Euro-Par 2013 Parallel Processing*, F. Wolf, B. Mohr, and D. an Mey, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2013, pp. 164–177.

[143] S. Iwasaki and K. Taura, "A static cut-off for task parallel programs," in *2016 International Conference on Parallel Architecture and Compilation Techniques (PACT)*, Sep. 2016, pp. 139–150.

[144] S. Iwasaki and K. Taura, "Autotuning of a cut-off for task parallel programs," in *2016 IEEE 10th International Symposium on Embedded Multicore/Many-core Systems-on-Chip (MCSoC)*. Los Alamitos, CA, USA: IEEE Computer Society, sep 2016, pp. 353–360. [Online]. Available: https://doi.ieeecomputersociety.org/10.1109/MCSoC.2016.51

[145] Y. Sun, G. Zheng, P. Jetley, and L. V. Kalé, "Parsse: an adaptive parallel state space search engine," *Parallel Processing Letters*, vol. 21, no. 3, pp. 319–338, 2011. [Online]. Available: https://doi.org/10.1142/S0129626411000242

[146] P. A. Grubel, "Dynamic adaptation in hpx - a task-based parallel runtime system," Ph.D. dissertation, 2016.

[147] S. Panneerselvam and M. Swift, "Rinnegan: Efficient resource use in heterogeneous architectures," in *Proceedings of the 2016 International Conference on Parallel Architectures and Compilation*. ACM, 2016, pp. 373–386.

[148] J. S. Vetter, R. Brightwell, M. Gokhale, P. McCormick, R. Ross, J. Shalf, K. Antypas, D. Donofrio, T. Humble, C. Schuman, B. V. Essen, S. Yoo, A. Aiken, D. Bernholdt, S. Byna, K. Cameron, F. Cappello, B. Chapman, A. Chien, M. Hall, R. Hartman-Baker, Z. Lan, M. Lang, J. Leidel, S. Li, R. Lucas, J. Mellor-Crummey, P. P. Jr., T. Peterka, M. Strout, and J. Wilke, "Extreme heterogeneity 2018 - productive computational science in the era of extreme heterogeneity: Report for DOE ASCR workshop on extreme heterogeneity," USDOE Office of Science (SC) (United States), Tech. Rep., 2018.

[149] Y. Jiang, X. Shen, J. Chen, and R. Tripathi, "The complexity and approximation of optimal job co-scheduling on chip multiprocessors," *IEEE Transactions on Parallel and Distributed Systems*, vol. 22, no. 7, 2011.

[150] A. Branover, D. Foley, and M. Steinman, "Amd fusion apu: Llano," *Ieee Micro*, vol. 32, no. 2, pp. 28–37, 2012.

[151] K. L. Spafford, J. S. Meredith, S. Lee, D. Li, P. C. Roth, and J. S. Vetter, “The tradeoffs of fused memory hierarchies in heterogeneous computing architectures,” in *Proceedings of the 9th conference on Computing Frontiers.* ACM, 2012, pp. 103–112.

[152] M. Damschen, F. Mueller, and J. Henkel, “Co-scheduling on fused cpu-gpu architectures with shared last level caches,” *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 37, no. 11, pp. 2337–2347, 2018.

[153] R. Cavicchioli, N. Capodieci, and M. Bertogna, “Memory interference characterization between cpu cores and integrated gpus in mixed-criticality platforms,” in *2017 22nd IEEE International Conference on Emerging Technologies and Factory Automation (ETFA).* IEEE, 2017, pp. 1–10.

[154] S.-Y. Lee and C.-J. Wu, “Performance characterization, prediction, and optimization for heterogeneous systems with multi-level memory interference,” in *2017 IEEE International Symposium on Workload Characterization (IISWC).* IEEE, 2017, pp. 43–53.

[155] G. Dhiman and T. S. Rosing, “Dynamic voltage frequency scaling for multi-tasking systems using online learning,” in *Proceedings of the 2007 international symposium on Low power electronics and design.* ACM, 2007, pp. 207–212.

[156] R. Barik, N. Farooqui, B. T. Lewis, C. Hu, and T. Shpeisman, “A black-box approach to energy-aware scheduling on integrated cpu-gpu systems,” in *Proceedings of the 2016 International Symposium on Code Generation and Optimization.* ACM, 2016, pp. 70–81.

[157] R. Kaleem, R. Barik, T. Shpeisman, C. Hu, B. T. Lewis, and K. Pingali, “Adaptive heterogeneous scheduling for integrated gpus,” in *2014 23rd International Conference on Parallel Architecture and Compilation Techniques (PACT).* IEEE, 2014, pp. 151–162.

[158] S. Williams, A. Waterman, and D. Patterson, “Roofline: An insightful visual performance model for floating-point programs and multicore architectures,” Lawrence Berkeley National Lab.(LBNL), Berkeley, CA (United States), Tech. Rep., 2009.

[159] N. P. Jouppi, C. Young, N. Patil, D. Patterson, G. Agrawal, R. Bajwa, S. Bates, S. Bhatia, N. Boden, A. Borchers *et al.*, “In-datacenter performance analysis of a tensor processing unit,” in *Proceedings of the 44th Annual International Symposium on Computer Architecture*, 2017, pp. 1–12.

[160] J. D. McCalpin, “Stream benchmarks,” 2002.

[161] Y. J. Lo, S. Williams, B. Van Straalen, T. J. Ligocki, M. J. Cordery, N. J. Wright, M. W. Hall, and L. Oliker, "Roofline model toolkit: A practical tool for architectural and program analysis," in *International Workshop on Performance Modeling, Benchmarking and Simulation of High Performance Computer Systems.* Springer, 2014, pp. 129–148.

[162] Leek, "Cxxpolyfit: A simple library for producing multidimensional polynomial fits for c++," Aug. 2020. [Online]. Available: https://github.com/LLNL/CxxPolyFit

[163] M. E. Belviranli, F. Khorasani, L. N. Bhuyan, and R. Gupta, "Cumas: Data transfer aware multi-application scheduling for shared gpus," in *Proceedings of the 2016 International Conference on Supercomputing.* ACM, 2016, p. 31.

[164] M. A. H. Monil, "Tegra parser: A tool to parse power consumption for nvidia tegra devices," Aug. 2020. [Online]. Available: https://github.com/monil01/tegra_parser/tree/master/c_parser

[165] S. Che, M. Boyer, J. Meng, D. Tarjan, J. W. Sheaffer, S.-H. Lee, and K. Skadron, "Rodinia: A benchmark suite for heterogeneous computing," in *Workload Characterization, 2009. IISWC 2009. IEEE International Symposium on.* Ieee, 2009, pp. 44–54.

[166] K. Pulli, A. Baksheev, K. Kornyakov, and V. Eruhimov, "Real-time computer vision with opencv," *Communications of the ACM*, vol. 55, no. 6, pp. 61–69, 2012.

[167] NVIDIA, "Nvidia® vision programming interface (vpi)," Apr. 2020. [Online]. Available: https://docs.nvidia.com/vpi/index.html

[168] Q. Zhu, B. Wu, X. Shen, L. Shen, and Z. Wang, "Understanding co-run degradations on integrated heterogeneous processors," in *International Workshop on Languages and Compilers for Parallel Computing.* Springer, 2014, pp. 82–97.

[169] V. Mekkat, A. Holey, P.-C. Yew, and A. Zhai, "Managing shared last-level cache in a heterogeneous multicore processor," in *Proceedings of the 22nd international conference on Parallel architectures and compilation techniques.* IEEE Press, 2013, pp. 225–234.

[170] V. García, J. Gomez-Luna, T. Grass, A. Rico, E. Ayguade, and A. J. Pena, "Evaluating the effect of last-level cache sharing on integrated gpu-cpu systems with heterogeneous applications," in *2016 IEEE International Symposium on Workload Characterization (IISWC).* IEEE, 2016, pp. 1–10.

[171] A. Pan and V. S. Pai, "Runtime-driven shared last-level cache management for task-parallel programs," in *SC'15: Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis.* IEEE, 2015, pp. 1–12.

[172] M. Hill and V. J. Reddi, "Gables: A roofline model for mobile socs," in *2019 IEEE International Symposium on High Performance Computer Architecture (HPCA).* IEEE, 2019, pp. 317–330.

[173] F. Zhang, J. Zhai, B. He, S. Zhang, and W. Chen, "Understanding co-running behaviors on integrated cpu/gpu architectures," *IEEE Transactions on Parallel and Distributed Systems*, vol. 28, no. 3, pp. 905–918, 2016.

[174] F. Zhang, B. Wu, J. Zhai, B. He, and W. Chen, "Finepar: Irregularity-aware fine-grained workload partitioning on integrated architectures," in *Proceedings of the 2017 International Symposium on Code Generation and Optimization.* IEEE Press, 2017, pp. 27–38.

[175] J. Liu, N. Hegde, and M. Kulkarni, "Hybrid cpu-gpu scheduling and execution of tree traversals," in *Proceedings of the 2016 International Conference on Supercomputing.* ACM, 2016, p. 2.

[176] E. Rotem, A. Naveh, A. Ananthakrishnan, E. Weissmann, and D. Rajwan, "Power-management architecture of the intel microarchitecture code-named sandy bridge," *Ieee micro*, vol. 32, no. 2, pp. 20–27, 2012.

[177] S. McKee, "Reflections on the memory wall," in *Proceedings of the 1st conference on Computing frontiers*, 2004, p. 162.

[178] S. Williams, A. Waterman, and D. Patterson, "Roofline: an insightful visual performance model for multicore architectures," *Communications of the ACM*, vol. 52, no. 4, pp. 65–76, 2009.

[179] D. Marques, H. Duarte, A. Ilic, L. Sousa, R. Belenov, P. Thierry, and Z. Matveev, "Performance analysis with cache-aware roofline model in intel advisor," in *2017 International Conference on High Performance Computing & Simulation (HPCS).* IEEE, 2017, pp. 898–907.

[180] C. Yang, "Hierarchical roofline analysis: How to collect data using performance tools on intel cpus and nvidia gpus," *arXiv preprint arXiv:2009.02449*, 2020.

[181] I. B. Peng, J. S. Vetter, S. V. Moore, and S. Lee, "Tuyere: enabling scalable memory workloads for system exploration," in *Proceedings of the 27th International Symposium on High-Performance Parallel and Distributed Computing.* Tempe, Arizona: ACM, 2018, pp. 180–191.

[182] M. Umar, S. V. Moore, J. S. Meredith, J. S. Vetter, and K. W. Cameron, "Aspen-based performance and energy modeling frameworks," *Journal of Parallel and Distributed Computing*, vol. 120, pp. 222–236, 2018.

[183] I. B. Peng and J. S. Vetter, "Siena: exploring the design space of heterogeneous memory systems," in *Proceedings of the International Conference for High Performance Computing, Networking, Storage, and Analysis.* Dallas, Texas: IEEE Press, 2018, pp. 1–14.

[184] C. Alappat, J. Hofmann, G. Hager, H. Fehske, A. Bishop, and G. Wellein, "Understanding hpc benchmark performance on intel broadwell and cascade lake processors," *arXiv preprint arXiv:2002.03344*, 2020.

[185] S. Saini, R. Hood, J. Chang, and J. Baron, "Performance evaluation of an intel haswell-and ivy bridge-based supercomputer using scientific and engineering applications," in *2016 IEEE 18th International Conference on High Performance Computing and Communications; IEEE 14th International Conference on Smart City; IEEE 2nd International Conference on Data Science and Systems (HPCC/SmartCity/DSS).* IEEE, 2016, pp. 1196–1203.

[186] S. Saini and R. Hood, "Performance evaluation of intel broadwell nodes based supercomputer using computational fluid dynamics and climate applications," in *2017 IEEE 19th International Conference on High Performance Computing and Communications Workshops (HPCCWS).* IEEE, 2017, pp. 58–65.

[187] D. Molka, D. Hackenberg, and R. Schöne, "Main memory and cache performance of intel sandy bridge and amd bulldozer," in *Proceedings of the workshop on Memory Systems Performance and Correctness*, 2014, pp. 1–10.

[188] J. Hofmann, G. Hager, G. Wellein, and D. Fey, "An analysis of core-and chip-level architectural features in four generations of intel server processors," in *International supercomputing conference.* Springer, 2017, pp. 294–314.

[189] J. Hofmann, D. Fey, J. Eitzinger, G. Hager, and G. Wellein, "Analysis of intel's haswell microarchitecture using the ecm model and microbenchmarks," in *International Conference on Architecture of Computing Systems.* Springer, 2016, pp. 210–222.

[190] S. Hammond, C. Vaughan, and C. Hughes, "Evaluating the intel skylake xeon processor for hpc workloads," in *International Conference on High Performance Computing & Simulation (HPCS18)*, 2018, pp. 342–349.

[191] I. B. Peng, R. Gioiosa, G. Kestor, J. S. Vetter, P. Cicotti, E. Laure, and S. Markidis, "Characterizing the performance benefit of hybrid memory system for HPC applications," *Parallel Computing*, vol. 76, pp. 57–69, 2018.

[192] L. Yu, D. Li, S. Mittal, and J. S. Vetter, "Quantitatively modeling application resiliency with the data vulnerability factor," *ACM/IEEE International Conference for High Performance Computing, Networking, Storage, and Analysis (SC)*, 2014.

[193] "Top 500 supercomputers published at sc20," https://www.top500.org/, accessed: 2021-01-01.

[194] D. Terpstra, H. Jagode, H. You, and J. Dongarra, "Collecting performance data with papi-c," in *Tools for High Performance Computing 2009*. Springer, 2010, pp. 157–173.

[195] "Tool to enable and disable prefetch in intel processor," https://github.com/deater/uarch-configure/tree/master/intel-prefetch, accessed: 2020-09-01.

[196] "Explanation of using different page size," https://community.intel.com/t5/Software-Tuning-Performance/Explanation-of-LLC-to-DRAM-write-count-in-Haswell/m-p/1205752#M7638, accessed: 2020-09-01.

[197] "Disclosure of hardware prefetcher control on some intel processors," https://software.intel.com/content/www/us/en/develop/articles/disclosure-of-hw-prefetcher-control-on-some-intel-processors.html, accessed: 2020-09-01.

[198] I. Peng, J. Vetter, S. Moore, and S. Lee, "Tuyere: Enabling scalable memory workloads for system exploration," in *Proceedings of the 27th International Symposium on High-Performance Parallel and Distributed Computing*, 2018, pp. 180–191.

[199] J. Tramm, A. Siegel, T. Islam, and M. Schulz, "Xsbench-the development and verification of a performance abstraction for monte carlo reactor analysis," *The Role of Reactor Physics toward a Sustainable Future (PHYSOR)*, 2014.

[200] I. Karlin, "Lulesh programming model and performance ports overview," Lawrence Livermore National Lab.(LLNL), Livermore, CA (United States), Tech. Rep., 2012.

[201] Y. Kim, W. Yang, and O. Mutlu, "Ramulator: A fast and extensible dram simulator," *IEEE Computer architecture letters*, vol. 15, no. 1, pp. 45–49, 2015.

[202] P. Rosenfeld, E. Cooper-Balis, and B. Jacob, "Dramsim2: A cycle accurate memory system simulator," *IEEE computer architecture letters*, vol. 10, no. 1, pp. 16–19, 2011.

[203] T. Allen and R. Ge, "Characterizing power and performance of gpu memory access," in *2016 4th International Workshop on Energy Efficient Supercomputing (E2SC).* IEEE, 2016, pp. 46–53.

[204] C. Dave, H. Bae, S.-J. Min, S. Lee, R. Eigenmann, and S. Midkiff, "Cetus: A source-to-source compiler infrastructure for multicores," *Computer*, vol. 42, no. 12, pp. 36–42, 2009.

[205] M. Lopez, O. Hernandez, R. Budiardja, and J. Wells, "Caascade: A system for static analysis of hpc software application portfolios," in *Programming and Performance Visualization Tools.* Springer, 2017, pp. 90–104.

[206] J. Treibig, G. Hager, and G. Wellein, "Likwid: A lightweight performance-oriented tool suite for x86 multicore environments," in *2010 39th International Conference on Parallel Processing Workshops.* IEEE, 2010, pp. 207–216.

[207] M. Monil, S. Lee, J. Vetter, and A. Malony, "Comparing llc-memory traffic between cpu and gpu architectures," in *RSDHA'21: Redefining Scalability for Diversely Heterogeneous Architectures.* IEEE, 2021.

[208] "Nvidia tesla a100 white paper," https://images.nvidia.com/aem-dam/en-zz/Solutions/data-center/nvidia-ampere-architecture-whitepaper.pdf, accessed: 2021-09-01.

[209] "Amd cdna architecture," https://www.amd.com/system/files/documents/amd-cdna-whitepaper.pdf, accessed: 2021-09-01.

[210] I. Paul, W. Huang, M. Arora, and S. Yalamanchili, "Harmonia: Balancing compute and memory power in high-performance gpus," *ACM SIGARCH Computer Architecture News*, vol. 43, no. 3S, pp. 54–65, 2015.

[211] "Nvidia tesla p100 white paper," https://images.nvidia.com/content/pdf/tesla/whitepaper/pascal-architecture-whitepaper.pdf, accessed: 2021-09-01.

[212] "Nvidia tesla v100 white paper," https://images.nvidia.com/content/volta-architecture/pdf/volta-architecture-whitepaper.pdf, accessed: 2021-09-01.

[213] "Nvidia tesla gpus," https://en.wikipedia.org/wiki/Nvidia_Tesla, accessed: 2021-09-01.

[214] "Ornl experimental computing laboratory (excl)," https://excl.ornl.gov/, accessed: 2020-09-01.

[215] "Oregon advanced computing institue for science and society (oaciss)," https://blogs.uoregon.edu/oaciss/, accessed: 2021-09-01.

[216] N. Ding and S. Williams, "An instruction roofline model for gpus," in *2019 IEEE/ACM Performance Modeling, Benchmarking and Simulation of High Performance Computer Systems (PMBS).* IEEE, 2019, pp. 7–18.

[217] S. Hong and H. Kim, "An analytical model for a gpu architecture with memory-level and thread-level parallelism awareness," in *Proceedings of the 36th annual international symposium on Computer architecture*, 2009, pp. 152–163.

[218] G. Chen, B. Wu, D. Li, and X. Shen, "Porple: An extensible optimizer for portable data placement on gpu," in *2014 47th Annual IEEE/ACM International Symposium on Microarchitecture.* IEEE, 2014, pp. 88–100.

[219] T. Ben-Nun, E. Levy, A. Barak, and E. Rubin, "Memory access patterns: The missing piece of the multi-gpu puzzle," in *SC'15: Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis.* IEEE, 2015, pp. 1–12.

[220] M. Burtscher, R. Nasre, and K. Pingali, "A quantitative study of irregular programs on gpus," in *2012 IEEE International Symposium on Workload Characterization (IISWC).* IEEE, 2012, pp. 141–151.

[221] R. H. Dennard, F. H. Gaensslen, V. L. Rideout, E. Bassous, and A. R. LeBlanc, "Design of ion-implanted MOSFET's with very small physical dimensions," *IEEE Journal of Solid-State Circuits*, vol. 9, no. 5, pp. 256–268, 1974.

[222] J. S. Vetter, R. Brightwell, M. Gokhale, P. McCormick, R. Ross, J. Shalf, K. Antypas, D. Donofrio, T. Humble, C. Schuman *et al.*, "Extreme heterogeneity 2018-productive computational science in the era of extreme heterogeneity: Report for doe ascr workshop on extreme heterogeneity," USDOE Office of Science (SC), Washington, DC (United States), Tech. Rep., 2018.