Achieving Resilience and Maintaining Performance in OpenSHMEM + X Applications

A Dissertation Presented

by

Md Abdullah Shahneous Bari

to

The Thesis Committee

in Partial Fulfillment of the Requirements

for the Degree of

Doctor of Philosophy

in

Computer Science

Stony Brook University

December 2021
Stony Brook University
The Graduate School

Md Abdullah Shahneous Bari

We, the dissertation committee for the above candidate for the Doctor of Philosophy degree, hereby recommend acceptance of this dissertation.

Dr. Barbara Chapman – Dissertation Advisor
Professor, Department of Computer Science

Dr. Rezaul Chowdhury – Chairperson of Defense
Associate Professor, Department of Computer Science

Dr. Xiangmin Jiao
Associate Professor, Applied Mathematics and Statistics,
Department of Computer Science (affiliated)

Dr. George Bosilca
Research Assistant Professor and Adjunct Assistant Professor,
Innovative Computing Laboratory
University of Tennessee, Knoxville

This dissertation is accepted by the Graduate School.

Eric Wertheimer
Dean of the Graduate School
Abstract of the Dissertation

Achieving Resilience and Maintaining Performance in OpenSHMEM + X Applications

by

Md Abdullah Shahneous Bari

Doctor of Philosophy

in

Computer Science

Stony Brook University

2021

Solving real-world problems such as climate simulation in a timely fashion requires High Performance Computing (HPC) systems with tens of thousands of processors running for weeks or even months. However, the Mean Time Between Failures (MTBF) of these systems are very small (often in hours) in comparison. As a result, these long-running applications need to adopt resilience strategies to defend against such failures. However, writing a correct resilient application requires significant effort and comes with performance overheads.

We have addressed the need to help application developers write correct, resilient, and performant OpenSHMEM-OpenMP hybrid applications using the Checkpoint/Restart (C/R) method, which is the most popular resilience technique in HPC due to its low overhead and portability.

To ensure correctness, we developed static analysis techniques to
detect OpenSHMEM-specific programming errors in an application that are not detected by a generic compiler framework. Such errors include wrong-kind of memory use, double-free, and unsynchronized access of program data (data race).

To assist the checkpointing process, we answered two fundamental questions related to C/R in the context of OpenSHMEM: where to put the checkpointing calls, and what data to checkpoint. We developed a compiler-based strategy that identifies program points where it is safe to put C/R calls and determines what data to checkpoint at those program points for successful, consistent, and deterministic restart.

To maintain performance after a restart, where the execution environment may have changed, we identified OpenMP runtime parameters that significantly impact performance variability and developed an adaptive runtime framework that automatically tunes them depending on the execution environment. We also suggest approaches that may help maintain GPU performance across different execution environments.
# Contents

<table>
<thead>
<tr>
<th>List of Figures</th>
<th>ix</th>
</tr>
</thead>
<tbody>
<tr>
<td>List of Tables</td>
<td>xii</td>
</tr>
<tr>
<td>Acknowledgements</td>
<td>xiv</td>
</tr>
</tbody>
</table>

## 1 Introduction

1.1 Background .......................... 1
1.2 Resilience and Correctness in HPC  
   1.2.1 Checkpoint/restart ...... 2
   1.2.2 Correctness ............. 4
1.3 OpenSHMEM ......................... 4
1.4 OpenMP ............................. 5
   1.4.1 Data Level Parallelism ... 5
   1.4.2 Task Level Parallelism ... 5
1.5 Research Statement ............. 6
1.6 Research Contribution .......... 7
1.7 Chapter-wise Layout .......... 8

## 2 OpenSHMEM, OpenMP, and Resilience

2.1 OpenSHMEM .......................... 11
   2.1.1 PGAS Programming Models ... 11
   2.1.2 OpenSHMEM ................ 11
2.2 OpenMP ............................. 16
   2.2.1 Memory Model ................ 17
   2.2.2 OpenMP Features .......... 18
2.3 Fault Tolerance Techniques in HPC  
   2.3.1 Key Concepts ............... 19
   2.3.2 Methods ................... 20
2.4 Checkpoint-Restart Methods for Fault Tolerance  
   2.4.1 Application-level Checkpointing and Challenges ... 22
3 Checking for Common Programming Mistakes Using OpenSHMEM Checker

3.1 Common Programming Mistakes in OpenSHMEM

| 3.1.1 Violation of OpenSHMEM Semantics- Using Wrong Kind of Memory | 25 |
| 3.1.2 Double-free of Heap Allocated Symmetric Memory | 27 |
| 3.1.3 Unsafe/Unsynchronized Access to Program Data | 28 |
| 3.1.4 Potential Performance Degradation Due to Excessive Synchronization | 29 |
| 3.1.5 Deadlock Due to Missing Synchronization | 30 |
| 3.1.6 OpenSHMEM and MPI | 31 |

3.2 Clang Static Analyzer | 31 |

3.3 Framework

| 3.3.1 Property Layer | 33 |
| 3.3.2 Base Layer | 35 |
| 3.3.3 Programming Model Layer | 36 |
| 3.3.4 Why Layered Design | 36 |

3.4 Integrated Checks

| 3.4.1 Violation of OpenSHMEM Semantics- Using Wrong Kind of Memory | 37 |
| 3.4.2 Double-free | 37 |
| 3.4.3 Unsynchronized Access to Program Data | 38 |

3.5 Evaluation

| 3.5.1 Case Study: Synthetic Benchmarks | 39 |
| 3.5.2 Case Study: Transpose, MM, Mandelbrot | 40 |
| 3.5.3 Overhead Analysis | 42 |

3.6 Concluding Remarks | 42 |

4 Checkpointing OpenSHMEM Programs Using Compiler Analysis

4.1 OpenSHMEM and Safe points

| 4.1.1 Global Consistency and Safe Points | 45 |
| 4.1.2 Why Global Synchronizations are the Best Safe Points for OpenSHMEM | 48 |

4.2 Choosing The Right Data to Checkpoint | 49 |

4.3 Compiler Tool: Putting it Together

| 4.3.1 Safe Point Identification Phase | 50 |
| 4.3.2 Data Optimization Phase | 52 |
| 4.3.3 User-feedback Phase | 52 |

4.4 Results and Analysis

| 4.4.1 Transpose | 53 |
| 4.4.2 Concluding Remarks | 54 |


<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>4.4.2 Matrix Multiplication (MM)</td>
<td>56</td>
</tr>
<tr>
<td>4.4.3 Mandelbrot Set</td>
<td>56</td>
</tr>
<tr>
<td>4.5 Concluding Remarks</td>
<td>57</td>
</tr>
<tr>
<td>5 OpenMP on a Power Constrained Node: A Case Study of</td>
<td>58</td>
</tr>
<tr>
<td>OpenMP Performance Variability Across Different Execution</td>
<td></td>
</tr>
<tr>
<td>5.1 A Motivating Example</td>
<td>59</td>
</tr>
<tr>
<td>5.2 Configuring the experimental search space</td>
<td>61</td>
</tr>
<tr>
<td>5.2.1 OpenMP Runtime Configuration</td>
<td>61</td>
</tr>
<tr>
<td>5.2.2 Power Capping or Power Clamping</td>
<td>63</td>
</tr>
<tr>
<td>5.2.3 Creating the Search Space</td>
<td>63</td>
</tr>
<tr>
<td>5.2.4 Measurement Details</td>
<td>64</td>
</tr>
<tr>
<td>5.3 Experimental Setup</td>
<td>65</td>
</tr>
<tr>
<td>5.3.1 Machine Configuration</td>
<td>65</td>
</tr>
<tr>
<td>5.3.2 Packages and Libraries</td>
<td>65</td>
</tr>
<tr>
<td>5.3.3 Benchmark Applications</td>
<td>66</td>
</tr>
<tr>
<td>5.4 Impact of the Optimal Runtime Configuration on a Power Capped Node</td>
<td>67</td>
</tr>
<tr>
<td>5.5 Finding Energy Efficient Configurations</td>
<td>75</td>
</tr>
<tr>
<td>5.5.1 Finding energy efficient configurations for a specific power level</td>
<td>77</td>
</tr>
<tr>
<td>5.5.2 Finding energy efficient configurations among all power levels</td>
<td>79</td>
</tr>
<tr>
<td>5.6 Potential Impact of these findings and Concluding Remarks</td>
<td>83</td>
</tr>
<tr>
<td>6 ARCS: Adaptive Runtime Configuration Selection for OpenMP Applications</td>
<td>85</td>
</tr>
<tr>
<td>6.1 ARCS Framework</td>
<td>86</td>
</tr>
<tr>
<td>6.1.1 OpenMP runtime with OMPT</td>
<td>86</td>
</tr>
<tr>
<td>6.1.2 APEX</td>
<td>87</td>
</tr>
<tr>
<td>6.1.3 Overhead</td>
<td>89</td>
</tr>
<tr>
<td>6.2 Experimentation</td>
<td>90</td>
</tr>
<tr>
<td>6.2.1 Test System</td>
<td>90</td>
</tr>
<tr>
<td>6.2.2 Compiler &amp; Libraries</td>
<td>90</td>
</tr>
<tr>
<td>6.2.3 Benchmarks</td>
<td>90</td>
</tr>
<tr>
<td>6.2.4 Experimental Details</td>
<td>91</td>
</tr>
<tr>
<td>6.3 Results and Analysis</td>
<td>92</td>
</tr>
<tr>
<td>6.3.1 SP</td>
<td>94</td>
</tr>
<tr>
<td>6.3.2 BT</td>
<td>97</td>
</tr>
<tr>
<td>6.3.3 LULESH 2.0</td>
<td>99</td>
</tr>
</tbody>
</table>
6.4 Concluding Remarks .................................................. 102

7 Measuring the Impact of Data Placement in Modern GPUs ........................................... 104
  7.1 GPU Memory Hierarchy .............................................. 106
  7.2 CUDA .................................................................. 108
  7.3 Design of Experiments .............................................. 108
    7.3.1 GPU Machines .................................................. 108
    7.3.2 Benchmarks .................................................... 109
    7.3.3 Testing Configurations ....................................... 115
  7.4 Results and Analysis ............................................... 116
    7.4.1 Microbenchmarking ............................................ 116
    7.4.2 CUDA Kernels .................................................. 117
    7.4.3 Benchmark Applications ..................................... 121
    7.4.4 Proxy Applications ............................................ 126
  7.5 Concluding Remarks ............................................... 128

8 Related Work .................................................................. 130
  8.1 OpenSHMEMChecker .............................................. 130
    8.1.1 Program Correctness of Parallel Programs .............. 130
    8.1.2 Static Analysis Techniques for Program Correctness 131
    8.1.3 OpenSHMEM Program Analysis ............................ 131
  8.2 Compiler Assisted Fault Tolerance ................................. 131
    8.2.1 Checkpoint-Restart Techniques .............................. 132
    8.2.2 Fault Tolerance in PGAS Programming Models ......... 132
    8.2.3 Compiler and Tool Based Fault Tolerance ............... 133
  8.3 Managing OpenMP Performance through Runtime Adaptation ..................................... 134
  8.4 GPU Performance Management: Impact of Data Placement on Modern GPUs .................. 135

9 Conclusion .................................................................... 137
  9.1 Application-level C/R library ...................................... 138
  9.2 Checkpoint data optimization ..................................... 138
  9.3 Safe point to checkpoint ............................................ 138
  9.4 Maintaining energy efficiency after restart ...................... 138

Bibliography ..................................................................... 139
# List of Figures

<table>
<thead>
<tr>
<th>Figure</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>2.1</td>
<td>Memory model in OpenSHMEM. The remotely accessible data consists of: 1) Global or static variables 2) Data on the symmetric heap (allocated by shmem_malloc).</td>
</tr>
<tr>
<td>3.1</td>
<td>OpenSHMEM Checker framework</td>
</tr>
<tr>
<td>4.1</td>
<td>An example showing a safe point in an OpenSHMEM program</td>
</tr>
<tr>
<td>4.2</td>
<td>A skeleton OpenSHMEM program showing the impact of Live variable analysis for checkpointing</td>
</tr>
<tr>
<td>4.3</td>
<td>Architectural Diagram of the tool</td>
</tr>
<tr>
<td>4.4</td>
<td>Checkpoint data sizes at all the safe points in Transpose. Smaller is better (some bars for lva-optimized are not visible, because the value for lva-optimized is less than 1% of “All data”).</td>
</tr>
<tr>
<td>4.5</td>
<td>Checkpoint data sizes at all the safe points in MM. Smaller is better (some bars for lva-optimized are not visible, because the value for lva-optimized is less than 1% compared to “All data”).</td>
</tr>
<tr>
<td>4.6</td>
<td>Checkpoint data sizes at all the safe points in Mandelbrot. Smaller is better (some bars for lva-optimized are not visible, because the value for lva-optimized is less than 1% of “All data”).</td>
</tr>
<tr>
<td>5.1</td>
<td>Execution time comparison of different OpenMP runtime configurations at different power levels for the <code>compute rhs</code> parallel region of SP.</td>
</tr>
<tr>
<td>5.2</td>
<td>Execution time and energy consumption improvement using optimal configurations over the default configuration for <strong>Non-Default parallel regions</strong>. We used the OpenUH runtime.</td>
</tr>
<tr>
<td>5.3</td>
<td>Execution time and energy consumption improvement using an optimal configuration compared to the default configuration for all power levels with the OpenUH runtime.</td>
</tr>
</tbody>
</table>
5.4 SP_zsolve1 parallel region’s dynamic hardware feature comparison between the optimal and the default configurations at 85W power level. Smaller is better. ........................................ 72

5.5 Parallel regions that have the best configuration at a lower power cap. Here we show the percent improvement in execution time and energy consumption using optimal configuration at optimal power cap compared to optimal configuration at TDP. Table 5.5 ................................................................. 73

5.6 Execution time and energy consumption improvement for SP using an optimal configuration per OpenMP parallel region compared to the default configuration with the OpenUH runtime. 76

5.7 Execution time penalty and energy consumption improvement using the best energy efficient configurations over the optimal configurations for Non-Default parallel regions at TDP power level (115W). We used the OpenUH runtime. .......................... 78

5.8 Execution time penalty and energy consumption improvement using the most energy efficient configurations over the optimal configurations across all power levels. We used the OpenUH runtime. ................................................................. 81

6.1 ARCS framework, based on the original APEX design. ........ 88

6.2 Feature comparison between the default and ARCS-Offline strategy at TDP power level. Comparison is done on four of the most time consuming regions of SP. Y-axis shows the normalized feature value. Smaller value is better. ...................... 96

6.3 Application level execution time and package energy comparison among the default and ARCS strategies in SP at data set B. Comparison is done on five different power levels. Smaller value is better. ........................................................................ 97

6.4 Execution time and energy consumption comparison of ARCS strategies and the default strategy in data set C of SP. Smaller value is better. ................................................................. 97

6.5 Feature comparison between the default and ARCS-Offline strategy at TDP power level for compute_rhs region of BT. Smaller value is better. ................................................................. 98

6.6 Application level execution time and package energy comparison among the default and ARCS strategies in BT with data set B. Comparison is done on five different power levels. Smaller value is better. ................................................................. 99
6.7 Application level execution time and package energy comparison among the default and ARCS strategies in LULESH, for mesh size 45. It shows results in both architectures. Smaller value is better.

6.8 OpenMP events data for top 5 time consuming regions from LULESH.

6.9 Feature comparison among default and ARCS strategies on CalcFbHourglassForceForElems_region.

7.1 GPU memory hierarchy (from NVIDIA [1]).

7.2 Bandwidth comparison of different GPUs.

7.3 Speedup achieved using constant memory compared to global memory.

7.4 Execution time using Constant memory.

7.5 Speedup achieved using shared memory compared to global memory.

7.6 Execution time using Shared memory.

7.7 Speedup achieved using 2D texture memory compared to global memory.

7.8 Execution time using 2D texture memory.

7.9 Speedup of SpMV.

7.10 Speedup of MM.

7.11 Speedup of CFD.

7.12 LULESH kernels using size 4.

7.13 LULESH kernels using size 45 and 90.
List of Tables

3.1 Description of some basic OpenSHMEM concepts and routines 26
3.2 Comparison of programming mistakes in MPI and OpenSHMEM 32
3.3 Overhead analysis. Compile-time shown in secs. 41

4.1 Safe points in OpenSHMEM API. It’s safe to checkpoint right after a successful call to these functions. 47

5.1 Top 5 supercomputers as of June 2021 (Top 500 list) 59
5.2 OpenMP Loop Scheduling Policy Overview 62
5.3 Configuration parameter details 64
5.4 Optimal configuration for `SP_z_solve_1` parallel region on different power caps & Performance penalty of using an optimal configuration of a certain power cap to another 70
5.5 Best configuration across all power levels. We show the power level, no. of threads (T), scheduling policy (S) and chunk size (C) that provides the best execution time. 74
5.5 Best configuration across all power levels. We show the power level, no. of threads (T), scheduling policy (S) and chunk size (C) that provides the best execution time. 75
5.6 Best Execution Time and Energy efficient configuration for Non-Default parallel regions at TDP power level (115W). We show no. of threads (T), scheduling policy (S) and chunk size (C) that provides the best execution time and the best energy efficiency. 79
5.7 Percent Package Energy improvement with 10% to 90% performance penalty compared to the optimal configuration across all power levels. 80
5.8 Best execution time and energy efficient configuration across all power levels using OpenUH runtime. We show power level, no. of threads (T), scheduling policy (S) and chunk size (C) that provides the best execution time and the best energy efficiency. 84
6.1 Set of ARCS search parameters for OpenMP parallel regions.

6.2 Optimal configuration chosen by ARCS-Offline strategy for SP regions.

7.1 GPU memory types

7.2 Experimental machines with GPUs

7.3 Key specifications of selected GPUs of different generations

7.4 Three representative kernels and their input data sizes

7.5 SpMV benchmark memory configurations

7.6 MM Benchmark Memory Configurations

7.7 CFD Benchmark Memory Configurations

7.8 Properties of selected LULESH kernels

7.9 Example thread block and grid configurations

7.10 Measured memory specification
Acknowledgements

Firstly, I would like to thank my research advisor Dr. Barbara Chapman, my mentors - Dr. Abid Malik and Tony Curtis, for taking me under their wings, and providing me with ample guidance within my research group - Exascale Programming Models Laboratory.

I would also like to thank my collaborators - Dr. Ahmad Qawasmeh from The Hashemite University, Jordan, Dr. Kevin Huck and Dr. Nicholas Chaimov from The Oregon State University, Dr. George Bosilca from the University of Tennessee, Knoxville for their help on several topics.

Thanks is also due to my Dissertation committee member Dr. Rezaul Chowdhury, Dr. Xiangmin Jiao, and Dr. Dr. George Bosilca for their valuable time and feedbacks.

I would also like to thank the Institute for Advanced Computational Science (IACS) at Stony Brook University, National Science Foundation (NSF), US Department of Energy (DOE), Lawrence Livermore National Laboratory, Brookhaven National Laboratory, and Research Computing Data Core at University of Houston for providing me with financial support and an excellent environment for research.

Last but not least, I would like to thank my parents, my friends, and my wife for their immense patience and support throughout this journey.
To my parents....

Md. Shamsul Bari
&
Umme Salma Mst. Shahnaz Bari
Chapter 1

Introduction

1.1 Background

Determining how to build and deploy computer systems to enable recovery from failure, and what importance to place upon such capabilities (e.g., architectural redundancy in the network), is a perennial problem for computer systems designers. The greater the size and complexity of a system, the more likely it is that one or more components will fail during the execution of an application code. HPC (High Performance Computing) systems are inherently large and with the growing heterogeneity of processors and memory subsystems they are also increasingly complex. In the race to achieve higher computational power, the number of components continues to increase and in consequence, the Mean Time Between Failure (MTBF) of these systems is decreasing \[2, 3\]. The smaller the MTBF, the less stable the system is. Especially for long-running applications, whose execution may take significantly longer than the MTBF of the platforms they are deployed on, it is vital that the interim results of a computation are not lost if a system fault occurs. This could otherwise lead to not just wasted computing resources and energy consumption, but potentially greater problems if the results are time-sensitive (e.g., weather forecasting). Thus strategies for resilience - in hardware, system software, and application software - are essential in HPC. Since an application program may run on a variety of platforms with differing MTBFs, application developers who need to ensure that failure does not compromise the timely production of results must pro-actively adopt a strategy to accomplish this. In other words, in most HPC contexts, the safe approach is to build resilience into the application code.

Unfortunately, most traditional HPC programming models (e.g., MPI, OpenSHMEM, GASNet) lack the support for resilience that is built into their
Data Analytics counterparts (e.g., Hadoop, Spark). Application developers must therefore rely on other means to achieve fault tolerance.

1.2 Resilience and Correctness in HPC

The collection of techniques for keeping applications running to a correct solution in a timely and efficient manner despite underlying system faults.

Snir et al. [4] on Resilience

According to Snir et al. [4], Resilience is essentially a collection of techniques. In the HPC context, researchers have explored different approaches to resilience. These solutions fall into following categories: prevention, prediction, tolerance, detection, containment, recovery, diagnosis, and repair [4]. Prevention methods focus on reducing the number of error occurrences while repair methods focus on repair of the execution environment. Detection and diagnosis methods focus on detection of the error and root cause behind it, respectively, while prediction methods try to predict future errors based on previous information [3]. Containment methods’ main focus is to reduce the impact of the failure; tolerance methods try to make sure that errors do not lead to failures. Recovery methods’ main objective is to bring the system back to the correct state.

In the HPC application domain, recovery methods are most prominent, especially backward recovery or Checkpoint/Restart methods to recover the application state from a failure.

1.2.1 Checkpoint/restart

Checkpoint-Restart (C/R) is the most-used approach to implement fault tolerance in HPC applications due to its versatility, and relatively low overhead [5]. Checkpoint-Restart follows a simple strategy; one saves the state of the program as a so-called checkpoint at certain intervals during the execution. If the program terminates for any reason (hardware or software failure/faults) prior to completion, it restarts from the last checkpoint using the saved information instead of starting again from the beginning. Since the successful restart of an application relies on the data saved during the check-pointing process, saving
enough information to restart is important.

However, care must be taken with respect to the amount of data saved during the check-pointing process. Since check-pointing results in a lot of data movement, oftentimes across nodes, the communication and I/O overhead it incurs can potentially result in severe performance degradation. Hence it is very important, but also extremely difficult, to determine the right amount of information to save, and a reasonable frequency at which to do so.

Moreover, there is another important challenge involved in implementing the Checkpoint-Restart process, that of finding suitable places to checkpoint. One has to make sure that the program is in a consistent and deterministic state with respect to computation, communication, and data when the state of the program is saved at a checkpoint, otherwise it may result in a non-deterministic and potentially incorrect restart of the program. Finding these consistent and deterministic “safe points” can be tricky in an application that uses a parallel programming model such as MPI. However, it can be significantly tougher for an application that uses a programming model with asynchronous communications such as OpenSHMEM and other PGAS programming models (e.g., GASNet, UPC++).

Hence, the successful deployment of C/R depends on solving these consistency and data optimization challenges efficiently and effectively. However, to do so one has to have a good understanding of not just the programming model, and the checkpointing library being used, but also of the entire application, which can be very difficult for large scale scientific applications that have hundreds of thousands of lines of code. Earlier research has coupled an understanding of the semantics of a programming interface with compiler techniques \cite{6-12} to ease the burden on application developers by identifying safe points for checkpointing, suggesting which data to save, and so on.

One final challenge that must be considered once the consistency and data optimization challenge is addressed, is maintaining the performance after a restart from a failure. This is crucial because a failure can significantly change the execution environment of the application (e.g., new nodes added to the computation may have different architectural configurations, application running on a power constrained environment may have a different power constraint after restart) which can significantly impact the intra-node performance which in-turn can impact the overall application performance. Hence maintaining the in-node performance after restart is extremely important.

In C/R, addressing data optimization and consistency challenges require primary focus on inter-node (distributed) aspect of application, hence, from a programming model perspective, on the distributed programming model used in the application. On the other hand, addressing the intra-node performance
maintainability challenge requires primary focus on the intra-node programming model. In this work, I chose OpenSHMEM to be the driving force in the distributed programming model front, and OpenMP on the intra-node programming model front.

1.2.2 Correctness

A discussion on resilience is not complete without the discussion on correctness. As I briefly described one aspect of the correctness challenge in the previous section in the form of consistency problem, the discussion on correctness goes far beyond that, especially in HPC applications, due to the complexities associated with parallel programming.

However, the library-based nature of some parallel programming models (e.g., MPI, OpenSHMEM) results in lack of error-detection support from generic compiler frameworks, which often is the primary source of error-detection for application developers. As a result, the burden of error detection with respect to the program’s parallelization falls on the application developers and requires them to be extra vigilant to avoid bugs in their program or rely on runtime error detection which can be extremely difficult. Therefore, specialized static error-detection support for these programming models is necessary.

1.3 OpenSHMEM

OpenSHMEM [13] [14] is a partitioned global address space (PGAS) programming model realized in the form of a library of routines (analogous to MPI). OpenSHMEM programs consist of processes (Processing Elements, or PEs; analogous to MPI ranks) that communicate using point-to-point or collective operations. Data in the PEs can be marked as “symmetric”, meaning it is exposed to the communication layer between PEs (typically an inter-connect such as Infiniband [15], or shared memory in a node) and can be read/written directly by other PEs. Infiniband and other networks enable native one-sided communication in hardware that frees the application or OS from dealing with progress issues.

The OpenSHMEM API defines a library interface with routines to satisfy the communication needs of parallel applications. Those of most relevance to this work include: point-to-point RDMA and atomic operations; and collective memory management, communication, and synchronization operations.
1.4 OpenMP

OpenMP is a standard API for intra-node parallelism. It provides a directive-based programming approach for generating parallel versions of programs from sequential ones. The users express the parallelism in their program through a sequence of compiler directives; an OpenMP compliant compiler takes that program and generates the parallel version of it. It gives the application developers an easy and incremental way to parallelize their programs.

OpenMP can be used to express two types of parallelism,

- Data Level Parallelism
- Task Level Parallelism

1.4.1 Data Level Parallelism

Data level parallelism is where multiple workers work on the same code segment simultaneously but on different parts of the data. One of the most prominent examples of data level parallelism is loop-level parallelism. Each iteration of a loop usually works on the same loop body but on different data (e.g. different segments of an array). So in loop-level parallelism, different iterations of the loop (hence, different segments of the data) are divided among different workers and they are executed concurrently by those workers, hence the name data level parallelism. This can only be done if the individual iteration is independent of each other.

1.4.2 Task Level Parallelism

In Task level parallelism, the work is divided into different tasks/working units; these tasks are in turn executed by the worker threads. A task usually has a code region and data environment. Task level parallelism can be used to handle irregular parallelism in OpenMP.

Although task level parallelism is gaining a lot of plaudits now-a-days, to this moment most of the OpenMP applications use data level parallelism or in other words loop-level parallelism. Therefore, to maximize the performance of an OpenMP application, maximizing the performance of these parallel loops is of utmost importance.
1.5 Research Statement

OpenSHMEM was designed to enable high performance by exploiting the support for Remote Direct Memory Access (RDMA) available in modern network interconnects. It allows for highly efficient data transfers without incurring the software overhead that comes with message-passing communication. As a result, it is highly efficient for applications with irregular communication pattern (e.g., Graph applications). With greater focus on Graph-based applications in recent times, OpenSHMEM is increasing in popularity. On top of the recent trends of nodes getting more and more complex (CPU, GPU/s, and FPGAs in a single node) and powerful, researchers are also looking into the prospect of OpenSHMEM + X (e.g., OpenMP, CUDA) hybrid approaches to get the best performance out of their applications. In OpenSHMEM+X hybrid approach, one tries to divide the work across the nodes using OpenSHMEM while X (e.g., OpenMP, CUDA) is used for intra-node parallelism just like in MPI + OpenMP hybrid. Running these applications in scale would require good resilience infrastructure. However, resilience support for OpenSHMEM is almost non-existent, or at best, at its infancy. Very few works have targeted OpenSHMEM resilience. Since C/R is the most prominent approach to resilience in HPC, developing a C/R infrastructure for OpenSHMEM-OpenMP is a logical step forward. Developing necessary infrastructures for C/R requires focus on multiple fronts:

- Development of C/R libraries
- Development of tools infrastructure: development of error-detection support, making the checkpointing process easier for the application scientists, optimize the amount data to be saved for better performance, and last but not least maintaining application performance and energy consumption after restart.

In this work, we focus on the tools aspect. We assume the availability of a C/R library. Development of the tools infrastructure requires significant effort, especially from the OpenSHMEM perspective.

To provide C/R support the necessary tools would have to address several key research questions which can be categorized as follows:

- Tools for error-detection: The main focus of these tools is to detect OpenSHMEM-specific errors that are not detected by generic compiler frameworks. Developing these tools require:
  - Identification and understanding of the common programming errors in OpenSHMEM programs, and
– Development of strategies to detect them.

• Tools to support the checkpointing process: The primary purpose of these tools is to help the application scientists write correct, resilient, and performant applications using OpenSHMEM. Some research questions these tools need to answer includes but not limited to:

  – Where to insert the checkpointing calls? In other words, identify ‘safe points’ in OpenSHMEM programs to put checkpointing calls.
  – What data to checkpoint? Or identify which data needs to be saved at a certain checkpoint.

• Tools to maintain performance and energy consumption after restart:

  – The focus of these tools is to maintain in-node performance after a restart. Since, most recent HPC nodes contain both CPUs and GPUs, these tools should be able to tackle performance maintainability/adaptation questions related to both CPU and GPU.

    * CPU: One possible way to investigate the adaptation of CPU performance would be through programming model parameters. Since we are focusing on OpenMP as the intra-node programming model, one possible way would be to investigate how to adapt CPU performance of OpenMP applications.

    * GPU: Tools should be able to address, how to maintain GPU performance across different execution environments, what aspect of a GPU is most responsible for performance variance across different execution environment?

1.6 Research Contribution

Our research contribution can be categorized into two main categories:

• Work focused on writing correct, resilient, and performant programs using OpenSHMEM:

  – Identify and understand the common programming errors in OpenSHMEM.

  – Develop of a static analysis tool to detect those error in an OpenSHMEM application. The tools is based on LLVM’s [16] Clang Static Analyzer Framework. LLVM is an open-source compiler framework which is becoming very popular in both academia and industry.
– Analyze the OpenSHMEM programming model from a resilience perspective and define “safe points” for inserting checkpointing library calls into OpenSHMEM applications.

– Utilize a compiler’s data analysis to determine what data to checkpoint.

– Develop an LLVM-based tool to support the insertion of checkpoints into OpenSHMEM code. It utilizes a safe point analysis and data analysis to provide checkpointing suggestions (where to checkpoint, what data to checkpoint) to the OpenSHMEM application developer.

• Work focused on maintaining in-node performance:

  – Identify parameters (e.g., OpenMP runtime parameters) that impacts OpenMP (intra-node) performance and energy efficiency.

  – Provide a quantitative experimental study on the impact of OpenMP runtime environment on performance and energy consumption under different execution environments. We used different power levels of a processor as a test bed. This study provides key insight into how the runtime configurations can be used to direct the optimization to a specific goal, such as execution time, energy consumption or a combination of both.

  – Develop a fully automated adaptation framework, ARCS (Adaptive Runtime Configuration Selector), to enable performance adaptation across different execution environments. Performance adaptation is done by choosing the right OpenMP runtime configurations for OpenMP parallel regions to optimize HPC applications under different execution environments. To the best of our knowledge, ARCS is the first fully automatic framework that chooses OpenMP runtime configurations with no involvement of the application programmer.

  – Identify which aspect (e.g., data placement) impacts the GPU performance most across different execution environments (e.g., different generations of GPUs).

  – Explore the impact of data placement across generations of GPUs using a range of microbenchmarks and computation kernels.

1.7 Chapter-wise Layout

This section describes the layout of this literature.
Chapter 2 provides a background on the concepts related to OpenSHMEM, OpenMP, and Resilience.

Chapter 3 explores the approach for OpenSHMEM-specific error checking.

Chapter 4 explores the compiler assisted checkpointing approach for OpenSHMEM applications.

Chapter 5 provides a detailed study of OpenMP applications performance variability across different execution environments.

Chapter 6 discusses the adaptive runtime tool to automatically manage an OpenMP application performance across different execution environments.

Chapter 7 explores the performance variability impact in GPUs and how to tackle that in modern GPUs.

Chapter 8 provides the state of the art of the literature related to this work.

Chapter 9 provides concluding remarks.
Chapter 2

OpenSHMEM, OpenMP, and Resilience

With the paradigm shift of using parallelism as the main source of performance improvement instead of clock frequency in modern computing systems, parallel programming models have gained prominence more than ever [17]. Although HPC systems have historically been using parallelism as the main source of performance improvement, availability of parallelism at the different levels of a system (e.g., node-level, thread-level, instruction-level) have made the HPC researchers to rethink parallelism and parallel programming models. As a result, different kinds parallel programming models exist to serve the need of different levels of parallelism and different types of applications. These programming models can be categorized based on different criteria. However, most parallel programming models fall into two broad areas:

- Distributed memory model: These programming models are primarily used to achieve parallelism across nodes. Different types of distributed memory model exist based on their communication and memory model such as, Message passing (e.g., MPI), Partitioned Global Address Space (e.g., OpenSHMEM).

- Shared memory model: These programming models are primarily used to achieve parallelism inside a node. OpenMP, OpenACC, pthread are examples of such programming model. However, with the advent of GPUs and other accelerators, a lot these models (e.g., OpenMP) supports multiple the components inside a node and therefore are not ‘shared memory’ anymore. Intra-node parallel programming model would be a more correct term, however, for simplicity we would these term interchangeably throughout this document.
Component-specific model: Some programming models are specifically developed for certain components (e.g., CUDA is used to program NVIDIA GPUs).

With the HPC systems moving towards more complex and heterogeneous architectures and enabling parallelism at multiple levels, hybrid of these programming models are often used to get the best out of these systems [18]. The hybrid models are diverse and most of them are out of the scope of this work. Due to relevance to this work we will be exploring OpenSHMEM, a PGAS programming model and OpenMP, an intra-node programming model in details.

2.1 OpenSHMEM

2.1.1 PGAS Programming Models
The Partitioned Global Address Space (PGAS) [19] is a parallel distributed programming model that typically uses a Single Program Multiple Data (SPMD) approach to provide local- and global-views of program data, split across communicating processes on 1 or more compute nodes. PGAS models often take advantage of network capabilities such as Remote Direct Memory Access (RDMA) [20] to allow efficient data movement that is decoupled from synchronization. The PGAS family includes libraries and languages: the former include Global Arrays, GASNet, and OpenSHMEM; and the latter, which are built on these libraries, include UPC, UPC++, and Fortran’s Co-Arrays.

2.1.2 OpenSHMEM
The OpenSHMEM Specification [13, 14] defines a library-based PGAS programming model/interface for C and C++. OpenSHMEM is an open-source community effort to develop a software ecosystem for the scientific community and hardware vendors to ensure portability. There is a number of open-source implementations, e.g. OSSS-UCX [21], Ohio State University [22], Sandia National Lab [23], Oak Ridge National Lab [24], Open-MPI [25]; and some from vendors, e.g. HPE/Cray [26], NVIDIA/Mellanox [27], IBM [28].

The OpenSHMEM API defines a library interface with routines to satisfy the communication needs of parallel applications. Those of most relevance to

---

1The OpenSHMEM copyright is owned by Open Source Software Solutions Inc., a non-profit organization, under an agreement with Hewlett Packard Enterprise.
this work include: point-to-point RDMA and atomic operations; and collective memory management, communication, and synchronization operations.

Listing 2.1 shows an example of a skeleton OpenSHMEM program that performs halo exchange. OpenSHMEM routines are painted red. It starts by initializing the OpenSHMEM library and necessary resources using `shmem_init` (analogous to `MPI_Init` in MPI) and finishes by releasing all resources used by the OpenSHMEM library using `shmem_finalize`. In between, other OpenSHMEM routines are used to perform collective memory management (`shmem_malloc`, `shmem_free`), point-to-point RDMA (`shmem_put`), and synchronization (`shmem_barrier_all`).

```c
#include <shmem.h>
...

int main()
{
    shmem_init(); // Initialize OpenSHMEM Library
    ...
    int* privMem = (int*) malloc(...);
    int* symMem = (int*) shmem_malloc(...);
    for (timeStep=0; timeStep<MAX_TIMESTEP; timeStep++)
    {
        do_some_computation();
        // Halo exchange to neighbor, One-sided write
        shmem_put(symMem, privMem, numElem, neighborPE);
        shmem_barrier_all(); // Global barrier
    }
    ...
    free(privMem);
    shmem_free(symMem);
    shmem_finalize(); // Finalize OpenSHMEM Library
    return 0;
}
```

Listing 2.1: OpenSHMEM program performing halo exchange

OpenSHMEM Memory Model

OpenSHMEM programs consist of processes (Processing Elements, or PEs; analogous to MPI ranks) that communicate using point-to-point or collective operations. Data in the PEs can be marked as “symmetric”, meaning it is exposed to the communication layer between PEs (typically an inter-connect such as Infiniband [15], or shared memory in a node) and can be read/written directly by other PEs. Infiniband and other networks enable native one-sided communication in hardware that frees the application or OS from dealing with progress issues. In OpenSHMEM, Symmetric memories can be defined in multiple ways, by declaring variables as global or static or by allocating memory in
the heap using OpenSHMEM-specific allocation routines (e.g., \texttt{shmem_malloc} which is analogous to \texttt{C malloc}). Figure 2.1 shows the OpenSHMEM memory model.

Figure 2.1: Memory model in OpenSHMEM. The remotely accessible data consists of: 1) Global or static variables 2) Data on the symmetric heap (allocated by \texttt{shmem_malloc})

### Library Setup and Query Routines

OpenSHMEM provides:

- **Initialization/de-initialization** routines to setup and release necessary resources for the OpenSHMEM library. All OpenSHMEM calls must be in-between the initialization and de-initialization.

- **Query** routines to query about specific information (e.g., PE number, total PE available).

- **Accessibility** routines that provides PE and data accessibility information.

### Remote Memory Access (RMA) Routines

Two of the main OpenSHMEM RMA routines are the generic forms \texttt{shmem_put} and \texttt{shmem_get} which allow 1 PE to respectively write to, or read from, the symmetric memory of another PE.

\footnote{Explicit typed versions also exist for basic C types such as “int”, “short”, “float”. The full list is in the specification.}
“Put” routines can allow for highly asynchronous low overhead access to another PE’s symmetric memory, which can be exploited by applications with irregular/sparse communication patterns [29]. However, this asynchrony means that “put” operations do not guarantee completion when they return. So the application must provide later-synchronization to ensure consistency when data is needed, with the most common method being a global barrier. Listing 2.2 shows an example of using OpenSHMEM RMA routines.

```c
int main(int argc, char const *argv[])
{
    shmem_init(); // Initialize OpenSHMEM Library
    ...
    // Allocate local/private memory
    int *src = (int *)malloc(...);
    // Allocate symmetric memory
    int *symMem = (int *)shmem_malloc(...);
    ...
    // Write to symMem of PE no. ‘1’, Write is not guaranteed to complete without proper synchronization
    shmem_put(symMem, src, 1, 1);
    ...
    // Global synchronization barrier, shmem_put in line 13 or any communication initiated in the default context before this point is guaranteed to complete after this
    shmem_barrier_all();
    // Reading from the updated symMem of PE no. ‘1’
    shmem_get(src, symMem, 1, 1);
    ...
    ...
    shmem_finalize(); // Finalize OpenSHMEM Library
    return 0;
}
```

Listing 2.2: An example of OpenSHMEM RMA routines

**Atomic Routines**

- **Swap**: Local PE reads the old value of the symmetric data object on a remote PE and copies a new value to it.

- **Set**: Atomically sets the value of a remote data object.
• Increment: Local PE adds 1 to the symmetric data object on the remote PE.

• Add: Local PE adds specified value to the symmetric data object on the remote PE.

• OR: Local PE performs a bitwise OR between the specified value and the symmetric data object on the remote PE and writes the result on the remote PE.

• AND: Local PE performs a bitwise AND between the specified value and the symmetric data object on the remote PE and writes the result on the remote PE.

• XOR: Local PE performs a bitwise XOR between the specified value and the symmetric data object on the remote PE and writes the result on the remote PE.

• Compare and Swap: Local PE reads the old value of the symmetric data object based on a value to be compared and copies a new value to the symmetric data object on the remote PE.

• Fetch and Increment: Local PE reads the old value of the symmetric data object on the remote PE and adds 1 to the symmetric data object on the remote PE.

• Fetch and Add: Local PE adds specific value to the symmetric data object on the remote PE and copies the old value.

• Fetch and OR: Local PE atomically perform a bitwise OR operation on a remote data object and copies the old value.

• Fetch and AND: Local PE atomically perform a bitwise AND operation on a remote data object and copies the old value.

• Fetch and XOR: Local PE atomically perform a bitwise XOR operation on a remote data object and copies the old value.

Synchronization and Ordering Routines

OpenSHMEM synchronization and ordering/completion is discussed below:

• shmem_barrier, shmem_barrier_all Provide collective synchronization over a subset of PEs and all PEs respectively.
**shmemquiet** The PE calling quiet ensures remote completion of remote access operations and stores to symmetric data objects.

**shmemfence** The PE calling fence ensures ordering of Put, AMO, and memory store operations to symmetric data objects with respect to a specific destination PE.

### Collective Communication Routines

OpenSHMEM provides collective routines for Broadcast, Collection, Reduction, All-to-All, synchronization/ barrier, and symmetric memory management. All or subsets of PEs determined by the team (analogous to communicator in MPI) can participate in the collective operations.

### Memory Management Routines

OpenSHMEM also provides a set of collective routines for symmetric memory management. They are primarily of three types:

- **Allocation**: Collective allocation of symmetric data by library provided routines.
- **Deallocation**: Collective deallocation of symmetric data by library provided routines.
- **Reallocation**: Collective reallocation of symmetric data by library provided routines.

### 2.2 OpenMP

OpenMP is a high-level directive-based programming model for intra-node parallel programming. The model consists of a set of directives, runtime library routines, and environment variables. The user usually inserts the directives into the existing sequential code, with minor changes or no changes to the code. A OpenMP-compliant compiler takes this directive annotated code and generates the parallel version of the code. Runtime library routines and environment variables are usually used to control different aspects of the execution model at runtime.

OpenMP adopts the fork-join model. The model begins with an initial main thread, then a team of threads will be forked when the program encounters a parallel construct, and all other threads will join the main thread at the end of the parallel construct. In the parallel region, each thread has its own
private variables and does the work on its own piece of data. The communication between different threads is performed via shared variables. In the case of a data race condition, different threads will update the shared variable atomically. Starting from 3.0, OpenMP introduced task concept that can effectively express and solve the irregular parallelism problems such as unbounded loops and recursive algorithms. To make the task implementation efficient, the runtime needs to consider the task creation, task scheduling, task switching, and task synchronization, etc. OpenMP 4.0 released in 2013 includes support for accelerators.

Listing 2.3 shows an example of OpenMP program that multiplies two matrices, a and b, and storing the result in a third matrix, c. OpenMP directives are painted red.

```c
#include <omp.h>
...
int main()
{
    #pragma omp parallel private(i,j,k) shared(a,b,c)
    {
        #pragma omp for schedule(auto)
        for (i = 0; i < N; i++)
            for (k = 0; k < K; k++)
                for (j = 0; j < M; j++)
                    c[i][j] = c[i][j]+a[i][k]*b[k][j];
    }
    /* end omp parallel */
    ...
    return 0;
}
```

Listing 2.3: OpenMP program performing matrix-matrix multiplication

### 2.2.1 Memory Model

OpenMP is based on the shared-memory model with support for GPUs; hence, by default, data is shared among the threads and is visible to all of them. Sometimes, however, one needs variables that have thread-specific values. When each thread has its own copy of a variable, so that it may potentially have a different value for each of them, we say that the variable is private. For example, when a team of threads executes a parallel loop, each thread needs its own value of the iteration variable. This case is so important that the compiler
enforces it; in other cases the programmer must determine which variables are shared and which are private. Data can be declared to be shared or private with respect to a parallel region or work-sharing construct.

2.2.2 OpenMP Features

Parallel Regions

OpenMP programs consists of both sequential and parallel part. Execution starts with a single thread of control like many sequential programs. The #pragma omp defines a line of code to be an OpenMP parallel directive, which specifies a structured block of code that should be treated as a parallel region. In this case the parallel region is enclosed within .

#pragma omp parallel [clause[[,]clause] ...]
structured-block

When the initial thread of control reaches a parallel directive, it creates a team of threads to execute the associated parallel region, which is the code dynamically contained within the parallel construct. The thread encountering the parallel construct becomes the master of the new team. Each thread in the team is assigned a unique thread number (also referred to as the “thread id”).

Each thread is allowed to follow a different path of execution. This construct ensures that computations are performed in parallel, but does not distribute the work of the region among the threads in a team. In fact, the work will be replicated if the programmer does not use the appropriate syntax to specify this action. A list of clauses may be used along with the parallel construct, they are: private, shared, firstprivate, copyin, reduction, num threads, if, default clauses. Some of the clauses (e.g., private, shared, firstprivate) designates how variables are to be treated in the parallel region. Parallel regions must have only one entry point and one exit point sicne they are formed with a structured block. A program is non-conforming if it branches into or out of a parallel region. Synchronization of threads in a team is achieved through another set of constructs. At the end of a parallel region, there is an implied barrier that forces all threads to wait until the work inside the region has been completed. Only the initial thread continues execution after the end of the parallel region. A barrier is a point of execution where threads must wait for all other threads is the current team before proceeding. While many constructs have implicit barriers, an explicit barrier is accomplished with the barrier directive.
Work Sharing

Worksharing is a term used in OpenMP to define the distribution of work among the threads. Work can be divided among multiple threads, different threads can operate on different portions of a shared data structure or different threads can even work on totally different tasks. A work-sharing region must bind to an active parallel region, otherwise the work-sharing region is simply ignored.

Loop Parallelism

A significant amount of work occurs in loops, therefore also provides a huge incentive to exploit any parallelism available there in the form of independent work (iteration of loops) that can be executed concurrently. If an iteration does not depend in some way on the outcome of a previous iteration, they can be executed concurrently. Existing loop-carried dependences can often be removed with some reorganization to reveal concurrency. The most commonly used worksharing construct is the loop construct, which is marked with:

```
#pragma omp for
```

where iterations of a loop can be distributed among multiple threads.

Only for loops in C/C++ and do loops in Fortran is supported since it requires the number of iterations in the loop to be countable with an integer with a fixed increment. How the work (iterations) are divided among the threads can be controlled by scheduling clauses (more on this in Chapter 5 and 6).

2.3 Fault Tolerance Techniques in HPC

2.3.1 Key Concepts

Faults, errors, and failures are concepts that are interchangeably used most of the time. But according to [4]:

- Fault: is the cause of errors. Faults can be active or inactive depending on whether they cause errors or not. They usually exist on one component of the system, and they can be hard, i.e. a system fault, or soft.
- Error: is a state of system which does not work properly and may start a failure. They can be detected, or stay silent/latent.
• Failure: brings the system to unwanted or incorrect service. Failures may be perceived differently or identically by the users. They can also be detected or not.

Many mechanisms exists to detect, reduce, predict, recover from, or repair faults, errors or failures. We explain some of these techniques and concepts in detail in this section.

2.3.2 Methods
Resilience in HPC achieved through a collection of techniques. These techniques address different aspects of resilience such prevention, prediction, tolerance, detection, containment, recovery, diagnosis, and repair. Some of the specific techniques used in HPC is detailed below:

Migration
This is a prevention method. In this method, based on prediction algorithms, part of the program that may be affected by a possible failure is migrated to another node or process. System error rates and scalability data based on the system log files are used to predict future failures. However, this method suffers from the lack of precise predictions as some failures are unreported or the reporting not accurate enough.

Redundancy
The idea of this method is to provide multiple instances of hardware or process or other components, in case one or some of them fail. Some of the notable approaches are: Process pair, and Triple Modular Redundancy (TMR)

Failure semantics
This type of method usually contains the anticipated failures and respective recovery actions. The accuracy of this kind of methods depend on the ability of the designer to provide a proper set of failures and proper actions associated with each of them.

Failure masking
Ensures that the system still provides for user needs even in case of a failure. Appropriate for systems where on-time service to user is more important than recoveries and detections.
Recovery

Primary focus is to escape an erroneous or failed state and return to a safe one. Two categories:

- **Forward Recovery**: Brings the system back to a safe and error-free state. Requires multiple copies of relevant components, such as multiple modules running the same part of an application. Used systems where on-time service to user is more important than recoveries and detections.

- **Backward Recovery**: Move the state of system to an older safe one and restart from there, and therefore need one or multiple checkpoints of previous states to be able to return to them. Another name for this method is Checkpoint/Restart (C/R). It is one of the most used technique in HPC. A detailed overview of C/R is given in the next section.

2.4 Checkpoint-Restart Methods for Fault Tolerance

Checkpoint-based rollback recovery methods can be primarily categorized into three main groups: uncoordinated checkpointing, coordinated checkpointing, and communication-induced checkpointing [5].

In **uncoordinated checkpointing**, each process is in charge of its own checkpoints and is allowed to take them at its own convenience without coordinating with other processes. As a result, coordination overhead during the checkpointing process is reduced. It also reduces synchronization cost and energy consumption during the recovery process [30] since a single process crash does not result in all processes reverting back to the last checkpoint and recomputing. However, since there is no coordination, each process might have to keep multiple checkpoints which could be detrimental to performance. These protocols are also subject to domino effects and do not guarantee progress [5].

In **Coordinated checkpointing**, where all processes coordinate their checkpoints to form a consistent global state, is free of domino effects and has a simple recovery process. However, in the case of a large number of processes, these protocols may suffer from scalability issues due to the necessity of global coordination. Researchers have explored different approaches such as, non-blocking checkpoint coordination, checkpoint with synchronized clocks, minimal checkpoint coordination, to alleviate this problem.

**Communication-induced checkpointing** (CIC) protocols are a bridge between uncoordinated and coordinated C/R protocols. These protocols avoid
the domino effect without requiring all checkpoints to be coordinated. In CIC, processes take two kinds of checkpoints: local and forced. Local checkpoints can be taken independently as with uncoordinated protocols, while forced checkpoints must be taken to guarantee progress. Forced checkpoints are not coordinated checkpoints; rather, they are taken at the discretion of each individual process based on the communication pattern and checkpoint information received from other processes. However, these protocols are unpredictable due to their dependence on the communication pattern of the application [31].

Among these protocols, coordinated checkpointing techniques are the simplest and often work best for PGAS programming models [5, 32].

All these protocols can be implemented either at the system level or at the application level. They differ in the abstraction level at which the state of a process is saved. In system-level checkpointing, the entire state of the process, such as the contents of the program counter, registers, and memory are saved [33]. However, the amount of data saved during system-level checkpointing can be extremely large and the overhead may impact the application performance adversely. As a result, system-level checkpointing is not a popular choice for large-scale HPC platforms [8].

In application-level checkpointing, applications implement their own checkpointing code, usually by means of a third party checkpointing library [34–36]. We assume the use of such a library in the following. Applications decide which data to checkpoint and at what points in the program to do so. As a result, the amount of data to be checkpointed can be significantly reduced, since the application developer can decide exactly how much data needs to be saved, and moreover, system-level data is not saved. Application-level checkpointing also provides the added cushion of portability (not tied to a specific system). As a result, it is a popular choice for large-scale HPC systems. However, it places a huge burden on the developer to choose the right data and right place to checkpoint, including ensuring that the program state is consistent at such locations.

2.4.1 Application-level Checkpointing and Challenges

Although popular, application-level checkpointing comes with its own specific challenges, of which two important ones are maintaining program correctness, and controlling checkpointing overhead.

Program Correctness

Program correctness must not be compromised when using an application-level checkpointing scheme. To maintain the correctness of the program, check-
pointing calls must be inserted into the program at places in the code where
the program and its data are in a deterministic and consistent state. Finding
these points in an application can be non-trivial and the complexity may vary
across programming models since it is dependent on the semantics of features
of the language or library. Section 4.1 discusses this issue further and outlines
how it can be handled for OpenSHMEM.

Checkpointing Overhead

Checkpointing overhead is a major issue for any checkpointing scheme. Since
checkpointing involves saving or copying a large amount of data (program
state) to a persistent storage or memory, the overhead associated with this
process can be extremely large. To reduce this overhead different optimization
techniques are often used.

Three major optimizations utilized in practice are:

- **Minimizing the amount of data to be saved at a certain check-
  point**: In a C/R scheme, we only need to save the data that are nec-
  essary for restart, and, not all data are necessary at every point of the
  program. Hence, to minimize the amount of data to be saved at a certain
  checkpoint, different techniques that utilize compiler program analysis
  or operating system support are used in practice. Some of these tech-
  niques are, incremental checkpointing, region exclusion, and live variable
  analysis.

- **Choosing, and potentially adapting, the frequency of the check-
  points**: Having checkpoints too frequently may result in overhead explo-
  sion, while too few checkpoints may result in losing a lot of computation
  in case of a failure. Hence, choosing the optimal checkpointing interval
  is important. Checkpointing interval depends on both the structure of
  the application and the MTBF of the system it is running on. Finding
  optimal places for checkpointing is an active research area.

- **Optimizing the way the data is saved**: Optimizing the way the
  checkpointing data is saved in persistent storage or in memory may result
  in a significant overhead reduction. Several optimization techniques such
  as buffering, where intermediate storage such as burst buffer is used, and
  saving different data to different types of storage, exist in practice.
Chapter 3

Checking for Common Programming Mistakes Using OpenSHMEM Checker

Parallel programming is complex; writing an error-free, portable, and performant parallel program can be a Herculean task for application developers. Compilers play a vital role in detecting errors in application programs. However, many existing parallel programming models (e.g., MPI, OpenSHMEM) are library-based, often supporting multiple programming languages (e.g., C/C++, Fortran) but lacking specialized compiler support for error detection. As a result, the burden of error detection with respect to the program’s parallelization falls on the application developers and requires them to be extra vigilant to avoid bugs in their program. Reliance on runtime error detection can be difficult and may also result in undetected bugs in production code. Thus researchers use static and dynamic tools [37–39] to detect compile-time and runtime bugs when possible. Yet most existing tools are focused on MPI due to its popularity in the HPC community.

Other programming models with a growing user-base do not have the same tool support. OpenSHMEM is one such programming model. It is a Partitioned Global Address Space (PGAS) programming model which can be beneficial for computations with an irregular communication pattern (e.g., graph-based applications). Its performance advantages are due to a reliance on asynchronous, one-sided communications along with RMA (Remote Memory Access) support in recent hardware and its careful implementation. Yet the features (e.g., asynchronous, one-sided communication) that provide some of its key benefits lead to challenges with respect to writing an error-free OpenSHMEM-based parallel program.
To address this issue, we designed and developed a custom static checker, ‘OpenSHMEM Checker’ for OpenSHMEM programs based on LLVM. LLVM’s Clang static analyzer (CSA) provides a framework that can be used to build custom, domain-specific checks for C/C++ programs. It offers two different techniques that can be used to create them: 1) AST-based checks, which utilize the information provided by the Clang Abstract Syntax Tree (AST), and, 2) Path sensitive checks, which can leverage the Symbolic Execution engine, which explores all paths in a program via path-sensitive analysis. We developed the OpenSHMEM checker based on its path-sensitive analysis to provide a comprehensive and accurate error detection mechanism. The checker will analyze a program with respect to its usage of OpenSHMEM routines and provide compile-time errors, warnings, and suggestions related to them. We designed it in such a way that it can be adapted for other library-based PGAS programming models with minimal effort.

In order to perform this work, we first identified common programming mistakes in the OpenSHMEM programming model. Based on this investigation, we decided to implement checks for three different errors that are frequent in OpenSHMEM code and amenable to static detection. Those are:

- Check that inspects whether a call to an OpenSHMEM routine conforms to the specification
- Check for double free in OpenSHMEM-specific allocations (symmetric heap allocation/de-allocation)
- Experimental support for data race detection

### 3.1 Common Programming Mistakes in OpenSHMEM

In this section we provide an overview of the most common errors in OpenSHMEM through representative examples. We use some core OpenSHMEM features and routines (described in Table 3.1) to explain these programming errors.

#### 3.1.1 Violation of OpenSHMEM Semantics- Using Wrong Kind of Memory

One of the most common mistakes observed in OpenSHMEM programming is using the wrong kind of memory. OpenSHMEM has the concept of private
Table 3.1: Description of some basic OpenSHMEM concepts and routines

<table>
<thead>
<tr>
<th>Concept</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Symmetric memory</td>
<td>Potion of the memory that can be directly accessed by remote processes.</td>
</tr>
<tr>
<td>Private memory</td>
<td>Portion of the memory that is only accessible to the process that owns it. Generic concept of memory found in C/C++.</td>
</tr>
<tr>
<td>void shmem_init ()</td>
<td>Initializes the OpenSHMEM library. It is a collective operation that all PEs must call before any other OpenSHMEM routine may be called. Analogous to MPI_Init.</td>
</tr>
<tr>
<td>void shmem_finalize ()</td>
<td>Finalizes the OpenSHMEM library by releasing all resources used by the library. Analogous to MPI_Finalize.</td>
</tr>
<tr>
<td>void shmem_put (TYPE *dest, const TYPE *source, size_t nelems, int pe)</td>
<td>Writes ‘nelems’ amount of data of type ‘TYPE’ from ‘source’ buffer to remote process no. ‘pe’ s symmetric memory ‘dest’. The routines return after the data has been copied out of the source array on the local PE. The delivery of the data is not guaranteed upon return from this routine. Further synchronization routines must be called to guarantee the delivery of the data.</td>
</tr>
<tr>
<td>void shmem_get (TYPE *dest, const TYPE *source, size_t nelems, int pe)</td>
<td>Copies/reads ‘nelems’ amount of data of type ‘TYPE’ from the symmetric memory ‘source’ of the remote process no. ‘pe’ to local memory buffer ‘dest’. The routines return after the data has been delivered to the dest array on the local PE.</td>
</tr>
<tr>
<td>void *shmem_malloc (size_t size)</td>
<td>Allocates symmetric memory in the heap. Analogous to generic C routine ‘malloc’ except it allocates symmetric memory.</td>
</tr>
<tr>
<td>void shmem_free (void *ptr)</td>
<td>Releases/frees previously allocated symmetric memory. Analogous to generic C routine ‘free’ except it frees symmetric memory.</td>
</tr>
</tbody>
</table>
and symmetric memory and violating their usage results in undefined behavior. Listing 3.1 shows an example of such a case. In line 5, call to \texttt{shmem\_put} requires a symmetric memory buffer of a remote process as the first argument (described in Table 3.1); however, in this example, a private memory buffer ‘privMem’ is used instead. Using private memory ‘privMem’ where a symmetric memory is expected would result in undefined behavior (and worse yet, termination of the program depending on the implementation).

Unfortunately, a generic compiler (e.g., LLVM) would not be able to detect this error. Since C/C++ does not have the concept of private and symmetric memory, it can not distinguish between them. Therefore, it is up to the programmer to make sure the correct type of memory is used or risk having a buggy code with undefined behavior at runtime.

```c
... 
1 int * privMem = (int *) malloc(...); // private mem
2 // Error: Expects a symmetric memory
3 // Not detectable by generic compiler
4 shmem\_put(privMem, \&src, 1, pe);
5 ...
```

Listing 3.1: Using wrong type memory (symmetric vs. private)

### 3.1.2 Double-free of Heap Allocated Symmetric Memory

Trying to free an already freed memory is a common programming mistake in C/C++. Some compilers add extra checks (e.g., LLVM has a dedicated static checker for this) to catch this error. However, OpenSMEM has a specific type of double-free error that the generic double-free checkers can not catch. It stems from the OpenSHMEM specific symmetric memory allocation/deallocation routines. OpenSMEM provides routines (e.g., \texttt{shmem\_malloc, shmem\_free}) to allocate and free heap-allocated symmetric memory. However, these symmetric memory allocation/free routines are OpenSHMEM-specific; therefore, the double-free error arising for symmetric memory is not caught by generic double-free checkers. Extra support must be added to catch this specific type of double-free error in OpenSHMEM.

Listing 3.2 shows an example of double-free error arising in an OpenSHMEM program. In line 5, symmetric memory ‘symMem’ is allocated using \texttt{shmem\_malloc}. In line 8, ‘symMem’ is freed using \texttt{shmem\_free}. Therefore, when \texttt{shmem\_free} is called again in line 11 to free ‘symMem’, it results in a double-free error for ‘symMem’.

```c
... 
1 int *symMem = (int *) malloc(...); // allocation
2 // Error: Expects a symmetric memory
3 // Not detectable by generic compiler
4 shmem\_malloc(symMem, size);
5 shmem\_free(symMem);
6 shmem\_free(symMem);
7 ...
```

Listing 3.2: An example of double-free error arising in OpenSHMEM

27
int main(void)
{
    ...
    // Allocate symmetric memory 'symMem'
    int *symMem = (int *)shmem_malloc(...);
    ...
    // Free symmetric memory 'symMem'
    shmem_free(symMem);
    ...
    //Error: Try to free 'symMem' twice
    shmem_free(symMem);
    ...
}

Listing 3.2: Error: Double free of heap allocated symmetric memory

3.1.3 Unsafe/Unsynchronized Access to Program Data

OpenSHMEM provides the means for asynchronous, one-sided communication by allowing access to remote process’ (PE) symmetric memory without that process’ acknowledgment. Due to this asynchronous communication nature, OpenSHMEM has a relaxed memory consistency; in other words, the completion of a communication routine usually has to be confirmed by synchronization routines. Therefore, non-blocking accesses to a PE’s symmetric memory between two synchronization points may result in an inconsistent state of that PE’s symmetric memory. We use the code example shown in Listing 3.3 to explain this in the context of an OpenSHMEM program. Here we use `shmem_put` in line 7 to write to the remote PE no ‘1’ s symmetric memory ‘symMem’. However, returning from `shmem_put` does not guarantee the completion of actual write in the ‘symMem’ in remote process ‘PE 1’; it just confirms that the data has been copied out of source array ‘src’ on the local PE. The actual write in the remote PE may happen at any time in the future. Only a synchronization routine (e.g., `shmem_barrier_all`) after `shmem_put` can guarantee the completion. Therefore, when we use `shmem_get` in line 12 to access (read from) the remote process ‘PE 1’s symmetric memory ‘symMem’ that we wrote to previously using `shmem_put`, we can not guarantee whether ‘symMem’ would have the old value (before `shmem_put`), or new value (after `shmem_put`), since we don’t know if `shmem_put` was able to complete the write to ‘symMem’. Hence, we may have a data race condition. Therefore, read from ‘symMem’ is unsafe without a synchronization routine before it.
// Allocate symmetric memory
int *symMem = (int *)shmem_malloc(...);

// Write to symMem of PE no. ‘1’, Write is not
// guaranteed to complete without proper
// synchronization
shmem_put (symMem, src, 1, 1);

// Reading from the symMem of PE no. ‘1’
// Missing synchronization construct
// Error: NOT safe to access ‘symMem’ variable
// Possible race condition
shmem_get(src, symMem, 1, 1);

Listing 3.3: Unsafe/unsynchronized access to program data

3.1.4 Potential Performance Degradation Due to Excessive Synchronization

While synchronization constructs are one of the main building blocks of Open-
SHMEM and an absolute necessity for writing a correct OpenSHMEM program
as explained in the previous section, using too many and unnecessary synchron-
ization could heavily degrade the application performance. However, appli-
cation developers tend to overuse synchronization constructs in their programs
in their quest for correctness. Although this does not affect the correctness
of the program but may degrade performance heavily, hence finding cases of
excessive synchronization is extremely important.

Listing 3.4 shows 2 such cases of excessive synchronization. In line 7, there
is a call to shmem_barrier_all to synchronize all processes globally. However,
call to this barrier is unnecessary, since just before this, there is a call to a
symmetric memory allocation routine shmem_malloc at line 5 which has an
implicit global barrier after a successful completion of the routine. Hence, the
explicit call to the shmem_barrier_all can be omitted.

In line 15, we have another explicit barrier shmem_barrier_all. However,
since the last barrier in line 7 (or shmem_malloc in line 5 in a more opti-
mized version), no process (PE) accesses symmetric variables neither locally
nor remotely via OpenSHMEM communication routines, therefore the explicit
barrier in line 15 is not necessary and can be removed.
Listing 3.4: Warning: Potential performance degradation due to excessive synchronization

3.1.5 Deadlock Due to Missing Synchronization

OpenSHMEM has the concept of collective routines like other parallel programming models (e.g., MPI collectives) that are executed by all processes or a subset of processes executing the program. However, if a process that is supposed to participate in a collective doesn’t do so, it may lead to a deadlock scenario.

Consider the example in Listing 3.5 where due to the if statement on line 4, all even-numbered PEs will execute the ‘if’ part and the odd-numbered PEs will execute the ‘else’ part. In OpenSHMEM, the `shmem_barrier_all` is a collective routine that must be executed by all PEs at the same point in their execution, so to be semantically correct, there must be a `shmem_barrier_all` in the else block as well. In the above code, the even PEs will wait at barrier ‘b1’ for all odd PEs while the odd PEs would have reached ‘b2’ and be waiting for even PEs, thus resulting in a deadlock.
int main(void)
{
    ...  
    if(my_pe % 2 == 0) {  // For even PEs
        ...
        shmem_barrier_all();  // b1
    }
    else {  // For odd PEs
        ...  // Error: possible deadlock
        // No barrier in else path
    }
    ...
    shmem_barrier_all();  // b2
    ...  
}

Listing 3.5: Error: Deadlock due to missing synchronization

3.1.6 OpenSHMEM and MPI

The Message Passing Interface (MPI) is widely used to write parallel programs in the HPC community. Although, traditionally it utilizes a two-sided communication approach, it provides semantics for one-sided communication as well. Since, MPI is well known across the HPC community, in Table 3.2 we provide a comparison of the programming mistakes prevalent in MPI and OpenSHMEM.

3.2 Clang Static Analyzer

A compiler gathers substantial information during compilation. The Clang compiler, which is part of the LLVM compiler suite, allows external tools to utilize this information for various purposes (e.g., the ‘clang-tidy’ tool uses it for diagnosing and fixing typical programming errors).

The ‘Clang Static Analyzer (CSA)’ is one such tool that tries to find defects in a program by symbolically executing (imaginary execution, as if reading the source code and imagining what would happen if it was run) it without actually compiling it.

To achieve this, the analyzer uses algebraic symbols (with constraints and bounds) to denote unknown values (e.g., variable values that are dependent on input and only available at runtime). During the symbolic execution process [40], it uses these symbols (and their bounds) and the Clang CFG (Control Flow Graph) to build a graph of reachable states, called an Exploded Graph.
Table 3.2: Comparison of programming mistakes in MPI and OpenSHMEM

<table>
<thead>
<tr>
<th>Programming Error</th>
<th>MPI</th>
<th>OpenSHMEM</th>
</tr>
</thead>
<tbody>
<tr>
<td>Type mismatch</td>
<td>Exists (Data type vs. MPI type)</td>
<td>N/A - OpenSHMEM is type aware</td>
</tr>
<tr>
<td>Incorrect buffer referencing (e.g., referencing a ‘int’ buffer as character)</td>
<td>Exists</td>
<td>N/A - OpenSHMEM is type aware (few exceptions exist)</td>
</tr>
<tr>
<td>Invalid use of programming model specific objects</td>
<td>Invalid communicator, groups, and operations</td>
<td>Invalid context, and team</td>
</tr>
<tr>
<td>Invalid use of different types of memory</td>
<td>N/A, MPI does not introduce any new memory concept</td>
<td>Exists, wrong use of symmetric vs. private memory</td>
</tr>
<tr>
<td>Unmatched point to point call</td>
<td>Exists</td>
<td>N/A - OpenSHMEM uses one-sided communication semantics</td>
</tr>
<tr>
<td>Deadlock due to missing synchronization</td>
<td>Exists</td>
<td>Exists</td>
</tr>
<tr>
<td>Potential performance degradation due to excessive synchronization</td>
<td>Exists</td>
<td>Exists</td>
</tr>
<tr>
<td>Double non-blocking</td>
<td>Exists</td>
<td></td>
</tr>
<tr>
<td>Unmatched wait</td>
<td></td>
<td>Similar errors can happen in OpenSHMEM but in a different way. They fall in the category of ‘Unsafe/unsynchronized access to program data’</td>
</tr>
<tr>
<td>Missing wait</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

32
The Exploded Graph consists of all feasible paths through the CFG that were found by the analyzer. Each node in the graph represents pairs of program states and program points. The program state represents the abstract state of the program (e.g., mapping from source code locations to values, mapping from memory locations to symbolic values and their constraints) while the program location represents the exact symbolic execution location (e.g., before/after a statement) and the symbolic stack frames.

Symbolic execution of the program allows the analyzer to find deep, rare bugs that may be missed during the testing process. However, it is not a universal solution that finds all bugs; rather it provides a framework that can be engineered to find particular bugs. Specialized tools created by adapting the framework to seek specific kinds of bugs are called ‘Checkers’. While the core analyzer framework executes the program in a symbolic manner, the checkers subscribe to different events (via callbacks to different program points), check various assumptions on symbolic values at these events, and throw errors/warnings if those assumptions fail on a given program path. Checks that are based on symbolic execution along a program path are called ‘path-sensitive checks’. We used this framework to develop a custom checker to detect specific OpenSHMEM bugs.

The clang static analyzer provides the ability to write another type of check that exploits syntactic information only. Called ‘AST-based checks’, they are easy to write and add very little overhead to the compilation process. However, since they primarily use information available in Clang’s Abstract Syntax Tree (AST) their ability to find bugs is limited to finding illegal or undesirable code patterns. The OpenSHMEM bugs we are trying to detect cannot be identified by an inspection of the AST alone and hence we chose to base our checker on the Clang Static Analyzer and its symbolic execution engine.

### 3.3 Framework

We designed the OpenSHMEM Checker in such a way that it is easy to adapt for other PGAS programming models with minimal effort. It is comprised of 3 main components.

#### 3.3.1 Property Layer

OpenSHMEM introduces several PGAS library-specific behaviors/properties (e.g., whether a variable is symmetric or private) to the program data structures. These properties are often prerequisites and necessary for different
OpenSHMEM routines to work (e.g., OpenSHMEM communication routines can only access the symmetric memories of a remote process, trying to access private memory through these routines would result in undefined behavior). OpenSHMEM routines also introduce different property-induced states to program data structures (e.g., synchronized and unsynchronized state of symmetric variables). These properties, associated states, and their transitions dictate an OpenSHMEM program behavior; therefore, modeling these properties, states, and their transition correctly is vital for finding programming errors.

Let us revisit the example in Listing 3.3 to explain how these properties and their states transition through a program and dictate the program behavior. For simplicity, we will focus on only one property, ‘symmetric memory’, and its states (synchronized and unsynchronized). In line 3, we use the OpenSHMEM allocation routine `shmem_malloc` to allocate memory in the heap. This is no ordinary heap-allocated memory (e.g., using C `malloc`); this memory has one extra property; it is symmetric and can be accessed by other processes directly. So, `shmem_malloc` introduces the ‘symmetric memory’ property to the allocated variable. Declaring variables as ‘Static’ or ‘Global’ in an OpenSHMEM program also introduces the ‘symmetric memory’ property to those variables.

Symmetric memories have different types of states associated with them (e.g., synchronization states, allocation states). Each of these types can have multiple state-values (e.g., synchronization type has two states, synchro-
nized and unsynchronized). These states are used to model the OpenSHMEM programming model behavior, and transition to and from these states is dictated by OpenSHMEM routines.

Let us use Listing 3.3 again to see how these states transition in an OpenSHMEM program. In line 7, `shmem_put` is used to write data ‘src’ to PE 1’s symmetric memory ‘symMem’. For this routine to work correctly, it must fulfill the pre-condition that the first argument must be a symmetric memory. Once the pre-condition is fulfilled, and the routine is executed, it changes the synchronization state of the destination variable ‘symMem’, it makes ‘symMem’ at PE 1 ‘unsynchronized’. The reason for ‘symMem’ becoming ‘unsynchronized’ or unsafe is down to the communication nature of `shmem_put`. Since `shmem_put` returns right after the data has been copied out of the source array on the local PE, the delivery of the data is not guaranteed upon return from this routine. As a result, ‘symMem’ in PE 1 can have the old, new, or combinations of these values, rendering it to a non-deterministic or ‘unsynchronized’ state. Therefore without further synchronization routines to guarantee the delivery of the data, ‘symMem’ would be ‘unsynchronized’ or unsafe for further access (possible race condition). This transition of ‘symMem’ to ‘unsynchronized’ state is the result of `shmem_put`. Consequently, a global synchronization (`shmem_barrier_all`) or a local synchronization (`shmem_quiet`) routine at PE 1 guarantees that the new data has been written to ‘symMem’ and thus transitions to synchronized.

We implement these generic PGAS library-specific behaviors/properties and their associated states using the Clang Static Analyzer interface. We utilize CSA to do the bookkeeping for different properties and states associated with a certain data structure at different program points. We expose simple interfaces to add, remove, and query different properties and their states that can be used in the pre- and post-condition callbacks of the checker to detect errors. This property layer acts as an interface between the compiler and the library. Some of the interfaces implemented in the property layer are: `MarkAsSymmetric`, `IsMemRegionSymmetric`, `IsSynchronized`, `MarkAsSynchronized`, `MarkAsUnSynchronized`, `IsFree`, `IsArgNonNegative`.

### 3.3.2 Base Layer

We use the properties from the Property Layer to implement checks for generic PGAS operations such as blocking and non-blocking PUTs/GETs, memory allocation, and barriers. A specific PGAS library (e.g., OpenSHMEM) can use these checks out of the box, or provide its own implementations. Routines in the API can have pre-checks, post-checks, or both. In the pre-check phase, one can check for properties that must be fulfilled for the PGAS routine to
work correctly (e.g., the destination variable of PUT must be a symmetric variable). In the post-check phase, one can check for a specific property after the PGAS routine is executed or specify actions that have to happen (e.g., after a PUT the destination variable becomes unsynchronized). So, a generic implementation of PUT would look like this:

API: PUT (dest, src, PE, ...);
Pre-checks: IsSymmetric(dest); .......
Post-checks: MarkAsUnSynchronized(dest);

The primary purpose of the Base Layer is to make the analyzer extensible for other PGAS programming models. However, in this work we only focused on OpenSHMEM, other programming models were not considered.

### 3.3.3 Programming Model Layer

We implement programming model-specific operations in this layer. For example, in OpenSHMEM, we implement checks for routines like `shmem_put`, `shmem_get`, `shmem_malloc`, `shmem_barrier_all`. We can either use the default implementation provided by the Base layer for that specific type of routine, or we can provide our own implementation using the property layer APIs. A implementation of `shmem_put` would look like this:

API: shmem_put(T *dest, const T *source, 
size_t nelems, int pe);
Pre-checks: IsSymmetric(dest);
IsArgNonNegative(pe);
Post-checks: MarkAsUnSynchronized(dest);

### 3.3.4 Why Layered Design

Compiler development has a steep learning curve, which often deters library developers from writing checkers for their libraries. This layer-based design of the OpenSHMEM Checker tries to alleviate that problem by allowing the developers to write a checker for a specific PGAS programming model with minimal effort. It minimizes the interaction of the checker developer with the compiler, and allows the developer to use the default implementation out of the box from the base layer, or use properties implemented in the property layer to write their own library-specific checker. The layered design also allows modeling (adding support) of specific APIs or a group of APIs separately and independently; this enables development of incremental support for available APIs in a programming model without compromising the checkers ability to detect bugs in already supported APIs.
3.4 Integrated Checks

In this work, we implemented checks to detect the first 3 types of programming errors described in Section 3.1. These checks are supported on a large number of frequently used OpenSHMEM routines. Currently supported routines covers most of the memory management, remote memory access, collective operations in OpenSHMEM. In this section, we briefly describe how these checks are implemented.

3.4.1 Violation of OpenSHMEM Semantics- Using Wrong Kind of Memory

To implement this check, we track all the variables with the symmetric memory property (global and static variables, and memory managed by the OpenSHMEM allocation routines). Heap-allocated variables are marked using the property layer interface `MarkAsSymmetric` in the `POST_CALL` callbacks of OpenSHMEM allocation routines. They are marked immediately after they are allocated, irrespective of their future use. It is not easy to mark global and static variables from their declaration, so we wait until their first use in an OpenSHMEM routine and mark them as symmetric using the `MarkAsSymmetric` routine. We track these symmetric variables using a path-sensitive `ImmutableMap`.

In the `PRE_CALL` callback of each OpenSHMEM routine that takes symmetric arguments, we check that symmetric property using the property layer interface `IsSymmetric`. `IsSymmetric` searches the `ImmutableMap` that contains the list of all symmetric memories, and determines whether the variable is symmetric. If not, we raise a bug and generate a report for ‘wrong kind of memory.’

During the tracking process, we also track if a symmetric memory is heap-allocated or not, as some routines require this (e.g., `shm_realloc`).

3.4.2 Double-free

To implement double-free of the symmetric variables allocated via OpenSHMEM, we track the allocation and de-allocation of each symmetric variable in all program paths. Once they are allocated, we change their state to ‘allocated’ by using the property layer interface `RecordThisAllocation` in the ‘POST_CALL’ callback of the allocation routine. Once a variable is freed using `shm_free`, we change the state of that variable to ‘freed’ using `FreeThisAllocation` from the property layer. As part of the process, `FreeThisAllocation`...
checks if it has been freed; if so, it raises a ‘double-free’ error and reports a bug.

### 3.4.3 Unsynchronized Access to Program Data

This check provides experimental support for detecting unsynchronized access to program data. In its current form, it works best if the memory region that is affected by the unsynchronized access is constant-reducible (e.g., after constant propagation, it results in a constant value).

To implement this check, we used ‘ranges’ (start to end) to track unsynchronized memory regions. Every time a portion of the memory becomes unsynchronized due to an OpenSHMEM call (e.g., `shmem_put`), we use symbolic expressions (LLVM `SVal` objects) to store the start-index and number of elements read/written, from which we derive the start-index and end-index of that portion of the memory using the `SValBuilder`. We use this approach because most OpenSHMEM routines use an ‘offset’ and ‘numElements’ to read from and write to a symmetric memory. We also use this information to change a memory region’s status from unsynchronized/unsafe to synchronized.

Beyond this, we also track the PE in which the unsynchronized array region resides in, thus providing exact information as to which memory is unsynchronized and removing false-positives in the process.

This tracking process provides a very accurate range of unsynchronized memory regions at any point in time. Our tracking process is currently handicapped by the region arithmetic support and communication analysis in the Core ‘Clang Static Analyzer’, hence only supporting memory ranges that result in a constant value after constant propagation. Once region arithmetic becomes more mature in the ‘Clang Static Analyzer,’ our checker should be able to support non-constant ranges.

### 3.5 Evaluation

We used a custom test suite consisting of synthetic benchmarks dedicated for each error type and 3 Benchmark applications (Transpose, Matrix Multiplication (MM), and Mandelbrot Set) to evaluate the OpenSHMEM checkers performance. We introduced different programming bugs into working versions of these applications. Due to the lack of available benchmark applications in OpenSHMEM, we developed two (Transpose, and MM) of the ones used here.

- **Synthetic benchmarks** are developed to check each of the error types
- **Transpose** transposes a $N \times N$ matrix using a blocked approach.
• *MM* provides an OpenSHMEM implementation of Matrix Multiplication based on a variant of Cannon’s algorithm.

• *Mandelbrot Set* generates a greyscale image of the Mandelbrot set using the quadratic iteration function.

We evaluated the checker’s ability to find the 3 types of bugs it implements (Using wrong kind of memory, Double-free, and Unsynchronized access to program data) and the overhead. We used LLVM version 10.0 in a machine with Intel(R) Xeon(R) Gold 5115 CPU @ 2.40GHz with 192GB of memory running Fedora 32.

### 3.5.1 Case Study: Synthetic Benchmarks

We developed a synthetic benchmark that utilizes different OpenSHMEM routines supported by the checker and use it to test different programming errors.

**Wrong Kind of Memory**

We introduced ‘wrong kind of memory’ bugs in various OpenSHMEM routines, and the checker was able to detect all of them. Since the checker utilizes aliasing information, it was able to detect bugs that involved pointer following, which is illustrated in Listing 3.6.

```c
int *GlobalPtr; // Global (symmetric) variable
int main(int argc, char *argv[])
{
    ...
    int *privMem = (int *) malloc(N*sizeof(int));
    int *source = (int *) malloc(N*sizeof(int));
    // GlobalPtr is a symmetric variable
    // but it points to a private memory
    GlobalPtr = privMem;
    // Error: GlobalPtr is expected to be symmetric
    shmem_put(GlobalPtr,source,1,pe);
    ...
}
```

Listing 3.6: Wrong Kind of Memory Error

Here, *GlobalPtr* is a symmetric variable (line 1), however, it points to private memory (line 9) allocated by *malloc*. When it is used as a destination variable (first argument of *shmem_put* which OpenSHMEM expects to be symmetric memory), the checker catches and throws the appropriate error.
**Double-free**

We implemented double-free errors in several different ways and the checker was able to detect them successfully. These included the use of pointer indirection (aliasing) as well as an inter-procedural Double-free error. One scenario is illustrated in Listing 3.7.

```
1  ...  
2  int *ArgPtr = (int *) shmem_malloc(N*sizeof(int));  
3  ...  
4  int *indirectArgPtr = ArgPtr;  
5  ...  
6  shmem_free(indirectArgPtr);    //Free 1  
7  ...  
8  // Error: Double-free, indirectArgPtr and ArgPtr  
9  // points to the same memory which has already  
10  // been freed previously by freeing  
11  // indirectArgPtr  
12  shmem_free(ArgPtr);           //Free 2  
13  ...  
```

Listing 3.7: Double-free Error

**Unsynchronized Access**

The checker was not able to detect all of the unsynchronized access errors that we introduced into the synthetic benchmark. In order to avoid false-positives, we currently only detect unsynchronized accesses whose indexes and the PE numbers can be propagated to constants. Where these conditions do not hold, the checker fails. Comprehensive communication and dependence analyses would make improve detection of this error. Listing 3.8 shows both detected and undetected errors.

```
3.5.2 Case Study: Transpose, MM, Mandelbrot
```

We also evaluated the checker’s ability to find Wrong kind of memory, Double-free, and Unsynchronized access errors that were injected into the Transpose, MM, and Mandelbrot benchmarks.

**Wrong Kind of Memory**

The checker was able to detect 4 out of 5 wrong kind of memory errors. It missed one error in ‘Transpose’ simply because the OpenSHMEM routine in question, shmem_fcollect is not currently modeled (supported) by the
Table 3.3: Overhead analysis. Compile-time shown in secs.

<table>
<thead>
<tr>
<th>Benchmarks</th>
<th>No Static Analysis (s)</th>
<th>With Static Analysis (s)</th>
<th>With OpenSHMEM Checker (s)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Transpose</td>
<td>5.69</td>
<td>5.81</td>
<td>5.83</td>
</tr>
<tr>
<td>MM</td>
<td>4.78</td>
<td>5.43</td>
<td>5.50</td>
</tr>
<tr>
<td>Mandelbrot</td>
<td>0.68</td>
<td>28.95</td>
<td>33.91</td>
</tr>
</tbody>
</table>

Checker. That did not affect the ability of the checker to detect other errors due to its layered design. We plan to add support for all OpenSHMEM routines in the future.

```
int main(int argc, char *argv[]) {
...
  int *symMem = (int *) shmem_malloc(N*sizeof(int));
  int pe = shmem_my_pe();
  if (pe != 0) {
    shmem_put(&symMem[0], source, 1, 0);
    ...
    // Error: Unsynchronized Access to symMem
    shmem_get(source, &symMem[0], 1, 0);
  }
  shmem_put(&symMem[pe], source, 1, pe);
  ...
  // Unsynchronized Error not detected since value
  // of 'pe' is not a constant
  shmem_get(source, &symMem[pe], 1, pe);
  ...
}
```

Listing 3.8: Unsynchronized access Error

**Double-free**

The checker detected all 4 injected Double-free errors. One Double-free scenario in Transpose was inter-procedural and the checker handled it successfully.

**Unsynchronized Access**

We introduced an Unsynchronized access error into MM. The checker was unable to detect it because the array indexes and PE numbers used in the program were not propagated to constants. Since this shortcoming can hamper the checker’s ability to detect Unsynchronized access errors in real world applications, we will work on the necessary support analyses in future.
3.5.3 Overhead Analysis

OpenSHMEM Checker is a static analysis tool; hence it does not have any runtime overhead. Our checker adds a small to negligible compilation overhead on top of the Clang Static Analyzer Core if any other path-sensitive checker is used. The overhead is mostly related to the generation of the exploded graph and the number of paths that need to be explored. Therefore, the overhead is dependent on the number of conditionals in a program rather than the code size. Table 3.3 shows the overhead result for the 3 benchmark applications. We compare compilation time without static analysis, with static analysis (with the default checkers), and finally with the OpenSHMEM Checker enabled with other default checkers. Among the benchmarks, Mandelbrot has the most conditional statements, resulting in a large exploded graph and a significant increase in compile-time if the static analysis is used with a small overhead added on top of that using the OpenSHMEM Checker.

3.6 Concluding Remarks

In this chapter we discuss the necessity of error checking in OpenSHMEM programs and describe the development of a static checker based on LLVM’s Clang Static Analyzer Framework to help find bugs in OpenSHMEM programs. The checker has a layered design to make it easy to use, extend, and modify. We also made sure that no false-positives are reported by the checker, so that programmers can use the report with confidence. Early evaluation result shows great promise. However, much needs to be done to provide a comprehensive error detection mechanism for OpenSHMEM and other PGAS programming models.
Chapter 4

Checkpointing OpenSHMEM Programs Using Compiler Analysis

Determining how to build and deploy computer systems to enable recovery from failure, and what importance to place upon such capabilities (e.g., architectural redundancy in the network), is a perennial problem for computer systems designers. The greater the size and complexity of a system, the more likely it is that one or more components will fail during the execution of an application code. HPC (High Performance Computing) systems are inherently large and with the growing heterogeneity of processors and memory subsystems they are also increasingly complex. In the race to higher computational power, the number of components continues to increase and in consequence, the Mean Time Between Failure (MTBF) of these systems is decreasing [2, 3]. The smaller the MTBF, the less stable the system is. Especially for long-running applications, whose execution may take significantly longer than the MTBF of the platforms they are deployed on, it is vital that the interim results of a computation are not lost if a system fault occurs. This could otherwise lead to not just wasted computing resources, but potentially greater problems if the results are time-sensitive (e.g., weather forecasting). Thus strategies for resilience - in hardware, system software, and application software - are essential in HPC. Since an application program may run on a variety of platforms with differing MTBFs, application developers who need to ensure that failure does not compromise the timely production of results must pro-actively adopt a strategy to accomplish this. In other words, in most HPC contexts, the safe approach is to build resilience into the application code.

Unfortunately, most traditional HPC programming models (e.g., MPI,
OpenSHMEM, GASNet) lack the support for resilience that is built into their Data Analytics counterparts (e.g., Hadoop, Spark). Application developers must therefore rely on other means to achieve fault tolerance. Established techniques include but are not limited to Checkpoint-Restart (C/R), containment domains, and replication of computations. These techniques can be employed at the application level, at the system level or both. Among them, Checkpoint-Restart (C/R) is the most-used approach to implement fault tolerance in HPC applications due to its versatility, and relatively low overhead.

Checkpoint-Restart follows a simple strategy; one saves the state of the program as a so-called checkpoint at certain intervals during the execution. If the program terminates for any reason (hardware or software failure/faults) prior to completion, it restarts from the last checkpoint using the saved information instead of starting again from the beginning. Since the successful restart of an application relies on the data output during the check-pointing process, saving enough information to restart is very important.

However, care must be taken with respect to the amount of data saved during the check-pointing process. Since check-pointing results in a lot of data movement, oftentimes across nodes, the communication and I/O overhead it incurs can potentially result in severe performance degradation. Hence it is very important, but also extremely difficult, to determine the right amount of information to save, and a reasonable frequency at which to do so.

Moreover, there is another important challenge involved in implementing the Checkpoint-Restart process, that of finding suitable places to checkpoint. One has to make sure that the program is in a consistent and deterministic state with respect to both computation and data when the state of the program is saved at a checkpoint, otherwise it may result in a non-deterministic and potentially incorrect restart of the program. Finding these consistent and deterministic “safe points” can be tricky in an application that uses a parallel programming model such as MPI. However, it can be significantly tougher for an application that uses a programming model with asynchronous communications such as OpenSHMEM and other PGAS programming models (e.g., GASNet, UPC++).

Hence, the successful deployment of C/R depends on solving these consistency and data optimization challenges efficiently and effectively. However, to do so one has to have a good understanding of not just the programming model, and the checkpointing library being used, but also of the entire application, which can be very difficult for large scale scientific applications that have hundreds of thousands of lines of code. Earlier research has coupled an understanding of the semantics of a programming interface with compiler techniques [6–12] to ease the burden on application developers by identifying safe
points for checkpointing, suggesting which data to save and so on. However, prior work of this kind has primarily focused on providing support for MPI and not for other programming models such as OpenSHMEM. OpenSHMEM’s user base is growing, especially as a result of its benefits for computations with an irregular communication pattern (e.g., graph based applications). This is a result of its reliance on asynchronous, one-sided communications along with RMA (Remote Memory Access) support in recent hardware, and its careful implementation. Hence, support for developing resilient OpenSHMEM applications is needed. Yet the features that provide some of its key benefits lead to challenges when approaches created for MPI are applied to it.

In this chapter:

- We analyze the OpenSHMEM programming model from a resilience perspective and define “safe points” for inserting checkpointing library calls into OpenSHMEM applications.
- We use a compiler’s data analysis to determine what data to checkpoint.
- We describe a tool that we have created to support the insertion of checkpoints into OpenSHMEM code. It utilizes a safe point analysis and data analysis to provide check-pointing suggestions (where to checkpoint, what data to checkpoint) to the OpenSHMEM application developer. The tool is based on an open-source compiler framework, LLVM [16], which is becoming very popular in both academia and industry.

## 4.1 OpenSHMEM and Safe points

### 4.1.1 Global Consistency and Safe Points

When checkpointing an OpenSHMEM application (or any other parallel application for that matter), one has to make sure that the checkpoint is taken at a point where the application’s computation, communication, and memory are in a consistent and deterministic state in order to ensure deterministic recovery. We call these points “safe points”. Due to the relaxed memory consistency nature of OpenSHMEM’s memory model, non-blocking accesses to a PE’s symmetric memory between two synchronization points may result in an inconsistent state of the PE’s symmetric memory.

We use the code example shown in Figure 4.1 to explain this in the context of an OpenSHMEM program. The code snippet shows a part of a halo exchange, where PEs share the updated values of elements at the edges of
their subregions with neighboring PEs. For simplicity we show the interaction between two neighboring processes only. Here, PE0 (the host PE) uses `shmem_put` to write/copy the first two elements of its `host_halo` array to the first two elements of PE1’s `neighbor_halo` array; `neighbor_halo` has been allocated in symmetric memory so that PE0 can access it directly via OpenSHMEM routines. However, between the call of `shmem_put` and that of `shmem_barrier_all`, the symmetric array `neighbor_halo` of PE1 is in an inconsistent state since there are 3 possible states of `neighbor_halo` (shown by 3 different scenarios in Figure 4.1). These states are no update (Scenario 1), partial update (Scenario 2) and complete update (Scenario 1). So if a checkpoint is taken at this place in the code, the checkpointed data (`neighbor_halo`) would be in an unknown state, resulting in non-deterministic behavior during recovery. However, OpenSHMEM’s synchronization routines (e.g., `shmem_barrier_all`, `shmem_quiet`) do confirm the completion of communication routines (e.g., `shmem_put`) and in OpenSHMEM guarantee consistency of some or all communications (no in-flight messages). The `shmem_barrier_all` routine used in this example is a global synchronization routine and it provides global consistency guarantees across all PEs. Hence, the point in the code immediately after `shmem_barrier_all` is a safe-point and a checkpoint taken there would guarantee deterministic recovery.

Figure 4.1: An example showing a safe point in an OpenSHMEM program
Table 4.1: Safe points in OpenSHMEM API. It’s safe to checkpoint right after a successful call to these functions.

<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
<th>Pre-requisite for Global Sync</th>
<th>Suggestion</th>
</tr>
</thead>
<tbody>
<tr>
<td>shmem_barrier_all</td>
<td>Global barrier</td>
<td>None</td>
<td>None</td>
</tr>
<tr>
<td>shmem_malloc</td>
<td>Symmetric memory allocation collective operations that require participation by all PEs in the world team. Global synchronization on exit if the calls are successful.</td>
<td>Call to these functions have to be successful</td>
<td>Add a check to make sure the allocation is successful</td>
</tr>
<tr>
<td>shmem_malloc_with_hints</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>shmem_calloc</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>shmem_align</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>shmem_realloc</td>
<td>Symmetric memory reallocation routine (collective). May use global synchronization on both entry and exit, depending on whether an existing allocation is modified and whether new memory is allocated, respectively.</td>
<td>Call this function has to be successful</td>
<td>Add a check to make sure the allocation is successful</td>
</tr>
<tr>
<td>shmem_free</td>
<td>Symmetric memory deallocation routine (collective). Global synchronization at the entry.</td>
<td>None</td>
<td>None</td>
</tr>
<tr>
<td>shmem_init</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>shmem_init_thread</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>shmem_finalize</td>
<td>OpensHMEM library initialization and finalization routines (collective).</td>
<td>None</td>
<td>None</td>
</tr>
<tr>
<td>shmem_quiet /</td>
<td>Combination of shmem_quiet/shmem_ctx_quiet routine called by all PEs, followed by a shmem_sync_all collective or shmem_team_sync (on the world team) results in global synchronization.</td>
<td>None</td>
<td>None</td>
</tr>
<tr>
<td>shmem_cxt_quiet +</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>shmem_sync_all /</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>shmem_team_sync(world)</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
4.1.2 Why Global Synchronizations are the Best Safe Points for OpenSHMEM

OpenSHMEM provides different types of synchronization or memory consistency routines: local consistency routines that only affect the operations initiated by the calling PE (shmem_quiet), and global consistency routines that affect all operations initiated by all PEs (shmem_barrier_all). In an ideal scenario, it is possible to reason about the local consistency routines across PEs and find safe-points. In a message-passing model such as MPI, each process knows what data it is sending and what it is receiving (send-receive rendezvous pair). Hence, by performing a communication analysis for a certain code-block, it is possible to reason about whether the process’s communication and data are in a consistent state or not. In contrast, PGAS models like OpenSHMEM are not based on send-receive communication pairs. An active PE in OpenSHMEM initiates communication to write to or retrieve data from the symmetric memory of another PE. Since the latter is not involved in the communication, it may not be aware that this is taking place. Hence it cannot locally reason about its own data consistency, since it does not know if any communication initiated by other processes is in flight which would leave the process’s symmetric memory in an inconsistent state. It is similarly hard for a compiler or tool to reason about the consistency of an application program at any given time.

Hence, the easiest and safest way to identify places in the code where data is in a consistent state in OpenSHMEM is to exploit the global consistency achieved immediately upon completion of global synchronization, where the network is quiet (no in-flight messages).

Table 4.1 shows the OpenSHMEM routines that provide global synchronization and can be used as safe points. However, OpenSHMEM has the concept of user-defined communication context or communication channel. A communication context is a container for communication operations. Each context provides an environment where the operations performed on that context are ordered and completed independently of other operations performed by the application. Each OpenSHMEM program comes with a default communication channel called “default context”.

The safe points described in Table 4.1 are for default context. Applications with user-defined contexts must additionally ensure that those contexts are quiet (all the update/communication to symmetric memory is completed) before using those routines as safe points. Calls to shmem_ctx_quiet by all PEs would ensure the completion of outstanding communication to symmetric objects (would quiet that specific context).
4.2 Choosing The Right Data to Checkpoint

In the previous section, we discussed the importance of finding safe points in order to ensure that, in the presence of a fault, the data saved in a checkpoint will enable a correct restart. We also identified OpenSHMEM routines that can be used for that purpose.

While finding the safe points, and ensuring that we insert checkpointing calls only at those points, solves the correctness issue, we still need to optimize the amount of data to be saved during the checkpointing process in order to avoid unnecessary overhead. If insufficient attention is paid to reducing the amount of data saved during a checkpoint, the overheads may become insurmountable. In order to alleviate this problem, researchers have investigated memory exclusion techniques both at compiler level [41] and at runtime level [42]. In memory exclusion, regions of a process’s memory are excluded from a checkpoint because they are either read-only, meaning their values have not changed since the previous checkpoint, or dead, meaning their values are not necessary for the successful completion of the program [43].

We apply and adapt the results of a standard compiler analysis called “Live Variable Analysis” to identify variables (scalars and arrays) that we do not need to checkpoint at a specific position in the program code. Live Variable Analysis is traditionally applied to scalar variables in order to optimize register usage, where it helps remove variables from registers when they are no longer needed. Live variable (or liveness) analysis is a data flow analysis that essentially determines whether the current value of a variable may be used in the future. A variable \( x \) is live at a program point/statement \( s \) if some computational path from \( s \) to the end of the program (or function) contains a use of \( x \) which is not preceded by a new definition. In other words, it is live if the current value of variable \( x \) may be used at a later point in the code. The set of variables \( \text{LIVE}_{\text{in}}[s] \) that are live at a given statement \( s \) can be calculated using the following formula:

\[
\text{LIVE}_{\text{in}}[s] = (\text{LIVE}_{\text{out}}[s] - \text{DEF}[s]) \cup \text{USE}[s] \tag{4.1}
\]

where \( \text{LIVE}_{\text{out}}[s] \) is the set of variables live after the statement \( s \), \( \text{USE}[s] \) represents the set of variables used by \( s \), and \( \text{DEF}[s] \) represents the set of variables defined by \( s \).

If a variable is not live at a certain checkpoint, then the value that it currently has is not used in any subsequent code. In other words, if the variable occurs at a later point in the code, then any uses of the variable will be preceded by new assignments to it. Therefore, we do not have to save it at that specific checkpoint. This can potentially help a user (or tool) identify and delete some
unnecessary saves during the checkpointing process. Figure 4.2 shows a simple OpenSHMEM code segment that utilizes live variable analysis information for checkpointing purposes. The program has 8 safe points, `shmem_init` at line 3, `shmem_malloc` at line 4 and 5, `shmem_barrier_all` at line 13 and 20, `shmem_free` at line 27 and 28, and finally `shmem_finalize` at line 29. Here, we only consider `shmem_barrier_all` for checkpointing. At line 13, both A and B are live, hence, we have to save both variables for a successful restart (shown in line 14). At line 20, we see that only B is live, therefore we only need to save B instead of both A and B (shown in line 21). At this checkpoint, live variable analysis helped us reduce the checkpoint data to half.

4.3 Compiler Tool: Putting it Together

We developed a tool that incorporates the OpenSHMEM-specific safe point analysis and data optimization discussed in Section 4.1 and 4.2 respectively. The purpose of this tool is to provide a starting point for application-level checkpointing. It identifies the safe points in an OpenSHMEM program and provides suggestions as to what data to checkpoint at each of them in case the user decides to use a given safe point as a checkpoint.

We developed this tool based on the LLVM (Low Level Virtual Machine) \[16\] compiler suite. LLVM is an open-source compilation framework that uses an intermediate representation in Static Single Assignment (SSA) form. It has front ends to support multiple languages that are transformed into LLVM IR for analysis and optimization. It has been widely adopted in both academia and industry which is one of the motivations for our choice. One other interesting aspect of LLVM is its “Pass Framework” where most of the interesting parts of the compiler exist. Passes are used to perform analysis, transformations, and optimizations at the IR level. It also allows for extension or addition of a new analysis or optimization in a structured manner. We implemented our tool as LLVM passes. It has three main parts: the safe point identification phase, data optimization phase, and user-feedback phase. Figure 4.3 shows the architectural diagram of the tool.

4.3.1 Safe Point Identification Phase

In this phase, we analyze the program to determine places where the data and computation are in a consistent and deterministic state (safe points). We utilize the analysis information described in Section 4.1. Since the safe points are dependent on calls to a certain function or a series of functions that
int main()
{
  shmem_init(); //safe point (SP)
  int *A = (int *)shmem_malloc(N * sizeof(int)); //SP
  int *B = (int *)shmem_malloc(N * sizeof(int)); //SP
  ...

  for(int i = 0; i<N ; i++)
  {
    A[i] = init_A(i);
    B[i] = init_B(i);
  }
  shmem_barrier_all(); //safe point; LIVE = {A, B}
  CHECKPOINT(A, B);

  for(int i = 0; i<N ; i++)
  {
    ... = A[i]; //uses A
    ...
  }
  shmem_barrier_all(); //safe point; LIVE = {B}
  CHECKPOINT(B); //A is not live anymore

  for(int i = 0; i<N ; i++)
  {
    ... = B[i]; //uses B
    ...
  }
  shmem_free(A); //safe point
  shmem_free(B); //safe point
  shmem_finalize(); //safe point
  return 0;
}

Figure 4.2: A skeleton OpenSHMEM program showing the impact of Live variable analysis for checkpointing

Figure 4.3: Architectural Diagram of the tool
result in global synchronization, we track these function calls and retain this information for further data optimization analysis discussed next.

### 4.3.2 Data Optimization Phase

In this phase, we perform live variable analysis on the program. Based on the results, we determine which variables are live at a given safe point. During the analysis process, we treat arrays as a single entity i.e., if any element of an array is live at a certain point, the whole array is live. However, LLVM IR is in SSA form and every LLVM IR variable has exactly one definition. Therefore, the access to an element of an array may occur via multiple indirections and just from the instruction that actually accesses the element, we may not know which array it belongs to. To handle this problem, we track each of the SSA definitions back to its source.

We adapt live variable analysis to allow it to correctly deal with OpenSHMEM routines by furnishing definition and use information on the variables referenced in associated calls. In other words, we add partial awareness of the semantics of OpenSHMEM calls to the LLVM compiler. For example, `shmem_free` is used to free the symmetric memory allocated by OpenSHMEM-specific allocation routines. Although `shmem_free` uses a pointer (pass by reference) argument passed to it to select which memory to free, it does not actually use the value of the elements allocated on those memories, nor would that value be used in the future (memory is already freed after this call). Therefore this is not a use of any variable and variables referenced in a `shmem_free` call should not, on the basis of this reference alone, be considered to be live. Without this information, the compiler would potentially be forced to extend the live range of the corresponding values to this point in the code.

### 4.3.3 User-feedback Phase

In this phase, we notify the user of the decisions made by the framework based on the data optimization phase. We provide information on where the safe points are and what data needs to be saved if a certain safe point is used as checkpoint. However, there is an additional problem. The analysis is performed by the tool at the IR level of the program where the source code level information (e.g., variable name, line number) is lost. In order to provide useful feedback to the user, we need to translate the IR-level information back to the source-level code, since the application developer using this will only have access to the latter. To solve this issue, we utilize the “LLVM source level debugging information” to:
• Find source-level positions (e.g., line numbers) for potential checkpoints
• Find source-level variable names and their position from IR level temporary variables

For this reason, this tool has to be run with the debug (-g) flag switched on and with no optimization enabled (optimization may result in source level information loss).

4.4 Results and Analysis

In this section, we evaluate the safe point analysis and data optimization carried out by the compiler tool. We expect that the suggestions provided by this tool will be used by an application developer to create a fault-tolerant OpenSHMEM program by manually using a checkpointing library of their choice. We assume that the checkpointing library allows users to choose which data to save at each checkpoint i.e., has the ability to register or deregister variables to be checkpointed at each checkpoint. However, application-level checkpointing library support for OpenSHMEM is very scarce, and to the best of our knowledge, none exist at this point with the above-mentioned capability. We are in the process of adapting an existing library to serve this purpose.

Since there is no such library available for experimentation, for this work, we present the calculated result (instead of the execution result) that assumes such a library will be utilized. The goal of this work is to find safe points for checkpointing in an OpenSHMEM program and to optimize the data sets. Evaluation of the former does not require program execution, but rather a manual analysis to ascertain whether the points reported by the tool are indeed safe points and vice versa. With respect to the data optimization, the total amount of data used by a program and the size of the optimized data set computed via live variable analysis can be calculated accurately from the source code if the input parameters are known. Therefore, the results presented here should closely resemble the actual execution run using a checkpointing library.

We use 3 Benchmark applications, Transpose, Matrix Multiplication (MM), and Mandelbrot Set to evaluate this work. Due to the lack of benchmark applications written in OpenSHMEM, we developed two of the benchmarks (Transpose, and MM) used here.

We evaluate this tool based on two criteria: safe point identification and data optimization. For safe point identification, we validate the safe points identified by the tool via a manual investigation of the program by an expert.
For data optimization, we compare the results of our optimization strategy against Hao et al. [44], which to the best of our knowledge, is the only prior work on OpenSHMEM application-level fault tolerance. Hao uses the “default” approach of saving all the available data during checkpointing. For a certain checkpoint, he saves all the data that are within the scope and potentially necessary for restart in any other checkpoints of the program. We use the term “All data” to refer to that result while using the term “lva-optimized” (Live variable analysis-optimized) to refer to the data optimization using our approach.

4.4.1 Transpose

The transposition is a very common communication pattern, which is commonly found in linear algebra calculations and FFT (Fast Fourier Transform) computations. This benchmark transposes a $N \times N$ matrix using a blocked approach. Each PE gets a sub-matrix containing a subset of rows, performs a local transpose, and then combines the results using an OpenSHMEM collective communication routine (`shmem_fcollect`). It has two functions, the `main` function, and the `transpose2D` function that does the transpose operation on the sub-matrix. We use the matrix size of $2048 \times 2048$, and 16 PEs for the experimental calculation.

Our tool is able to identify all 13 safe points that are present in the program; 6 of them are in `transpose2D`, and 7 of them are in `main`.

We compare the result of data optimization using our tool (lva-optimization) with Hao (“All data”) in Figure 4.4. We show the result for all the safe points, although some of them may not be suitable for checkpointing (e.g., `shmem_init` and `shmem_finalize` are often the first and last parts of an OpenSHMEM program; hence a checkpoint may not be necessary). We leave the choice of choosing checkpoints from these safe points up to the user.

In Figure 4.4, the X-axis shows the safe points in the program along with their source line number; the Y-axis shows the checkpoint data size at a specific safe point. We use normalized data sizes, normalized by “All data” (all the available data at that safe point). The blue bars (left) represent the amount of data to be checkpointed at a certain safe point using the “All data” approach, while the red bars (right) represent the amount of data to be checkpointed using our approach. We observe that our approach is able to optimize the amount of data to be checkpointed significantly (checkpoints less than 1% of all available data) compared to the “All data” approach for some safe points, while requiring the same or close to the same data for others. For the cases where the amount of data to be checkpointed using our approach is negligible compared to the “All data” approach, the red bar (lva-optimized) is so small
that it is not visible in the graph (e.g., \texttt{shmem\_finalize} in the \texttt{main} function).

In the \texttt{main} function, we see that our approach does significantly better for the safe points that are at the latter part of the function. Often the memory (both symmetric and private) allocation and deallocation occur at the beginning and the end of a function, respectively, for better code readability. As a result, the allocated memory may be freed much later than its last use. This is potentially a problem for the “All data” checkpointing approach. Although the data is not used anymore, it is still part of the program memory; hence in the “All data” approach, this unused data must also be checkpointed. Our Live variable analysis approach can resolve this.

In the \texttt{transpose2D}, we see a slightly different behavior for the safe points in the latter part of the function. Although we see improvement using our approach, it is not as significant as we observed in the \texttt{main} function. This lack of improvement is because we do live variable analysis at the intra-procedural level; hence, we assume that any argument passed to a function by reference is “live” at the end of the function (the calling function may use updated values). In \texttt{transpose2D}, one of the arguments is a large array that is passed by reference. Since this variable can not be eliminated in our current approach, we see smaller gains than with the \texttt{main} function.

![Figure 4.4: Checkpoint data sizes at all the safe points in Transpose. Smaller is better (some bars for lva-optimized are not visible, because the value for lva-optimized is less than 1% of “All data”).](image)
4.4.2 Matrix Multiplication (MM)

This benchmark provides an OpenSHMEM implementation of Matrix Multiplication based on a variant of Cannon’s algorithm. We use the matrix size of $2048 \times 2048$, and 16 PEs for the experimental calculation.

Figure 4.6 shows the comparison of our approach (lva-optimized) with the “All data” approach. The benchmark has 13 safe points, all of them in the main function. Most of the safe points are initialization/finalization and allocation/de-allocation routines while 3 of them are global barriers (shmem-barrier-all). Our tool is able to identify all of the safe points correctly.

In terms of data optimization, we observe similar results to the main function of Transpose; and for the same reason. Our approach is able to reduce the checkpoint data size significantly for the safe points in the latter part of the main function.

Figure 4.5: Checkpoint data sizes at all the safe points in MM. Smaller is better (some bars for lva-optimized are not visible, because the value for lva-optimized is less than 1% compared to “All data”).

4.4.3 Mandelbrot Set

This benchmark generates a greyscale image of the Mandelbrot set using the quadratic iteration function. The image is partitioned evenly across the PEs, and the computation is embarrassingly parallel. We use a $3200 \times 3200$ image of the Mandelbrot set, and 16 PEs for the experimental calculation.

The application has a total of 7 safe points, and our tool is able to identify all of them. 5 of the safe points are in the draw_mandelbrot function
which does the actual computation, and 2 are in the main function which initializes/finalizes the OpensHMEM library and calls the draw_mandelbrot function. Therefore, the important safe points are in draw_mandelbrot, where all the data allocation, de-allocation, and computation occur. For data optimization, we see the familiar behavior already observed in Transpose and MM; we achieve significant savings using our approach at the latter safe points of the draw_mandelbrot function.

Figure 4.6: Checkpoint data sizes at all the safe points in Mandelbrot. Smaller is better (some bars for lva-optimized are not visible, because the value for lva-optimized is less than 1% of “All data”).

4.5 Concluding Remarks

In this chapter, we analyzed the OpenSHMEM programming model from a resilience perspective and defined “safe points” for inserting checkpointing library calls into OpenSHMEM applications. Moreover, we provide the set of data to be saved at each potential checkpoint. We adapted live variable analysis for OpensHMEM to optimize the amount of data to be checkpointed at a certain safe point. We show that this optimization can sometimes result in a large improvement in the amount of data to be checkpointed (in some cases, less than “1%” of the available data is checkpointed).
Chapter 5

OpenMP on a Power Constrained Node: A Case Study of OpenMP Performance Variability Across Different Execution Environments

Performance portability has become a growing concern in the HPC community due to the differences in system environments (both hardware and software) in different HPC systems. Often, these performance portability issues start at the intra-node level due to the node-level changes across systems as shown in Table 5.1. The table shows the node-level differences in Top 5 supercomputers (according to top500 list as of June, 2021 [45]) in the world. These systems differ in processor technologies, instruction sets, processor frequencies, the number of cores per node, network interconnects, power consumptions, and other aspects. On top of these differences that are prevalent across these systems, the execution environment of a single system may also change; which may also happen during the execution of an application. One example of such a scenario is a power-constrained system, where, to meet a given power budget, the system constrains the power consumption of different components (e.g., constraining processor, memory power).

Power constrained systems, which can be designed by overprovisioning the hardware nodes [46] have gained popularity in recent times due to their ability to manage the total power consumption of a system. Recent advances in processor and memory hardware designs have allowed the user to control the power consumption of CPU, GPU, and memory through software. This ability
Table 5.1: Top 5 supercomputers as of June 2021 (Top 500 list)

<table>
<thead>
<tr>
<th>Rank</th>
<th>Name</th>
<th>Power (kW)</th>
<th>Processor Cores Frequency</th>
<th>Processor Technology</th>
<th>Processor Speed (MHz)</th>
<th>Accelerator/Co-Processor</th>
<th>Cores per Socket</th>
<th>Interconnect</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Supercomputer Fugaku</td>
<td>29,899.23</td>
<td>ARM A64FX 48C 2.2GHz</td>
<td>Fujitsu ARM</td>
<td>2,200</td>
<td>None</td>
<td>48</td>
<td>Tofu interconnect D</td>
</tr>
<tr>
<td>2</td>
<td>Summit</td>
<td>10,096.00</td>
<td>IBM POWER9 22C 3.07GHz</td>
<td>Power</td>
<td>3,070</td>
<td>NVIDIA Volta 22</td>
<td>22</td>
<td>Mellanox EDR Infiniband</td>
</tr>
<tr>
<td>3</td>
<td>Sierra</td>
<td>7,438.28</td>
<td>IBM POWER9 22C 3.1GHz</td>
<td>Power</td>
<td>3,100</td>
<td>NVIDIA Volta 22</td>
<td>22</td>
<td>Mellanox EDR Infiniband</td>
</tr>
<tr>
<td>4</td>
<td>Sunway TaihuLight</td>
<td>15,371.00</td>
<td>Sunway SW26010 26C 1.45GHz</td>
<td>ShenWei</td>
<td>1.450</td>
<td>None</td>
<td>260</td>
<td>Sunway</td>
</tr>
<tr>
<td>5</td>
<td>Perlmutter</td>
<td>2,528.00</td>
<td>AMD EPYC 7741 64C 2.45GHz</td>
<td>AMD Zen-3 (Milan)</td>
<td>2,430</td>
<td>NVIDIA A100 64</td>
<td></td>
<td>Slingshot-10</td>
</tr>
</tbody>
</table>

To control the power consumption of different sub-systems is used to design large-scale power-constrained systems to meet the growing power budget since power is becoming the limiting factor for a large-sale HPC system design.

Power-constrained systems can also play an important role in performance variance study. Since changing the power consumption of the processor and memory changes their behavior significantly (e.g., change in operating frequency, clock gating, power gating), it essentially changes the execution environment. As a result, a power-constrained system can be used as a viable alternative to using multiple machines with different execution environments for performance variance study.

5.1 A Motivating Example

The OpenMP programming model is an integral part of many important HPC legacy codes in the form of hybrid programming models (e.g. MPI + OpenMP). Therefore, tuning an OpenMP code to get a better per node performance for a given power budget is an important research problem. In this section, we motivate the reader about this problem by studying the performance behavior of an OpenMP code under various power levels.

We took a loop based OpenMP parallel region from the SP benchmark application of NPB. The parallel region belongs to the compute_rhs function, and has 11 different parallel loops, i.e., #pragma omp for directives. We ran it at different power levels or power caps\(^1\) using a different number of threads, scheduling policies, and chunk sizes (148 different configurations).

**Definition 1 (Optimal Configuration).** An OpenMP runtime configuration

\(^1\)We use the two words synonymously in this work.
Figure 5.1: Execution time comparison of different OpenMP runtime configurations at different power levels for the compute rhs parallel region of SP.

within the configuration search space that gives the best execution time at a certain power level.

Definition 2 (Default Strategy). An OpenMP runtime configuration used by an OpenMP runtime library by default.

Figure 5.1 shows the comparison of execution time using the optimal configuration (see Definition 1) and the default configuration (see Definition 2) at different power levels. We used the OpenUH OpenMP runtime library for this experimentation. For the default configuration, the OpenUH runtime library uses the maximum number of available threads, static scheduling policy, and a chunk size that is calculated dynamically by dividing the total number of loop iterations by the number of threads. Figure 5.1 presents two key findings. First, the optimal configuration is different from the default configuration at all power levels. The optimal configuration improves the execution time of the parallel region up to 19% compared to the default configuration at the same power level. Second, the optimal configuration at a lower power level yields better execution time performance than the default configuration at the maximum power level or at Thermal Design Power (TDP). For example, the optimal configuration at 70W power cap improves execution time by 15% as compared to the default configuration at TDP (115W). These observations are useful for the improvement of OpenMP performance in a power constrained system.

Different OpenMP runtime libraries may have different default strategies.
Therefore, a detailed study of the OpenMP runtime parameters under different power levels for a power constrained system is necessary.

5.2 Configuring the experimental search space

Here, we briefly describe our experimental parameters and how we performed our experimentation. Our experimentation mainly revolved around two main concepts, OpenMP runtime parameter configuration & Power capping/clamping level. First, we describe these two concepts and then we discuss how we worked with these concepts.

5.2.1 OpenMP Runtime Configuration

OpenMP is the de-facto parallel programming model for shared memory programming. In OpenMP, a certain region or block of a code is parallelized to run on multiple threads. These regions are called parallel regions. The main parameter of a parallel region is the number of threads that’s going to execute it. Now inside the parallel region, work can be divided among different threads in many different ways. These are called work sharing constructs. Although in OpenMP there are different work sharing constructs such as tasks for irregular parallelism (e.g. recursive algorithms), sections, OpenMP’s main impact comes from its ability to efficiently parallelize loops. OpenMP parallelizes a loop by effectively dividing the loop iterations among different threads. In this way, each thread executes only a portion of the total iterations. In this work, we mainly considered OpenMP’s loop level parallelism. Along with the number of threads, there are two more parameters that are used to tweak how a specific loop is going to be parallelized. They are Scheduling Policy & Chunk Size. These two parameters are used together. Combined, they describe how the loop iterations are divided among the threads. Scheduling policy describes a way to divide the work among the threads, while chunk size controls how many of the iterations are assigned to each thread at once. There are 5 existing scheduling policies as of the current OpenMP 4.5 standard [47]. Table 5.2 provides a brief overview of different types of scheduling policies. Chunk sizes are non-negative integer values which are less than or equal to the total number of loop iterations.

In our experimentation, we only STATIC, DYNAMIC, GUIDED and RUN-TIME. We did not use AUTO, because AUTO delegates the scheduling decision to the compiler and/or runtime system. As a result, there is no way for
Table 5.2: OpenMP Loop Scheduling Policy Overview

<table>
<thead>
<tr>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>STATIC</td>
<td>Loop iterations are divided into pieces of size chunk and then statically assigned to threads. If a chunk is not specified, the iterations are evenly (if possible) divided contiguously among the threads.</td>
</tr>
<tr>
<td>DYNAMIC</td>
<td>Loop iterations are divided into pieces of size chunk, and dynamically scheduled among the threads; when a thread finishes one chunk, it is dynamically assigned another. The default chunk size is 1.</td>
</tr>
<tr>
<td>GUIDED</td>
<td>Iterations are dynamically assigned to threads in blocks as threads request them until no blocks remain to be assigned. Similar to DYNAMIC except that the block size decreases each time a parcel of work is given to a thread. The chunk parameter defines the minimum block size. The default chunk size is 1.</td>
</tr>
<tr>
<td>RUNTIME</td>
<td>The scheduling decision is deferred until runtime by the environment variable \texttt{OMP_SCHEDULE}.</td>
</tr>
<tr>
<td>AUTO</td>
<td>The scheduling decision is delegated to the compiler and/or runtime system.</td>
</tr>
</tbody>
</table>
us to know how a certain loop is scheduled. This stops us from analyzing and characterizing a loop behavior.

5.2.2 Power Capping or Power Clamping

Intel Sandybridge and subsequent processors have introduced a technique called power capping to limit the power consumption of different subsystems (CPU, memory, GPU) through RAPL\cite{48} interface. This does not limit the power consumption in the strictest sense, rather it guarantees that it will not exceed the average power usage given a time window and power limit. Different processor versions offer power capping on different sub systems. Our experimental machine does not allow DRAM power capping, so we only capped the CPU power (Package Power). Also, each subsystem has a minimum cap size, as for our system 54W is the minimum power cap size and TDP of our system is 115W. We picked 5 power cap levels (55W, 70W, 85W, 100W and 115W) within this limit. Also, the minimum time window to cap the power for the system is 0.000977s; in all our experiments, we used this minimum limit to cap the power of the CPU sub system. We used libmsr\cite{49}, a library developed at Lawrence Livermore National Laboratory to cap the CPU power of our system.

5.2.3 Creating the Search Space

Our goal in this experiment was to create a comprehensive configuration search space of the above explained 4 parameters (Power Capping level, number of threads, Scheduling Policy & Chunk sizes). We tried to create the search space in a way that is manageable and still large enough to provide us key insight on the impact of these parameters on a specific OpenMP parallel region. We used a parallel region as the basic block for our measurement.

Although there are situations where multiple parallel loops can be part of a single parallel region, we assumed all the parallel loops inside a single parallel region conforms to a single optimal configuration. There are two main reasons for this. First, number of threads is a parallel region parameter rather than a parallel loop parameter. By that, we mean that designated number of threads are created before execution enters a specific parallel region and those threads are used during the execution of the parallel loop inside that parallel region. Second, it provides simplicity during experimentation and measurement. For example, a single loop may not be large enough to have a power cap impact (described later), but all the loops inside a parallel region combined may be large enough to have a power cap impact. It is also easier to collect information for a parallel region than a specific loop inside a parallel region.
Table 5.3 shows the different values of these 4 parameters we used during our experimentation. We used all possible combination of these given parameters. So, we used 735 (5*7*3*7) different configurations on each parallel region. We changed these configurations on the runtime.

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Used Values</th>
</tr>
</thead>
<tbody>
<tr>
<td>Power Cap Size (5)</td>
<td>55W, 70W, 85W, 100W &amp; 115W</td>
</tr>
<tr>
<td>Number of threads (7)</td>
<td>1, 2, 4, 8, 16, 24 &amp; 32</td>
</tr>
<tr>
<td>Scheduling Policy (3)</td>
<td>STATIC, DYNAMIC &amp; GUIDED</td>
</tr>
<tr>
<td>Chunk Sizes (7)</td>
<td>1, 8, 32, 64, 128, 256 &amp; 512</td>
</tr>
</tbody>
</table>

5.2.4 Measurement Details

We developed a tool to collect execution time and energy consumption information for each OpenMP parallel region. It used RAPL API. We chose 28 OpenMP parallel regions for our analysis. The selection was made based on execution time and load balancing behavior of these parallel regions. We did not consider parallel regions with execution time less than 10 milliseconds, because it is advised to have around 10 milliseconds time window between two RAPL calls for a reliable reading [50].

We accumulated the execution time of each parallel region every time it was called throughout the application. There were two reasons for this. First, it showed the parallel region behavior on that configuration throughout the application. It is important for an application with load balancing issue across different invocations of the parallel region. For example, a parallel region can have a small workload first time it was invoked but a larger workload in later invocations. Second, we used RAPL to enforce a new power cap at the beginning of an application. The RAPL framework takes up to 200 milliseconds to warm-up before imposing a new power cap. Therefore, if a parallel region is called within this period, it may not experience the effect of the selected power level. However, in our experimentation, the parallel regions were called tens to hundreds of times during a complete run of an application. Therefore, even if a parallel region did not experience a new power cap in its first run, it experienced the new power cap in the later runs. Thus, the accumulated execution time provided a reliable reading. Besides that, we ran all the experiments three times and averaged it to get a more reliable reading.

In our experimentation, we capped the processor power, also known as the package power. The package consists of the cores, caches and other internal
circuitry. We also reported the package energy\(^3\) energy consumed by the processor package.

### 5.3 Experimental Setup

In this section, we provide an overview of our experimental environment.

#### 5.3.1 Machine Configuration

We used a dual socket machine with two 2.4 GHz quad-core Intel\textregistered Xeon\textregistered E5-2665 processors (based on the Intel Sandy Bridge architecture) with OpenSUSE 13.1 installed on it. The machine has a total of 16 cores (32 hyper-threaded threads) and 16 GB of memory. Each socket has a minimum and maximum (TDP) operating processor/package power of 54W and 115W respectively specified by the manufacturer. The system was running on ‘ondemand’ power governor settings with turbo enabled.

#### 5.3.2 Packages and Libraries

**Compilers**

We used two compilers for our experimentation. These compilers are,

- **OpenUH** \[51\] is a research compiler based on Open64 developed at the University of Houston for Fortran/C/C++. It has been used to research OpenMP features and their implementation. We used version 3.0.33 of this compiler.

- **Intel compiler** is industry standard compiler developed by Intel corporation. We used version 16.0 of this compiler.

We used \(-O3\) optimization level of these compilers to compile the programs.

**OpenMP Tools Interface (OMPT)**

OMPT\[52\] is an Application Programming Interface (API) that enables portable tools to collect performance analysis information of OpenMP programs.

\(^3\)We used the words *package energy* and *energy* synonymously.
Intel RAPL (Running Average Power Limit)

Intel provides an interface called Running Average Power Limit (RAPL) for on board energy measurement and power capping in Intel Sandy Bridge Processor family and onward. The RAPL framework provides energy and power consumption information via a set of Machine Specific Registers (MSR). For our experimentation, we used the libmsr library \[49\] to access the MSRs. RAPL caps the power on a subsystem based on average power consumption. It requires two input, a time window and a power limit. It ensures that an average power consumption during a given time window won’t exceed the power limit. The RAPL framework allows a minimum time window of 0.000977s. We used the minimum time window for our experiments.

PAPI (Performance Application Programming Interface)

PAPI \[53\] is a standard API for accessing hardware performance counters available on modern microprocessors. For our experiments, we used PAPI 5.3.0.

TAU (Tuning and Analysis Utilities)

TAU \[54\] is a portable profiling and tracing toolkit for performance analysis of parallel programs. We used the TAU framework to collect parallel region level dynamic feature (e.g. cache miss rate) information.

5.3.3 Benchmark Applications

We used five applications from the NAS Parallel Benchmark (NPB) suite \[55\]. Two of the applications from NPB; BT, and SP, are proxy applications. Three applications, CG, EP, and FT, are based on computational kernels. The benchmark applications can be run with six different data sets; S, A, B, C, D, and E. S is the smallest, and E is the largest data set. We used dataset B for our experimentation.

Here is a brief overview of these applications:

**BT:** A simulated CFD computational kernel that uses an implicit algorithm to solve 3-dimensional (3-D) compressible Navier-Stokes equations. The finite differences solution to the problem is based on an Alternating Direction Implicit (ADI) approximate factorization that decouples the x, y and z dimensions.

**SP:** A simulated CFD application that has a similar structure to BT. The finite differences solution to the problem is based on a Beam-Warming approximate factorization that decouples the x, y and z dimensions.
CG: Uses a Conjugate Gradient method to compute an approximation to the smallest eigenvalue of a large, sparse, unstructured matrix.

EP: An Embarrassingly Parallel Benchmark. It generates pairs of Gaussian random deviates according to a specific scheme.

FT: Contains the computational kernel of a 3-D fast Fourier Transform (FFT)-based spectral method.

5.4 Impact of the Optimal Runtime Configuration on a Power Capped Node

In this section, we present our results and analysis using 28 parallel regions from the NPB applications. Due to the space constraint, we present the results primarily with the OpenUH runtime. However, we use the Intel runtime results to generalize the findings. We determined the optimal configuration of each parallel region based on its execution time. Some parallel regions showed the optimal performance with the default configuration, and some showed the optimal performance with a non-default configuration. Based on this observation, we divided the parallel regions into the following two groups:

- *Default parallel regions* are the parallel regions whose optimal configuration is the default configuration.

- *Non-Default parallel regions* are the parallel regions whose optimal configuration is different from the default configuration.

With the OpenUH runtime, we found 35% of the total parallel regions as *Non-Default parallel regions* across all power levels. For example, at 85W power level, 10 out 28 parallel regions were *Non-Default parallel regions*. These *Non-Default parallel regions* showed significant execution time and energy consumption improvement with optimal configurations as compared to the default configuration\(^4\). Figure 5.2 shows the improvement in execution time and energy consumption for *Non-Default parallel regions* at 85W power level. The pink bar in the figure represents execution time improvement while the greenish bar represents energy consumption improvement. On average, we observed 26% execution time and 30% energy consumption improvement. The largest improvement is observed in \texttt{SPx_solve_1} parallel

\(^4\)OpenUH and Intel runtimes have the same default configuration. They both use the maximum number of available threads (e.g., 32 in our case), static scheduling, and chunk sizes calculated dynamically by dividing the total number of loop iterations by the number of threads.
region from SP; where we achieve 67% in execution time and 72% in energy consumption improvement. We observed similar patterns at other power levels as well.

We witnessed a similar behavior with the Intel runtime. On average, 30% of the total parallel regions were Non-Default parallel regions. Using the optimal configurations, these parallel regions showed an average execution time and energy consumption improvement around 29% and 32% respectively. The largest improvement was observed in SP.x.solve.1 parallel region from SP where an improvement of 51% in execution time and 63% in energy consumption was achieved.

These results show that using the default configuration blindly could result in sub-optimal performance for many parallel regions. This observation brings us to some interesting questions.

![Figure 5.2: Execution time and energy consumption improvement using optimal configurations over the default configuration for Non-Default parallel regions. We used the OpenUH runtime.](attachment:figure5.2.png)

Does the performance improvement using the optimal configuration persist across all the power caps for Non-Default parallel regions?

Figure 5.3 shows the percentage improvement in execution time and energy consumption for Non-Default parallel regions using optimal configura-
tions. We show the result for all the five power caps. Figure 5.3a shows the execution time improvement of the parallel regions. Each line in the figure represents execution time improvement for a specific power level. It is evident from the figure that almost all the lines coincide with each other. This result shows that the execution time improvement is comparable across all power caps. However, an optimal configuration for a parallel region may change across different power caps (Section 5.4). Figure 5.3b shows the improvement in energy consumption across the different power caps. We observe similar behavior in this case as well. We observed a similar trend in execution time and energy consumption improvement using the Intel runtime as well.

Based on this analysis, we can conclude that for Non-Default parallel regions, the scope for improvement is persistent across all power levels.

![Graphs showing execution time and energy consumption improvement](image)

Is an optimal configuration for a parallel region uniform across all the power caps?

We analyzed the optimal configurations of the experimental parallel regions across different power caps for both runtimes (OpenUH and Intel). We observed that parallel regions which were Non-Default at the highest power level (115W) remained Non-Default at the lower power levels (100W, 85W, 70W, 55W). However, these Non-Default parallel regions frequently changed their optimal configurations across power caps. The parallel region discussed in Section 5.1 is an example of Non-Default parallel region. Figure 5.1 clearly shows that the parallel region has different optimal configurations at different power caps.
To elaborate on this behavior of Non-Default parallel regions, we found that the optimal configuration does a variable workload distribution to different threads and the work distribution depends on the current state of the cores and caches. With the change of power caps, the behavior of cores and caches changes. As a result, the workload distribution also needs to be changed to optimize the usage of cores and caches. To facilitate the changed workload distribution, the optimal configuration changes with the change of power cap. We also noticed that an optimal configuration from a different power cap might degrade execution time significantly; especially if the difference between the power levels is high. For example, when \texttt{SP.z.solve.1} parallel region from the SP application was run with the optimal configuration of the 100W power level at 55W, the execution time performance degraded about 15% as compared to the optimal configuration of 55W. Table 5.4 shows the detailed results across all power caps.

Table 5.4: Optimal configuration for \texttt{SP.z.solve.1} parallel region on different power caps & Performance penalty of using an optimal configuration of a certain power cap to another

<table>
<thead>
<tr>
<th>Power Cap</th>
<th>Configurations (Thrd Schl Chunk)</th>
<th>%Performance Loss</th>
</tr>
</thead>
<tbody>
<tr>
<td>55W</td>
<td>32 GUIDED 8</td>
<td>0 2.78 2.99 1.04 3.37</td>
</tr>
<tr>
<td>70W</td>
<td>32 DYNAMIC 8</td>
<td>12.27 0 1.46 0.63 0.68</td>
</tr>
<tr>
<td>85W</td>
<td>32 GUIDED 8</td>
<td>11.10 0.66 0 0.19 0.01</td>
</tr>
<tr>
<td>100W</td>
<td>16 DYNAMIC 16</td>
<td>14.52 0.39 0.62 0 0</td>
</tr>
<tr>
<td>115W</td>
<td>16 GUIDED 16</td>
<td>14.52 0.39 0.62 0 0</td>
</tr>
</tbody>
</table>

Based on this analysis, we can conclude that a Non-Default parallel region at higher power levels remain Non-Default parallel region at lower power levels and optimal configurations for Non-Default parallel regions are not uniform across all power caps.

Why does an optimal configuration improve execution time and energy consumption for Non-Default parallel regions?

An optimal configuration optimizes cache behavior and load balancing to improve performance. To verify this, we analyzed the hardware behavior of Non-Default parallel regions using both optimal and default configura-
tions. We collected 19 dynamic hardware features for these parallel regions using PAPI. We used TAU to collect these features. The collected features included cache behavior, branch behavior, instruction mix, and idle state behavior.

We observed that features related to cache and idle state of the processor played a decisive role in the performance gain. Features like $L_1$, $L_2$, $L_3$ cache miss rate, TLB (Transfer Look-Aside Buffer) miss rate represent cache behavior. The smaller the miss rate, the better the performance. CPI (Cycles Per Instruction) and stall cycles per instruction capture the waiting/stall state behavior of the processor. The waiting-state behavior of the processor characterizes the load balancing behavior of the parallel region. When the time spent by the processor in the waiting state becomes less, the load balance becomes better. Figure 5.4 shows the dynamic feature comparison between the default and optimal configurations for $SP_z\_solve\_1$ parallel region at 85W power level using the OpenUH runtime. This parallel region has 61% execution time improvement and 67% energy consumption improvement using the optimal configuration at 85W power cap. The figure shows that the optimal configuration improves both cache and idle state behaviors. Similar behavior was observed with other power levels. For other $Non$-$Default$ parallel regions, we also observed that optimal configuration improved cache and stall state behaviors. We saw similar results with the Intel runtime.

Based on this analysis, we can conclude that the optimal configuration improves load balancing and cache behavior of a parallel region, and in the process, improves execution time and energy consumption.

Do we always get the best performance at the highest power cap (TDP)?

A well-known concept in the HPC area is that more power provides more performance. So, if everything else (e.g., runtime configuration, data set size) remains constant, performance in higher power caps will be at least the same if not better than performance in lower power caps. However, when we compared the best execution time configurations of all the power caps, we observed that more than 85% parallel regions had absolute best (best among all power caps) configuration in lower power caps than TDP. These parallel regions had a configuration in lower power cap which outperformed the best configuration in TDP (115W in our case). Figure 5.5 shows the performance and energy improvement of these parallel regions in the absolute best configuration (in lower power cap) compared to the local best configuration in TDP. Table 5.5 shows the details of these absolute best configurations for these parallel regions. In the figure, we show the parallel regions divided into 4 different power caps.
These power caps represent the power cap on which they had the absolute best configuration.

We see these results primarily because these parallel regions have a lower power profile. That means without power capping (running in TDP) these parallel regions consume less power than TDP. As a result, running at a lower power level does not impact the natural progression of these parallel regions. However, why are we getting additional performance improvement at a lower power level? There are two possible explanations.

The first one is that the CPU frequency decreases at the lower powers caps. As a result, CPU speed becomes more synced with the cache and memory speed. So, the cache and memory accesses become more effective, and the number of cycles processors have to wait for data decreases. This, in turn, provides better performance. The behavior can be explained with a simple analogy. Imagine a New York City subway station, where a train can carry a certain number of passengers (e.g., 100 passengers) and passes the station every hour. However, the rate at which passengers can get into the train is 50 passengers per 45 minutes, and 15 more minutes to check for the train safety. Now if the train is in service for 10 hours, it will be able to carry 500 passengers. Now if we decrease the frequency of the train, say now it passes the station every 1:45 hour. Then to carry the same 500 passengers the train
would take only 8 hours and 45 minutes instead of 10 hours. Here the train frequency is analogous to processor frequency, while the passenger frequency is the memory frequency. Researchers have observed similar behavior with I/O bound applications \cite{56}. However, we plan to explore this possibility in our case in-depth in our future work.

The second one is that the configurations we selected for our experimentation are discrete. They are not comprehensive in the purest sense (e.g., we used 7 (1, 8, 32, 64, 128, 256 & 512) chunk size level out of possible 512 (1-512)). If we have used all possible combinations, it would have taken us more than a year to run all these configurations, and this would have made this study extremely difficult to manage. Hence, we chose the configuration search space in such a way that provides us with good insight while maintaining a reasonable size. We tried to make sure that the performance difference between the configuration search space that we explored and the pruned configurations is not significant. Now at a certain power cap, we might have overlooked the best configuration.

Based on this analysis, we can conclude that an optimal power level for a parallel region may not always be the highest power level. A parallel region may exhibit the best performance at a lower power level in a certain system.

![Parallel Regions](image)

Figure 5.5: Parallel regions that have the best configuration at a lower power cap. Here we show the percent improvement in execution time and energy consumption using optimal configuration at optimal power cap compared to optimal configuration at TDP. Table 5.5
Table 5.5: Best configuration across all power levels. We show the power level, no. of threads (T), scheduling policy (S) and chunk size (C) that provides the best execution time.

<table>
<thead>
<tr>
<th>Parallel Region</th>
<th>Power</th>
<th>T</th>
<th>S</th>
<th>C</th>
</tr>
</thead>
<tbody>
<tr>
<td>BT_initialize_1</td>
<td>55W</td>
<td>32</td>
<td>STATIC</td>
<td>1</td>
</tr>
<tr>
<td>SP_txinvr_1</td>
<td>55W</td>
<td>32</td>
<td>DEFAULT</td>
<td>0</td>
</tr>
<tr>
<td>SP_tzetar_1</td>
<td>55W</td>
<td>32</td>
<td>DEFAULT</td>
<td>0</td>
</tr>
<tr>
<td>BT_compute_rhs_1</td>
<td>70W</td>
<td>16</td>
<td>GUIDED</td>
<td>16</td>
</tr>
<tr>
<td>SP_exact_rhs_1</td>
<td>70W</td>
<td>32</td>
<td>DEFAULT</td>
<td>0</td>
</tr>
<tr>
<td>BT_x_solve_1</td>
<td>85W</td>
<td>32</td>
<td>GUIDED</td>
<td>1</td>
</tr>
<tr>
<td>CG_main_1</td>
<td>85W</td>
<td>32</td>
<td>DEFAULT</td>
<td>0</td>
</tr>
<tr>
<td>FT_c_i_c_1</td>
<td>85W</td>
<td>32</td>
<td>DEFAULT</td>
<td>0</td>
</tr>
<tr>
<td>SP_compute_rhs_1</td>
<td>85W</td>
<td>32</td>
<td>GUIDED</td>
<td>8</td>
</tr>
<tr>
<td>SP_rhs_norm_1</td>
<td>85W</td>
<td>32</td>
<td>STATIC</td>
<td>1</td>
</tr>
<tr>
<td>SP_z_solve_1</td>
<td>85W</td>
<td>8</td>
<td>STATIC</td>
<td>1</td>
</tr>
<tr>
<td>BT_add_1</td>
<td>100W</td>
<td>32</td>
<td>DEFAULT</td>
<td>0</td>
</tr>
<tr>
<td>BT_exact_rhs_1</td>
<td>100W</td>
<td>32</td>
<td>DEFAULT</td>
<td>0</td>
</tr>
<tr>
<td>BT_y_solve_1</td>
<td>100W</td>
<td>32</td>
<td>STATIC</td>
<td>1</td>
</tr>
<tr>
<td>BT_z_solve_1</td>
<td>100W</td>
<td>32</td>
<td>STATIC</td>
<td>1</td>
</tr>
<tr>
<td>CG_conj_grad_1</td>
<td>100W</td>
<td>32</td>
<td>DEFAULT</td>
<td>0</td>
</tr>
<tr>
<td>FT_cffts3_1</td>
<td>100W</td>
<td>32</td>
<td>DEFAULT</td>
<td>0</td>
</tr>
<tr>
<td>FT_c_i_1</td>
<td>100W</td>
<td>32</td>
<td>DEFAULT</td>
<td>0</td>
</tr>
<tr>
<td>FT_evolve_1</td>
<td>100W</td>
<td>32</td>
<td>DEFAULT</td>
<td>0</td>
</tr>
<tr>
<td>FT_init_ui_1</td>
<td>100W</td>
<td>32</td>
<td>DEFAULT</td>
<td>0</td>
</tr>
</tbody>
</table>
Table 5.5: Best configuration across all power levels. We show the power level, no. of threads (T), scheduling policy (S) and chunk size (C) that provides the best execution time.

<table>
<thead>
<tr>
<th>Parallel Region</th>
<th>Power</th>
<th>T</th>
<th>S</th>
<th>C</th>
</tr>
</thead>
<tbody>
<tr>
<td>SP_add_1</td>
<td>100W</td>
<td>32</td>
<td>DEFAULT</td>
<td>0</td>
</tr>
<tr>
<td>SP_initialize_1</td>
<td>100W</td>
<td>32</td>
<td>DEFAULT</td>
<td>0</td>
</tr>
<tr>
<td>SP_ninvr_1</td>
<td>100W</td>
<td>32</td>
<td>DEFAULT</td>
<td>0</td>
</tr>
<tr>
<td>SP_pinvr_1</td>
<td>100W</td>
<td>32</td>
<td>DEFAULT</td>
<td>0</td>
</tr>
<tr>
<td>SP_x_solve_1</td>
<td>100W</td>
<td>8</td>
<td>DYNAMIC</td>
<td>1</td>
</tr>
<tr>
<td>SP_y_solve_1</td>
<td>100W</td>
<td>8</td>
<td>DYNAMIC</td>
<td>1</td>
</tr>
</tbody>
</table>

How does the OpenMP parallel region level improvement affect the overall application execution time and energy consumption?

We ran the NPB applications with each OpenMP parallel region using its specific optimal configuration and compared the application performance with the default configuration performance. Figure 5.6 shows the execution time and energy consumption improvement for the SP application with the OpenUH runtime. The figure shows 18 – 26% improvement in execution time and 22 – 38% improvement in energy consumption across different power levels. We observed similar performance behavior in other applications as well. We observed comparable performance behavior with the Intel runtime.

Based on this analysis, we can conclude that parallel region level performance improvement has an impact on the overall application-level performance.

5.5 Finding Energy Efficient Configurations

In the previous section, the main focus of our analysis was on execution time at different power levels. In this section, we move our primary focus to energy consumption. We investigate whether the runtime configuration that gives the best (smallest) execution time always leads to the best (lowest) energy consumption.
Figure 5.6: Execution time and energy consumption improvement for SP using an optimal configuration per OpenMP parallel region compared to the default configuration with the OpenUH runtime.

For this analysis, we used 23 out of our 28 experimental parallel regions. We removed 5 parallel regions from our previous analysis due to the following reason: in our previous analysis, we considered parallel regions whose accumulated execution time is more than ten milliseconds (Section ??). This means that we were evaluating the execution time of a parallel region during the overall application run, since a parallel region may be executed several times during the whole application run. Five of these parallel regions have a single run execution time of less than 10 milliseconds, while their overall execution time during the span of the application is more than 10 milliseconds. Considering these five parallel regions in our previous analysis allowed us to run our analysis on more parallel regions, while not affecting the overall outcome of our findings. This is because a slight change in package energy reading has a negligible impact on the overall execution time and power capping based on our experiments.

However, the main focus of our analysis here is on energy consumption; therefore, a slight misreading may impact the analysis outcome. To get rid of this scenario and provide an accurate analysis, we removed any parallel region whose single run execution time is less than 10 milliseconds even if its accumulated execution time is larger. There was a total of 5 parallel regions with such behavior.
Now, we analyzed the energy efficient configurations through two different settings:

1. Finding energy efficient configurations for a specific power level
2. Finding energy efficient configurations among all power levels

We started with energy efficient configurations for individual power level, and then we moved on to energy efficient configurations across all power levels.

5.5.1 Finding energy efficient configurations for a specific power level

In this setting, we tried to find if there exist any configurations that result in a better energy consumption than the optimal configuration. As the optimal configuration is the best execution time configuration, these energy efficient configurations will incur some execution time penalty. We wanted to analyze these energy efficient configurations if there exist any and see how much energy improvement they bring and at what cost (execution time penalty).

The first thing we observed from our analysis is that default and non-default parallel regions showed slightly different behaviors.

For non-default parallel regions even though there exist configurations that result in better energy consumption than the optimal configuration, the energy improvement was very small. On average, we observed an energy consumption improvement of 4.2% at the expense of an execution time penalty of 10% in 9 out of 10 non-default parallel regions. The highest energy improvement we saw in these 9 parallel regions was about 10%. Only one parallel region ($SP\_compute\_rhs$) showed a slightly higher energy improvement (26%) but with a larger execution time penalty (48%).

Figure 5.7 shows the percentages of energy consumption improvement and execution time penalty using the best energy efficient configurations compared to the optimal configurations. We also show the detailed optimal configuration (Best Execution Time Configuration) and Best Energy Efficient Configuration in Table 5.6. We only show the behavior of non-default parallel regions at TDP power level (115W) in Figure 5.7. We observed similar behavior at other power levels.

We see this kind of behavior because we are only comparing configurations within a specific power level. As we have shown before (Section 6.3), the optimal configuration on a certain power level optimizes processor utilization, cache behavior and load balancing for that specific power level. As a result, energy consumption for the optimal configuration is also going to be very
close to optimal. Furthermore, although there is one optimal configuration for a certain parallel region at a certain power level, there exist configurations that perform very close to the optimal configuration. Inherently, these configurations utilize the available resources in a slightly different manner than the optimal configuration. Some of these close performing configurations result in a slightly better energy consumption but with a small execution time penalty.

As for the default parallel regions, we observed that for 10 of the 13 parallel regions the default configuration was also the best energy efficient configuration at almost all power levels. This is expected because of the default parallel regions’ characteristics. As we have explained in (Section 6.3), the default parallel regions showed good load balancing and cache behavior. In other words, they were able to utilize the available resources in the best possible way. So it is understandable that these parallel regions achieve the best energy utilization along with the best execution time using the default configuration.

However, 3 of the default parallel regions (FT_evolv_1, FT_init_1 and SP_tzetar_1) showed their best energy behavior at configurations different from the default. Beside this, it was also observed that the energy consumption improvement for these configurations was substantial (up to 55%). To elaborate on this significant improvement, we found that 2 (FT_evolv_1 and
Table 5.6: Best Execution Time and Energy efficient configuration for Non-Default parallel regions at TDP power level (115W). We show no. of threads (T), scheduling policy (S) and chunk size (C) that provides the best execution time and the best energy efficiency.

<table>
<thead>
<tr>
<th>Parallel Region</th>
<th>Best Execution Time Configuration</th>
<th>Best Energy Efficient Configuration</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Threads Schedule Chunk Size</td>
<td>Threads Schedule Chunk Size</td>
</tr>
<tr>
<td>BT_compute_rhs_1</td>
<td>16 GUIDED 16</td>
<td>4 GUIDED 32</td>
</tr>
<tr>
<td>BT_initialize_1</td>
<td>32 STATIC 1</td>
<td>32 DEFAULT 0</td>
</tr>
<tr>
<td>BT_x_solve_1</td>
<td>32 STATIC 1</td>
<td>24 DYNAMIC 1</td>
</tr>
<tr>
<td>BT_y_solve_1</td>
<td>32 STATIC 1</td>
<td>24 GUIDED 1</td>
</tr>
<tr>
<td>BT_z_solve_1</td>
<td>32 STATIC 1</td>
<td>24 GUIDED 1</td>
</tr>
<tr>
<td>EP_main_3</td>
<td>32 DYNAMIC 1</td>
<td>32 DYNAMIC 8</td>
</tr>
<tr>
<td>SP_compute_rhs_1</td>
<td>32 GUIDED 8</td>
<td>2 GUIDED 32</td>
</tr>
<tr>
<td>SP_x_solve_1</td>
<td>8 DYNAMIC 1</td>
<td>8 GUIDED 16</td>
</tr>
<tr>
<td>SP_y_solve_1</td>
<td>8 DYNAMIC 1</td>
<td>8 GUIDED 8</td>
</tr>
<tr>
<td>SP_z_solve_1</td>
<td>8 DYNAMIC 1</td>
<td>8 DYNAMIC 16</td>
</tr>
</tbody>
</table>

FT_init_ui_1) of these 3 parallel regions are in fact initialization regions. This means that they do not perform any explicit computation. This initialization behavior might be a reason for this kind of behavior for these two parallel regions. However, it does not explain the behavior of SP_tzetar_1, which is not an initialization region. Unfortunately, due to the lack of similarity between SP_tzetar_1 and the other 2 parallel regions, we could not provide a specific explanation of its behavior.

5.5.2 Finding energy efficient configurations among all power levels

Our previous analysis showed that the difference in resource utilization might result in better energy consumption with minor execution time penalty, especially for Non-default parallel regions. Although the energy improvement was small, it gave us an indication that it is possible to find more energy efficient configuration than the optimal configuration.

To analyze this behavior in details we added one more factor to our analysis that we did not consider in our previous analysis. That factor is the power level. Power level has a huge impact on energy consumption behavior. So what
Table 5.7: Percent Package Energy improvement with 10% to 90% performance penalty compared to the optimal configuration across all power levels.

<table>
<thead>
<tr>
<th>Parallel Region Name</th>
<th>%Package Energy Improvement with 10%-90% Execution Time Penalty</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>10%</td>
</tr>
<tr>
<td>BT_compute_rhs_1</td>
<td>5.29</td>
</tr>
<tr>
<td>BT_exact_rhs_1</td>
<td>7.35</td>
</tr>
<tr>
<td>BT_initialize_1</td>
<td>0.99</td>
</tr>
<tr>
<td>BT_x_solve_1</td>
<td>3.34</td>
</tr>
<tr>
<td>BT_y_solve_1</td>
<td>4.19</td>
</tr>
<tr>
<td>CG_conj_grad_1</td>
<td>18.59</td>
</tr>
<tr>
<td>CG_main_1</td>
<td>4.69</td>
</tr>
<tr>
<td>FTcffts1_1</td>
<td>3.37</td>
</tr>
<tr>
<td>FTcffts3_1</td>
<td>11.45</td>
</tr>
<tr>
<td>FTcffts2_1</td>
<td>9.18</td>
</tr>
<tr>
<td>FT_compute_indexmap_1</td>
<td>9.60</td>
</tr>
<tr>
<td>FT_compute_initial_conditions_1</td>
<td>6.18</td>
</tr>
<tr>
<td>FT_evaluate_1</td>
<td>26.32</td>
</tr>
<tr>
<td>FT_init_ui_1</td>
<td>54.43</td>
</tr>
<tr>
<td>SP_compute_rhs_1</td>
<td>30.01</td>
</tr>
<tr>
<td>SP_initialize_1</td>
<td>1.94</td>
</tr>
<tr>
<td>SP_tzetar_1</td>
<td>0.00</td>
</tr>
<tr>
<td>SP_x_solve_1</td>
<td>13.93</td>
</tr>
</tbody>
</table>
happens if we extend our analysis to all possible power levels? Is it possible to find a better energy efficient configuration at a different power level? If yes, is the improvement substantial or small? To answer these questions, we compared all the configurations across all power levels with the absolute optimal configuration (the configuration that results in the best execution time across all power levels). We performed this analysis with Energy-Time trade-off in mind. We were not just looking for the most energy efficient configurations; we were looking for configurations with better energy efficiency within a bounded performance penalty compared to the absolute optimal configuration. This way the findings can be useful for someone who is interested in energy efficient configurations within a certain level of performance penalty. We tried to find the best energy efficient configurations that have $10\% – 90\%$ performance penalty compared to the absolute optimal configuration. Table 5.7 shows the Package Energy improvement of those configurations within $10\% – 90\%$ performance penalty compared to the absolute optimal configuration. The first column shows the parallel region names, while other columns show the %Package Energy improvement with a certain performance penalty. For example, parallel region FT_cffts1_1 (blue row in Table 5.7) has a configuration that provides a $3.37\%$ energy improvement but has performance penalty less than $10\%$ compared to the optimal configuration. If we increase the performance penalty...
penalty range to 30% we can find a configuration that provides a 15.02% energy improvement. If we increase the penalty range to 40% there is no new configuration, so the energy efficient configuration found in 30% penalty is used. For 50% penalty there is a configuration with 22.40% energy improvement and so on.

We see different behaviors with different types of parallel regions. Some parallel regions show a huge energy gain with a small performance penalty (e.g., FT\_evolve\_1, FT\_init\_ui\_1, SP\_compute\_rhs\_1 has 26%, 54% and 30% package energy improvement with only 10% performance penalty), while some show nominal energy improvement even with large performance penalty (e.g., BT\_compute\_rhs\_1, BT\_initialize\_1, SP\_initialize\_1 has only 5%, 3% and 2% package energy improvement with 90% performance penalty). To understand the behavior of these parallel regions we now focus only on the configurations that provide the best energy improvement within 10% - 90% performance penalty. Figure 5.8 shows the comparison (Energy-Time trade-off) and Table 5.8 shows the detailed optimal configuration (Best Execution Time Configuration) and Best Energy Efficient Configuration.

We observed that these highly energy efficient configurations are usually obtained at the lower power level than the absolute optimal configuration’s power level. In fact, for 19 out of 23 parallel regions in this analysis, the most energy efficient configuration was found at the lowest power level (55W). This result demonstrates that lower power levels are usually better for energy efficiency. This energy efficient behavior at the lower power levels probably comes from the possible lower frequency at those power levels.

However, there are 4 parallel regions whose most energy efficient configurations were found at a higher power level than 55W. Interestingly, these 4 parallel regions (FT\_evolve\_1, FT\_init\_ui\_1, SP\_compute\_rhs\_1 and SP\_tsetar\_1) are the ones that showed a deviant behavior (much higher energy improvement) in the first part of this analysis (Section 5.5.1). We believe these two behaviors are connected. Unfortunately, we do not have enough evidence to provide a definitive explanation as to why they show this behavior. This is primarily because the number of parallel regions showing this behavior is small (only 4) and without clear similarity patterns among them. We plan to investigate this phenomenon in depth in our future work.

Based on the analysis in this section, we can conclude that it is possible to find a more energy efficient configuration than the optimal execution time configuration. However, the energy improvement from these configurations produces a cost of execution time penalty. Moreover, most of the parallel regions showed their most energy efficient behavior at the lower power levels.
5.6 Potential Impact of these findings and Concluding Remarks

The findings we have presented in this chapter can have a large and long-lasting impact on future power aware HPC. Following is a list of possible scenarios that can be impacted by these findings.

- We have shown that an optimal runtime configuration is dependent on a specific parallel region and choosing a suboptimal configuration may degrade the performance significantly. As most of the application has parallel regions with different behavior, one should focus on fine-grain parallel region level optimization to achieve the best performance.

- We have also shown that optimal configuration changes at different power levels. Now in the exascale era, we may see situations where the power level may change during the program execution. This could happen due to various reasons such as power budget issue, node failure, node migration. In those scenarios, a predefined configuration may not be feasible. It would require a system like ARCS [57] that could adapt to the changing environment and choose runtime configurations accordingly.

- We have demonstrated that with careful configuration selection one can achieve better performance at lower power level depending on the parallel region behavior. This opens up a lot of possibilities. One such possibility is fine-grained power shifting across nodes. If we can classify the parallel regions based on their behavior, we can come up with appropriate power level for these parallel regions. Then based on the availability and necessity we can shift power across nodes when we have more power allocated than we need, or we can request more power from other nodes if necessary.

- Finally, we have observed that the best execution time configuration does not always result in the best energy consumption. It is also possible to achieve better energy efficiency than the optimal configuration but with an execution time penalty. When energy consumption in a system is of most importance, one could focus on energy efficient configurations rather than optimal configurations. One could also decide on execution time and energy consumption trade-offs based on his/her need.

These are only some of the scenarios which can be benefited from this work but the possibilities are not limited to these. We believe this work would provide a guideline for future intra-node power aware work focusing on OpenMP.
Table 5.8: Best execution time and energy efficient configuration across all power levels using OpenUH runtime. We show power level, no. of threads (T), scheduling policy (S) and chunk size (C) that provides the best execution time and the best energy efficiency.

<table>
<thead>
<tr>
<th>Parallel Region</th>
<th>Best Execution Time Configuration</th>
<th>Best Energy Efficient Configuration</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Power Level</td>
<td>T</td>
</tr>
<tr>
<td>BT_compute_rhs_1</td>
<td>70</td>
<td>16</td>
</tr>
<tr>
<td>BT_exact_rhs_1</td>
<td>100</td>
<td>32</td>
</tr>
<tr>
<td>BT_initialize_1</td>
<td>55</td>
<td>32</td>
</tr>
<tr>
<td>BT_x_solve_1</td>
<td>85</td>
<td>32</td>
</tr>
<tr>
<td>BT_y_solve_1</td>
<td>100</td>
<td>32</td>
</tr>
<tr>
<td>BT_z_solve_1</td>
<td>100</td>
<td>32</td>
</tr>
<tr>
<td>CG_conj_grad_1</td>
<td>100</td>
<td>32</td>
</tr>
<tr>
<td>CG_main_1</td>
<td>85</td>
<td>32</td>
</tr>
<tr>
<td>EP_main_3</td>
<td>115</td>
<td>32</td>
</tr>
<tr>
<td>FT_c_j_1</td>
<td>100</td>
<td>32</td>
</tr>
<tr>
<td>FT_c_j_c_l</td>
<td>85</td>
<td>32</td>
</tr>
<tr>
<td>FT_cffts1_c_l</td>
<td>115</td>
<td>32</td>
</tr>
<tr>
<td>FT_cffts3_c_l</td>
<td>100</td>
<td>32</td>
</tr>
<tr>
<td>FT_cffts2_c_l</td>
<td>115</td>
<td>32</td>
</tr>
<tr>
<td>FT_evolve_1</td>
<td>100</td>
<td>32</td>
</tr>
<tr>
<td>FT_init_ui_1</td>
<td>100</td>
<td>32</td>
</tr>
<tr>
<td>SP_compute_rhs_1</td>
<td>70</td>
<td>32</td>
</tr>
<tr>
<td>SP_exact_rhs_1</td>
<td>70</td>
<td>32</td>
</tr>
<tr>
<td>SP_initialize_1</td>
<td>100</td>
<td>32</td>
</tr>
<tr>
<td>SP_zetar_1</td>
<td>55</td>
<td>32</td>
</tr>
<tr>
<td>SP_x_solve_1</td>
<td>100</td>
<td>8</td>
</tr>
<tr>
<td>SP_y_solve_1</td>
<td>100</td>
<td>8</td>
</tr>
<tr>
<td>SP_z_solve_1</td>
<td>85</td>
<td>8</td>
</tr>
</tbody>
</table>
Chapter 6

ARCS: Adaptive Runtime Configuration Selection for OpenMP Applications

In the previous chapter, we explored the intra-node performance variance of applications using OpenMP loop-level parallelism. We observed that different OpenMP parallel regions or regions\(^1\) have different execution behavior. Therefore, a runtime execution environment that gives the best performance at a certain power level also differs for various OpenMP regions. If the OpenMP runtime parameters are not properly selected, one may see a severe performance degradation.

However, application developers usually choose to use the default parameter settings provided by an OpenMP runtime library. As a result, one gets sub-optimal performance for most of the existing OpenMP applications. The performance degradation problem becomes even more severe when the system environment changes of some/all nodes after a restart from a fault. Therefore, it is of paramount importance to have an adaptive mechanism that can choose the runtime parameters of different parallel regions of an application using OpenMP based on application and system behavior.

To address this issue, In this work, we present the ARCS (Adaptive Runtime Configuration Selection) framework that chooses the best OpenMP runtime configurations for parallel loops in an HPC application. We define OpenMP configurations as: (1) Number of Threads, (2) Scheduling Policy, and (3) Chunk Sizes. We test ARCS using the NAS Parallel Benchmark, and a proxy application LULESH. We show that for a given power level, efficient OpenMP runtime parameter selection can improve the execution time and

\(^1\)We use *OpenMP parallel regions* and *regions* synonymously.

85
energy consumption of an application up to 40% and 42% respectively.

The major contributions of this work are listed below:

- We present ARCS framework that selects the best OpenMP runtime
  configurations for OpenMP regions to optimize HPC applications under
  a power constraint.

- To the best of our knowledge, ARCS is the first fully automatic frame-
  work that chooses OpenMP runtime configurations with no involvement
  of the application programmer.

- ARCS chooses and adapts OpenMP runtime configurations dynamically
  based on OpenMP region and underlying architecture characteristics,
  resulting in efficient execution on a number of applications under a power
  constraint across different architectures.

6.1 ARCS Framework

The ARCS runtime is composed of two key software components. The first
component is a modified OpenMP runtime. The second component is the
APEX instrumentation and adaptation library. APEX integrates the Active
Harmony search engine, integrated as part of the APEX library. Figure 6.1
shows the integration of the components in the ARCS runtime.

6.1.1 OpenMP runtime with OMPT

A broad group of interested parties has been working on extending the
OpenMP specification to include a formal performance and debugging tool
interface \[58\]. In order to provide support for both instrumentation (event-
based) and sampling based tools, OMPT includes both events and states. The
OMPT draft specification is available as a Proposed Draft Technical Report
at the OpenMP Forum website \[59\]. The key OMPT design objectives are to
provide low overhead observation of OpenMP applications and the runtime in
order to collect performance measurements, provide stack frame support for
sampling tools and incur minimal overhead when not in use. OMPT specifies
support for a large set of events and states, covering the OpenMP 4.0 standard.
In addition, OMPT specifies additional insight into the OpenMP runtime in
the form of data structures populated by the runtime itself. These data struc-
tures include the parallel region and task identifiers, wait identifiers and stack
frame data. A reference OpenMP runtime implementation with OMPT support based on the open-source Intel runtime is available\footnote{https://github.com/OpenMPToolsInterface/LLVM-openmp} and OMPT has been integrated into performance tools such as TAU\cite{60} and APEX.

### 6.1.2 APEX

We have implemented a measurement and runtime adaptation library for asynchronous multitasking runtimes called Autonomic Performance Environment for eXascale (APEX)\cite{61,62}. The APEX environment supports both introspection and policy-driven adaptation for performance and power optimization objectives. APEX aims to enable autonomic behavior in software by providing the means for applications, runtimes, and operating systems to observe and control performance. Autonomic behavior requires both performance awareness (introspection), and performance control. APEX can provide introspection from timers, counters, node- or machine-wide resource utilization data, energy consumption, and system health, all accessed in real-time. The introspection results are analyzed in order to provide the feedback control mechanism.

The most distinguishing component in APEX is the policy engine. The policy engine provides controls to an application, library, runtime, and/or operating system using the aforementioned introspection measurements. Policies are rules that decide on outcomes based on the observed state captured by APEX. The rules are encoded as callback functions that are periodic or triggered by events. The policy rules access the APEX state in order to request profile values from any measurement collected by APEX. The rules can change runtime behavior by whatever means available, such as throttling threads, changing algorithms, changing task granularity, or triggering data movement.

APEX was originally designed for use with runtimes based on the ParalleX\cite{63} programming model, such as HPX\cite{64} or HPX-5\cite{65}. However, the APEX design has proven to be flexible enough to be broadly applied to other thread-concurrent runtimes such as OpenMP.

APEX integrates the auto-tuning and optimization search framework Active Harmony\cite{66}. In APEX, Active Harmony is directly integrated into the library to receive APEX performance measurements and suggest new parametric options in order to converge on an optimal configuration. Active Harmony implements several search methods, including exhaustive search, Parallel Rank Order and Nelder-Mead. In this work, we used the exhaustive and Nelder-Mead search algorithms. In our experiments, the ARCS-Offline method uses an exhaustive search to find the best configuration during one execution, then
executes again with that optimal configuration. Only the second execution with the optimal configuration is measured. The ARCS-Online method uses the Nelder-Mead search algorithm to search for and use an optimal configuration in the same execution.

Prior to running the examples with the framework, the NPB 3.3-OMP-C OpenMP benchmarks were exhaustively parameterized to explore the full search space for the OpenMP environment variables `OMP_NUM_THREADS` and `OMP_SCHEDULE` (schedule type and chunk size). From that initial dataset, the search space was manually reduced. Unlike the initial parameter search, ARCS can tune the settings for each OpenMP parallel region independently. The reduced set of search parameters was used to limit the search space that had to be explored at runtime. The final ranges explored by ARCS are listed in Table 6.1.

Using the policy engine, we designed a policy to tune OpenMP thread count, schedule, and chunk size based upon the reduced search space described above. At program initialization, the policy registers itself with the APEX policy engine, and receives callbacks whenever an APEX timer is started or stopped. The OMPT interface starts a timer upon entry to an OpenMP parallel region and stops that timer upon exit. When a timer is started for a parallel region which has not been previously encountered, the policy starts an Active Harmony tuning session for that parallel region. When a timer is stopped, the policy reports the time to complete the parallel region. When a

Figure 6.1: ARCS framework, based on the original APEX design.
Table 6.1: Set of ARCS search parameters for OpenMP parallel regions.

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Set of values</th>
</tr>
</thead>
<tbody>
<tr>
<td>Number of threads (Crill)</td>
<td>2, 4, 8, 16, 24, 32, default</td>
</tr>
<tr>
<td>Number of threads (Minotaur)</td>
<td>10, 20, 40, 80, 120, 160, default</td>
</tr>
<tr>
<td>Schedule Type</td>
<td>dynamic, static, guided, default</td>
</tr>
<tr>
<td>Chunk Size</td>
<td>1, 8, 16, 32, 64, 128, 256, 512, default</td>
</tr>
</tbody>
</table>

timer is started for a parallel region which has been previously encountered, the policy sets the number of threads, schedule, and chunk size to the next value requested by the tuning session, or, if tuning has converged, to the converged values. When the program completes, the policy saves the best parameters found during the search. When the same program is run again in the same configuration in the future, the saved values can be used instead of repeating the search process.

6.1.3 Overhead

The main overhead of ARCS can be characterized into three different types.

- **Configuration changing overhead**: ARCS changes the runtime configuration each time a region is executed. To change these configurations, ARCS uses the OpenMP runtime library routine `omp_set_num_threads()` and `omp_set_schedule()`. Time consumed during these routine calls adds some extra overhead. We call this overhead _Configuration Changing overhead_. This overhead is present in both Online and Offline strategies. In Crill, we calculated this overhead to be about 0.0008 sec in each region call. If a region is large enough, this overhead becomes insignificant. However, if the the region time is not large enough this overhead can become a significant factor.

- **APEX instrumentation overhead**: Overhead incurred due to APEX runtime instrumentation. Just like _Configuration changing overhead_, the impact of this overhead is also dependent on the region execution time. It is present in both Online and Offline strategies.
• **Search overhead**: In the online search strategy, finding the optimal configuration requires ARCS to test several runtime configurations before converging. Many of these configurations are not optimal, and as a result these sub-optimal configurations incur extra execution time. This additional execution time can be termed as *Search overhead*. This overhead is only present in the Online strategy. It is not present in *Offline strategy*, because in *Offline strategy* ARCS does not search for the optimal configuration, it reads it from the history file only once during the whole application lifetime. We observed this overhead to vary across regions based on how fast they converge to the optimal configuration. During our experimentation, we observed this overhead to reach as high as 10% of the total execution time.

### 6.2 Experimentation

#### 6.2.1 Test System

We evaluated our framework on two different systems, Crill and Minotaur. These systems differ in architecture, number of cores, memory size and power consumption.

**Crill** (hosted at the University of Houston) is a dual socket machine with two 2.4 GHz quad-core Intel® Xeon® E5-2665 processors (Sandy Bridge architecture). It has a total of 16 cores (32 hyper-threaded threads) and 16 GB of memory. It runs on OpenSUSE 13.1 and has a TDP limit of 115W.

**Minotaur** (hosted at the University of Oregon) is an IBM® S822LC system equipped with two 10-core IBM POWER8® processors that operate at 2.92 GHz. It has support for 160 hardware threads (8 per core) and 256 GB of memory. It is running Ubuntu Linux, version 15.04.

#### 6.2.2 Compiler & Libraries

We used GCC compiler version 4.9.2, the reference OpenMP runtime with OMPT support for our experimentation, and libmsr, a library that facilitates access to MSRs via RAPL interface for energy measurement and power capping.

#### 6.2.3 Benchmarks

We used three proxy applications, LULESH 2.0, BT and SP to evaluate ARCS. We selected these benchmarks because they exhibit performance and load
balancing behavior typical for a broad range of HPC applications.

**LULESH 2.0**\(^\text{[67]}\) is a shock hydrodynamics computational kernel from Lawrence Livermore National Laboratory. It approximates the hydrodynamics equations discretely by partitioning the spatial problem domain into a collection of volumetric elements defined by a mesh. It is built on the concept of an unstructured hex mesh. It is one of the most used proxy applications in the HPC area, and it shows excellent load balancing and cache behavior. We used mesh sizes of 45 and 60 for our experimentation.

**BT** is a simulated CFD computational kernel that uses an implicit algorithm to solve 3-dimensional (3-D) compressible Navier-Stokes equations. The finite differences solution to the problem is based on an Alternating Direction Implicit (ADI) approximate factorization that decouples the x, y and z dimensions. The resulting systems are Block-Tridiagonal of 5 × 5 blocks and are solved sequentially along each dimension. This application shows good load balancing behavior. We used data set sizes B (102 × 102 × 102) and C (164 × 164 × 164) with custom 1000 time steps.

**SP** is a simulated CFD computational kernel that has a similar structure to BT. The finite differences solution to the problem is based on a Beam-Warming approximate factorization that decouples the x, y and z dimensions. The resulting system has Scalar Pentadiagonal bands of linear equations that are solved sequentially along each dimension. It shows good load balancing behavior but poor cache behavior. For SP, we also used data set sizes B (102 × 102 × 102) and C (164 × 164 × 164) with custom 1000 time steps.

Both **BT** and **SP** are from NAS parallel benchmark suite\(^\text{[55]}\), version 3.3-OMP-C.

### 6.2.4 Experimental Details

We carried out extensive experiments to evaluate the impact of ARCS. We considered both the execution time and energy consumption during the evaluation. An optimal OpenMP runtime configuration for a region is dependent on the region’s characteristics, power cap level, workload size, and architecture. For that reason, we designed our experiments in such a way that they cover all these scenarios. We tested ARCS on five different power levels, two different workloads, and two distinct architectures (Intel Sandy Bridge and IBM POWER8).

As mentioned before, our primary experimental resource Crill is equipped with Intel Sandy Bridge processors, and our secondary resource Minotaur with IBM POWER8 architecture. In Crill, we had power capping privilege and access to the energy counters. For that reason we were able to evaluate the impact of ARCS at different power levels. We experimented on 55W, 70W,
85W, 100W and 115W (TDP for this processor) power level. We only limited the processor power (package power). A package consists of cores, caches and other internal circuitry. We used maximum power for other components (DRAM, Network card, etc.), because we did not have capping capability on these subsystems. We used RAPL for power capping and collecting energy information. We tried to tackle known issues of RAPL such as counter update frequency and the warm up period after enforcing a power cap during the experimentation to get reliable energy readings. We ran each experiments three times. We report the average of these runs for Crill(Sandybridge) as it was a dedicated resource. However, we report the minimum of these three runs for Minotaur(Power8) as it was a shared resource. We did this to minimize any interference.

As Minotaur is a relatively new resource, we did not have energy counter access nor power capping privilege. Therefore all the experiments conducted on this machine were using the default (TDP) power level of this machine. Also, all the evaluation done on this machine is based on execution time only. We evaluated both Online and Offline ARCS strategies in the above-mentioned environments.

6.3 Results and Analysis

In this section we present our experimental results. Through these results we show the impact of ARCS on different types of OpenMP applications. As mentioned previously, we evaluated ARCS on three different OpenMP applications. These applications vary in scalability, load balancing, and cache behavior. **LULESH** is a well-balanced application with good cache behavior. **BT** is also fairly well balanced with good cache behavior. **SP** is well balanced but shows poor cache behavior. We mainly concentrated on scalability, load balancing and caching because these are the behaviors that impact OpenMP performance the most.

In an OpenMP application with loop level parallelism, these behaviors can be controlled by the number of threads, scheduling policy and chunk sizes. The number of threads has a significant impact on scalability while scheduling policy and chunk sizes are very important for good load balancing and cache behavior. These behaviors not only affect the execution time performance, but they also impact energy consumption. Load balancing and cache behavior of an application are two of the main factors that define an application’s energy profile.

Applications with bad cache behavior tend to consume more energy [58]. If there is a cache miss, the system has to do the extra work of fetching the data
from the next level of cache or memory and in the process use I/O path which leads to extra energy consumption.

On the other hand, load balancing affects the energy consumption in a different way. Poor load balancing of an application leads the cores to wait in idle states in the synchronization points (barriers). Lightly loaded threads wait for highly loaded threads to finish their work. Even though current processors do a decent job at saving energy by entering the sleep state while waiting, entering and exiting sleep states incurs non-trivial overheads and can cause negative savings if the idle duration is short\cite{69}. In OpenMP regions, the waiting time is usually short. Therefore, improving the load balancing behavior is crucial to improving the energy profile of an OpenMP application. Not only that but also these behaviors impact an OpenMP application’s power profile, as power is the ratio of the energy consumption and execution time.

Moreover, cores and caches are the main power consuming components of a processor\cite{70}. The total power of a processor is divided between these two components. So when a power cap is imposed on a processor, it not only affects the performance of the cores but also impacts the cache performance. As a result, the load balancing and cache behavior also change with the change of the power cap.

Furthermore, these behaviors vary across different regions of an application. Therefore, choosing an optimal configuration (number of threads, scheduling policy, and chunk sizes) for each regions separately is no trivial task. But we show through extensive analysis that ARCS is able to do this job very proficiently.

In the following discussion, we analyze each application separately. We show that ARCS can potentially improve performance across different types of applications. We also demonstrate the effect of ARCS strategies at both application and region level using detailed analysis of dynamic features. We show the performance behavior across different power caps and different workload sizes. Finally, we show the ARCS performance across different architectures.

We compare the performance of ARCS strategies with the default configuration. The default configuration uses maximum number of available threads, static scheduling, and chunk sizes calculated dynamically by dividing total number of loop iterations by number of threads. We concentrate on both online and offline strategies for ARCS. Results shown here is based on Crill, unless mentioned otherwise. The same applies for the power cap; if nothing is mentioned, that means we are using the highest power cap (TDP).
6.3.1 SP

SP is an application which shows a good load balancing behavior and poor cache behavior with the default configuration. SP consists of 13 loop based OpenMP regions. However, almost 75% of its execution time is spent on four regions (compute rhs, x.solve, y.solve and z.solve). Among them, compute rhs has a poor load balancing and cache behavior, x.solve, y.solve and z.solve regions have good load balancing behavior but show poor cache behavior. To improve these regions’ performance, their load balancing and cache behavior has to be improved. Therefore, we need to find configurations that improve the load balancing and cache behavior of these regions. To find such configurations we applied ARCS on this application. Table 6.2 shows the optimal configuration chosen by ARCS-Offline strategy for these regions at TDP power.

Table 6.2: Optimal configuration chosen by ARCS-Offline strategy for SP regions.

<table>
<thead>
<tr>
<th>Region</th>
<th>Optimal Configuration</th>
</tr>
</thead>
<tbody>
<tr>
<td>compute rhs</td>
<td>16, guided, 8</td>
</tr>
<tr>
<td>x.solve</td>
<td>16, guided, 1</td>
</tr>
<tr>
<td>y.solve</td>
<td>8, static, default</td>
</tr>
<tr>
<td>z.solve</td>
<td>4, static, 32</td>
</tr>
</tbody>
</table>

In Figure 6.2 we show the feature comparison between the default configuration and the configurations chosen by ARCS-Offline, the best ARCS strategy. We compare the L1 cache miss rate in Figure 6.2a, L2 cache miss rate in Figure 6.2b, L3 cache miss rate in Figure 6.2c and OpenMP barrier (OMP_BARRIER) time in Figure 6.2d. The L1, L2 and L3 cache miss rates show the cache behavior of these regions. The OMP_BARRIER time shows the load balancing behavior; greater OMP_BARRIER time is a symptom of poor load balancing. For all of these metrics, lower values indicate better performance.

From these figures, we observe that all four regions show better cache and load balancing behavior with the ARCS strategy. Using the configuration chosen by ARCS, the OMP_BARRIER time is decreased by more than 50% in
all four regions compared to the default configuration, shown in Figure 6.2d. The best improvement, which is more than 80% is achieved in the \texttt{z_solve} region while a relatively smaller improvement (around 50%) is achieved in \texttt{compute_rhs}.

We also observed L1, L2 and L3 cache miss rate improvement. Although L1 and L2 cache behaviors show good improvement, the biggest improvement (up to 90%) is visible in L3 cache behavior. This is important for performance because L3 cache misses have the highest cache miss penalty. The improvement also shows that these configurations enabled different cores to maximize their use of the shared L3 cache.

The above analysis shows that ARCS strategies can improve the cache behavior and load balancing of SP regions. This leads to the question: how much do these improvements affect the overall application’s execution time and energy consumption? In Figure 6.3 we show the execution time and energy consumption comparisons between the default and ARCS strategies (\texttt{ARCS-Online} and \texttt{ARCS-Offline}). We show the results for five different power levels. We compare both execution time (in Figure 6.3a) and energy consumption (in Figure 6.3b). In Figure 6.3a we see that all the strategies in all five power levels outperform the default configuration by a large margin. The improvement varies between 26-40%. We observe similar behavior in energy consumption, shown in Figure 6.3b with the highest improvement touching 40% limit.

We were able to achieve so much improvement using ARCS because most of these time-consuming regions have a slight load imbalance and poor cache behavior. However there are applications which may have a very good load balance and cache behavior. In those kind of applications, the improvement will likely not be that significant, because there is very little room for ARCS to work on. In the later part of this section, we will look into such applications as well.

We discussed in previous chapter that the behavior of a region changes across different workloads. To see how efficient ARCS in choosing optimal configurations across workloads, we used ARCS on data set C of SP. Dataset C is four times larger than data set B. Figure 6.4 shows the execution time and energy consumption improvement at TDP (highest power cap). Even in this workload, we achieve execution time improvement of up to 40% and energy consumption improvement of up to 42% using ARCS strategies. It shows that ARCS can find optimal configurations across different workloads. We also observed that the configurations of the regions from SP differed across workloads which also supports the result we found in the previous chapter. To validate ARCS’s consistency across different architectures, we used ARCS on a new architecture, IBM POWER8 (Minotaur). Minotaur differs significantly
Figure 6.2: Feature comparison between the default and **ARCS-Offline** strategy at TDP power level. Comparison is done on four of the most time consuming regions of SP. Y-axis shows the normalized feature value. Smaller value is better.
Figure 6.3: Application level execution time and package energy comparison among the default and ARCS strategies in SP at data set B. Comparison is done on five different power levels. Smaller value is better.

compared to Crill. Even so, when we ran SP with data set B in Minotaur, we observed 37% execution time improvement compared to the default strategy. This result demonstrates ARCS’s versatility across architectures.

Figure 6.4: Execution time and energy consumption comparison of ARCS strategies and the default strategy in data set C of SP. Smaller value is better.

6.3.2 BT

BT is an application with good load balancing and cache behavior. BT is very similar to SP in structure although the approximate factorization is different. Like SP, majority of its execution time is also dependent on four regions (compute rhs, x.solve, y.solve and z.solve). However, the behavior
of these regions is slightly different. Three of these regions \((x_{\text{solve}}, y_{\text{solve}}\) and \(z_{\text{solve}}\)) show very good load balancing and cache behavior in the default configuration. Only \(\text{compute}_{\text{rhs}}\) shows poor scaling, load balancing, and cache behavior. As a result, ARCS has a limited opportunity to improve the performance of this application. \(\text{compute}_{\text{rhs}}\) is the only region where ARCS strategies can have a significant effect, as all other regions already perform very well with the default strategy. In addition, \(\text{compute}_{\text{rhs}}\) is algorithmically hard to optimize due to its long stride memory access. Specifically, the second-order stencil operation in \(\text{rhs}z\) uses the \(K \pm 2, K \pm 1\) and \(K\) elements of the solution array to compute RHS for the \(z\) direction:

\[
\text{RHS}(I, J, K) = A \cdot U(I, J, K - 2) + B \cdot U(I, J, K - 1) + C \cdot U(I, J, K) + D \cdot U(I, J, K + 1) + E \cdot U(I, J, K + 2)
\]

Figure 6.5: Feature comparison between the default and \(\text{ARCS-Offline}\) strategy at TDP power level for \(\text{compute}_{\text{rhs}}\) region of BT. Smaller value is better.

Such memory accesses are not cache friendly, so finding an optimal configuration for such a region is not trivial. However, ARCS does a very good job in finding an optimal configuration \((24, \text{guided}, 1)\) for \(\text{compute}_{\text{rhs}}\) that improves the \(\text{OMP}_{\text{BARRIER}}\) and cache behavior of the region. The comparison between the \(\text{ARCS-Offline}\) and default strategy is shown in Figure 6.5. We compare the cache (L1, L2 and L3 cache miss rate) and load balancing (\(\text{OMP}_{\text{BARRIER}}\) time) behavior. We are only showing the result for \(\text{compute}_{\text{rhs}}\) region, because in other regions the improvement is negligible. For \(\text{compute}_{\text{rhs}}\), the ARCS configuration shows a significant load balancing behavior improvement which is demonstrated by 80% \(\text{OMP}_{\text{BARRIER}}\) time improvement. It also
shows good L3 cache miss rate improvement indicating better cache utilization among different cores.

The impact of these behaviors is also visible in the overall application level execution time and energy consumption comparison in Figure 6.6. Here, we compare the execution time (6.6a) and energy consumption (6.6b) among the default and ARCS strategies (ARCS-Online and ARCS-Offline). We show the results for all five power levels. We observe that the execution time improvement is small across all power levels, with the highest improvement recorded is 13% at 85W power cap with ARCS-Offline strategy. In some cases ARCS actually performs worse than the default strategy (e.g., ARCS-Online at 85W). This is because in those cases small improvement achieved by ARCS is offset by the overhead. Similar behavior is visible for package energy in Figure 6.6b.

We also observed similar trend at Power8 architecture. Only the ARCS-Offline strategy was able to achieve an application level improvement of 18%.

6.3.3 LULESH 2.0

In Figure 6.7 we show the comparison of execution time and energy consumption comparison between the default strategy and ARCS-Online and ARCS-Offline strategies on both Crill and Minotaur. In Minotaur, We achieved a 40% execution time improvement using the ARCS-Offline strategy, while with ARCS-Online we achieved around a 4% improvement.

However, in Crill, the improvement is not evident. With ARCS-Offline strategy, we achieved about 3% execution time improvement in the smallest
(55W) and the highest (115W) power levels. However, we lost performance on other three power levels. We achieved energy consumption improvement in all five power levels with maximum of 26% coming in 85W power level. As for ARCS-Online strategy, we observed a degradation in both execution time and energy consumption for every power levels as compared to default.

To understand why ARCS is performing poorly with LULESH on Crill, we did an extensive analysis. We used TAU [71] for our analysis. We profiled LULESH running with the default configuration at the highest power cap. In Figure 6.8 we show the top five regions based on total time (inclusive time). Through three OMPT events we show how these regions spent their time. These OMPT events are,

- **OpenMP.IMPLICIT.TASK**, it reports the total time spent by an implicit task, in other words it shows the overall execution time of the region.

![Figure 6.7](image)

(a) Execution time (Crill)  
(b) Package energy (Crill)

(c) Execution time (Minotaur)

Figure 6.7: Application level execution time and package energy comparison among the default and ARCS strategies in LULESH, for mesh size 45. It shows results in both architectures. Smaller value is better.
- **OpenMP_LOOP** reports the execution time that is spent only on the loop body.

- **OpenMP_BARRIER/OMP_BARRIER** reports the time spent on the implicit and explicit barriers.

Figure 6.8: OpenMP events data for top 5 time consuming regions from LULESH.

We observe from Figure 6.8 that in terms of **OpenMP.IMPLICIT_TASK** the most time consuming region is **EvalEOSForElems_1**. But most of its time is spent on **OpenMP.BARRIER**. Only a small portion of time is spent on real computation which can be attributed by **OpenMP.Loop** time. The same applies for the **CalcPressureForElems_1** region. Both of these regions have a very small execution time per region call, **EvalEOSForElems_1** with 0.000828 sec and **CalcPressureForElems_1** with 0.000139 sec. And as we explained in the Overhead section, for each region run ARCS has a **Configuration changing overhead** of around 0.0008 sec. For these regions this overhead becomes a huge issue. In fact the overhead becomes almost 100% and 600%. Combined with APEX instrumentation overhead, ARCS looses a significant amount of performance in these tiny regions and in the process adds a fair amount of extra execution time.

As for other three regions in Figure 6.8 although they have reasonable region time (execution time per region call), **CalcKinematicsForElems_1** and **CalcMonotonicQGradientsForElems_1** show near perfect load balancing behavior with only 1.8% and 0.26% of their total execution time spent in **OpenMP.BARRIER**. So there is not much ARCS can do to improve these regions’
performance. However, the CalcFBHourglassForceForElems_1 region shows slightly worse load balancing behavior with 16% of its total execution time spent in OpenMP BARRIER, so ARCS can have some impact on its performance. ARCS was able to do so, which is evident in Figure 6.9. The figure shows OpenMP BARRIER, L1, L2 and L3 cache miss rate comparison between the default and ARCS-Offline strategy on CalcFBHourglassForceForElems_1 region. From the figure we can see that the configuration (4, guided, 32) chosen by the ARCS-Offline strategy is able make the OpenMP_BARRIER time almost zero. It also shows that the configuration also improved the L1 and L3 cache miss rate significantly.

But execution time improvement from just this region was not enough to offset the overhead incurred by those tiny regions in Crill. However, these overheads are not energy hungry computation, that’s why we still achieved overall energy improvement in all power levels.

As for Minotaur, we achieved execution time improvement for the following reason: Minotaur can support up to 160 threads without oversubscribing, which causes a bit more load imbalance in larger regions. As a result, ARCS improvement in those regions overcomes the overhead incurred by the smaller ones, which in turn results in overall application level improvement.

![Figure 6.9: Feature comparison among default and ARCS strategies on CalcFBHourglassForceForElems_1 region.](image)

**6.4 Concluding Remarks**

In this chapter, we presented the ARCS framework that selects the best runtime configurations under different execution environment for OpenMP appli-
cations. Our framework handles a larger configuration search space as compared to prior work. We show that our framework is practical with varying data sets as well as architectures. We tested ARCS using three proxy applications, SP, BT and LULESH. We show that for a given power level, efficient OpenMP runtime parameter selection can improve the execution time and energy consumption of an application up to 40% and 42% respectively.

ARCS can be improved to enable selective tuning for OpenMP regions to avoid overheads on the smaller regions.
Chapter 7

Measuring the Impact of Data Placement in Modern GPUs

In the past few decades there has been an influx in the use of GPUs to harness computational power as well as address the issue of energy efficiency in the field of high performance computing. For example, three of the top five most powerful supercomputers (Summit, Sierra, and Perlmutter) as ranked in June 2021 worldwide use NVIDIA GPUs [45]. However, it is notoriously difficult for developers to obtain optimal performance for applications running on GPUs. One of the major reasons for this is because GPUs have a complex memory hierarchy consisting of different types of memories and caches, such as global memory, shared memory, texture memory, constant memory, L1 cache, and L2 cache (as shown in Fig. 7.1). Each type of memory or cache is associated with its own characteristics such as capacities, latencies, bandwidths, supported data access patterns, and read/write constraints. Where to put application data, or data placement optimization, was an essential yet challenging step to exploit the abundance of GPU threads which has been reported by previous studies [72–74].

Although all NVIDIA GPUs have a similar high-level design, different generations of GPUs introduce new memory properties or implement the same memories using different physical organizations. All these hardware changes may influence the effectiveness of memory optimization, therefore codes optimized for one platform have no guarantee of retaining the same performance on newer platforms. This becomes a problem for maintenance, as targeting optimizations using these different memory types is not a straightforward process. Moreover, optimizations on one platform might slow an application down on another platform. In other words, maintaining performance across systems becomes increasingly difficult.
Figure 7.1: GPU memory hierarchy (from NVIDIA [1])
In this chapter, we present a set of experiments designed to explore the impact of data placement optimization on several generations of NVIDIA GPUs (Kepler, Maxwell, Pascal, and Volta) and different codes, including a set of microbenchmarks, CUDA kernels, and a proxy application. The experiments are configured to include different CUDA thread block sizes, data inputs, and data placement choices. This chapter includes the following contributions:

- We show comparisons of memory properties across several generations of GPUs to highlight their similarities and differences,
- Our work uses a range of microbenchmarks and kernels to explore the impact of data placement across generations of GPUs,
- We analyze performance results and summarize the general trends of using special memories across different generations of GPUs.
- To the best of our knowledge, this work is the first to study the impact of data placement on recent Pascal and Volta GPUs.

The remainder of this chapter is organized as follows. Section 7.1 gives more information about the GPU memory hierarchy and its special memories. Section 7.3 presents our design of a set of experiments to study GPU memory properties and the impact of data placement optimization. Experimental results and analysis are given in Section 7.4. We then conclude our findings in Section 7.5.

### 7.1 GPU Memory Hierarchy

GPUs have a highly complex memory hierarchy in order to exploit their massive parallel computing potentials.

<table>
<thead>
<tr>
<th>Type</th>
<th>Location</th>
<th>Access</th>
<th>Cached</th>
<th>Scope</th>
</tr>
</thead>
<tbody>
<tr>
<td>Global Memory</td>
<td>Off-Chip</td>
<td>Read Write</td>
<td>Y</td>
<td>Global</td>
</tr>
<tr>
<td>Shared Memory</td>
<td>On-Chip</td>
<td>Read Write</td>
<td>N</td>
<td>SM</td>
</tr>
<tr>
<td>Constant Memory</td>
<td>Off-Chip</td>
<td>Read Only</td>
<td>Y</td>
<td>Global</td>
</tr>
<tr>
<td>Texture Memory</td>
<td>Off-Chip</td>
<td>Read Only</td>
<td>Y</td>
<td>Global</td>
</tr>
</tbody>
</table>

Table 7.1: GPU memory types
Table 7.1 gives a high-level overview of the major types of memories which are exposed to programmers via the CUDA API. Below we describe each in more detail:

- **Global Memory**: Also called device memory, this is the largest off-chip memory on a GPU. It also serves as the main memory. Global memory accesses have long latencies and limited memory bandwidth, when compared with accessing on-chip memory or cache.

- **L1 and L2 Caches**: While old Tesla GPUs (CUDA Compute Capability 1.x) did not have caches for global memory, later GPUs from Fermi onward (CUDA Compute Capability 2.x and later) are equipped with a cache hierarchy to improve the performance of global memory accesses. Each Streaming Multiprocessor (SM) has a dedicated L1 cache while all SMs share a single L2 cache.

- **Shared Memory**: This is a software-managed, on-chip data cache for each SM. It has low-latency (similar to a register access) and high-bandwidth. However, the size of shared memory is very limited. It is visible to only active threads within a SM.

- **Constant Memory**: This is implemented as a predefined part of the global memory space which is set to be read-only. It is cached and globally visible to all threads. The latency of constant memory accesses can be as fast as reading from a register if threads of a warp read the same address from cached data. Otherwise it is the same as accessing the device memory.

- **Texture Memory**: This type of memory is similar to constant memory in that it is an off-chip memory space that is cached and read-only. However, texture memory can be as large as the entire global memory space bound to the texture unit. Texture cache is specially optimized for 2D spatial locality, therefore it is best suited to serve threads accessing the memory addresses that are closer to each other in 2D. Every SM has several texture fetch units.

More than just differing by the various kinds of memories available, different generations of GPUs introduce new memory properties or implement the same memories using a different physical structure and organization. Fermi GPUs introduced a true cache hierarchy for global memory while previous GPUs did not have such a design. In Kelper GPUs, L1 cache and shared memory are combined together and texture cache has its on-chip memory. Volta GPUs have merged L1 cache, shared memory and texture cache into a
single unified 128 KB physical memory, while shared memory in Pascal and Maxwell enjoys its own dedicated physical memory. All of these changes to memory design and implementation can directly influence the effectiveness of data placement optimization for GPU applications.

7.2 CUDA

7.3 Design of Experiments

In this section, we discuss the design of our experiments, including the choices of machines, benchmarks and experimental configurations.

7.3.1 GPU Machines

As shown in Table 7.2, we selected four machines for our experiments in order to compare the impact of data placement optimization on different generations of GPUs. In total, they contain four generations, namely Kepler, Maxwell, Pascal and Volta. Two machines are located at Livermore Computing Center of Lawrence Livermore National Laboratory. One machine is provided by the NVIDIA PSG cluster. We have also built a customized Google Cloud machine with a Volta GPU.

<table>
<thead>
<tr>
<th>Name</th>
<th>Location</th>
<th>CPU</th>
<th>Memory</th>
<th>GPU</th>
<th>OS</th>
</tr>
</thead>
<tbody>
<tr>
<td>Surface</td>
<td>Livermore Computing</td>
<td>Intel Xeon E5-2670 @2.60 GHZ</td>
<td>256 GB</td>
<td>K40m</td>
<td>RHEL 7.5</td>
</tr>
<tr>
<td>PSG</td>
<td>Nvidia PSG cluster</td>
<td>Intel Xeon E5-2690 v2 @ 3.00GHz</td>
<td>128 GB</td>
<td>M60</td>
<td>CentOS 7.5</td>
</tr>
<tr>
<td>Ray</td>
<td>Livermore Computing</td>
<td>IBM Power8 @2.2GHz</td>
<td>256 GB</td>
<td>P100-SXM2-16GB</td>
<td>RHEL 7.5</td>
</tr>
<tr>
<td>Custom</td>
<td>Google Cloud</td>
<td>Intel Xeon E5-2699 @2.20GHz</td>
<td>30 GB</td>
<td>V100-SXM2-16GB</td>
<td>Ubuntu 16.04</td>
</tr>
</tbody>
</table>

Table 7.2: Experimental machines with GPUs
Table 7.3 summarizes some of the key specifications of the four types of GPUs. As shown in the table, Kepler K40m has a combined 64KB L1 cache and shared memory. The combined cache can be configured as 48KB and 16 KB (or 32 vs. 32, 16 vs. 48) for L1 and shared memory, respectively. Texture cache is dedicated for Kepler. Maxwell and Pascal GPUs also use a different implementation with separate shared memory but combined L1 cache and texture cache. On Volta, all three caches (L1, texture cache and shared memory) are merged together into a 128 KB unified cache, in which shared memory can take up to 96KB.

7.3.2 Benchmarks

Our goal is to study the impact of data placement optimization within the GPU memory hierarchy, in particular given the choice of utilizing four different types of memories (Constant memory, Shared memory, Texture memory and Global memory). First we attempt to understand the fundamental properties of these GPUs and the impact of using individual memories, using simple CUDA kernels. However, as real applications tend to behave very differently from small kernels, we also test more complex CUDA codes using a mixture of memories. Overall we will showcase four different kinds of benchmarks in our experiments: (1) microbenchmarks measuring memory specification (2) simple CUDA kernels evaluating the impact of using one type of memory (3) CUDA codes assessing more complex use of memories (4) a proxy application with multiple kernels using a mixture of data placement choices.

Microbenchmarks

We surveyed research literature in order to find available microbenchmarks to measure the memory specification for the selected GPUs. GPUmembench used in [75] collects multiple microbenchmarks to measure different memories in previous generations of GPUs. We adopted the microbenchmarks and revised them accordingly to measure the memory bandwidth for the selected GPUs. We also collected the microbenchmark from [76] that revised the traditional pointer-chasing benchmark to use GPU shared memory to store a sequence of data access latencies and eliminate the interference with normal data access.

CUDA kernels

As shown in Table 7.4 we have picked three representative CUDA kernels [77] that are designed to show the advantages of individual special memories (e.g.
Table 7.3: Key specifications of selected GPUs of different generations

<table>
<thead>
<tr>
<th></th>
<th>Kepler (K40)</th>
<th>Maxwell (M60)</th>
<th>Pascal (P100)</th>
<th>Volta (V100)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Computation capability</td>
<td>3.5</td>
<td>5.2</td>
<td>6.0</td>
<td>7.0</td>
</tr>
<tr>
<td>SMs</td>
<td>15</td>
<td>16</td>
<td>56</td>
<td>80</td>
</tr>
<tr>
<td>Cores/SM</td>
<td>192 SP cores/64 DP cores</td>
<td>128 cores</td>
<td>64 cores</td>
<td>64 SP cores/32 DP cores</td>
</tr>
<tr>
<td>Texture Units/SM</td>
<td>16</td>
<td>8</td>
<td>4</td>
<td>4</td>
</tr>
<tr>
<td>Register File Size/SM</td>
<td>256 KB</td>
<td>256 KB</td>
<td>256 KB</td>
<td>256 KB</td>
</tr>
<tr>
<td>L1 Cache/SM</td>
<td>Combined 64K L1+Shared</td>
<td>Combined 24KB</td>
<td>Combined 24 KB</td>
<td>128 KB Unified</td>
</tr>
<tr>
<td>Texture Cache</td>
<td>48KB</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Shared Memory/SM</td>
<td>Combined 64K L1+Shared</td>
<td>96 KB</td>
<td>64 KB</td>
<td></td>
</tr>
<tr>
<td>L2 Cache</td>
<td>1536 KB</td>
<td>2048 KB</td>
<td>4096 KB</td>
<td>6144KB</td>
</tr>
<tr>
<td>Constant Memory</td>
<td>64 KB</td>
<td>64 KB</td>
<td>64 KB</td>
<td>64 KB</td>
</tr>
<tr>
<td>Global Memory</td>
<td>12 GB</td>
<td>8 GB</td>
<td>16 GB</td>
<td>16 GB</td>
</tr>
</tbody>
</table>

constant memory, shared memory and texture memory) compared to the global memory.

(a) Constant memory: The Ray Tracing benchmark is a kernel which shows the benefits of using constant memory. The kernel simulates light reflecting off three-dimensional spheres on a two dimensional image in a three dimensional cube. The computational complexity of this kernel depends on the number of spheres used and the dimensions of the cube. In the constant memory version,
the array representing spheres is put in the constant memory while in global memory version it is put into the global memory. We used three different data sizes for our experimentation: Small, Medium and Large. Details of these data sizes are explained in Table 7.4. We chose these data sizes in such a way that they utilize the constant memory in different proportion (e.g. data size ‘Small’ uses only a fraction of constant memory while data size ‘Large’ uses all 64K available constant memory in the GPU).

(b) *Shared memory:* We present a matrix matrix multiplication kernel (MM) to test the impact of shared memory. The experiments we performed looked at variations of the MM benchmark utilizing global and shared memories. In the shared memory version, blocks of sub-matrices are stored in shared memory and are computed over separately by different threads while in the global version, all values are accessed and stored in global memory. We also used three different data sizes for our experimentation: Small, Medium and Large. Details of these data sizes are explained in Table 7.4.

(c) *Texture memory* To demonstrate the impact of texture memory we use a simple two-dimensional five-point stencil computation kernel simulating heat transferring. The simulation assumes that a rectangular room is divided into a grid. Inside the grid, a handful of ‘heaters’ are randomly scattered with various fixed temperature. We use three different types of grids: Small, Medium and Large. For each grid, the kernel is run 90 iterations (time steps) for each grid. Details of these grids are explained in Table 7.4. Both 1D and 2D texture versions are used to compare their results with respective to global memory versions. However, we only show the results for 2D versions due to space constraint and also the fact that texture memory has more importance in 2D spatial locality.

### Benchmark Applications

To further evaluate the impact of data placement, we selected three benchmarks used in a previous study [73] to re-run: sparse matrix vector multiplication, matrix matrix multiplication and computational fluid dynamics. Sparse Matrix Vector Multiplication (SpMV) is of the form $y = Ax$ and is the mul-

<table>
<thead>
<tr>
<th>Kernel</th>
<th>Memory Focused</th>
<th>Small</th>
<th>Medium</th>
<th>Large</th>
</tr>
</thead>
<tbody>
<tr>
<td>Ray Tracing</td>
<td>Constant Memory</td>
<td>No. of spheres: 200</td>
<td>No. of spheres: 1000</td>
<td>No. of spheres: 2340</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Cube dimension: 2048</td>
<td>Cube dimension: 2048</td>
<td>Cube dimension: 2048</td>
</tr>
<tr>
<td>Matrix Matrix Multiplication</td>
<td>Shared Memory</td>
<td>Matrix size: 512 × 512</td>
<td>Matrix size: 1024 × 1024</td>
<td>Matrix size: 2048 × 2048</td>
</tr>
<tr>
<td>Heat Transfer</td>
<td>Texture Memory</td>
<td>Grid size: 1024 × 1024</td>
<td>Grid size: 4096 × 4096</td>
<td>Grid size: 8192 × 8192</td>
</tr>
</tbody>
</table>
tiplication of a matrix $A$ containing mostly zeros and a vector $x$, which is a commonly found function in scientific codes. Matrix-Matrix Multiplication (MM) is selected again but with more versions using differing combinations of memories. Computational Fluid Dynamics (CFD) represents a simulation of the behavior of fluid dynamic phenomena, which can be used for prediction in a wide range of areas.

The experiments we performed looked at variations of the SpMV benchmark utilizing global, constant, texture and shared memories. This benchmark is loosely based on the public version which can be found in the SHOC benchmark suite\cite{78}. Nine versions were run using a combination of global, texture, shared and constant memory (shown in Table \ref{table:7.5}). This benchmark itself has four arrays (values, columns, vector values and rows) and contains the most broad picture of different memory usage of the three benchmarks we looked at.

For the MM benchmark we ran experiments utilizing global, texture and shared memories. This experiment ran different versions of codes with the two matrices (shown in Table \ref{table:7.6}). There is one version with only global memory and one with only shared memory. The rest of the versions are combinations shared memory and different types of texture memory - including versions using 2D and surface texture memory.

The CFD benchmark, which is loosely based on the same version found in the Rodinia benchmark suite was tested out using global memory and a variety of texture memories\cite{79}. As this benchmark has a large number of arrays (eight), it would be difficult to show a comparison of all possible combinations of these arrays in different memories. Instead, these experiments (unlike those for MM or SpMV) focused exclusively on global and texture memory. There are eight versions of the codes with varying numbers of different arrays being stored in texture memory (shown in Table \ref{table:7.7}), as well as one version using global memory.

**Proxy application**

LULESH \cite{80} is a shock hydrodynamics code which is a part of the DARPA UHPC effort and is now used in DOE’s ExMatEx co-design efforts. LULESH is a hexahedral mesh-based physics code with two centerings: the element centering and the nodal centering. The element centering refers to the center of each hexahedron and stores thermodynamic variables, such as energy and pressure. The nodal centering refers to the corners of hexahedra intersect, and stores kinematics values, such as positions and velocities. The main computation of LULESH happens via time-stepping using a Lagrange leapfrog algorithm followed by a time constraint calculation. We summarize the selected kernels and
Table 7.5: SpMV benchmark memory configurations

<table>
<thead>
<tr>
<th>Version</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>rows in shared</td>
</tr>
<tr>
<td>2</td>
<td>rows in constant</td>
</tr>
<tr>
<td>3</td>
<td>vector in texture1D, rows in shared</td>
</tr>
<tr>
<td>4</td>
<td>matrix values in texture1D</td>
</tr>
<tr>
<td>5</td>
<td>vector in constant, rows in texture1D</td>
</tr>
<tr>
<td>6</td>
<td>vector in texture</td>
</tr>
<tr>
<td>7</td>
<td>matrix values and columns in texture1D, rows in constant</td>
</tr>
<tr>
<td>8</td>
<td>matrix values, columns and vector in texture1D</td>
</tr>
<tr>
<td>9</td>
<td>matrix values, columns, rows and vector in texture1D</td>
</tr>
</tbody>
</table>

Table 7.6: MM Benchmark Memory Configurations

<table>
<thead>
<tr>
<th>Version</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>both matrix A and B in texture2D and shared</td>
</tr>
<tr>
<td>2</td>
<td>matrix A in texture2D and both matrices in shared</td>
</tr>
<tr>
<td>3</td>
<td>matrix B in texture2D and both matrices in shared</td>
</tr>
<tr>
<td>4</td>
<td>matrix B in texture1D and both matrices in shared</td>
</tr>
<tr>
<td>5</td>
<td>both matrix A and B in texture1D and shared</td>
</tr>
<tr>
<td>6</td>
<td>both matrices in shared</td>
</tr>
<tr>
<td>7</td>
<td>matrix A in surface texture and both matrices in shared</td>
</tr>
</tbody>
</table>
Table 7.7: CFD Benchmark Memory Configurations

<table>
<thead>
<tr>
<th>Version</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>neighbors in texture1D</td>
</tr>
<tr>
<td>2</td>
<td>mx, mz, energy in texture1D</td>
</tr>
<tr>
<td>3</td>
<td>mx, my, mz in texture1D</td>
</tr>
<tr>
<td>4</td>
<td>energy, my, mz in texture1D</td>
</tr>
<tr>
<td>5</td>
<td>mx, my, mz, energy, neighbors and normals in texture1D</td>
</tr>
<tr>
<td>6</td>
<td>energy, neighbors and normals in texture1D</td>
</tr>
<tr>
<td>7</td>
<td>mx, my, mz, energy, normals and density in texture1D</td>
</tr>
<tr>
<td>8</td>
<td>my, mz, energy, normals, neighbors and density in texture1D</td>
</tr>
</tbody>
</table>

their array read/write pattern in Table 7.8. The actual data size used in the kernel changes according the problem size. The ranking column shown in the table shows the importance of the selected kernels in terms of the execution time.

We prepared four versions of LULESH. Each version uses only one type of memory for all data in the five kernels. The version using global memory as data storage serves as the baseline version. Other versions using constant memory, shared memory and texture memory are compared to the baseline performance and the speedup between the two is shown. The execution time does not include the data transferring time from other memory locations to the selected type of memory. We also use three data sizes (4, 45, and 90) to explore the impact of varying inputs. For data sizes 45 and 90, constant and shared memory are not big enough to hold all the data so only the global and texture memory versions are evaluated.

Table 7.8: Properties of selected LULESH kernels

<table>
<thead>
<tr>
<th>ID</th>
<th>Kernel</th>
<th>Read-only array count</th>
<th>Write-only array count</th>
<th>Ranking</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>AddNodeForcesFromElems</td>
<td>6</td>
<td>2</td>
<td>4</td>
</tr>
<tr>
<td>2</td>
<td>AddNodeForcesFromElems2</td>
<td>6</td>
<td>2</td>
<td>5</td>
</tr>
<tr>
<td>3</td>
<td>CalcHourglassForceForElems</td>
<td>6</td>
<td>7</td>
<td>2</td>
</tr>
<tr>
<td>4</td>
<td>CalcHourglassControlForElems</td>
<td>13</td>
<td>3</td>
<td>1</td>
</tr>
<tr>
<td>5</td>
<td>CalcMonotonicQRegionForElems</td>
<td>16</td>
<td>1</td>
<td>8</td>
</tr>
</tbody>
</table>
7.3.3 Testing Configurations

We present additional information of the configurations of our experiments. There are several ways to control the available resource utilization in GPUs. The Kepler GPU we use has configurable 64KB caches for both L1 and shared memory. Although there is a CUDA API to control the actual partition sizes between them, we have used the default partition (48 KB shared vs. 16KB L1) on the machine for simplicity. There are also configurations for how to launch a CUDA kernel.

For the CUDA kernels experimentation (Section 7.3.2) we used a predefined number of Thread Block Sizes \( (4 \times 4 = 16, \ 8 \times 8 = 64, \ 16 \times 16 = 256, \ 32 \times 32 = 1024) \). Number of Grids was selected based on the Thread Block Size and Application Data Size (e.g. Application Data Size / Thread Block size). The default number of Warp Size was used. This approach allows us to investigate the performance on the individual SMs (through Thread Block Size) level as well as in the whole GPU level (through Grid Size). Since all the three kernels described in Section 7.3.2 work on 2D data, we used 2D Thread Blocks and Grids for our experimentation. In Table 7.9 we show an example of different configurations that were used assuming the application uses a 2048 × 2048 grid.

Table 7.9: Example thread block and grid configurations

<table>
<thead>
<tr>
<th>Thread Block Size</th>
<th>No. of Grids</th>
</tr>
</thead>
<tbody>
<tr>
<td>4 (4 × 4 = 16)</td>
<td>512 × 512 = 262144</td>
</tr>
<tr>
<td>8 (8 × 8 = 64)</td>
<td>256 × 256 = 65536</td>
</tr>
<tr>
<td>16 (16 × 16 = 256)</td>
<td>128 × 128 = 16384</td>
</tr>
<tr>
<td>32 (32 × 32 = 1024)</td>
<td>64 × 64 = 4096</td>
</tr>
</tbody>
</table>

Our experiments were run using a script to collect average values over ten iterations and the median of these values is then reported in Section 7.4. Each kernel was run five times to warm up the GPU before timings were taken. In order to collect consistent performance times, the performance benchmarks all used cudaEventRecord to denote kernel stop and start places and cudaEventElapsedTime to calculate the time elapsed. cudaDeviceSynchronize was used after these calls and data transfer times are excluded in order to isolate performance differences from memory usage. A value reported as speedup of using a special memory means a ratio relative to the global memory version.
(ie. the version using only global memory - speedup = \( \frac{\text{global memory version}}{\text{new memory configuration}} \)).

7.4 Results and Analysis

7.4.1 Microbenchmarking

All the measured memory specifications of the four GPUs we used are shown in Table 7.10. The change of global memory from GDDR5 to HBM2 has had a big impact on memory performance as the bandwidth has improved 3.5X from the Kepler GPU to the most recent Volta GPU. For all platforms, our measured global memory bandwidth is roughly 80 to 85% of the theoretical bandwidth. We also observe improvement in all memory specifications, including bandwidth and latency, from the earlier generations to the most recent GPUs. For the latest Volta GPU, the highest bandwidth and lowest latency are measured here. Figure 7.2 compares the bandwidth of these different GPUs. It is noticeable that L1 bandwidth of Volta is significantly better than previous generations. Another interesting point is that the bandwidth of shared memory of Maxwell is smaller than its predecessor (Kelper).

<table>
<thead>
<tr>
<th>Memory /Cache</th>
<th>Properties</th>
<th>Kepler (K40)</th>
<th>Maxwell (M60)</th>
<th>Pascal (P100)</th>
<th>Volta (V100)</th>
</tr>
</thead>
<tbody>
<tr>
<td>L1 data cache</td>
<td>size (KB)</td>
<td>16 ˜48</td>
<td>24</td>
<td>24</td>
<td>32 ˜128</td>
</tr>
<tr>
<td></td>
<td>hit latency</td>
<td>35</td>
<td>82</td>
<td>82</td>
<td>28</td>
</tr>
<tr>
<td></td>
<td>line size</td>
<td>128B</td>
<td>32B</td>
<td>32B</td>
<td>32B</td>
</tr>
<tr>
<td></td>
<td>bandwidth (GB/s)</td>
<td>602</td>
<td>1103</td>
<td>2379</td>
<td>13414</td>
</tr>
<tr>
<td>L2 data cache</td>
<td>size (KB)</td>
<td>1536</td>
<td>2048</td>
<td>4096</td>
<td>6144</td>
</tr>
<tr>
<td></td>
<td>line size (B)</td>
<td>32</td>
<td>32</td>
<td>32</td>
<td>64</td>
</tr>
<tr>
<td></td>
<td>hit latency</td>
<td>˜200</td>
<td>˜207</td>
<td>˜234</td>
<td>˜193</td>
</tr>
<tr>
<td></td>
<td>bandwidth (GB/s)</td>
<td>154</td>
<td>488</td>
<td>1579</td>
<td>2629</td>
</tr>
<tr>
<td>-------------------------</td>
<td>------------------</td>
<td>-------</td>
<td>-------</td>
<td>-------</td>
<td>-------</td>
</tr>
<tr>
<td><strong>Shared memory</strong></td>
<td>size per SM (KB)</td>
<td>48</td>
<td>96</td>
<td>64</td>
<td>up to 96</td>
</tr>
<tr>
<td>Banks</td>
<td>32</td>
<td>32</td>
<td>32</td>
<td>32</td>
<td></td>
</tr>
<tr>
<td>Theoretical bandwidth</td>
<td>2912</td>
<td>2410</td>
<td>9519</td>
<td>13800</td>
<td></td>
</tr>
<tr>
<td>Measured bandwidth</td>
<td>2540</td>
<td>1213</td>
<td>7681</td>
<td>10057</td>
<td></td>
</tr>
<tr>
<td><strong>Constant memory</strong></td>
<td>Measured bandwidth (GB/s)</td>
<td>665</td>
<td>817</td>
<td>1776</td>
<td>2470</td>
</tr>
<tr>
<td><strong>Texture memory</strong></td>
<td>Measured bandwidth (GB/s)</td>
<td>619</td>
<td>1068</td>
<td>2378</td>
<td>3547</td>
</tr>
<tr>
<td><strong>Global memory</strong></td>
<td>Memory bus</td>
<td>GDDR5</td>
<td>GDDR5</td>
<td>HBM2</td>
<td>HBM2</td>
</tr>
<tr>
<td>Max clock rate (MHz)</td>
<td>2505</td>
<td>2505</td>
<td>715</td>
<td>877</td>
<td></td>
</tr>
<tr>
<td>Theoretical bandwidth</td>
<td>240</td>
<td>160</td>
<td>732</td>
<td>900</td>
<td></td>
</tr>
<tr>
<td>Measured bandwidth</td>
<td>191</td>
<td>127</td>
<td>510</td>
<td>750</td>
<td></td>
</tr>
</tbody>
</table>

### 7.4.2 CUDA Kernels

In this section we analyze the experimental results related to the three CUDA kernels. We investigate how the impact of a specific memory has changed over the generations of GPUs. We primarily use two metrics to explain our results:
Speedup and Execution time. Speedup is calculated with respect to the default configuration. This default configuration means using global memory and a Thread Block Size of 32. The base speedup is one, denoted by a red dotted line in the graphs. Anything over one is performance improvement compared to default configuration while anything lower is performance degradation.

### Constant Memory

Figure 7.3 shows the speedup using constant memory across different GPUs. The x-axis represents the GPU Thread Block Size used while the y-axis represents the speedup. Each block represents a certain generation of GPU. Figures 7.3a, 7.3b and 7.3c show results for Small, Medium and Large data sizes respectively.

This section of our experiments can be divided into three different parts. First, the impact of constant memory across different generation of GPUs is investigated. Secondly, we analyze the impact of data size. Lastly, we discuss the importance of Thread Block Sizes.

We observe first that the impact of constant memory has started to diminish in newer generations of GPUs since Maxwell. In fact, with Volta, usage of constant memory degrades the performance in all different data sizes. This phenomenon is evident as the speedup using constant memory on Volta is below one in all three data sets. We find high percentage of stalling due to pipeline busy for the constant memory version based on results of the Nvidia profiler. With Pascal we see a similar phenomenon with larger data sets.
is primarily due to the architectural change in the GPUs. Improvement in global memory bandwidth through HBM2 in Pascal and Volta has improved the global memory access which is accompanied by the improvement in L2 and L1 cache performance.

![Speedup in Small dataset](image1.png)

**(a) Speedup in Small dataset**

![Speedup in Medium dataset](image2.png)

**(b) Speedup in Medium dataset**

![Speedup in Large dataset](image3.png)

**(c) Speedup in Large dataset**

Figure 7.3: Speedup achieved using constant memory compared to global memory

However, this is not to say that the performance of constant memory has not improved across GPU generations. In Figure 7.4 we compare the execution time across different generations of GPUs using constant memory. We show the *medium* data set size which uses the GPU thread block size of 32. From the Figure, steady execution time improvement can be seen across generations of GPUs. The key take-away from these results is that even though constant memory performance has improved over time, new breakthroughs in global memory along with L1 and L2 cache improvement have come a long way to diminish the impact of dedicated constant memory, at least for some
applications. As shown in Figure 7.2, the significantly enhanced L1 cache on Volta has 5X times higher bandwidth than constant memory does.

![Execution Time Graph](image)

Figure 7.4: Execution time using Constant memory

We observe that with the increase in data size, global memory becomes a more viable option compared to constant memory. We see this result probably due to the bandwidth utilization of constant memory. With the increase in data size, utilization of constant memory increases, hence constant memory bandwidth starts to become a bottleneck which in turn degrades performance.

We observe that for this application, speedup (performance) improves with the number of Thread Block Size. We believe, this is primarily due to the improvement in SM utilization. We observe a similar trend across different generation of GPUs and different data sizes.

### Shared Memory

In Figure 7.5 we present the speedup achieved using shared memory compared to the default configuration. The x-axis represents the GPU Thread Block Size used and the y-axis represents the speedup. Each block represents a certain generation of GPU. We also show the results for three different data sizes. Figure 7.5a, 7.5b and 7.5c show results for Small, Medium and Large data sizes respectively.

From this figure we see that with the newer generation of GPUs the influence of shared memory is consistently diminishing except for Pascal. We understand this is happening due to the architectural differences in shared memory, L1 cache and texture memory units. Kepler has a configurable shared memory that is shared with the L1 cache while Maxwell has a dedicated shared memory unit. As for Pascal, the memory structure is similar to Maxwell, while in Volta shared memory, L1 cache and texture memory share the same unit.
We see speedup going down from Kepler to Maxwell primarily due to the decrease in shared memory bandwidth. Although in Maxwell there is a dedicated shared memory unit, the application we used never exhausts the available shared memory limit. As a result, we don’t see the impact of dedicated shared memory.

However, we do see a dramatic speedup using shared memory on Pascal, primarily due to its use of HBM2 and high bandwidth of shared memory compared to previous GPUs. One possible reason for this is the fact that shared memory and the L1 cache have different units in Pascal, hence different latencies. While the latency for the L1 cache does not change from Maxwell to Pascal \([7.1]\), the shared memory latency seems to have improved from Maxwell to Pascal \([7.6]\). This also explains the diminishing speedup in Volta, as in Volta shared memory and L1 cache are in the same unit, hence their hit latency is also the same.

We also compare the actual performance of different generation of GPUs in Figure \([7.6]\). For a representative configuration using the Medium data size and GPU Thread Block Size of 32, we observe steady improvement across the generations of GPUs, except for Maxwell. This is probably due to low bandwidth of the shared memory unit in Maxwell.

### Texture Memory

Figure \([7.7]\) shows the results of the 2D texture memory version of the Heat transfer kernel’s speedup compared to the default global memory version. The 1D version results are very similar and are therefore omitted here.

We observe that the texture memory slightly outperforms the global memory in Kepler when the thread block size is 16. But with the newer generations of GPUs global memory starts to outperform texture memory, especially with the larger data sizes. This is primarily due to the architectural change in texture memory unit. Among the experimental GPUs, only Kepler has a separate texture cache unit while all other GPUs have the texture cache unit shared with L1.

As for the performance of the texture memory unit across different GPUs, we observe a similar trend as constant memory for the 2D \([7.8]\) texture version. We see a steady improvement in the newer generation of GPUs.

### 7.4.3 Benchmark Applications

This section describes our results from comparing the three benchmark applications run across the four different platforms with variations of memory
Figure 7.5: Speedup achieved using shared memory compared to global memory

Figure 7.6: Execution time using Shared memory
Figure 7.7: Speedup achieved using 2D texture memory compared to global memory

(a) Speedup with Small dataset
(b) Speedup with Medium dataset
(c) Speedup with Large dataset

Figure 7.8: Execution time using 2D texture memory

123
conﬁgurations. Tables 7.5, 7.6 and 7.7 have short descriptions of the versions of memory conﬁgurations found in these ﬁgures.

Figures 7.9, 7.10 and 7.11 shows these separate applications exhibiting very different performance measurements for particular memory types across a range of hardware for two different benchmarks. In Figure 7.9 it can be seen that choosing different memory for SpMV types generally has some improvement with texture and combinations of texture and shared memories, but as the GPU architectures evolve from Kepler to Volta, the improvement seems to become decrease and even become detrimental. However, Figure 7.10 paints a somewhat different picture. Here it can be seen that the different data placements can still be relevant as the GPUs evolve, even on the Volta architecture. Finally, in Figure 7.11 we see an example where the usage of different texture memories seems to have very little effect overall on the benchmark, no matter which generation of GPU.

Figure 7.9: Speedup of SpMV

Figure 7.9 shows that for some applications, there can be a wide range of performance differences between utilizing different memory versions across these selected platforms. This indicates how important it is for some applications to re-evaluate optimizations that were made for a speciﬁc architecture. In particular, version 9 shows a greater than 1.5X speedup on Kepler, but a 20% slowdown on Volta. Yet, alternatively the shared and constant versions maintain relatively similar performance across the platforms.

Figure 7.10 shows that for some applications, performance gains can still be made even on the newer platforms. On Volta, there is still up to a 1.2X improvement using version 5, which utilizes texture1D memory for both matrices A and B. The shared memory version (6) also shows marked improvement across all platforms - almost 4X on Pascal and retaining 1.5X on Volta. How-
ever, it should be noted that this particular version (and its corresponding
global version) are slightly different from the other versions in their use of a
different data object for the arrays. This may explain why the speedup is so
pronounced - because the original version uses a non-optimal data object to
begin with.

Figure 7.11 shows that for some applications, performance will stay rela-
tively consistent across generations of GPUs even when utilizing different
memory types. For the CFD benchmark, at best there is an 8% speedup on
the Kepler architecture. At worst, on Maxwell, there is a 14% slowdown.
However, on the newest architecture Volta, the worst slowdown is about 10%.
For this application it is clearly still important to get the data placement right,
however it is less detrimental than some of the losses clearly shown for SpMV
Of all the memories looked at, although texture was investigated in the most detail, it also seems to be the one retaining the most improvement. On Volta, using texture memory can still result in improved performance for the MM and CFD (marginally) benchmarks. Shared memory also seems to show improvement for some applications - the MM benchmark retained better performance across all versions and all platforms than others using shared and texture memory in combination. Overall it appears that using different memory optimizations on newer platforms overall does not produce as great of speedup as it did on earlier platforms. However, these applications were all run with the same data size and block size, so the results should be considered only part of an overall picture.

7.4.4 Proxy Applications

We performed experiments on four generations of GPUs using four data placements for five selected kernels in the LULESH proxy application. Each variant was tested with three different data sizes: 4, 45 and 90. For data size 4, data is small enough to be placed in all four different memory locations. Data can be placed only in global memory or texture for data size 45 and 90.

Figure 7.12 shows the speedups comparing the baseline performance using global memory with performances using constant memory, shared memory and texture memory. The corresponding kernel names for Figures 7.12 and 7.13 can be found in Table 7.8. Results shown in Figure 7.12 use data size 4. The red line is the reference for speedup of one. No performance improvement is observed for data placement using constant memory and texture memory. Using shared memory in the CalcMonotonicQRegionForElems kernel delivers 20% improvement in speedup on the Kepler GPU, marginal speedup on the Maxwell, and no speedup for Pascal and Volta GPUs. Experiments with data size 4 will use a small number of GPU threads. This leads to low achieved GPU occupancy and under used GPU resources. The GPU thread number used in the experiment might not be sufficient to hide the memory latency.

Experiments using data size 45 and 90 are shown in figure 7.13a and figure 7.13b respectively. With larger data sizes, the performance factor that GPU resources are under-utilized can be eliminated. Meanwhile however, we can only evaluate texture memory placement compared to the default global memory placement.

CalcHourglassControlForElems_kernel and CalcFBHourglassForceForElems_kernel gain performance improvements with the texture memory placement. However speedups compared to the global memory placement shrink in the newer generations of GPU. Both figures shows less benefit us-
ing texture memory placement in the newer generations of GPU. With more advanced memory specification used for global memory in Pascal and Volta GPUs, reading data in the global memory and caching through L1 and L2 cache shows better outcome than reading through the texture memory cache.

The experiment with LULESH proxy application shows only one kernel has a potential performance benefit from using the shared memory in early generations of GPU. The experiment did not exploit cache blocking transformations for the shared memory, however. This could be a more optimal way for shared memory data placement, but it would increase the complexity in programming and tuning. No performance gain is observed with the constant memory placement. However, the bottleneck is low capacity of constant memory and the low GPU occupancy in the experiments. Constant memory is good for read-only data that needs to be repetitively broadcast to all GPU threads. Our implementation with constant memory might not be practical in real applications for the following reasons: (1) low capacity to host data (2) limited read-only data (3) data is allocated at global scope and flexibility to change

Figure 7.12: LULESH kernels using size 4
the data during runtime. We see more benefit using texture memory placement in the early generations of GPU. With the advanced memory adopted as global memory in Pascal and Volta GPUs, reading data from global memory and caching through L1 and L2 caches show higher performance than reading data through the texture cache. Texture memory is designed for its specialized purpose to read 2D or 3D data access patterns. Data stored in texture memory is read-only and would also limit its applicability in real applications. The extra code changes required to exploit texture memory additionally bring in more programming burden to the application developers.

7.5 Concluding Remarks

Previous studies have shown that data placement optimization, i.e. optimizing where to put data into different memories within a GPU, is essential for obtaining optimal performance on older generations of GPUs, such as Fermi and Kelper. In this chapter, we presented the work that focuses on designing a set of experiments to explore the relevance of data placement optimization on newer generations of NVIDIA GPUs. Our experiments include a range of CUDA kernels running on four generations of GPUs including Kepler, Maxwell, Pascal and Volta.

Our key findings in this work include: 1) All types of memories on newer GPUs have improved performance compared to previous generations. 2) The unified cache design of Volta GPUs helps narrow the performance gap between the default global memory and all other special memories. In particular, constant memory and texture memory seemed to be much less important
on recent GPUs. 3) Of those applications that benefited, texture and shared memory showed the most promise of speedup gain through considered data placement optimization. 4) The memory properties of special memories significantly limit their use in real applications. Constant memory and texture memory are read-only; constant memory and shared memory have small capacity. Also significant programming efforts are required for using these special memories, especially for shared memory. Automated code transformation is needed to exploit different types of GPU memory.
Chapter 8

Related Work

This work combines research from few related areas such as Static code analysis, Compiler assisted fault tolerance, OpenMP Performance management, and GPUs’ performance maintainability. In this chapter, we try to provide a brief overview of the state of the art of those areas.

8.1 OpenSHMEMChecker

This work has roots in three overlapping research areas:

8.1.1 Program Correctness of Parallel Programs

A variety of methods have been explored to detect and remove bugs from parallel applications, including runtime error-verification, trace-based error detection, model checking and static analysis. Each has its strengths and weaknesses.

Runtime approaches usually utilize instrumented code to check for anomalies in the runtime behavior. Such systems include UPC-Check [81], MPI-Check [82], MARMOT [37, 83], UMPIRE [38]. While these techniques are specially good for detecting runtime bugs such as race conditions, deadlock, erroneous arguments, they add over-head in the runtime. Also they focus on a specific part of the program and are agnostic regarding the program control structure. As a result, finding the source of the error may be difficult, since the position where the error is detected and the source of the error may not be the same. In contrast, our work uses static analysis techniques.

Trace-based error detection tools such as BoundsChecker [? ], the Intel Message Checker [81] analyzes trace files to detect errors such as mis-matched buffer types, race conditions and deadlocks. Trace-based techniques work well
with 2-sided communication as in MPI (matching send-recv pair), but de-
tecting errors in 1-sided communication models such OpenSHMEM can be
extremely difficult.

Another error detection technique is model checking, which uses formal
methods to check the validity of a program. The user typically models the in-
put/output, logically represents the program as a finite state model, and makes
assertions for different states using a modeling language: MPI-SPIN [85], UPC-
SPIN [86], MAGIC [87], SLAM [88] use this technique.

8.1.2 Static Analysis Techniques for Program
Correctness

Static analysis is a popular method to check for program correctness. It uses
static program information (e.g., compile-time information) and symbolic ex-
ecution to find errors in a program. It usually exploits existing compiler in-
formation. Droste et al. [39], Ye et al. [89], Yu et al. [90] use LLVM’s Static
Analyzer while Aananthakrishnan et al. [91] use ROSE compiler framework.
However, these mostly focus on MPI.

8.1.3 OpenSHMEM Program Analysis

Work in the area of program analysis and error detection for OpenSHMEM
is scarce. The OpenSHMEM Analyzer [92, 93] is the only prior work in this
domain. Like our work, it uses compiler-based static analysis to detect bugs
in OpenSHMEM applications. However, the OpenSHMEM Analyzer is built
on top of OpenUH (a branch of the Open64 compiler) which is no longer
supported. Our checker is based on the popular LLVM framework, which has
a modular infrastructure that makes our work easily extensible.

8.2 Compiler Assisted Fault Tolerance

This work can trace its roots back to three overlapping research areas,

• Checkpoint-Restart Techniques
• Fault Tolerance in PGAS Programming Models
• Compiler and Tool Based Fault Tolerance
8.2.1 Checkpoint-Restart Techniques

Research on Checkpoint-Restart Techniques for Fault tolerance can be traced back to work performed as early as the 1960s by David Jasper [94]. Over the years, different checkpointing schemes such as uncoordinated, coordinated, communication-induced, incremental and multi-level checkpointing have been developed to meet the need of ever-changing computing environments and applications. Most of these schemes have been implemented to be used at the system level, at the application level, or both. In recent times many of these techniques have been developed or adapted for use in conjunction with Message-Passing Systems and Programming Models (e.g., MPI) as a result of their popularity in High Performance Computing. Surveys by Elnozahy et al. [5] and Dongara et al. [95] are excellent resources for information on these techniques.

8.2.2 Fault Tolerance in PGAS Programming Models

PGAS programming models such as OpenSHMEM are becoming popular in the HPC community due to their programmability, utilization of modern hardware, and performance affinity for applications with irregular communication patterns. However, consideration of fault tolerance for PGAS programming models is still scarce. Besta et al. [32] was one of the first to develop a generic model for reasoning about resilience in applications that use Remote Memory Access (RMA) and to introduce schemes for in-memory checkpointing and logging based protocols for them. Among other notable works that focus on specific PGAS programming models other than OpenSHMEM are [96] and [97]. In [96], Ellis et al. introduced a coordinated checkpointing protocol for UPC applications while in [97], Shahzad et al. developed tools to support fault tolerance in GASPI applications. The tools included a Health Check Library and a Fault-aware C/R library which in combination provide fault detection, propagation, and communication recovery.

Hao et al. [44, 98] was one of the first works to explore fault tolerance in the context of OpenSHMEM. Here the authors proposed an application-level checkpointing mechanism based on User Level Fault Mitigation (ULFM) where the shared global memory (symmetric memory) regions are replicated (backed up) across peer processes. However, it was up to the programmer to make sure that any data that may be necessary for restart is allocated in the symmetric memory. Since it was an application-level C/R scheme, the user code was responsible for managing the checkpoint and restart operations. Our work is designed to help application developers navigate the process of managing C/R operations and use schemes like this efficiently.
Garg et al. [99] introduced a different approach that utilized a system-level transparent checkpointing scheme for achieving OpenSHMEM fault tolerance. In their approach, they saved the checkpoints in stable storage which allowed them to save the computation to be used at a later time or on a different cluster. Despite these efforts, OpenSHMEM lacks an error model to provide proper error detection, propagation, and recovery. To address this issue, Bouteiller et al. [100] proposed such an error model as an extension to the OpenSHMEM API to solve these issues. However, so far this has not resulted in any modification to the specification. Compared to those works, ours is the first work to define and identify safe points for checkpointing in OpenSHMEM.

8.2.3 Compiler and Tool Based Fault Tolerance

Although application-level C/R can be the most effective C/R technique for overhead efficiency, doing it in a large-scale application can introduce huge implementation effort. Hence, researchers have looked into compilers and tools to ease this process via checkpointing suggestions or automatic checkpointing to recover from both soft [101] and hard failures [6–12].

Most of these works are based on source to source compilers and primarily focus on MPI programs. Porch [7] utilized a source to source compiler and user inputs (checkpointing routines and frequency) to insert checkpoint operations in a sequential C program. Bronevetsky et al. [8, 9], Yang et al. [10], and Rodriguez et al. [11] used source to source compilers to automatically insert checkpoints in MPI applications. In contrast to these efforts, our work focuses on OpenSHMEM, a PGAS programming model which introduces the added complexity of global consistency issues and utilizes an open source general-purpose compiler, LLVM. As a result, we are able to benefit from the on-going innovations in this rapidly evolving compiler infrastructure and plan to exploit the analyses already available in the LLVM infrastructure for future extensions of our tool. Rodriguez et al. [12] also provide an LLVM-based implementation that builds on top of their previous work [11], however it still targets MPI applications.

Another notable difference of our work from other research is that we do not perform automatic checkpointing, rather we focus on providing user feedback and facilitating checkpointing by the application developer. This is primarily due to the lack of checkpointing libraries in OpenSHMEM. Due to this scarcity, a user may have to modify or use application-based checkpointing libraries developed for other programming models (e.g., MPI). Hence, in this work we focus on helping the application developer use whatever library they choose. In the future, we plan to extend this work to do automatic checkpointing as well.
8.3 Managing OpenMP Performance through Runtime Adaptation

Bull et al. [102] is one of the first work which provides an insight into the impact of the number of threads, scheduling policy and synchronization on an OpenMP application’s performance. Suleman et al. [103] proposes a framework to dynamically control the number of threads using run-time information. However, none of these works consider power budget or energy consumption in their analysis. Their only consideration was execution time. But in our work, we consider both execution time and energy consumption under a power budget.

As the number of threads and processor frequency have an enormous impact on performance and energy consumption of a given OpenMP application, many researchers have studied energy efficient performance prediction models for parallel applications. The work by Curtis-Maury et al. [104, 105] falls under this category. They employ dynamic voltage and frequency scaling (DVFS), dynamic concurrency throttling (DCT) and simultaneous multi threading (SMT) to implement various online and offline configuration selection strategies for OpenMP applications. They concentrate on decreasing energy consumption without degrading execution time. However, the work does not consider power budget. Peter Baily et al. [106] implements an adaptive configuration selection scheme for both homogeneous and heterogeneous power constrained systems. It considers only two parameters, the number of threads and processor frequency. Dong Li et al. [107, 108] uses DVFS and DCT to select energy efficient configuration for threads and operating frequency for MPI/OpenMP hybrid applications. They do not consider a power budget. Their main target is to save energy without the loss of execution time. The work by Wei et al. [109] shows the impact of optimal operating frequency on energy consumption improvement for parallel loops. It uses different operating frequency across different loops using frequency modulation techniques. Nandamuri et al. [110] characterizes performance and energy consumption of OpenMP parallel regions using OpenMP Runtime API. Lima et al. [111] compares the impact of different runtime systems to leverage energy efficiency for task based applications. Dong et al. [112] focuses on harnessing energy efficiency through loop scheduling and processor shutdown technique. In contrast, our work concentrates on a complete set of runtime parameters on a strict power constrained system.

Nowadays, power has become a limiting factor for large-scale HPC centers. As a result, work on overprovisioned systems with strict power budget is gaining popularity in the HPC community. Work by Rountree et al. [113] is one
of the first to explore the impact of power capping. They investigate how different power levels impact the performance of different types of applications. Work by Patki et al. [114] explores the impact of hardware overprovisioning on a system level performance. The main contribution of their work is to select the number of nodes, the number of cores per node, and power cap per node. Work by Aniruddha et al. [115] and Bailey et al. [116] consider only two parameters, DVFS, and the number of threads, as configuration options. They focus on overall system level performance on an MPI/OpenMP hybrid application. Compared to these works, our work focuses on single node performance improvement at the OpenMP parallel region level using OpenMP runtime parameters.

8.4 GPU Performance Management: Impact of Data Placement on Modern GPUs

Previous studies have explored various data placement strategies on early generations of GPUs. For example, PORPLE [73, 117] is a portable GPU data placement approach combining a memory specification, a compiler framework and a run-time library. A lightweight performance model based on reuse distance is used by PORPLE to guide run-time selection of optimal data placement policies. Its effectiveness has been evaluated on Tesla, Fermi, and Kepler GPUs. Huang and Li [74] have proposed a new performance modeling approach for optimal data placement on GPUs. They have analyzed correlations among different data placements and used a sample data placement to predict performance for other data placements. Jang et al. [72] have presented several rules based on data access patterns to guide the memory selection for a Tesla GPU. Yang et al. [118] proposed compiler-based approach to generate kernel variants for exploiting memory characteristics. We believe our work is the first to study the impact of data placement on more recent generations of GPUs such as Volta.

Researchers have developed several microbenchmark suites to understand various aspects of memory characteristics of different generations of GPUs. For example, GPUBench [119] is one of the early benchmark suites designed to analyze the performance of GPUs. Volkov et al. [120] measured hardware characteristics, including texture caches, of 8800GTX GPU relevant to optimizing dense linear algebra. Fang et al. [121] made a benchmark of a GPU memory system to quantify the capability of parallel accessing and broadcasting. Wong et al. [122] have used microbenchmarking to measure the CUDA-visible architectural characteristics of the early generation Tesla GPU (GT200). Ba-
belStream \cite{123} measures memory transfer rates of global device memory on
GPUs. It has multiple implementations using various programming models,
such as CUDA, OpenCL, OpenMP, OpenACC, RAJA, and Kokkos. gpumem-
bench \cite{75} contains a set of micro-benchmarks to measure bandwidth of on-
chip special memories of GPUs. Mei et al. \cite{76} have proposed a new fine-
grained microbenchmarking approach to expose unknown cache parameters
of three generations of GPUs, including Fermi, Kelper, and Maxwell. More
recently, Jia et al. \cite{124} have used microbenchmarking to analyze the Volta
GPU architecture. In comparison, our work use more comprehensive CUDA
kernels and applications to study the impact of data placement choices among
multiple different memory types across different generations of GPUs.
Chapter 9

Conclusion

Resilience is fast becoming one of the most critical design factors for HPC applications as the Mean Time Between Failure (MTBF) of large HPC systems is becoming smaller and smaller. However, writing an error-free resilient application requires effort and comes with performance overheads which led researchers to explore different avenues for efficient resilience techniques. One of the most popular resilience techniques used in HPC is Checkpoint-Restart (C/R) due to its low overhead and portability. However, adopting such a technique and deploying it successfully requires a lot of effort from the application developers. Therefore, the supporting tools infrastructure that can take a lot of burdens away from the application developers is of utmost importance if we are to deploy a C/R technique successfully. The key areas where the infrastructure is highly needed are: programming model aware error-detection; assistance in the checkpointing process by identifying where to put the checkpoint calls and what data checkpoint; and maintaining the performance and energy consumption after a restart where the execution environment of some or all nodes may change compared to before.

In this work, we have developed strategies and tools to address those aspects of OpenSHMEM-OpenMP hybrid applications, which are becoming popular due to their compatibility with applications that have asynchronous communication pattern, such as graph applications. We show that our work makes it easier for an application developer to write correct, resilient, and performant applications while also providing support to adapt and maintain performance across different execution environments after a restart from a failure.

Clearly, much needs to be done to provide comprehensive resilience support for OpenSHMEM-OpenMP applications based on C/R. Based on the findings of this work, we discuss some of the next steps that may be taken to improve the resilience support of OpenSHMEM-OpenMP hybrid apps. However, this
is in no way an exhaustive list. We believe the findings of this work will open the door for other research related to resilience.

9.1 Application-level C/R library

In this work, we assumed the existence of a C/R library. However, to provide full resilience support, an optimized C/R library for OpenSHMEM is essential. We are exploring the possibility of using the existing C/R library developed for other programming models (e.g., MPI) with OpenSHMEM. We are also working to develop an application-level checkpointing library for OpenSHMEM that would enable application developers to exploit the results of our work directly.

9.2 Checkpoint data optimization

In this work, we used live variable analysis to optimize the amount of data to be checkpointed. However, additional techniques would further reduce the amount of checkpointed data. In particular, applying "memory exclusion" techniques that are based on memory read/write operations in OpenSHMEM programs may help further optimize the amount of data to be checkpointed.

9.3 Safe point to checkpoint

Not all safe points are suitable for use as checkpoints. In this work, we left the decision of choosing which safe points should be used as checkpoints to the user. However, further work is needed to suggest suitable positions for checkpointing based on program analysis and the detected safe points.

9.4 Maintaining energy efficiency after restart

In this work, our main focus was to utilize the OpenMP runtime parameters to maintain the performance of a node after a restart with a secondary focus on energy efficiency. However, there may be scenarios where energy efficiency is of more importance than execution time. In those cases, we may want a better energy-efficient configuration with a slight execution time penalty.
Bibliography


[21] Open Source Software Solutions (OSSS) OpenSHMEM Implementation on top of OpenUCX (UCX) and PMIx. [https://github.com/openshmemborg/osss-ucx](https://github.com/openshmemborg/osss-ucx)

[22] MVAPICH2-X. [http://mvapich.cse.ohio-state.edu/](http://mvapich.cse.ohio-state.edu/)


[58] Alexandre E Eichenberger, John Mellor-Crummey, Martin Schulz, Michael Wong, Nawal Copty, Robert Dietrich, Xu Liu, Eugene Loh,


[75] Elias Konstantinidis and Yiannis Cotronis. A quantitative performance evaluation of fast on-chip memories of GPUs. In *Parallel, Distributed,


[97] Building and utilizing fault tolerance support tools for the gaspi applications.


