UC Irvine
UC Irvine Electronic Theses and Dissertations

Title
A Multiple Compiler Approach for Improved Performance and Efficiency

Permalink
https://escholarship.org/uc/item/3c00m7d6

Author
Shivam, Aniket

Publication Date
2021

Peer reviewed|Thesis/dissertation

eScholarship.org Powered by the California Digital Library
University of California

UNIVERSITY OF CALIFORNIA,
IRVINE

A Multiple Compiler Approach for Improved Performance and Efficiency

DISSERTATION

submitted in partial satisfaction of the requirements
for the degree of

DOCTOR OF PHILOSOPHY

in Computer Science

by

Aniket Shivam

Dissertation Committee:
Professor Alexander V. Veidenbaum, Chair
Professor Alexandru Nicolau
Professor Tony Givargis

2021

Portion of Chapter 4 © Springer Nature Switzerland AG 2019
All other materials © 2021 Aniket Shivam

DEDICATION

To my parents.

TABLE OF CONTENTS

Page

LIST OF FIGURES **vi**

LIST OF TABLES **viii**

ACKNOWLEDGMENTS **ix**

VITA **x**

ABSTRACT OF THE DISSERTATION **xii**

1 Introduction **1**
- 1.1 Loop Nest Optimizations 2
- 1.2 A Synergistic Compilation Approach 4
- 1.3 *Learning* about the Impact of Optimizations on Performance 5
- 1.4 Using Performance-Oriented Optimizations to Achieve Energy Efficiency . . 6
- 1.5 Tool for Compiler Researchers 7
- 1.6 Contributions 8

2 MCompiler - A Synergistic Compilation Framework **9**
- 2.1 Loop Nest Optimizations 9
- 2.2 Compilation and Optimization Frameworks 10
- 2.3 Motivation 14
  - 2.3.1 Example 1: Intel's `icc` performs the best 15
  - 2.3.2 Example 2: GNU's `gcc` performs the best 16
  - 2.3.3 Example 3: LLVM `clang` performs the best 16
- 2.4 Overall Framework Architecture 17
- 2.5 Loop Extraction Phase 19
- 2.6 Optimization Phase 23
- 2.7 Exploratory Search Phase 24
- 2.8 Synthesis Phase 24
- 2.9 Using and Expanding the Framework 25

3 Evaluation of the Multiple Compiler Approach for Improved Performance **27**
- 3.1 Benchmarks, Code Optimizers and Target Architecture 27
- 3.2 Comparing all Code Optimizers with the *MCompiler* 29

3.3 *MCompiler* with Exploratory Search . . . 31
3.3.1 Serial Code . . . 32
3.3.2 Auto-Parallelized Code . . . 34
3.3.3 OpenMP Code . . . 36
3.3.4 Analysis of Results . . . 36
3.4 Summary . . . 37

4 Predicting the Best Code Optimizer for the Loop Nests 39
4.1 Towards an Achievable Performance for Loop Nests . . . 39
4.2 Experimental Methodology and Training the Machine Learning Models . . . 42
4.2.1 Collecting Hardware Performance Counters using Profiling . . . 43
4.2.2 Most Suited Code Optimizer . . . 43
4.2.3 Random Decision Forest Classifier . . . 45
4.2.4 Machine Learning Model Configuration . . . 46
4.2.5 Benchmarks . . . 47
4.2.6 Experimental Platforms and Data Collection . . . 48
4.3 Evaluation of the Machine Learning Models . . . 49
4.3.1 Predicting the Most Suited Code Optimizer for Serial Code . . . 50
4.3.2 Predicting the Most Suited Code Optimizer for Auto-Parallelized Code 51
4.3.3 Overall Analysis and Discussion . . . 52
4.4 An Explanation for why Hardware Performance Counters are good ML Features 53
4.5 A Framework for Improving Performance using Machine Learning Predictions 57
4.5.1 Collecting Hardware Performance Counters for the Loop Nests . . . 59
4.6 Evaluation of the *MCompiler* with Machine Learning Prediction . . . 60
4.6.1 Serial Code . . . 62
4.6.2 Auto-Parallelized Code . . . 63
4.7 Summary . . . 63

5 Applying the Multiple Compiler Approach to Improve Energy Efficiency 65
5.1 Optimizing for Energy Efficiency on Modern Architectures . . . 65
5.2 Impact of Performance-Oriented Loop Nest Optimizations on Energy Efficiency 67
5.3 Evaluation of Different Compilers in Terms of Energy Efficiency . . . 71
5.3.1 Loop Nests Optimized by Different Compilers . . . 72
5.3.2 Reduction in EDP when Selecting the Most Energy Efficient Version 73
5.4 Performance and Energy Consumption Implications of using Different Vector Extensions . . . 76
5.4.1 Compiler's Ability to Auto-Vectorize and Impact of Selecting the Best Vector Length . . . 76
5.4.2 Impact on Performance and Energy Consumption when Increasing the Number of Active Cores . . . 78
5.5 A Framework for Improving Energy Efficiency . . . 80
5.6 Evaluation of the *MCompiler* for Improving Energy Efficiency . . . 82
5.6.1 Serial Code . . . 85
5.6.2 Auto-Parallelized Code . . . 85
5.7 Summary . . . 86

6 Prior Art **87**

7 Conclusions and Future Directions **91**

Bibliography **95**

LIST OF FIGURES

Page

2.1 *MCompiler* Framework 18
2.2 *MCompiler* Command Line Options 25

3.1 Performance of individual Code Optimizers vs *MCompiler* on TSVC benchmark (top 50 loop nests). A value of 1 indicates the same performance as the *MCompiler*, less than 1 means a slower performance than the *MCompiler*. . 29
3.2 *MCompiler* Speedup for Serial Benchmarks 32
3.3 *MCompiler* Speedup for Auto-Parallelized Benchmarks 33
3.4 Distribution of best performing code per Code Optimizer. Breakdowns per benchmarks suite showcase benefits of specialized code optimizers. 34
3.5 *MCompiler* Speedup for OpenMP Benchmarks 35

4.1 Speedup of Predictions for Serial Code 49
4.2 Confusion Matrix for Serial Code Predictions 50
4.3 Distribution of Predictions for Serial Code 51
4.4 Speedup and Confusion Matrix of Predictions for Auto-Parallelized Code . . 52
4.5 Distribution of Predictions for Auto-Parallelized Code 52
4.6 *MCompiler* Framework with Machine Learning Predictions 57
4.7 *MCompiler* + ML Predictions Performance for Serial and Auto-Parallelized Benchmarks 62

5.1 EDP comparison between different compilers w.r.t. ICC for TSVC. Y-axis is log scaled. Improvement in EDP means the factor of reduction in EDP compared to the EDP of ICC generated code. 70
5.2 Comparison of CPU Energy and Speedup between different Clang and ICC for TSVC loop nests. Only cases with more than 10% EDP improvement are shown. 71
5.3 Comparison of CPU Energy and Speedup between different GCC and ICC for TSVC loop nests. Only cases with more than 10% EDP improvement are shown. 72
5.4 EDP comparison between different compilers w.r.t. ICC for Polybench. Y-axis is log scaled. 73
5.5 Comparison of CPU Energy, DRAM Energy and Speedup between Clang and ICC for Polybench loop nests. 74

5.6 Comparison of CPU Energy, DRAM Energy and Speedup between GCC and ICC for Polybench loop nests. 75
5.7 Comparison of CPU Energy, DRAM Energy and Speedup between Polly and ICC for Polybench loop nests. Y-axis is log scaled. 76
5.8 EDP comparison between different vector length w.r.t. ICC -Ofast for TSVC. Y-axis is log scaled. Only cases that showed more than 10% EDP improvement are shown. 77
5.9 Variation in Average Runtime, Total Dynamic Energy Consumption and EDP when running multiple copies of TSVC with different Vectorized versions. . . 79
5.10 *MCompiler* Framework for Improving Energy Efficiency 81
5.11 *MCompiler* EDP Improvement for Serial Benchmarks 82
5.12 *MCompiler* EDP Improvement for Auto-Parallelized Benchmarks 83
5.13 Distribution of most energy efficient code per Code Optimizer. Breakdowns per benchmarks suite showcase benefits of specialized code optimizers. 84

LIST OF TABLES

Page

2.1 Sequence of LLVM's Loop Transformation Passes at the highest optimization setting. 11
2.2 Compilers and Domain Specific Optimizers integrated in the *MCompiler* Framework. 23

4.1 Candidate Code Optimizers used for the ML Experiments. 44
4.2 Top Ranking ML Features for Serial Code Predictions. 55
4.3 Top Ranking ML Features for Auto-Parallelized Code Predictions. 56

5.1 Change in Maximum Core Frequency with different Vector Extensions for a sixteen-core Intel® Xeon® Skylake Gold 6142 processor. 69

ACKNOWLEDGMENTS

I would like to thank my advisor Professor Alex Veidenbaum for his mentorship and for many interesting discussions we had over the last few years. I would also like to thank Professor Alex Nicolau for his guidance and support. Also, thanks to Professor Tony Givargis for serving on the Advancement Committee and the Defense Committee.

I feel fortunate that I get to work with some great researchers and engineers over the course of my graduate studies. I have high appreciation for Rosario Cammarota for being a mentor and a friend, and also motivating me during the first few crucial years of my graduate studies. I learned a lot from Michael Wolfe (NVIDIA) about being a good researcher and an even better engineer. Seeing his passion for work, the ability to share his knowledge and his career advice for me have been really helpful. Another source of inspiration has been David Kuck (Intel) and seeing his continuous passion for research and a great career spanning more than five decades. I would like to thank David Wong (Intel) for guidance over the course of the internship. Finally, I would like to thank my mentors and managers at the Portland Group (now NVIDIA HPC SDK) at NVIDIA, the Fast Kernels team at NVIDIA, the Performance Tools group at Intel and the team at Advanced Processor Lab, Samsung Research America for giving me valuable experience as part of my internships.

I consider myself fortunate to meet some really good people and forge some great friendships over the last few years. Juan Besa, Hirak Kashyap and Neftali Watkinson have been wonderful friends who enriched my life with their support, kindness and knowledge. Additionally, I would like to thank my lab colleagues over the years for their support, collaboration and valuable feedback.

Last but not least, I have great appreciation for my family and friends who gave me strength throughout the course of my studies.

VITA

Aniket Shivam

EDUCATION

Doctor of Philosophy in Computer Science **2021**
University of California, Irvine *Irvine, CA*

Master of Science in Computer Science **2016**
University of California, Irvine *Irvine, CA*

Bachelor of Technology in Computer Science and Engineering **2014**
National Institute of Technology, Uttarakhand *Uttarakhand, India*

WORK EXPERIENCE

Deep Learning Performance Library Intern **June–Sept 2020**
NVIDIA Corporation *Santa Clara, CA*

Performance Tools Pathfinding Intern **June–Sept 2019**
Intel Corporation *Austin, TX*

Compiler Engineer (PGI OpenACC/GPGPU team) Intern **June–Sept 2017**
NVIDIA Corporation *Hillsboro, OR*

Compiler Engineer (PGI OpenACC/GPGPU team) Intern **July–Sept 2016**
NVIDIA Corporation *Hillsboro, OR*

Compiler Engineer (GPU Compiler team) Intern **June–Sept 2015**
Samsung Research America (SRA) *Mountain View, CA*

Teaching Assistant **2015–2021**
University of California, Irvine *Irvine, CA*

SELECTED PUBLICATIONS

A Multiple Compiler Framework for Improved Performance **2021**
In submission.

Using Performance-Oriented Loop Nest Optimizations to achieve Energy Efficiency **2021**
In submission.

OpenACC Routine Directive Propagation using Interprocedural Analysis **2018**
Workshop on Accelerator Programming Using Directives (WACCPD), SC

Towards an Achievable Performance for the Loop Nests **2018**
Languages and Compilers for Parallel Computing (LCPC)

Load Balancing with Polygonal Partitions **2018**
International Workshop on Polyhedral Compilation Techniques (IMPACT), HiPEAC

Using Hardware Counters To Predict Vectorization **2017**
Languages and Compilers for Parallel Computing (LCPC)

Polygonal Iteration Space Partitioning **2016**
Languages and Compilers for Parallel Computing (LCPC)

SOFTWARE

MCompiler
A Synergistic Compilation Framework

ABSTRACT OF THE DISSERTATION

A Multiple Compiler Approach for Improved Performance and Efficiency

By

Aniket Shivam

Doctor of Philosophy in Computer Science

University of California, Irvine, 2021

Professor Alexander V. Veidenbaum, Chair

Production compilers have achieved a high level of maturity in terms of generating efficient code. Compilers are embedded with numerous code optimization techniques, with special focus on loop nest optimizations, that have been developed over the last four decades. The code generated by any two production compilers can turn out to be very different based on pros and cons of their respective Intermediate Representation (IR), implemented loop transformations and their ordering, cost models used and even instruction selection (such as vector instructions) and scheduling. The compilers also need to predict the behavior of a multi-core processor which has complex pipelines, multiple functional units, complex memory hierarchy, etc. on the overall performance. Hence, the performance of produced code for a program segment by a given compiler may not necessarily be matched by other compilers. Additionally, there is no way of knowing how close a compiler gets to optimal performance or if there is any headroom for improvement.

The complexity and rigidity of the compilation process makes it very difficult to modify a given compiler to improve the performance of generated code for every case where it couldn't produce the best possible code. Therefore, this thesis presents a compilation approach that turns the differences between compilation processes and performance optimizations in each compiler from a weakness to a strength. This approach is implemented as a novel compila-

tion framework, the *MCompiler*. This meta-compilation framework allows different segments of a program to be compiled using an ensemble of compilers/optimizers and combined into a single executable. Utilizing the highest performing code for each segment, identified via Exploratory Search, can lead to a significant overall improvement in performance. The framework is shown to produce performance improvements for serial (including auto-vectorized code), auto-parallelized and hand-optimized (using OpenMP) parallel code.

Next, this thesis explores the possibility of learning which compiler will produce the best code for a segment. This is accomplished using Machine Learning. The Machine Learning models learn about inherent characteristics of loop nests and then predict which code optimizer is the most suited for each loop nest in an application. These Machine Learning models are then incorporated into the *MCompiler* to predict the best code optimizer, during compilation, for each code segment of the application. This feature allows the *MCompiler* to replace the expensive Exploratory Search with Machine Learning predictions and still keep performance very close to the Exploratory Search.

Finally, this thesis expands the compilation approach to achieve energy efficiency on modern architectures. Prior research has advocated both for and against the hypothesis that optimizing for performance translates into optimizing for energy efficiency. No production compiler optimizes for energy efficiency directly, expecting optimizing for performance to translate into higher energy efficiency. Optimizing for performance is complex for recent generations of processors and, with automatic DVFS management in these processors, optimizing for energy efficiency would add another level of complexity for compilers with no guarantee of success. Using the *MCompiler*, this thesis shows how the performance-oriented compiler optimizations can be used to achieve energy efficiency.

Chapter 1

Introduction

State-of-the-art compilers optimize applications for better performance on target architectures. The developers and the users of applications trust the compilers to be able to generate the most efficient code for the architecture of choice. Specially in the domain of High-Performance Computing, and recently in other domains such as Computer Vision and Deep Learning, a lot of attention is paid to getting the best out of target architecture. Application developers are responsible to come up with efficient algorithms and source code. Once this step is done, writing hand-optimized code or libraries for a target system is a possible option, but the complexities are such that even these approaches cannot guarantee the absolute best results. In addition to that, this requires massive efforts from developers to understand and optimize for each applicable system and later to port these applications to newer architectures and systems. Hence, more often than not, applications are written using the best known algorithm and language features, but the rest is left to the compilers to optimize for different architectures and systems.

Compiler optimizations are essential to reach an achievable performance for many applications. The means to reach the goal of producing high performance code may, and in most

cases, do differ between any two production compilers. Each may have their own flavor of Intermediate Representation (IR), vary in implemented loop transformations and their ordering, and even differ in instruction selection (such as vector instructions) and scheduling. Each compiler uses a specific, ordered set of optimization techniques and different profitability models and can, therefore, generate code significantly different from other compilers. For program segments, such as loop nests, the performance of generated code from a compiler may either turn out to be better or worse compared to other compilers. And given the complexity of the entire compilation/optimization process it is very difficult to modify a given compiler so as to produce the best performing code for every case where it couldn't match a different compiler. Discrepancies in performance between compilers are not merely engineering shortcomings that can be fixed in the next update. They are the unavoidable result of many NP-Hard or NP-Complete problems encountered in the compilation/optimization process[80, 129]. Compilers try to approximately solve NP-Hard problems efficiently and effectively by using profitability models that are based on many assumptions. Compiler writers try to find the optimal solutions, based on experimentation, that work well for a large portion of target applications for their compiler, but not all. Therefore, it is quite apparent why different compilers produce different results for a given program segment. This calls for a strategy to harness the strengths of multiple compilers, while substituting the weakness of individual compilers. Hence, this thesis presents a compilation approach that will provide both the users of compilers and compiler writers a means to find best possible solution for their target applications.

1.1 Loop Nest Optimizations

Optimizing loop nests, in particular, contributes significantly towards achieving better performance. State-of-the-art architectures have multiple cores on a chip, where each core has

Single Instruction Multiple Data (SIMD), or vector, capabilities. These architectural features provide opportunities for a compiler to expose parallelism in applications on multiple levels, but with a caveat of additional complexity in the decision making for the compiler. The code optimization techniques to *auto-vectorize* the loop nests [106, 4, 146], so as to generate SIMD instructions, require careful analysis of data dependences, memory access patterns, etc. Several auto-parallelization techniques [105, 90, 8, 92, 91, 93, 88, 21, 39, 100] and directive based parallel programming models, such as OpenMP [104], have been developed to take advantage of multiple cores. In fact, most auto-parallelization implementations in modern compilers, which take serial code as input, generate OpenMP code [68, 111, 113].

Key loop transformation techniques [15, 12, 146, 143, 16, 77] include Distribution, Fusion, Interchange, Skewing, Tiling and Unrolling. Code optimizers apply a semantic-preserving sequence of transformations to generate a better performing code, either serial or parallel. But evaluating if a sequence of transformations is optimal is NP-Hard and the search for the best sequence of transformations and their profitability is guided by heuristics and/or approximate analytical models. Thus, a code optimizer may end up with a sub-optimal result and different code optimizers may, for the same source code segment, generate code with significant performance differences on the same architecture.

A major challenge in developing the heuristics and profitability models is predicting the behavior of a multi-core processor which has complex pipelines, multiple functional units, complex memory hierarchy, hardware data prefetching, etc. Parallelization of loop nests involves further challenges for the code optimizers, since communication costs based on the temporal and spatial data locality among iterations have an impact on the overall performance. Evaluation studies [102, 134, 94, 54] have shown that state-of-the-art code optimizers may miss out on opportunities to auto-vectorize and auto-parallelize the loop nests for modern architectures. For optimizing applications written in `C`, there are several compilers and domain specific loop optimizers that perform auto-vectorization and, in some cases, auto-

parallelization of code. From a given code optimizer's point of view, the sequence it used is the best it could do but there is no way of knowing how close it gets to optimal performance or if there is any headroom for improvement.

1.2 A Synergistic Compilation Approach

In this thesis a compiler framework, called the *MCompiler*, is presented and its design is discussed in Chapter 2. The design allows each loop nest in the application to be optimized by the best optimizer available for it. The *MCompiler* identifies loop nests in `C` applications, optimizes the loop nests using different code optimizers, times each optimized code version in execution of its complete application, and links the best performing code to generate the complete application binary. This is referred to as the Exploratory Search method of the *MCompiler*. The *MCompiler* currently incorporates code optimizers from Intel's C compiler, GNU GCC and LLVM Clang. In addition to these, two Polyhedral Model based loop optimizers, Polly [56, 113] and Pluto [22, 111] are used, if applicable. The best loop nest code selection allows the *MCompiler* to produce higher-performing code than the best of the code optimizers in the framework. The *MCompiler* benefits from the entire compilation process (loop transformations and optimizations, and code generation) implemented in each of the code optimizers. The framework allows for easy integration of newer versions and newer configurations of the available code optimizers as well as the addition of new code optimizers.

The framework can be used to optimize applications, first, for serial execution with auto-vectorization of loop nests. This optimizes loop nests for SIMD or vector code generation, in addition to optimizing loop nests for data locality, memory hierarchy, etc. Second, the framework can also target multi-core processors, by taking serial loop nest code as input and auto-parallelizing those loop nests using the available code optimizers to generate multi-

threaded code. Auto-parallelized code is also optimized for SIMD execution within each thread. In this case, the original loop nests are transformed such that loop iterations can be reordered and scheduled for parallel execution across the multiple cores. Third, the framework can target OpenMP applications, i.e., applications with OpenMP directives inserted across sections of the code meant for parallel execution.

The framework extracts loop nests from the applications' source files into separate source files as a function, together with any additional information needed. It then replaces loop nests with a function call in the original source files. This allows for separate code optimizers to focus on just the loop nests and also allows the framework to insert the best performing code, i.e., linking object files to generate the executable.

Chapter 3 presents a study of the potential of the proposed approach. It optimizes each extracted loop nest, or a consecutive set of loop nests, separately with all available code optimization *candidates*. The performance of each optimized loop nest is measured as part of the complete application execution. The best performing code for a loop nest is selected for linking into the final executable. This step, referred to as Exploratory Search, shows that the framework can indeed improve the resulting code's performance.

1.3 *Learning* about the Impact of Optimizations on Performance

The benefits of the proposed approach comes from the fact that different optimizers try to solve a certain problem differently, i.e., optimizing a certain type of loop nest(s) in a unique manner. Loop nests have different inherent characteristics based on memory access patterns, types and counts of operations, presence of branches, etc. This leads to a research question: can we get an insight into these inherent characteristics of the loop nest? If so, can this

learning be useful in predicting which optimizer will produce the best code for a loop nest?

The thesis presents Machine Learning (ML) based techniques to learn such inherent characteristics and predict the most suited code optimizer for a given loop nest in Chapter 4. The approach used in this thesis relies on the hardware performance counters collected for a loop nest to learn about its inherent characteristics. Hardware performance counters capture intricate details about data movement across levels of caches, memory footprint, and count and types of instructions retired that determine the performance of loop nests on an architecture. These hardware performance counters are used as features/input to the ML algorithms to predict the most suited code optimizer for the loop nests.

These ML models are then incorporated into the *MCompiler* and provides a substitute for the expensive Exploratory Search step of the framework. The hardware performance counters are collected from a single profile of the applications, i.e., the applications are compiled with just one code optimizer and then executed ones. However, as with any prediction, it can lead to a potential performance loss compared to search-based selection due prediction errors, i.e., when the ML model or classifier does not choose the best code optimizer. The results show that by using well-trained ML models this potential loss in performance can be quite small.

1.4 Using Performance-Oriented Optimizations to Achieve Energy Efficiency

Energy efficiency is a major issue for domains from embedded systems to Exascale computing [71]. The goals for optimizing applications, in terms of energy or power, differ from domain to domain and may even differ from user to user. Energy consumption for different processors (even from the same architecture) is driven by dynamic parameters, such as Dynamic Voltage Frequency Scaling (DVFS). These parameters cannot be modeled while doing

static compilation, hence implementing energy efficiency driven compiler optimization may provide no guaranteed results. In general, production compilers optimize for performance, with various optimization levels with increasing aggressiveness towards generating better performance, but no such optimization levels for better energy efficiency, and understandably so.

Prior works [44, 149, 73, 75, 109, 137, 76, 151, 152, 53, 120, 138, 107, 55, 89, 52, 70] have explored the impact of compilers and their optimizations on performance and energy consumption for the CPU and memory, some focusing on loop nest transformations. However, Chapter 5 of this thesis shows the overall impact of the sequence of loop nest transformations implemented in *several* compilers on energy efficiency.

Then, the Exploratory Search method of the *MCompiler* is expanded to optimize applications for better energy efficiency by choosing the most energy efficient version possible for each loop nest. The results show that optimizations oriented towards performance improvement may not have the same impact on energy consumption improvement and these differences vary from loop nest to loop nest. Also, by using the Exploratory Search method, the *MCompiler* is able to measure the impact of dynamic parameters before generating the optimized binary.

1.5 Tool for Compiler Researchers

The *MCompiler* framework can also serve as an important tool for compiler researchers who regularly implement and test their optimization techniques and/or tweak analytical or heuristic models to improve performance and/or efficiency for applications. The framework design allows for adding new code optimizers and monitoring their performance on entire application or just on particular hotspots.

The framework also allows for training new Machine Learning models and using them for

making predictions. Various flags are available for choosing the target architecture and choosing particular optimizations such as auto-parallelization optimizations or enabling particular passes such as data prefetching pass. The framework also allows for running Hardware Counter Collector independently, i.e., collect hardware performance counters for all the hotspots in an application while disabling the ML predictions.

1.6 Contributions

Overall, this thesis makes the following contributions:

- It presents a meta-compilation framework that improves performance for `C` applications for serial as well as parallel execution, including OpenMP applications.
- It shows that using the framework can achieve better performance over state-of-the-art compilers.
- It demonstrates that prediction for the most suited code optimizer (serial as well as parallel) for a loop nest can be accurately made using Machine Learning classifiers.
- It explores the impact of performance-oriented optimizations on energy efficiency and then uses the framework to generate energy efficient version for applications.
- It provides an open source framework for researchers and compiler developers to analyze and compare their code optimization techniques.

Chapter 2

MCompiler - A Synergistic Compilation Framework

This chapter primarily presents the design of the framework that implements the multiple compiler approach for improving performance. The chapter starts with an overview of the prior art in the field of compilation and optimization frameworks. Then, it presents interesting cases that justify the need for a multiple compiler approach. Finally, the details of the *MCompiler* framework are presented and its benefits and flexibility are explained.

2.1 Loop Nest Optimizations

In the field of compilers, loop nest optimizations have been a focus for decades. The reason being that the majority of the application execution time is spent in executing a set of instructions repeatedly, i.e., the loop nests. Improvements in performance may come from various avenues. These may include Loop Nest (or Iteration Space) Transformations [82, 106, 4, 45, 46, 15, 12, 146, 143, 16, 77, 117] such as Distribution, Fusion, In-

terchange, Skewing, Tiling [144, 145] and Unrolling. These transformations are performed at the Intermediate Representation (IR) level. They try to optimize the code for data locality and memory management. The order of these transformations and the correctness of the profitability models for directing these transformations have significant impact on the performance. Next step includes taking advantage of the architecture specific components such as Vector/SIMD units and multi-core processors (with multi-level caches and shared memory). This is accomplished using auto-vectorization techniques [105, 106, 4, 3, 146, 83, 77, 94], so as to generate SIMD instructions, that require careful analysis of data dependences [118, 16], memory access patterns, etc. For improving performance for multi-core processors several auto-parallelization techniques [105, 90, 95, 6, 8, 59, 92, 91, 148, 74, 93, 88, 22, 21, 39, 100] have been proposed. But due to the complexity of the applications and their source code, compilers are not able to model the definite behavior of the loop nests/hotspots during compilation. For such cases, hand-optimized code using directive based parallel programming models, such as OpenMP [104], aid compilers in generating the high-performance machine code and are allow for code portability. Other available options for hand-optimizing code that are restrictive to specific architectures and compilers are using assembly-coded functions [66] and specialized libraries [67].

2.2 Compilation and Optimization Frameworks

To solve the NP-Hard or NP-Complete problems encountered during the optimization process, several compilation frameworks have been proposed. These include compilation frameworks for program analysis and transformation such as `LLVM` [84], similar to GNU GCC and Intel C/C++ compilers. `LLVM` uses a static sequence of loop transformations for optimizing code for performance. Table 2.1 shows the sequence of loop transformations at the highest optimization setting (`-O3`). Few of these transformations are optional and need to be

switched on explicitly by the users. Many other compiler optimization passes are executed in and around these loop transformations.

Loop Transformations	**Include Profitability Models**	**Optional Pass**	**Link-Time Optimization Pass**
Rotate Loops			
Loop Invariant Code Motion			
Unswitch Loops			
Loop Flatten		Y	Y
Recognize Loop Idioms			
Delete Dead Loops			Y
Loop Interchange	Y	Y	Y
Unroll Loops	Y		Y
Loop Invariant Code Motion			
Reroll Loops		Y	
Loop Versioning LICM		Y	
Loop Invariant Code Motion			
Rotate Loops			
Loop Distribution			Y
Loop Vectorization	Y		Y
Loop Load Elimination			
Loop Invariant Code Motion	Y		
Unswitch Loop			
SLP Vectorizer	Y		
Loop Unroll-And-Jam	Y	Y	
Unroll Loops	Y		Y
Loop Invariant Code Motion			
Loop Sink			
Loop Fusion		Y	

Table 2.1: Sequence of `LLVM`'s Loop Transformation Passes at the highest optimization setting.

These order of transformations and their profitability models differ from one compiler to the other. Compiler writers try to find the optimal solutions that work well for a large portion of target applications for their compiler. That is why, different compilers produce different results for a loop nest and hence, the entire application.

Several classes of code optimizers have been proposed in the past. `Polaris` [20] operated

on Fortran 77 programs and it's IR is Fortran-oriented. `SUIF` [59] was a source-to-source parallelizing compiler framework. `PIPS` [78] is a tool to implement and evaluate various interprocedural compilation, parallelization, analysis and optimization techniques. `Cetus` [88] is a source-to-source parallelizing compiler for `C`. Whereas, `ROSE` [119] provide tools for static analysis, program optimization, arbitrary program transformation, domain-specific optimizations, complex loop optimizations, performance analysis, and cyber-security analysis.

There are several domain-specific compilation frameworks that have been proposed and show significant performance improvement over traditional compilers and techniques. One such domain is Polyhedral Model based optimizations [7, 47, 48]. Compilation frameworks built on the concepts of the Polyhedral Model are Pluto [22, 111], Polly [56, 113], PoCC [115, 116, 112], CHiLL [33, 133], AlphaZ [150] and Tiramisu [13].

Recently, compilation frameworks for Deep Learning and Machine Learning, such as TensorFlow XLA [1] and Apache TVM [35], and for Tensor Algebra, such as TACO [79], have been shown to improve performance of applications in their respective domains. MLIR [85] is an extensible compiler infrastructure (based on `LLVM`) that aims to address software fragmentation, improve compilation for heterogeneous hardware, significantly reduce the cost of building domain specific compilers.

Several programming models, such as OpenMP [104], aim to provide better performance by allowing users to explicitly expose parallelism in the applications. For example, `OmpSs` [49] extends OpenMP with new directives to support asynchronous parallelism and heterogeneity. Whereas, Tapir/LLVM [124] compiles and optimizes `Cilk` programs to allow for efficient parallel execution on shared-memory multicore machines.

Compilers provide several options and configurations that let users explore various configurations that may suit their application the best, rather than the default setting embedded into the compilers. In order to take advantage of these options and configurations, several it-

erative compilation [136, 5, 2, 115, 116, 36] and auto-tuning frameworks have been proposed in the past that explore, search or predict good combinations of compiler flags to improve performance. MilepostGCC [51] presents an auto-tuning framework that explores `GCC` and its flags, and uses ML techniques to predict good combinations of compiler flags. Another similar work, the OpenTuner framework [9], searches for the best performing compiler flag combinations. There are few options [81, 132] available to control transformation orders at finer granularity but require user assistance.

Production compilers also provide option for Interprocedural optimizations (IPO) [38, 24, 60], sometimes also known as Whole Program Optimizations (WPO) or Link-Time Optimizations (LTO), and Profile-guided Optimization (PGO) [34, 108]. IPO and LTO passes perform several optimizations based on whole program analysis or interprocedural analysis and may re-address loop nests' optimization based on the new information. In case of LLVM, few loop nest transformations are repeated in the Link-Time Optimization (LTO) phase, as mentioned in Table 2.1, after analyzing the entire program and may help optimize the code further. Whereas, PGO, for example, in the Intel compilers improves performance by reducing code size, reducing branch mispredictions and reorganizing code layout to reduce instruction-cache problems. Also, determine profitability of loop nests with small iteration counts [66].

Why a synergistic compilation framework could be helpful?

To take advantage of these several compilers and code optimizers, a unified and extensible compilation framework that can incorporate these tools is one approach. The *MCompiler* framework presented in this chapter targets different classes of compilers, code optimizers and programming models in a single framework. *MCompiler* starts with using ROSE to transform the application source code in a desired manner so as to facilitate an infrastructure that can combine optimized code from several optimizers in one single executable. Next, it applies traditional compilers such as Intel's `icc`, GNU's `gcc` and LLVM `clang` to optimize

loop nests. It also applies two domain specific optimizers: Polly and Pluto (wherever applicable). Polly is an extension to LLVM optimizer, that transforms LLVM IR to Polyhedral representation to do transformations, and finally generates machine code using LLVM code generator. Whereas, Pluto is a source-to-source optimizer that lowers the loop nests in a Polyhedral representation and uses Polyhedral Model based code generators to generate optimized `C` code. Using these compilers/optimizers, the *MCompiler* can generate serial (with SIMD/vector code) code and auto-parallelized code. Finally, *MCompiler* can also optimize hand-parallelized programs written using OpenMP programming model and generate multi-threaded programs. But, *MCompiler* can be extended to other domain specific optimizers and auto-tuning frameworks.

2.3 Motivation

Let us start with three motivating examples that highlight the fact that different compilers do perform different loop nest optimizations at highest optimization levels. This leads to a difference in performance on the same architecture. The presented framework exploits this fact to improve performance of applications. These examples also highlight that performance improvements can be attributed to complex loop transformation techniques. In these cases, even phase-ordering or changing flag combinations on a compiler may not help. The performance improvement comes from specific loop transformation and auto-vectorization techniques. The following three examples compare and analyze auto-vectorized code from three compilers that produce the best performance in terms of execution time. Their performance is compared on Intel Xeon Scalable Gold Skylake processor with AVX-512 vector extensions. The three compilers are Intel's `icc`, GNU's `gcc` and LLVM `clang`. The three loop nests are taken from Test Suite for Vectorizing Compilers (TSVC) by Callahan et al. [25] and Maleki et al. [94].

2.3.1 Example 1: Intel's icc performs the best

Listing 2.1: Example 1

```
1  for (int nl = 0; nl < 100*(ntimes/LEN2); nl++) {   // Loop 1
2      for (int i = 1; i < LEN2; i++) {               // Loop 2
3          for (int j = 1; j < LEN2; j++) {           // Loop 3
4              aa[j][i] = aa[j-1][i] + cc[j][i];
5          }
6          for (int j = 1; j < LEN2; j++) {           // Loop 4
7              bb[i][j] = bb[i-1][j] + cc[i][j];
8          }
9      }
10 }
```

Here `icc` performs 14.5x better than `gcc` and 12.9x better than `clang`. Listing 2.1 is loop nest `s2233` from the TSVC benchmark. `icc` distributes `Loop 2`, over `Loop 3` and `Loop 4`, into two separate loop nests. The first loop nest, `Loop Nest 1`, then consists of `Loop 1` { `Loop 2` { `Loop 3` } }. Second loop nest, `Loop Nest 2`, is `Loop 1` { `Loop 2` { `Loop 4` } }. Next, in `Loop Nest 1`, `Loop 2` and `Loop 3` were interchanged to provide sequential memory accesses. In `Loop Nest 2`, the memory accesses are sequential already inside innermost loop. Therefore, innermost loops in both loop nests were auto-vectorized. Finally, `Loop 2` and `Loop 4` were completely unrolled as they were left with only 15 iteration after vectorization. Interestingly, `icc` chose to use AVX and AVX-2 registers only and skip AVX-512 based on the cost model. `gcc` on the other hand, did not distribute or interchange loops. It simply vectorized for AVX-512 and then completely unrolled `Loop 4` only. Lastly, `clang` followed `gcc`'s transformation and added one more transformation on `Loop 3`, i.e., unroll by factor of 5. This one last transformation from `clang` improved code performance over `gcc`. This example clearly shows why different compilers perform so differently. The application and order of loop transformations are primarily responsible for this contrast. These transformation and their

order may also be tightly embedded and fixed in most compilers, therefore they can't be controlled by flags or hints.

2.3.2 Example 2: GNU's gcc performs the best

Listing 2.2: Example 2

```
1 for (int nl = 0; nl < 3*ntimes; nl++) {
2     for (int i = LEN - 2; i >= 0; i--) {
3         a[i+1] = a[i] + b[i];
4     }
5 }
```

In this example, `gcc` performs 6.1x better than `icc` and 5.3x better than `clang`. Listing 2.2 is loop nest `s112` from the TSVC benchmark. `gcc` vectorized the innermost loop with the factor of 16, array `a` is type `float`, therefore using AVX-512 registers. Followed by unrolling the innermost loop by a factor of 16. The innermost loop can indeed be vectorized since the only dependence present is an anti-dependence or write-after-read on `a[i]`. On the other hand, `icc` decided against vectorization (vector width = 2, unrolled factor = 4) of the innermost loop based on the cost model. Hence, it simply unrolled the innermost loop by a factor of 2. This resulted in 6.1x performance loss compared to `gcc`. `clang` also decided against vectorization and unrolled innermost loop by a factor of 4, hence lost an opportunity for performance gain. This example showcases the importance of cost models for decision making process for loop transformations and optimizations. The cost models used here were the default ones at the highest optimization level. There are compiler flags available to the users that can override these default cost models.

2.3.3 Example 3: LLVM clang performs the best

Listing 2.3: Example 3

```
1 for (int nl = 0; nl < ntimes; nl++) {
2     for (int i = 0; i < LEN; i++) {
3         a[i * inc] += b[i];
4     }
5 }
```

Here, `clang` performs 4.4x better than `gcc` and 3.8x better than `icc`. Listing 2.3 is loop nest `s171` from the TSVC benchmark. In this example, the innermost loop nest can only be vectorized if variable `inc` has value 1. Otherwise, there could be flow-dependence and anti-dependence on `a[i * inc]`. This variable comes in as a function argument, hence compilers cannot resolve its value at compile-time. `gcc` decided to do no transformations or optimizations for this loop, it just used AVX and AVX2 registers for scalar computations. `icc` unrolled the innermost loop with a factor of 2. Interestingly, `clang` analyzed the peculiarity here and added an additional branch inside the outermost loop to check the value of variable `inc`. This branch checked if `inc` had value 1 at runtime, if so, the code executed is vectorized for AVX-512 with vector width of 16, array `a` is type `float`. Also, `clang` unrolled, or interleaved as they call it, this loop by a factor of 4. If the above mentioned branch fails then the code executed is a loop unrolled by a factor of 4 with scalar operations. The value of `inc` was indeed 1 in the benchmark, and hence `clang` produced higher performance than the other two compilers. This example highlights the importance of handling edge or special cases by the compiler to gain performance. Again, these details are not possible for users to control by either flags or annotating code with hints. These are simply optimizations embedded into the compiler.

2.4 Overall Framework Architecture

The overall architecture of the *MCompiler* framework and the technical details about the individual phases of the framework are discussed in this section. Fig. 2.1 shows the structure

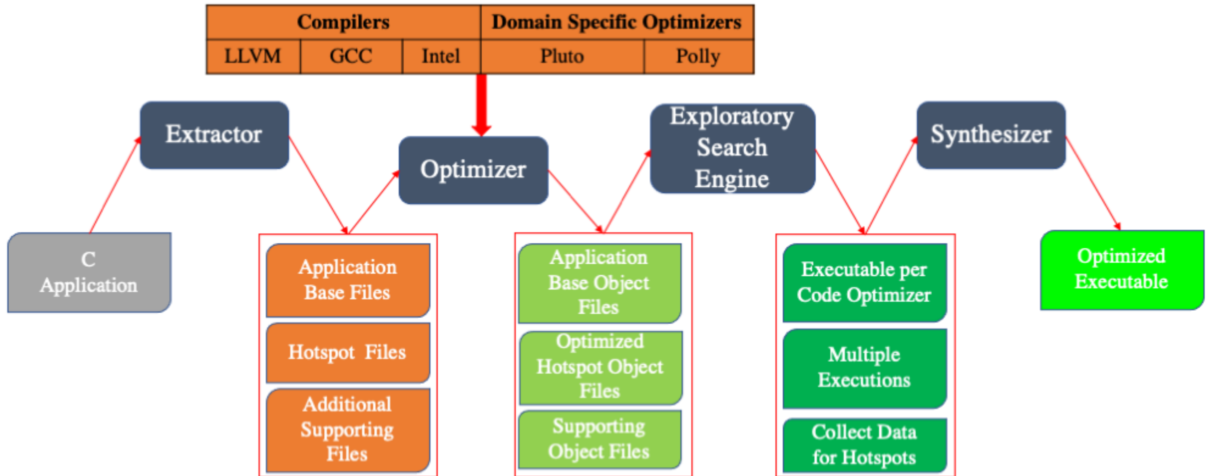


Figure 2.1: *MCompiler* Framework

of the *MCompiler* framework.

The first phase is *Loop Extraction* from `C` applications. The *Extractor* parses the source files to find loop nests, extract those loop nests as `functions` into separate, independently compilable files and replaces the loop nests with the corresponding function call in the *base* source file. Base files are similar to the original source files but with loop nests replaced with function calls. Whereas hotspot files are newly generated files which define the function containing the loop body and supporting components to make them compile successfully.

The second phase is the *Optimization* phase. The *Optimizer* compiles each hotspot file with the available code optimizers. Also, it compiles the base files and additional *MCompiler* files, i.e., files added to support the functioning of the framework. For source-to-source code optimizers, a *default compiler* is used to compile optimized hotspot files, the base files and additional files.

The third phase is the Exploratory Search phase, where an application is executed to record

the execution times of the extracted loop nests. Executables generated for each code optimizer are executed and reported execution times for the loop nests are collected.

The final phase is the *Synthesis* phase. Here, for each extracted loop nest, the collected loop execution times from every code optimizer are compared and the best performing code/optimizer is selected, i.e., the optimized code that executes the loop body in the shortest time. Finally, the default compiler links the selected object files for every loop nest file, plus the object files generated by the default compiler for the base files. This step also requires linking libraries that code optimizers may have used or taken support of for generating code for the hotspot files.

For large applications, if `-c` flag is provided, i.e., compile to object files only, then just the Extractor and the Optimizer are enabled. In such cases, the Exploratory Search Engine and the Synthesizer are enabled only at link-time. The *MCompiler* framework handles flags for macro definitions, paths to header files and libraries for linking, etc. similar to other compilers.

2.5 Loop Extraction Phase

The loop extractor is implemented using `ROSE`, a source-to-source compiler infrastructure [119], and is inspired by the loop extractor described in the work by Chen et. al. [37] that encapsulate loop nests into standalone executables.

The Extractor works in three phases. First, it traverses the abstract syntax tree (AST) and locates the `for` loop nests that are eligible for extraction. Second, the extractor creates a new file for this loop, adds necessary headers and macro definitions in the hotspot file, and also adds `extern` declarations for global variables and global functions, as well as for functions called in the scope of the loop body. It encloses the loop body in a function definition with

parameters being the variables and pointers to the data structures required by the loop body in order to compile and run correctly. Third, in the base file's AST it replaces the loop body with a function call (with required arguments) and adds an `extern` declaration to this function. Finally, it generates the modified base source file and the new hotspot files.

A sample hotspot file for the main kernel from Polybench's Matrix Multiplication benchmark is shown in listing 2.4.

Listing 2.4: Loop Nest extracted from Polybench's Matrix Multiplication Benchmark

```
1  void gemm_kernel_gemm_line89(int* ni_primitive, int* nj_primitive, int*
   nk_primitive, double* alpha_primitive, double* beta_primitive, double C
   [2000][2300], double A[2000][2600], double B[2600][2300]){
2      int ni = *ni_primitive;
3      int nj = *nj_primitive;
4      int nk = *nk_primitive;
5      double alpha = *alpha_primitive;
6      double beta = *beta_primitive;
7      #pragma scop
8      for(int i = 0; i < ni; i++) {
9          for(int j = 0; j < nj; j++) {
10             C[i][j] *= beta;
11         }
12         for(int k = 0; k < nk; k++) {
13             for(int j = 0; j < nj; j++) {
14                 C[i][j] += alpha * A[i][k] * B[k][j];
15             }
16         }
17     }
18     #pragma endscop
19     *ni_primitive = ni;
20     *nj_primitive = nj;
21     *nk_primitive = nk;
```

```
22      *alpha_primitive = alpha;
23      *beta_primitive = beta;
24  }
```

A function of the base file from which this kernel was extracted is shown in listing 2.5.

Listing 2.5: Loop Nest replaced by a Function Call for Polybench's Matrix Multiplication Benchmark

```
1  static void kernel_gemm(int ni,int nj,int nk,double alpha,double beta,
       double C[2000][2300],double A[2000][2600],double B[2600][2300])
2  {
3      int i;
4      int j;
5      int k;
6      // BLAS PARAMS TRANSA = 'N' TRANSB = 'N'
7      // A is NIxNK, B is NKxNJ, C is NIxNJ
8      // C := alpha*A*B + beta*C,
9      extern void gemm_kernel_gemm_line89(int *ni,int *nj,int *nk,double *
           alpha,double *beta,double C[2000][2300],double A[2000][2600],double
            B[2600][2300]);
10     gemm_kernel_gemm_line89(&ni,&nj,&nk,&alpha,&beta,C,A,B);
11 }
```

While traversing the AST for eligible loop nests, the extractor skips loop nests with irregular control flow that hinders extraction, i.e., contains `return` and `goto` statements. Also, it skips loop nests with calls to static functions and static variables since those properties hinder their usage in the new hotspot files.

The extractor generates two versions for each hotspot file, where one version is instrumented to collect the execution time for the loop nest. This version is used during the Exploratory Search phase. The other version does not contain any instrumentation code and is used to generate the final executable for the applications.

Function Definition enclosing the Loop Nests

The extractor generates the list of variables, with their data types, used inside the scope of the loop body. All primitive data types (`int`, `float`, etc.) are passed by reference, as well as the user-defined types such as arrays, `structs` and `typedefs`. The extractor also does an optimization to maintain properties of the loop from the point of view of the code optimizers. This optimization copies the function parameters of primitive types (passed by reference) into local variables (with same names as original variables) before the loop body and correspondingly copies the local variables into the function parameters at the end of the loop body. This optimization prevents any change to loop body and is also critical to performance since usage of pointers can prevent some code optimizations.

The extractor also annotates loop nests with `pragma scop/endscop` so as to aid source-to-source Polyhedral optimizers, such as Pluto, in locating Static Control Parts (SCoP)[47, 48]. If the loop nest was indeed not a SCoP, then Polyhedral optimizers can't optimize them. The framework will recognize that in the Optimization Phase and discard Polyhedral optimizers as a candidate for those loop nests. For loop nests with OpenMP directives, the extractor moves the directives with loop body and sanitizes the clauses of variables that are not present in the scope of the loop nest. For OpenMP `for` loops that are enclosed in a `omp parallel` region, extracting the loop body with `omp for` directive doesn't change the behavior of the program. One issue with extracting OpenMP `for` loops that are enclosed in a `parallel` region in such manner is that in the presence of `threadprivate` variables, synthesizer encounters a link-time error because compilers may generate different symbols for the same `threadprivate` variable.

2.6 Optimization Phase

The framework currently uses five candidate code optimizers: Intel's `icc`, GNU's `gcc`, LLVM `clang`, LLVM based polyhedral loop optimizer `Polly` and source-to-source polyhedral loop optimizer `Pluto`. `icc` is chosen as the default compiler because its performance is, on average, the best of the compilers included, as the results show in Chapter 3. It is also used to compile source files generated by a source-to-source loop optimizer, i.e., Pluto. Table 2.2 shows the flags used for optimizing loop nests for serial execution and parallel execution. These flags also include target architecture specific flags to enable optimizations that can generate better performing code on the specific architecture. For OpenMP applications, flags from serial configuration are used in addition to the OpenMP flags. The optimizer can

Compiler	**Version**	**Optimization Flags**	**Auto-Parallelization**
clang (LLVM)	10.0.0	-O3 -march=native	No
gcc (GNU)	10.1.0	-Ofast -march=native	No
icc (Intel)	19.1.0	-Ofast -xHost	Yes (-parallel)
Domain Specific Optimizer	**Version**	**Optimization Flags**	**Auto-Parallelization**
pluto (source-to-source) + icc	0.11.4	--tile	Yes (--parallel)
polly (LLVM)	10.0.0	-polly-tiling -polly-vectorizer=polly	Yes (-polly-parallel)

Table 2.2: Compilers and Domain Specific Optimizers integrated in the *MCompiler* Framework.

compile hotspot files and base files in parallel. This is similar to `-j` option of Makefiles, but here all candidate code optimizers are invoked in parallel to compile the source files. This reduces the overall compilation time for the *MCompiler* framework. The optimizer generates multiple executables of the application (with instrumentation code) where each executable is completely compiled and linked by a candidate code optimizer.

2.7 Exploratory Search Phase

The Exploratory Search Engine invokes executables generated by the code optimizers one-by-one and performs multiple runs for stable data, if requested. Exploratory Search Engine at the end of each execution collects the information for each of the loop nests and forwards it to the Synthesizer. For applications that need input through command line, the Exploratory Search Engine runs the application with the input given to the *MCompiler* framework using a `--input` flag.

2.8 Synthesis Phase

The synthesizer compares the collected execution times for each loop nest from different code optimizers and chooses the code optimizer that performed the best as the most suited code optimizer. For loop nests with no information, i.e., the code that was not executed during Exploratory Search phase, the default compiler is used. The synthesizer then generates the final executable that contains no instrumentation code. For an OpenMP application, the synthesizer links OpenMP runtime libraries that are used by different compilers, e.g., `icc` and `clang` use compatible OpenMP runtime libraries whereas `gcc` doesn't. Therefore, for example, if for an application *MCompiler* chooses a `omp parallel for` region from `icc` and another from `gcc`, then the parallel regions will be executed by different OpenMP runtime libraries. Static libraries specific to compilers are also linked to successfully generate the final executable.

```
$ MCompiler
Usage:  MCompiler [options] file1 [file2 ...]
Options:
    -h,--help               Print usage
    --[no]extract           Extract hotspots
    --[no]profile           Profile extracted hotspots
    --[no]synthesize        Combine best performing hotspots to generate binary
    --adv-profile           Advanced Profiling
    --predict               Predict candidate using ML
    --energy                Choose most energy efficient candidate
    --test                  Test performance compared to other compilers optimized code
    --parallel              Generate multi-threaded code based on OpenMP directives
                            Default: Serial code generation (with vectorization)
    --auto-parallel         Auto-parallelize the hotspots
    --extract-kernel        Extract consecutive loop nests, if possible.
    --restrict              Add restrict keyword.
    --static                Perform static analysis to determine read-only values.
    --prefetch              Enable software data prefetching
    --no-vec                Disable vectorization
    --max-vec               Generate code for maximum vector length
    --disable-polyhedral    Disable Polyhedral Model based loop optimizers
    --disable-s2s           Disable Source-to-Source loop optimizers
    --profile-runs=<num>    Number of time profiler should run the program to collect data. Default: 3
    --input=<args>          Input to the program. Needed to generate profiling information.
    --predict-model=<args>  Path to the trained ML model. Default: MC_trained_model.yml
    --haswell               Compile for Intel Haswell processor. Default: Skylake
    --knl                   Compile for Intel Knights Landing processor. Default: Skylake
    --skylake               Compile for Intel Skylake processor.
    --c99                   Conforms to ISO C99 standards. Default: C11
    -j                      Compile hotspots in parallel
    -c[<arg>]               Compile to object file
    -o[<arg>]               Output object/binary name
    -I[<arg>]               Directory to include file search path
    -L[<arg>]               Directory to search for libraries
    -l[<arg>]               Instruct the linker to link in the -l<string> library
    -D[<arg>]               Macro definition
    --debug                 Debug the MCompiler workflow
    --info                  Print information for MCompiler workflow
```

Figure 2.2: *MCompiler* Command Line Options

2.9 Using and Expanding the Framework

The *MCompiler* framework can also be used for testing and comparing optimization techniques and/or cost models among compilers or versions of a individual compiler. Various flags are available for choosing the target architecture and choosing particular optimizations such as auto-parallelization optimizations or enabling particular passes such as data prefetching pass. The command line options for the *MCompiler* framework are shown in Fig. 2.2.

The *MCompiler* framework allows for addition of code optimizers so as to give more options for generating the optimized applications. In addition to that, the framework allows for

adding different combinations of compiler flags or code optimizer flags to optimize applications. This allows users to explore how different code optimizer flags impact the performance of applications and use *MCompiler* framework to generate even better performing executables. By its design the framework can also include auto-tuning frameworks, such as domain-specific auto-tuner called OpenTuner [9], for optimizing applications.

Chapter 3

Evaluation of the Multiple Compiler Approach for Improved Performance

This chapter describes the experimental methodology and presents the results and their analysis demonstrating the effectiveness of the *MCompiler* framework.

3.1 Benchmarks, Code Optimizers and Target Architecture

Several different benchmark suites are used to evaluate the effectiveness of the *MCompiler* framework. One is Test Suite for Vectorizing Compilers (TSVC) by Callahan et al. [25] and Maleki et al. [94]. This benchmark was developed to assess the auto-vectorization capabilities of compilers. Therefore, these loop nests are only used in the serial code related experiments. The second benchmark suite used is Polybench [114]. This suite consists of 30 benchmarks that perform numerical computations used in various domains, such as linear algebra computations, image processing, physics simulation, etc. The benchmarks in

Polybench have been demonstrated to have performance gain on parallelization, therefore these loop nests are used for auto-parallelized code experiments as well. The third benchmark suite is NAS Benchmark Suite [14], especially NPB-ACC [147]. The fourth benchmark suite is Parboil [131]. These benchmarks are used in serial code, auto-parallelized code and OpenMP parallel code experiments. Lastly, a set of `C` benchmarks from SPEC OMP 2012 were used for OpenMP experiments. The largest datasets were used for the results of these benchmarks, for example, XL datasets were used for Polybench benchmarks and class A datasets were used for NPB benchmarks. If the datasets can be specified at execution time (such as for Parboil), then the smaller datasets were used for the Exploratory Search and evaluation was done on the largest datasets. The `train` dataset was used for SPEC benchmarks during the exploratory search phase, whereas the results are shown for `ref` dataset.

Table 2.2 showed the five code optimizers incorporated in the *MCompiler* framework. All five optimizers are used for serial and OpenMP experiments. Of the five optimizers, only three optimizers (`icc`, `Polly` and `Pluto`) can auto-parallelize the serial code and are used for auto-parallelized code experiments. The baseline for performance comparison is `icc` (`-Ofast -xHost [-parallel]`) compiled benchmarks for all experiments. `icc` was chosen as the baseline because `icc` generated code performed better for more benchmarks than other code optimizers as shown in Fig. 3.4. The source codes used for the baseline are the original benchmark codes and not the modified source codes generated by the *MCompiler*'s Loop Extractor.

The target architecture for the experiments is a two-socket, sixteen-core Intel Skylake Xeon Gold 6142 [43]. Each Xeon processor has 32KB L1 cache, 1MB L2 cache, 22MB L3 cache. The Skylake architecture supports SIMD instruction set extensions, i.e., SSE, AVX, AVX2, AVX-512CD and AVX-512F. CPU Hyper Threading (SMT) is turned off and cores are operating at the maximum frequency. For the auto-parallelization and OpenMP experiments, only one thread is mapped per core by setting the environment variables for OpenMP run-

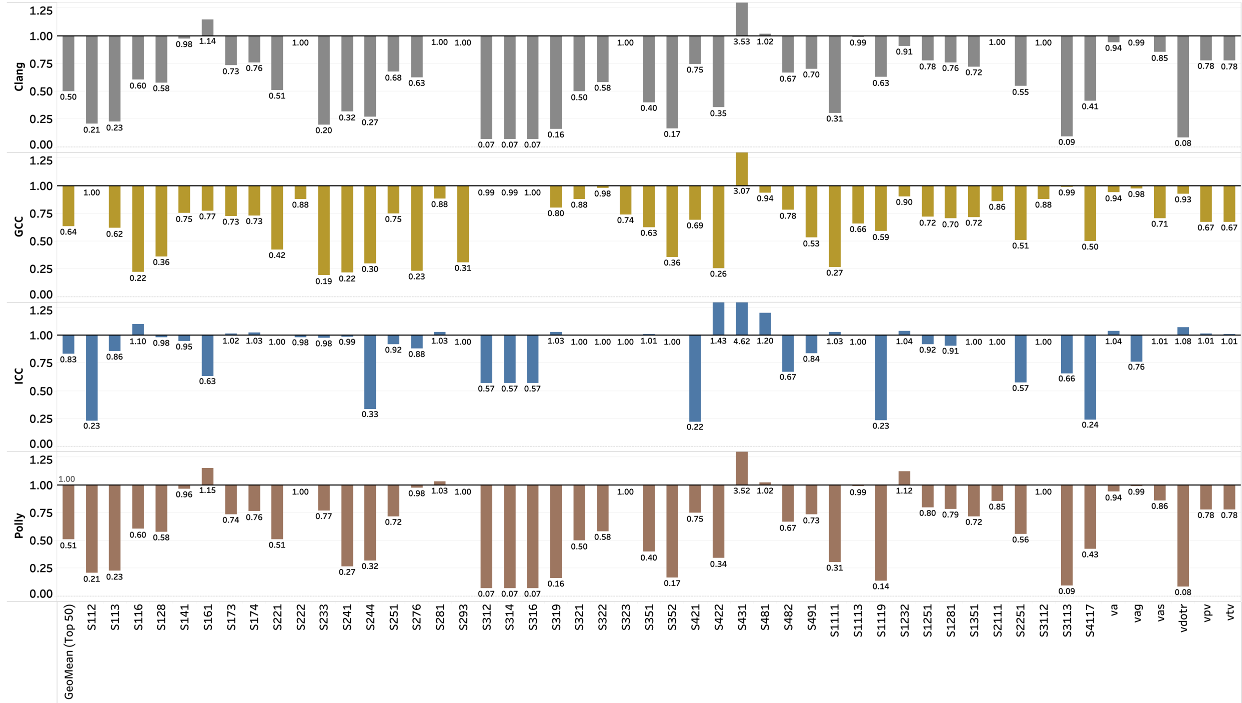


Figure 3.1: Performance of individual Code Optimizers vs *MCompiler* on TSVC benchmark (top 50 loop nests). A value of 1 indicates the same performance as the *MCompiler*, less than 1 means a slower performance than the *MCompiler*.

times.

3.2 Comparing all Code Optimizers with the *MCompiler*

This section presents and analyzes the performance of loop nests optimized by all code optimizers individually and comparing them to the *MCompiler*'s Exploratory Search. The TSVC benchmark suite consists of 151 unique loop nests designed especially for evaluating code optimizers. The original TSVC kernels were compiled by each code optimizer and the loop nest execution times were collected for each. Note that these execution times are not affected by the *MCompiler* Loop Nest Extractor. Hence, this study also allows for analyzing the impact of Loop Nest extraction on optimizations and performance. `Pluto` is left out from this evaluation, since it needed manual annotation of loop nests for locating Static Control

Parts (SCoP) in the original TSVC code. `Pluto` was also disabled inside *MCompiler* in this experiment to be consistent in this particular case study. Only results for the top 50 loop nests in terms of execution time are shown in Fig. 3.1 to allow better visualization. The performance of individual code optimizers is shown against the *MCompiler*, i.e., the execution time for *MCompiler* generated code is divided by the execution time of the individual code optimizer's generated code. Performance of less than 1 for a Code Optimizer indicates that the *MCompiler* generated faster code. Performance of at least one code optimizer against the *MCompiler*, ideally, must be equal to 1 for every loop nest. Overall, the results show that code produced by the *MCompiler* is faster or equal for almost all loop nests, in many cases significantly faster, than individual code optimizers. The *MCompiler* thus performed better than each individual code optimizer, hence demonstrating the strength of the proposed approach.

The results also show that `icc` performed better than other code optimizers, with the Geometric Mean performance of 0.83 w.r.t the *MCompiler*. It also has the fewest loop nests that the *MCompiler* improved, compared to other optimizers. The gap in performance of the best code optimizer versus the worst code optimizer can be large is some cases. For example, for `vdotr`, that performs a reduction on a product of two vectors, *MCompiler* picks `icc` generated code which is almost 12 times faster than `clang` and `polly` generated code. In other cases there is not much difference between optimizers in the performance of generated code. For example, for `va`, that performs a simple vector assignment, there is around 11% difference between the generated code from the best code optimizer and the worst code optimizer. Such examples demonstrate that for some loop nests that are either simpler to optimize or don't have much room for optimizations, most code optimizers are able to generate similar, high-quality code in terms of performance. It is the cases where the optimization space becomes quite large that the code optimizers start to show large differences in the generated code and therefore large variations in performance.

There are 2 cases in Fig. 3.1 which are particularly interesting since the *MCompiler* framework performance matches none of the code optimizers. There is `S421` where the *MCompiler* is performing better than all the code optimizers it used to optimizer loop nests. Then there is `S431` where the *MCompiler* is performing worse than all the code optimizers.

In `S421`, `icc` was chosen as the most suited code optimizer by the *MCompiler*. This case is quite peculiar. `icc` assumed a dependence between two arrays, even though one of the arrays was marked with the `restrict` keyword. Therefore, `icc` did not generate vectorized code in this case. The *MCompiler* extracted the loop nest and used the `extern` definition of arrays marked with the `restrict` keyword as information to resolve aliasing issue. Therefore, the *MCompiler* generated vectorized code for the loop nest.

In `S431`, the *MCompiler* performance is worse than all the code optimizers. The reason is that on extracting the loop nest, the value of variable `k` in a statement `a[i] = a[i+k] + b[i]` is unknown as it is passed by reference. Therefore vectorized code couldn't be generated. The value of variable `k` can be computed to be zero using constant propagation and constant folding and this information allows code optimizers to generate vectorized code for the loop nest.

3.3 *MCompiler* with Exploratory Search

This section presents results of the exploratory search by the *MCompiler* for choosing the most suited code optimizer for four benchmark suites: TSVC, Polybench, NAS Benchmark Suite (NPB) and Parboil Benchmark Suite. Each application was executed 3 times for each of the code optimizers and the median execution time was chosen for deciding the most suited code optimizer.

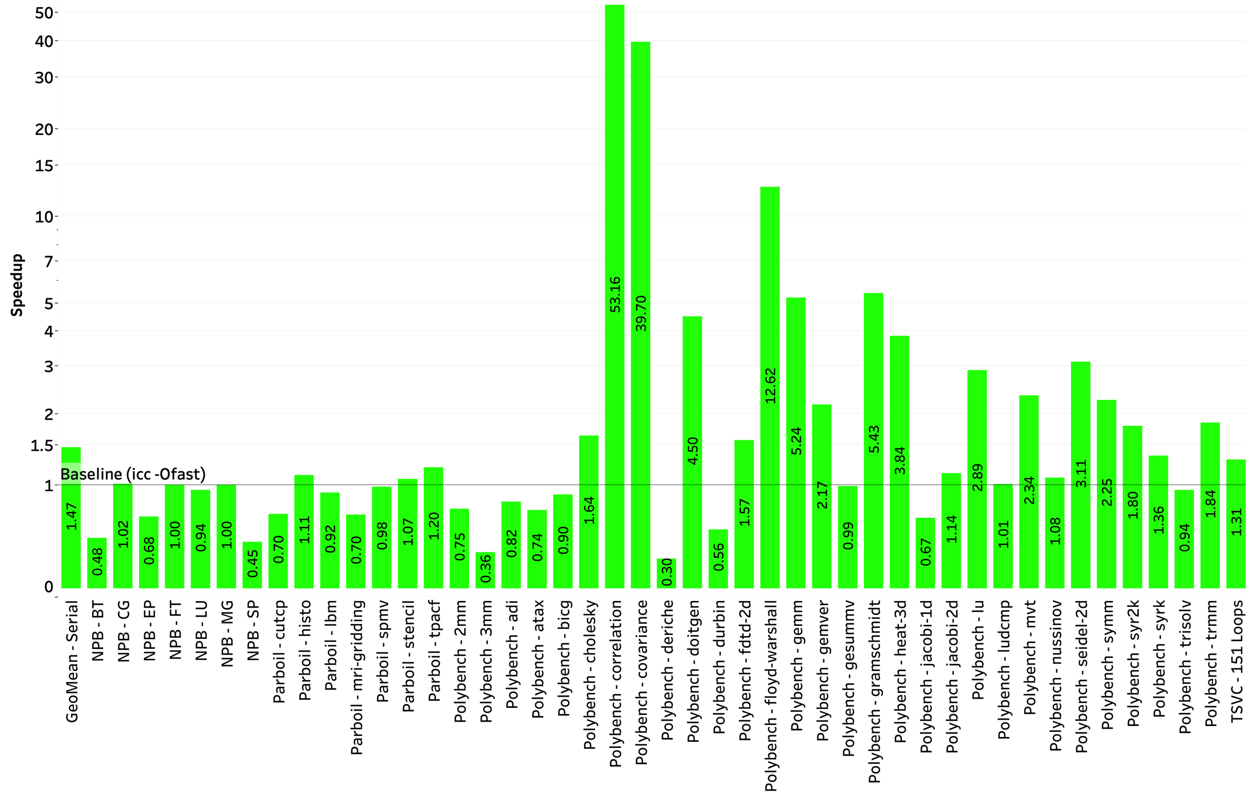


Figure 3.2: *MCompiler* Speedup for Serial Benchmarks

3.3.1 Serial Code

The results are shown in Fig. 3.2. The benchmark labels show the benchmark suite that a particular benchmark belongs to. The speedup across the 151 loop nests from TSVC is 1.31x over `icc`. As shown in Fig. 3.4, `icc` was chosen as the most suited code optimizer for 43.7% of the loop nests, followed by `Pluto` (source-to-source optimizer, compiled with `icc`) at 20.5%. In many of those 20.5% cases, loop tiling and automatically added `#pragma ivdep` (hint for the compiler to ignore assumed vector dependencies) on the inner-most loop from `Pluto` (followed by vector code generation from `icc`) provided better performance than just `icc` itself.

The performance of the *MCompiler* for Polybench benchmarks is 1.92x (GeoMean) better than `icc`. As expected, the two polyhedral model based optimizers were chosen as the most

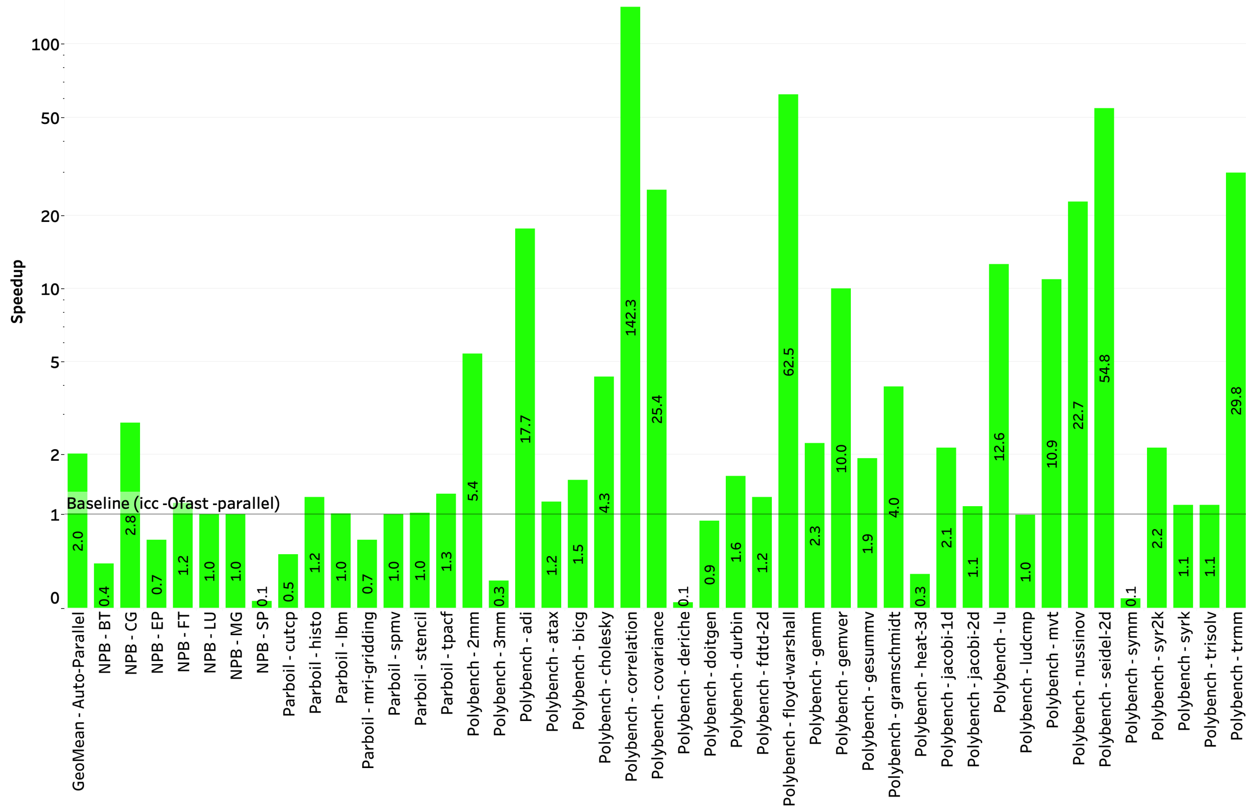


Figure 3.3: *MCompiler* Speedup for Auto-Parallelized Benchmarks

suited code optimizer for 60% of the loop nests that dominate execution time of the main kernels for Polybench benchmarks. `icc` was chosen as the most suited code optimizer for 21% of the kernel loop nests, with the remaining loop nests split between `clang` and `gcc`. `icc` was chosen as the most suited code optimizer for 133 out of 261 (53%) loop nests from NPB benchmarks (not counting loop nests such as array initialization loops). Similarly, `icc` was chosen as the most suited code optimizer for 45% loop nests from Parboil benchmarks.

Overall the percentage of loop nests chosen from each code optimizer can be seen in Fig. 3.4. For analysis shown in Fig. 3.4, trivial loop nests that perform tasks that do not test the optimization capabilities of the code optimizers are removed. For example, loop nests that are used to allocate dynamic memory, to perform array initialization, etc. It shows that across all benchmarks, while `icc` dominates overall, 45% of loop nests are best optimized by other code optimizers (with approximately equal distribution among them, except for `gcc`).

	TSVC		Polybench		NPB		Parboil		Total	
	Loop Nests	Percentage	Loop Nests	Percentage	Loop Nests	Percentage	Loop Nests	Percentage	Loop Nests	Percentage
Clang	16	10.6%	8	12.1%	31	11.9%	2	6.9%	57	11.2%
GCC	30	19.9%	4	6.1%	66	25.3%	9	31.0%	109	21.5%
ICC	66	43.7%	14	21.2%	133	51.0%	13	44.8%	226	44.6%
Pluto*	31	20.5%	22	33.3%	2	0.8%	1	3.4%	56	11.0%
Polly	8	5.3%	18	27.3%	29	11.1%	4	13.8%	59	11.6%

(a) Serial Code

	Polybench		NPB		Parboil		Total	
	Loop Nests	Percentage	Loop Nests	Percentage	Loop Nests	Percentage	Loop Nests	Percentage
ICC	15	22.7%	190	72.0%	18	62.1%	223	62.1%
Pluto*	18	27.3%	1	0.4%	1	3.4%	20	5.6%
Polly	33	50.0%	73	27.7%	10	34.5%	116	32.3%

(b) Auto-Parallelized Code

* Pluto optimized code was compiled with ICC.

Figure 3.4: Distribution of best performing code per Code Optimizer. Breakdowns per benchmarks suite showcase benefits of specialized code optimizers.

More details for specific cases are explained in section 3.3.4.

3.3.2 Auto-Parallelized Code

These experiments were performed with 32 threads for both the exploratory search phase and evaluating the performance. The code optimizers optimized the loop nests with their default setting for statically deciding the profitability of the parallel code and for choosing the runtime settings, such as scheduling policies.

Benchmarks from Polybench, NPB-ACC and Parboil were used in these experiments. Poly-

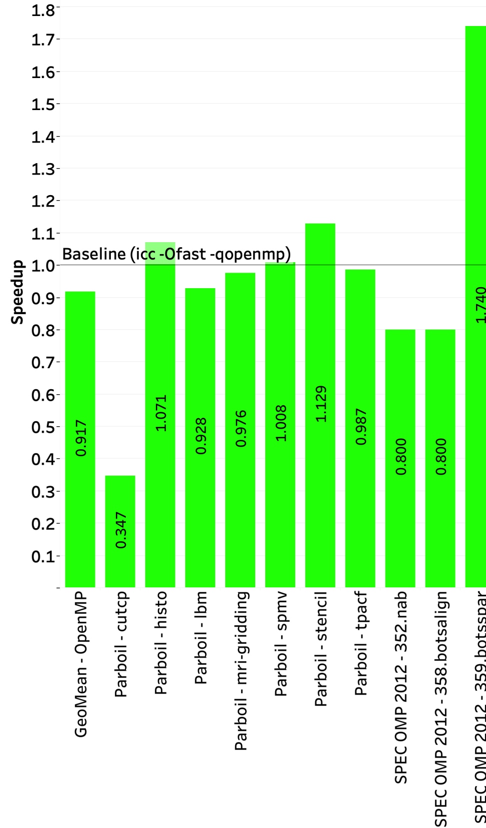


Figure 3.5: *MCompiler* Speedup for OpenMP Benchmarks

bench was shown to have auto-parallelizable loop nests in previous works. NPB benchmarks use either OpenMP or OpenACC parallel directives and therefore have potential for auto-parallelization. The directives were removed from the source code prior to processing for Auto-Parallelized code experiments.

The results in Fig. 3.3 show that the *MCompiler* improves performance over `icc`, by at least 5%, for 28 out of 44 benchmarks. Several additional benchmarks have no change in performance. Five have a significant performance loss, which is explained in section 3.3.4. Overall the percentage of loop nests chosen from each code optimizer can be seen in Fig. 3.4. Similar to the trend seen for serial code benchmarks, `icc` dominates for NPB benchamrks, whereas polyhedral model based optimizers perform better for Polybench benchmarks.

3.3.3 OpenMP Code

The results are shown in Fig. 3.5. Loop nests that were not marked by OpenMP directives were optimized by the *MCompiler* as serial loop nests. Much performance improvement is not expected from OpenMP regions, since code optimizers lose flexibility to optimize the OpenMP regions due to issues such as early outlining [19, 41] of code. The results show that in a few cases high speedups can indeed be achieved using the *MCompiler*. The reason for such performance gains is explained in section 3.3.4.

3.3.4 Analysis of Results

Analysis of the benchmarks that get slowdowns from the *MCompiler*, such as 3mm (serial and parallel), deriche (serial and parallel), heat-3d (parallel), symm (parallel) from Polybench, and SP (parallel) from NAS benchmark showed that the main reason for performance loss, based on compiler generated reports, is the early outlining of loop nests into individual functions. Early outlining hinders the interprocedural and alias analysis, and therefore compilers may generate sub-optimal code for such loop nests. This issue is similar to the ones faced by the OpenMP compilers [41] that outline OpenMP sections early in the compiler toolchain and therefore inhibit compiler optimizations later in the process.

Another reason for slowdowns can be attributed to the presence of loop nests that have a very short execution time and/or executed multiple times (in a `while` loop, for example), and perform trivial tasks such as iterating through a linked list. For such loop nests, the *MCompiler* extraction adds performance overheads. This problem can be solved in the Extractor by automatically identifying trivial loop nests with low loop trip count. This is subject of future work. For now, if users want to manually mark such trivial loop nests with low loop trip count, they can add `pragma MC skiploop` directive.

Also, the baseline compiler, i.e. `icc`, analyzes the entire source file and can find more opportunities for optimization, including single-file interprocedural optimizations such as inlining.

Fig. 3.5 shows speedups for OpenMP benchmarks. It shows significant speedups for *MCompiler* on `359.botsspar` from `SPEC OMP 2012`, and `histo` and `stencil` from `Parboil`. The reason for speedup in `359.botsspar` is a loop nest with a computation similar to matrix multiplication that is enclosed in a function, that is called inside a `omp task` region. The *MCompiler* optimized this particular loop nest as it do would for a serial loop nest and chose `Polly` generated code for this loop nest. The reason for speedup in `histo` and `stencil` is that, inside the `OMP parallel for` region, the inner most loop was vectorized and unrolled by `gcc` better than the other code optimizers.

Key factors contributing to performance difference between code optimizers, other than their loop transformations, are as follows. First, a difference in unroll factor, which leads to the difference in type of vector instructions selected and also leads to more consecutive load/store of data. Second, generation of multi-variant code, which chooses the best code during execution based on runtime analysis of dependences. Third, use of specialized libraries, such as the vectorized math library.

3.4 Summary

This chapter presents and analyzes the performance improvements achieved by using the *MCompiler* framework for serial (auto-vectorized) code, auto-parallelized code and hand-optimized code.

The *MCompiler* framework improves the overall performance for applications over state-of-the-art compiler (compiled at equivalent of `-O3`) by a geometric mean of 1.47x for serial,

auto-vectorized code and 2.00x for auto-parallelized code. Hand-optimized parallel applications (with OpenMP directives) are also improved by the MCompiler, with performance improvement up to 1.74x.

Chapter 4

Predicting the Best Code Optimizer for the Loop Nests

This chapter, first, presents the Machine Learning based techniques that learn about the inherent characteristics of the loop nests and then predict the most suited code optimizer for a given loop nest. Second, these ML models are then incorporated into the *MCompiler* framework for predicting the most suited code optimizer for the loop nests and the results are compared to the Exploratory Search step of the framework.

4.1 Towards an Achievable Performance for Loop Nests

Based on the exploration and evaluation of different compilation and optimization techniques shown in the previous chapters, we know finding an optimal sequence of transformations is a complex problem. Each code optimizer has a unique set of transformations for generating and optimizing code for a program segment, specially loop nests. Additionally, code optimizers have to decide the heuristics and profitability models for predicting the behavior of

a multi-core processor, which oftentimes has complex pipelines, multiple functional units, complex memory hierarchy, hardware data prefetching, etc. Parallelization of loop nests involves further challenges for the code optimizers. The impact on the performance stemming from workload balancing and communication costs related to the temporal and spatial data locality among iterations becomes increasing harder to model. Studies [102, 134, 94, 54] have shown that state-of-the-art code optimizers may not auto-vectorize and auto-parallelize the loop nests for modern architectures.

Loop nests have different inherent characteristics based on iteration count, data dependences, memory access patterns, types and counts of operations, presence of branches, etc. Analyzing and evaluating these characteristics of the loop nests is critical for the code optimizers' transformations and cost models to make decisions. Few examples where these decisions need to be made are as such. Evaluating if the reuse of the cached data can be improved by interchanging the order of loops without violating the data dependences. Deciding if the iteration count is large enough and/or data access patterns are such that the vectorized code will produce better performance than scalar code. Vector instructions have different latency and throughput than the scalar instructions for the same operation and this needs to be taken into account for judging the profitability of vectorization. Loop Unrolling, for example, can be beneficial but the optimal unrolling factor can differ on case by case basis. Unrolling (with or without SIMD instructions) will benefit from less branch instructions and less loop counter increments, and also more data reuse if the data is shared between iterations. But, unrolled loops require more instruction decoding and will use more instruction cache (I-cache). An optimal unrolling factor also depends on architectural features such as a Loop Stream Decoder (LSD), that provides the processor with better μOp supply for small loops that can be cached in the LSD buffer. Another example would be Loop Tiling, where selecting the optimal tile sizes for L1-D cache, L2 cache and beyond for a given processor can significantly improve the performance by improving data locality.

There are two research questions that stem from these observations:

- Can we learn about the inherent characteristics of the loop nests in terms of its behavior on the architecture?
- Can we use this knowledge to predict which code optimizer (with its transformations and cost models) would be the most beneficial for a loop nest?

This chapter presents use of Machine Learning (ML) algorithms to learn such inherent characteristics and predict the most suited code optimizer for a given loop nest. Applying Machine Learning models in compilers is continuously being explored by the research community [101, 129, 128, 2, 28, 134, 139, 51, 110, 130, 26, 9, 87, 140, 10, 126, 11, 97, 99, 98, 57]. Most of the previous work used Machine Learning in the domain of auto-tuning, auto-vectorization, phase-ordering and parallelism runtime settings. This work is the first to show the possibility of predicting the best code optimizer for loop nests. Machine Learning models in these studies either used a mix of static features (collected from source code at compile time) and dynamic features (collected from profiling) [42, 134, 139], or exclusively used dynamic features [28, 110, 140, 10, 126]. The approach used here belongs to the latter class and exclusively relies on hardware performance counters collected for a loop nest. Most recent works exclusively use dynamic features for training Machine Learning models, since models trained using dynamic features exclusively have been shown to learn more about inherent code characteristics, making the use of static features redundant. Previous studies have shown that hardware performance counters can successfully capture the characteristic behavior of a loop nest [140, 126]. Hardware performance counters capture intricate details about data movement across levels of caches, memory footprint, and count and types of instructions retired that determine the performance of loop nests on an architecture.

For the work presented in this chapter, the best results were achieved using the hardware performance counter data collected from profiling a serial minimally unoptimized version of

a loop nest. The reason for using the minimally unoptimized version of the loop nest is that this version shows inherent code characteristics, i.e., generated code is an unoptimized version without any complex code transformations. This version is similar across compilers and therefore provides a good common baseline. Another advantage of using this version is that the hardware performance counter data collected from one architecture can be used as features for making Machine Learning predictions for another architecture. Although, this still requires training new Machine Learning models for different architectures with architecture specific most suited code optimizer as the *target* class.

The focus here is to consider state-of-the art code optimizers and then use Machine Learning algorithms to make predictions for better, yet clearly achievable performance for the loop nests using these code optimizers.

These ML models are incorporated into the *MCompiler* and provides a substitute for the expensive Exploratory Search step of the framework. However, as with any prediction, it can lead to a potential performance loss compared to search-based selection due prediction errors, e.g., when the ML model or classifier does not choose the best code optimizer. The results show that by using well-trained ML models this potential loss in performance can be quite small.

4.2 Experimental Methodology and Training the Machine Learning Models

This sections describes the methodology for collecting data, transforming data for Machine Learning models and finally training the *Classification* algorithms.

4.2.1 Collecting Hardware Performance Counters using Profiling

The features, i.e., the hardware performance counters used for the Machine Learning (ML) models are collected by profiling loop nests using Intel's VTune Amplifier. The code is generated from Intel's ICC compiler to generate the executable that is then used for profiling. All the loop optimizations are disabled during this compilation by using the `-O1` flag. In addition to that, the optimizations that are responsible for vector code generation and parallel code generation are disabled too. The profiling information, therefore, provides an insight into the characteristics of the loop nests while eliminating the influence of compiler transformations and behavioral changes incurred from special architectural features of the underlying architecture. The performance counters that are collected include instruction-based (instruction types and counts), CPU clock cycles-based (including stalls), memory-based (D-TLB, L1 cache, L2 cache, L3 cache).

Once the hardware performance counters are collected for the loop nests, dynamic instruction count is skipped as a feature and the rest of the hardware performance counters are normalized in terms of *per kilo instructions* (PKI). Based on the analysis that is done for this work, this allows the Machine Learning models to learn about the inherent characteristics of the loop nests and not bias them towards characteristics such as loop trip count.

4.2.2 Most Suited Code Optimizer

To train the ML models, each feature vector is correlated with a *target* class. Since this work focuses on using hardware performance counters as features to predict a code optimizer, ML *Classification* algorithms are used with code optimizers being the *target* class.

To create the dataset, the loop nests are optimized with the candidate code optimizers and then executed to record their performance in terms of wall clock time. The code optimizers

are configured to optimize the loop nests for the underlying architectures for which the ML models are trained. These optimizations include the loop transformations and code generation optimizations for the specific architectures. For serial code experiments, transformations for auto-vectorization are enabled. Whereas, for parallel code experiments, transformations for auto-vectorization and auto-parallelization are enabled.

The most suited candidate for a loop nest on a particular architecture, i.e., the code optimizer that produces the best performing code for the loop nests, is appended in the dataset as the target class for that specific loop nest. The feature vector in the dataset remains same across all experiments, just the target class differs based on the architecture the predictions are made for and the purpose of the classifier.

In this work, 4 candidate code optimizers are considered, as shown in Table 4.1, including Polly[56, 113], a Polyhedral Model based optimizer for LLVM. 2 out of those 4 optimizers can perform auto-parallelization of the loop nests. The hardware performance counters are collected using an executable generated by `icc` with flags `-O1 -no-vec`, in order to disable all loop transformations, and disable vector code and parallel code generation.

Code Optimizer	**Optimization Flags**	**Auto-Parallelization**
clang (LLVM)	-O3 -march=native	No
gcc (GNU)	-Ofast -march=native	No
icc (Intel)	-Ofast -xHost	Yes (-parallel)
polly (LLVM)	-O3 -march=native -polly -polly-vectorizer=stripmine -polly-tiling	Yes (-polly-parallel)

Table 4.1: Candidate Code Optimizers used for the ML Experiments.

4.2.3 Random Decision Forest Classifier

Random Decision Forest (RF) [62, 23] is, for this work, the classification algorithm of choice to predict the optimal compiler for the loop nests. RF performed better overall than other classification algorithms such as Support Vector Machines (SVM), k-nearest neighbors (KNN), Gradient Boosting Machine (GBM) [50] and AdaBoost [123] in terms of Classification Accuracy and Area under the ROC Curve (AUC). RF have previously been shown to be one of the best Supervised Learning algorithm for classification problems [27].

RF is a learning algorithm that builds on the principles behind Decision Trees. Generally, the Decision Tree algorithm, learns from training data by building a structured and hierarchical representation of the correlation between features and classes. Features represent the nodes in the trees and classes are leaves at the deepest level. An optimal Decision Tree would perfectly and accurately divide the data among the target classes. However, finding an optimal tree is an NP-Complete problem [86], therefore heuristics such as greedy search are needed.

Decision Trees suffer from several issues. The main one being a tendency towards overfitting, that is, the tree loses generalization the deeper the tree goes, modeling the trend for training data but be inaccurate for new instances. RF provides a better solution for overfitting and classification bias by adding two stochastic steps to the Decision Tree Algorithm. From the training dataset, RF creates a bootstrapped subset by stochastically choosing the instances or features (with repeats allowed) that will be used for building the decision trees. This is called Bootstrap Aggregating or Bagging. After creating the Bagged dataset, an arbitrary number of decision trees are built using subsets of randomly chosen features. Each decision tree accuracy is evaluated using the remaining instances that weren't part of the Decision Tree building phase.

Classification is achieved through a voting algorithm, where a target value is generated by

each Random Tree, the one with the highest number of trees will be the class assigned to the new instance.

Since Random Forest is constantly evaluating the performance of the subsets of features, it is easy to detect the ones that were used in the better performing trees. Therefore, it requires little to no feature filtering before running the algorithm. This is useful when approaching a new problem where the correlation between the input features and the output class is not entirely known.

4.2.4 Machine Learning Model Configuration

The dataset, or the loop repository, is partitioned to create a training dataset and a validation dataset. The training dataset is used to train and tune the ML models. The trained models are evaluated on Accuracy and Area Under Curve (AUC). Whereas, the validation is a set of unseen loop nests that are used to make predictions. The results reported in the following sections in regards to the performance of the ML predictions as based on the loops from the validation dataset. For training and evaluating the Machine Learning model, Orange[40] is used.

The dataset is randomly partitioned into Training dataset (75%) and Validation dataset (25%). Whereas, the validation dataset is a set of unseen loop nests that are used for making the predictions. For serial code experiments, there are 209 instances (loop nests) in the training dataset and 69 instances in the validation dataset. For auto-parallelized code experiments, there are 147 instances in the training dataset and 49 instances in the validation dataset. The predicted optimizer's execution time as compared to that of the most suited optimizer's execution time will be same in case of correct predictions and higher in case of mispredictions.

The ML experiments were repeated thrice in order to validate the results, i.e., the dataset is randomly split, train new ML models and then make the predictions. The unique instances from the three validation datasets are taken into the account for the measurements. Therefore, the number of instances differ between similar experiments.

Serial Code

The dataset consists of hardware performance counters values for the loops and the most suited optimizer for an architecture as the target. Later, the RF classifier is trained to make the predictions. A Majority Classifier serves as the baseline to evaluate the trained model. A Majority Classifier is a feature-agnostic classifier that determines the output for every instance to be equal to the target that has most instances in the training set.

Auto-Parallelized Code

In this case, the dataset uses the same set of hardware performance counters as those used to predict the most suited serial code optimizer. The model predicts the most suited code optimizer for the loop using an RF classifier. Therefore, for the dataset the target is the optimizer that produces the best performing auto-parallelized code for the loop nests.

4.2.5 Benchmarks

The first benchmark suite that is used for the experiment is Test Suite for Vectorizing Compilers (TSVC) as used by Callahan et al.[25] and Maleki et al.[94] for their works. This benchmark was developed to assess the auto-vectorization capabilities of compilers. Therefore, those loop nests are only used in the serial code related experiments. The second benchmark suite that is used for collecting the loop nests is Polybench[114]. This suite

consists of 30 benchmarks that perform numerical computations used in various domains such as linear algebra computations, image processing, physics simulation, etc. Polybench is used for experiments involving both serial and auto-parallelized code. The two largest datasets from Polybench are used to create the ML dataset. Based on experience, the variance of both the hardware performance counter values and the most suited code optimizer for the loop nests across the two datasets, was enough to treat them as two different loop nests. This variance can be attributed to two main reasons. First, a different set of optimizations being performed by the optimizers based on the built-in analytical models/heuristics that drive those optimizations, since properties like loop trip counts usually vary across datasets. Second, the performance across datasets on an architecture with a memory hierarchy, where the behavior of memory may change on one or more levels. This analysis was required to prevent the ML algorithms from *overfitting*.

4.2.6 Experimental Platforms and Data Collection

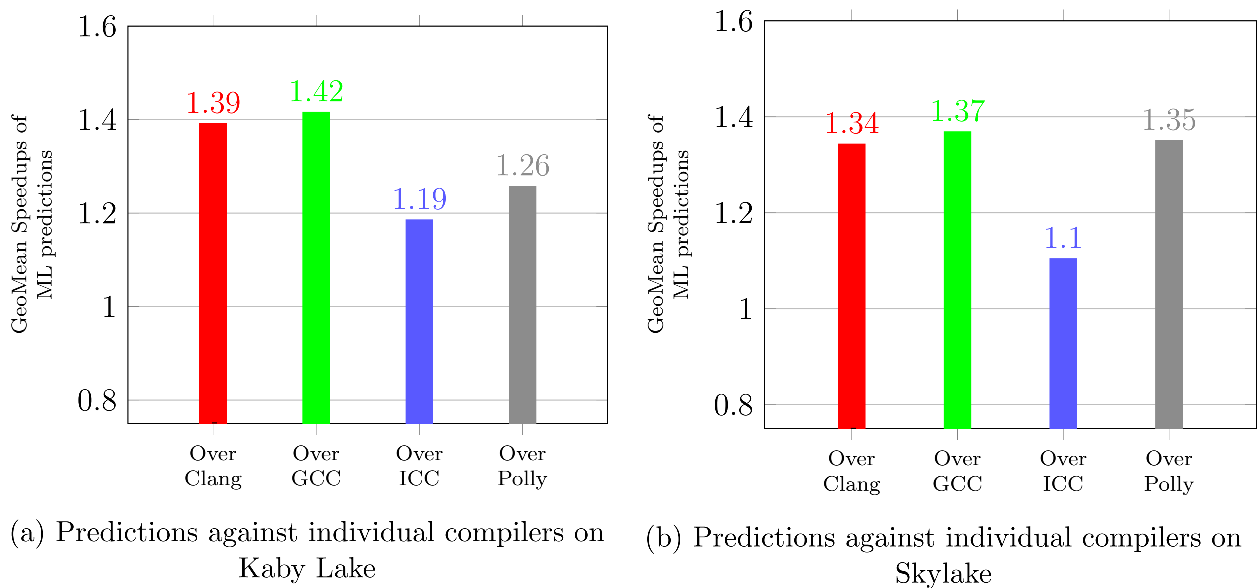
For the experiments, two recent Intel architectures are used. The first architecture is a four-core Intel Kaby Lake Core i7-7700K. This architecture supports Intel's SSE, AVX and AVX2 SIMD instruction set extensions. The second architecture is a two sixteen-core Intel Skylake Xeon Gold 6142. The Skylake architecture supports two more SIMD instruction set extensions, i.e., AVX-512CD and AVX-512F than the Kaby Lake architecture. For the auto-parallelization related experiments, only one thread is mapped per core.

Dynamic instruction count is skipped as a feature and normalize the rest of the hardware performance counters in terms of *per kilo instructions* (PKI). Loop nests that have low value for crucial hardware performance counters such as instructions retired are also skipped. From the experiments, two interesting correlations among hardware performance counters and the characteristic behavior of the loop nests were observed. First, the hardware performance

counters values from Kaby Lake architecture (after disabling loop transformations and vector code generation) were sufficient to get well trained ML model to make predictions for a similar architecture like the Skylake architecture. Second, for predicting the most suited candidate for serial code and for the auto-parallelized code for a loop nest, the same set of hardware performance counters, collected from profiling a serial version, can be used to train the ML model and achieve satisfactory results.

4.3 Evaluation of the Machine Learning Models

For evaluating the results, the speedup of ML predictions is calculated over candidate code optimizers, i.e., the speedup obtained if the code optimizer recommended by the ML model was used to optimize loop nests instead of a candidate code optimizer.



(a) Predictions against individual compilers on Kaby Lake

(b) Predictions against individual compilers on Skylake

Figure 4.1: Speedup of Predictions for Serial Code

		Predicted				
		Clang	GCC	ICC	Polly	
Actual	Clang	**5**	0	13	7	25
	GCC	1	**0**	13	2	16
	ICC	2	0	**96**	2	100
	Polly	2	0	15	**14**	31
		10	0	137	25	172

(a) Confusion Matrix for Kaby Lake

		Predicted				
		Clang	GCC	ICC	Polly	
Actual	Clang	**4**	4	14	1	23
	GCC	4	**10**	11	3	28
	ICC	4	5	**68**	4	81
	Polly	0	5	8	**15**	28
		12	24	101	23	160

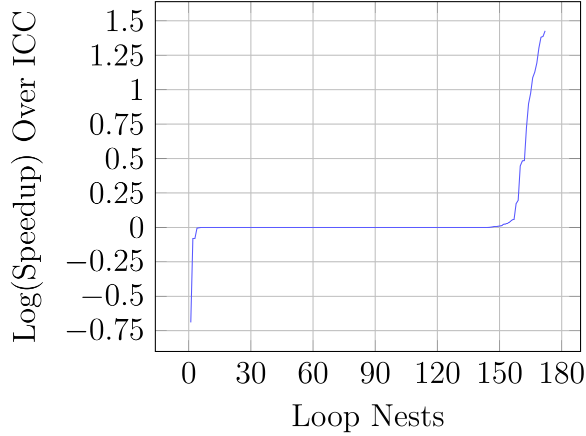
(b) Confusion Matrix for Skylake

Figure 4.2: Confusion Matrix for Serial Code Predictions

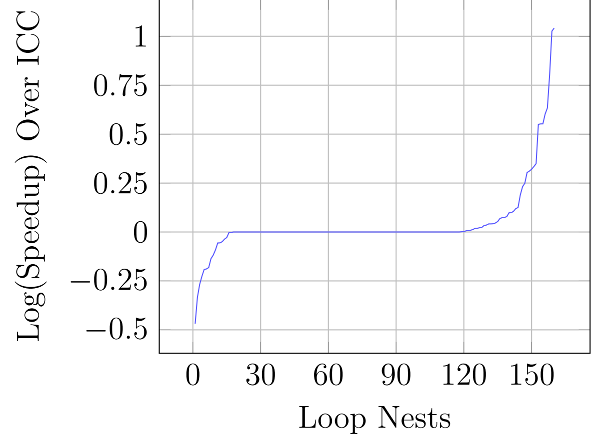
4.3.1 Predicting the Most Suited Code Optimizer for Serial Code

Fig. 4.1a and Fig. 4.1b show the results for the performance gains from the predictions for the Kaby Lake and Skylake architectures, respectively. These predicted gains can be viewed as the achievable headroom for each compiler. On the validation dataset, RF classifier predicted with an overall accuracy of 67% for Kaby Lake (with AUC as 0.71) and 61% for Skylake (with AUC as 0.68) as shown in the confusion matrices in Fig. 4.2a and Fig. 4.2b respectively.

In the confusion matrices, the main diagonal (in bold) represents the correct predictions, whereas the values outside the matrices represents the sum of values in the corresponding row or column. For example, the 67% overall accuracy for Kaby Lake is calculated as (5+0+96+14)/172. The 'Actual' target refers to the correct target, i.e., the code optimizer that produced the best performing code for the loop nests based on the comparative analysis, whereas the 'Predicted' target refers to the ML recommended target/code optimizer. For calculating the speedup of ML prediction over Clang, for example, the execution time of the



(a) Speedup over ICC for Kaby Lake



(b) Speedup over ICC for Skylake

Figure 4.3: Distribution of Predictions for Serial Code

ML predicted code optimizer is compared with the execution time of Clang. This speedup is calculated for all candidate code optimizers and for each loop nest in the validation dataset.

Across both architectures, Intel compiler performs well on majority of the loop nests. Therefore, the Majority Classifier predicted ICC with 58% overall accuracy for Kaby Lake and 50% overall accuracy for Skylake. The distribution of performance of the ML predictions compared to ICC are shown in Fig. 4.3a and Fig. 4.3b. The maximum performance gain on a loop nest was 27x, whereas the maximum slowdown was 0.2x.

4.3.2 Predicting the Most Suited Code Optimizer for Auto-Parallelized Code

For the auto-parallelization experiments, there are only two candidates: ICC and Polly. The RF classifier predicted with an overall accuracy of 85% for Kaby Lake (with AUC as 0.92) and 72% for Skylake (with AUC as 0.76) as shown in Fig. 4.4c. Since the validation dataset was well balanced for the two targets, the Majority Classifier produced an overall accuracy of 64% for Kaby Lake and 50% for Skylake. The distribution of performance of the ML

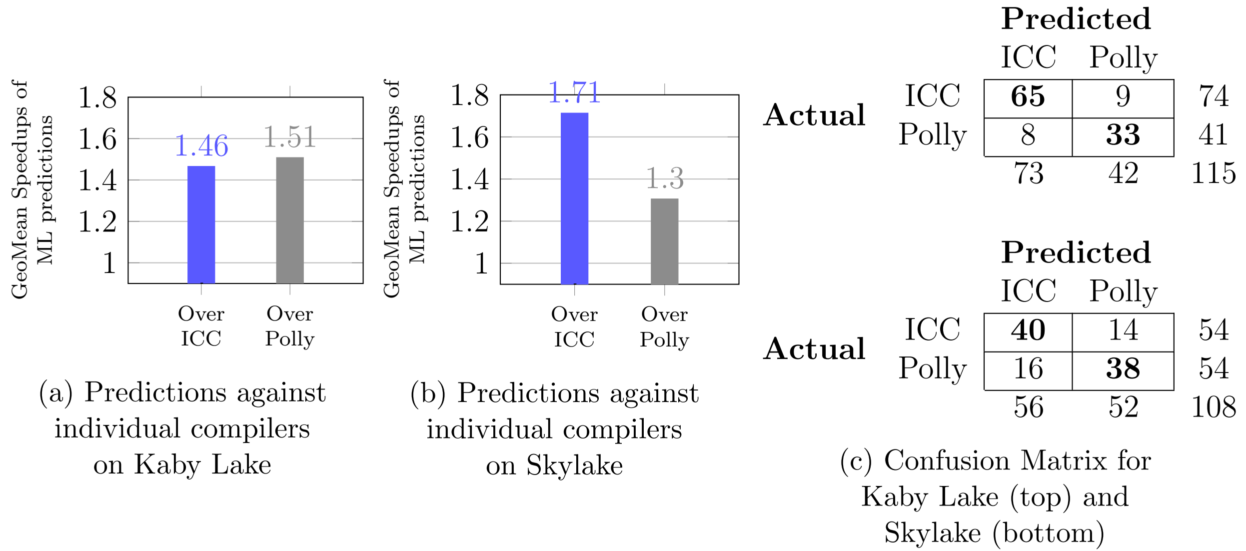


Figure 4.4: Speedup and Confusion Matrix of Predictions for Auto-Parallelized Code

predictions, when compared to ICC, are shown in Fig. 4.5a and Fig. 4.5b. The maximum gain on a loop nest was 91x whereas the maximum slowdown was 0.09x.

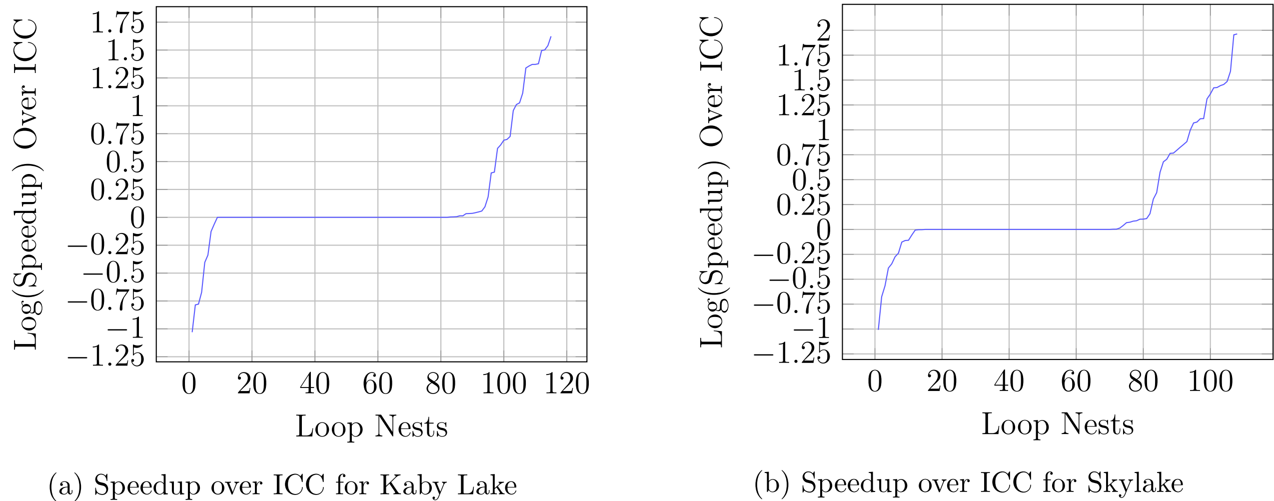


Figure 4.5: Distribution of Predictions for Auto-Parallelized Code

4.3.3 Overall Analysis and Discussion

The performance gain from the ML predictions over the candidate code optimizers range from 1.10x to 1.42x for the serial code and from 1.30x to 1.71x for the auto-parallelized code

across two multi-core architectures. Counters related to Cycles Per Instruction (CPI), D-TLB, memory instructions, cache performance (L1, L2 and L3) and stall cycles were crucial indicators of the inherent behavior of the loop nests.

On analyzing the validation datasets for serial code experiments, on an average for 95% of the loop nests, there was at least 5% performance difference between the most suited code optimizer and the worse suited code optimizer. For auto-parallelized code experiments, on an average for 91.5% of the loop nests, there was at least 5% performance difference between the most suited code optimizer and the worse suited code optimizer.

On the other hand, for the serial code experiments, for 68% of the loop nests, there was at least 5% performance difference between the most suited code optimizer and the second most suited code optimizer. That suggests that for the remaining 32% of the loop nests, it would be harder to make a distinction between the most suited code optimizer and the second one. Since the ML models' overall accuracy are 67% for Kaby Lake and 61% for Skylake, it can be inferred that they are doing very well on the loop nests that have a clear distinction about the most suited code optimizer.

4.4 An Explanation for why Hardware Performance Counters are good ML Features

Machine Learning algorithms learn from very complex selection and combination of features to make decisions, as explained in section 4.2.3 for the algorithm used in this work. Most helpful features for a Machine Learning algorithms are difficult to explain precisely, but understanding them can help infer their importance. Prior work that automatically generated features from the compiler's intermediate representation (IR) [87] has shown that Machine Learning algorithms do learn from features that may not be intuitive even to an expert

compiler writer. The most important features for the Machine Learning classifier are listed in Table 4.2 for serial code and in Table 4.3 for auto-parallelized code.

Some of the key reasons for why hardware performance counters as features are able to capture the characteristic behavior of loop nests are as follows. First, instruction count and stalls related to data movement such as load and store instructions retired, either scalar or vector, characterize traffic effect from the TLBs, caches and RAM. Second, stall cycle counters (for hardware resources) determine if the loop nest performance is limited by a particular resource. Third, L1, L2 and L3 hit/miss counters determine the memory footprint and provide information about the data access pattern. For example, the counters for a loop nest with a stride-1 access pattern have lower L1 and/or L2 cache misses than a loop nest with larger or non-linear strides. Stride-1 access also correlates with vectorizability. Also, the hardware prefetchers are stride 1 or next line, resulting in further latency reduction. Lastly, instructions per cycle retired for arithmetic operations on different data types such as Floating Point (both single and double precision) or Integer provide information about the throughput/latency of computations.

The code optimizers perform a set of optimizations/loop transformations that are based on properties of the loop nest. For example, different code optimizers may choose to unroll the innermost loop by different factor or just not unroll at all based its loop trip count and memory access pattern. Such properties of the loop nests are captured by the hardware counters. Similarly, a lot of cache misses at L1 and L2 level, may suggest transformations like loop interchange or loop tiling can be beneficial. Another example would be if L1 and L2 have higher cache misses than L3, then loop interchange, if possible, could benefit towards getting better performance. Transformations, such as unrolling, tiling and interchange, may also lead to vector code generation that often has high impact on performance. So counters do correlate with potentially beneficial transformations and one compiler may perform such transformations or combinations thereof better than another.

Top 30 features for Serial Machine Learning Classifier
MEM_LOAD_RETIRED.L3_MISS_PS
OFFCORE_REQUESTS_OUTSTANDING.CYCLES_WITH_DEMAND_RFO
CYCLE_ACTIVITY.STALLS_L3_MISS
OFFCORE_REQUESTS_BUFFER.SQ_FULL
CYCLE_ACTIVITY.STALLS_L1D_MISS
MEM_LOAD_RETIRED.L3_HIT_PS
L1D_PEND_MISS.PENDING
RS_EVENTS.EMPTY_CYCLES
OFFCORE_REQUESTS_OUTSTANDING.ALL_DATA_RD:cmask=4
L2_RQSTS.RFO_HIT
FP_ARITH_INST_RETIRED.SCALAR_DOUBLE
DTLB_STORE_MISSES.WALK_ACTIVE
CPU_CLK_UNHALTED.THREAD
MEM_LOAD_RETIRED.L1_MISS_PS
CYCLE_ACTIVITY.STALLS_L2_MISS
FP_ARITH_INST_RETIRED.SCALAR_SINGLE
DTLB_LOAD_MISSES.WALK_ACTIVE
UOPS_DISPATCHED_PORT.PORT_6
MEM_LOAD_RETIRED.L1_HIT_PS
MEM_INST_RETIRED.ALL_STORES_PS
IDQ.MS_UOPS
EXE_ACTIVITY.1_PORTS_UTIL
MEM_INST_RETIRED.STLB_MISS_LOADS_PS
DTLB_LOAD_MISSES.STLB_HIT
FRONTEND_RETIRED.LATENCY_GE_2_BUBBLES_GE_1_PS
DSB2MITE_SWITCHES.PENALTY_CYCLES
CYCLE_ACTIVITY.STALLS_MEM_ANY
DTLB_STORE_MISSES.STLB_HIT
UOPS_ISSUED.ANY
MEM_LOAD_RETIRED.L2_HIT_PS

Table 4.2: Top Ranking ML Features for Serial Code Predictions.

Top 30 features for Auto-Parallel Machine Learning Classifier
CPU_CLK_UNHALTED.THREAD
IDQ.MITE_UOPS
CYCLE_ACTIVITY.STALLS_L1D_MISS
LD_BLOCKS_PARTIAL.ADDRESS_ALIAS
RS_EVENTS.EMPTY_CYCLES
CYCLE_ACTIVITY.STALLS_MEM_ANY
MEM_INST_RETIRED.STLB_MISS_LOADS_PS
INST_RETIRED.PREC_DIST
UOPS_EXECUTED.THREAD
IDQ.ALL_MITE_CYCLES_ANY_UOPS
L1D_PEND_MISS.FB_FULL:cmask=1
EXE_ACTIVITY.2_PORTS_UTIL
DTLB_LOAD_MISSES.STLB_HIT
CYCLE_ACTIVITY.STALLS_L2_MISS
IDQ.ALL_DSB_CYCLES_4_UOPS
MEM_LOAD_RETIRED.L3_MISS_PS
CYCLE_ACTIVITY.STALLS_L3_MISS
EXE_ACTIVITY.1_PORTS_UTIL
FRONTEND_RETIRED.DSB_MISS_PS
MEM_LOAD_RETIRED.L2_HIT_PS
FP_ARITH_INST_RETIRED.SCALAR_DOUBLE
DTLB_STORE_MISSES.STLB_HIT
MEM_INST_RETIRED.STLB_MISS_STORES_PS
UOPS_DISPATCHED_PORT.PORT_5
EXE_ACTIVITY.BOUND_ON_STORES
IDQ.MS_UOPS
MEM_LOAD_RETIRED.L1_MISS_PS
IDQ.DSB_UOPS
INT_MISC.RECOVERY_CYCLES
L1D_PEND_MISS.PENDING

Table 4.3: Top Ranking ML Features for Auto-Parallelized Code Predictions.

4.5 A Framework for Improving Performance using Machine Learning Predictions

The framework for choosing the most suited code optimizer for loop nest using ML prediction is shown in Fig. 4.6. The goal is to eliminate the time-consuming Exploratory Search step of the framework and use the ML prediction to select the best code optimizers during compilation. The ML predictions are used to predict the most suited code optimizer for both serial, auto-vectorized code as well as auto-parallelized code.

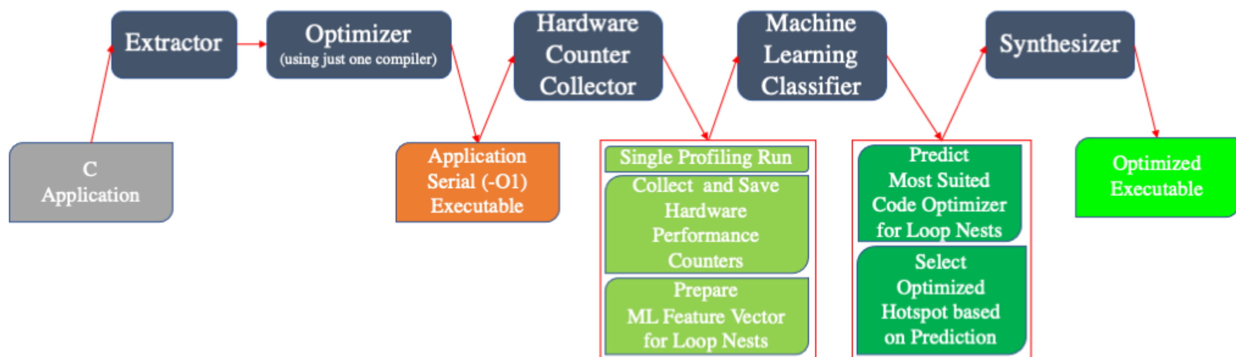


Figure 4.6: *MCompiler* Framework with Machine Learning Predictions

The architecture of the *MCompiler* framework is modified in the following ways for ML prediction of the most suited code optimizer for a loop nest. First, the Optimizer now generates an executable that is compiled by the default compiler for serial execution with `-O1` optimization level. Second, the Exploratory Search Engine is replaced by the Hardware Counter Collector for making ML predictions.

The Hardware Counter Collector executes the serial (`-O1`) code and collects hardware performance counters for each loop nest. This is done only once using the default compiler, in contrast to the exploratory search that has to run every compiler. As mentioned earlier, using the `-O1` version of the loop nest helps in preserving the inherent code characteristics, since the generated code is not optimized using complex code transformations, and is similar across compilers, therefore providing a good common baseline. If a loop nest is not executed or the hardware performance counters are not present (e.g. for loop nests with very few computations), the default compiler is chosen by the Synthesizer.

Next, the collected hardware performance counters for each loop nest are transformed into the feature vector, i.e., the input to the ML classifier. These hardware performance counters are also saved in a CSV file for future reference of the users. Third, the ML classifier makes the prediction for the most suited code optimizer for a loop nest based on the feature vector. The ML classifier is a trained ML model. There are two separately trained ML models, one for serial code predictions while the other is for parallel code predictions. Finally, these predictions from the ML classifier are forwarded to the Synthesizer, which uses the code from the predicted optimizer and links the selected optimized loop object files and generates the final executable for the application.

The *MCompiler* driver invokes the ML prediction part of the framework over the original *MCompiler* flow with exploratory search if the `--predict` flag is set. The ML models are trained and incorporated in the *MCompiler* framework using OpenCV's Machine Learning module [103].

4.5.1 Collecting Hardware Performance Counters for the Loop Nests

The features, i.e., the hardware performance counters used for the Machine Learning models, are collected by profiling loop nests using Intel's VTune Amplifier, although other tools may be used, e.g. Linux `perf`. The Intel compiler is used to generate the executable that is then used for profiling. All loop optimizations are disabled during this compilation by using the `-O1` flag. In addition to that, the optimizations that are responsible for vector code generation and parallel code generation are disabled too. The profiling information, therefore, provides an insight into the characteristics of the loop nests while eliminating the influence of compiler transformations and behavioral changes incurred from special architectural features of the underlying architecture. The performance counters that are collected include, but not limited to, instruction-based (instruction types and counts) counters, CPU clock cycles-based (including stalls) counters and memory-based (D-TLB, L1 cache, L2 cache, L3 cache) counters.

Once the hardware performance counters are collected for the loop nests, the dynamic instruction count is not used as a feature. The other hardware performance counters are normalized in terms of *per kilo instructions* (PKI). Based on the analysis, this allows the Machine Learning models to learn about the inherent characteristics of the loop nests and not bias them towards characteristics such as loop trip count.

4.6 Evaluation of the *MCompiler* with Machine Learning Prediction

The training dataset for training the serial code classifier included loop nests from TSVC and Polybench benchmark suites and had a total of 274 instances (loop nests). The loop nests from NAS Parallel Benchmarks (NPB) were not included in the training dataset. Therefore, the experimental results for the *MCompiler* performance with ML predictions are shown for NPB benchmarks only.

The auto-parallelized code classifier was trained using the training dataset, which included loop nests from Polybench benchmark suite and has 194 instances (loop nests). Again, the experimental results for the *MCompiler* performance with ML prediction are shown for NPB benchmarks only, since these loop nests were not used in training the ML model. The reason for choosing benchmark suites such as Polybench and TSVC for creating the training dataset was to expose the ML models to a diverse set of loop nests that exhibit different characteristics. The specifics for creating the training datasets, characteristics of the training dataset and evaluating the models are similar to this work.

The properties of the trained RF classifier are as follows. Maximum depth of the tree was set at 25 after analyzing that the model was neither underfitting or overfitting on cross-validation. The maximum sub-categories were set at 15. The minimum sample count at the leaf node was set at 5. Lastly, the size of the randomly selected subset of features at each tree node that are used to find the best split was set at 20.

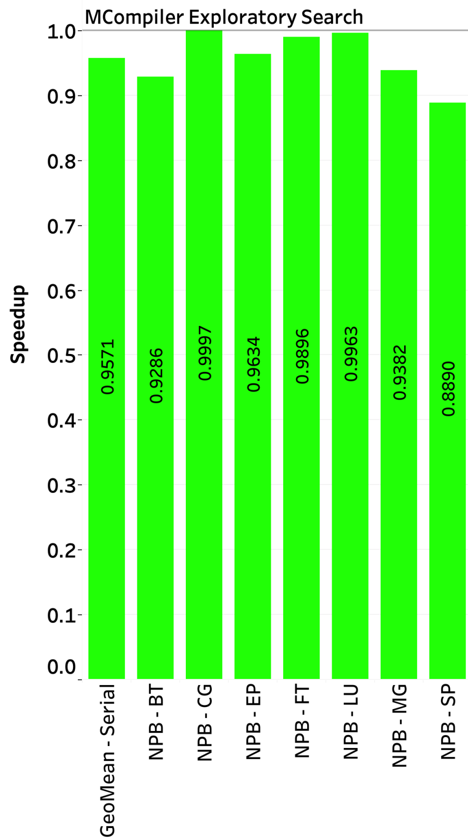
The serial code classifier targets (e.g. most suited code optimizers) were `clang`, `gcc`, `icc` and `Polly`. The auto-parallelized code classifier targets were `icc` and `Polly`.

The source-to-source code optimizer (Pluto) is left out as a target code optimizer since it requires another compiler to generate code and creates noise for ML models in cases where

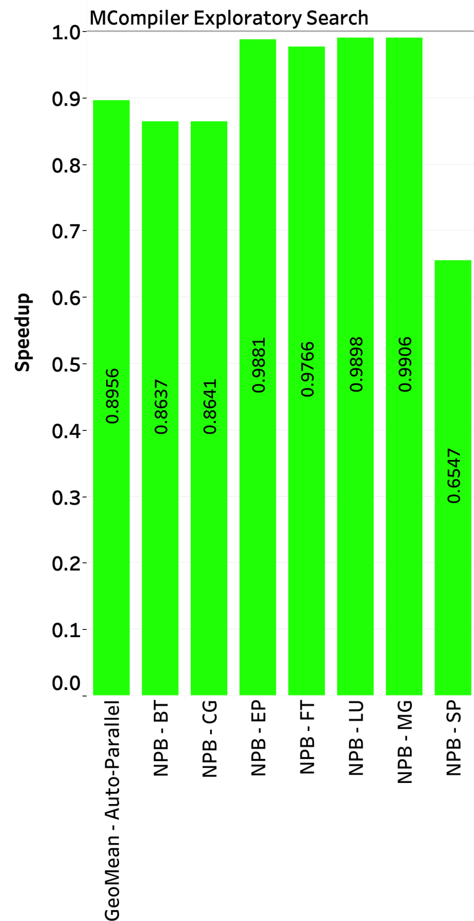
the performance benefits are not significant from the source-level transformations. In the training dataset, the target for the instances with `Pluto` as the most suited code optimizer were replaced by the second best code optimizer. The evaluation of ML models showed that replacing `Pluto` as a target class improved both AUC and classification accuracy. The reason why this strategy worked in improving the ML models is as follows. The lack of enough instances in the training dataset with target class being `Pluto`, i.e., not enough loop nests with best code optimizer being `Pluto`. As previously shown in Chapter 3, Fig. 3.4 (Distribution of best performing code per Code Optimizer). The training dataset is skewed towards `icc` as the target class due to `icc` being chosen as the most suited code optimizer. This leads to an unbalanced dataset where minority classes have a very small share of the training instances and therefore are a major challenge for ML classifiers [141, 72, 142, 61]. Removing `Pluto` improved the share of other minority classes such as `clang`, `gcc` and `Polly`. Hence, improving both performance measures, AUC and classification accuracy, significantly.

Since `Polly` optimizations/passes run along with the standard `LLVM` pass pipeline, `Polly` is considered as the most suited code optimizer only when it shows at least 5% performance improvement over `clang`. There are two reasons for setting a 5% performance improvement threshold before attributing a loop nest to `Polly` over `clang`. First, from an ML point-of-view, if there are two very similar loop nests (that will lie very close in a multi-dimensional feature space), one has `Polly` as the target class whereas the other has `clang`, then a ML algorithm will try to overfit in order to reduce the classification error, while the actual impact on the performance will be quite minimal. Second, a 5% execution time difference could just be a time variance between multiple runs, i.e., experimental error.

The ML models are not trained to predict the most suited code optimizer for the OpenMP regions for primarily one reason: the code optimizers lose flexibility to optimize the OpenMP regions as mentioned before.



(a) Serial Benchmarks



(b) Auto-Parallelized Benchmarks

Figure 4.7: *MCompiler* + ML Predictions Performance for Serial and Auto-Parallelized Benchmarks

4.6.1 Serial Code

The performance results for ML predictions are shown in Fig. 4.7a relative to the exploratory search. The most predicted code optimizer was `icc` (45%), followed by `clang` (34%). The GeoMean performance loss over the exploratory search is 4.3%.

The case with the highest performance loss compared to the Exploratory Search is the SP benchmark where 72 predictions were made. In the predicted code optimizer for SP loop

nests, the distribution of `clang` increased to 43% compared to the Exploratory Search's 16%. Whereas, the distribution of `gcc` decreased to 12% from Exploratory Search's 26%. This would explain the drop in performance for SP compared to the Exploratory Search version.

4.6.2 Auto-Parallelized Code

The performance results for ML predictions are shown in Fig. 4.7b relative to the exploratory search. The most predicted code optimizer was `polly` (60%) and the rest was `icc` (40%). For BT and SP benchmarks, `Polly` was predicted as the most suited code optimizer for multiple loop nests used in computing the right hand side (rhs), whereas `icc` was chosen as the most suited code optimizer by the exploratory search for those loop nests. The mis-predictions from the ML classifier were found to have a larger impact on performance when most of the execution time is dominated by one or very few kernels. The effect of a mis-prediction can thus be significantly magnified. One such case is the CG benchmark, where 60% of the execution time is spent in one loop nest and `polly` was mis-predicted as the most suited code optimizer for that loop nest instead of `icc`. Whereas the other loop nest that covered 37% of the execution time was correctly predicted as `polly`. The impact of mis-predictions is, in general, higher for auto-parallelized code as compared to serial code. Still, the Geometric Mean of performance loss over the exploratory search is rather small - 10.5%.

4.7 Summary

This chapter shows that it is possible to learn about the inherent characteristics of the loop nests by using the hardware performance counters as features for the Machine Learning algorithms. These Machine Learning models are then used to predict which code optimizer (with its transformations and cost models) would be the most beneficial for a loop nest.

The *MCompiler* framework is expanded to incorporate these Machine Learning models. The use of Machine Learning prediction achieves performance very close to the exploratory search for choosing the most suited code optimizer: within 5% for auto-vectorized code and within 11% for auto-parallelized code.

Chapter 5

Applying the Multiple Compiler Approach to Improve Energy Efficiency

In this chapter, first, the relationship between optimizing for performance using several compiler and the improvement in energy efficiency is studied. Second, the Exploratory Search method of the *MCompiler* is expanded to optimize applications for better energy efficiency and its results are presented.

5.1 Optimizing for Energy Efficiency on Modern Architectures

Compilers have traditionally focused on optimizing for performance. And optimizing for performance on modern architectures oftentimes improves energy consumption too, although by different factors [107, 52, 70, 30]. The goals for optimizing applications, in terms of

energy or power, differ from domain to domain and may even differ from user to user. Energy consumption for different processors (even from the same architecture) is determined by features, such as number of cores, and dynamic parameters, such as Dynamic Voltage Frequency Scaling (DVFS). These dynamic parameters cannot be modeled while doing static compilation, hence implementing energy efficiency driven compiler optimization may not provide guaranteed results. Production compilers optimize for performance, with various optimization levels focused towards generating better performance, but no such optimization levels for better energy efficiency, and understandably so.

Prior work on optimizing applications for energy consumption using compilers falls into two categories. First, prior work [44, 149, 73, 75, 109, 137, 76, 151, 152, 53, 120, 138, 107, 55, 89, 52, 70] that explored the impact of compiler optimizations on application execution times and energy consumption for the CPU and memory, some focusing on loop nest transformations. Second, research [64, 122, 63, 31, 127, 96, 121] that explored improving the energy consumption for applications by managing software and hardware knobs, such as Dynamic Voltage Frequency Scaling (DVFS), active core counts, degree of parallelism, instruction set selection, etc.

The approach used in this chapter falls in the first category and focuses on performance and energy consumption of loop nests instead of entire applications. However, instead of looking at the impact of specific loop nest transformations or specific compilers on energy consumption, as studied in [149, 75, 138, 52], this chapter examines the overall impact of the sequence of loop nest transformations in *several* compilers. Other prior works have also looked at various optimization levels of a specific compiler [137, 70], auto-tuning frameworks [53, 120, 138, 32] and at the influence of two compilers on energy consumption for entire application [70]. Few previous studies [29, 58, 30, 18] have looked at energy consumption of different vector lengths for manually vectorized codes for multi-core architectures, but this chapter also studies the auto-vectorization capabilities of the compilers too.

Finally, the Exploratory Search method of the *MCompiler* is expanded to optimize applications for better energy efficiency by choosing the most energy efficient version possible for each loop nest. The metrics for judging the energy efficiency for a loop nest is chosen as Energy-Delay-Product (EDP). The results show that loop nest optimizations oriented towards performance may not always have the same impact on energy consumption and these differences vary from loop nest to loop nest. The Exploratory Search allows for the architectural intricacies and dynamic parameters, such as DVFS, taken into account before generating the optimized binary. It uses performance-oriented loop nest optimizations to produce energy efficient code, in this case *the end does justify the means.*

5.2 Impact of Performance-Oriented Loop Nest Optimizations on Energy Efficiency

This section presents a comparative study of state-of-the art compilers and studies the correlation of loop nest optimizations in various compilers on performance and energy efficiency, for Intel Skylake architecture. The results show that over a large diverse set of loop nests, if we were to pick the most energy efficient version for each of the loop nests, optimized using GNU GCC, LLVM Clang and a LLVM based domain specific optimizer (Polly), EDP can be reduced by a GeoMean of 1.71x over the Intel compiler. EDP is an established metric to analyze the trade-off between execution time and energy consumption, which takes into account the trade-off between (slower) performance and (lower) energy consumption.

Results show that the quality of generated code from different compilers has a significant impact on both execution times and energy. For instance, 79% of loop nests exhibit more than 10% EDP difference for Intel compiler generated code compared to code from other compilers. Also, results for 13% of the loop nests show that the best performing code

version (in terms of execution time) was not same as the most energy efficient version. This demonstrates that performance oriented optimizations does not always entail energy efficiency.

A processor core today includes the capability to process Single Instruction Multiple Data (SIMD) or Vector instructions. Most modern processors support vector processing capabilities, e.g. Intel AVX, ARM Neon, IBM VSU, AMD AVX, etc. Compilers attempt to automatically vectorize code for these processors as this improves performance. Another goal of this work is to better understand the auto-vectorizing ability of the compilers. Vector instructions are available with different vector lengths on recent Intel architectures (128-bit, 256-bit and 512-bit). Some 256-bit (AVX-2) and 512-bit (AVX-512) instructions (e.g. arithmetic operations) *may* cause the processor to run at different and reduced maximum frequencies[65, 125]. Therefore, the maximum available vector length may not be the most energy efficient choice. The results show that by choosing the right vector length for each of the loop nests, rather than letting the compiler's performance oriented cost models decide, can reduce the EDP by a GeoMean of 1.39x. Even after processor reduces frequency for large vectors to save energy, results show that in 15% of the loop nests the code with best performing vector length was not same as the most energy efficient version.

Experimental Platform

The target architecture used is a two socket, sixteen-core Intel Skylake Xeon Gold 6142 @ 2.6 GHz (with Turbo Boost @ 3.7GHz). Each Xeon processor has 32KB L1 cache, 1MB L2 cache, 22MB L3 cache. The Skylake architecture supports Intel's SSE, AVX, AVX-2, AVX-512CD and AVX-512F vector instruction set extensions. CPU Hyper Threading (SMT) is turned off, cores are operating at the maximum frequency and C-state setting is adjusted so as to reduce the leakage power from idle cores. An application was pinned to the last core of the processor in the experiments. On this processor, 'heavy' AVX-2 (256-bits) and AVX-512

(512-bits) instructions may lower processor frequencies [65] but these changes are minimal when there is only one active core on the processor, which is the case for the experiments presented in the next section. Table 5.1 shows the variation in maximum core frequency on using different vector extensions and additional reduction in maximum core frequency on increasing number of active cores.

Vector Extension	**Base Core Frequency (GHz)**	**# of Active Cores / Max Core Frequency in Turbo Mode (GHz)**				
		1-2	**3-4**	**5-8**	**9-12**	**13-16**
Non-AVX	2.6	3.7	3.5	3.4	3.4	3.3
AVX-2	2.2	3.6	3.4	3.3	3.2	2.9
AVX-512	1.6	3.5	3.3	2.8	2.4	2.2

Table 5.1: Change in Maximum Core Frequency with different Vector Extensions for a sixteen-core Intel® Xeon® Skylake Gold 6142 processor.

Compilers

With the flags mentioned in the table, compilers are allowed to use their cost models to make the best decision for the target architecture. ICC with `-Ofast -xHOST` is the baseline for all the experiments presented in this work. For the auto-vectorization experiments with different vector lengths, flags are added to force the Intel compiler to generate vector code, whenever possible, for a specific vector length. The analysis shows the energy efficiency of ICC generated code for AVX-512 (512-bits), AVX-2 (256-bits) and the Scalar version with respect to default ICC flags (`-Ofast -xHOST`). The default ICC flags allow the compiler to have the freedom to choose between the AVX-512, AVX-2 or the Scalar version based on the profitability predicted by the in-built cost models.

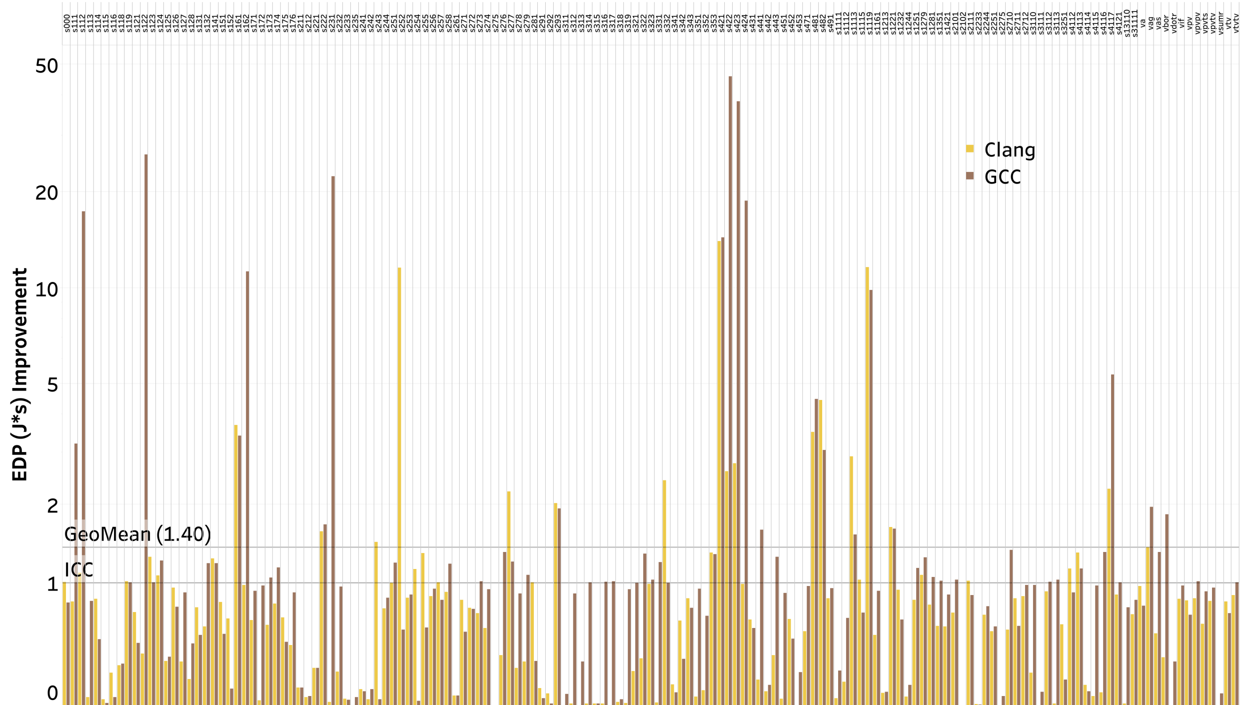


Figure 5.1: EDP comparison between different compilers w.r.t. ICC for TSVC. Y-axis is log scaled. Improvement in EDP means the factor of reduction in EDP compared to the EDP of ICC generated code.

Benchmarks

The benchmarks chosen for the evaluation present a wide variety of unique loop nests to test compilation strategies. One is Test Suite for Vectorizing Compilers (TSVC) by Callahan et al. [25].This benchmark contains 151 different loop nests that test loop transformation and auto-vectorization capabilities of the compilers. The analysis shows the performance and energy efficiency of Clang and GCC generated code for TSVC loop nests with respect to ICC's. TSVC is also the focus for this study of the energy efficiency of different vector lengths. The second benchmark suite used is Polybench/C 4.1.This suite consists of 30 benchmarks that perform numerical computations used in various domains, such as linear algebra computations, stencils, image processing, physics simulation, etc. This main kernels (set of loop nests) of these benchmarks are commonly found in other popular benchmark suites. For Polybench loop nests, the results from Polly (LLVM based domain specific loop

optimizer) are shown too in this comparative study. Polly is not suited for TSVC loop nest, hence Polly results are only presented for Polybench related experiments. Overall, the compilers are exposed to 181 unique loop nests with aim of exploring how well they optimize loop nests and their impact on energy efficiency.

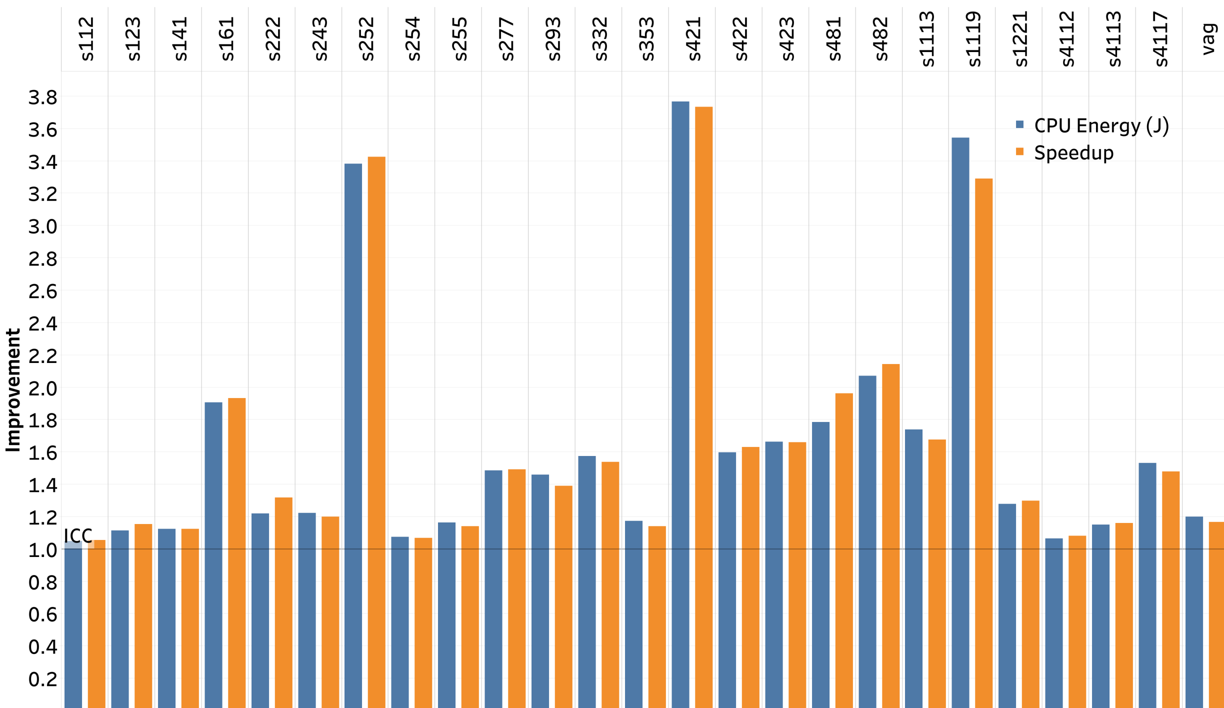


Figure 5.2: Comparison of CPU Energy and Speedup between different Clang and ICC for TSVC loop nests. Only cases with more than 10% EDP improvement are shown.

5.3 Evaluation of Different Compilers in Terms of Energy Efficiency

This section presents results for performance, CPU and DRAM energy, and Energy-Delay-Product of the four compilers and analyzes the results.

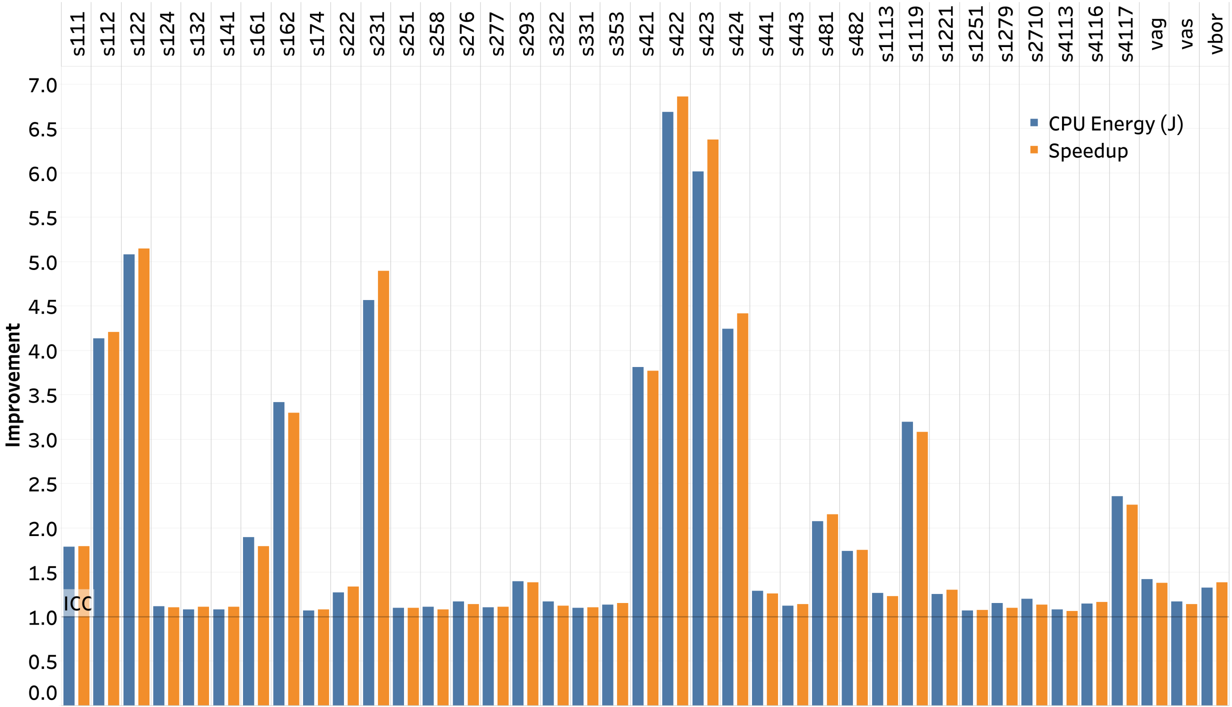


Figure 5.3: Comparison of CPU Energy and Speedup between different GCC and ICC for TSVC loop nests. Only cases with more than 10% EDP improvement are shown.

5.3.1 Loop Nests Optimized by Different Compilers

Data was collected from each compiler initially and then the relative change between compilers was examined. This allowed us to observe and compare, first, how much performance change for the same loop nest a different compiler may give and, second, if this change in runtime also translates into change in energy consumption. The Intel compiler (`icc`) was chosen as the baseline compiler because its performance was better than the other compilers for 55% of the loop nests. The relative performance and energy consumption results for `clang` and `gcc` against `icc` for TSVC are shown in Fig. 5.1, Fig. 5.2 and Fig. 5.3. Similarly the results for `clang`, `gcc` and `polly` against `icc` for Polybench are shown in Fig. 5.4, Fig. 5.5, Fig. 5.6 and Fig. 5.7.

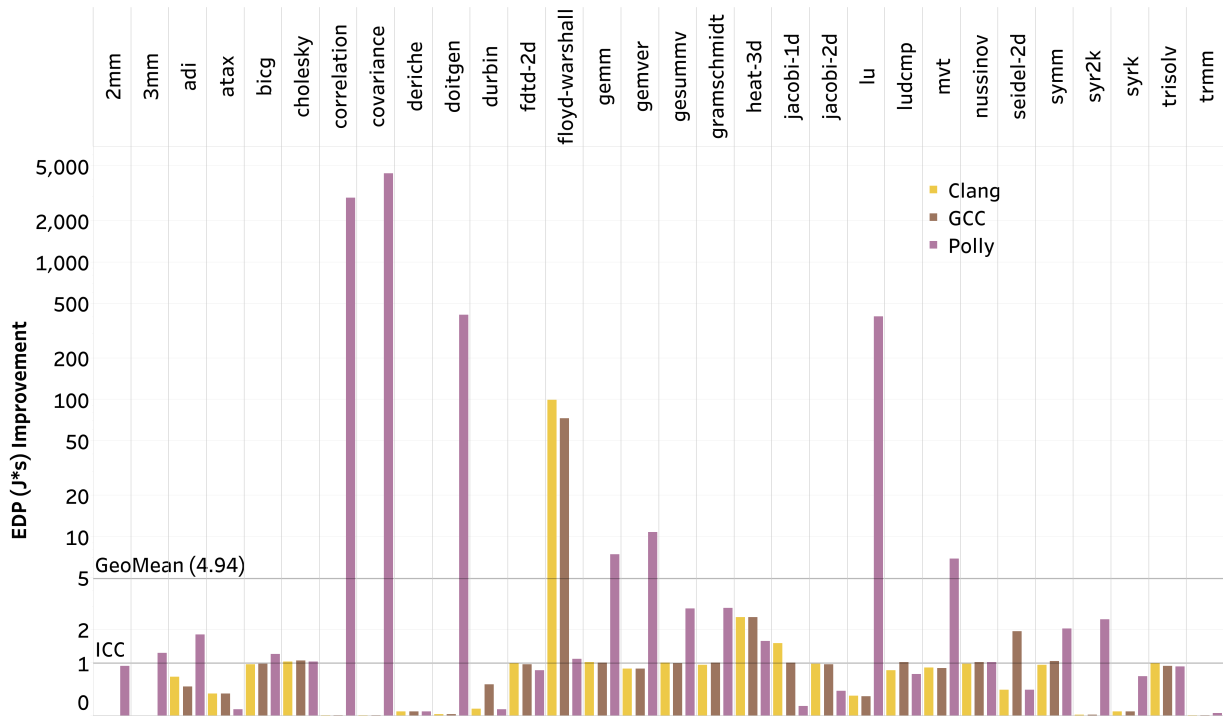


Figure 5.4: EDP comparison between different compilers w.r.t. ICC for Polybench. Y-axis is log scaled.

5.3.2 Reduction in EDP when Selecting the Most Energy Efficient Version

The results in Fig. 5.1 and Fig. 5.4 show that if one were select the most energy efficient code for each of the loop nest, EDP can be reduced by, GeoMean of, 1.4x for TSVC and 4.94x for Polybench (1.71x combined) over the Intel compiler. TSVC loop nests that do not have any DRAM access, unlike Polybench loop nests where doing cache and locality optimization have a huge impact on performance and energy consumption for CPU and DRAM. Hence, for Polybench there are large performance improvements when using a domain specific code optimizer, i.e., Polly that specializes in doing optimizations such as Loop Tiling, Interchange and Skewing that improve data locality and expose SIMD parallelism. For TSVC, certain loop transformations and generating the best code by selecting the optimal unroll factor and SIMD length have huge impact on performance. This shows the impact of loop nest

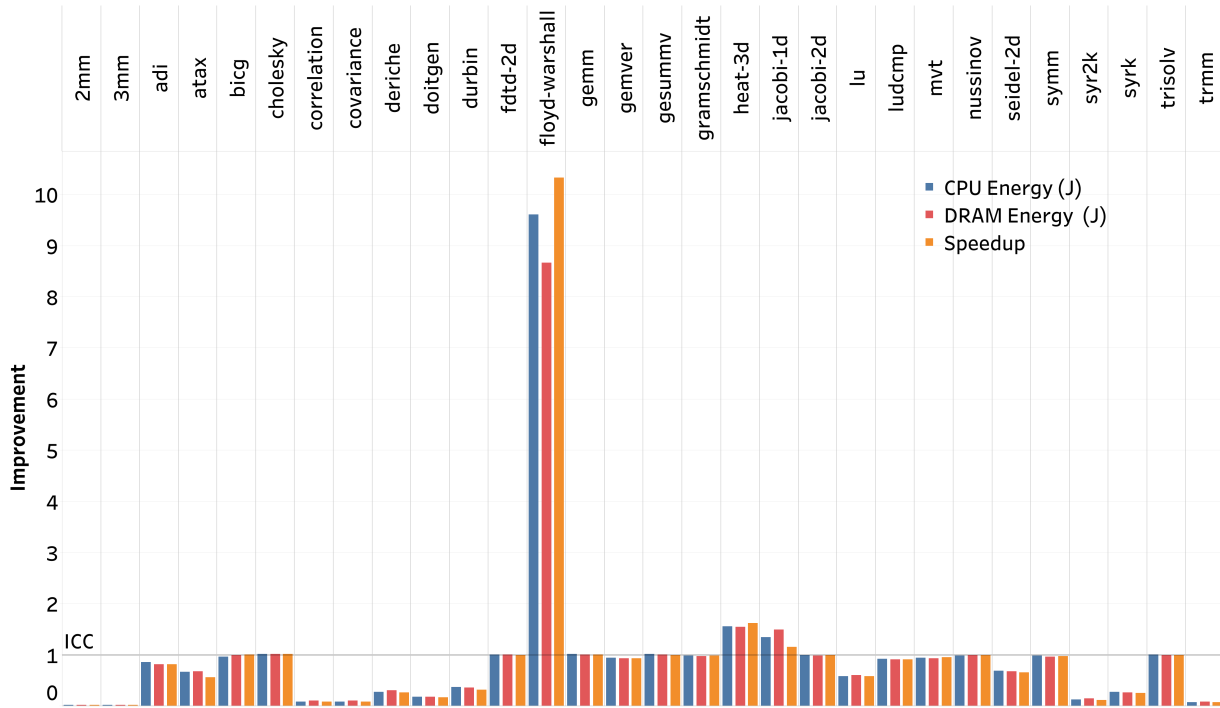


Figure 5.5: Comparison of CPU Energy, DRAM Energy and Speedup between Clang and ICC for Polybench loop nests.

optimizations and the quality of code generation by different compilers on performance as well the energy consumption for the modern processors. The change in EDP was more than 10% for 79% of the loop nests, when they were optimized by different compilers. For 13% of the loop nests the best performing code version (in terms of execution time) was not same as the most energy efficient version. Therefore, although most compilers are guided by performance oriented optimization, they do have beneficial impact in terms of energy efficiency for most but not all cases.

Analysis

This section presents the analysis of few interesting cases that show how differently each compiler could optimize the same loop nest. In S241 (TSVC), `icc` performs loop distribution, scalar expansion, strip-mine by a factor of 64 and finally partially vectorize the loop

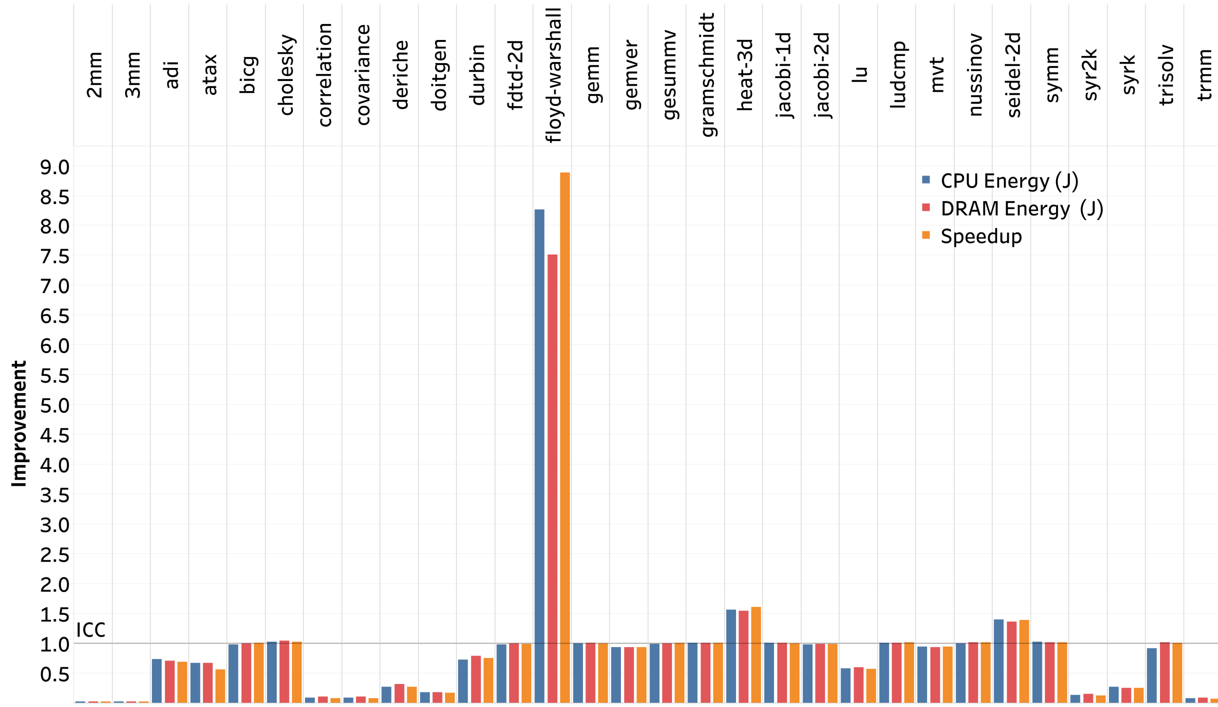


Figure 5.6: Comparison of CPU Energy, DRAM Energy and Speedup between GCC and ICC for Polybench loop nests.

nest, whereas `gcc` and `clang` didn't do those optimization to expose SIMD parallelism. In S231 (TSVC), `gcc` do loop interchange to expose SIMD parallelism, whereas `icc` and `clang` generated scalar code with loop unrolled by factor 2 and 4 respectively. In S252 (TSVC), `clang` used optimizations such as scalar and array expansion to vectorize the inner most loop nest, whereas other two compilers generated scalar code. In S423 and S424 (TSVC), `gcc` unrolling the loop nests with a different factor than `icc` contributed to the improvements. For correlation and covariance from Polybench, in both bases the main kernel had 3-4 loop nests. `icc` was able to optimizes individual loop nest by using loop interchange, distribution, etc. and generate vectorized code, whereas `polly` was able to fuse multiple loop nests which improved locality and removed redundant operations. For floyd-warshall (Polybench), both `gcc` and `clang` generated vectorized integer add and minimum instructions, whereas `icc` performed loop interchange which inhibited generation of vectorized instructions.

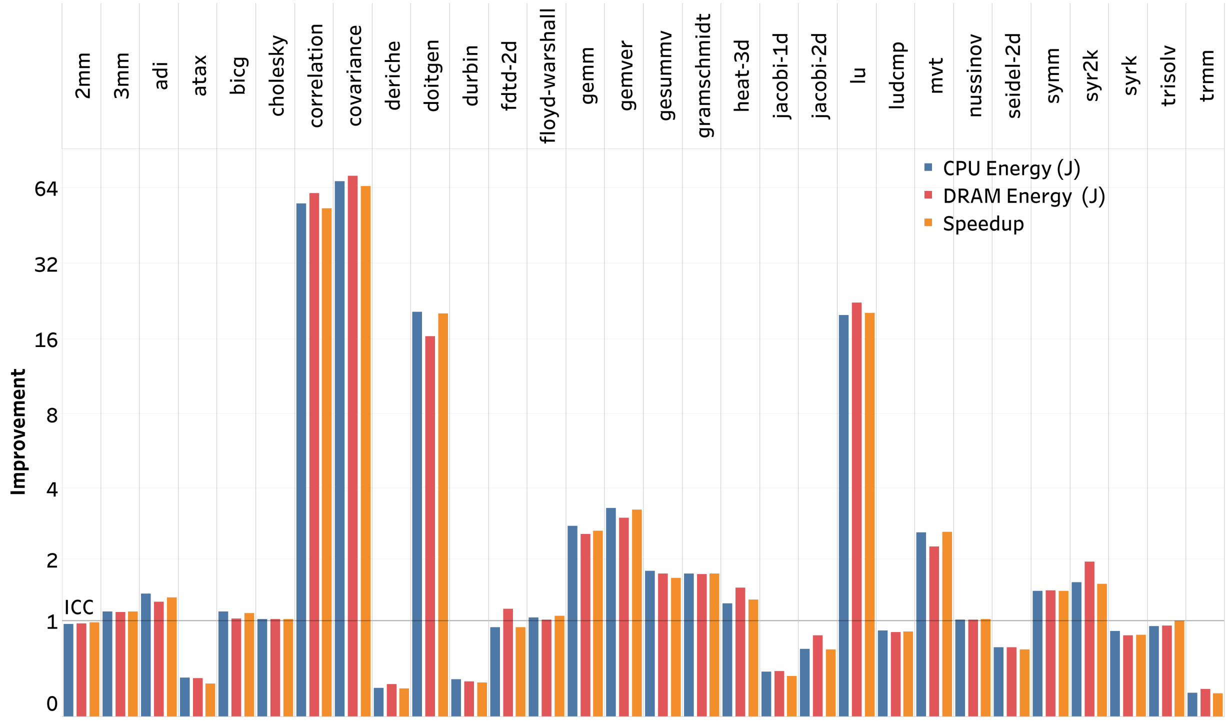


Figure 5.7: Comparison of CPU Energy, DRAM Energy and Speedup between Polly and ICC for Polybench loop nests. Y-axis is log scaled.

5.4 Performance and Energy Consumption Implications of using Different Vector Extensions

This section examines the selection of the target vector extension and its impact on energy efficiency.

5.4.1 Compiler's Ability to Auto-Vectorize and Impact of Selecting the Best Vector Length

The impact of auto-vectorization techniques and vector code generation capabilities implemented in the Intel compiler are analyzed here. Only results for `icc` are shown because more loop nests optimized by `icc` had greater than 5% performance improvement from vectoriza-

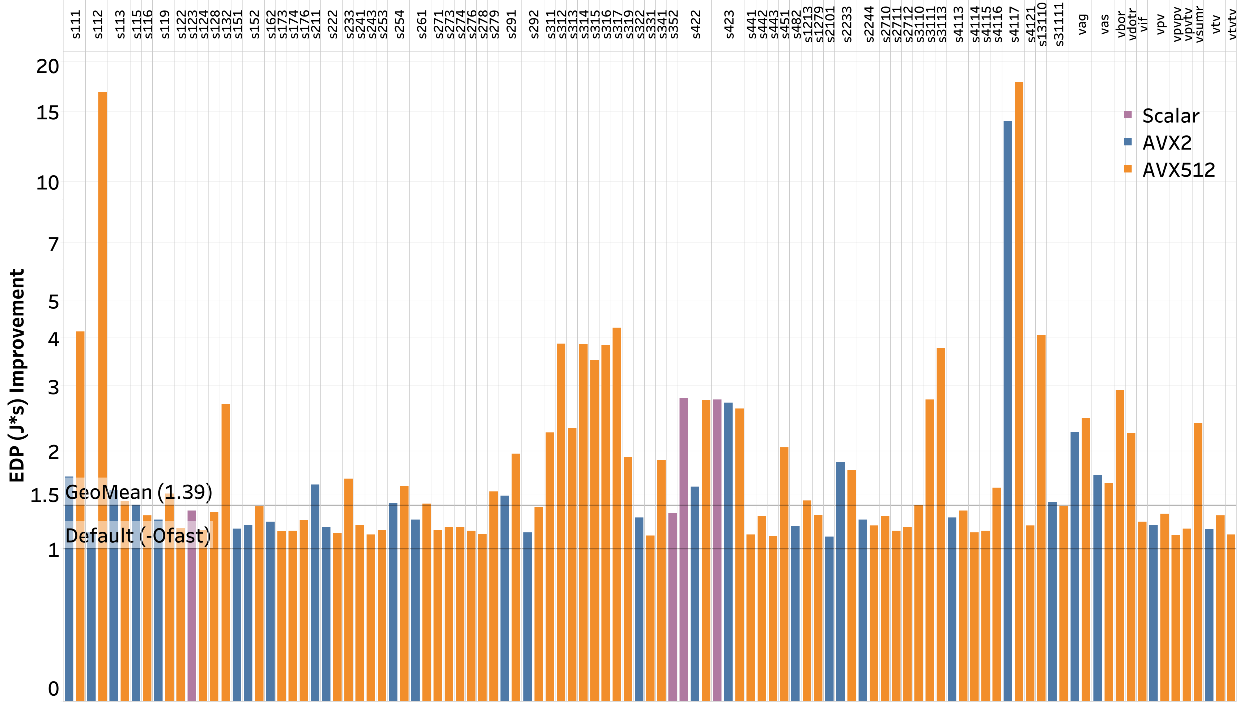


Figure 5.8: EDP comparison between different vector length w.r.t. ICC -Ofast for TSVC. Y-axis is log scaled. Only cases that showed more than 10% EDP improvement are shown.

tion over the non-vectorized version than in other compilers. `icc` was able to auto-vectorize 80% of the loop nests in TSVC with 5% or higher speedup, compared to 75% by `gcc` and 57% by `clang`. The relative improvements in EDP of auto-vectorized versions with different vector lengths over default `-Ofast` version for `icc` are shown in Fig. 5.8.

The results in Fig. 5.8 show that if one were to select the most energy efficient vector length for each of the loop nests, EDP can be reduced by, GeoMean of, 1.39x over the performance oriented cost models of the compiler for picking a vector length. Compiler can choose an incorrect vector length because of an incorrect prediction of profitability of auto-vectorizing the loop nests, different behavior of memory hierarchy than expected by the compilers, etc. The change in EDP was more than 10% for 63% of the loop nests, when selecting a different vector length than what the compiler's cost models decided. For 15% of the loop nests the best performing code version (in terms of execution time) was not same as the most energy efficient version. Therefore, emphasizing the previous observation that the best performing

vector length may not be the most energy efficient for all cases. Fig. 5.8 also shows how large of a positive impact vectorization could have on energy efficiency when compared to scalar code. AVX-512 versions improved energy efficiency by a factor of 9.7x (GeoMean) over the scalar versions, whereas AVX-2 versions improved energy efficiency by a factor of 7.6x (GeoMean).

Analysis

In most cases, it is evident that using AVX-512 improves energy efficiency by a factor ranging from 2 to 16 over AVX-2 (preferred by the default cost models in the compiler). For example, in S112, the compiler's cost models decided to not vectorize the loop nest, instead unrolled the innermost loop nest by a factor of 2, whereas on forcing the compiler to generate AVX-512 code lead to an EDP improvement by a factor of 16. In S352, compiler decided to use AVX-2 instruction that required generation of vectorized stride-5 loads/gather instructions which lead to inefficiency, whereas the scalar version would have been more beneficial in terms of improving energy efficiency. In S4117, compiler deemed generating vectorized MOV and FMA instructions for this loop nest as unprofitable compared to the scalar version, but on forcing the generation of AVX-2 and AVX-512 instructions lead to energy efficiency improvement of more than 14x. In S423 and S424, on forcing the compiler to vectorize, compiler performs loop peeling to partially vectorize the loop nest, resulting in a small improvement.

5.4.2 Impact on Performance and Energy Consumption when Increasing the Number of Active Cores

As mentioned in Table 5.1, on using longer vector extensions, there is a significant reduction in maximum core frequency if the number of active cores is increased. To analyze the

impact of this frequency scaling on performance and energy consumption, this experiment runs multiple copies of highly auto-vectorized TSVC benchmark simultaneously on different cores, while scaling number of active cores/benchmark copies from 1 to 16.

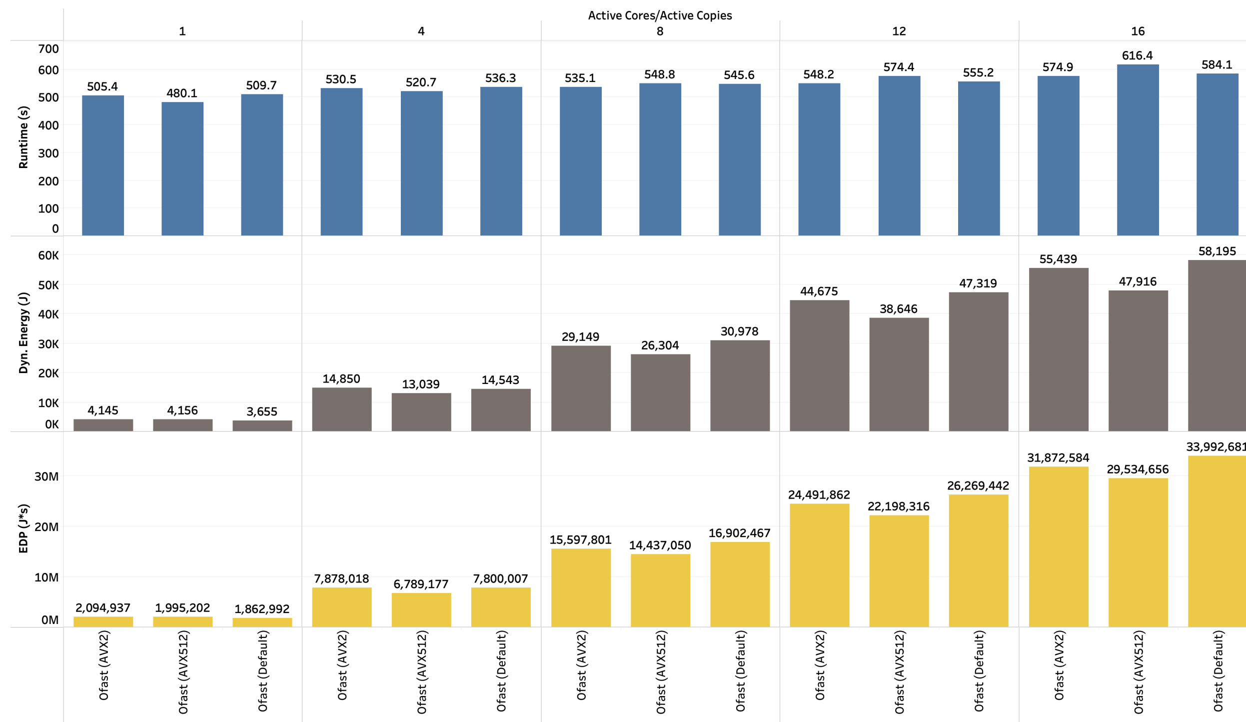


Figure 5.9: Variation in Average Runtime, Total Dynamic Energy Consumption and EDP when running multiple copies of TSVC with different Vectorized versions.

Fig. 5.9 presents execution times, dynamic energy consumption and EDP variance between the default version (cost-model chosen vector length), AVX-2 and AVX-512 version of the benchmark, while increasing number of active cores. The benchmark is auto-vectorized by the Intel Compiler, so out of the 151 unique and consecutive loop nests at least 105 are auto-vectorized by the compiler [140]. The maximum core frequency is 3.5 GHz and 2.2 GHz when running heavy AVX-512 instruction on 1 core and 16 cores respectively, that is a 37% reduction. The average runtime per copy increases by 28% for the AVX-512 version when increasing copies from 1 to 16. Whereas, the dynamic energy increases by 11.5x on increasing copies from 1 to 16. Similarly for AVX-2, while the maximum core frequency reduces by 19.5% on increasing copies from 1 to 16, the average runtime per copy increases

by 14%. Whereas, the dynamic energy increases by 13.4x on increasing copies from 1 to 16.

For single core, AVX-512 version performs better than the AVX-2 (by 1.05x) and default (by 1.06x) versions. Whereas in terms of energy efficiency, default version is more efficient than AVX-2 (by 1.12x) and AVX-512 (by 1.07x) versions. AVX-512 version goes from being the best performing version to worst performing version when scaling the number of active cores, but remains the most energy efficient version (except for the single core setting) overall as the max. core frequencies are significantly reduced.

5.5 A Framework for Improving Energy Efficiency

After analyzing the energy efficiency improvement seen in the previous sections, the same idea is incorporated in the *MCompiler* framework. The framework for choosing the most suited code optimizer for the loop nests using the Exploratory Search is extended to use the energy efficiency metrics (EDP) as the selection criteria rather than the execution times as previously shown in Chapter 2. The framework for improving the energy efficiency of the applications is shown in Fig. 5.10. The *MCompiler* driver invokes this part of the framework, if the `--energy` flag is set. The architecture of the *MCompiler* framework is modified in the following ways for improving the energy efficiency.

The Extractor instruments the loop nest body with LIKWID [135] APIs. LIKWID uses the RAPL interface [69, Chapter 14.9] to measure the consumed energy on the package (socket) and DRAM level. The Extractor adds `LIKWID_MARKER_INIT` and `LIKWID_MARKER_CLOSE` APIs at the beginning and the end of the program respectively, i.e., towards the beginning and the end of the `main` function. Next, in the hotspot files, the Extractor adds `LIKWID_MARKER_START(<LOOP ID>)` statement before the loop nest body and adds `LIKWID_MARKER_STOP(<LOOP ID>)` statement after the loop nest body.

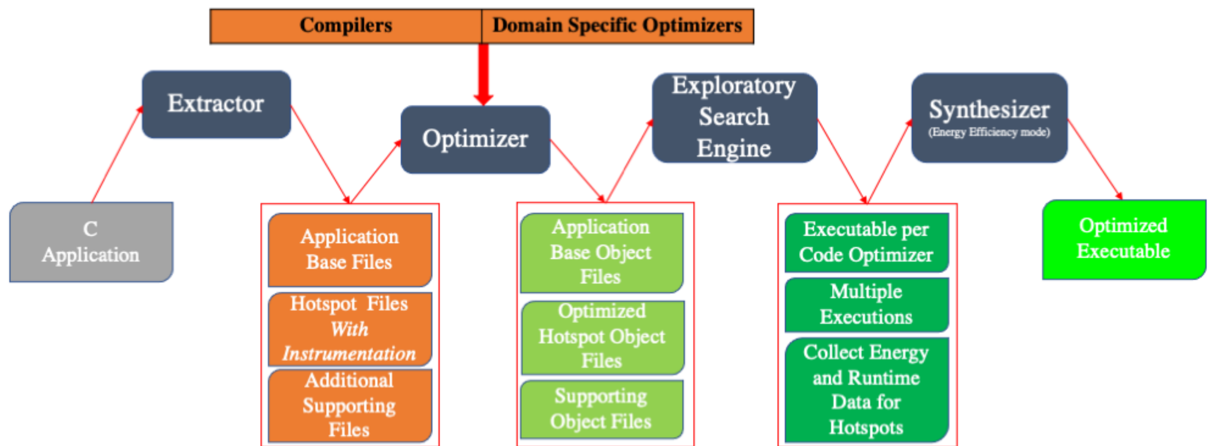


Figure 5.10: *MCompiler* Framework for Improving Energy Efficiency

The Optimizer compiles the hotspot files, for each of the available code optimizers, with an additional macro definition `-DLIKWID_PERFMON`. Then, the Exploratory Search Engine generates an executable and runs the executables with `likwid-perfctr -C <processor ID> -g ENERGY`, which pins the application to a particular processor or cores, and produces the energy measurements (including Runtime, Energy and Power Consumption for both the Package and DRAM). These measurements are stored as a CSV file for future reference of the users.

The synthesizer calculates and compares the Energy-Delay-Product (EDP) for each loop nest from different code optimizers and chooses the code optimizer that generated the most energy efficient code, as the most suited code optimizer. For loop nests with no information, i.e., the code that was not executed during Exploratory Search phase, the default compiler is used. The synthesizer then generates the final executable that contains no instrumentation code.

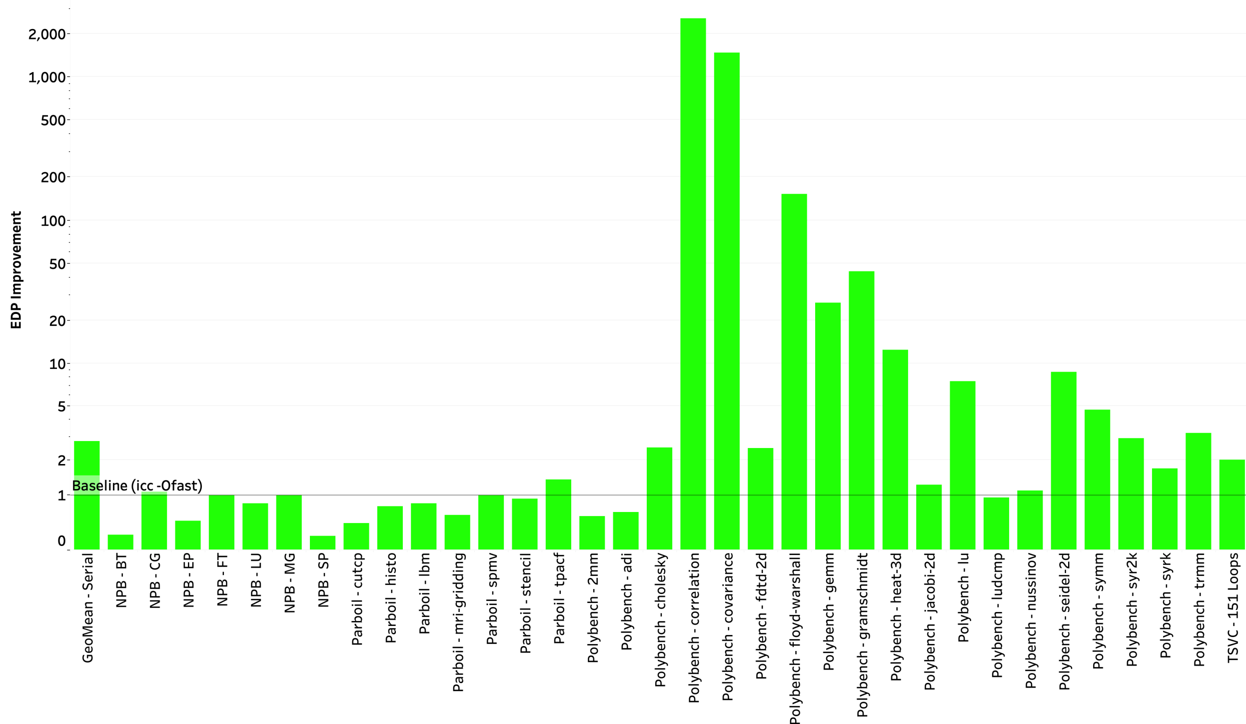


Figure 5.11: *MCompiler* EDP Improvement for Serial Benchmarks

The goal of adding this feature is to generate code that improves energy efficiency on intended architectures and not just the execution times. Furthermore, this feature provides the users with more information and insight about the application's energy and power consumption.

5.6 Evaluation of the *MCompiler* for Improving Energy Efficiency

This section presents results of the exploratory search by the *MCompiler* for choosing the most energy efficient code optimizer for four benchmark suites: TSVC, Polybench, NAS Benchmark Suite (NPB) and Parboil Benchmark Suite. Each application was executed 3 times for each of the code optimizers and the median EDP was chosen for deciding the most suited code optimizer.

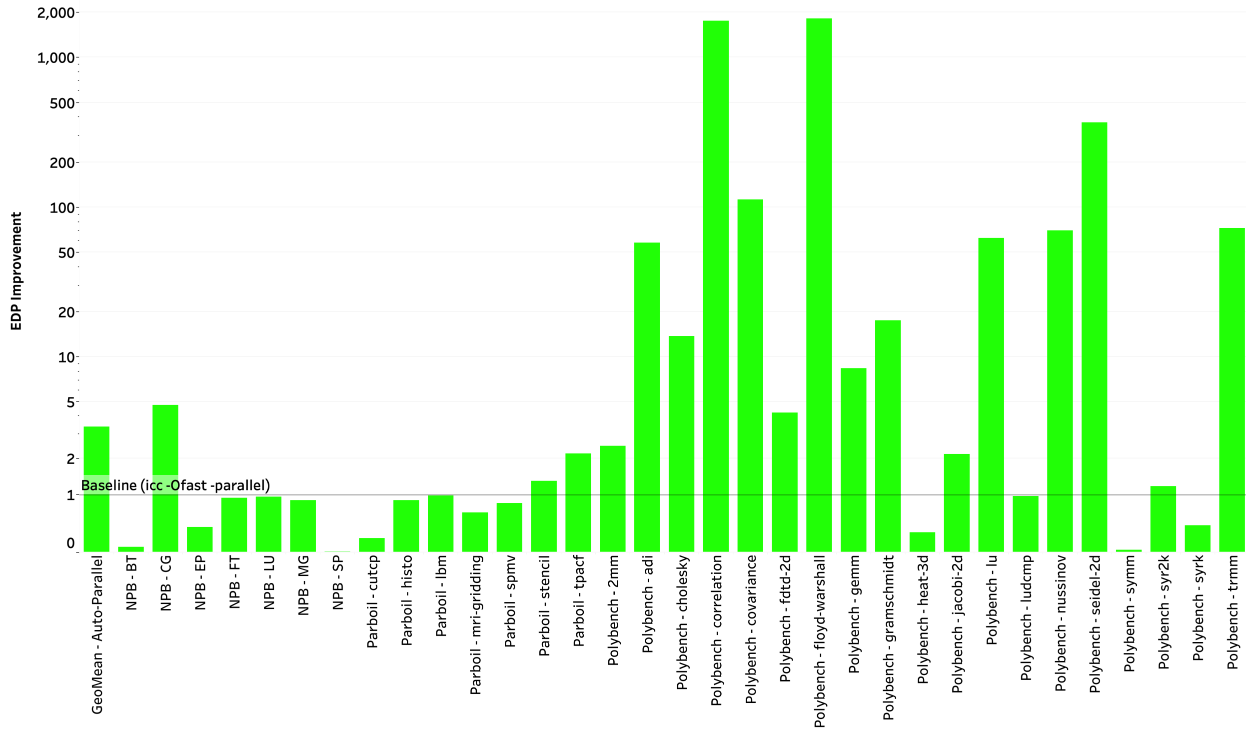


Figure 5.12: *MCompiler* EDP Improvement for Auto-Parallelized Benchmarks

Benchmarks, Code Optimizers and Target Architecture

The same four benchmark suites are chosen as the ones shown in Chapter. 3, where the goal was to improve performance, here the goal is to improve energy efficiency. The same five code optimizers, as previously shown in Chapter 2, Table 2.2 are incorporated in the *MCompiler* framework here too. All five optimizers are used for serial experiments. Of the five optimizers, only four optimizers (`icc`, `Polly` and `Pluto`) can auto-parallelize the serial code and are used for auto-parallelized code experiments. The baseline for comparison is `icc` (`-Ofast -xHost [-parallel]`) compiled benchmarks for all experiments. `icc` was chosen as the baseline because `icc` generated code performed better for more benchmarks than other code optimizers as shown in Fig. 5.13. The source codes used for the baseline are the original benchmark codes and not the modified source codes generated by the *MCompiler*'s Loop Extractor.

	TSVC		Polybench		NPB		Parboil		Total	
	Loop Nests	Percentage	Loop Nests	Percentage	Loop Nests	Percentage	Loop Nests	Percentage	Loop Nests	Percentage
Clang	12	7.9%	5	7.6%	27	10.3%	3	10.3%	47	9.3%
GCC	37	24.5%	5	7.6%	73	28.0%	9	31.0%	124	24.5%
ICC	59	39.1%	22	33.3%	143	54.8%	13	44.8%	237	46.7%
Pluto*	29	19.2%	20	30.3%	0	0.0%	2	6.9%	51	10.1%
Polly	14	9.2%	14	21.2%	18	6.9%	2	6.9%	48	9.5%

(a) Serial Code

	Polybench		NPB		Parboil		Total	
	Loop Nests	Percentage	Loop Nests	Percentage	Loop Nests	Percentage	Loop Nests	Percentage
ICC	18	27.3%	190	72.0%	20	69.0%	228	63.5%
Pluto*	40	60.6%	1	0.4%	0	0.0%	41	11.4%
Polly	8	12.1%	73	27.7%	9	31.0%	90	25.1%

(b) Auto-Parallelized Code

* Pluto optimized code was compiled with ICC.

Figure 5.13: Distribution of most energy efficient code per Code Optimizer. Breakdowns per benchmarks suite showcase benefits of specialized code optimizers.

The target architecture for the experiments is a two-socket, sixteen-core Intel Skylake Xeon Gold 6142 [43]. Each Xeon processor has 32KB L1 cache, 1MB L2 cache, 22MB L3 cache. The Skylake architecture supports SIMD instruction set extensions, i.e., SSE, AVX, AVX2, AVX-512CD and AVX-512F. CPU Hyper Threading (SMT) is turned off and cores are operating at the maximum frequency (same as mentioned before in Table 5.1).

For serial experiment measurements (including running the Exploratory Search Engine), the applications are pinned to the last core of the second processor. For the auto-parallelization experiment measurements (including running the Exploratory Search Engine), only 16 cores of the second processor are used and just one thread is mapped per core by setting the

environment variables for OpenMP runtimes.

5.6.1 Serial Code

The results for the Exploratory Search for improving energy efficiency are shown in Fig. 5.11. The benchmark labels show the benchmark suite that a particular benchmark belongs to. The EDP improvement across the 151 loop nests from TSVC is 2.0x over `icc`. As shown in Fig. 5.13, `icc` was chosen as the most suited code optimizer for 39% of the loop nests, followed by `gcc` at 24.5%. This is a deviation from the distribution in the Exploratory Search for performance improvement, where `icc` and `gcc` were chosen 44% and 20% of the times respectively.

As expected, the two polyhedral model based optimizers were chosen as the most energy efficient code optimizer for 72% of the loop nests that dominate execution time of the main kernels for Polybench benchmarks. These optimizers improved performance by huge margin in a few cases and that also lead to similar improvement in energy consumption. The percentage of loop nests chosen from each code optimizer can be seen in Fig. 5.13.

Overall, the *MCompiler* improves EDP for serial benchmarks from four different suites by GeoMean of 2.77x.

5.6.2 Auto-Parallelized Code

These experiments were performed with 16 threads for both the exploratory search phase and measuring EDP of the optimized executables. The code optimizers optimized the loop nests with their default setting for statically deciding the profitability of the parallel code and for choosing the runtime settings, such as scheduling policies. Benchmarks from Polybench, NPB-ACC and Parboil were used in these experiments.

The benchmarks that showed large loss in performance with the *MCompiler*, also show similar trend for EDP. Yet, the *MCompiler* improves EDP for auto-parallelized benchmarks from four different suites by GeoMean of 3.37x.

5.7 Summary

This chapter shows the impact of different compilers and their performance driven optimizations on energy efficiency for the loop nests. The exploratory analysis shows that there is potential for compilers to optimize and generate code for loop nests differently, if the goal was to improve the energy efficiency of the processor. Also, the impact of using different vector extensions on performance and energy efficiency is studied.

Based on this exploration, the Exploratory Search method of the *MCompiler* is expanded to optimize applications for better energy efficiency by choosing the most energy efficient version possible for each loop nest. The *MCompiler* framework improves the overall energy efficiency for applications over state-of-the-art compiler (compiled at equivalent of `-O3`) by a geometric mean of 2.77x for auto-vectorized code and 3.37x for auto-parallelized code.

Chapter 6

Prior Art

This chapter discusses recent related works that have tried to improve performance using a compilation framework, apply Machine Learning to achieve better performance or used a compilation framework to improve energy efficiency of the applications.

Compilation Frameworks and Tools

The OptiScope infrastructure by Moseley et. al. [102] performed function-level and loop-level quantitative comparisons of application compiled by different compilers and/or optimization settings. Similar to the analysis showed in this thesis, they looked at the impact of interaction of optimization techniques for complex target architectures. But their tool performed binary analysis with the goal of assisting compiler developers in discovering new opportunities and evaluate changes.

The work by Fursin et. al. [51], called Milepost GCC, presents an iterative compilation and auto-tuning framework that predicts good combinations of compiler flags to improve execution time. Their tool explores `gcc` and its flags and uses ML techniques to predict

good optimizations based on program features.

Another similar work, the OpenTuner framework by Ansel et. al. [9], searches for the best performing GCC/G++ flag combinations for C/C++ applications, in addition to searching configurations for Halide and other domain specific applications. Both Milepost GCC and OpenTuner frameworks explore different combinations of code optimizer flags that has been extensively studied.

MiCOMP, proposed by Ashouri et. al. [10], performs phase ordering of the optimizations in LLVM's highest optimization level using optimization sub-sequences and machine learning. They cluster the optimization passes of LLVM into different clusters and predict the speedup of a complete sequence of these clusters of optimizations. The iterative compilation performed using these sub-sequences outperforms the default sequence of optimization passes enabled at optimization levels such as `-O3`.

In this thesis, the results are presented with applications optimized using the most influential or recommended flag combinations, for improving performance, from each code optimizer. By its design the framework can also include auto-tuning and iterative compilation [115, 116, 36] frameworks, such as the MilepostGCC, OpenTuner and MiCOMP, for optimizing applications.

Applying Machine Learning to Compiler Optimizations and Frameworks

Most prior work in the area of applying ML to the compilation process has been done in order to perform auto-tuning [11] of the applications. Other prior works that have addressed challenges in compiler optimizations using Machine Learning have focused on auto-vectorization [130, 140] and on scalability and scheduling configurations for the parallelism [17, 139, 134].

Tournavitis et. al.[134] use a mix of static and dynamic features to develop a platform-agnostic, profiling-based parallelism detection method for sequential applications. Their method requires user's approval for parallelization decisions that cannot be proven conclusively. They use ML models to judge the profitability on parallelization and to select the scheduling policy. In contrast, the *MCompiler* uses just the dynamic features to train ML models and let the ML models choose the most suited candidate that can generate a profitable auto-parallelized code. In future, these mechanisms can be incorporated into the *MCompiler* to predict number of threads and select the scheduling policy as well. Stock et. al. [130] developed a ML-based performance model to guide SIMD compiler optimizations for vectorizing tensor contraction computations. Watkinson et. al. [140] use ML models to predict opportunities for auto-vectorization and its profitability across multiple compilers and architectures. However, this thesis explores the use of ML on a coarser level on kernels from a variety of computations to predict an optimizer that can generate an efficient serial code, which includes auto-vectorized code, as well as parallel code.

NeuroVectorizer [57] proposed an approach for handling loop vectorization and an end-to-end solution using deep reinforcement learning (RL). It finds two vectorization parameters via RL, the loop unrolling factor and the interleaving factor. Similarly, Vemal [99] uses Imitation Learning to produce a vectorization scheme that is better than the heuristics implemented in the LLVM compiler. One can also incorporate tools like NeuroVectorizer and Vemal into the *MCompiler* and potentially obtain additional speedups.

Using Compilation Framework to improve Energy Efficiency

The work by Wang et. al. [138] performed energy auto-tuning using polyhedral model based tools such as PoCC [112]. They tried to understand the correlation between performance and energy consumptions for optimized serial, including auto-vectorized code, and auto-parallelized code. Jager et. al. [70] present a comparative study between `gcc` and `icc`

generated code in terms of energy consumption. Whereas, the work by Georgiou et. al. [52] show that performing fewer of the optimizations passes available at LLVM's standard optimization levels, such as `-O2`, while preserving their original ordering, significant savings can be achieved in both execution time and energy consumption. They technique generate multiple optimization configurations by removing the last transformation flag of the current optimization configuration and select the most best configuration based on the execution results. These findings and improvements are similar to the ones shown in this thesis for the purpose of improving the energy efficiency, and such tools can easily be incorporated into the *MCompiler*.

Chapter 7

Conclusions and Future Directions

The goal of this thesis was to synthesize multiple compilation and optimization techniques into a single compilation framework. The implemented framework, called the *MCompiler*, harnesses the strengths of multiple compilers, while substituting the weakness of individual compilers. It can optimize applications for different objectives. These objective can range from improving the performance of serial (auto-vectorized) codes, auto-parallelized codes or hand-parallelized codes to improving the energy efficiency for such codes on modern architectures. Through the use of Exploratory Search and Machine Learning algorithms, the framework optimizes application hotspots for achieving better performance and energy efficiency over state-of-the-art compilers.

The need for a multiple compiler approach is discussed in Chapter 2 and the design of the *MCompiler* framework is presented. The framework incorporates optimized loop nest code - serial code, auto-parallelized code or OpenMP code - from a collection of state-of-the-art code optimizers to generate a single executable. The framework can be used with a exploratory search to choose the most suited code optimizer for the loop nests. Exploratory Search results, presented in Chapter 3, showed that the *MCompiler* with five code optimizers

can significantly improve application performance. The framework benefits from the ability of different code optimizers to apply different sequences of loop nest transformations and use their in-built heuristics and cost-models to make decisions in order to produce most suited code for the underlying architecture. *MCompiler* framework is designed to be extendable with more code optimizers, optimizer flag combinations and more features. It can also be used as a tool for compiler researchers to incorporate and analyze the performance of their code optimization techniques and compare to other code optimizers. Even though the code optimizers have the capability of performing similar set of loop transformation techniques, they can produce optimized versions of the code that have significant difference in performance. This shows that state-of-the-art code optimizers miss out on opportunities for producing efficient code and the multiple compiler approach can help towards identifying those opportunities and making improvements in order to achieve better performance.

In Chapter 4, this thesis explored the possibility of learning about the inherent characteristics of the loop nests, in terms of its behavior on the architecture, that can be detected from the hardware performance counters. Results shows that these inherent characteristics of the loop nests can be successfully captured using Machine Learning algorithms. These Machine Learning models are then used to predict which code optimizer (with its transformations and cost models) would be the most beneficial for a loop nest. The *MCompiler* framework is expanded to incorporate these Machine Learning models and to replace the exploratory search with an efficient Machine Learning based prediction for the most suited code optimizer for a loop nest. The results show that the Machine Learning models can predict the most suited code optimizer with a small performance loss compared to the Exploratory Search.

Finally, in Chapter 5, this thesis explored the impact of different compilers and their performance driven optimizations on energy efficiency for the loop nests. The exploratory analysis shows that there is potential for compilers to optimize and generate code for loop nests differently, if the goal was to improve the energy efficiency of the processor. Based on this

exploration, the Exploratory Search method of the *MCompiler* is expanded to optimize applications for better energy efficiency by choosing the most energy efficient version possible for each loop nest. The framework is able to measure the impact of various architectural intricacies and dynamic parameters, such as accounting for DVFS changes, before generating the optimized binary.

Future Directions

The multiple compiler approach presented and analyzed in this thesis has shown promising results and highlighted the strengths of various optimization techniques currently implemented in the state-of-the-art compilers and domain-specific optimizers. With increasing complexities of modern architectures, the task for compiler optimizations to achieve the optimal performance and/or efficiency on these architectures is only going to become more complex. Expecting a single compiler to utilize the target architecture to its fullest potential for a variety of applications is wishful thinking. With the continuous rise in newer domain-specific languages and domain-specific accelerators, compilers and compiler optimizations are going in the direction of becoming fine-grained and precise. But even then, there will remain a need to explore various optimizations, techniques and configurations to achieve the *best* performance or the *best* energy efficiency on an architecture for any given application.

The *MCompiler* is designed to be flexible and extendable. This thesis shows how the framework can be extended to incorporate Machine Learning models and optimize for different metrics such as energy efficiency. In the near future, the framework can be improved in following ways:

- Perform optimizations to the source code before extracting the hotspots in order to provide compilers with more information while optimizing hotspot files.
- Add heuristics or Machine Learning techniques to predict most suited runtime config-

uration for the hotspots such as degree of parallelism or hardware knob settings.

- Apply exploration of fine grained options per compiler.

Bibliography

[1] M. Abadi, A. Agarwal, P. Barham, E. Brevdo, Z. Chen, C. Citro, G. S. Corrado, A. Davis, J. Dean, M. Devin, S. Ghemawat, I. Goodfellow, A. Harp, G. Irving, M. Isard, Y. Jia, R. Jozefowicz, L. Kaiser, M. Kudlur, J. Levenberg, D. Mané, R. Monga, S. Moore, D. Murray, C. Olah, M. Schuster, J. Shlens, B. Steiner, I. Sutskever, K. Talwar, P. Tucker, V. Vanhoucke, V. Vasudevan, F. Viégas, O. Vinyals, P. Warden, M. Wattenberg, M. Wicke, Y. Yu, and X. Zheng. TensorFlow: Large-scale machine learning on heterogeneous systems, 2015. Software available from tensorflow.org.

[2] F. Agakov, E. Bonilla, J. Cavazos, B. Franke, G. Fursin, M. O'Boyle, J. Thomson, M. Toussaint, and C. Williams. Using machine learning to focus iterative optimization. In *International Symposium on Code Generation and Optimization (CGO'06)*, pages 11 pp.–305, 2006.

[3] R. Allen and S. Johnson. Compiling c for vectorization, parallelization, and inline expansion. In *Proceedings of the ACM SIGPLAN 1988 Conference on Programming Language Design and Implementation*, PLDI '88, page 241–249, New York, NY, USA, 1988. Association for Computing Machinery.

[4] R. Allen and K. Kennedy. Automatic translation of fortran programs to vector form. *ACM Trans. Program. Lang. Syst.*, 9(4):491–542, Oct. 1987.

[5] L. Almagor, K. D. Cooper, A. Grosul, T. J. Harvey, S. W. Reeves, D. Subramanian, L. Torczon, and T. Waterman. Finding Effective Compilation Sequences. In *Proceedings of the 2004 ACM SIGPLAN/SIGBED Conference on Languages, Compilers, and Tools for Embedded Systems*, LCTES '04, page 231–239, New York, NY, USA, 2004. Association for Computing Machinery.

[6] S. P. Amarasinghe and M. S. Lam. Communication optimization and code generation for distributed memory machines. In *Proceedings of the ACM SIGPLAN 1993 Conference on Programming Language Design and Implementation*, PLDI '93, page 126–138, New York, NY, USA, 1993. Association for Computing Machinery.

[7] C. Ancourt and F. Irigoin. Scanning polyhedra with DO loops. In *Proceedings of the Third ACM SIGPLAN Symposium on Principles & Practice of Parallel Programming (PPOPP), Williamsburg, Virginia, USA, April 21-24, 1991*, pages 39–50, 1991.

[8] J. M. Anderson and M. S. Lam. Global Optimizations for Parallelism and Locality on Scalable Parallel Machines. *SIGPLAN Not.*, 28(6):112–125, June 1993.

[9] J. Ansel, S. Kamil, K. Veeramachaneni, J. Ragan-Kelley, J. Bosboom, U. O'Reilly, and S. Amarasinghe. Opentuner: An extensible framework for program autotuning. In *2014 23rd International Conference on Parallel Architecture and Compilation Techniques (PACT)*, pages 303–315, 2014.

[10] A. H. Ashouri, A. Bignoli, G. Palermo, C. Silvano, S. Kulkarni, and J. Cavazos. MiCOMP: Mitigating the compiler phase-ordering problem using optimization subsequences and machine learning. *ACM Transactions on Architecture and Code Optimization (TACO)*, 14(3):29, 2017.

[11] A. H. Ashouri, W. Killian, J. Cavazos, G. Palermo, and C. Silvano. A Survey on Compiler Autotuning Using Machine Learning. *ACM Comput. Surv.*, 51(5), Sept. 2018.

[12] D. F. Bacon, S. L. Graham, and O. J. Sharp. Compiler transformations for high-performance computing. *ACM Comput. Surv.*, 26(4):345–420, Dec. 1994.

[13] R. Baghdadi, J. Ray, M. B. Romdhane, E. Del Sozzo, A. Akkas, Y. Zhang, P. Suriana, S. Kamil, and S. Amarasinghe. Tiramisu: A polyhedral compiler for expressing fast and portable code. In *Proceedings of the 2019 IEEE/ACM International Symposium on Code Generation and Optimization*, CGO 2019, page 193–205. IEEE Press, 2019.

[14] D. H. Bailey, E. Barszcz, J. T. Barton, D. S. Browning, R. L. Carter, L. Dagum, R. A. Fatoohi, P. O. Frederickson, T. A. Lasinski, R. S. Schreiber, H. D. Simon, V. Venkatakrishnan, and S. K. Weeratunga. The nas parallel benchmarks summary and preliminary results. In *Supercomputing '91:Proceedings of the 1991 ACM/IEEE Conference on Supercomputing*, pages 158–165, Nov 1991.

[15] U. K. Banerjee. *Loop Transformations for Restructuring Compilers: The Foundations.* Kluwer Academic Publishers, Norwell, MA, USA, 1993.

[16] U. K. Banerjee. *Dependence Analysis.* Kluwer Academic Publishers, Norwell, MA, USA, 1996.

[17] B. J. Barnes, B. Rountree, D. K. Lowenthal, J. Reeves, B. De Supinski, and M. Schulz. A regression-based approach to scalability prediction. In *Proceedings of the 22nd annual international conference on Supercomputing*, pages 368–377. ACM, 2008.

[18] A. Barredo, J. M. Cebrian, M. Valero, M. Casas, and M. Moreto. Efficiency analysis of modern vector architectures: vector alu sizes, core counts and clock frequencies. *The Journal of Supercomputing*, pages 1–20, 2019.

[19] A. Bataev, A. Bokhanko, and J. Cownie. Towards OpenMP support in LLVM. In *2013 European LLVM Conference*, 2013.

[20] B. Blume, R. Eigenmann, K. Faigin, J. Grout, J. Hoeflinger, D. Padua, P. Petersen, B. Pottenger, L. Rauchwerger, P. Tu, and S. Weatherford. Polaris: The Next Generation in Parallelizing Compilers. In *International Workshop on Languages and Compilers for Parallel Computing*, pages 10–1. Springer-Verlag, Berlin/Heidelberg, 1994.

[21] U. Bondhugula, M. Baskaran, S. Krishnamoorthy, J. Ramanujam, A. Rountev, and P. Sadayappan. Automatic transformations for communication-minimized parallelization and locality optimization in the polyhedral model. In *International Conference on Compiler Construction*, pages 132–146. Springer, 2008.

[22] U. Bondhugula, A. Hartono, J. Ramanujam, and P. Sadayappan. A practical automatic polyhedral parallelizer and locality optimizer. In *Proceedings of the 29th ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 101–113, 2008.

[23] L. Breiman. Random forests. *Machine learning*, 45(1):5–32, 2001.

[24] D. Callahan, K. D. Cooper, K. Kennedy, and L. Torczon. Interprocedural Constant Propagation. In *Proceedings of the 1986 SIGPLAN Symposium on Compiler Construction*, SIGPLAN '86, page 152–161, New York, NY, USA, 1986. Association for Computing Machinery.

[25] D. Callahan, J. Dongarra, and D. Levine. Vectorizing compilers: A test suite and results. In *Proceedings of the 1988 ACM/IEEE Conference on Supercomputing*, Supercomputing '88, pages 98–105, Los Alamitos, CA, USA, 1988. IEEE Computer Society Press.

[26] R. Cammarota, L. A. Beni, A. Nicolau, and A. V. Veidenbaum. Optimizing program performance via similarity, using a feature-agnostic approach. In *Advanced Parallel Processing Technologies*, pages 199–213, Berlin, Heidelberg, 2013. Springer.

[27] R. Caruana and A. Niculescu-Mizil. An empirical comparison of supervised learning algorithms. In *Proceedings of the 23rd International Conference on Machine Learning*, ICML '06, page 161–168, New York, NY, USA, 2006. Association for Computing Machinery.

[28] J. Cavazos, G. Fursin, F. Agakov, E. Bonilla, M. F. O'Boyle, and O. Temam. Rapidly selecting good compiler optimizations using performance counters. In *International Symposium on Code Generation and Optimization (CGO'07)*, pages 185–197. IEEE, 2007.

[29] J. M. Cebrian, M. Jahre, and L. Natvig. Parvec: vectorizing the parsec benchmark suite. *Computing*, 97(11):1077–1100, 2015.

[30] J. M. Cebrian, L. Natvig, and M. Jahre. Scalability Analysis of AVX-512 Extensions. *The Journal of Supercomputing*, pages 1–16, 2019.

[31] J. M. Cebrián, L. Natvig, and J. C. Meyer. Improving energy efficiency through parallelization and vectorization on intel core i5 and i7 processors. In *2012 SC Companion: High Performance Computing, Networking Storage and Analysis*, pages 675–684, Nov 2012.

[32] M. Chadha and M. Gerndt. Modelling DVFS and UFS for Region-Based Energy Aware Tuning of HPC Applications. In *2019 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, pages 805–814, 2019.

[33] C. Chen, J. Chame, and M. Hall. Chill: A framework for composing high-level loop transformations. Technical report, Citeseer, 2008.

[34] D. Chen, T. Moseley, and D. X. Li. AutoFDO: Automatic feedback-directed optimization for warehouse-scale applications. In *2016 IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*, pages 12–23, 2016.

[35] T. Chen, T. Moreau, Z. Jiang, L. Zheng, E. Yan, H. Shen, M. Cowan, L. Wang, Y. Hu, L. Ceze, C. Guestrin, and A. Krishnamurthy. TVM: An automated end-to-end optimizing compiler for deep learning. In *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18)*, pages 578–594, Carlsbad, CA, Oct. 2018. USENIX Association.

[36] Y. Chen, Y. Huang, L. Eeckhout, G. Fursin, L. Peng, O. Temam, and C. Wu. Evaluating Iterative Optimization across 1000 Datasets. In *Proceedings of the 31st ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '10, page 448–459, New York, NY, USA, 2010. Association for Computing Machinery.

[37] Z. Chen, Z. Gong, J. J. Szaday, D. C. Wong, D. Padua, A. Nicolau, A. V. Veidenbaum, N. Watkinson, Z. Sura, S. Maleki, J. Torrellas, and G. DeJong. Lore: A loop repository for the evaluation of compilers. In *2017 IEEE International Symposium on Workload Characterization (IISWC)*, pages 219–228, 2017.

[38] K. D. Cooper and K. Kennedy. Efficient Computation of Flow Insensitive Interprocedural Summary Information. In *Proceedings of the 1984 SIGPLAN Symposium on Compiler Construction*, SIGPLAN '84, page 247–258, New York, NY, USA, 1984. Association for Computing Machinery.

[39] A. Darte, Y. Robert, and F. Vivien. *Scheduling and automatic Parallelization*. Springer Science & Business Media, 2012.

[40] J. Demšar and et. al. Orange: data mining toolbox in python. *The Journal of Machine Learning Research*, 14(1):2349–2353, 2013.

[41] J. Doerfert and H. Finkel. Compiler optimizations for openmp. In *International Workshop on OpenMP*, pages 113–127. Springer, 2018.

[42] C. Dubach, T. M. Jones, E. V. Bonilla, G. Fursin, and M. F. O'Boyle. Portable compiler optimisation across embedded programs and microarchitectures using machine

learning. In *2009 42nd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pages 78–88, 2009.

[43] D. Duplyakin, R. Ricci, A. Maricq, G. Wong, J. Duerig, E. Eide, L. Stoller, M. Hibler, D. Johnson, K. Webb, A. Akella, K. Wang, G. Ricart, L. Landweber, C. Elliott, M. Zink, E. Cecchet, S. Kar, and P. Mishra. The design and operation of CloudLab. In *Proceedings of the USENIX Annual Technical Conference (ATC)*, pages 1–14, July 2019.

[44] G. Esakkimuthu, N. Vijaykrishnan, M. Kandemir, and M. J. Irwin. Memory system energy: Influence of hardware-software optimizations. In *ISLPED'00: Proceedings of the 2000 International Symposium on Low Power Electronics and Design (Cat. No.00TH8514)*, pages 244–246, July 2000.

[45] P. Feautrier. Array expansion. In *In ACM Int. Conf. on Supercomputing*, pages 429–441, 1988.

[46] P. Feautrier. Dataflow Analysis of Array and Scalar References. *International Journal of Parallel Programming*, 20, 1991.

[47] P. Feautrier. Some efficient solutions to the affine scheduling problem. I. One-dimensional time. *International Journal of Parallel Programming*, 21(5):313–347, 1992.

[48] P. Feautrier. Some efficient solutions to the affine scheduling problem. Part II. Multidimensional time. *International Journal of Parallel Programming*, 21(6):389–420, 1992.

[49] A. Fernández, V. Beltran, X. Martorell, R. M. Badia, E. Ayguadé, and J. Labarta. Task-based programming with ompss and its application. In *European Conference on Parallel Processing*, pages 601–612. Springer, 2014.

[50] J. H. Friedman. Greedy Function Approximation: A Gradient Boosting Machine. *Annals of Statistics*, 29:1189–1232, 2000.

[51] G. Fursin, Y. Kashnikov, A. W. Memon, Z. Chamski, O. Temam, M. Namolaru, E. Yom-Tov, B. Mendelson, A. Zaks, E. Courtois, et al. Milepost GCC: Machine learning enabled self-tuning compiler. *International journal of parallel programming*, 39(3):296–327, 2011.

[52] K. Georgiou, C. Blackmore, S. Xavier-de Souza, and K. Eder. Less is more: Exploiting the standard compiler optimization levels for better performance and energy consumption. In *Proceedings of the 21st International Workshop on Software and Compilers for Embedded Systems*, SCOPES '18, pages 35–42, New York, NY, USA, 2018. Association for Computing Machinery.

[53] S. V. Gheorghita, H. Corporaal, and T. Basten. Iterative compilation for energy reduction. *J. Embedded Comput.*, 1(4):509–520, Dec. 2005.

[54] Z. Gong, Z. Chen, J. Szaday, D. Wong, Z. Sura, N. Watkinson, S. Maleki, D. Padua, A. Veidenbaum, A. Nicolau, and J. Torrellas. An empirical study of the effect of source-level loop transformations on compiler stability. *Proc. ACM Program. Lang.*, 2(OOPSLA):126:1–126:29, Oct. 2018.

[55] N. Grech, K. Georgiou, J. Pallister, S. Kerrison, J. Morse, and K. Eder. Static Analysis of Energy Consumption for LLVM IR Programs. In *Proceedings of the 18th International Workshop on Software and Compilers for Embedded Systems*, SCOPES '15, page 12–21, New York, NY, USA, 2015. Association for Computing Machinery.

[56] T. Grosser, A. Groesslinger, and C. Lengauer. Polly - Performing Polyhedral Optimizations on a Low-level Intermediate Representation. *Parallel Processing Letters*, 22(04), 2012.

[57] A. Haj-Ali, N. K. Ahmed, T. Willke, Y. S. Shao, K. Asanovic, and I. Stoica. NeuroVectorizer: End-to-End Vectorization with Deep Reinforcement Learning. In *Proceedings of the 18th ACM/IEEE International Symposium on Code Generation and Optimization*, CGO 2020, pages 242–255, New York, NY, USA, 2020. Association for Computing Machinery.

[58] J. Haj-Yihia, A. Yasin, Y. B. Asher, and A. Mendelson. Fine-Grain Power Breakdown of Modern Out-of-Order Cores and Its Implications on Skylake-Based Systems. *ACM Trans. Archit. Code Optim.*, 13(4), Dec. 2016.

[59] M. Hall, J. Anderson, S. Amarasinghe, B. Murphy, S.-W. Liao, E. Bugnion, and M. Lam. Maximizing multiprocessor performance with the SUIF compiler. *Computer*, 29(12):84–89, 1996.

[60] M. W. Hall and K. Kennedy. Efficient Call Graph Analysis. *ACM Lett. Program. Lang. Syst.*, 1(3):227–242, Sept. 1992.

[61] H. He and E. A. Garcia. Learning from imbalanced data. *IEEE Transactions on Knowledge and Data Engineering*, 21(9):1263–1284, 2009.

[62] T. K. Ho. Random decision forests. In *Document analysis and recognition, 1995., proceedings of the third international conference on*, volume 1, pages 278–282. IEEE, 1995.

[63] H. Hoffmann, S. Sidiroglou, M. Carbin, S. Misailovic, A. Agarwal, and M. Rinard. Dynamic Knobs for Responsive Power-Aware Computing. In *Proceedings of the Sixteenth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS XVI, page 199–212, New York, NY, USA, 2011. Association for Computing Machinery.

[64] C.-H. Hsu, U. Kremer, and M. Hsiao. Compiler-directed dynamic voltage/frequency scheduling for energy reduction in microprocessors. In *ISLPED'01: Proceedings of the 2001 International Symposium on Low Power Electronics and Design (IEEE Cat. No.01TH8581)*, pages 275–278, 2001.

[65] Intel® Xeon® Processor Scalable Family: Specification Update, October 2020.

[66] Intel® C++ Compiler Classic Developer Guide and Reference.

[67] Intel® oneAPI Math Kernel Library.

[68] Automatic Parallelization with Intel®Compilers, August 2018.

[69] Intel® 64 and IA-32 Architectures Software Developer's Manual Volume 3 (3A, 3B, 3C & 3D): System Programming Guide, January 2019.

[70] A. Jäger, J.-P. Lehr, and C. Bischof. The influence of two modern compiler infrastructures on the energy consumption of the hpcg benchmark. *Computer Science - Research and Development*, May 2018.

[71] W. Jalby, D. Kuck, A. D. Malony, M. Masella, A. Mazouz, and M. Popov. The long and winding road toward efficient high-performance computing. *Proceedings of the IEEE*, 106(11):1985–2003, Nov 2018.

[72] N. Japkowicz and S. Stephen. The class imbalance problem: A systematic study. *Intelligent data analysis*, 6(5):429–449, 2002.

[73] I. Kadayif, M. Kandemir, and M. Karakoy. An energy saving strategy based on adaptive loop parallelization. In *Proceedings of the 39th Annual Design Automation Conference*, DAC '02, pages 195–200, New York, NY, USA, 2002. ACM.

[74] M. Kandemir, A. Choudhary, J. Ramanujam, and M. Kandaswamy. A unified framework for optimizing locality, parallelism, and communication in out-of-core computations. *IEEE Transactions on Parallel and Distributed Systems*, 11(7):648–668, 2000.

[75] M. Kandemir, N. Vijaykrishnan, and M. J. I. and. Influence of compiler optimizations on system power. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, 9(6):801–804, Dec 2001.

[76] M. Kandemir, N. Vijaykrishnan, and M. J. Irwin. *Compiler Optimizations for Low Power Systems*, pages 191–210. Springer US, Boston, MA, 2002.

[77] K. Kennedy and J. R. Allen. *Optimizing Compilers for Modern Architectures: A Dependence-based Approach*. Morgan Kaufmann Publishers Inc., 2002.

[78] R. Keryell, R. K. (presenting, C. Ancourt, B. Creusillet, F. Coelho, P. Jouvelot, and F. Irigoin. PIPS: a Workbench for Building Interprocedural Parallelizers, Compilers and Optimizers. Technical report, 1996.

[79] F. Kjolstad, S. Kamil, S. Chou, D. Lugato, and S. Amarasinghe. The tensor algebra compiler. *Proc. ACM Program. Lang.*, 1(OOPSLA):77:1–77:29, Oct. 2017.

[80] U. Kremer. Optimal and near – optimal solutions for hard compilation problems. *Parallel Processing Letters*, 7(04):371–378, 1997.

[81] M. Kruse and H. Finkel. User-directed loop-transformations in clang. In *2018 IEEE/ACM 5th Workshop on the LLVM Compiler Infrastructure in HPC (LLVM-HPC)*, pages 49–58, 2018.

[82] D. J. Kuck, R. H. Kuhn, D. A. Padua, B. Leasure, and M. Wolfe. Dependence graphs and compiler optimizations. In *Proceedings of the 8th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '81, page 207–218, New York, NY, USA, 1981. Association for Computing Machinery.

[83] S. Larsen and S. Amarasinghe. Exploiting superword level parallelism with multimedia instruction sets. In *Proceedings of the ACM SIGPLAN 2000 Conference on Programming Language Design and Implementation*, PLDI '00, page 145–156, New York, NY, USA, 2000. Association for Computing Machinery.

[84] C. Lattner and V. S. Adve. LLVM: A compilation framework for lifelong program analysis & transformation. In *CGO '04, 20-24 March 2004, San Jose, CA, USA*, pages 75–88, 2004.

[85] C. Lattner, M. Amini, U. Bondhugula, A. Cohen, A. Davis, J. Pienaar, R. Riddle, T. Shpeisman, N. Vasilache, and O. Zinenko. Mlir: Scaling compiler infrastructure for domain specific computation. In *2021 IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*, pages 2–14, 2021.

[86] H. Laurent and R. L. Rivest. Constructing optimal binary decision trees is np-complete. *Information processing letters*, 5(1):15–17, 1976.

[87] H. Leather, E. Bonilla, and M. O'Boyle. Automatic feature generation for machine learning–based optimising compilation. *ACM Trans. Archit. Code Optim.*, 11(1), Feb. 2014.

[88] S.-I. Lee, T. A. Johnson, and R. Eigenmann. Cetus – an extensible compiler infrastructure for source-to-source transformation. In L. Rauchwerger, editor, *Languages and Compilers for Parallel Computing*, pages 539–553, Berlin, Heidelberg, 2004. Springer Berlin Heidelberg.

[89] E. A. León, I. Karlin, R. E. Grant, and M. Dosanjh. Program optimizations: The interplay between power, performance, and energy. *Parallel Computing*, 58:56 – 75, 2016.

[90] W. Li and K. Pingali. A singular loop transformation framework based on non-singular matrices. In *International Workshop on Languages and Compilers for Parallel Computing*, pages 391–405. Springer, 1992.

[91] A. W. Lim, G. I. Cheong, and M. S. Lam. An Affine Partitioning Algorithm to Maximize Parallelism and Minimize Communication. In *Proceedings of the 13th International Conference on Supercomputing*, ICS '99, pages 228–237, New York, NY, USA, 1999. ACM.

[92] A. W. Lim and M. S. Lam. Maximizing Parallelism and Minimizing Synchronization with Affine Partitions. *Parallel Comput.*, 24(3-4):445–475, May 1998.

[93] A. W. Lim, S.-W. Liao, and M. S. Lam. Blocking and Array Contraction Across Arbitrarily Nested Loops Using Affine Partitioning. In *Proceedings of the Eighth ACM SIGPLAN Symposium on Principles and Practices of Parallel Programming*, PPoPP '01, pages 103–112, New York, NY, USA, 2001. ACM.

[94] S. Maleki, Y. Gao, M. J. Garzar´n, T. Wong, and D. A. Padua. An evaluation of vectorizing compilers. In *2011 International Conference on Parallel Architectures and Compilation Techniques*, pages 372–382, Oct 2011.

[95] D. Maydan, S. Amarsinghe, and M. Lam. Data dependence and data-flow analysis of arrays. In *International Workshop on Languages and Compilers for Parallel Computing*, pages 434–448. Springer, 1992.

[96] A. Mazouz, D. C. Wong, D. Kuck, and W. Jalby. Power-constrained optimal quality for high performance servers. In *Proceedings of the 47th International Conference on Parallel Processing Companion*, ICPP '18, pages 38:1–38:10, New York, NY, USA, 2018. ACM.

[97] C. Mendis, A. Jain, P. Jain, and S. Amarasinghe. Revec: Program rejuvenation through revectorization. In *Proceedings of the 28th International Conference on Compiler Construction (CC)*, CC 2019, pages 29–41, New York, NY, USA, 2019. ACM.

[98] C. Mendis, A. Renda, S. Amarasinghe, and M. Carbin. Ithemal: Accurate, portable and fast basic block throughput estimation using deep neural networks. In K. Chaudhuri and R. Salakhutdinov, editors, *Proceedings of the 36th International Conference on Machine Learning (ICML)*, volume 97 of *Proceedings of Machine Learning Research*, pages 4505–4515, Long Beach, California, USA, Jun 2019. PMLR.

[99] C. Mendis, C. Yang, Y. Pu, S. Amarasinghe, and M. Carbin. Compiler auto-vectorization with imitation learning. In H. Wallach, H. Larochelle, A. Beygelzimer, F. d'Alché-Buc, E. Fox, and R. Garnett, editors, *Advances in Neural Information Processing Systems 32 (NeurIPS)*, pages 14598–14609. Curran Associates, Inc., Dec 2019.

[100] S. P. Midkiff. Automatic parallelization: an overview of fundamental compiler techniques. *Synthesis Lectures on Computer Architecture*, 7(1):1–169, 2012.

[101] A. Monsifrot, F. Bodin, and R. Quiniou. A machine learning approach to automatic production of compiler heuristics. In *Proceedings of the 10th International Conference on Artificial Intelligence: Methodology, Systems, and Applications*, AIMSA '02, pages 41–50, London, UK, UK, 2002. Springer-Verlag.

[102] T. Moseley, D. Grunwald, and R. Peri. Optiscope: Performance accountability for optimizing compilers. In *Proceedings of the 7th Annual IEEE/ACM International Symposium on Code Generation and Optimization*, CGO '09, pages 254–264, Washington, DC, USA, 2009. IEEE Computer Society.

[103] OpenCV (Open Source Computer Vision Library), Version 4.0.0. `https://opencv.org`, November 2018.

[104] The OpenMP Application Programming Interface, Version 5.0. `https://www.openmp.org/`, November 2018.

[105] Padua, Kuck, and Lawrie. High-speed multiprocessors and compilation techniques. *IEEE Transactions on Computers*, C-29(9):763–776, Sept 1980.

[106] D. A. Padua and M. Wolfe. Advanced Compiler Optimizations for Supercomputers. *Commun. ACM*, 29(12):1184–1201, 1986.

[107] J. Pallister, S. J. Hollis, and J. Bennett. Identifying compiler options to minimize energy consumption for embedded platforms. *The Computer Journal*, 58(1):95–109, 2015.

[108] M. Panchenko, R. Auler, B. Nell, and G. Ottoni. BOLT: A Practical Binary Optimizer for Data Centers and Beyond. In *Proceedings of the 2019 IEEE/ACM International Symposium on Code Generation and Optimization*, CGO 2019, page 2–14. IEEE Press, 2019.

[109] P. R. Panda, F. Catthoor, N. D. Dutt, K. Danckaert, E. Brockmeyer, C. Kulkarni, A. Vandercappelle, and P. G. Kjeldsberg. Data and memory optimization techniques for embedded systems. *ACM Trans. Des. Autom. Electron. Syst.*, 6(2):149–206, Apr. 2001.

[110] E. Park, L. Pouchet, J. Cavazos, A. Cohen, and P. Sadayappan. Predictive modeling in a polyhedral optimization space. In *International Symposium on Code Generation and Optimization (CGO 2011)*, pages 119–129, 2011.

[111] PLUTO: An automatic parallelizer and locality optimizer for affine loop nests, 2015.

[112] PoCC: The Polyhedral Compiler Collection.

[113] Polly: LLVM Framework for High-Level Loop and Data-Locality Optimizations.

[114] PolyBench/C 4.1. `http://web.cse.ohio-state.edu/~pouchet.2/software/polybench/`, 2015.

[115] L.-N. Pouchet, C. Bastoul, A. Cohen, and J. Cavazos. Iterative optimization in the polyhedral model: Part ii, multidimensional time. In *Proceedings of the 29th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '08, page 90–100, New York, NY, USA, 2008. Association for Computing Machinery.

[116] L.-N. Pouchet, U. Bondhugula, C. Bastoul, A. Cohen, J. Ramanujam, and P. Sadayappan. Combined iterative and model-driven optimization in an automatic parallelization framework. In *Proceedings of the 2010 ACM/IEEE International Conference for High Performance Computing, Networking, Storage and Analysis*, SC '10, page 1–11, USA, 2010. IEEE Computer Society.

[117] L.-N. Pouchet, U. Bondhugula, C. Bastoul, A. Cohen, J. Ramanujam, P. Sadayappan, and N. Vasilache. Loop Transformations: Convexity, Pruning and Optimization. In *Proceedings of the 38th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '11, page 549–562, New York, NY, USA, 2011. Association for Computing Machinery.

[118] W. Pugh and D. Wonnacott. Eliminating False Data Dependences Using the Omega Test. In *Proceedings of the ACM SIGPLAN 1992 Conference on Programming Language Design and Implementation*, PLDI '92, pages 140–151, New York, NY, USA, 1992. ACM.

[119] D. Quinlan. Rose: Compiler support for object-oriented frameworks. *Parallel Processing Letters*, 10:215–226, 2000.

[120] S. F. Rahman, J. Guo, and Q. Yi. Automated empirical tuning of scientific codes for performance and power consumption. In *Proceedings of the 6th International Conference on High Performance and Embedded Architectures and Compilers*, pages 107–116. ACM, 2011.

[121] S. Ramesh, S. Perarnau, S. Bhalachandra, A. D. Malony, and P. Beckman. Understanding the Impact of Dynamic Power Capping on Application Progress. In *2019 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, pages 793–804, 2019.

[122] H. Saputra, M. Kandemir, N. Vijaykrishnan, M. J. Irwin, J. S. Hu, C.-H. Hsu, and U. Kremer. Energy-conscious Compilation Based on Voltage Scaling. In *Proceedings of the Joint Conference on Languages, Compilers and Tools for Embedded Systems: Software and Compilers for Embedded Systems*, LCTES/SCOPES '02, pages 2–11, New York, NY, USA, 2002. ACM.

[123] R. E. Schapire. A Brief Introduction to Boosting. In *Proceedings of the 16th International Joint Conference on Artificial Intelligence - Volume 2*, IJCAI'99, page 1401–1406, San Francisco, CA, USA, 1999. Morgan Kaufmann Publishers Inc.

[124] T. B. Schardl, W. S. Moses, and C. E. Leiserson. Tapir: Embedding fork-join parallelism into llvm's intermediate representation. In *Proceedings of the 22nd ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPoPP '17, page 249–265, New York, NY, USA, 2017. Association for Computing Machinery.

[125] R. Schöne, T. Ilsche, M. Bielert, A. Gocht, and D. Hackenberg. Energy Efficiency Features of the Intel Skylake-SP Processor and Their Impact on Performance. In *2019 International Conference on High Performance Computing Simulation (HPCS)*, pages 399–406, 2019.

[126] A. Shivam, N. Watkinson, A. Nicolau, D. Padua, and A. V. Veidenbaum. Towards an achievable performance for the loop nests. In M. Hall and H. Sundar, editors, *Languages and Compilers for Parallel Computing*, pages 70–77, Cham, 2019. Springer International Publishing.

[127] S. Sridharan, G. Gupta, and G. S. Sohi. Holistic Run-Time Parallelism Management for Time and Energy Efficiency. In *Proceedings of the 27th International ACM Conference on International Conference on Supercomputing*, ICS '13, page 337–348, New York, NY, USA, 2013. Association for Computing Machinery.

[128] M. Stephenson and S. Amarasinghe. Predicting unroll factors using supervised classification. In *Proceedings of the International Symposium on Code Generation and Optimization*, CGO '05, pages 123–134, Washington, DC, USA, 2005. IEEE Computer Society.

[129] M. Stephenson, S. Amarasinghe, M. Martin, and U.-M. O'Reilly. Meta optimization: Improving compiler heuristics with machine learning. In *Proceedings of the ACM SIGPLAN 2003 Conference on Programming Language Design and Implementation*, PLDI '03, pages 77–90, New York, NY, USA, 2003. Association for Computing Machinery.

[130] K. Stock, L.-N. Pouchet, and P. Sadayappan. Using machine learning to improve automatic vectorization. *ACM Transactions on Architecture and Code Optimization (TACO)*, 8(4):50, 2012.

[131] J. A. Stratton, C. Rodrigues, I.-J. Sung, N. Obeid, L.-W. Chang, N. Anssari, G. D. Liu, and W.-m. W. Hwu. Parboil: A revised benchmark suite for scientific and commercial throughput computing. *Center for Reliable and High-Performance Computing*, 127, 2012.

[132] S. F. X. Thiago Teixeira, C. Ancourt, D. Padua, and W. Gropp. Locus: A system and a language for program optimization. In *2019 IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*, pages 217–228, 2019.

[133] A. Tiwari, C. Chen, J. Chame, M. Hall, and J. K. Hollingsworth. A scalable autotuning framework for compiler optimization. In *2009 IEEE International Symposium on Parallel Distributed Processing*, pages 1–12, 2009.

[134] G. Tournavitis, Z. Wang, B. Franke, and M. F. O'Boyle. Towards a holistic approach to auto-parallelization: Integrating profile-driven parallelism detection and machine-learning based mapping. In *Proceedings of the 30th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '09, pages 177–187, New York, NY, USA, 2009. ACM.

[135] J. Treibig, G. Hager, and G. Wellein. LIKWID: A Lightweight Performance-Oriented Tool Suite for x86 Multicore Environments. In *2010 39th International Conference on Parallel Processing Workshops*, pages 207–216, Sep. 2010.

[136] S. Triantafyllis, M. Vachharajani, N. Vachharajani, and D. I. August. Compiler Optimization-Space Exploration. In *Proceedings of the International Symposium on Code Generation and Optimization: Feedback-Directed and Runtime Optimization*, CGO '03, page 204–215, USA, 2003. IEEE Computer Society.

[137] M. Valluri and L. K. John. *Is Compiling for Performance == Compiling for Power?*, pages 101–115. Springer US, Boston, MA, 2001.

[138] W. Wang, J. Cavazos, and A. Porterfield. Energy auto-tuning using the polyhedral approach. In S. Rajopadhye and S. Verdoolaege, editors, *Proceedings of the 4th International Workshop on Polyhedral Compilation Techniques*, Vienna, Austria, Jan 2014.

[139] Z. Wang and M. F. O'Boyle. Mapping parallelism to multi-cores: A machine learning based approach. In *Proceedings of the 14th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPoPP '09, pages 75–84, New York, NY, USA, 2009. ACM.

[140] N. Watkinson, A. Shivam, Z. Chen, A. Veidenbaum, A. Nicolau, and Z. Gong. Using hardware counters to predict vectorization. In L. Rauchwerger, editor, *Languages and Compilers for Parallel Computing*, pages 3–16, Cham, 2019. Springer International Publishing.

[141] G. M. Weiss and F. Provost. The effect of class distribution on classifier learning: an empirical study. 2001.

[142] G. M. Weiss and F. Provost. Learning when training data are costly: The effect of class distribution on tree induction. *Journal of artificial intelligence research*, 19:315–354, 2003.

[143] M. E. Wolf, D. E. Maydan, and D.-K. Chen. Combining loop transformations considering caches and scheduling. In *Proceedings of the 29th Annual ACM/IEEE International Symposium on Microarchitecture*, MICRO 29, page 274–286, USA, 1996. IEEE Computer Society.

[144] M. Wolfe. Iteration Space Tiling for Memory Hierarchies. In *Proceedings of the Third SIAM Conference on Parallel Processing for Scientific Computing*, pages 357–361, Philadelphia, PA, USA, 1989. SIAM.

[145] M. Wolfe. More Iteration Space Tiling. In *SC '89*, pages 655–664, New York, NY, USA, 1989. ACM.

[146] M. J. Wolfe. *High Performance Compilers for Parallel Computing*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1995.

[147] R. Xu, X. Tian, S. Chandrasekaran, Y. Yan, and B. Chapman. NAS Parallel Benchmarks for GPGPUs Using a Directive-Based Programming Model. In J. Brodman and P. Tu, editors, *Languages and Compilers for Parallel Computing*, pages 67–81, Cham, 2015. Springer International Publishing.

[148] J. Xue. *Loop Tiling for Parallelism*. Kluwer Academic Publishers, Norwell, MA, USA, 2000.

[149] H. Yang, G. R. Gao, A. Marquez, G. Cai, and Z. Hu. Power and energy impact by loop transformations. In *In Proceedings of the Workshop on Compilers and Operating Systems for Low Power 2001, Parallel Architecture and Compilation Techniques*, 2000.

[150] T. Yuki, G. Gupta, D. Kim, T. Pathan, and S. Rajopadhye. Alphaz: A system for design space exploration in the polyhedral model. In *International Workshop on Languages and Compilers for Parallel Computing*, pages 17–31. Springer, 2012.

[151] J. Zambreno, M. T. Kandemir, and A. Choudhary. Enhancing compiler techniques for memory energy optimizations. In A. Sangiovanni-Vincentelli and J. Sifakis, editors, *Embedded Software*, pages 364–381, Berlin, Heidelberg, 2002. Springer Berlin Heidelberg.

[152] Y. Zhu, G. Magklis, M. L. Scott, C. Ding, and D. H. Albonesi. The energy impact of aggressive loop fusion. In *Proceedings of the 13th International Conference on Parallel Architectures and Compilation Techniques*, PACT '04, pages 153–164, Washington, DC, USA, 2004. IEEE Computer Society.